

UNIVERSIDADE FEDERAL DE SANTA CATARINA - UFSC
CENTRO TECNOLÓGICO - CTC
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA – INF
CIÊNCIAS DA COMPUTAÇÃO

Filipe Ferreira

ESTUDO E A UTILIZAÇÃO DO MODELO DE SEGURANÇA JAVA

Florianópolis, maio. 2008

ESTUDO E A UTILIZAÇÃO DO MODELO DE SEGURANÇA JAVA

Filipe Ferreira

Este Trabalho de Conclusão de Curso foi julgado adequado para a obtenção do título de Bacharel em Ciências da Computação e aprovado em sua forma final pelo Departamento de Informática e Estatística da Universidade Federal de Santa Catarina.

Prof. Luís Fernando Friedrich, D.Sc.

Coordenador do Curso

Banca Examinadora

Prof. João Bosco M. Sobral, D.Sc.

Fernando A. S. Cruz, D.Sc.

Guilherme Arthur Gerônimo

AGRADECIMENTOS

Agradeço a todos os colegas do curso, que sempre estiveram dispostos a incentivar e ajudar nos momentos em que precisamos, aos amigos que estiveram ao meu lado e à minha família por todo apoio que me deram durante estes anos de curso.

SUMARIO

Resumo	7
Abstract.....	8
1 Introdução.....	9
1.1 Objetivos Gerais.....	11
1.2 Objetivos Específicos	11
1.3 Metodologia	11
1.4 Estrutura do trabalho	11
2 O Modelo Sandbox.....	13
3 Segurança na Linguagem Java.....	18
3.1 Construtores da Linguagem Java.....	18
3.2 Compilador	20
3.3 O Verificador de <i>Bytecodes</i>.....	20
3.4 A Máquina Virtual	20
4. O Gerenciador de Segurança	21
4.1 Métodos relacionados ao acesso ao sistema de arquivos.....	22
4.2 Métodos relacionados ao acesso a rede.....	23
4.3 Métodos de proteção à máquina virtual Java	24
4.4 Métodos de proteção das <i>threads</i> do programa	25
5 O Controlador de Acesso	27
5.1 A Classe <i>CodeSource</i>	28
5.3 A classe <i>PermissionCollection</i>	30
5.4 A classe <i>Policy</i>	31
5.5 A classe <i>AccessController</i>	32
5.6 Objetos Protegidos (<i>Guarded Objects</i>)	34
6. Provedores de Serviços Criptográficos	35
6.1 Como os provedores são implementados	37

7 Chaves Criptográficas	39
7.1 A Interface <i>Key</i>	39
7.2 Chaves assimétricas	40
7.2.1 A classe <i>KeyPairGenerator</i>	41
7.2.1.1 A classe <i>KeyPair</i>.....	43
7.2.2 A classe <i>KeyFactory</i>	44
7.3 Chaves simétricas.....	46
7.3.1 A classe <i>KeyGenerator</i>	46
7.3.1.1 Exemplo utilizando <i>KeyGenerator</i>	48
7.3.2 A classe <i>SecretKeyFactory</i>	48
8 Resumos de Mensagem.....	50
8.1 A classe <i>MessageDigest</i>.....	51
8.2 A classe <i>MAC</i>.....	53
9. Criptografia Baseada em Cifradores	56
9.1 Classe <i>Cipher</i>	56
9.1.1 Cifrar e Decifrar dados.....	61
9.1.2 Encapsulamento de chaves	63
9.1.2.1 Exemplo de encapsulamento de chaves.....	64
9.1.3 A Classe <i>SealedObject</i>.....	64
9.1.3.1 Exemplo utilizando <i>SealedObject</i>	65
10 Assinaturas Digitais.....	66
10.1 A classe <i>Signature</i>	67
10.2 A classe <i>SignedObject</i>.....	70
11 Certificados Digitais.....	72
11.1 A classe <i>Certificate</i>	73
11.2 A classe <i>CertificateFactory</i>	74
11.3 A classe <i>X509Certificate</i>	76
11.4 CRL (Certificate Revocation List)	78
11.4.1 A classe <i>CRL</i>	78

11.4.2 A classe X509CRL	79
12 KeyTool	82
12.1 Comandos para adicionar dados a um keystore	85
12.2 Comando para exportar dados	87
12.3 Comandos para visualizar dados	87
12.4 Comandos de gerenciamento da keystore	88
13 Policy Tool	89
Considerações Finais	93
Referências Bibliográficas	94

RESUMO

Este trabalho versa sobre o uso prático da tecnologia de segurança Java, no sentido de orientar o aprendizado dos componentes fundamentais desta tecnologia. A escolha da linguagem Java deve-se ao fato de ser uma linguagem já bastante utilizada no contexto comercial e que apresenta um modelo de segurança bem definido e caracterizado.

A tecnologia de segurança Java inclui um conjunto de APIs que implementam algoritmos de segurança, ferramentas, protocolos. As APIs compreendem as áreas de criptografia simétrica e assimétrica, assinaturas digitais, infra-estrutura de chaves públicas, segurança nas comunicações, autenticação e controle de acesso a recursos protegidos. No decorrer do desenvolvimento deste trabalho, foram testadas as diversas classes da API de segurança.

Como pré-requisito para o entendimento do modelo de segurança Java é necessário, a priori, o conhecimento da linguagem Java com certa profundidade, bem como o conhecimento sobre segurança computacional.

Com a utilização da abordagem seguida neste trabalho, pode-se, a partir de um programa Java qualquer, o qual necessite de segurança, usarmos os pacotes da API de segurança diretamente sobre o programa. Por outro lado, existem ferramentas que podem auxiliar o desenvolvedor a construir aplicações seguras em Java.

O estudo da tecnologia de segurança Java mostra que o modelo de segurança é relativamente abstrato, não trivial, mas é um diferencial existente na linguagem o que justifica a larga utilização da linguagem no mercado de software.

Palavras chave: Segurança, Java, Criptografia.

ABSTRACT

This work is about the practical use of Java technology security, to guide the learning of the fundamental components of this technology. The choice of Java is due to the fact of being a language already used in the commercial context and provides a security model well defined

The security technology includes a set of Java APIs that implement security algorithms, tools and protocols. The APIs include symmetric and asymmetric cryptography, digital signatures, public keys infrastructure, security in communications, authentication and access control to protected resources. During the development of this work, were tested the various classes of security API.

As a prerequisite to the understanding of Java security model is necessary, a priori, the knowledge of Java with some depth and knowledge on computer security.

With the use of the approach followed in this work, you can, from any Java program, to use the security packages directly on the program. On the other hand, there are tools that can help the developer to build secure applications in Java.

The study of Java technology security shows that the model of security is fairly abstract, not trivial, but it is a differential in the language which justifies the wide use of language in the software market.

KeyWords: Security, Java, Cryptography.

1 INTRODUÇÃO

Este trabalho versa sobre **o uso prático da tecnologia de segurança Java**, no sentido de **orientar o aprendizado dos componentes fundamentais** desta tecnologia. A escolha da linguagem Java deve-se ao fato de ser uma linguagem já bastante utilizada no contexto comercial e que apresenta um modelo de segurança bem definido e caracterizado.

A **tecnologia de segurança Java** inclui um conjunto de APIs que implementam algoritmos de segurança, ferramentas, protocolos. As APIs compreendem as áreas de criptografia simétrica e assimétrica, assinaturas digitais, infra-estrutura de chaves públicas, segurança nas comunicações, autenticação e controle de acesso a recursos protegidos.

O Java provê ao desenvolvedor, um *framework* de segurança para escrever aplicações e também um conjunto de ferramentas para gerenciar seguramente as aplicações.

No início, a questão que introduziu a noção de segurança em Java, era o **modelo de distribuição de programas Java**: “carregar na rede um código de algum lugar e executá-lo em outro, dentro de um ambiente computacional”. Isto correspondia ao modelo de segurança Java para a Internet. **Modelo de Segurança** significa que uma definição de segurança bem definida é seguida.

A premissa básica para a segurança em Java foi originalmente e fundamentalmente projetada para proteger as informações em arquivos em um computador, para que essas informações não fossem acessadas ou modificadas enquanto o programa Java fosse executado. As seguintes características são

relevantes:

- Segurança quanto a programas maliciosos:
Não são permitidos a programas, causar danos aos dados numa máquina e a seu usuário.
- Não intrusivo: programas **não devem poder obter informações privadas do computador onde ele é executado** ou de **outras máquinas** na sua rede.
- Verificação das permissões: **regras de operações devem ser especificadas e verificadas.**

A necessidade desses aspectos originou a idéia de um **modelo de segurança** para a linguagem Java, da mesma forma como a **abstração de uma “caixa de areia”** (em inglês, *sandbox*) fornecendo um ambiente seguro para uma criança brincar. Os brinquedos, na caixa de areia, são os recursos de lazer que a criança tem. A criança não pode brincar com brinquedos fora da caixa de areia e esta é responsável por proteger os diversos brinquedos fora da caixa.

A partir deste modelo, um programa Java é seguro ou possui segurança, quando apresenta as seguintes características:

- Criptografia: **dados armazenados, enviados e recebidos são cifrados.**
- Autenticação: a **identificação das partes envolvidas em um programa é verificada.**
- Certificação: programas **devem possuir certificados** que indicam que **certas medidas de segurança estão sendo tomadas.**
- Auditoria: **são mantidos registros de “log” para as operações sensíveis**

ou importantes.

1.1 Objetivos Gerais

O objetivo geral é o uso prático da tecnologia de segurança Java.

1.2 Objetivos Específicos

Como objetivos específicos podemos destacar:

- O entendimento do que é o modelo de segurança Java.
- O entendimento dos componentes do modelo, tais como: o verificador de *bytecodes*, o carregador de classe, o controlador de acesso, o gerenciador de segurança, os provedores de segurança.
- A utilização dos componentes do pacote de segurança (resumo de mensagem, chaves, assinaturas, certificados, criptografia).

1.3 Metodologia

O foco desta monografia segue uma metodologia voltada ao ensino de como se deve usar a tecnologia de segurança Java. De uma forma geral, os capítulos apresentam os seguintes aspectos: (1) descrição de conceito; (2) descrição do método; (3) descrição das classes (APIs) apropriadas para implementar o método escolhido; (4) apresentação de exemplo.

1.4 Estrutura do trabalho

No que segue, este trabalho contém no segundo capítulo, o modelo de segurança *SandBox*. O capítulo 3 expõe sobre a segurança da linguagem. No capítulo 4 temos o Gerenciador de Segurança. O capítulo 5 diz respeito ao Controlador de Acesso. O capítulo 6 trata dos Provedores de Segurança. O sétimo capítulo contém o processo de geração de chaves e o problema do gerenciamento

de chaves. O capítulo 8 contém Resumos de Mensagens. No capítulo 9, criptografia baseada em cifradores. O capítulo 10 sobre as assinaturas digitais. O capítulo 11 trata de Certificação Digital.

2 O MODELO SANDBOX

Na plataforma Java, o modelo de segurança pode se aplicar a qualquer aplicação Java, e esse modelo é configurável e está inserido dentro da arquitetura da plataforma Java. O modelo é configurável pelo usuário final ou por um administrador de sistema, de modo que esses possam restringir a execução de um programa, dentro de um ambiente de execução. O modelo de segurança é centrado na idéia de um *sandbox*, ou seja, um programa é hospedado num computador e um ambiente de execução é configurado para ele com certas restrições para a execução. É dado ao programa, o direito de acesso a certos recursos do sistema. Esta idéia faz com que o programa fique confinado no *sandbox*. Assim, parâmetros para o *sandbox* podem ser modificados e o programa executará com certas permissões [4].

Modificar os parâmetros do *sandbox* de significa alterar as políticas de segurança que o programa utiliza. Em algumas circunstâncias, não é permitido modificar as políticas programaticamente. Se um programa desejar ter acesso a um arquivo, o usuário terá que primeiramente modificar as políticas de segurança de sua máquina antes de rodar o programa. Fazendo uma analogia, seria como expandir ou diminuir os limites da caixa de areia para uma criança brincar, dando a essa criança mais ou menos brinquedos para ela brincar, dependendo do seu comportamento [4].

O modelo *Sandbox* (caixa de areia) é composto por cinco elementos:

Permissions (Permissões), **Code Sources** (Fontes de Código), **Protection Domains** (Domínios de Proteção), **Policy Files** (Arquivos de Políticas) e **KeyStores** (Banco de chaves).

■ Permissões

Uma permissão é uma ação que pode ser praticada pelo código. As permissões são compostas por um tipo (*type*), um nome (*name*) e suas ações (*actions*) caso exista.

O tipo de uma permissão é o próprio nome da classe. Os nomes e as ações são relativos ao tipo da permissão. Por exemplo: o nome de uma permissão de arquivos é o nome do arquivo ou o diretório ao qual pertence e as ações permitidas são *read* (ler), *write* (escrever), *execute* (executar) e *delete* (excluir).

■ *Code Sources*

As fontes de código representam a localização de onde uma classe é carregada juntamente com informações sobre quem assinou esta classe. A assinatura de uma fonte de código é opcional e a localização é representada por uma URL. A combinação de uma URL com o seu *Code Source* é denominado codebase.

■ *Protection Domains*

Domínios de proteção é um conjunto de permissões de um *code source* particular. Por exemplo, códigos da URL www.sun.com podem ler arquivos de um computador pessoal.

■ *Policy Files*

Arquivos de políticas são os elementos administrativos que controlam a caixa de areia. Contém as permissões de um determinado domínio de proteção.

■ *Keystores*

Armazenam uma coleção de chaves criptográficas e certificados digitais.

Este modelo de segurança denominado *Sandbox*, está inserido dentro da arquitetura da plataforma Java.

