UNIVERSIDADE FEDERAL DE PERNAMBUCO

CENTRO DE INFORMÁTICA

TRABALHO DE GRADUAÇÃO EM SEGURANÇA DA INFORMAÇÃO

Sentinel: um engenho Java para controle de acesso RBAC

Recife, Agosto de 2003

Autor: Cristiano Lincoln de Almeida Mattos – lincoln@tempest.com.br
Orientador: André Medeiros Santos – alms@cin.ufpe.br

SUMÁRIO

Este relatório apresenta os resultados obtidos durante o trabalho de graduação do autor, com o objetivo de modelar e implementar um engenho de controle de acesso RBAC, denominado Sentinel.

O Sentinel tem como objetivos ser flexível e genérico o suficiente para que possa ser integrado a diferentes estilos de aplicações sem muito esforço. Ao oferecer uma arquitetura baseada em *plugins,* prima pela extensibilidade, permitindo que diferentes aplicações expressem suas necessidades particulares de controle de acesso implementando pequenos componentes que se integram a um framework maior.

Dentre os vários modelos de controle de acesso que podem ser utilizados em um engenho, foi escolhido o RBAC – *role-based access control*. Este modelo vem ganhando considerável força no mundo acadêmico e comercial por ser expressivo e poderoso, tanto em critérios de proteção quanto em flexibilidade.

AGRADECIMENTOS

Gostaria de agradecer a todos que contribuíram para a execução deste trabalho. Agradeço à minha esposa pelo amor e apoio, meus pais pela minha formação e ao meu orientador, bem como os amigos da Tempest Security Technologies e CESAR.

ÍNDICE

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE TRECHOS DE CÓDIGO-FONTE

1 Introdução

Acesso é a possibilidade de realizar alguma operação (usar, modificar, visualizar, etc.) com algum recurso computacional (arquivo, *device*, etc.). Controle de acesso é o meio pelo qual a capacidade de realizar a operação é explicitamente habilitada ou restringida de alguma forma. Estes controles podem estar implementados em vários níveis, desde o físico até um controle interno a um sistema computacional, como uma aplicação ou o próprio sistema operacional.

Existem uma série de modelos de controle de acesso em um sistema computacional, em que os mais conhecidos são MAC (Mandatory Access Control) e DAC (Discretionary Access Control). O primeiro nasceu para proteger sistemas militares, enquanto o segundo sempre foi indicado como a solução para sistemas civis e comerciais.

Um modelo em particular que vem ganhando cada vez mais aceitação científica e prática é o RBAC (Role-Based Access Control), principalmente em substituição ao DAC, oferecendo significativas vantagens na definição e administração do controle de acesso a recursos computacionais.

Enquanto tanto o MAC quanto DAC costumam controlar apenas acesso a informações, RBAC trata o acesso tanto de funções quanto de informações. Além disso, introduz o conceito de *role* (perfil, ou cargo), que pode ser definido como um conjunto de ações e responsabilidades associados com um cargo em particular. Usuários e operações são associados aos *roles* (ao invés de associados diretamente entre si), tornando a manutenção de um sistema RBAC mais consistente e organizada, já que as relações entre permissões e cargos costumam ser mais persistentes que entre permissões e usuários específicos.

A especificação RBAC introduz também o conceito de hierarquia de cargos, com herança de permissões. Além disso, RBAC traz o importante conceito de *constraints* (restrições), predicados que podem operar sobre a autorização de um acesso, negando-o ou permitindo-o com base em critérios mais complexos do que o envolvido em operações simples. Por exemplo, um *constraint* comum é o de separação de privilégios, em que um mesmo usuário não pode pertencer a dois *roles* que poderiam dar um excesso de poder ao mesmo: por exemplo, um mesmo usuário participando dos *roles* de "gerente de contas" e "auditor" daria ao mesmo um excesso de poder que poderia ajudá-lo a praticar ou mascarar fraudes. *Constraints* mais complexos e específicos de determinadas aplicações podem vir a ser definidos.

Apesar de contar com uma base teórica e acadêmica forte, existem poucas implementações disponíveis do modelo RBAC, ao menos fora de ambientes fechados. O objetivo deste trabalho é modelar e implementar um *engine* de autorização RBAC em Java, de forma extensível, flexível e segura, chamado Sentinel.

Primeiramente o leitor será contextualizado com uma visão geral de controle de acesso: do que se trata, importância, conceitos básicos e principais modelos acadêmicos para descrição do controle de acesso.

Em seguida, uma seção tratará especificamente dos conceitos do modelo de controle de acesso provido pelo Sentinel, o RBAC, sendo descritos suas motivações e características, culminando com a exposição do padrão proposto pelo NIST.

A partir daí, as próximas duas seções tratam especificamente do Sentinel, baseando-se em todo o corpo de conhecimento apresentado anteriormente. Descreve-se a arquitetura do Sentinel e o seu relacionamento com os modelos apresentados. São explicados vários aspectos de implementação do engenho, as suas vantagens e desvantagens e as justificativas para sua escolha.

Ao longo do texto, figuras, tabelas, diagramas UML e trechos de código-fonte ilustram aspectos importantes. A conclusão finaliza o trabalho, ressaltando pontos que demandam mais trabalho, e caminhos interessantes de pesquisa.

2 Visão geral de controle de acesso

Uma definição do conceito de acesso, para nossos fins, é a possibilidade de realizar alguma operação sobre algum recurso computacional. Tanto as operações quanto os recursos variam de acordo com o contexto em que o acesso está sendo realizado – por exemplo, a operação poderia ser uma leitura, escrita, execução, remoção ou outras mais complexas; enquanto o recurso computacional poderia estar caracterizado como um arquivo, uma área de memória, um device ou um socket.

Em conjunto com as operações de acesso, a necessidade do controle deste acesso é um fator que vem desde os primeiros sistemas operacionais [EA94] [SCYF96]. De fato, muitos dos trabalhos seminais no campo de controle de acesso computacional vem das décadas de setenta e oitenta. Esta seção fornecerá uma visão geral dos conceitos básicos de controle de acesso, e posteriormente os modelos mais populares de controle de acesso, apresentados em detalhes suficientes a contextualizar o leitor para a discussão mais profunda e comparação com RBAC, a ser realizada na próxima seção.

2.1 Conceitos básicos de controle de acesso

Esta seção define alguns conceitos básicos sobre controle de acesso, visando substanciar o leitor no assunto.

2.1.1 Autenticação, autorização e auditoria

O termo controle de acesso pode ser definido em cima da divisão em três subcampos distintos, que agem em conjunto: autenticação, autorização e auditoria, conhecidos como os "três As".

Autenticação trata de identificar o usuário a acessar o sistema. Autorização trata do que esse usuário poderá realizar (está autorizado a fazer) no sistema. Auditoria mantém os registros necessários das ações do usuário.

O leitor deve estar atento ao fato de que na maior parte da literatura, o termo controle de acesso é utilizado como sinônimo para autorização. De fato, o assunto principal deste artigo é o modelo de controle de acesso RBAC, que poderia ser mais precisamente descrito como modelo de autorização RBAC. A seguir cada um dos três subcampos são explicitados.

Autenticação

Autenticação refere-se ao processo de fornecer ao sistema informações que identifiquem com um grau de certeza suficiente (para os fins do sistema em questão) o usuário que está

requisitando o acesso aos recursos computacionais. Em termos gerais, é o processo de provar ao sistema que o usuário é realmente quem diz ser, e não alguém se passando por ele. O processo de autenticação pode ser realizado ao sistema de três maneiras principais:

- **Autenticação utilizando algo que você sabe**: é o método mais comum, em que o usuário autentica-se ao sistema fornecendo-lhe alguma informação que (assume-se) só é de conhecimento do usuário: uma senha de acesso, por exemplo. É o mecanismo mais comum de autenticação, e por isso mesmo o mais fácil e mais sujeito a abusos. A principal vulnerabilidade deste tipo de autenticação é que se outra pessoa descobrir ou adivinhar o segredo de acesso, poderá facilmente utilizar o sistema passando-se pelo usuário legítimo. Por isso, o segredo deve ser protegido – de preferência ficando apenas armazenado na memória do usuário (ao invés de anotado em algum lugar). Isso levanta o fino balanço entre senhas fáceis de serem decoradas e adivinhadas, ou senhas difíceis de serem adivinhadas – mas também difíceis de serem decoradas, fomentando que sejam registradas fora da memória do usuário.

- **Autenticação utilizando algo que você possui**: é o segundo método mais comum, em que o usuário autentica-se através da apresentação de algum *token* ou objeto que esteja em sua possessão – uma chave para uma fechadura, um arquivo no seu sistema ou um cartão de crédito são exemplos desse tipo de autenticação, que recentemente vem ganhando força na forma de *smartcards* [AJSW]. A principal vulnerabilidade desse tipo de autenticação é parecida com a anterior, ainda que mais difícil de acontecer: se o usuário perder o objeto de autenticação, ou for roubado, será possível autenticar-se no sistema. Em geral, quanto mais móvel (leve, prático) for o objeto de autenticação, mais fácil será de ser extraviado.

- **Autenticação utilizando algo que você é**: esse é o tipo menos comum de autenticação, baseado na apresentação em alguma propriedade fundamental e integrante de uma pessoa para a autenticação. A biometria é extensivamente utilizada neste caso, para autenticação utilizando características únicas de indivíduos, como padrões de íris, impressão digital, impressão baseada na mão, padrões de voz ou de escrita (assinaturas são considerados como biometria também). A autenticação por técnicas biométricas possui uma série de variações e detalhes específicos, mas em geral funcionam da seguinte maneira:

    - Primeiro, as informações biométricas são capturadas do usuário do sistema – por exemplo, é realizada uma análise da retina do indivíduo para autenticação por padrões da íris.

    - Segundo, os dados biométricas são extraídos da colheita realizada – em geral, são utilizadas técnicas de sumários digitais para resumir a massa de dados em seqüências de tamanho físico. Uma característica destes sumários é que eles são função exclusiva dos dados de entrada, e não é possível recuperar os dados de entrada a partir do sumário final.

    - Terceiro, o sumário biométrico calculado é comparado com um sumário já armazenado no sistema para aquele usuário. Se forem iguais, a autenticação é realizada com sucesso.

Apesar da atenção em cima de tecnologias biométricas de autenticação, elas tem uma vulnerabilidade básica: biométricos são identificadores únicos, mas não são segredos. Ou seja, é perfeitamente possível um atacante conseguir retirar um sumário da impressão digital de um usuário de forma ilícita, e utilizá-lo para se autenticar ao sistema. Pior, ao contrário dos dois outros tipos de autenticação, não é possível revogar uma informação biométrica. No exemplo dado acima, isso equivaleria a cortar o dedo do usuário, uma experiência nada agradável.

Como se pode ver, utilizar um único tipo de autenticação costuma ser insuficiente, já que os três tipos tem vulnerabilidades individuais. Para controle de acesso, o mais efetivo é utilizar pelo menos duas das três técnicas de autenticação em conjunto. Por exemplo, aliar a identificação biométrica ou a possessão de um cartão de acesso ao fornecimento de uma senha individual.

A autenticação é o primeiro passo para o controle de acesso. Depois de identificar unicamente o indivíduo que estará utilizando o sistema, é necessário avaliar a extensão da autorização do indivíduo naquele sistema.

Autorização

O processo de autorização rege exatamente que operações, sob que recursos computacionais, o usuário poderá executar no sistema. Para que sejam efetivadas quaisquer avaliações de autorização é necessário ter passado pela etapa de autenticação.

No processo de autorização é que vêm à tona toda a complexidade e riqueza dos modelos de controle de acesso, e das políticas que permitem implementar. Como este é o foco deste artigo, os principais modelos de autorização serão definidos e detalhados adiante, em seção própria.

Auditoria

Um aspecto muitas vezes ignorado de sistemas de controle de acesso é o aspecto de auditoria: manter registro das principais transações executadas pelo sistema. A idéia é prover uma "trilha" que permita reconstruir operações relevantes do usuário no sistema.

A definição de "relevante" é dependente das particularidades de cada sistema. Por exemplo, pode-se registrar os horários em que o usuário entrou e saiu do sistema ("logon" e "logoff"), ou os detalhes de cada operação realizada pelo usuário durante a sua sessão de trabalho.

2.1.2 Sujeito, objeto, operações e permissões

Define-se sujeito (*subject)* como sendo a representação do usuário dentro do sistema. Pode-se entender o conceito de sujeito como o de um processo no sistema aliado a um conjunto de credenciais de acesso que associam aquele processo a um usuário da base do sistema. Em geral, as regras de autorização são expressas utilizando-se o identificador do usuário (seu *login* ou identificação única no sistema). No momento da autenticação, as credenciais de acesso são geradas para aquela sessão do usuário. A partir daí, qualquer execução de processo ou rotina dentro do sistema é encarada como um *subject*, já que alia as credenciais de acesso do usuário com um processo que o representa no sistema.

Neste trabalho será seguido o exemplo da literatura, em que se utiliza os termos sujeito e usuário de forma intercambiável, ficando claro a distinção técnica entre os dois.

O objeto (*object*) representa o recurso computacional cujo acesso é controlado. Ele pode ser, na prática, qualquer estrutura de dados ou abstração fornecida pelo sistema, incluindo: arquivos, área de memória, um dispositivo externo (mouse, ou monitor), um *socket*, uma conta bancária, um processo, e assim por diante.

Operações são realizadas pelos sujeitos do sistema sob seus objetos. As operações podem ter vários níveis de abstração – incluindo operações comuns como leitura, escrita, remoção até operações mais complexas e dependentes da natureza de cada sistema, como operações de débito e crédito em um sistema bancário, por exemplo.

Uma permissão é a manifestação de que uma determinada operação é permitida para um determinado objeto; por exemplo: "permissão de leitura do arquivo da fila de impressão".

2.1.3 Matrizes de acesso, listas de controle de acesso e *capabilities*

Uma técnica simples para modelar regras de autorização de usuários a objetos é utilizando matrizes de acesso (*access matrices*). A partir de uma matriz de acesso desdobram-se naturalmente listas de controle de acesso e *capabilities*, técnicas utilizadas na maior parte dos sistemas operacionais modernos [RA01].

A matriz de acesso é organizada com colunas representando os objetos, e os usuários em linhas. Em cada célula da tabela representam-se as permissões que o respectivo usuário possui sobre o objeto. No exemplo da Tabela 1, segue-se a convenção do UNIX, em que a letra "R" representa permissão de leitura, a letra "W" representa permissão de escrita e a letra "X" representa permissão de execução.

	Arquivo de impressão	Arquivos do SO	Arquivos públicos	Planilhas financeiras
João	R-W-X	R-W-X	R-W	R
José	X	X	R-W	-
Maria	R-X	R	R	R

Tabela 1: exemplo de matriz de acesso

Matrizes de acesso podem ser utilizadas para modelar mecanismos de autorização simples. Mas não são recomendados para implementação, já que não escalam bem: um sistema com 50.000 usuários e 300 arquivos precisaria de uma matriz com 15 milhões de células, gerando problemas de espaço e maior possibilidade de erros na administração. Duas abordagens amplamente utilizadas para resolver essa questão são listas de controle de acesso (ACL – access control lists) e capabilities.

Listas de controle de acesso

Uma maneira de simplificar o gerenciamento das permissões de acesso é indexar a matriz de controle de acesso pela coluna, indicando que usuários possuem que permissão para cada objeto. Esta lista de controle de acesso é armazenada junto com cada objeto e relaciona

quais permissões cada usuário possui naquele objeto. A Tabela 2 mostra como ficaria a lista de controle de acesso para as planilhas financeiras do nosso exemplo anterior.

	Planilhas financeiras
João	R
José	-
Maria	R

Tabela 2: Lista de controle de acesso para o objeto planilhas financeiras

Listas de controle de acesso (ACLs) são extensamente utilizadas em praticamente todos os sistemas operacionais modernos (UNIX, Windows), além de estarem presentes em dispositivos de rede como roteadores e *firewalls*.

Capabilities

A outra maneira de gerenciar a matriz de acesso é indexando-a pelos usuários, apontando para cada um suas permissões no sistema, sendo conhecida como *capabilities*. A Tabela 3 mostra o exemplo anterior modificado para dar essa visão.

	Arquivo de impressão	Arquivos do SO	Arquivos públicos	Planilhas financeiras
João	R-W-X	R-W-X	R-W	R

Tabela 3: *capabilities* do usuário João

Comparando ambas as abordagens

ACLs são simples de implementar, mas difíceis de manter em ambientes de alta rotatividade de usuários ou arquivos, já que é necessário configurá-los para cada objeto. O agrupamento de usuários ajuda, mas não resolve a questão. Como as ACLs são indexadas pelo objeto a ser protegido, é mais custoso saber exatamente que usuários tem acesso a que arquivos do sistema.

As vantagens e desvantagens do uso de *capabilities* são basicamente o contrário das mesmas de listas de controle de acesso. A implementação de *capabilities* tende a ser um pouco mais eficiente em sistemas operacionais [RA01], enquanto é mais difícil saber que usuários tem acesso a um arquivo, já que a informação está espalhada.

Listas de controle de acesso tem gozado de uma popularidade muito maior no mundo comercial de sistemas operacionais que *capabilities*. Isso se deve ao fato de que o modelo DAC (Discretionary Access Control – explicado em seção própria) é o mais popular nos sistemas operacionais modernos, e ACLs adequam-se bem a este modelo, por facilitar que cada usuário gerencie as permissões de seus objetos no sistema.

2.2 Modelos de controle de acesso: DAC e MAC

Modelos de controle de acesso (ou, rigorosamente, modelos de autorização de acesso) definem características primitivas de um determinado conjunto de regras de autorização a

serem utilizadas. Essas características influenciam os limites da semântica de autorização que pode ser expressa no modelo e conseqüentemente a sua implementação. Os principais modelos de controle de acesso hoje são DAC (Discretionary Access Control), MAC (Mandatory Access Control) e RBAC (Role-Based Access Control).

Dentro de um determinado modelo de controle de acesso podem existir diferentes políticas de controle de acesso (ou, rigorosamente políticas de autorização de acesso) são uma declaração sucinta das propriedades de proteção quem um sistema ou uma classe genérica de sistemas precisa possuir. Seus pontos-chave em geral cabem em uma única página, e é o documento que expressa os objetivos da proteção e pode ser a base para uma análise matemática formal. Existem uma grande variedade de políticas de controle de acesso para sistemas aplicados em diferentes setores, nos quais se destacam políticas para sistemas militares [BL75] [BIBA77], financeiros [CW87], e de saúde [RA96].

Os conceitos acima, de modelo de controle de acesso e política de controle de acesso, em alguns casos são utilizado com outros sentidos na literatura. Dessa forma, as definições acima foram retiradas de [SCYF96], [SAND96] e [RA01], fontes autoritativas na área de controle de acesso.

Esta seção apresentará ao leitor os dois modelos de controle de acesso mais populares atualmente: DAC e MAC. Além de apresentar os modelos acima, serão caracterizadas algumas das principais políticas que foram desenvolvidas e são atualmente utilizadas dentro destes modelos.

2.2.1 DAC: Discretionary Access Control

Controle de acesso discrecionário tem sua origem no contexto de pesquisa acadêmica em sistemas de tempo compartilhado que surgiram no começo dos anos setenta. O artigo [LAM71] introduziu as idéias básicas. DAC é baseado na noção de que usuários individuais são "donos" de objetos e portanto tem controle (discreção) total em quem deve ter permissões para acessar o objeto. Um usuário transforma-se em dono do objeto ao criá-lo.

No modelo discrecionário, se João é dono de um objeto (um arquivo, por exemplo), ela pode conceder a José a permissão de acessá-lo em um modo qualquer de operação. Posteriormente, ela pode revogar essa permissão a qualquer momento. O princípio básico de DAC é possessão do objeto pelo usuário que o criou.

Atualmente, o DAC é o modelo mais popular de controle de acesso, pela sua utilização em grande escala em sistemas operacionais comerciais. Todas as variantes do UNIX, o Netware e a série Windows NT, 2000 e XP utilizam o modelo DAC como seu modelo básico de controle de acesso. Estes sistemas operacionais utilizam extensamente a técnica de listas de controle de acesso para conceber a implementar as suas checagens de autorização, dispondo também do conceito de grupos de usuários para facilitar na administração e concessão de permissões.

Ao contrário de MAC, não existem políticas de controle de acesso em DAC popularmente difundidas e em utilização, já que o modo de operação "padrão" do modelo é suficiente para as necessidades de uma boa parcela do mundo comercial. O leitor interessado pode pesquisar as políticas existentes em mais detalhes em [LUNT88] e [RBKW91].

O modelo DAC possui uma fraqueza inerente: o fato de que informação pode ser copiada de um objeto para outro, de modo que acesso a uma cópia é possível mesmo que o dono do objeto original não tenha provido acesso ao original [SAND96]. Usuários que possuem acesso ao objeto original podem inadvertidamente permitir a realização de cópias não-autorizadas, ao executar um programa "cavalo de tróia" que faça a cópia dos dados do objeto sem a explícita autorização ou cooperação do usuário.

Essa fraqueza do modelo DAC tornou-o insuficiente para sistemas militares, em que a informação precisava ter um alto nível de controle e ser resistente a ataque por cavalos de tróia, fomentando a criação do modelo MAC.

2.2.2 MAC: Mandatory Access Control

Enquanto o ponto-chave do DAC é o fato de que os usuários são considerados donos do objeto e portanto responsáveis pelas suas permissões de acesso, o modelo mandatório prevê que usuários individuais não tem escolha em relação a que permissões de acesso eles possuem ou a que objetos podem acessar [SCYF96] [SAND93].

Neste modelo, os usuários individuais não são considerados donos dos objetos, e não podem definir suas permissões – isso é realizado pelos administradores do sistema.

O modelo MAC é conhecido – a tal ponto de ser as vezes confundido – pela sua utilização em políticas de acesso multinível, em que se deseja controlar o fluxo de informações em um sistema. Em geral, se quer garantir que a informação só flua em um determinado sentido: por exemplo, de níveis mais baixos de confidencialidade para níveis maiores de confidencialidade – nunca de níveis mais altos para níveis mais baixos.

Tal área de pesquisa se desenvolveu fortemente nos anos setenta, com base nas necessidades de confidencialidade dos sistemas militares norte-americanos.

O fluxo de informações dentro destes sistemas deve seguir regras claras para garantir a sua confidencialidade. Esse fluxo, e as regras de acesso impostas sobre ele, costumam ser expressos utilizando *lattices*, de tal modo que MAC é em certos contextos conhecido como LBAC – Lattice-Based Access Control.

Para explicar como funciona uma política de acesso mandatória, será utilizada a política definida por Bell-Lapadula para sistemas militares norte-americanos, sendo uma referência no contexto de segurança computacional. Existem várias outras políticas que utilizam de recursos mandatórios – outro exemplo seria a política de acesso Biba, que ao contrário da Bell-Lapadula prima pela integridade das informações ao invés da confidencialidade. O leitor interessado pode consultar as referências deste trabalho para tratamentos mais detalhados destas políticas.

Bell-Lapadula

Essa política de acesso é decorrente das regras de segurança da informação praticadas pelos militares norte-americanos. Ela é expressada em termos de rótulos de segurança associados a sujeitos e objetos dentro do sistema. Um rótulo em um objeto é chamado de *security classification*, enquanto o rótulo em um sujeito do sistema é chamado de *security clearance*.

Entre os rótulos originalmente utilizados estão "TOP SECRET", "SECRET", "CONFIDENTIAL" e "UNCLASSIFIED", em ordem parcial com o nível mais alto de confidencialidade sendo o primeiro citado. Atualmente existem outras classificações, mas isso não faz diferença para os fins de explicação da política.

Expressa informalmente, a política é simples: um usuário no sistema (sujeito) só poderia ter acesso de leitura a um objeto se o seu *clearance* for maior ao igual à classificação do objeto. Assim, um oficial com acesso "TOP SECRET" poderia ler um documento "SECRET", mas o inverso não seria permitido. O efeito é que a leitura de informação só funciona em um sentido, de baixo para cima no sistema. Para evitar que um funcionário com uma classificação alta indevidamente "desclassifique" uma informação, a escrita de informações também é controlada com outra regra: um sujeito só pode ter acesso de escrita a um objeto se o seu clearance for menor ou igual à classificação do objeto. Desse modo, um oficial com acesso "SECRET" não poderia nunca escrever um objeto com a classificação de "CONFIDENTIAL" (mais baixa), impedindo-o de copiar uma informação restrita para um nível mais baixo que um usuário menos privilegiado poderia ler [RA01].

Deve ficar claro que o usuário com classificação "SECRET" pode utilizar um mesmo programa (como um editor de textos) ou atuando como um subject com rótulo "SECRET" ou com um rótulo menor como "CONFIDENTIAL". Mesmo com os dois sujeitos executando o programa representando o mesmo usuário, eles obtém privilégios diferentes de acordo com os seus rótulos de segurança. Assume-se que os rótulos de segurança no objeto e no sujeito, uma vez definidos, só podem ser modificados pelo administrador do sistema [SAND93].

Essas duas regras simples e mandatórias são expressas na política Bell-Lapadula através de duas propriedades, em que Φ significa o rótulo de segurança do sujeito ou objeto fornecido.

- ***Simple-Security Property***: Sujeito s pode ler objeto o somente se Φ(s) >= Φ(o)
- ****-Property (star-property)***: Sujeito s pode escrever objeto o somente se Φ(s) <= Φ(o)

Essas propriedades são especificadas em termos de operações de leitura e de escrita; em um sistema real existirão operações adicionais (criar e remover objetos, por ex.). Todavia, para ilustrar os pontos principais, é suficiente considerar apenas leitura e escrita: por exemplo, criação e remoção podem ser restritas pela *star-property* já que modificam o estado do objeto em questão. Em muitos casos os sistemas reais militares combinam proteção discrecionária com proteção mandatória, só acionada depois que a avaliação discrecionária é realizada [LEE88].

A conseqüência direta da existência do *star-property* é a proteção contra o roubo de informação por cavalos de tróia, um problema do modelo discrecionário. Esta propriedade se aplica basicamente a sujeitos, já que os usuários humanos são confiados a não roubar informações. Um usuário com classificação "SECRET" pode entrar no sistema utilizando a classificação "UNCLASSIFIED" (criando um sujeito com esse rótulo) e criar um documento com essa classificação, porque assume-se que ele irá colocar apenas informações "UNCLASSIFIED" nele. Por outro lado, não se confia nos programas porque podem possuir

cavalos de tróia embutidos. Um programa (sujeito) rodando com classificação "TOP SECRET" não consegue escrever em objetos "UNCLASSIFIED" porque é restringido pela *star-property*. Um usuário que deseje escrever em um objeto "UNCLASSIFIED" deve entrar no sistema com tal rótulo. É importante observar que apesar de tornar ineficaz a utilização de cavalos de tróia, um atacante obstinado pode ainda tentar "vazar" informação por outros meios, utilizando *covert-channels* [AGAT00].

Existem uma série de nuances e sutilezas geradas por essas duas propriedades básicas da política, mas não será necessário entrar em detalhes para os fins do texto – o leitor interessado dispõe de uma vasta literatura sobre o assunto [RA01] [BL75] [SAND93].

2.2.3 DAC e MAC na atualidade

Tanto os modelos DAC e MAC são utilizados atualmente – o DAC em maior escala, estando presente em diversos sistemas operacionais comerciais como o UNIX, Windows e Netware. Implementações de sistemas MAC são comuns em ambientes militares e de *mainframe*.

Apesar da sua popularidade, eles apresentam problemas próprios que estão possibilitando o crescimento de outro modelo de acesso, o RBAC a crescer, visando resolver estas questões.

O MAC, apesar de ser reconhecido genericamente como mais controlável e potencialmente mais seguro que DAC não tem obtido grande uso fora dos circuitos militares. Isso se deve principalmente pela dificuldade em adaptar fluxos de negócio e hierarquia comerciais ao modelo formal e estritamente hierarquizado imposto por políticas como Bell-Lapadula e Biba, que são as mais amplamente implementadas. Isso faz com que seja considerado imprático implantá-lo em sistemas que não sejam militares, pelo custo de administração e de *overhead* que seria gerado.

O DAC, por sua vez, goza de grande popularidade no mundo comercial, mas tem em seu maior problema a questão da dificuldade no gerenciamento das permissões. Sistemas operacionais modernos de rede possuem milhares de usuários e potencialmente milhões de arquivos espalhados pelos seus sistemas. O gerenciamento das permissões de cada um destes objetos em uma escala como esta não é um problema simples de se resolver, já que cada objeto possui sua própria informação de acesso individual.

Por exemplo, quando um usuário é removido do sistema, costuma ser necessário remover suas informações de acesso a todos os objetos em que possuía o acesso. Da mesma forma, quando um objeto novo é criado, é necessário definir as suas permissões. Em grandes sistemas computacionais multi-usuário, essas questões pesam bastante.

A utilização de grupos de usuários e de *containers* de objetos (diretórios, por exemplo) para facilitar na administração das permissões ajuda, mas não resolve todos os problemas [OSM00].

O que acaba acontecendo é que relaxa-se as permissões de acesso para não causar um *overhead* excessivo no seu gerenciamento, em que o resultado é que a grande maioria dos sistemas possuem usuários com acesso a mais informações do que seria necessário, pela simples complexidade de defini-las e gerenciá-las. Esse tipo de situação acontece em sistemas operacionais como Windows 2000 e UNIXes: pela enorme quantidade de arquivos

de sistema, configuração e chaves de registro, simplesmente não se define informações de acesso apropriadas para cada uma delas.

Uma outra desvantagem tanto do MAC quanto do DAC é que é complicado estender suas regras de acesso para incluir mecanismos de proteção e exceções que acontecem normalmente em sistemas. Como exemplo, toma-se o caso de um sistema hospitalar. Expressar a seguinte regra em sistemas com MAC ou DAC tradicionais é difícil: "Na maior parte do tempo, cada médico só tem acesso aos dados dos seus pacientes. Quando estiver no seu turno na UTI, ele terá também acesso aos pacientes que estão internados na UTI. Ao acabar seu turno, perde este acesso adicional". A complexidade de se expressar esse tipo de comportamento faz com que acabe-se configurando os sistemas com proteções mais relaxadas do que o estritamente necessário.

3 RBAC: Role-Based Access Control

Esta seção fornecerá uma visão abrangente do que consiste RBAC, os problemas que visa atacar, como é modelado, e suas principais características. Assume-se que o leitor tenha um mínimo de familiaridade com conceitos básicos de controle de acesso como DAC e MAC.

3.1 Motivação e contexto

O controle de acesso em geral preocupa-se com determinar que usuários ou grupo de usuários podem realizar que operações em que recursos computacionais. O problema fundamental é que cada sistema e aplicação que inclui mecanismos de controle de acesso tem um método próprio para criação e gerenciamento de usuários, grupos e principalmente uma significado particular de operação e objetos.

Para muitas organizações, o número de sistemas pode ser da ordem de centenas ou milhares, com o número de usuários variando de centenas a centenas de milhares, e o número de objetos a serem protegidos podem facilmente exceder os milhões.

Como a utilização de RBAC ajuda num contexto como esse? RBAC foi projetado para gerenciar centralmente privilégios ao prover uma camada de abstração, conhecida como *role* (papel), mais alinhado à estrutura da organização. A noção central de RBAC é que permissões são associadas à papéis, e usuários são associados aos seus papéis corretos na organização. Isso simplifica bastante a administração e gerenciamento de permissões de acesso em grandes organizações. Papéis são criados para as várias funções de negócio da organização, e os usuários são associados a esses papéis de acordo com suas responsabilidades e qualificações. Usuários podem ser facilmente reassociados de um papel para outro. Papéis podem receber novas permissões de acesso à medida que novas aplicações ou funcionalidades são adicionadas ao sistema, e permissões podem ser revogadas sempre que necessário.