A arquitetura desta plataforma é composta pelos elementos mostrados na figura abaixo:

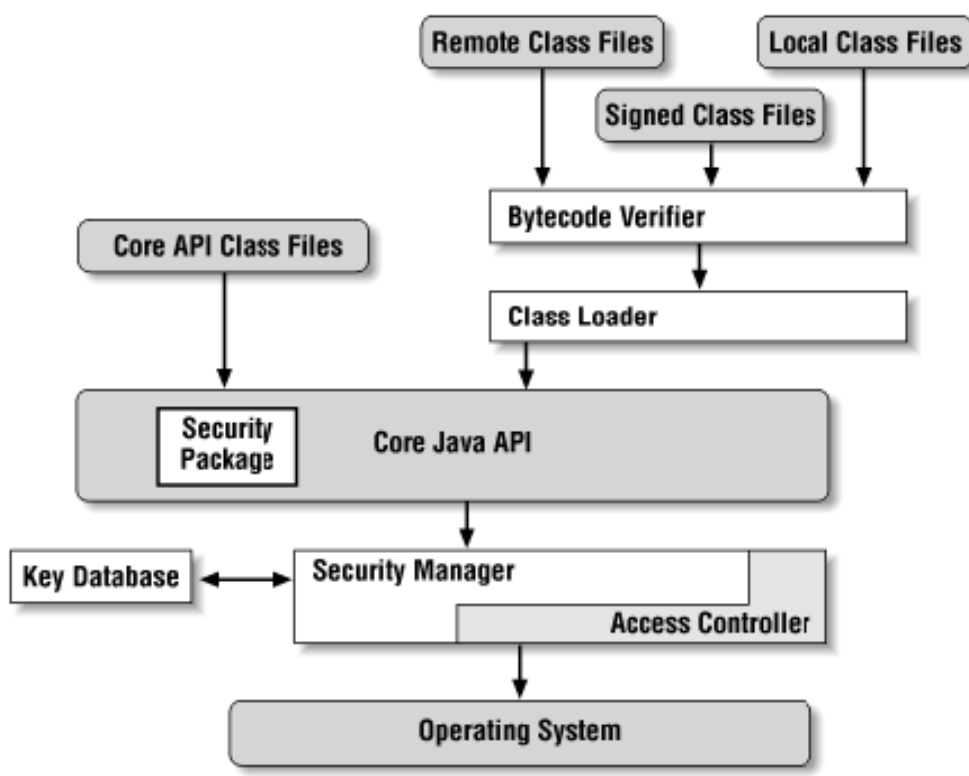


Figura 2.1 Elementos da plataforma Java

■ *Bytecode verifier*

O verificador de bytes deve ser discutido a partir da compreensão do processo de compilação de um arquivo Java. Primeiramente os arquivos com a extensão Java são compilados em arquivos com a extensão class pelo compilador javac. Um arquivo class não contém o código nativo ao processador da máquina. Ao invés disso, possui os denominados *bytecodes* que são interpretados pela *Java Virtual Machine* (Máquina Virtual Java) e traduzidos para a linguagem suportada pelo processador.

O verificador garante que os arquivos de classes Java sigam as regras da linguagem de programação Java. Como mostra a figura, nem todos os arquivos .class são submetidos ao verificador de bytecodes. Apenas os códigos carregados fora do *classpath* (caminho de onde as classes locais são carregadas).

■ *Class loader*

Os carregadores de classes são responsáveis por carregar todas as classes Java. Programaticamente, os carregadores de classes podem dar permissões para cada classe que é carregada.

■ *Access Controller*

O controlador de acesso permite ou previne a maioria dos acessos do núcleo da API Java ao sistema operacional, obedecendo as políticas de segurança.

■ *Security Manager*

O gerenciador de segurança é a interface entre o núcleo da API Java e o sistema operacional. O gerenciador recebe uma requisição, repassa ao controlador de acesso que verificará a permissão dentro do arquivo de políticas de segurança.

- *Security Package*

É conjunto de classes que permite aplicar características de segurança aos aplicativos e cobre os seguintes referentes às Interfaces Provedores de Segurança, Resumos de Mensagem, Chaves, Certificados, Assinaturas Digitais, Criptografia e Autenticação.

- *Key Database*

São as implementações das *KeyStores* que armazenam as chaves e certificados digitais.

3 SEGURANÇA NA LINGUAGEM JAVA

Os primeiros componentes de estudo da caixa de areia são aqueles voltados a segurança da linguagem Java. Estes componentes, primariamente, protegem os recursos da memória da máquina de um usuário e seguem uma série de regras que a linguagem impõe para a garantia da integridade e segurança dos dados [4].

Em termos de *applets*, os componentes devem garantir o acesso restrito ao espaço de memória alocado para a *applet*, sem interferir no espaço de memória de uma outra *applet*.

Os elementos responsáveis pela aplicação das regras de segurança citadas são o compilador, o verificador de bytecodes e a máquina virtual.

3.1 Construtores da Linguagem Java

Dentro de um programa Java, toda entidade, um objeto ou um tipo primitivo de dados, possui um nível de acesso associado. Dentre os níveis, podemos destacar:

- *private*: a entidade só pode ser acessada pelo código contido dentro da classe;
- *default* or *package*: a entidade pode ser acessada pelo código da classe, ou por uma classe do mesmo pacote que a classe que define a entidade;
- *protected*: pode ser acessada pelo código da classe, por uma classe

do pacote e pela subclasse [4];

- *public*: qualquer classe possui acesso a entidade.

Estes níveis de acesso garantem o acesso restrito a uma entidade, dependendo do nível de acesso associado à entidade.

Um exemplo da importância da integridade da memória está no uso das *applets*. Suponha que ocorra um acesso a jogo dentro de uma *applet* carregada através de um site qualquer. Ao mesmo tempo, outra compra é efetuada em um shopping virtual. A *applet* que está rodando o jogo não deve ter acesso aos dados do cartão de crédito do usuário que está realizando compras pelo shopping virtual.

Sendo assim, a linguagem Java foi criada tendo como base as seguintes regras:

- **Métodos de acesso são rigorosamente respeitados;**
- **Programas não podem acessar locais de memórias quaisquer;**
- **Entidades declaradas como *final* não devem ser modificadas:** variáveis do tipo *final* são consideradas constantes e são imutáveis após a inicialização da mesma;
- **Variáveis não podem ser usadas antes de serem inicializadas:** seria como um acesso de leitura a um local de memória aleatório;
- **Limites dos arrays devem ser verificados antes de serem acessados:** por exemplo, se um array de inteiros reside ao lado de uma string na memória, caso ocorra uma escrita no array sem antes verificar o limite do mesmo, poderá sobrescrever o valor da string;
- **Objetos não podem ser arbitrariamente convertidos em outro objeto qualquer:** Os objetos só podem ser convertidos em objetos de

suas superclasses ou subclasses.

3.2 Compilador

O compilador é responsável por transformar arquivos java em arquivos de classe que contém os *bytecodes*. E, também, por aplicar todas as regras exceto as duas últimas: verificação dos limites de um array e a conversão de objetos [4].

3.3 O Verificador de *Bytecodes*

Este verificador existe para prevenir que *bytecodes* oriundos de outros compiladores que não seguem as mesmas regras de compilação do fabricante, sejam utilizados [4].

Isto porque um compilador pode ser criado para criar e explorar um furo de segurança por ignorar alguma regra da linguagem Java.

3.4 A Máquina Virtual

A máquina virtual é responsável pelas últimas duas regras: a verificação dos limites de um array e a conversão de objetos em um outro objeto qualquer é feito em tempo real de execução [4].

4. O GERENCIADOR DE SEGURANÇA

O gerenciador de segurança é a interface entre a API do Java e o sistema operacional. Como gerenciador, ele dita as regras de acesso aos recursos do sistema operacional, seguindo as políticas de segurança do aplicativo [5].

Caso o acesso ao recurso seja negado, ocorre o disparo de uma exceção do tipo `SecurityException`.

Fazendo uma analogia com a caixa de areia, seria como se houvesse alguém controlando a criança que está dentro da caixa de areia. Se a criança quisesse buscar algum brinquedo fora da caixa, isto não seria permitido, pois o monitor da criança conhece os limites dela. Quando o monitor negasse o brinquedo desejado à criança, ele daria um aviso informando que a criança não poderia brincar com este brinquedo que está fora da caixa.

A arquitetura Java permite que um novo gerenciador de segurança seja utilizado. Programadores que desejam, por algum motivo, criarem seus próprios gerenciadores de segurança, podem instalá-los.

A manipulação de um gerenciador de segurança é feita pela classe `System`. Esta classe possui dois métodos para trabalhar com o gerenciador de segurança [5]:

`public static SecurityManager getSecurityManager()`

Retorna o gerenciador de segurança atual.

`public static void setSecurityManager(SecurityManager sm)`

Instala o novo gerenciador de segurança

O método `setSecurityManager` só pode ser chamado uma única vez dentro de um programa. Uma vez instalado, o gerenciador de segurança será usado

por todas as classes executadas dentro da máquina virtual.

O gerenciador implementa diversos métodos para proteção de acesso dos recursos do sistema. Dentre os métodos estão os de proteção de acesso ao sistema de arquivos, à rede, à máquina virtual, às *threads*.

Abaixo serão demonstrados os métodos públicos do gerenciador de segurança e será feita uma análise lógica sobre cada um deles.

4.1 Métodos relacionados ao acesso ao sistema de arquivos

Os métodos mais conhecidos da classe `SecurityManager` estão relacionados com o acesso aos arquivos de uma rede local. Neste escopo estão os arquivos de um disco local bem como arquivos que estão fisicamente localizados em uma outra máquina, mas que fazem parte do sistema de arquivos local.

Abaixo, os métodos de controle de acesso de arquivos para ler, escrever ou excluir um arquivo qualquer:

public void checkRead(FileDescriptor fd)

public void checkRead(String file)

public void checkRead(String file, Object context)

Verificam se o programa tem permissão para ler um arquivo qualquer.

public void checkWrite(FileDescriptor fd)

public void checkWrite(String file)

Verificam se o programa tem permissão para gravar o arquivo do parâmetro.

public void checkDelete(String file)

Verifica se o programa tem permissão para excluir um arquivo qualquer.

Por default, na maioria dos navegadores que possuem Java, as classes

não confiáveis (classes carregadas fora do CLASSPATH) não têm permissão para acesso a qualquer tipo de arquivo. Isto porque, se uma classe não confiável tiver permissão para leitura de arquivos, dados sigilosos poderiam ser lidos, modificados ou excluídos facilmente.

4.2 Métodos relacionados ao acesso a rede

O acesso a uma rede em Java é realizado por meio de abertura de um *socket* de rede. Uma classe não confiável pode abrir somente um *socket* para a máquina a partir do qual foi carregada.

Esta restrição nas classes não confiáveis foi projetada para evitar dois tipos de ataques. O primeiro refere-se a uma *applet* que usa sua máquina maliciosamente para atacar uma terceira máquina, o que pode causar problemas legais para o dono da máquina.

O segundo tipo de ataque diz respeito à coleta de informações sigilosas contidas em uma rede local. Para este tipo de ataque, geralmente é utilizado um *firewall* que restringe o acesso de máquinas não autorizadas.

Existem dois tipos de *sockets*: *socket* cliente e *socket* servidor. O cliente inicia uma conversa com o soquete servidor e o servidor aguarda uma solicitação do *socket* cliente.

Para verificar a possibilidade de acesso a uma rede, os seguintes métodos são utilizados:

public void checkConnect(String host, int port)

public void checkConnect(String host, int port, Object context)

Verificam se o programa pode abrir um soquete cliente para a porta e o *host* fornecidos.

public void checkListen(int port)

Verifica se o programa pode criar um soquete servidor que está escutando a porta fornecida como parâmetro.

public void checkAccept(String host, int port)

Verifica se o programa pode aceitar (em um *socket* servidor existente) uma conexão cliente que foi originada do *host* e da porta fornecidos.

public void checkMulticast(InetAddress addr)

public void checkMulticast(InetAddress addr, byte ttl)

Verifica se o programa pode criar um *multicast socket* em um endereço *multicast*.

public void checkSelfFactory()

Verifica se o programa pode modificar a implementação de um soquete padrão que é baseada no protocolo TCP. Entretanto, um programa pode instalar um gerador de soquete baseado no protocolo SSL.

4.3 Métodos de proteção à máquina virtual Java

Os métodos de proteção a máquina virtual Java tem como objetivo proteger a integridade da máquina virtual Java e o gerenciador de segurança. Os métodos protegem a máquina virtual de forma que as classes não confiáveis não burlem as proteções do gerenciador de segurança e da própria API do Java.

public void checkCreateClassLoader()

O carregador de classes (*Class Loader*) desempenha um papel importante no modelo de segurança Java porque inicialmente apenas ele conhece determinadas informações sobre as classes que foram carregadas na máquina virtual. Somente o carregador de classes sabe onde uma determinada classe foi originada e se uma classe foi assinada ou não.

Para obter informações a respeito das classes, o gerenciador de

segurança obtém através do carregador de segurança. Por esta razão classes não confiáveis não tem permissão de criar carregadores de classes. Para esta operação ocorrer com sucesso, o domínio da proteção utilizada deve conter uma permissão em tempo de execução com o nome de *createClassLoader*.

public void checkExec(String cmd)

Este método é usado para evitar a execução de comandos arbitrários do sistema por classes não confiáveis. Uma classe não confiável não pode, por exemplo, um processo separado que remova todos os arquivos do seu disco. Para o sucesso da operação, o domínio da proteção atual deve conter um arquivo de permissão com o nome que indique o comando e a ação de execução.

public void checkLink(String lib)

O método *checkLink* evita que um código não confiável importe um código nativo. Por exemplo, a execução de uma função C que a classe não confiável deseja invocar, deve residir em uma biblioteca carregada e colocada em sua máquina.

Esta operação será executada com sucesso se, em seu domínio de proteção existir permissão em tempo de execução com o nome de *loadlibrary.<lib>*.

public void checkExit(int status)

Este método evita que uma classe não confiável finalize a máquina virtual.

public void checkPermission(Permission p)

public void checkPermission(Permission p, Object context)

Verifica se o processo atual possui a permissão atribuída. O método procede se o domínio da proteção atual caso a permissão tenha sido concedida.