Um papel é apropriadamente entendido como uma construção semântica à volta do qual a política de controle de acesso é formulada. A coleção particular de usuários e permissões associadas a um papel é transitória. O papel é mais estável porque as atividades e funções da organização em geral mudam menos do que o conjunto de usuários ou de permissões.

Isso torna a administração de um sistema RBAC muito mais fácil e escalável do que a de um sistema DAC, por exemplo, que precisa associar usuários a objetos diretamente, sem a construção do papel entre os dois, atuando como componente estabilizador. Diversos estudos mostram a eficácia deste tipo de abordagem [NISTR02].

Além do forte motivo de facilitar o gerenciamento das permissões de acesso, um outro ponto motivador de RBAC é a sua flexibilidade de adaptação a regras de acesso particulares de cada sistema, através do recurso de *constraints*, explicados em seção própria. Isso permite expressar, na política de acesso do sistema, restrições como a separação de deveres, em que um mesmo usuário não pode subverter a segurança do sistema ao exercer dois papéis conflitantes ao mesmo tempo. Por exemplo, um mesmo usuário não poderia ser ao mesmo tempo o gerente de compras (que toma a decisão de realizar uma compra), e o gerente financeiro (que passa o cheque da compra), já que isso poderia abrir espaço para fraudes em que ele indicaria uma compra fraudulenta e autorizaria a sua fatura por conta própria. A separação de deveres é um princípio clássico de segurança, utilizado extensamente no

mundo dos negócios, e é possível de ser expressado como regra de acesso em sistemas RBAC.

A política de controle de acesso está incorporada em vários componentes de RBAC, como as relações papel – permissão, usuário – papel e papel – papel. Estes componentes coletivamente determinar se um usuário particular terá permissão de acesso a um conjunto de dados oferecidos pelo sistema. O engenho RBAC pode ser configurado diretamente pelo administrador do sistema ou indiretamente por papéis apropriados delegados por este administrador. A política em utilização em um dado sistema advém do conjunto das configurações dos vários componentes RBAC realizadas pelo administrador do sistema. Além disso, a política de controle de acesso pode evoluir incrementalmente ao longo do ciclo de vida do sistema – um fato quase certo em sistemas de grande porte. A capacidade de modificar a política de acesso para atender às necessidades mutáveis da organização é outro ponto forte do RBAC.

RBAC é "neutro" em termos de política, já que a sua configuração irá ditar qual a política em ação. Estudos [OSM00] [SM98] mostram que RBAC pode ser configurado para suportar políticas de controle de acesso advindas do MAC, como Bell-Lapadula; outros estudos mostram como RBAC pode ser utilizado para simular DAC em sistemas. Essa flexibilidade permite que um mesmo engenho possa ser utilizado tanto no mundo comercial quanto no mundo militar, apenas mudando as particularidades de adaptação para cada política.

O leitor pode então indagar: RBAC por si, é um mecanismo de controle de acesso discrecionário ou mandatório ? Segundo [SCYF96], esta resposta depende da natureza e configuração dos papéis, permissões e usuários no sistema RBAC. Entendendo mandatório como significando que usuários individuais não tenham nenhuma escolha em relação a que permissões ou usuários são associados a um papel, e discrecionário onde os usuários individuais podem tomar essas decisões. Como dito anteriormente, RBAC é um modelo por si só, sendo independente de política. Configurações particulares de RBAC podem ter um forte componente mandatório, enquanto outros podem ter um forte componente discrecionário.

Dito isso, a utilização mais comum de engenhos RBAC é de caráter mandatório, em que os usuários não têm controle, mas sem ter características de sistemas multinível, com rótulos de segurança.

3.2 Papéis e constraints

3.2.1 Papéis

O conceito de papel vem sendo utilizado desde o início dos anos setenta, em diversos sistemas e aplicações. É um conceito central ao modelo RBAC, sendo uma construção que modela funções organizacionais, unindo usuários e direitos de acesso correspondentes àquele papel. Papéis na organização tem a tendência de mudar muito menos do que usuários ou permissões de acesso, facilitando bastante a administração.

Uma pergunta extremamente freqüente é "qual a diferença entre papéis e grupos?". Grupos de usuários como unidades de controle de acesso são comumente providos em muitos sistemas de controle de acesso. A maior diferença entre o conceito de grupos e de papéis é que grupos são tipicamente tratados como uma coleção de usuários e não uma coleção de

permissões. Um papel é tanto uma coleção de usuários, de um lado, e uma coleção de permissões no outro. O papel serve como um intermediário para juntar essas duas coleções.

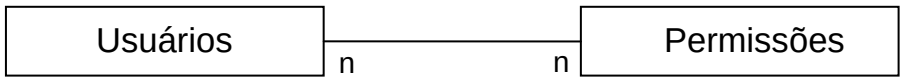


Figura 1: relacionamento usuário - objeto

Figura 2: relacionamento usuário – papel - objeto

Modelos mais modernos de RBAC prevêem também a herança de papéis em que usuários pertencentes a um papel "filho" herdam as permissões de acesso do papel "pai". Por exemplo, uma empresa de software pode possuir o papel de "Membro de projeto", com um conjunto básico de direitos. Além disso, pode possuir outros papéis que herdam desse papel: "Engenheiro de testes" e "Engenheiro de software", cada um herdando os direitos básicos do papel ascendente, e acrescentando um conjunto próprio de permissões de acesso. Herança múltipla também é permitida, de modo que poderia existir um papel "Gerente de projeto" que herdasse os direitos de ambos os papéis de engenharia.

É importante deixar claro que um usuário pode fazer parte de mais de um papel ao mesmo tempo, do mesmo modo que uma pessoa pode acumular funções em uma organização. Alguns engenhos RBAC só permitem, todavia, que se entre no sistema escolhendo um único papel de cada vez, de modo que seria necessário escolher que conjunto de direitos serão necessários no momento do login. Outros permitem que se possa entrar no sistema exercendo mais de um papel ao mesmo tempo, caso em que os direitos de acesso se acumulam.

3.2.2 Constraints

Constraints são um aspecto importante de RBAC, as vezes utilizadas como a principal justificação deste modelo de controle de acesso. Constraints são predicados que, aplicados a relações e funções do modelo, retornam um valor "aceito" ou "não aceito". Eles funcionam como um mecanismo poderoso para expressar políticas organizacionais de nível mais alto, como separação de atividades, em que um usuário não pode exercer dois papéis conflitantes, sendo considerados mutuamente exclusivos. Uma vez que dois papéis são declarados mutuamente exclusivos, não é necessário preocupar-se tanto com a alocação de usuários a papéis, com esta atividade podendo ser delegada com menos risco de comprometimento dos objetivos da política organizacional.

Os constraints podem ser aplicados em vários pontos do modelo RBAC, nas relações entre usuários e papéis (UA – User Assignment) e entre papéis e permissões (PA – Permission Assignment). Além disso, constraints também podem ser aplicados às sessões estabelecidas pelo usuário ao realizar o *login* no sistema.

Constraints são um conceito, por definição, genérico, flexível e dependente de implementação. Entre alguns exemplos do que pode ser feito com constraints estão:

- Exclusão mútua – o exemplo clássico de constraints para permitir a separação de deveres. Atuaria em cima da relação entre usuários e papéis ou no estabelecimento da sessão;
- Permitir ou negar o acesso em determinados horários, de acordo com uma política estabelecida seria um exemplo de constraint em cima da relação papel – permissões: as permissões só seriam válidas em determinados horários.
- Permitir que um marinheiro de baixo escalão (com um papel correspondente) exerça o papel de comandante de navio (um papel com privilégios mais altos) apenas se ele for o último remanescente no barco, em caso de acidente por exemplo. Isso seria possível com um *constraint* em cima da relação usuário – papel.
- Um constraint na relação usuário – papel é forçar uma quantidade máxima de pessoas por papel, um chamado *cardinality constraint*. Um exemplo de utilização é no caso de papel de um gerente de projeto, em que só deve existir uma pessoa exercendo-o.
- Permitir ou negar o acesso quando o usuário utilizar o sistema a partir de algumas origens específicas – de máquinas não confiadas na rede, por exemplo. Isso seria caracterizado como um *constraint* em cima do estabelecimento da sessão do usuário.

Constraints também podem ser vistos como sentenças em alguma linguagem formal apropriada [AS00] [GI96], mas vários estudos seminais da área [SCYF96] fazem um tratamento informal do conceito, ficando uma modelagem mais específica como sendo a cargo da implementação.

3.3 Modelos RBAC do NIST

RBAC ganhou gradualmente mais apoio da comunidade acadêmica e comercial, com diversas implementações aparecendo no mercado. Todavia, a inexistência de um modelo padronizado para RBAC resultava em incerteza e ineficiência sobre seu significado e utilidade. Diversos trabalhos [SCYF96] [SFK00] [FBK99] começaram a propor um modelo padronizado para esse tipo de controle de acesso, que acabou resultando no modelo RBAC do NIST, o padrão de fato para RBAC hoje.

Esta seção traz a visão deste modelo, que é utilizado na implementação do Sentinel. Note-se que o padrão definido pela NIST é extenso e detalhado, sendo grande demais para reproduzir neste trabalho. Além disso, não está no escopo deste trabalho as considerações sobre modelos para administração de sistemas RBAC. Desse modo, aqui serão expostos os pontos norteadores e principais classificações do modelo, ficando o leitor interessado encarregado de consultar a literatura apropriada quando desejado.

3.3.1 Visão geral do modelo

RBAC é um conceito rico e de certo modo aberto, indo de controles simples em um extremo a controles relativamente complexos e sofisticados no outro. Dessa forma, já foi reconhecido [SFK00] que um único molde definitivo ou iria incluir demais ou excluir demais, de forma que o modelo RBAC é organizado em quatro passos de recursos e funcionalidades crescentes. Estes níveis são cumulativos no sentido em que cada um inclui os requisitos do

nível anterior da seqüência. Eles foram definidos e formalizados em um conjunto de artigos [SFK00] [SFGK01]. São eles:

- RBAC1: Core RBAC (também conhecido como "Flat RBAC")
- RBAC2: Hierarchical RBAC
- RBAC3: Constrained RBAC

O recomendado da NIST para RBAC é o RBAC3, o mais completo dos três níveis.

3.3.2 RBAC1: Core RBAC

O Core RBAC é ilustrado na Figura 3. Os recursos requeridos do Core RBAC são obrigatórios a qualquer implementação de RBAC, e são praticamente óbvios. Este nível prevê apenas usuários, papéis e permissões. Usuários (ou grupos de usuários) e permissões são associados a papéis.

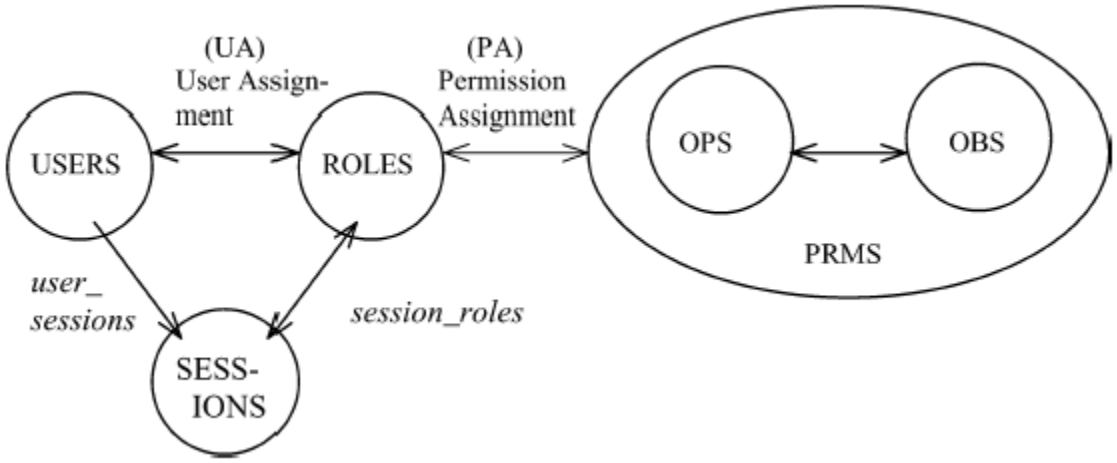


Figura 3: Core RBAC

Uma permissão é a aprovação de um modo de acesso em um ou mais objetos do sistema. Permissões são sempre positivas e conferem a capacidade do detentor da permissão a executar alguma operação no recurso do sistema. A natureza da permissão depende fortemente dos detalhes de implementação e da natureza do sistema. Por isso, um modelo geral de autorização deve tratar, até certo ponto, as permissões como símbolos não-interpretados. A natureza exata das permissões em um sistema é um ponto deixado em aberto por RBAC. Para exemplificar:

- Em um sistema operacional, as permissões poderiam envolver as operações de "leitura", "escrita" e "execução", e os objetos considerados como sendo arquivos;
- Em um sistema bancário, as permissões poderiam envolver operações mais complexas como "débito" e "crédito" em um objeto representando uma conta bancária.

O RBAC1 define duas relações básicas: a relação entre usuários e papéis (UA – User Assignment), e a relação entre papéis e permissões. Ambas são relações *many-to-many*. O modelo RBAC não discorre sobre revogação de permissões ou de usuários de papéis.

3.3.3 RBAC2: Hierarchical RBAC

RBAC hierárquico é ilustrado na Figura 4 - ele difere do RBAC1 somente na introdução da hierarquia de papéis denominada como relação RH. Hierarquias de papéis são comuns na discussão acadêmica e na maior parte das implementações existentes hoje.

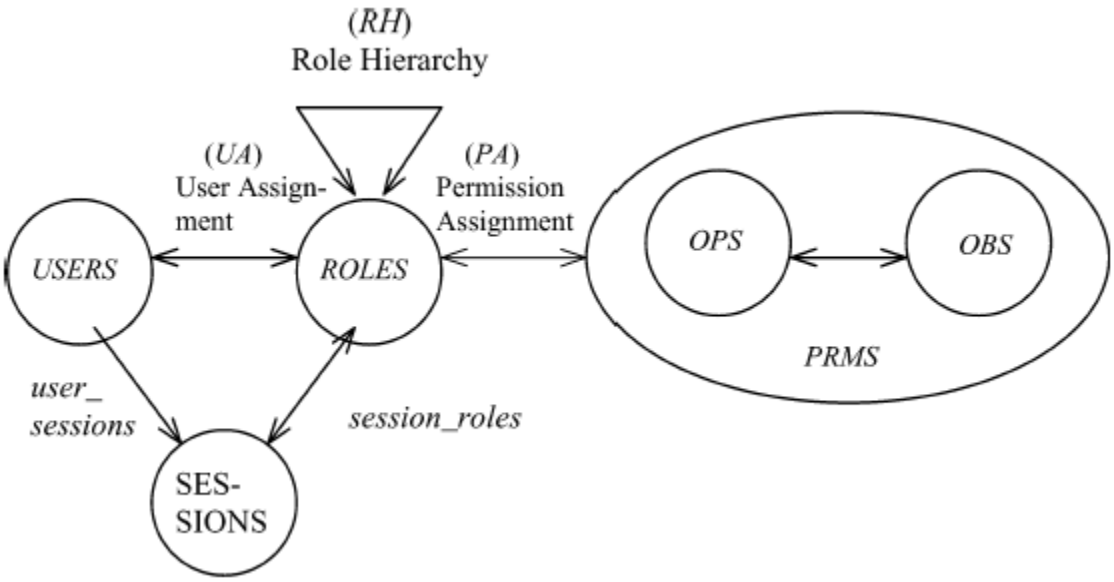


Figura 4: Hierarchical RBAC

Hierarquias de papéis são uma maneira natural de estrutura papéis de forma a representar as responsabilidades de organizações. Como exemplo disso, temos a Figura 5.

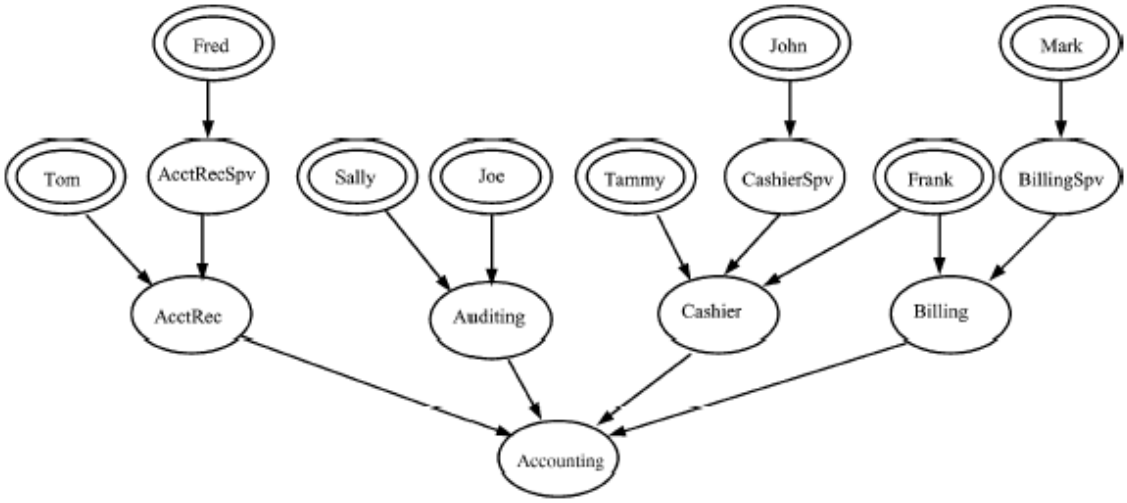


Figura 5: exemplo de hierarquia de papéis e seus usuários

Matematicamente, hierarquias de papéis são uma ordem parcial, possuindo propriedades de reflexão, transitividade e anti-simetria. Um usuário de um determinado papel possui todas as permissões de acesso que os papéis inferiores a ele, do qual ele herda.

3.3.4 RBAC3: Constrained RBAC

RBAC nível 3 é mostrado na Figura 6 e Figura 7, e adiciona o recurso de constraints, explicados anteriormente, ao modelo RBAC. Os constraints podem estar associados à relação usuário – papel e papel – permissão ou à ativação de papéis dentro de sessões do usuário no sistema.

O conceito de constraint é genérico, como visto anteriormente. Por isso, o RBAC3 só especifica um tipo de constraint obrigatório a ser apresentado pelo sistema: a separação de deveres, referente ao particionamento de tarefas e privilégios entre diferentes papéis de modo a garantir que um único usuário não tenha poder demais nas mãos, diminuindo o risco de abuso.

Apesar de só especificar esse tipo de constraint, qualquer outro tipo de constraint pode ser implementado, de acordo com as políticas de cada organização e as características do engenho de controle de acesso. O nível 3 de RBAC do padrão prevê dois tipos de separação de deveres: estático e dinâmico.

Separação de deveres estático

Esse tipo de separação de deveres visa diminuir o conflito de interesses em um sistema baseado em papéis ao não permitir que um mesmo usuário pertença a dois papéis considerados mutuamente exclusivos. Exemplos desse tipo de papel já foram fornecidos anteriormente. A separação é estática porque ela sempre está ativa no sistema, agindo na relação usuário – papel: se um usuário é autorizado como membro de um papel, ele é proibido de ser membro de um papel conflitante. Constraints são herdados na hierarquia de papéis.

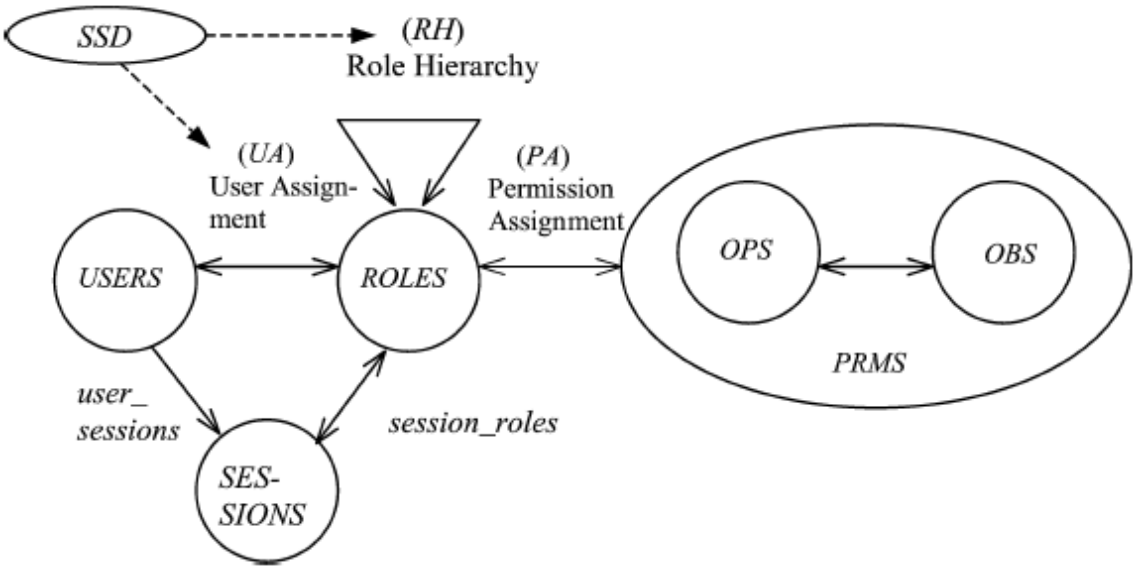


Figura 6: constraints para separação estática de deveres (SSD)

Separação de deveres dinâmico

Esse tipo de separação de deveres fornece a capacidade de tratar situações de conflitos de interesse no momento em que o usuário estaria entrando no sistema como pertencente a

um papel. Ou seja, o constraint age sobre o estabelecimento da sessão do usuário dentro do sistema. Com a separação de deveres dinâmica é possível que um usuário pertença a um conjunto de papéis que não constituem conflito de interesse quando utilizados de modo independente, mas gerem preocupação se utilizados em conjunto.

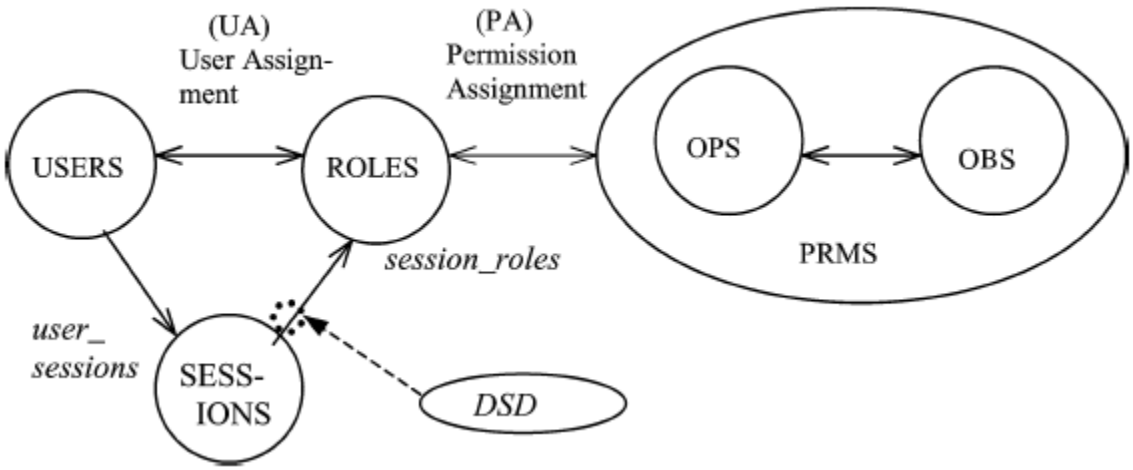


Figura 7: constraints para separação dinâmica de deveres (DSD)

Por exemplo, um usuário pode pertencer ao mesmo tempo aos papéis "Caixa" e "Supervisor de caixa", onde o supervisor tem a permissão de aceitar correções e retiradas maiores do caixa que guarda o dinheiro. Neste caso, uma mesma pessoa pode exercer os dois papéis, mas não ao mesmo tempo: estar logado no sistema como "Caixa" e "Supervisor de caixa" permitiria-o retirar dinheiro sem uma inspeção de terceiros. Com a separação dinâmica de deveres esse tipo de situação é tratada. O usuário seria obrigado a escolher um dos dois papéis para estar logado no sistema, não podendo utilizar ao mesmo tempo os direitos de ambos.

4 Arquitetura do Sentinel

O objetivo deste trabalho é um engenho de controle de acesso baseado em RBAC, chamado Sentinel. O Sentinel deve funcionar como um engenho genérico e abrangente de controle de acesso para aplicações de propósito geral. Um dos objetivos do Sentinel é ser um engenho genérico, que possa ser incorporado a uma aplicação sem muitas modificações. Esta seção descreverá os critérios de design utilizados, e a arquitetura geral projetada para o software.

4.1 Critérios de Design

Para o início de qualquer projeto de software dessa natureza, sempre é interessante deixar bem claro quais são os critérios de design utilizados na sua construção. Esses critérios deixam claro ao leitor e aos desenvolvedores quais são as linhas norteadoras do software no seu desenvolvimento (do ponto de vista técnico).

Quaisquer *trade-offs* no design ou na implementação do Sentinel são realizados tendo esses critérios básicos em mente como requisitos que precisam ter prioridade para serem atendidos.

4.1.1 Segurança

Este é um critério fundamental para um engenho de controle de acesso. Segurança neste contexto de um requisito básico do sistema refere-se principalmente a dois pontos:

- O atendimento às regras de autorização e segurança estabelecidas pelo modelo RBAC, dentro do limite do implementável e do padrão utilizado. Ou seja, o sistema deve realizar autorizações de modo previsível e de acordo com o modelo RBAC, sendo consistente e não oferecendo exceções ou brechas conceituais que permitam burlar uma regra de autorização.

- A implementação do sistema deve ser realizada tendo em vista ataques ao sistema, de modo que deve empregar técnicas de programação segura, filtrando entradas e tendo interfaces bem definidas, por exemplo. O Sentinel deverá funcionar como o centro das decisões de controle de acesso de um sistema que o utilize, de modo que pode vir a ser considerado parte do TCB – Trusted Computing Base [RA01], aumentando os seus requisitos de segurança.

4.1.2 Flexibilidade

Um dos objetivos principais do Sentinel é que ele seja um engenho de controle de acesso RBAC o mais genérico possível, para que possa ser utilizado em uma grande gama de aplicações. Por isso, a flexibilidade na sua utilização é fundamental, sem perder a sua natureza básica de engenho de controle de acesso.

Implementar um engenho RBAC de forma flexível é um desafio, já que a maioria dos artigos e estudos acadêmicos prevê engenhos de controle de acesso sempre amarrados a implementações específicas, e cheios de detalhes próprios. Por exemplo, em toda a literatura o conceito de permissão é notadamente dependente da plataforma ou sistema no qual o engenho de controle de acesso estará inserido.

Isso porque permissão pode ser entendida como a aprovação de uma determinada operação em um determinado objeto. Tanto operação quanto objeto são conceitos dependentes de cada sistema – exemplos anteriores em relação a sistemas operacionais e aplicações bancárias mostram isso.

O Sentinel foi desenvolvido para prover mecanismos que suportem a sua flexibilidade de atuação. Esses mecanismos abrangem basicamente o suporte a alguns tipos básicos de operações e objetos (recursos), e a possibilidade de estender esses conceitos para dependentes da aplicação, sem perder a generalidade do engenho de autorização.

4.1.3 Extensibilidade

A extensibilidade é conseqüência direta dos critérios de flexibilidade que o Sentinel deve suportar, já que muitos deles envolvem a extensão de conceitos para adaptação às particularidades da aplicação, sem perder o componente genérico de autorização.

Além disso, o Sentinel se propõe a ser um componente genérico de controle de acesso, de modo que isso também abrange autenticação e auditoria. Para isso, o Sentinel é baseado em uma arquitetura fortemente orientada a plugins que trazem funcionalidade específica, integrando-se ao framework do engenho.

Por exemplo, diferentes plugins de autenticação estendem as capacidades do Sentinel de identificar usuários utilizando diferentes técnicas: senhas, certificados digitais e tokens vêm à mente.

4.2 Estruturação do engenho

O Sentinel é estruturado de acordo com os "três A's" do controle de acesso: autenticação, autorização e auditoria, de acordo com a Figura 8 abaixo.

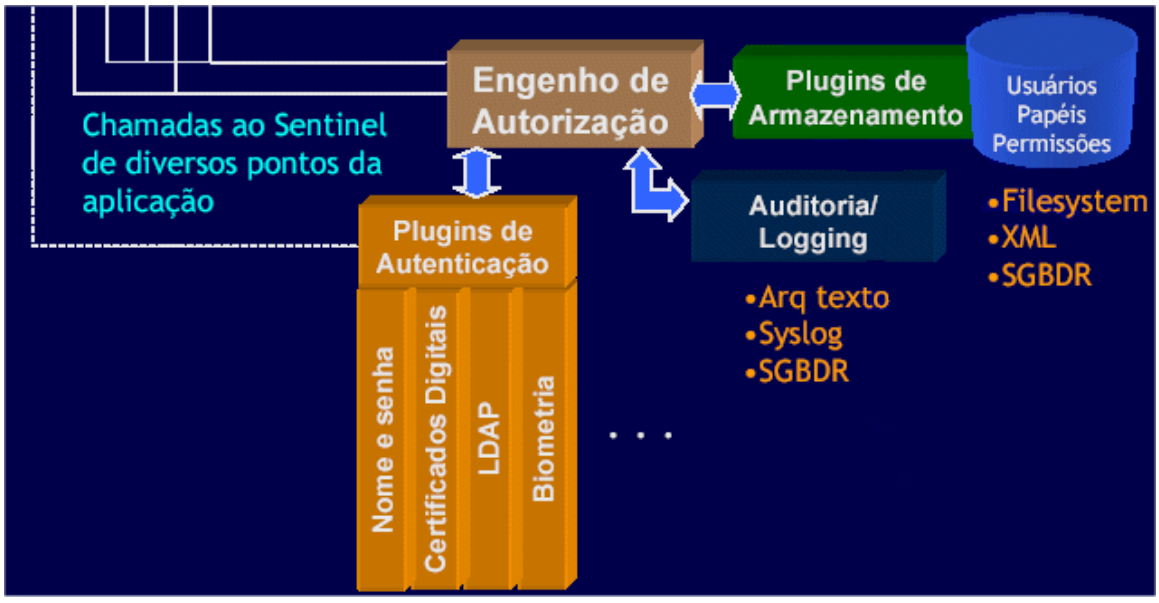


Figura 8: estrutura do Sentinel em Autenticação, Autorização e Auditoria

4.2.1 Autenticação

O processo de autenticação merece atenção especial do Sentinel, já que é nele que o usuário se identifica, e fornece as credenciais de acesso ao sistema.

O módulo de autenticação do Sentinel é flexível e extensível, baseado no conceito de "plugins" de autenticação. Para cada tipo de autenticação desejada, basta implementar um plugin que irá receber os dados de autenticação do usuário, e avaliá-los sobre o conjunto de critérios necessários para o processo de autenticação específico. Alguns exemplos de plugins de autenticação:

- Um plugin de autenticação baseado em senha: recebe o login e uma senha da aplicação, e valida isso contra uma base de usuários do sistema. O plugin retorna um "verdadeiro" ou "falso" de acordo com o resultado da validação. Esta validação vai seguir os critérios implementados por esse plugin em particular; por exemplo, o plugin pode trabalhar com senhas planas ou senhas utilizando *hashes* para melhorar a segurança.
- Um plugin de autenticação baseado em certificados digitais de cliente, funcionando de forma análoga ao anteriormente exemplificado, mas utilizando como critério de autenticação a validade do certificado digital fornecido.