4.4 Métodos de proteção das *threads* do programa

As *threads* são divisões de um programa em duas ou mais tarefas que

rodam simultaneamente. A Java Virtual Machine permite ter múltiplas *threads* rodando concorrentemente.

O gerenciador de segurança realiza a proteção das *threads* com os seguintes métodos:

public void checkAccess(Thread g)

Verifica se o programa tem permissão para alterar o estado da *thread* fornecida.

public void checkAccess(ThreadGroup g)

Neste caso o programa verifica a permissão para alterar o estado do grupo de *threads* fornecidos.

public ThreadGroup getThreadGroup()

Retorna o grupo de *threads* ao qual a *thread* pertence.

Uma classe não confiável pode manipular suas próprias *threads* e manipular grupos de *threads* criadas pela classe. Além disso, *threads* pertencentes a classes não confiáveis devem pertencer a um grupo de *thread*. Devemos ter cuidado ao interromper uma *thread*, pois isto corrompe o estado da máquina virtual [3]

O método `getThreadGroup()` não é responsável por decidir se o acesso a um recurso particular deve ser concedido ou não, e não emite uma exceção de segurança. O objetivo do método é determinar um grupo de *threads* que uma determinada *thread* deve pertencer. Quando uma *thread* é criada e não é colocada em um grupo de *threads* particular, o método `getThreadGroup()` fica responsável por atribuir a esta *thread* um grupo a qual ela deve pertencer. Por default, uma *thread* pertencerá ao grupo de *threads* que a chamou [3].

5 O CONTROLADOR DE ACESSO

O gerenciador de segurança, quando recebe uma requisição de acesso a algum recurso do sistema operacional, verifica através do controlador de acesso se existe permissão para usufruir o recurso.

Esta permissão é verificada através das políticas de segurança do programa. Estas políticas de segurança encontram-se no arquivo *java.policy* localizado no diretório */jre/lib/security* dentro da pasta onde o Java foi instalado.

Continuando com a analogia da criança dentro da caixa de areia, seria como se o monitor da criança, representado pelo gerenciador de segurança, possuísse um assistente que verificasse as permissões da criança quando esta solicitasse algum brinquedo.

O controlador de acesso é desenvolvido com base em alguns conceitos:

- *Code Sources* (fontes do código): encapsulamento do local onde as classes Java são obtidas.
- Permissões: encapsulamento de uma requisição para realizar uma operação particular.
- Políticas: encapsulamento de todas as permissões específicas que devem ser concedidas a determinados *code sources*.
- *Protection domains* (domínios de proteção): encapsulamento de um *code source* particular e as permissões admitidas à *code source* em questão.

As classes que dão suporte a estes conceitos, não são manipuladas diretamente pelos programadores. A análise lógica feita serve apenas para demonstrar como internamente o gerenciador de segurança e o controlador de acesso utilizam estas classes para o controle de acesso aos recursos do sistema

operacional.

As classes serão demonstradas abaixo.

5.1 A Classe `CodeSource`

Um *code source* é um objeto que reflete a URL de onde uma classe é carregada e as chaves utilizadas para assinar esta classe. O carregador de classes utiliza os *code sources* para carregarem as classes java.

Para a criação de um objeto `CodeSource` é necessário especificar de onde as classes serão carregadas. O construtor da classe é demonstrado a seguir.

`public CodeSource(URL url, Certificate cers[])`.

Cria um objeto *code source* para o código carregado a partir da URL especificada. O array de certificados é opcional, e contém as chaves públicas que identificam o código carregado a partir da URL. Estes certificados são obtidos através de um arquivo jar assinado.

Dois *code sources* são considerados iguais, se foram carregados da mesma URL e se os arrays de certificados forem iguais.

`public final URL getLocation()`

Retorna a URL que de onde as classes serão carregadas.

`public final Certificate[] getCertificates()`

Retorna uma cópia do array de certificados que foram utilizados para assinar o *code source*.

`public boolean implies (CodeSource cs)`

Determina se o objeto *code source* implica no parâmetro passado ao método, de acordo com as regras da classe `Permission`. Um objeto da classe `CodeSource` implica em outro caso, como dito anteriormente, possuir a mesma URL

e o mesmo array de certificados.

5.2 A classe `Permission`

A classe `Permission` é uma classe abstrata que representa um acesso a determinado recurso do sistema.

Devido aos diferentes tipos de ações, as permissões são especializadas de acordo com o recurso do sistema. Por exemplo, permissão de acesso ao sistema de arquivos é representada pela classe `java.io.FilePermission`. Já a permissão de acesso a rede é representada pela classe `java.net.NetPermission`.

As classes especializadas estendem a classe `Permission`, o que leva a implementarem os métodos abstratos da classe `Permission`.

Os métodos abstratos são demonstrados abaixo:

`public abstract boolean equals (Object o)`

Tem o objetivo de testar a igualdade de objetos. Geralmente verificam a igualdade de nome e das ações (*actions*) que os objetos implementam.

`public abstract int hashCode()`

Este método retorna o código *hash* da classe. Para que o controle de acesso funcione corretamente, o código *hash* de um objeto `Permission` não deve ser alterado durante a execução da máquina virtual.

`public abstract Boolean implies(Permission p)`

É uma das chaves da classe `Permission`. Responsável por determinar se é ou não concedida uma permissão a uma classe que já tem outra permissão.

`public abstract String getActions()`

Retorna as ações da permissão na forma de uma string.

As ações comuns da classe `Permission` são:

`public PermissionCollection newPermissionCollection()`

Retorna uma coleção de permissões para guardar as instâncias adequadas deste tipo de permissão. Esta coleção de permissões será explicada adiante.

`public void checkGuard(Object o)`

Verifica através do gerenciador de segurança se uma permissão foi concedida ao objeto, gerando uma exceção do tipo `SecurityException` caso não tenha permissão.

5.3 A classe `PermissionCollection`

A classe `PermissionCollection` é uma classe abstrata que representa uma coleção de objetos de permissões. Isto facilita a chamado do método *implies* para todos os objetos `Permission` contidos na coleção. Por exemplo, suponha que um usuário possa acessar diversos diretórios do sistema de arquivos. Quando o usuário for acessar determinado diretório, o controlador de acesso verificará na coleção de permissões, se possui permissão de acesso através das permissões contidas nesta coleção de permissões.

Toda classe `Permission` deve implementar uma coleção de permissões que agrupará objetos de permissão da classe em questão e, assim, operados como uma unidade única. Esta implementação é feita através do método `newPermissionCollection()` da classe `Permission`. Através deste método é criada uma coleção de permissões de um tipo de permissão específica, como por exemplo, permissões de arquivo. A permissão de arquivo é representada pela classe `FilePermission` e sempre que for necessário adicionar uma permissão a esta coleção, ela deve ser, neste caso particular, uma permissão de arquivo.

A classe `PermissionCollection` é definida desta maneira:

public abstract PermissionCollection()

Cria uma instância da classe. Este objeto será usado para agrupar um conjunto homogêneo de permissões. Por exemplo, para agrupar todas as permissões de arquivo.

public abstract void add (Permission p)

Adiciona uma permissão ao conjunto de permissões.

public abstract boolean implies(Permission p)

Verifica se determinada permissão é concedida através da verificação do conjunto de permissões da instância da classe.

public abstract Enumeration elements()

Retorna um `Enumeration` (conjunto de elementos em seqüência) de todas as permissões da coleção.

5.4 A classe Policy

É uma classe abstrata que representa as políticas de segurança para um determinado aplicativo Java. As políticas de segurança representam as permissões disponíveis para códigos de diferentes fontes. Este conjunto de permissões formam as denominadas estratégias de segurança.

Os métodos utilizados pelo controlador de acesso para manipular as políticas de segurança são:

public static Policy getPolicy()

Retorna o objeto de políticas utilizado.

public static void setPolicy(Policy p)

Instala o novo objeto de políticas.

public abstract Permissions getPermissions(CodeSource cs)

Cria um objeto de permissões contendo o conjunto de permissões fornecidas as classes originadas de um determinado *code source*.

public abstract void refresh()

Atualiza o objeto de políticas, quando, por exemplo, um novo objeto de políticas tenha sido instalado.

Em termos de programação, escrever uma classe de política de segurança envolve implementar estes métodos. A classe de políticas de segurança padrão é a `PolicyFile`.

A classe `PolicyFile` não é disponível aos programadores. Contudo é possível criar sua própria classe de políticas e instalar esta classe. Instalar uma nova classe de políticas envolve alterar o arquivo *java.security*. Dentro deste arquivo existe a propriedade `policy.provider` que, por default, aponta para a classe que implementa as políticas de segurança. A propriedade é a seguinte:

```
policy.provider = sun.security.provider.PolicyFile
```

Alterando esta propriedade, apontando para a classe de políticas implementada, esta nova política será utilizada pelo aplicativo.

5.5 A classe `AccessController`

Esta é a classe que representa o controlador de acesso e é responsável pelas operações de controle de acesso aos recursos e decisões.

A classe `AccessController` não possui instâncias. Ela possui um construtor privado de modo que objetos desta classe não podem ser instanciados.

A classe possui diversos métodos estáticos que podem ser chamados

para determinar se uma operação em particular pode ser realizada.

O principal método da classe requer uma permissão particular e determina, baseado no objeto `Policy` instalado, quando ou não uma permissão deve ser concedida:

public static void checkPermission(Permission p)

Verifica se a permissão é fornecida em relação à política de segurança em questão. Se a permissão é concedida o método retorna normalmente, caso contrário será disparado uma exceção de controle de acesso: `AccessControlException`.

Existe uma forma de driblar o controle de acesso utilizando o método `doPrivileged` conforme abaixo:

public static void doPrivileged(PrivilegedAction action)

public static Object doPrivileged(PrivilegedExceptionAction action)

Concede permissão temporária para a ação ser realizada com as permissões contidas no domínio de proteção. A diferença do primeiro método para o segundo é que caso uma exceção seja disparada na execução da *action*, esta exceção poderá ser recuperada através do método:

public Exception getException()

Retorna a exceção disparada pela ação que estende a interface `PrivilegedExceptionAction`.

`PrivilegedAction` e `PrivilegedExceptionAction` são interfaces que contém um único método chamado `run()`. Quando a ação `doPrivileged()` é invocada, o método `run()` contido em uma das interfaces é executado. Dentro deste método, não é verificado se existe permissão para realizar os métodos que serão executados, por terem privilégios de acesso.

5.6 Objetos Protegidos (Guarded Objects)

Um objeto desta classe tem a função de proteger o acesso a um objeto encapsulado. Funciona como se um objeto possuísse um protetor ou um guarda. Caso alguém queira acessar o objeto protegido, terá que primeiro verificar se o guarda fornece acesso ao objeto.

O construtor da classe precisa de dois objetos. O primeiro refere-se ao objeto que será protegido pelo seu guarda e o segundo é o objeto guarda que implementa a interface `Guard`.

public GuardedObject(Object objeto, Guard guarda)

Cria um `GuardedObject` embutindo o objeto em seu guarda. O acesso ao objeto será fornecido se seu guarda permitir.

public Object getObject()

Retorna o objeto embutido caso a permissão seja concedida. Se não há permissão de acesso ao objeto, a exceção `AccessControlException` será disparada.

O objeto que realiza a proteção do objeto em questão pode ser qualquer classe que implemente a interface `Guard`. Esta interface possui um único método:

public void checkGuard(Object o)

Verifica se é permitido acessar o objeto do parâmetro. Este método é executado quando ocorre uma chamada ao método `getObject()` e dispara a exceção `AccessControlException` caso a permissão de acesso ao objeto não seja concedida.

6. PROVEDORES DE SERVIÇOS CRIPTOGRÁFICOS

A principal classe responsável por implementar os chamados provedores de segurança é a classe `Provider`. Cada provedor de serviços criptográficos (CSP - *Cryptographic Service Provider*) contém uma instância da classe `Provider` que contém o nome do provedor e uma lista de algoritmos que este implementa. Quando uma instância de um algoritmo particular for requerida, esta é procurada na base de dados do provedor e, se existir, uma instância é criada [6].

Os provedores implementam os chamados motores de criptografia. Um motor é uma classe de operação que fornece uma interface a um determinado serviço de criptografia, independente de um provedor específico ou um algoritmo de criptografia particular. Dentre os provedores que implementam os serviços criptográficos estão `Sun`, `SunJSSE`, `SunJCE` e `SunRsaSign` [6].

Estes provedores contêm os pacotes que fornecem as implementações concretas para o algoritmo de criptografia específico. Provedores adicionais podem ser adicionados estática ou dinamicamente através das classes `Provider` e `Security`, mas isto não será objeto de estudo. Informações de como criar e instalar um provedor de segurança pode ser encontrado na documentação da SUN (<http://java.sun.com/javase/6/docs/technotes/guides/security/>) [6].

Os usuários podem configurar seu ambiente de execução para especificar a ordem de busca dos provedores. Esta é a ordem em que os provedores serão consultados para um serviço requisitado, quando nenhum provedor específico for solicitado. O arquivo `java.security` contém os provedores instalados. Estes estão listados em ordem de preferência de busca [6].

```
security.provider.1=sun.security.provider.Sun
security.provider.2=sun.security.rsa.SunRsaSign
```

Esta disposição dos provedores pode ser alterada administrativamente,

modificando a ordem de busca dos mesmos.