A autenticação em si é realizada dentro dos plugins de autenticação. O fluxo de autenticação acontece da seguinte maneira:

1. A aplicação colhe os dados necessários para a autenticação (exibindo tela de login e senha, recebendo o certificado digital, etc.) e fornece-os ao Sentinel, que submete-os ao plugin de autenticação apropriado.

2. O plugin retorna uma resposta "autenticação positiva" ou "autenticação falha", que determina o comportamento da aplicação que utiliza o Sentinel. O plugin de autenticação deve consultar diretamente a base de usuários do sistema.

3. No caso de uma autenticação negativa, a aplicação recebe esse resultado, ficando livre para informar isso da maneira mais conveniente ao seu usuário.

4. No caso de uma autenticação positiva, a aplicação recebe também uma credencial de acesso (uma estrutura de dados). Essa credencial de acesso deve ser mantida pela aplicação para todos os outros acessos do usuário submetidos ao Sentinel – ou seja, deve ser mantida pela aplicação durante a sessão do usuário.

A credencial de acesso representa o fato de que aquele usuário já foi autenticado pelo engenho, e qualquer outro pedido de autorização ao Sentinel deve vir acompanhado da credencial de acesso do usuário que o requisita.

No Código-fonte 1 é exibido um curto trecho de código exemplificando a autenticação de uma aplicação utilizando certificados digitais de cliente, já colhidas a partir do servidor web da aplicação que faz a conexão SSL. O objeto "sentinel" é a fachada do Sentinel, discutida adiante.

```
Credential credential;
try {
```

```
      credential = sentinel.certAuthenticate(scope, certificateInfo);
      System.out.println("DEBUG: Autenticado com sucesso.");

      sess = req.getSession();
      sess.setAttribute("credential", credential);
} catch (AuthenticationException e) {
      System.out.println("DEBUG: Autenticação falha");
      this.showInvalidAuthenticationPage();
}
```

Código-fonte 1: trecho de código demonstrando autenticação por certificado digital

É importante que a credencial de acesso tenha propriedades de checagem e validação, para tornar mais difícil falsificações e ataques – principalmente no caso em que as autenticações acontecem via rede, e as credenciais podem ficar mais vulneráveis durante o trânsito ou por comprometimento da origem. Um exemplo de checagem é exibido no Código-fonte 2.

```
Credential cred = (Credential) sess.getAttribute("credential");
if (cred == null || ! sentinel.checkValidCredential(cred)) {
      System.out.println("DEBUG:   credencial   invalida   durante   a
sessao");
      this.showPageInvalidCredentials();
}
```

Código-fonte 2: trecho mostrando checagem de credencial durante a sessão

4.2.2 Autorização

O funcionamento do sistema de autorização do Sentinel é o seu propósito principal de existência, e o seu aspecto mais bem explicado até o momento, já que ele visa implementar RBAC nível 3.

Uma particularidade no Sentinel em relação ao modelo é a existência da estrutura de dados "Direito". Em geral, modela-se um papel está associado a usuários e permissões. No caso do Sentinel, um direito funciona como um conjunto de permissões para tornar a gerência mais fácil. Um papel associa-se então a vários direitos, que por sua vez desdobram-se em permissões. Para todos os efeitos, a relação papel – direito funciona da mesma forma, e com os mesmos requisitos e restrições que a relação papel – permissão anteriormente explicada.

O Sentinel implementa hierarquia de papéis: se um dado papel do usuário não contiver a permissão necessária, ele irá checar automaticamente o "papel-ascendente" do qual o papel atual descende. Não é prevista herança múltipla de papéis.

Um outro conceito particular implementado pelo Sentinel é a noção de escopos. Escopos ajudam a implementar um tipo de segurança chamada segurança multilateral [RA01] [BN89], em contraste com a segurança multinível dos modelos militares. A idéia é particionar o domínio de autorização (incluindo usuário e objetos) em vários subdomínios distintos, que não se comunicam entre si. Inclusive, os usuários podem possuir o mesmo papel e direitos, mas estando em domínios diferentes, só terão acesso aos seus próprios dados.

Isso permite que um mesmo engenho Sentinel possa implementar autorização para aplicações que precisam separar seus usuários de alguma forma. Por exemplo, uma empresa pode fazer uma extranet para seus parceiros, com o requisito de que os usuários de um parceiro não vejam nenhum tipo de informação sobre os outros parceiros. Neste caso, cada parceiro estaria num escopo separado. O usuário de um determinado escopo só teria acesso a objetos dentro daquele escopo, mesmo que seu perfil permitisse mais.

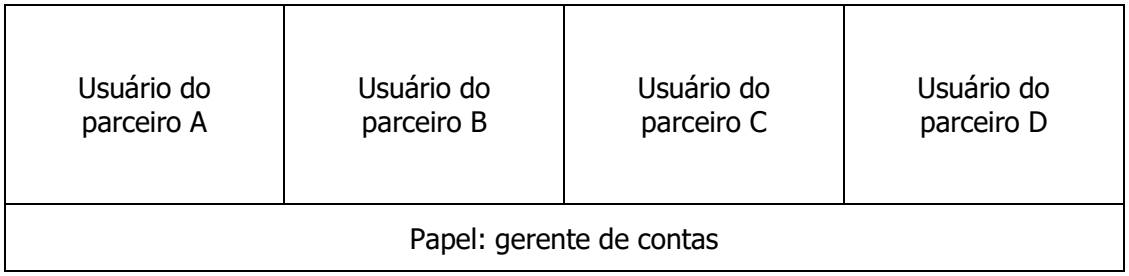


Figura 9: exemplo de usuários de diferentes escopos, possuindo o mesmo papel

O escopo funciona como um primeiro nível de autorização. Implementa-se adicionando um rótulo de escopo a cada usuário, e a cada recurso controlado do sistema. Antes de avaliar qualquer outra regra de autorização, checa-se se o escopo do usuário é igual ao escopo do recurso desejado – se não for, autorização negada. Se for, o processo de autorização continua, para ver se o papel do usuário permite o acesso.

Os *constraints* do sistema são implementados de modo análogo aos plugins de autenticação: pequenos componentes que realizam a checagem apropriada, de acordo com algum critério de política próprio, respondendo com "autorização concedida", ou "autorização não-concedida. Os constraints registram-se em um determinado ponto do Sentinel, de acordo com a relação sobre a qual vão atuar (usuário – papel, papel – direito, etc.). A implementação em si é descrita na próxima seção.

Uma vez que o usuário já está "logado" no sistema (com uma credencial válida em uma sessão), o fluxo de autorização quando a aplicação requisita o Sentinel acontece da seguinte maneira:

1. A aplicação identifica o objeto que deseja requisitar o acesso, através da instanciação de uma classe ResourceID (explicada em seção adiante), passando o identificar do objeto desejado, e o escopo atual do usuário.
2. A aplicação especifica a operação que deseja realizar no objeto anteriormente identificado através da instanciação de uma classe Operation (explicada em seção adiante) e a definição de exatamente que operação que se deseja realizar (por exemplo, leitura ou escrita) no objeto instanciado.
3. De posse da identificação do recurso e da operação que se deseja (ResourceID e Operation), bem como da credencial de acesso válida obtida da autenticação, a aplicação pode submeter o pedido de autorização, passando ao Sentinel esses parâmetros.
4. O Sentinel checa o pedido de autorização, e retorna um resultado correspondente ao status da autorização: aceita ou não-aceita.

Como se vê, tanto a identificação do recurso quanto da operação que se deseja realizar nele devem ser realizadas pelo usuário, através do ResourceID e do Operation. Esses dois componentes estabelecem pontes de ligação com a semântica particular de cada aplicação, já que ambos podem ser específicos para cada aplicação.

Note-se que o repositório onde são armazenadas as informações de autorização (papéis, direito e relações usuário – papel e papel – direito) são independentes do engenho de controle de acesso em si. Apesar de não existir uma estrutura completamente independente (como no caso dos plugins de autenticação), é possível ter vários tipos de armazenamento diferentes (SGBD, arquivos, etc.), já que existe uma camada de dados própria no engenho.

4.2.3 Auditoria

O módulo de auditoria do Sentinel irá realizar o registro dos principais acontecimentos monitorados pelo engenho. Esse módulo é relativamente simples no seu estado atual, já que a atenção maior é concentrada no módulo de autorização.

Não existe uma regra geral do que deve existir dentro de um registrou ou *log* de auditoria – é dependente da aplicação, mas a recomendação geral é um balanço entre informação suficiente para reconstruir os passos do usuário no sistema, e pouca informação o suficiente para não carregar desnecessariamente o sistema.

Dessa forma, o Sentinel registra atualmente (incluindo timestamp):

- Eventos de autenticação com sucesso e falha
- Eventos de autorização com sucesso e falha
- Logoff de usuário

As informações são registradas em um arquivo de *log* em formato texto. Futuras versões do engenho podem incluir outros tipos de informação, ou registrá-las em outro repositório (uma base de dados relacional, por exemplo).

5 Aspectos de implementação e integração

Esta seção tratará dos aspectos de implementação do Sentinel, bem como a sua integração com a aplicação e a sua extensão.

5.1 Integração com a aplicação

O objetivo do Sentinel é ser um engenho de controle de acesso relativamente genérico, que pode ser integrado com aplicações sem que essas tenham que reimplementar um engenho de controle de acesso próprio e completo.

A quantidade de adaptação necessária ao Sentinel irá variar com a complexidade das políticas de controle de acesso da aplicação. Se forem simples e puderem ser expressadas com os plugins, constraints e permissões pré-existentes no Sentinel, será simples. Se não, poderá ser necessário estender o Sentinel para acomodar necessidades particulares. Nestes casos, o desenvolvedor se beneficiará da arquitetura do Sentinel, que foi pensada em termos de extensibilidade, com o conceito de plugins de autenticação e outros controles do engenho.

Independente da complexidade da política, a aplicação precisará minimamente mapear os seus objetos (ou recursos) a serem controlados pelo engenho. Esta seção traz considerações sobre como poderiam ser realizadas essas adaptações

5.1.1 Constraints

Constraints, como explicado em mais detalhes anteriormente no texto, são predicados que, aplicados a relações e funções do modelo, retornam um valor "aceito" ou "não aceito". Constraints agem em pontos específicos do fluxo do Sentinel, especificamente: na relação usuário – papel, na relação papel – direito e no momento de estabelecimento da sessão do usuário, logo após a autenticação.

Constraints no Sentinel funcionam de forma parecida com os plugins de autenticação, sendo expressados como objetos Java que implementam uma interface bem-definida e registram-se junto ao framework provido pelo Sentinel, indicando em qual das relações especificadas acima o constraint atuará. Em cada um dos casos, o plugin de constraint receberá as informações necessárias para tomar a sua decisão:

- **Constraint na relação usuário – papel**
  - Evento de acionamento: um constraint dessa categoria é acionado pelo Sentinel sempre que ocorre alguma alteração na relação entre usuários ou papéis. Ou seja, sempre que um usuário é adicionado ou removido de um determinado papel.
  - Interface de acesso: o plugin de constraint recebe a identificação do usuário e do papel envolvidos. Retorna um valor de aceitou ou não-aceito.
  - Exemplos deste tipo de constraints:
    - Separação de deveres estática, em que um usuário não pode pertencer a dois papéis conflitantes (mutuamente exclusivos)
    - Cardinalidade, em que um papel só pode ter um número máximo de usuários associados

- **Constraint na relação papel - direitos**
  - Evento de acionamento: um constraint dessa categoria é acionado pelo Sentinel sempre que ocorre algum pedido de autorização de acesso de um usuário a um objeto.
  - Interface de acesso: o plugin de constraint recebe a identificação do papel do usuário, do objeto, e das permissões de acesso que se deseja ao objeto. Retorna um valor de aceitou ou não-aceito.
  - Exemplos deste tipo de constraints:
    - Conceder a um médico direitos adicionais nos registros de um paciente da UTI, se o usuário for o médico de plantão da UTI naquele momento

- **Constraint no estabelecimento da sessão**
  - Evento de acionamento: um constraint dessa categoria é acionado pelo Sentinel sempre que um usuário completa a autenticação com sucesso, no momento da construção da sessão.
  - Interface de acesso: o plugin de constraint recebe a identificação do usuário e do papel envolvidos, bem como informações de origem do usuário. Retorna um valor de aceitou ou não-aceito.
  - Exemplos deste tipo de constraints:
    - Separação de deveres dinâmica, em que um usuário não pode estar logado no sistema simultaneamente com os direitos de dois papéis conflitantes (mutuamente exclusivos)
    - Autorizar o usuário de acordo com a origem dele, em que só é permitido o acesso a um sistema se ele vier da rede interna, por exemplo
    - Fazer restrição por horário, em que um determinado papel só pode logar no sistema no horário comercial por exemplo.

Como se vê, os constraints podem variar dos mais simples aos mais complexos, dependendo apenas de como funcionar a implementação do plugin de constraint. O desenvolvedor deste plugin precisa implementar apenas o estritamente necessário, e fazer as adaptações na sua aplicação para registrar o plugin de constraint junto ao Sentinel.

O Sentinel mantém sua independência ao definir interfaces (especificadas em UML) gerais e para cada um dos tipos de constraints acima; o desenvolvedor precisa apenas implementar a interface, com seus critérios internos indisponíveis ao Sentinel. Isso também implica que se o plugin de constraint exigir algum tipo de configuração particular, deve ser implementada também pelo desenvolvedor.

5.1.2 Operações de acesso

Da mesma forma que os constraints, as operações são uma parte essencial do Sentinel, em geral sendo dependentes da natureza da aplicação e podendo ser implementadas em componentes distintos, tal qual os plugins explicados anteriormente.

As operações são diretamente relacionadas aos objetos da aplicação a serem controlados pelo Sentinel. Por exemplo, num sistema que controla o acesso a arquivos, como o sistema

operacional, as operações poderiam ser "leitura", "escrita" e "execução". Como demonstrado em seção anterior, tanto a identificação da operação quanto a identificação do objeto que se deseja controlar são pontos em que o Sentinel identifica-se com a semântica de cada aplicação, já que podem envolver representações particulares de cada sistema que utilize o Sentinel.

Nesta seção serão tratadas as questões de implementação das operações, enquanto considerações sobre os recursos acessados ficarão para a próxima seção.

Deve-se entender uma operação como um componente que representa a operação a ser executada no objeto, não a implementação das funções da operação em si. Por exemplo, se uma aplicação precisa ler de um arquivo, pediria autorização ao Sentinel antes – que iria checar se a operação "leitura" é permitido no objeto arquivo – o componente de operação dentro do Sentinel não iria de fato ler o conteúdo do arquivo, já que isso é tarefa da aplicação. O componente de operação do Sentinel iria apenas validar se a operação é válida, e servir como uma indicação da intenção do usuário. Um outro exemplo seria no caso de operações mais complexas como "débito" e "crédito" em um sistema bancário: o componente de operação do Sentinel não iria de fato implementar as retiradas e entradas de dinheiro. Dessa forma, um componente de operação pode ser bastante simples.

O componente de operação é utilizado em duas ocasiões:

- Pelo engenho, no momento do cadastro das permissões, em que se especifica os recursos a serem controlados, e as operações permitidas naquele objeto;
- Pela aplicação que usa o Sentinel, quando quer autorização para um determinado objeto. A aplicação especifica o recurso que se deseja, e a operação desejada naquele objeto, conforme demonstrado anteriormente.