Para utilizar um serviço criptográfico, uma aplicação requisita por um tipo de objeto, por exemplo um `MessageDigest` e um algoritmo particular, por exemplo “MD5”, e requisita uma implementação de um dos provedores instalados aleatoriamente ou de um provedor específico. A especificação desta classe será demonstrada adiante. [6]

```
md = MessageDigest.getInstance("MD5");  
md = MessageDigest.getInstance("MD5", "ProviderC");
```

A figura 6.1 abaixo, mostra a requisição de um algoritmo “MD5” implementado pelo objeto `MessageDigest`. Nesta figura existem três provedores diferentes. Os provedores estão ordenados por ordem de preferência da esquerda pra direita. Na ilustração da esquerda, o algoritmo é requisitado sem especificar o provedor. Na ilustração da direita, o provedor “ProviderC” é especificado

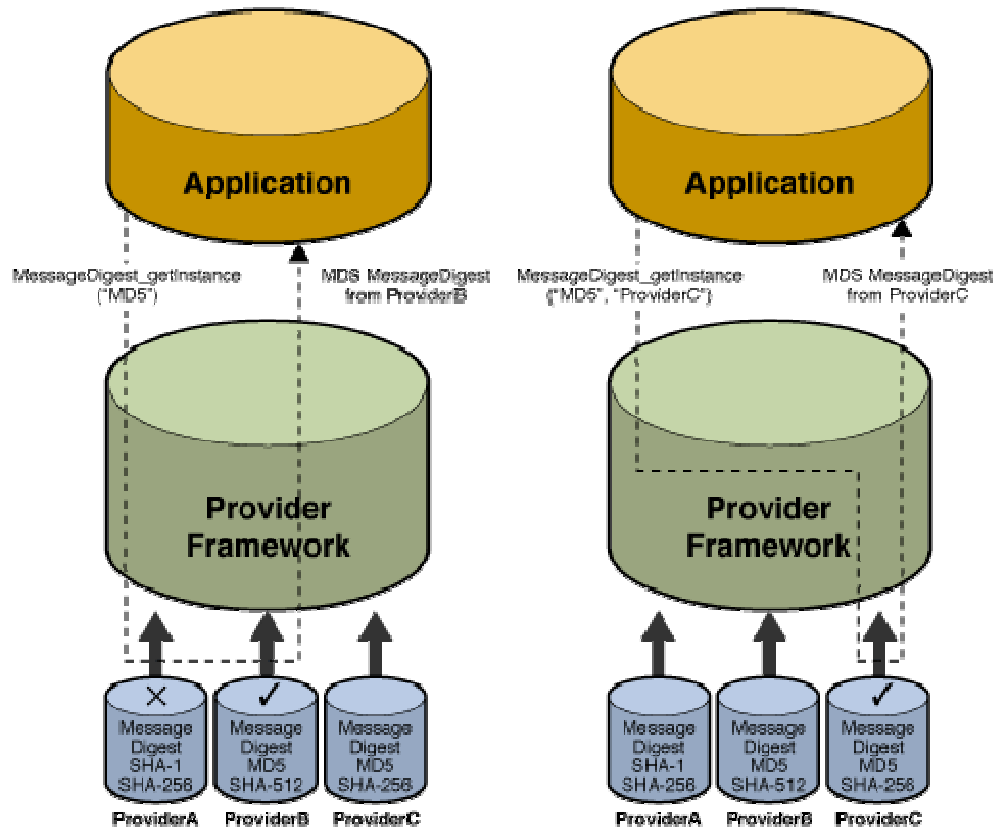


Figura 6.1 Provedores de Serviços Criptográficos

6.1 Como os provedores são implementados

Em cada motor de criptografia, as classes são direcionadas para a implementação do provedor através de classes que implementam a Interface Provedora de Serviços (SPI - *Security Provider Interface*). Isto é, para cada classe motora, existe uma classe SPI abstrata correspondente. Esta classe define os métodos que cada provedor deverá implementar. O nome da classe SPI é o mesmo da classe motora seguido de SPI. Por exemplo, o motor `Signature` provê as funcionalidades de um algoritmo de assinatura digital. A implementação do provedor é fornecida pela subclasse `SignatureSpi`.

As implementações concretas dos motores são fornecidas pelo provedor de segurança e obtidas via o método `getInstance()` que utiliza o padrão de projeto *factory*, para recuperar instâncias de diferentes classes.

7 CHAVES CRIPTOGRÁFICAS

Em criptografia, a chave é um número secreto que é parte da entrada de um algoritmo e determina a saída de um algoritmo. Dentro da linguagem Java, uma chave pode ser gerada automaticamente, ou pode ser gerada através da escolha dos parâmetros para a criação de uma determinada chave [1].

Criptografar é o ato de transformar dados legíveis em dados ilegíveis, para a garantia do sigilo das informações [1].

Uma chave criptográfica funciona da mesma forma que uma chave convencional. Para proteger o patrimônio pessoal, você instala uma fechadura na porta e esta é operada através de uma chave. A chave é responsável pelo acesso ou bloqueio de uma porta.

Em um algoritmo de criptografia, a chave é um mecanismo utilizado para proteger o conteúdo de um arquivo. Fazendo uma analogia, a fechadura representa o algoritmo de criptografia e a chave é utilizada para operar sobre o algoritmo. Este realiza sua lógica utilizando a chave a chave para realizar o processo de criptografia.

O mecanismo de chaves foi criado para evitar que um texto simples fosse decifrado através do entendimento do algoritmo de criptografia. É mais fácil guardar uma chave do que um algoritmo.

A interface que dá suporte a uma chave, independente do tipo de chave é a interface `Key`.

7.1 A Interface `Key`

O conceito de chaves é modelado pela interface `Key`:

public interface Key extends Serializable

Modela o conceito de chave simples. Devido ao fato de que as chaves devem ser transformadas em entidades e vindas de várias entidades, as chaves devem ser *serializadas*. Existem diversos algoritmos disponíveis para geração de chaves. As chaves dependem do provedor de segurança instalado na máquina virtual. Sendo assim, a primeira coisa que uma chave deve saber dizer é o algoritmo usado para a geração da chave em questão [4].

Esta interface disponibiliza três métodos:

public String getAlgorithm()

Retorna a descrição do algoritmo utilizado para gerar a chave.

public String getFormat()

Retorna o nome do formato de codificação primária da chave ou `null` caso a chave não suporte codificação.

public byte[] getEncoded()

Retorna os bytes que formaram a chave particular na sua forma codificada. Os bytes codificados são a representação externa da chave no formato binário.

As chaves são divididas em dois grupos: chaves simétricas e chaves assimétricas.

7.2 Chaves assimétricas

A criptografia de chaves assimétricas utiliza dois tipos de chaves: uma chave pública e uma privada. Em um algoritmo de criptografia assimétrica, uma mensagem cifrada com a chave pública só pode ser decifrada com a chave privada

correspondente.

Para estes dois tipos de chaves, a API Java contém estas duas interfaces adicionais:

public interface PrivateKey extends Key

public interface PublicKey extends Key

Estas interfaces não possuem métodos adicionais. Foram criadas apenas para especializar o tipo de chave que está sendo utilizada.

Os tipos de chaves assimétricas suportadas são as chaves `DSA`, `EC`, `RSA` e a `Diffie-Helman`.

Para cada tipo de chave existe três tipos de interfaces: uma sendo a interface que não discrimina se é uma chave pública ou privada, por exemplo: `DSAPublicKey`. Uma interface de chaves públicas específicas do algoritmo, por exemplo: `DSAPublicKey`. E a outra a interface de chaves privadas específicas do algoritmo, por exemplo: `DSAPrivateKey`.

Estas interfaces contêm os parâmetros utilizados para a geração de cada chave. Cada tipo de chave segue uma especificação, e esta determina os parâmetros para a criação de um determinado tipo de chave.

Existem duas classes responsáveis pela geração de chaves assimétricas: a classe `KeyPairGenerator` e a classe `KeyFactory`.

7.2.1 A classe `KeyPairGenerator`

Esta classe é responsável pela geração das chaves públicas e privadas de um determinado algoritmo.

A classe `KeyPairGenerator` é uma classe abstrata, porém é possível recuperar instâncias da classe através dos métodos:

public static KeyPairGenerator getInstance(String algorithm)

public static KeyPairGenerator getInstance(String algorithm, String provider)

Retorna a implementação do motor que gera o par de chaves de acordo com o nome do algoritmo fornecido. Para o parâmetro *algorithm*, os nomes suportados pelo provedor de segurança da SUN são “DSA”, “RSA”, “EC” e “Diffie-Hellman”.

Depois de gerado o par de chaves, os métodos abaixo podem ser utilizados:

public String getAlgorithm()

O qual retorna o nome do algoritmo que o gerador de pares de chaves implementa (ex. DSA).

public void initialize(int strength)

public abstract void initialize(int strength, SecureRandom random)

Geram as chaves de acordo com o parâmetro *strength* (força). A idéia de força se refere à quantidade de bits usados como entrada pelo motor para calcular o par de chaves. No caso dos algoritmos DSA e Diffie–Hellman, a força deve ser entre 512 e 1024 e dentre esses valores, um valor múltiplo de 64. Para o RSA, a força deve ser um valor entre 512 e 2048. Se um valor inválido for passado como argumento, uma exceção do tipo `InvalidParameterException` será disparada.

Para a geração de um par de chaves, é necessário um gerador de números aleatórios para a criação das mesmas. Este gerador é representado pela classe `SecureRandom` e utiliza um algoritmo para a geração dos números aleatórios. O único algoritmo fornecido para a criação deste gerador é o “SHA1PRNG” (que utiliza o algoritmo SHA para a geração de um PRNG – *pseudo-random number generation*). O gerador pode ser obtido pela recuperação de uma instância através do método `SecureRandom.getInstance(String algoritmo)` ou `SecureRandom.getInstance(String alg, String`

nomeProvedor).

Uma outra forma de gerar o par de chaves é passando um objeto que segue a interface `AlgorithmParameterSpec`. Este objeto contém os parâmetros para a geração das chaves.

public void initialize(AlgorithmParameterSpec params)

public void initialize(AlgorithmParameterSpec params, SecureRandom r)

A interface `AlgorithmParameterSpec` pode ser especializada de acordo com o algoritmo de criptografia. As especializações desta classe contém os parâmetros necessários a criação de determinado tipo de algoritmo da chave. Por exemplo, para um algoritmo DSA, existe a classe `DSAParameterSpec`, que pode ser especializada de acordo tipo de chave: `DSAPrivateKeySpec`, `DSAPublicKeySpec`

O par de chaves pode ser gerado através de um dos métodos abaixo:

public abstract KeyPair generateKeyPair()

public final KeyPair genKeyPair()

Gera o par de chaves de acordo com a inicialização dos parâmetros especificadas pelo método `initialize()`.

Estes dois últimos métodos podem ser chamados diversas vezes, de modo que cada nova chamada gera um novo par de chaves. O método `genKeyPair()` apenas faz a chamada do método `generateKeyPair()`.

O par de chaves gerado é representado pela classe `KeyPair`:

7.2.1.1 A classe *KeyPair*

Modela um objeto que contém uma chave pública e uma chave privada. A

classe `KeyPair` contém apenas dois métodos:

```
public PublicKey getPublic( )
```

```
public PrivateKey getPrivate( )
```

Retorna a chave desejada do par de chaves.

Um objeto da classe pode ser instanciado através do construtor:

```
public KeyPair(PublicKey pub, PrivateKey priv)
```

Este construtor cria um objeto que contém um par de chaves. É obrigatório que os parâmetros deste método não possuam valores nulos.

Abaixo um exemplo utilizando a classe `KeyPairGenerator` para a criação de chaves assimétricas. Este exemplo utiliza o algoritmo DAS para a geração das chaves, um gerador de números aleatórios do provedor de segurança SUN e um tamanho de 1024 bits.

```
KeyPairGenerator generator = KeyPairGenerator.getInstance("DSA",
"SUN");
SecureRandom random = SecureRandom.getInstance("SHA1PRNG", "SUN");
generator.initialize(1024, random);
KeyPair par = generator.generateKeyPair();
DSAPrivateKey chavePrivada = (DSAPrivateKey) par.getPrivate();
DSAPublicKey chavePublica = (DSAPublicKey) par.getPublic();
```

7.2.2 A classe *KeyFactory*

Outra maneira de gerar chaves é através da classe `KeyFactory`. Esta classe é muito útil para a importação de chaves, ou a tradução de chaves de provedores desconhecidos para um provedor confiável utilizado pela aplicação.

Um objeto da classe pode ser obtido através de um dos dois métodos abaixo:

public static final KeyFactory getInstance(String alg)

public static final KeyFactory getInstance(String alg, String provider)

Gera um objeto `KeyFactory` de acordo com o algoritmo especificado e, se fornecido, de acordo com o provedor de segurança informado. Caso não seja encontrado o algoritmo ou o provedor de segurança, será disparada exceções do tipo `NoSuchAlgorithmException` e `NoSuchProviderException` respectivamente.

Outros métodos da classe da classe `Provider`:

public final Provider getProvider()

Retorna o provedor de segurança que implementa o objeto `KeyFactory`.

public final PublicKey generatePublic(KeySpec ks)

public final PrivateKey generatePrivate(KeySpec ks)

Gerador de chave pública ou privada de acordo com a especificação da chave fornecida. Se a chave não puder ser criada, será disparada uma exceção do tipo `InvalidKeySpecException`.

public final KeySpec getKeySpec(Key key, Class keySpec)

Retorna uma especificação da chave fornecida. O parâmetro `keySpec` pode ser, por exemplo, `DSAPublicKeySpec.class` para informar que o retorno deve ser uma instância da classe `DSAPublicKeySpec` que possui a especificação da chave pública do tipo DSA.

public final Key translateKey(Key key)

Traduz uma chave, em que um provedor pode ser não confiável ou desconhecido, em um objeto do tipo `Key` correspondente ao provedor do objeto `KeyFactory`.

public final String getAlgorithm()

Retorna o algoritmo que o objeto `KeyFactory` suporta.

A seguir será demonstrado um método que recebe os parâmetros das chaves DSA pública e privada uma chave e, com estes parâmetros, realiza a criação do par de chaves:

```
private void importandoChavesDSAUtilizandoKeyFactory(byte[]
pPrivateKey, byte[] pPublicKey, byte[] pP, byte[] pQ, byte[] pG) throws
NoSuchAlgorithmException, InvalidKeySpecException {

    BigInteger privateKey = new BigInteger(pPrivateKey);
    BigInteger publicKey = new BigInteger(pPublicKey);
    BigInteger p = new BigInteger(pP);
    BigInteger q = new BigInteger(pQ);
    BigInteger g = new BigInteger(pG);

    KeyFactory kf = KeyFactory.getInstance("DSA");
    DSAPrivateKeySpec specPrivateKey = new
DSAPrivateKeySpec(privateKey, p, q, g);
    DSAPublicKeySpec specPublicKey = new DSAPublicKeySpec(publicKey, p,
q, g);
    DSAPrivateKey dsaPrivateKey = (DSAPrivateKey)
kf.generatePrivate(specPrivateKey);
    DSAPublicKey dsaPublicKey = (DSAPublicKey)
kf.generatePublic(specPublicKey);
}
```

7.3 Chaves simétricas

É o tipo de chaves mais simples, onde o emissor e o receptor fazem uso da mesma chave para cifrar e decifrar uma informação. Chaves simétricas são definidas segundo a interface abaixo:

```
public interface SecretKey extends Key
```

Diferente das chaves assimétricas, chaves simétricas não possuem suas próprias interfaces.