A segunda situação é demonstrada no Código-fonte 3.

```
try {
      ResourceID rID = new ResourceID("dados.relatorios.vendas",
                                      credential.getScope());
      Operation op = sentinel.newOperation();
      op.enableRead();
      op.enableDelete();
      sentinel.checkAccess(credential, rID, op);
      System.out.println("DEBUG: ACESSO PERMITIDO.");
} catch (AuthorizationException e) {
      System.out.println("DEBUG: ACESSO NEGADO.");
      this.showAccessDenied();
}
```

Código-fonte 3: trecho de código solicitando autorização para acesso a um recurso

Note que neste exemplo, as operações são de leitura, escrita, execução e remoção. Como são simples, podem ser implementadas em um único objeto de classe Operation, e chamam-se métodos para habilitar ou desabilitar individualmente cada operação que se deseja ter acesso ao recurso. No exemplo, deseja-se ter acesso de leitura e de remoção ao relatório de vendas especificado.

Internamente ao Sentinel, essa autorização é checada para avaliar se ela é válida. O processo de validação da operação é realizado pela própria classe Operation associada ao papel, que recebe a tentativa de operação especifica no exemplo, e checa internamente se é válido. Assim, mantém-se a independência no processo de checagem de cada tipo de Operation do resto do *framework* do Sentinel.

Por exemplo, se esse papel possuir direito apenas à operação de "leitura", o objeto Operation associado à permissão do papel iria receber o objeto de tentativa de operação especificado acima pelo usuário e iria rapidamente verificar que o usuário está pedindo um acesso que não possui (já que ele também pediu remoção, que não tem direito).

5.1.3 Mapeando objetos à aplicação

Ainda mais do que as operações e constraints, é sempre necessário que os objetos (daqui por diante chamado de "recursos" para não confundir com objetos Java) a serem autorizados pelo Sentinel tenham um mapeamento e sejam dependentes da aplicação.

Isso acontece porque dependendo da complexidade da aplicação, ela pode sobreviver bem com os constraints e operações que já vem implementadas no Sentinel, e não precise estendê-las para necessidades particulares.

Todavia, cada aplicação controla o acesso a recursos diferentes – algumas a arquivos, outras a funcionalidades ("emitir relatório de vendas", "realizar transferência"), outras a repositórios de dados. Para que o Sentinel possa ser um engenho de controle de acesso genérico e independente de aplicação, ele não pode se amarrar a um único tipo de recurso a ser controlado. É necessário que a aplicação mapeie e identifique os seus recursos a serem controlados para o Sentinel.

Para concretizar o conceito genérico de recurso do parágrafo anterior, a aplicação deve implementar uma classe abstrata conhecida como ResourceID, que define o *namespace* e identifica os recursos a serem controlados pelo Sentinel. Essa classe ResourceID é utilizada em conjunto com o componente de operações, no momento de se configurar as permissões para um papel, e no momento que a aplicação quer ter acesso àquele recurso.

Vários exemplos devem concretizar o papel do ResourceID:

- O primeiro e clássico exemplo é quando a aplicação controla o acesso a arquivos de um sistema (Unix, por exemplo). Neste caso, o namespace é igual ao conjunto dos arquivos do sistema, e um ResourceID pode identificar os recursos (arquivos) simplesmente pelo seu nome:
    - ResourceID: "/etc/passwd"
    - ResourceID: "/home/Lincoln/relatorio.txt"
    - ResourceID: "/lib/libg*"
- Um outro exemplo mais aplicável é quando o sistema controla o acesso a funções dentro da aplicação – instâncias disso seriam "gerar relatório de vendas", "inserir cadastro", "cancelar transferência", ou outros do gênero. Nesse caso, a aplicação deve definir um namespace para identificar unicamente cada uma dessas funcionalidades. Várias opções podem surgir: utilizar como identificador de cada funcionalidade uma indicação do

requisito funcional que a gerou, o nome do *use-case*, ou simplesmente um esquema de nomeação hierárquico que descreva bem as funcionalidades:

  - ResourceID: "funcoes.relatorios.vendas-semestre" para identificar a função de relatório de vendas semestrais convencionando um namespace hierárquico
  - ResourceID: "Funcionalidade001" um código para identificar aquela funcionalidade, convencionando-se um namespace plano
  - ResourceID: "RelatorioVendasSemestral" o mesmo caso anterior, também utilizando um namespace plano

- Um terceiro exemplo, bastante parecido com o anterior, é quando o sistema irá controlar acesso a dados de alguma espécie (identificados e armazenados num SGBD, por exemplo). Os casos são análogos ao anterior, tratando-se de funções:
  - ResourceID: "dados.planilhas.custos-mensais-filial-SP" para identificar a planilha de custos, utilizando um namespace hierárquico
  - ResourceID: ".1.3.6.1.2.1.1.1" para identificar o OID do campo "sysDescr" numa MIB do protocolo SNMP.
  - ResourceID: "PlanilhaDeCustosMensaisFilialSP" para identificar a planilha de custos, utilizando um namespace plano.

Uma das vantagens da utilização de um namespace hierárquico é que pode-se utilizar wildcards como "dados.planilhas.*" ou "funcoes.relatorios.*" para denotar que o usuário tem acesso a todas as planilhas e a todas as funções geradoras de relatórios. A Figura 10 mostra como fica a arquitetura do Sentinel com a adição do ResourceID como "mapeador" entre a aplicação usuária e o Sentinel.

Como se vê, cada aplicação irá implementar o seu próprio ResourceID, de acordo com o tipo de dados que manipula, e da melhor maneira de identificar esses dados de forma única para o controle de acesso. Note-se que os "dados" controlados podem ser na realidade funções da aplicação executadas pelo usuário. De fato, o controle de acesso a funções ao invés de diretamente a dados em repositório deve ser bastante comum. Entre outras coisas, isso deve-se dar pela dificuldade de identificar um dado (registro, coluna) em um SGBD sem impactar em performance; nesses casos, é mais interessante controlar o acesso às funções que consultam o SGBD ao invés de tentar mapear cada possível item de dados do SGBD para os ResourceID's do Sentinel.

Ainda mais que no caso das operações, o ResourceID é um componente bastante independente do *framework* do Sentinel, sendo sempre implementado pela aplicação usuária, já que o mapeamento e identificação são dependentes de cada aplicação. Por isso, a checagem de se um recurso pretendido realmente equipara-se a um recurso cadastrado é realizada dentro do objeto ResourceID. Isso encapsula toda a lógica de definição do namespace e da sua checagem, tornando tudo completamente independente do Sentinel.

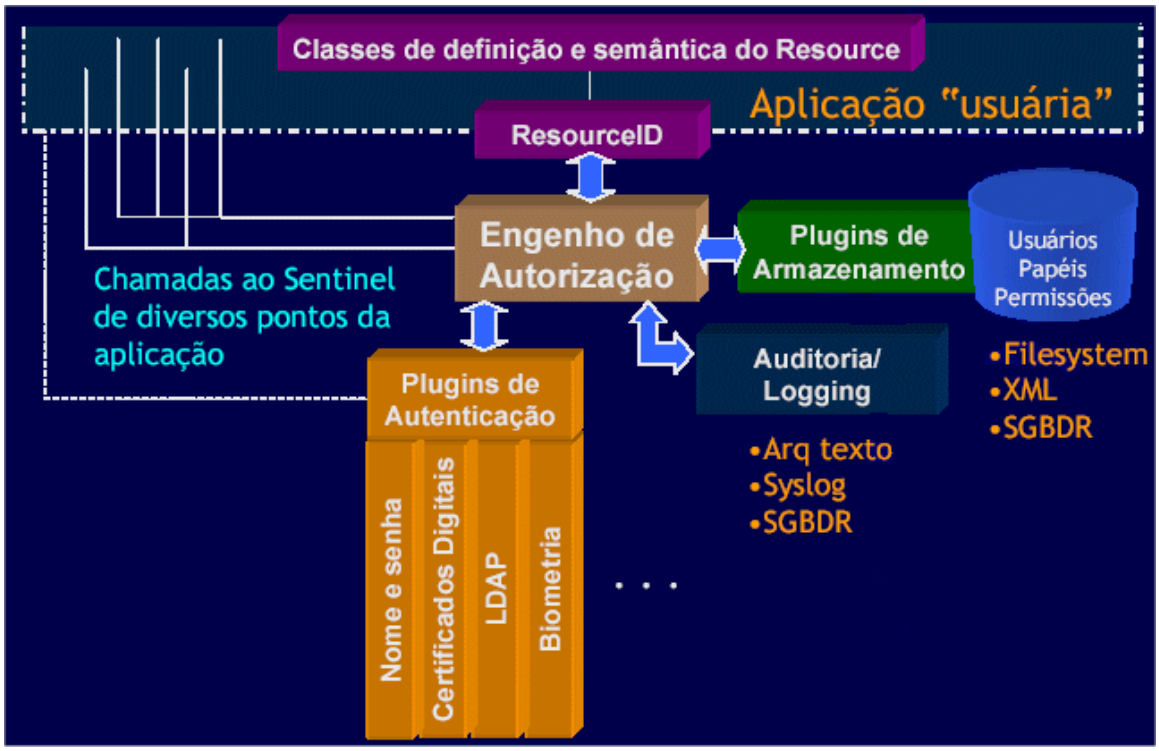


Figura 10: estrutura do Sentinel levando em conta o mapeamento provido pelo ResourceID

Por exemplo, no caso de um namespace hierárquico separado por pontos, a checagem de se o identificador de recurso "dados.planilhas.custos-mensais-filial-SP" pertence ao expressado em "dados.planilhas.*" é encapsulada **dentro** da classe ResourceID provida pela aplicação: é ela que irá mapear o "*" e trazer a semântica que ele equivale a todos os documentos daquele nível para baixo ou não, etc.

O conceito de ResourceID é fundamental para o funcionamento do Sentinel como um engenho de controle de acesso independente de aplicações. É o principal ponto de adaptação, em termos de definição semântica, que a aplicação precisa fazer com o Sentinel. É através do ResourceID que a aplicação usuário descreve o que quer controlar: especificando que recursos cada papel tem acesso, e especificando os recursos que se deseja acessar no momento da autorização.

É bom lembrar que o ResourceID recebe também a identificação do escopo ao qual pertence, para que usuários só tenham acesso a dados do seu escopo. Esse processo é exemplificado no Código-fonte 3.

5.1.4 Métodos de integração do Sentinel à aplicação

Esta seção trata de como o Sentinel irá integrar-se à aplicação como engenho de controle de acesso, descrevendo as diferentes opções que o sistema possui na sua utilização.

Existem duas formas básicas de integrar um engenho de controle de acesso a uma aplicação, cada uma com seus prós e contras: integração estilo "kernel" e integração "opcional".

A opinião autor é que a primeira forma, denominada Integração opcional, será a mais comum em aplicações por ser menos trabalhosa, apesar de potencialmente menos segura.

Integração opcional

Neste estilo de integração, o Sentinel funciona como um módulo independente dentro da aplicação. Em cada momento em que é necessário controlar o acesso a algum recurso, a aplicação pergunta ao Sentinel se o usuário está autorizado a executar aquela operação antes de realizá-la. De acordo com a resposta da aplicação, decide-se o acesso ou não. Essa situação é exibida no trecho Código-fonte 3.

Esse estilo de integração é chamada de opcional porque a rigor a aplicação não precisaria pedir a autorização do Sentinel para então acessar o recurso – poderia dar um *bypass* e acessá-lo diretamente.

Pontos positivos: é muito mais fácil implementar ou retroequipar esse estilo de integração na aplicação, já que não é necessário mudá-la arquiteturalmente de modo significativo: basta incluir as checagens de autorização nos pontos corretos.

Pontos negativos: pode ser considerada menos segura, pois abre a possibilidade de um programador esquecer (maliciosamente ou acidentalmente) de implementar uma checagem de autorização e acessar o recurso diretamente.

A Figura 11 ilustra esse estilo de integração.

Figura 11: estilo de integração opcional

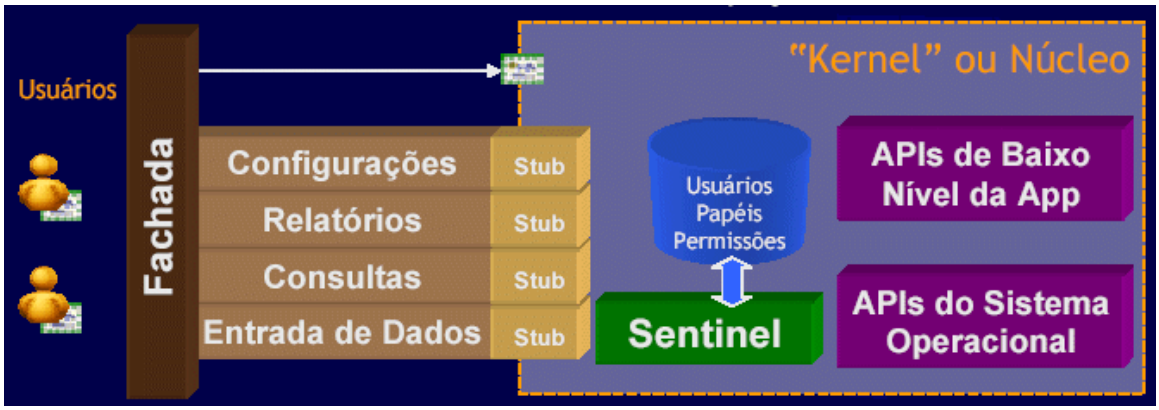
Integração estilo kernel

Neste estilo de integração, a aplicação não pede autorização ao Sentinel para depois acessar o recurso pretendido, como no modelo anterior. Todos os recursos que podem ser acessados pela aplicação são realizadas através de uma interface única, que já incorpora as checagens de autorização.

Esse estilo de integração é chamado de "kernel" porque é parecido com a divisão de *kernel-level* e *user-level* comum em sistemas operacionais. O núcleo da aplicação exporta APIs para execução das funcionalidades e acesso aos recursos; estas APIs já embutem as medidas de controle de acesso do Sentinel. A aplicação utiliza-se dessas APIs para implementar suas tarefas.

Pontos positivos: pode ser considerada mais segura e controlada, já que o programador não tem como esquecer (maliciosa ou acidentalmente) a requisitar autorização de acesso, já que no próprio pedido de acesso aos recursos já existe uma checagem de autorização embutida. Todo acesso será controlado, sempre.

Pontos negativos: requer repensar ou reestruturar radicalmente a aplicação (ou que ela já tenha sido projetada assim) para adaptá-la a esse esquema em que todos os dados ou funcionalidades que precisem ser controladas sejam "embutidas" em um núcleo da aplicação que já implemente as checagens de autorização junto ao Sentinel. Isso também torna o sistema menos flexível a novos tipos de funcionalidades ou dados controláveis, já que será necessário mudar uma interface/fachada já definida.



A Figura 12 ilustra esse estilo de integração.

Figura 12: estilo de integração kernel

5.2 Pontos de interesse na implementação

Essa seção traz uma série de questões miscelâneas da implementação do Sentinel que são de interesse.

5.2.1 Linguagem, concorrência e tratamento de erros

O Sentinel é implementada na linguagem Java da Sun Microsystems. Java foi escolhido por prover uma série de vantagens:

- Independência de plataforma, o que combina bem com o ideal do Sentinel de ser um engenho de controle de acesso RBAC genérico;
- Fortemente orientado a objeto, o que facilita a modelagem e a implementação do sistema de forma extensível, atendendo a outro critério de design do sistema;
- Uma grande biblioteca de classes (utilizando JDK 1.4) contendo diversas funcionalidades utilizadas no sistema, indo desde bibliotecas de rede e criptografia até acesso a dados;
- Bom suporte a ambientes web através de Servlets, já que estima-se que um dos maiores usos de um engenho como o Sentinel seja em aplicações web.

O sistema foi implementado e testado utilizando JDK 1.4, e integrado a aplicações web utilizando Apache/TomCat como *servlet container*.

O aspecto de tratamento de erros do Sentinel é orientado em torno de algumas exceções exportadas pela API do sistema, que a aplicação usuário deve receber e esperar ao fazer acesso às APIs do Sentinel. Essas seções estão estruturadas de acordo com o diagrama UML da Figura 13.