Existem duas classes responsáveis pela geração de chaves simétricas: A classe `KeyGenerator` e a classe `SecretKeyFactory`.

7.3.1 A classe *KeyGenerator*

Esta classe difere da classe `KeyPairGenerator` apenas pelo fato de

gerar chaves simétricas.

public class KeyGenerator

Gera instâncias de chaves que usam algoritmos de chaves simétricas.

Uma instância da classe é obtida através dos seguintes métodos:

public static final KeyGenerator getInstance(String algorithm)

public static final KeyGenerator getInstance(String algorithm, String provider)

Retorna uma instância geradora de chaves de acordo com o algoritmo especificado e, se necessário, de acordo com o provedor de segurança fornecido. Caso o algoritmo não exista na lista de provedores de segurança, será disparada a exceção `NoSuchAlgorithmException`. Um provedor não encontrado pelo algoritmo implica a exceção `NoSuchProviderException`.

Uma vez que o objeto tenha sido obtido, o gerador deve ser inicializado utilizando um dos seguintes métodos:

public final void init(SecureRandom sr)

public final void init(AlgorithmParameterSpec aps)

public final void init(AlgorithmParameterSpec aps, SecureRandom sr)

public final void init(int strength)

public final void init(int strength, SecureRandom sr)

Da mesma forma que o gerador de chaves assimétricas, o gerador de chaves simétricas necessita de um gerador de números aleatórios, que caso não seja fornecido, uma instância default será utilizada. Além disso, alguns geradores de chaves simétricas aceitam parâmetros específicos para o algoritmo utilizado para gerar a chave.

A geração de uma chave ocorre da seguinte forma:

public final SecretKey generateKey()

Gera uma chave secreta. Estas podem ser produzidas por diversas vezes, apenas repetindo a chamada do método que retornará uma nova chave.

Dentre os algoritmos aceitos para a criação de chaves simétricas estão: “AES”, “ARCFOUR”, “Blowfish”, “DES”, “DESede”, “HmacMD5”, “HmacSHA1”, “HmacSHA256”, “HmacSHA384”, “HmacSHA512” e “RC2”.

7.3.1.1 Exemplo utilizando KeyGenerator

```
KeyGenerator kg = KeyGenerator.getInstance("Blowfish");
SecretKey chaveSecreta = kg.generateKey();
String conteudoDaChave = chaveSecreta.getEncoded();
```

7.3.2 A classe SecretKeyFactory

A classe `SecretKeyFactory` é semelhante a classe `KeyFactory`, porém é responsável pela geração de chaves simétricas. Possui os mesmos métodos para recuperar uma instância e difere apenas do método para geração de chaves.

Os algoritmos para recuperar uma instância da classe são: “AES”, “ARCFOUR”, “DES”, “DESede”, “PBKDF2WithHmacSHA1”, “PBEWith<digest>And<encryption>”, “PBEWith<prf>And<encryption>”.

Os dois últimos são algoritmos baseados em senhas (*password-based encryption*). A documentação da SUN afirma que o parâmetro <digest> pode ser trocado por um algoritmo de resumo de mensagem que será visto adiante, por exemplo, MD5. O parâmetro <encryption> pode ser trocado por um algoritmo de criptografia simétrica, por exemplo, DES. E o parâmetro <prf>, trocado por um algoritmo de funções pseudo-aleatórias como a HMacSHA1.

Para exemplificar, dois exemplos de possíveis algoritmos serão mostrados, o primeiro utilizando o parâmetro <digest> e o segundo utilizando <prf>:

“PBEWithMD5andDES” e “PBEWithHMacSHA1andDESede”.

O método para gerar chaves simétricas é o seguinte:

public final SecretKey generateSecret(KeySpec ks)

Gera uma chave simétrica de acordo com a especificação da chave fornecida. Caso a especificação seja inválida, uma exceção do tipo `InvalidKeySpecException` será disparada.

A seguir, um exemplo demonstrando a importação de uma chave DES a partir do conteúdo da mesma e utilizando a classe `SecretKeyFactory`:

```
public void importandoChaveDESUtilizandoSecretKeyFactory(byte[]
conteudoDaChave) throws Throwable {

    DESKeySpec desKeySpec = new DESKeySpec(conteudoDaChave);
    SecretKeyFactory keyFactory = SecretKeyFactory.getInstance("DES");
    SecretKey secretKey = keyFactory.generateSecret(desKeySpec);
    byte[] conteudoDaChave = secretKey.getEncoded();
}
```

8 RESUMOS DE MENSAGEM

Os resumos de mensagem (*Message Digest*) ou funções *hash* realizam a transformação de uma mensagem em um representante: o resumo da mensagem. Este é a uma seqüência de caracteres gerado pelo algoritmo de criptografia especificado. Cada mensagem utilizada como entrada deverá produzir um MD (*Message Digest*) específico.

Este mecanismo é utilizado para verificar a integridade de uma mensagem. Por exemplo: suponha que uma mensagem seja enviada ao destinatário. Esta mensagem pode sofrer alterações no meio do caminho e chegar uma mensagem diferente. Imagine que a mensagem contenha um pedido de dez quantidades de um determinado produto feito sob medida. A mensagem sofreu alteração no meio da transmissão e a empresa recebe o pedido com a quantidade de cem produtos. Isto acarretará em prejuízo a empresa.

Para garantir a integridade é gerado um resumo da mensagem, e este resumo é enviado juntamente com a mensagem. A garantia de integridade se dá pela geração do resumo da mensagem recebida e é verificado se o resumo da mensagem enviada corresponde ao resumo da mensagem gerada pelo destinatário.

Estas funções são denominadas *one-way hash function* (funções hash de um único sentido) porque, uma vez cifrada, uma mensagem não pode ser decifrada.

A classe responsável pela geração de chaves é denominada `MessageDigest`.

O algoritmo de resumos garante a integridade, mas não garante a autenticidade do emissor da mensagem. Uma variação do algoritmo de resumos de mensagem é o *MAC* (*Message Authentication Code* – Código de Autenticação de Mensagem). O processo de geração deste código é feito da seguinte maneira: o

algoritmo utiliza o resumo da mensagem concatenado com a chave pública do emissor da mensagem. A combinação da chave e do resumo submetido a um algoritmo *MAC*, gera o código de autenticação de mensagem.

Para verificação do *MAC*, o destinatário deverá ter posse da chave pública do emissor e aplicar o algoritmo de *MAC* na mensagem. O *MAC* garante que a mensagem enviada é do emissor em questão e que a mensagem não sofreu alterações durante o envio.

A classe responsável pela geração de um *MAC* é a classe `MAC`.

Serão demonstradas as classes `MessageDigest` e `MAC`.

8.1 A classe *MessageDigest*

A classe `MessageDigest` fornece aos aplicativos as funcionalidades de algoritmos resumos de mensagem.

Os algoritmos suportados para criação de um resumo de mensagem são: “MD2”, “MD5”, “SHA-1”, “SHA-256”, “SHA-384” e “SHA-512”.

Um objeto `MessageDigest` é obtido através de um dos métodos abaixo:

```
public static MessageDigest getInstance(String algorithm)
```

```
public static MessageDigest getInstance(String algorithm, String provider)
```

Cria um objeto *MessageDigest* que implementa o algoritmo e suportado pelo provedor de segurança fornecido, caso este parâmetro seja fornecido.

```
public void update(byte input)
```

```
public void update(byte[ ] input)
```

```
public void update(byte[ ] input, int offset, int length)
```

Atualiza os dados que serão utilizados para gerar a função *hash*. A primeira função adiciona um único byte para a função *hash*, a segunda adiciona o

array de bytes inteiro, e a terceira adiciona apenas o subconjunto de bytes do array fornecido.

public byte[] digest()

public byte[] digest(byte[] input)

public int digest(byte[] output, int offset, int len)

Gera a função *hash* retornando um array de bytes que é o resultado do processamento do algoritmo fornecido à instância da classe. A segunda função permite adicionar um array de bytes aos dados que serão utilizados pela função *hash*, antes de completar a operação. A terceira função adiciona o subconjunto de bytes contidos no intervalo especificado pelos parâmetros, antes de gerar o *hash*. Após o uso de uma destas funções, o objeto `MessageDigest` é automaticamente submetido a função `reset()` que permite ao objeto gerar uma nova função *hash*.

public static boolean isEqual(byte digestA[], byte digestB[])

Verifica a igualdade de duas funções *hash*.

public void reset()

Inicializa o objeto para uso posterior.

public final String getAlgorithm()

Retorna uma string que identifica o algoritmo utilizado para gerar a função *hash*.

public Object clone() throws CloneNotSupportedException

Retorna uma cópia do objeto, se o objeto puder ser clonado. Este método é utilizado quando se quer preservar o estado do objeto original, já que após a chamada do método `digest()`, o objeto perde suas propriedades originais. Se o objeto não puder ser clonado, a exceção `CloneNotSupportedException` será disparada.

public final int getDigestLength()

Retorna o tamanho do array gerado pela função *hash*.

Abaixo será demonstrado um exemplo para gerar um resumo de mensagem sobre um texto qualquer e

```
MessageDigest md = MessageDigest.getInstance("SHA-1");    String
textoQualquer = "texto qualquer";
byte[] textoResumido = md.digest(textoQualquer.getBytes());
```

8.2 A classe MAC

Semelhante a classe `MessageDigest`, um Código de Autenticação de Mensagem fornece um meio de verificar a integridade da informação transmitida por um meio inseguro, porém inclui uma chave simétrica para o cálculo da função *hash*. Somente quem dispor da chave estará apto a examinar o conteúdo da mensagem[1].

Os algoritmos suportados para criação de um objeto `MAC` são: “HmacMD5”, “HmacSHA1”, “HmacSHA256”, “HmacSHA384”, “HmacSHA512” e `PBEwith<mac>` sendo que `<mac>` é qualquer um dos nomes algoritmos Hmac citados anteriormente, por exemplo: “PBEwithHmacMD5”, “PBEwithHmacSHA1”.

A criação de um objeto `MAC` é feita através de um dos métodos abaixo:

public static MAC getInstance(String algorithm)

public static MAC getInstance(String algorithm, String provider)

Retorna um objeto `MAC` segundo o algoritmo especificado e, opcionalmente, implementado pelo provedor especificado.

O objeto `Mac` deve ser inicializado com uma chave simétrica e opcionalmente com um conjunto de parâmetros, dependendo do algoritmo utilizado.

public void init(Key key)

public void init(Key key, AlgorithmParameterSpec params)

Após inicializado, o objeto deverá ser atualizado com a informação que será resumida através de um dos métodos abaixo::

public final void update(byte b)

public final void update(byte[] b)

public final void update(byte b[], int offset, int length)

Estes métodos adicionam a informação que será processada para o cálculo de um MAC. A informação pode ser um byte, um array de bytes, ou parte de um array de bytes fornecido através do intervalo *offset* que é aponta para o início da informação que será extraída e *length* que é o tamanho da informação extraída.

Os métodos para a criação de um *MAC* são:

public byte[] doFinal()

Processa e retorna o valor *MAC* em um array de bytes.

public byte[] doFinal(byte[] input)

Atualiza a informação, concatenando o array de bytes *input* à informação que foi fornecida pelo método `update()` e retorna o valor *MAC* em um array de bytes.

public void doFinal(byte[] output, int offset)

Escreve o valor *MAC* no array de bytes *output*, iniciando da posição especificada pelo parâmetro *offset*.

A seguir, um exemplo para gerar um *MAC* utilizando um texto qualquer como entrada e o algoritmo *HmacSHA1*:

```
Mac mac = Mac.getInstance("HmacSHA1");
SecretKey chave = KeyGenerator.getInstance("HmacSHA1").generateKey();
mac.init(chave);
```

```
String textoQualquer = "Este é o texto que será utilizado para gerar  
o MAC"
```

```
textoResumido = mac.doFinal(textoQualquer.getBytes());
```

9. CRIPTOGRAFIA BASEADA EM CIFRADORES

Cifrar é o processo de transformar dados ou informações legíveis em dados incompreensíveis.

Existem dois tipos de criptografia: criptografia de chaves simétricas e a criptografia de chaves públicas.

A criptografia de chaves públicas utiliza o par de chaves assimétricas para a criptografia. O processo de cifrar é feito com a chave pública e a decifragem é feito com a chave privada correspondente.

Na criptografia simétrica, a mesma chave é utilizada tanto para cifrar quando para decifrar dados.

Para realizar este processo, utiliza-se um algoritmo dependendo do tipo de criptografia que será aplicada.

Em Java, a classe responsável por este processo é a classe `Cipher` e será mostrada em detalhes na seção seguinte.

9.1 Classe Cipher

A classe `Cipher` provê as funcionalidades usadas no processo de cifrar e decifrar dados.

Um objeto da classe `Cipher` é criado como a maioria das outras classes motoras, utilizando o método `getInstance()`. É um método estático que retorna uma instância da classe e é definido da seguinte forma:

```
public static Cipher getInstance(String transformation)  
public static Cipher getInstance(String transformation, String provider)
```

Este método requer o parâmetro obrigatório *transformation* e, se

necessário pode-se especificar o provedor de segurança utilizado para a criação do objeto.