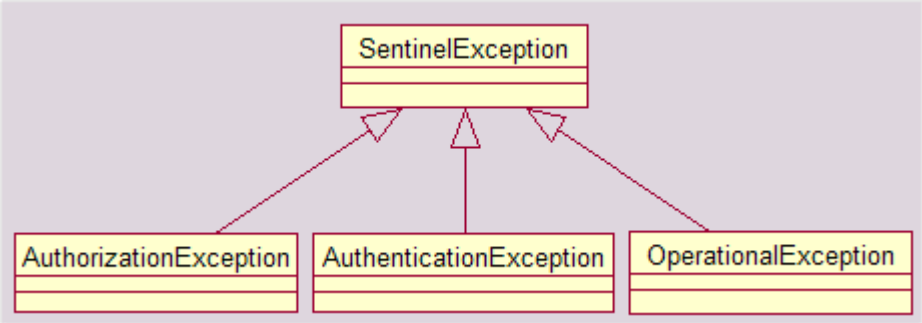


Figura 13: diagrama UML da estrutura de exceções do Sentinel

Todas as exceções exportadas pelo Sentinel descendem da classe SentinelException, sendo elas:

- AuthenticationException: exceção levantada ao ocorrer uma falha na autenticação (uma senha errada, por exemplo), advinda do plugin de autenticação adequado. Pode conter uma mensagem detalhando o problema da autenticação;
- AuthorizationException: exceção levantada quando a aplicação pede autorização de acesso a algum recurso, que é negado. Pode conter uma mensagem detalhando o porquê da falha de autorização;
- OperationalException: uma exceção "genérica" de falha operacional do Sentinel, por algum motivo não-previsto. Deve encapsular outros tipos de exceção internas que não se encaixem nas acima. Por exemplo, se o Sentinel não conseguir contactar o seu banco de dados, ou algum outro tipo de erro interno, será essa a exceção levantada.

Concorrência

O Sentinel precisa ser robusto e tratar bem aspectos de concorrência, dado que será um módulo compartilhado pela aplicação e que receberá a todo momento vários pedidos de autenticação e autorização, em diferentes *threads* de execução.

Além disso, ele possui uma série de estruturas compartilhadas que estarão sendo lidas e escritas constantemente, e que precisam ter sua integridade e consistência protegida. Um exemplo disso são itens como os direitos associados a um determinado papel, ou até mesmo as permissões que compõe um direito: elas poderão estar sendo lidas a cada pedido de autorização, que pode acontecer simultaneamente com uma mudança de configuração (escrita) no papel, removendo ou acrescentando um direito.

Esse controle de concorrência é realizado protegendo as estruturas de dados compartilhadas entre os vários *threads* de requisição com um *reader-writer lock* [LEA99]. Esse mecanismo

não permite que leituras aconteçam enquanto escritas estão acontecendo e vice-versa, além de prevenir múltiplas escritas simultâneas na mesma estrutura de dados. Múltiplas leituras simultâneas são permitidas.

Operações de leitura terão prioridade sobre escritas, de modo que quaisquer operações de leitura que estejam pendentes enquanto uma escrita está acontecendo irão ter executar antes de outras operações de escrita. Garante-se que as escritas sejam executadas na ordem em que foram requisitadas: os pedidos de escrita mais antigos são processados primeiro.

As estruturas de dados compartilhadas no Sentinel em geral são implementadas como Hashtables ou HashSets do Java – por exemplo, os direitos associados a um papel são expressados na forma de um HashSet. Essa estrutura precisa ser protegida contra aspectos de concorrência, utilizando o *reader writer lock* descrito acima.

A estratégia utilizada no Sentinel é criar um *wrapper* que encapsule as chamadas de acesso ao HashSet, envolvendo as chamadas utilizadas em *read-write locks* como explicado acima, distinguindo as chamadas de read e as chamadas de write. Essa é uma estratégia que pode não funcionar quando se quer encapsular objetos mais complexos, mas no caso de objetos com interfaces simples como o Hashtable e o Hashset, funciona bem, gerando as classes SafeHashtable e SafeHashSet, utilizadas pelo Sentinel sempre que precisa-se controlar o acesso a essas estruturas.

Um trecho do SafeHashSet é mostrado no Código-fonte 4, utilizando o objeto auxiliar Reader_writer, que implementa a mecânica do *reader-writer lock*.

```
package sentinel.util;

import java.util.*;

/**
 * @author Cristiano Lincoln Mattos
 * 17/11/2002
 */
public class SafeHashSet {
	private HashSet set = new HashSet();
	private Reader_writer lock = new Reader_writer();

	public SafeHashSet () { }

	public String toString() {
		String res = null;
		try {
			lock.request_write();
			res = this.toString();
		} finally {
			lock.write_accomplished();
		}
		return res;
	}

	public boolean add (Object o) {
```

```
		boolean res = false;
		try {
			lock.request_write();
			res = this.set.add(o);
		} finally {
			lock.write_accomplished();
		}
		return res;
	}

	public boolean isEmpty () {
		boolean res = false;
		try {
			lock.request_write();
			res = this.set.isEmpty();
		} finally {
			lock.write_accomplished();
		}
		return res;
	}

	public boolean remove (Object o) {
		boolean res = false;
		try {
			lock.request_write();
			res = this.set.remove(o);
		} finally {
			lock.write_accomplished();
		}
		return res;
	}
	public boolean contains (Object o) {
		boolean res = false;
		try {
			lock.request_write();
			res = this.set.contains(o);
		} finally {
			lock.write_accomplished();
		}
		return res;
	}

	public Iterator iterator() {
		Iterator res = null;
		try {
			lock.request_write();
			res = this.set.iterator();
		} finally {
			lock.write_accomplished();
		}
		return res;
	}

}
```

Código-fonte 4: trecho do SafeHashSet, mostrando controle de concorrência com *read-write lock*

5.2.2 Configuração do Sentinel

O Sentinel deverá ser utilizado normalmente pela aplicação, em seus aspectos de autenticação e autorização, mas também existem uma série de tarefas de configuração do engenho e de suas regras que precisa ser realizado.

Essa configuração abrange:

- Interfaces de configuração para configuração das regras de acesso, expressas nos usuários, papéis e direitos. Inclui a inclusão de papéis, direitos e as suas associações: por exemplo, adicionando ou removendo um usuário de um papel.
- Configuração de objetos que podem estender o modelo do Sentinel ao serem implementados pela aplicação: *constraints*, operações e ResourceID's. É necessário fornecer mecanismos para que se configure que objetos serão utilizados;
- Outros fatores como informações da camada de dados (endereço do SGBD, etc.) e de auditoria; por exemplo, o nome do arquivo onde seria gravado o *log*.

Esses pontos de configuração ainda podem ser bastante melhorados e estendidos – o arqui exposto deve servir como ponte de partido, sem considerar o assunto acabado.

Usuários, papéis e direitos

Os apêndices detalham os diagramas UML das principais classes do sistema; as interfaces exportadas à aplicação para realização das chamadas de controle de autorização são relativamente poucas e simples.

Uma outra interface também exportada para realizar a configuração dessas estruturas de dados. Essas interfaces são até certo ponto auto-explicativas em relação às suas funções: adicionar um usuário a um grupo, adicionar usuários e direitos a papéis e adicionar permissões a um direito.

As Figura 14 e Figura 15 dão um exemplo dessas interfaces de configuração, que podem ser utilizadas pela aplicação para gerenciar esses componentes. É tarefa da aplicação fazer a interface com o usuário para o gerenciamento desses componentes, utilizando o Sentinel como backend.

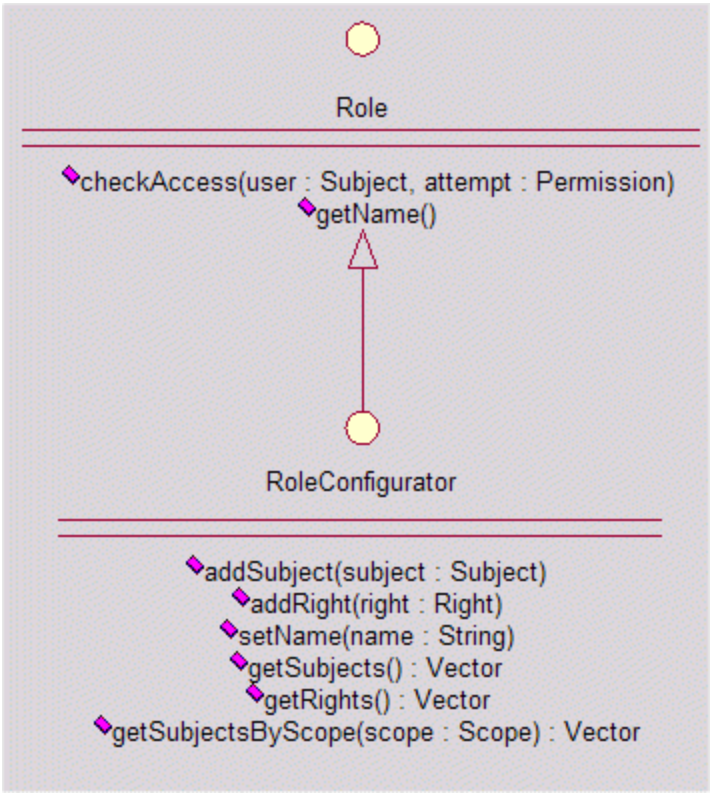


Figura 14: diagrama UML da interface de configuração de papéis

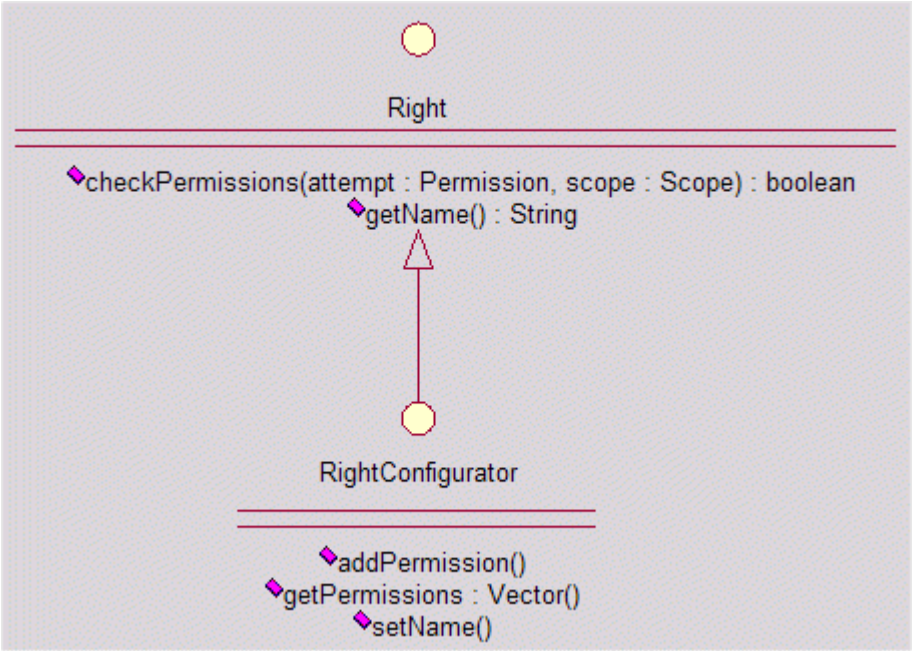


Figura 15: diagrama UML da interface de configuração de direitos

Extensões ao Sentinel

Extensões ao Sentinel podem vir na forma de *constraints*, operações e ResourceID's implementados pela aplicação para estendê-la de alguma forma.

Como essas alterações envolvem reimplementar ou estender classes Java do Sentinel, é necessário que as classes em bytecode (.class) sejam carregadas pelo Sentinel na sua inicialização. Como essas classes irão variar de aplicação a aplicação, é necessário que isso seja especificado em um arquivo de configuração lido pelo Sentinel.

Isso é realizado através de arquivos de *properties* do Java dizendo quais classes devem ser utilizadas para cada uma das extensões. Os arquivos das classes em si devem ficar em um diretório particular onde o Sentinel poderá procurá-las, de acordo com o especificado no arquivo de *properties.*

Isso é exemplificado no Código-fonte 5. O mesmo princípio pode ser utilizado para configuração de classes de *constraints* a serem utilizadas.

```
authenticator.plugin.password = PasswordAuthenticator
authenticator.plugin.certificate = CertificateAuthenticator
authenticator.plugin.smartcard = SmartCardAuthenticator
```

Código-fonte 5: arquivo de properties especificando plugins de autenticação

Outras informações

O mesmo conceito de armazenar opções de configuração em arquivos de *properties* pode ser utilizado para outros tipos de informação miscelânea, como informações de acesso a dados, especificação de nível de *logging* para o sistema de auditoria, etc. Esses pontos devem ser tratados em um detalhamento futuro da implementação do Sentinel, sendo o Código-fonte 6 um exemplo do tipo de informação que pode ser definido.

```
auditing.logfilename = auditlog-Sentinel.txt
auditing.loglevel = DEBUG
data.sgbd.connection.driver = org.gjt.mm.mysql.Driver
data.sgbd.connection.url = jdbc:mysql://10.0.0.1:3306/matrix
data.sgbd.connection.user = sentinel
data.sgbd.connection.password = s23kslJHdf
```

Código-fonte 6: arquivo de properties especificando informações de auditoria e acesso a banco de dados

5.3 Funcionamento como serviço de autorização em rede

O Sentinel como engenho de controle de acesso é independente da aplicação, provendo mecanismos de extensão e integração com aplicações que o utilizam. Dessa forma, uma possibilidade de utilização interessante seria a do Sentinel atuar como um serviço de autorização em um ambiente de rede para sistemas distribuídos.

Essa idéia traz consigo uma série de complexidades que por sua própria conta dariam um trabalho próprio – o leitor interessado possui uma boa introdução em [WL98] e [LABW92]. Reconhecendo ao final do texto que essa é uma linha de pesquisa extremamente

interessante para trabalhos futuros, colocam-se aqui algumas características de funcionamento que uma solução como esta provavelmente possuiria:

- Pelo fato do Sentinel ser implementado em Java, a possibilidade mais interessante para a utilização em sistemas distribuídos seria com CORBA, pelas facilidades inerentes à este *approach* e pela robustez da plataforma. No caso, ao invés de ter uma aplicação nativa chamando as API's oferecidas pelo Sentinel isso seria realizado através de chamadas pela rede utilizando CORBA – sem mudança na API exportada pelo Sentinel.

- Uma outra possibilidade interessante pode ser a utilização de Web Services como mecanismo, se aplicável ao contexto das aplicações distribuídas. Novamente, as APIs exportadas pelo Sentinel não sofreriam maiores alterações.

- Independente do mecanismo utilizado, é essencial que os dados sejam cifrados ao trafegarem pela rede – seja utilizando esquemas próprios de criptografia ou baseando-se em um protocolo conhecido como SSL/TLS – Transport Layer Security. Isso ajuda a prevenir atacantes que interceptem o tráfego de capturar informações de autenticação e de realizar ataques ativos para forjamento de respostas do autorizador central.

- Além da proteção do transporte com criptografia, extremo cuidado precisa ser tomado em relação à estrutura de dados de credencial de acesso gerada – deve ser verificável e dificilmente forjável. Também deve conter dados temporais para diminuir a possibilidade de ataques de *replay*. A utilização da criptografia recomendada anteriormente também ajuda nestes aspectos.

Conclusões e trabalhos futuros

Segurança é um tema cada vez mais freqüente e importante no universo de ciência da computação. De fato, vários fatores conspiram para isso: o aumento dos ataques através da Internet e de redes "internas" às instituições; a massificação e velocidade de propagação de informação de segurança, inclusive com instruções para ataques; e a crescente utilização e até mesmo dependência da sociedade em sistemas computacionais, para gerenciar desde sistemas próprios até redes bancárias e médicas com altos requisitos de segurança.

Neste contexto, o aspecto de controle de acesso em aplicações pode ser considerado uma "fronteira negligenciada" [SAND96] por muitos da área de segurança. Existe uma noção de que controle de acesso é um problema resolvido, com modelos como DAC e MAC em funcionamento há décadas.

No entanto, a realidade mostra que para a grande maioria das aplicações, DAC e MAC não são inadequados. E vemos os mesmos erros e inconsistências sendo cometidos pelas aplicações que implementam controle de acesso por conta própria: desconhecimento do assunto levam a falhas conceituais e de implementação que introduzem riscos desnecessários às aplicações.

Espera-se que o apresentado neste trabalho possa ter sido útil em duas frentes: no sentido de educar desenvolvedores e cientistas da computação sobre os conceitos básicos de controle de acesso em aplicações, apresentando RBAC como um modelo apropriado para uma grande gama de sistemas; e no sentido de prover informação prática sobre um engenho de controle de acesso que pode ser integrado por aplicações com um mínimo de esforço.

Existem várias possibilidades de melhoria no Sentinel e no modelo que ele propõe – este trabalho não pretende ser exaustivo neste sentido. De fato, ele se concentrou nos aspectos mais preementes de um engenho de controle de acesso com requisitos como os do Sentinel, ficando vários outros aspectos com possibilidade de melhoria, tanto de modelagem quanto de implementação.

Em alguns casos, não se detalhou alguns aspectos por serem mais simples e/ou bem entendidos (camada de dados e interface de gerenciamento de usuários, por exemplo). Em outros, por apresentarem oportunidades de pesquisas inteiras por si mesmo, sendo o tempo insuficiente para exploração (autenticador e autorizador via rede, por exemplo). Dessa forma, entre os vários trabalhos futuros que podem ser vislumbrados, estão:

- Melhor tratamento da camada de dados, tornando-a mais flexível e extensível. Possivelmente utilizando uma camada ou *engine* externo, como o Torque[1];
- Como conseqüência do acima, suportar plugins de saída de auditoria, permitindo o registro em vários tipos de repositório: arquivo de log, SGBD, serviço de *syslog*, etc.;
- Construção de uma biblioteca robusta e extensa de plugins de autenticação e de *constraints* que se adequem à grande variedade de aplicações existente, utilizando-se do *framework* definido pelo Sentinel;

[1] http://jakarta.apache.org/

- Construção de interfaces e métodos de configuração padronizadas para os plugins de autenticação e para os constraints que possam ser desenvolvidos para estendê-lo, aumentando a interoperabilidade e a facilidade de gerenciamento do engenho;
- Implementar constraints utilizando princípios de especificação formal de constraints, aproveitando vários artigos interessantes neste aspecto [AS00];
- Pesquisar mais a fundo estender melhor o Sentinel para funcionamento como autorizador centralizado em um ambiente de rede com aplicações distribuídas, com avanços na modelagem e implementação do *framework*.

Entre as várias conclusões que podem ser tiradas a partir do trabalho, resolveu-se destacar uma. Conclusão esta relativamente simples, mas que passa desapercebido por boa parte da comunidade. Não é fácil projetar e implementar um bom e seguro engenho de controle de acesso. Quem é introduzido ao problema pela primeira vez pode ter essa noção, mas entendendo-se as sutilezas da área, e as várias considerações de implementação que precisam ser realizadas para suportar aplicações díspares, distribuídas e com diferentes particularidades, pode-se começar a entender porque existem tão poucos pacotes de controle de acesso com os mesmos objetivos que o proposto neste trabalho na comunidade.

Dessa forma, o autor considerará o trabalho um sucesso se como resultado de sua análise algum leitor aprender mais sobre segurança da informação e controle de acesso, ou for levado a se aprofundar na pesquisar/desenvolvimento da área: isso fatalmente resultará em sistemas mais seguros, um objetivo a ser almejado por todos que trabalham com computação.

6 Apêndice A – diagramas UML

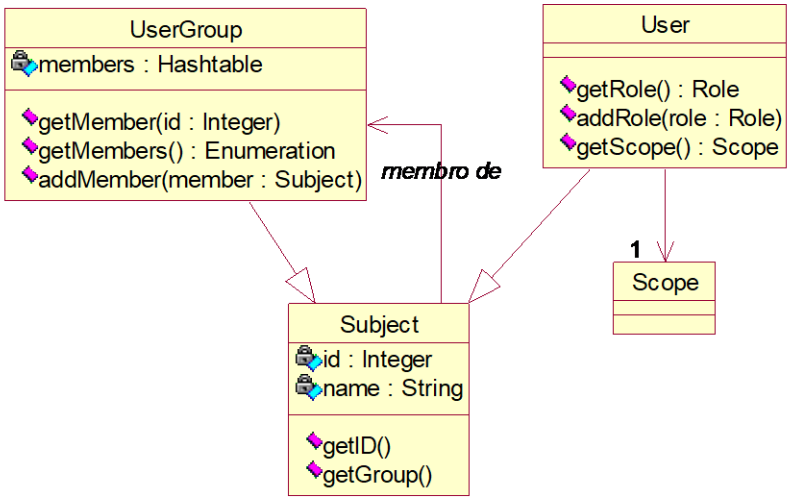


Figura 16: diagrama UML da estrutura de sujeito e sua relação com grupos, usuários e escopos

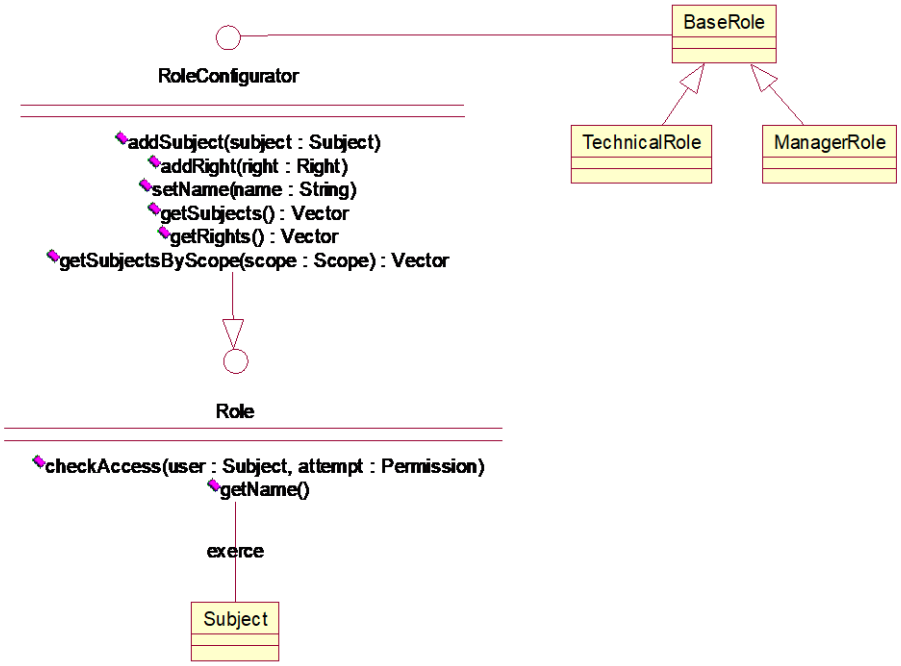


Figura 17: diagrama UML mostrando a relação entre as interface de papel e uma hierarquia de papéis

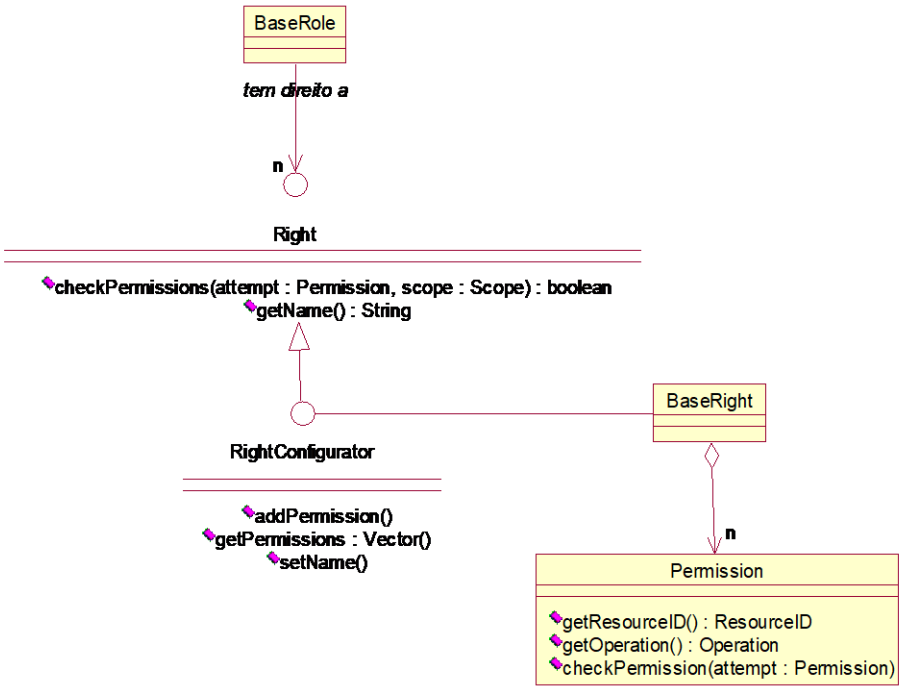


Figura 18: diagrama UML mostrando as interface do papel, a relação com os direitos e como os direitos constituem um conjunto de permissões

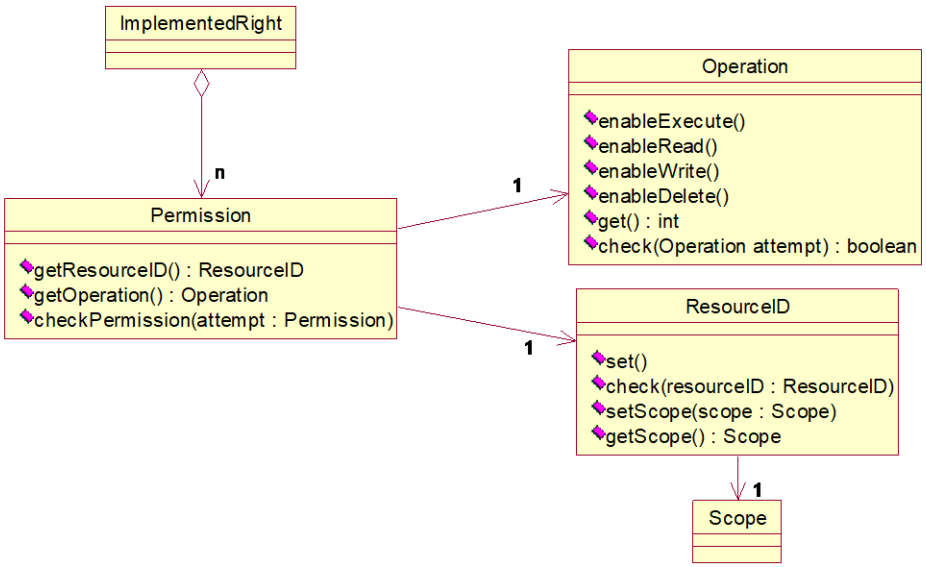


Figura 19: diagrama UML mostrando as relações entre permissão, operações e resource ID (utilizando uma modelo de operação simples, estilo UNIX)

7 Referências bibliográficas

- [AGAT00] J. Agat. Transforming out timing leaks. In POPL '00. Proceedings of the Principles of Programming Languages. ACM Press, 2000.
- [AJSW] N. Asokan, P. A. Janson, Michael Steiner, and M. Waidner, "The State of the Art in Electronic Payment Systems", IEEE Computer, September 1997, pp. 28-35.
- [AS00] G. Ahn and R. Sandhu. Role-based Authorization Constraint Specification. ACM Trans. of Info. and System Security, 3(4), 2000.
- [BIBA77] K.J. Biba. Integrity considerations for secure computer systems. Technical Report TR-3153, The Mitre Corporation, Bedford, MA, April 1977.
- [BL75] D.E. Bell and L.J. LaPadula. Secure computer systems: Unified exposition and Multics interpretation. Technical Report ESD-TR-75-306, The Mitre Corporation, Bedford, MA, March 1975.
- [BN89] Brewer, D.F.C., Nash, M.J. The Chinese Wall Model. Proceedings of the 1989 IEEE Computer Society Symposium on Security and Privacy, pages 215-228.
- [CW87] D.D Clark and D.R. Wilson. A comparison of commercial and military computer security policies. In Proceedings IEEEComputer Society Symposiumon Security andPrivacy, pages 184{194, Oakland, CA, May 1987.
- [EA94] Amoroso, E. Fundamentals of Computer Security Technology, Englewood Cliffs, NJ: Prentice Hall, 1994
- [FBK99] David F. Ferraiolo, John F. Barkley, and D. Richard Kuhn. A role based access control model and reference implementation within a corporate intranet. ACM Transactions on Information and System Security, 2(1), February 1999
- [GI96] Giuri, L. and Iglio, P. A formal model for role-based access control with constraints. In Proceedings of 9th IEEE Computer Security Foundations Workshop, pages 136-145, Kenmare, Ireland, June 1996.
- [LABW92] Butler W. Lampson, Martin Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: Theory and practice. ACM Transactions on Computer Systems, 10(4):265--310, November 1992.
- [LAM71] B.W. Lampson. Protection. In 5th Princeton Symposiumon Information Science and Systems, pages 437-443, 1971. Reprinted in ACMOperating Systems Review8(1):18-24, 1974.
- [LEA99] D. Lea. Concurrent Programming in Java. Addison-Wesley, Reading, Massachusetts, 1999.
- [LEE88] T.M.P. Lee. Using mandatory integrity to enforce commercial security. In Proceedings IEEEComputer Society Symposiumon Security and Privacy, pages 140-146, Oakland, CA, May 1988.

- [LUNT88] Teresa Lunt. Access control policies: Some unanswered questions. In IEEE Computer Security Foundations Workshop II, pages 227-245, Franconia, NH, June 1988.
- [NISTR02] Gallaher, Michael P., O'Connor, Alan C., Krop, Brian. NIST Planning Report 02-1 - The economic impact of role-based access control. National Institute of Standards & Technology. Program Office strategic planning and economic analysis group. March 2002.
- [OSM00] Sylvia Osborn, Ravi Sandhu, and Qamar Munawer. Configuring role-based access control to enforce mandatory and discretionary access control policies. ACM Transactions on Information and System Security, 3(2), May 2000.
- [RA01] Anderson, R.; Security Engineering - a guide to building dependable distributed systems. Wiley Computer Publishing, 2001.
- [RA96] Anderson, R. Security in Clinical Information Systems. British Medical Association, 1996
- [RBKW91] F. Rabitti, E. Bertino, W. Kim, and D. Woelk. A model of authorization for next-generation database systems. ACM Transactions on Database Systems, 16(1), 1991.
- [SAND93] Ravi S. Sandhu. Lattice-based access control models. IEEE Computer, 26(11):9--19, November 1993
- [SAND96] Sandhu, Ravi S. Access Control: the Neglected Frontier. Proc. First Australian Conference on Information Security and Privacy, Springer 1996.
- [SCYF96] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. IEEE Computer, 29(2):38-47, February 1996.
- [SFGK01] Sandhu, R., Ferraiolo, D., Gavrila, S., Chandramouli, R. and Kuhn, R. Proposed NIST Standard for Role-based access control. ACM Transactions on Information and System Security, Vol. 4, No. 3, August 2001, Pages 224–274.
- [SFK00] Sandhu, R., Ferraiolo, D., and Kuhn, R. 2000. The nist model for role-based access control: Towards a unified standard. In Proceedings of 5th ACM Workshop on Role-Based Access Control (Berlin, Germany, July 2000).
- [SM98] R. Sandhu, Q. Munawer. How to do Discretionary Access Control Using Roles. In Proc. of the 3rd ACM Workshop on Role Based Access Control (RBAC-98), Fairfax, VA, USA, October 1998, ACM Press.
- [WL98] Thomas Y.C. Woo and Simon S. Lam. Designing a Distributed Authorization Service. In Proceedings IEEE INFOCOM '98, San Francisco, March 1998.