O parâmetro *transformation* é uma string que descreve a operação, ou conjunto de operações para a produção da instância da classe. Este parâmetro obrigatoriamente inclui o nome do algoritmo (como por exemplo DES), e pode ser seguido por um modo de operação de cifragem (*mode*) e um esquema de preenchimento (*padding*).

Os modos de cifragem e os esquemas de preenchimento estão presentes na classe `Cipher` porque ela implementa o que é conhecido como criptografia de bloco. Isto significa operar nos dados de um bloco (por ex. de 8 bytes) de uma única vez. Os esquemas de preenchimento são necessários para assegurar que a extensão dos dados seja um número inteiro de blocos, ou seja, se em um bloco não estiver o número necessário de bytes para formar o bloco, o bloco será preenchido de acordo com o esquema de preenchimento fornecido [3].

Os modos de cifragem são fornecidos para alterar os dados codificados para tornar mais difícil a quebra da codificação. Por exemplo, se os dados a serem codificados seguem um padrão similar, como nomes repetidos, este padrão poderá ser utilizado para a decodificação dos dados. Os diferentes modos de decodificação ajudam a evitar estes tipos de ataques. Alguns modos requerem um vetor de inicialização que é um bloco que inicializa o processo de cifragem para o primeiro bloco. Este vetor de inicialização não precisa ser secreto, mas é importante que o mesmo vetor não seja utilizado com a mesma chave [3].

Alguns dos modos especificados para o ambiente de segurança em Java são:

- **ECB** (*Electronic CodeBook*): é o modo mais simples e não requer vetor de inicialização. Ele obtém um bloco de dados simples (8 bytes, que é o default) e codifica todo o bloco de uma única vez. Nenhuma tentativa é feita para ocultar os

padrões nos dados e os blocos podem ser reorganizados sem afetar a decodificação (embora o texto resultante fique desordenado). Em virtude destas limitações, o ECB é recomendado apenas para dados binários [3].

O modo ECB pode operar somente em blocos de dados cheios, portanto geralmente é utilizado com algum esquema de preenchimento [3].

- **CBC** (*Cipher-block Chaining*): Este é o modo de encadeamento de blocos cifrados. Neste modo, a entrada de um bloco é usada para modificar a codificação do próximo bloco de dados. Isto ajuda a ocultar padrões e, desse modo, torna-se adequado para os dados que representam textos [3].

No modo CBC é feita uma operação XOR entre cada novo bloco de texto normal com o bloco cifrado obtido na etapa anterior. Por este motivo é necessário um vetor de inicialização para operar no primeiro bloco de texto puro. O modo CBC opera em blocos de dados cheios, precisando, geralmente, de algum modo de preenchimento e requer um vetor de inicialização [3].

- **CFB** (*Cipher FeedBack*): Similar ao CBC, porém o modo CFB pode começar a codificação com uma quantidade menor de dados. Portanto, este modo é adequado para a codificação de texto, especialmente quando esse texto precisar ser processado com um caractere por vez. Por default, este modo opera em blocos de 8 bytes, mas pode ser anexado um número de bits (ex: CFB8) indicando o número de bits no qual o modo deverá operar. Este número deverá ser múltiplo de 8. [3]

O CFB requer que os dados sejam preenchidos de forma que preencham um bloco completo. Já que esse tamanho pode variar, o esquema de preenchimento usado por ele também pode variar. Para CFB8, nenhum esquema de preenchimento será necessário, já que os dados serão sempre alimentados em um número inteiro de bytes. O modo CFB requer um vetor de inicialização [3].

- **OFB** (*Output FeedBack*): É usado com mais frequência quando existe a

possibilidade dos bits de dados codificados poderem ser alterados em trânsito (ex.: por um modem com interferência). Embora um erro de 1 bit possa fazer com que todo um bloco seja perdidos nos outros modos, ele acarreta apenas na perda de um bit neste modo. Por default, o modo OFB opera em blocos de 8 bytes, mas pode ser anexado um número de bits múltiplo de 8 junto a OFB (ex.: OFB8), indicando o número de bits no qual o modo deverá operar [3].

Da mesma forma que o CFB, o OFB requer que os dados sejam preenchidos de forma que preencham um bloco inteiro e o esquema de preenchimento pode variar de acordo com o tamanho do bloco.

O OFB também requer um vetor de inicialização.

- **PCBC** (*Propagating Cipher-Block Chaining*): Este é o modo de propagação de blocos encadeados. É muito utilizado em sistemas conhecidos como *Kerberos* (protocolo de transporte de rede), que permite comunicações individuais seguras, em uma rede insegura.

Este modo necessita de um vetor de inicialização e requer que a entrada seja preenchida com um múltiplo de 8 bytes.

Os esquemas de preenchimento especificados são:

- **PKCS5Padding**: Este esquema de preenchimento assegura que os blocos de entrada sejam preenchidos com um valor múltiplo de 8 bytes.

- **NoPadding**: Neste caso, nenhum preenchimento será usado. Utilizando este esquema de preenchimento.

O parâmetro *transformation* é especificado da seguinte forma:

algoritmo/modo/padding ("DES/CBC/PKCS5Padding") ou
algoritmo ("DES")

Caso nenhum esquema de modo ou preenchimento seja especificado será utilizado o valor default para o modo e esquema de preenchimento para cada instância da classe.

Quando o modo é especificado, pode-se, opcionalmente especificar o número de bits processados por vez, concatenando ao parâmetro *mode* o número de bits, por exemplo:

```
"DES/OFB32/NoPadding"
```

Caso nenhum valor seja especificado será utilizado o valor default para cada modo.

Uma instância da classe obtido pelo método `getInstance()` deve ser inicializado para um dos quatro modos de operação (*opmode*):

- ENCRYPT_MODE: cifrar dados
- DECRYPT_MODE: decifrar dados
- WRAP_MODE: encapsular de uma chave em bytes para o transporte seguro
- UNWRAP_MODE: desencapsula a chave, uma vez que esta tenha sido encapsulada anteriormente.

O parâmetro *opmode* é utilizado em todos os métodos para a inicialização do objeto da classe. Outros parâmetros são aceitos como uma chave (*key*), um certificado (*certificate*) contendo uma chave, parâmetros de algoritmos (*params*) e um objeto *random* para geração de números pseudo-aleatórios.

Para inicializar um objeto, um dos métodos `init()` é utilizado. As variações do método `init` estão mostradas a seguir:

```
public void init(int opmode, Key key);  
public void init(int opmode, Certificate certificate);  
public void init(int opmode, Key key, SecureRandom random);
```

```

public void init(int opmode, Certificate certificate, SecureRandom random)
public void init(int opmode, Key key, AlgorithmParameterSpec params);
public void init(int op, Key k, AlgorithmParameterSpec p, SecureRandom r);
public void init(int opmode, Key key, AlgorithmParameters params);
public void init(int opmode, Key k, AlgorithmParameters p, SecureRandom r);

```

No caso de um objeto *Cipher* ser inicializado para cifrar dados e nenhum parâmetro for fornecido para o método `init()`, será utilizado parâmetros aleatórios ou parâmetros default do provedor de segurança especificado. No entanto, se um objeto for inicializado para decifrar dados e nenhum parâmetro for informado, a exceção `InvalidKeyException` ou `InvalidAlgorithmParameterException` será disparada, dependendo do método `init()` utilizado. E para a criação de uma instância para decifrar dados, os mesmos parâmetros do objeto, utilizados para cifrar, devem ser utilizados.

Tendo visto como funciona a inicialização do objeto, será demonstrado como cifrar/decifrar dados, o encapsulamento de chaves e a classe `SealedObject` que utiliza um cifrador para codificar o objeto.

9.1.1 Cifrar e Decifrar dados

Os processos de cifrar e decifrar dados podem ser realizados em uma única operação ou em mais de uma operação. Para cifrar ou decifrar os dados em uma única operação, basta utilizar um dos métodos `doFinal()` abaixo:

```

public byte[ ] doFinal(byte[ ] input);
public byte[ ] doFinal(byte[ ] input, int inputOffset, int inputLen);
public int doFinal(byte[ ] input, int inputOffset, int inputLen, byte[ ] output);
public int doFinal(byte[ ] input, int inputOffset, int inputLen, byte[ ] output, int
outputOffset)

```

Estes métodos irão cifrar ou decifrar os dados de acordo com os parâmetros fornecidos ao método e de acordo com o modo de inicialização do objeto.

Porém, caso exista mais informações a serem cifradas ou decifradas, um dos métodos `update()` deve ser chamado antes da chamada do método `doFinal()`.

```
public byte[ ] update(byte[ ] input);
public byte[ ] update(byte[ ] input, int inputOffset, int inputLen);
public int update(byte[ ] input, int inputOffset, int inputLen, byte[ ] output);
public int update(byte[ ] input, int inputOffset, int inputLen, byte[ ] output, int
outputOffset)
```

Após a atualização dos dados, utilizando um dos métodos `update()` acima, deverá ser utilizado caso seja necessário um acréscimo de informação para a operação de cifrar ou decifrar. Caso não haja informações adicionais, deve-se utilizar um dos métodos `doFinal()` já mostrados anteriormente.

Abaixo um exemplo para cifrar e decifrar um texto qualquer utilizando o algoritmo de criptografia RSA. Neste exemplo

```
public void cifrandoDecifrando() throws NoSuchAlgorithmException,
    NoSuchPaddingException, InvalidKeyException,
    IllegalBlockSizeException, BadPaddingException, IOException,
    InvalidAlgorithmParameterException{

    Cipher cifrar = Cipher.getInstance("RSA");
    KeyPair parDeChaves =
    KeyPairGenerator.getInstance("RSA").generateKeyPair();
    PublicKey chavePublicaRSA = parDeChaves.getPublic();
    PrivateKey chavePrivadaRSA = parDeChaves.getPrivate();
    cifrar.init(Cipher.ENCRYPT_MODE, chavePublicaRSA);
    String textoQualquer = "outro texto qualquer";
    byte[] textoCifrado = cifrar.doFinal(textoQualquer.getBytes());
```

```

String textoCifradoString = new
    BASE64Encoder().encode(textoCifrado);
System.out.println("cifrado = " + textoCifradoString);
Cipher decifrar = Cipher.getInstance("RSA");
decifrar.init(Cipher.DECRYPT_MODE, chavePrivadaRSA);
System.out.println("decifrado = " + new
String(decifrar.doFinal(textoCifrado)));
}

```

9.1.2 Encapsulamento de chaves

O propósito de encapsular ou proteger uma chave é garantir uma transferência segura de uma chave de um lugar para o outro.

Para encapsular uma chave deve-se, primeiramente, inicializar um objeto `Cipher` para o modo `WRAP_MODE`. Após a inicialização, o método `wrap()` deve ser invocado.

```
public final byte[] wrap (Key key);
```

O método para desencapsular uma chave é o `unwrap()`. Para este método, além da chave encapsulada em um array de bytes, deve ser fornecido o nome do algoritmo utilizado para a criação da chave através do método:

```
public String getAlgorithm( );
```

Além do nome do algoritmo, o tipo da chave deve ser fornecido através de um dos atributos estáticos da classe (`Cipher.SECRET_KEY`, `Cipher.PRIVATE_KEY`, `Cipher.PUBLIC_KEY`).

Tendo os atributos necessários, o método `unwrap()` poderá ser chamado:

```
public final Key unwrap (byte[] wrappedKey, String wrappedKeyAlgorithm, int
```

wrappedKeyType)

Este método, desencapsulará a chave caso os parâmetros do método `unwrap()` sejam os mesmos utilizados para encapsular a chave.

9.1.2.1 Exemplo de encapsulamento de chaves

```
public void encapsulamentoDeChaves() throws
NoSuchAlgorithmException, NoSuchPaddingException, InvalidKeyException,
IllegalBlockSizeException{
    //encapsular
    Cipher cifrar = Cipher.getInstance("AES");
    KeyGenerator kg = KeyGenerator.getInstance("AES");
    SecretKey chaveAES = kg.generateKey();
    cifrar.init(Cipher.WRAP_MODE, chaveAES);
    byte[] chaveEncapsulada = cifrar.wrap(chaveAES);

    //desencapsular
    cifrar.init(Cipher.UNWRAP_MODE, chaveAES);
    SecretKey chaveDesencapsulada = (SecretKey);
    cifrar.unwrap(chaveEncapsulada, cifrar.getAlgorithm(),
Cipher.SECRET_KEY);
}
```

9.1.3 A Classe SealedObject

Esta classe é muito parecida com a classe `SignedObject` com a diferença de que ao invés de o objeto serializado ser assinado, ele é codificado pelo objeto `Cipher` que foi fornecido ao construtor da classe.

Dado um objeto quem implemente a interface `Serializable`, pode-se instanciar um `SealedObject` que encapsula o objeto original, na forma serializada, e cifra seu conteúdo, utilizando um algoritmo de criptografia para proteger sua confidencialidade. O conteúdo pode ser decifrado utilizando o algoritmo de criptografia correspondente e a chave correta.

O construtor da classe é definido da seguinte forma:

public SealedObject (Serializable object, Cipher c)

Os métodos da classe são:

public String getAlgorithm()

Retorna o algoritmo usado para codificar o objeto.

public Object getObject(Cipher c)**public Object getObject(Key key)****public Object getObject(Key key, String Provider)**

Retorna o objeto original passando como parâmetro o cifrador ou a chave utilizada para codificar o objeto. Juntamente com a chave pode ser passado o nome do provedor de segurança utilizado para a criação do cifrador.

9.1.3.1 Exemplo utilizando SealedObject

```
public void codificandoObjeto() throws InvalidKeyException,
    IllegalBlockSizeException, BadPaddingException, IOException,
    ClassNotFoundException, NoSuchAlgorithmException,
    NoSuchPaddingException{

    Cipher c = Cipher.getInstance("AES");
    KeyGenerator kg = KeyGenerator.getInstance("AES");
    SecretKey chaveAES = kg.generateKey();
    c.init(Cipher.ENCRYPT_MODE, chaveAES);
    Integer senha = new Integer(123456);
    SealedObject so = new SealedObject(senha, c);
    c.init(Cipher.DECRYPT_MODE, chaveAES);
    Integer senhaDecodificada = (Integer)so.getObject(c);
    System.out.println(senhaDecodificada);

}
```

10 ASSINATURAS DIGITAIS

A assinatura digital é um tipo de criptografia assimétrica. É uma forma de representar digitalmente uma assinatura a mão em um documento e garantir a autenticidade de documentos digitais.

O processo de geração de uma assinatura digital pode ser realizado de duas formas:

- A primeira forma é aplicar um algoritmo de assinaturas digitais em uma mensagem utilizando a chave privada do proprietário do documento. Esta é uma forma lenta, pois o algoritmo utiliza a mensagem inteira para realizar a assinatura e o tempo da criação da assinatura vai variar de acordo com o tamanho do documento.
- A segunda maneira é criar um resumo da mensagem e aplicar o algoritmo de assinaturas utilizando a chave privada do proprietário do documento. Isto torna o processo mais rápido, pois o processo de geração de um resumo de mensagem é rápido.

Por questões de desempenho, o Java, através da classe `Signature`, utiliza o segundo método para criar assinaturas digitais.

Em Java, existe a possibilidade de assinar digitalmente um objeto. A classe responsável por este processo é a classe `SignedObject`.

Abaixo serão demonstradas as duas classes de assinaturas

10.1 A classe `Signature`

A classe `Signature` fornece um motor para a criação e verificação de assinaturas digitais.

Uma instância da classe é obtida utilizando um dos dois métodos abaixo:

```
public static Signature getInstance(String algorithm)
```

```
public static Signature getInstance(String algorithm, String provider)
```

Cria um objeto `Signature` que implementa o algoritmo fornecido, opcionalmente, utilizando o nome do provedor de segurança. Se uma implementação do algoritmo não for encontrada, a exceção `NoSuchAlgorithmException` será disparada. Da mesma forma, se o provedor de segurança não for encontrado a exceção `NoSuchProviderException` será disparada.

Uma vez criado o objeto `Signature`, ele deverá ser inicializado para verificar uma assinatura ou para realizar a assinatura:

```
public void final initVerify(PublicKey publicKey)
```

Inicializa o objeto `Signature`. Um objeto de assinatura deve ser inicializado antes da utilização. A inicialização prepara o objeto para verificar uma assinatura. Caso o tipo de chave pública não for correta para o algoritmo, a exceção `InvalidKeyException` será disparada.

```
public final void initSign(PrivateKey privateKey)
```

Inicializa o objeto para criar uma assinatura. Assim como para verificar uma assinatura o objeto deve ser inicializado, esta regra serve também para criar assinaturas. Da mesma forma, caso o tipo da chave privada não for correta para o algoritmo a exceção `InvalidKeyException` será disparada.

Após o objeto ser inicializado para uma das duas funções citadas, o objeto deverá ter a informação adicionada ao objeto, tanto para verificar quanto para assinar.

public final void update(byte b)

public final void update(byte[] b)

public final void update(byte b[], int offset, int length)

Adiciona a informação do parâmetro aos dados que serão assinados ou verificados. Se o objeto não for inicializado anteriormente, a exceção `SignatureException` será disparada.

O método `sign()` cria uma assinatura digital:

public final byte[] sign()

Retorna os bytes da assinatura dos dados adicionados pelo método `update()`. Caso o objeto não tenha sido inicializado, a exceção `SignatureException` será disparada.

public final int sign(byte[] outbuf, int offset, int len)

Finaliza a operação de assinatura e armazena os bytes assinados parâmetro `outbuf`. Caso o objeto não tenha sido inicializado, a exceção `SignatureException` será disparada.

O método `verify()` verifica a assinatura do documento. O conteúdo do documento é fornecido em forma de um array de bytes:

public final boolean verify(byte[] signature)

Verifica a validade da assinatura fornecida, assumindo que o objeto tenha sido inicializado para a verificação. Caso o objeto não tenha sido inicializado, a exceção `SignatureException` será disparada.

Outros métodos da classe são:

public final void setParameter(AlgorithmParameterSpec param)

Inicializa o motor de assinatura de acordo com os parâmetros especificados pelo algoritmo utilizado.

public final Object getParameters()

Retorna os parâmetros utilizados pelo objeto *Signature*.

public final Provider getProvider()

Retorna o provedor de segurança utilizado pelo objeto *Signature*.

No exemplo abaixo será demonstrado como assinar um texto qualquer utilizando um par de chaves DSA e o algoritmo de assinatura SHA1withDSA.

```
public void assinarEverificarAssinatura() throws
    NoSuchAlgorithmException, SignatureException,
    CertificateEncodingException, InvalidKeyException, IOException,
    ClassNotFoundException, NoSuchProviderException{

    //assinatura do documento
    Signature assinador = Signature.getInstance("SHA1withDSA");

    KeyPairGenerator generator = KeyPairGenerator.getInstance("DSA",
        "SUN");
    SecureRandom random =
        SecureRandom.getInstance("SHA1PRNG", "SUN");
    generator.initialize(1024, random);
    KeyPair par = generator.generateKeyPair();
    DSAPrivateKey chavePrivada = (DSAPrivateKey) par.getPrivate();
    DSAPublicKey chavePublica = (DSAPublicKey) par.getPublic();

    String texto = "Este texto será assinado digitalmente";

    assinador.initSign(chavePrivada);
    assinador.update(texto.getBytes());

    byte[] assinaturaDigital = assinador.sign();

    //verificação da assinatura
```

```

    assinador.initVerify(chavePublica);
    assinador.update(texto.getBytes());

    System.out.println(assinador.verify(assinaturaDigital));
}

```

10.2 A classe *SignedObject*

Uma instância da classe `SignedObject` encapsula um objeto serializável e sua assinatura. O objeto encapsulado precisa ser serializável conforme o construtor abaixo:

public SignedObject(Serializable o, PrivateKey pk, Signature engine)

Constrói um `SignedObject` através de um objeto serializável qualquer. O objeto é assinado de acordo com a chave privada `pk` e utiliza o motor de assinatura passado como parâmetro do construtor. Dispara as exceções `IOException` se algum erro ocorrer durante a serialização, `InvalidKeyException` se a chave for inválida e `SigInatureException` caso a assinatura falhar.

Abaixo alguns métodos usados para manipular um `SignedObject`:

public String getAlgorithm()

Retorna o nome do algoritmo de assinatura.

public Object getObject ()

Retorna o objeto encapsulado.

public byte[] getSignature()

Retorna a assinatura do objeto assinado na forma de um array de bytes.

public verify(PublicKey verificationKey, Signature verificationEngine)

Verifica a validade da assinatura do objeto armazenado por um `SignedObject`, de acordo com a chave pública e o motor de verificação de assinatura fornecido.

Abaixo, um exemplo utilizando a classe `SignedObject` para assinar e verificar a assinatura de um objeto utilizando o algoritmo DSA para gerar as chaves e o algoritmo de assinatura SHA1withDSA.

```
public void utilizandoSignedObject() throws InvalidKeyException,
    SignatureException, IOException, NoSuchAlgorithmException{

    Integer senha = new Integer(123456);
    KeyPairGenerator kpg = KeyPairGenerator.getInstance("DSA");
    KeyPair kp = kpg.generateKeyPair();
    DSAPrivateKey chavePrivada = (DSAPrivateKey) kp.getPrivate();
    DSAPublicKey chavePublica = (DSAPublicKey) kp.getPublic();
    Signature assinador = Signature.getInstance("SHA1withDSA");
    //assina o objeto
    SignedObject so = new SignedObject(senha, chavePrivada,
    assinador);
    // verifica a assinatura do objeto
    System.out.println(so.verify(chavePublica, assinador));
}
```

11 CERTIFICADOS DIGITAIS

As assinaturas digitais fornecem um meio de autenticação, porém este meio não é totalmente confiável. Isto porque a chave pública enviada ao destinatário pode não ser a real chave do emissor. Algum invasor pode ter capturado a chave de alguém e utilizado essa chave para receber informações. Desta forma o invasor pode enviar uma mensagem fraudulenta, fazendo se passar pela entidade a qual a chave pública foi capturada.

Então como saber se uma chave pública pertence à entidade em questão?

Quem responde esta pergunta são os certificados digitais. Os certificados digitais fornecem uma maneira de saber se uma chave pública pertence ao emissor da mensagem.

Um certificado, entre outras informações, possui um nome, uma chave pública e uma assinatura. O nome e a chave pública se refere à entidade e a assinatura é realizada por uma autoridade certificadora (AC) [1].

As ACs são entidades públicas ou privadas com estrutura segura o suficiente para guardar, sigilosamente, os dados de seus clientes. Sua função essencial é verificar se o titular do certificado possui a chave privada correspondente à chave pública que faz parte do certificado. É uma AC que emite os certificados para os usuários finais [7].

Para verificar a autenticidade do certificado é necessário verificar a assinatura contida no certificado através da chave pública disponível no certificado. Se algum impostor realizar alguma alteração no certificado digital, a assinatura não será verificada e isto significa que a chave pública contida no certificado não é confiável e, provavelmente, não pertencerá à entidade em questão [7].

As classes Java que dão suporte a certificados digitais são: `Certificate`, `X509Certificate`, `CRL`, `X509CRL`, `CertificateFactory`. Dentro de cada classe será explicado como ela funciona e o que ela representa.

Será mostrado apenas um exemplo de como importar um certificado digital, devido a falta de recursos para lidar com estas classes e testar as funcionalidades das mesmas.

11.1 A classe `Certificate`

A classe `Certificate` é uma classe abstrata que serve para o gerenciamento de certificados digitais.

Esta classe é uma abstração dos certificados que possuem diferentes formatos, mas funcionalidades importantes em comum. Por exemplo, diferentes tipos de certificados, como X509 e PGP, compartilham funcionalidades gerais (como a verificação de um certificado, por exemplo) e alguns tipos de informação (como chaves públicas). Vale lembrar que o formato X509 é o único suportado pela API Java. Se for necessário o suporte a outro formato de certificado, a subclasse de `Certificate` deverá ser implementada.

Abaixo alguns métodos que podem ser utilizados pela classe `Certificate`

`public abstract byte[] getEncoded()`

Retorna um array de bytes do certificado na sua forma codificada.

`public abstract void verify(PublicKey pk)`

`public abstract void verify(PublicKey pk, String provider)`

Verifica se um certificado foi assinado usando a chave privada que corresponde com a chave pública fornecida. Opcionalmente, o método usa o motor de verificação de assinatura do provedor de segurança especificado. Para um

certificado ser válido, a assinatura da autoridade certificadora deve ser válida. Se um certificado for inválido, este método dispara a exceção `CertificateException`.

A assinatura da autoridade certificadora é verificada de acordo com sua assinatura digital. O processo de criação de um objeto para verificação de assinaturas digitais, assim com a verificação das assinaturas podem disparar as exceções `NoSuchProviderException`, `NoSuchAlgorithmException`, `InvalidKeyException`, e a `SignatureException`.

public abstract PublicKey getPublicKey()

Retorna a chave pública do certificado.

11.2 A classe `CertificateFactory`

A classe `CertificateFactory` define as funcionalidades de um fábrica de certificados, que é usada para importar certificados, `CertPaths` (cadeia de certificados em comum) e `CRLs` (Certificate Revocation List - Lista de certificados revogados que será abordado adiante).

Um `CertPath` é uma classe que representa uma seqüência imutável de certificados. Cada objeto da classe possui um tipo que identifica o tipo dos certificados, uma lista de certificados que devem conter o mesmo tipo e uma ou mais formas de codificação..

O suporte a uma ou mais formas de codificação é necessário para que o objeto possa ser transformado em um array de bytes para armazenamento ou transmissão para outras partes. As formas de codificação devem ser baseadas em padrões bem documentados como, por exemplo, o padrão PKCS#7 (Public Key

Cryptography Standards - Padrão de Criptografia de Chaves Públicas). O PKCS#7 se refere à sintaxe padrão para dados que contém criptografia aplicada como assinaturas digitais.

Os métodos da classe são os seguintes:

public Certificate generateCertificate(InputStream inStream)

Cria um certificado com os dados contidos no parâmetro *inStream*

public Collection generateCertificates(InputStream inStream)

Retorna uma coleção de certificados a partir do parâmetro *inStream*

public CertPath generateCertPath(InputStream inStream)

public CertPath generateCertPath(InputStream inStream, String encoding)

public CertPath generateCertPath(List certificates)

Gera uma cadeia de certificados com o parâmetro *inStream*, e se desejado o tipo de codificação pode ser informado. Esta cadeia pode ser gerada através de uma lista de certificados.

public CRL generateCRL(InputStream inStream)

Gera um certificado revogado a partir do parâmetro *inStream*.

public Collection generateCRLs(InputStream inStream)

Gera uma lista de certificados revogados a partir do parâmetro *inStream*.

public Iterator getCertPathEncodings()

Retorna um objeto `Iterator` que contém as codificações do objeto `CertPath`. A primeira codificação retornada é a default.

public static CertificateFactory getInstance(String type)

public static CertificateFactory getInstance(String type, String Provider)

Retorna uma instância da classe `CertificateFactory` que implementa

o tipo do certificado especificado pelo parâmetro. Opcionalmente, o provedor de segurança pode ser passado como parâmetro.

public Provider getProvider()

Retorna o provedor do objeto `CertificateFactory`

public String getType()

Retorna o tipo do certificado associado ao objeto.

11.3 A classe X509Certificate

Um dos formatos de certificados muito utilizados e suportados pela API Java são os certificados que possuem o formato X509. Um certificado X509 possui algumas propriedades específicas e que não são abordadas pela classe de certificados base, a classe `Certificate`.

As propriedades de um certificado X509 são:

- Período de validade: que contém a data inicial e final correspondente a validade do certificado.

- Número serial (serial number): cada certificado emitido por uma entidade certificadora possui número de série único. Este número é único para a autoridade certificadora em questão

- Versão (version): existem várias versões do padrão X509. A implementação default desta classe usa a versão 3 do padrão X509.

- Nome distinto (distinguished name) da autoridade certificadora e do sujeito representado pelo certificado: um distinguished name ou simplesmente DN, é uma identificação única composta por alguns parâmetros:

- 1) Common Name (CN): nome completo da entidade certificadora ou do sujeito

- 2) Organizational Unit (OU): unidade da organização pertencente
- 3) Organization (O): organização associada
- 4) Location (L): cidade onde o sujeito ou a entidade certificadora está localizada
- 5) State (S): estado ou província
- 6) Country (C): país referente ao sujeito ou a entidade em questão.

A especificação de um DN permite informações adicionais, mas em Java são aceitos apenas os parâmetros descritos anteriormente.

Abaixo um exemplo de um *distinguished name*:

CN=Filipe Ferreira OU=Núcleo de Desenvolvimento O=JExperts Tecnologia L=Florianópolis S=Santa Catarina C= Brasil

As propriedades de um certificado podem ser recuperadas através dos métodos abaixo:

public abstract void checkValidity()

public abstract void checkValidity(Date d)

Verifica se a data de validade do certificado(ou o dia atual se nenhuma data for especificada) está entre a data inicial e final da validade do certificado. Se o parâmetro data for menor que a data inicial da validade do certificado, a exceção `CertificateNotYetValidException` será disparada. Caso a data for maior que a data final da validade, a exceção `CertificateExpiredException` será disparada.

public abstract int getVersion()

Retorna a versão da especificação X509 que o certificado foi criado.

public abstract BigInteger getSerialNumber()

Retorna o número de série do certificado

public abstract Principal getIssuerDN()

Retorna o *distinguished name* do emissor do certificado.

public abstract Principal getSubjectDN()

Retorna o *distinguished name* do sujeito do certificado.

public abstract Date getNotBefore()

Retorna a data inicial da validade do certificado.

public abstract Date getNotAfter()

Retorna a data final da validade do certificado.

11.4 CRL (Certificate Revocation List)

Algumas vezes, uma autoridade certificadora precisa revogar um certificado emitido devido a razões de condutas ilegais na utilização do certificado por um usuário, por exemplo, e em situações como estas o certificado deve ser invalidado imediatamente.

Esta invalidação é o resultado de uma lista de certificados revogados (certificate revocation list). As autoridade certificadoras são responsáveis por emitir as CRLs. Validadores de certificados devem realizar uma consulta nesta lista para verificar se o certificado em questão é válido ou não.

11.4.1 A classe CRL

Esta classe é uma abstração de uma lista de certificados revogados (CRL) que possuem formatos diferentes, mas funcionalidades em comum. Por exemplo, todas as CRLs possuem a funcionalidade de verificar se um dado certificado está revogado ou não.

Esta classe possui três métodos abstratos que devem ser implementados

pela subclasse que estende a classe `CRL`.

public String getType()

Retorna o tipo da CRL.

public boolean isRevoked(Certificate cert)

Verifica se um certificado foi revogado ou não.

public String toString()

Retorna a CRL em uma representação String.

11.4.2 A classe `X509CRL`

Enquanto a noção de certificados revogados não é necessariamente específica a um certificado X509, para a implementação Java é. O provedor de segurança *default* Java fornece suporte apenas para lista de certificados revogados X509.

A noção de uma lista de certificados revogada X509 é representada pela classe `X509CRLEntry`:

public abstract class X509CRLEntry implements X509Extension

Os métodos desta classe são simples e baseados nos atributos presentes em um certificado revogado X509:

public abstract BigInteger getSerialNumber()

Retorna o número de série de um certificado revogado.

public abstract Date getRevocationDate()

Retorna a data em que o certificado foi revogado.

public abstract boolean hasExtensions()

Indica se a implementação da classe possui alguma extensão X509.

Certificados revogados são modelados através da classe `X509CRLEntry`:

public abstract class X509CRLEntry implements X509Extension

Instâncias da classe `X509CRLEntry` são obtidas via método `getInstance()` da classe `CertificateFactory`. Uma vez instanciado, o objeto pode operar sobre os métodos abaixo:

public abstract void verify(PublicKey pk)

public abstract void verify(PublicKey pk, String sigProvider)

Verifica que a CRL foi assinada usando a chave privada que corresponde à chave pública do parâmetro e, opcionalmente, utilizando o provedor de segurança fornecido.

public abstract Principal getIssuerDN()

Retorna o *distinguished name* do emissor da CRL.

public abstract Date getNextUpdate()

Retorna a próxima data de emissão da CRL.

public abstract X509CRLEntry getRevokedCertificate(BigInteger serialNumber)

Retorna o certificado revogado X509 de acordo com o número de série do certificado, caso exista este certificado na lista dos certificados revogados.

public abstract Set getRevokedCertificates()

Retorna todos os certificados revogados da lista de certificados revogados.

public abstract byte[] getSignature()

Retorna o valor da assinatura da CRL.

public abstract String getSigAlgName()

Retorna o nome do algoritmo de assinatura para o algoritmo de assinatura

da CRL.

public abstract String getSigAlgOID()

Retorna o OID (*object identifier* – identificador do objeto) da CRL.

Abaixo, um exemplo de como importar um certificado X509 a partir de um arquivo:

```
public void importandoCertificado(String caminhoCertificado) throws
    FileNotFoundException, java.security.cert.CertificateException {

    FileInputStream fis = new FileInputStream(new
    File(caminhoCertificado));
    CertificateFactory cert =
    CertificateFactory.getInstance("X509");
    X509Certificate x509 = (X509Certificate)
    cert.generateCertificate(fis);

    System.out.println("Versão = " + x509.getVersion());
    System.out.println("Válido de " + x509.getNotBefore() + " até " +
    x509.getNotAfter());
    System.out.println("Número de série = " +
    x509.getSerialNumber());
    Principal emissor = x509.getIssuerDN();
    Principal destinatario = x509.getSubjectDN();
    System.out.println("Emissor = " + emissor.getName());

    System.out.println("Destinatário = " + destinatario.getName());

}
```

Neste exemplo deve-se ter em mãos o Certificado X509 versão 3, pois o Java apenas dá suporte a este tipo de certificado.

12 KEYTOOL

A *Keytool* é uma ferramenta utilizada para o gerenciamento de chaves criptográficas e certificados digitais. Permite aos usuários administrar suas próprias chaves e certificados para autenticação própria ou para a integridade de dados utilizando assinaturas digitais.

Um certificado digital, como veremos adiante, é uma declaração digitalmente assinada por uma organização responsável, afirmando que uma determinada chave pública pertence a uma entidade. Uma entidade pode ser uma pessoa física ou jurídica.

As chaves e os certificados são armazenados nos chamados *keystores*. As entradas de um *keystore* são:

- Chaves: são armazenadas junto a uma senha para prevenir acessos não autorizados
- Certificados: cada um contendo uma chave pública pertencente a uma outra parte.

As *keystores* armazenam suas entradas juntamente a um *alias* ou apelido. Estes apelidos não são caso-sensitivo indicando que as letras maiúsculas e minúsculas não serão diferenciadas. Por exemplo, o apelido Hugo equivale a hugo ou HuGo.

Por exemplo, a linha de comando abaixo associa o nome filipe ao par de chaves que será gerado para este apelido juntamente com a senha informada:

```
keytool - genkey -alias filipe -keypass pwd
```

A ferramenta *keytool* está localizada dentro do diretório da instalação do

java/bin, por exemplo, /jdk1.6.0_10/bin.

Um *keystore* é criado mediante o comando `-genkey`, `-import` ou `-identitydb`, sendo que o último não será exemplificado. A opção `-keystore` permite especificar o local e o nome do arquivo da base de chaves conforme o exemplo abaixo. Neste exemplo, realizou-se a criação da *keystore*, juntamente a uma senha obrigatória para o keystore, no diretório `c:\ks\filipe.keystore`. Caso o nome do arquivo não seja especificado, será criado um arquivo sem nome, com a extensão `.keystore` no diretório *home* do usuário.

```
C:\Arquivos de programas\Java\jre6\bin>keytool -genkeypair -alias filipe -keypas
s 123456 -keystore c:\ks\filipe.keystore
Enter keystore password:
Re-enter new password:
What is your first and last name?
 [Unknown]: Filipe Ferreira
What is the name of your organizational unit?
 [Unknown]: Desenvolvimento
What is the name of your organization?
 [Unknown]: JExperts
What is the name of your City or Locality?
 [Unknown]: Florianópolis
What is the name of your State or Province?
 [Unknown]: Santa Catarina
What is the two-letter country code for this unit?
 [Unknown]: SC
Is CN=Filipe Ferreira, OU=Desenvolvimento, O=JExperts, L=Florianópolis, ST=Santa
Catarina, C=SC correct?
 [no]: yes

C:\Arquivos de programas\Java\jre6\bin>_
```

A implementação de um keystore é feito pela classe `KeyStore` contida no pacote `java.security`. Esta classe fornece interfaces bem definidas para acessar e modificar informações contidas em um keystore. As implementações dos keystores são baseadas em provedores e cada provedor possui tipo ou formato proprietário. Por ser baseado em provedores, diferentes formatos de keystore podem ser implementados pelos provedores de segurança.

Os aplicativos podem optar por diferentes formatos implementado por diferentes provedores. O método que permite a escolha do tipo e do provedor é o `getInstance()`.

```
public static Keystore getInstance(String type)
public static Keystore getInstance(String type, String
provider)
public static Keystore getInstance(String type, Provider
provider)
```

O tipo da *keystore* define o modo de armazenamento e o formato dos dados, e os algoritmos usados para proteger as chaves privadas contidas em um *keystore*.

O tipo padrão adotado pelo provedor de segurança da SUN está informado no arquivo `java.security` através do parâmetro:

```
keystore.type = jks
```

Esta linha pode ser modificada caso exista uma implementação de um *keystore* com um tipo proprietário.

O parâmetro *type* padrão pode ser retornado através do método `KeyStore.getDefaultType()`.

A ferramenta `keytool` possui uma diversidade de comandos para adicionar, visualizar, exportar e importar dados. Antes de apresentar os comandos, será demonstrado um conjunto de opções globais para a maioria dos comandos:

-v: esta opção mostra as informações detalhadas de um certificado digital;

`-storetype storetype`: especifica o formato do keystore.

`-storepass storepass`: é a senha usada para proteger a integridade de um keystore

`-provider provider-class-name`: especifica a classe do provedor de serviços criptográficos quando esta classe não está especificada no arquivo de propriedades de segurança `java.security`.

Visto as opções globais que podem ser utilizadas, serão mostrados os comandos para manipulação de um keystore.

12.1 Comandos para adicionar dados a um keystore

```
-genkey {-alias alias} {-keyalg keyalg} {-keysize keysize} {-sigalg sigalg} {-dname dname} {-keypass keypass} {-validity valDays} {-storetype storetype} {-keystore keystore} {-storepass storepass} {-provider provider_class_name} {-v}
```

Gera um par de chaves, uma chave pública e uma chave privada associada e encapsula a chave pública em um certificado X509. Abaixo a descrição dos parâmetros.

`-keyalg` é o algoritmo usado para gerar o par de chaves

`-keysize` especifica o tamanho da chave.

`-sigalg` é o algoritmo utilizado para criar a assinatura contida no certificado digital.

`-dname` é o distinguished name associado ao apelido `alias` utilizado para gerar as chaves. Se o usuário não fornecer o distinguished name, a ferramenta solicita os dados para este parâmetro.

`-keypass` é a senha usada para proteger a chave privada. Da mesma

forma que o *distinguished name*, a ferramenta obrigará o usuário a informar sua senha.

`-validity`: indica a quantidade de dias que este certificado será validado.

```
-import {-alias alias} {-file cert_file} [-keypass keypass] {-noprompt} {-trustcacerts} {-storetype storetype} {-keystore keystore} [-storepass storepass] [-provider provider_class_name] {-v}
```

Lê o certificado, ou a cadeia de certificados, contido arquivo `cert_file`, e armazena no *keystore* identificado pelo apelido *alias*.

A opção `-trustcacerts` é utilizada para obter certificados digitais de autoridades certificadoras que realizaram a assinatura dos certificados. Os certificados são armazenados em um arquivo chamado “cacerts”. Este arquivo é automaticamente criado no diretório de instalação do java, dentro da pasta `/lib/security`.

Quando um usuário deseja importar um certificado que não é assinado por uma autoridade certificadora, ele será interrogado se deseja realmente importar este certificado. Através da opção `-noprompt`, o certificado é automaticamente instalado, sem a intervenção da ferramenta.

```
-selfcert {-alias alias} {-sigalg sigalg} {-dname dname} {-validity valDays} [-keypass keypass] {-storetype storetype} {-keystore keystore} [-storepass storepass] {-provider provider_class_name} {-v}
```

Gera um certificado X509 que possui uma assinatura própria. Caso o parâmetro `dname` não seja informado, será utilizado o *distinguished name* associado ao apelido *alias*. Este *distinguished name*, como visto anteriormente, é solicitado quando um *keystore* é criado.

12.2 Comando para exportar dados

```
-certreq {-alias alias} {-sigalg sigalg} {-file certreq_file} [-keypass keypass] {-storetype storetype} {-keystore keystore} {-storepass storepass} {-provider provider_class_name} {-v}
```

Gera uma Requisição de Assinatura de Certificado (CSR – Certificate Signing Request) utilizando o formato PKCS#10.

Um CSR é destinado a uma autoridade certificadora (AC). Uma AC autenticará o certificado do requisitante e o retornará.

```
-export {-alias alias} {-file cert_file} {-storetype storetype} {-keystore keystore} [-storepass storepass] [-provider provider_class_name] {-rfc} {-v} {-Jjavaoption}
```

Lê o certificado associado ao alias e armazena em um arquivo especificado pelo parâmetro *cert_file*.

12.3 Comandos para visualizar dados

```
-list {-alias alias} {-storetype storetype} {-keystore keystore} [-storepass storepass] [-provider provider_class_name] {-v | -rfc} {-Jjavaoption}
```

Imprime o conteúdo de uma *keystore* associada ao alias especificado. Caso não seja informado o alias, o conteúdo da *keystore* inteira será impresso.

```
-printcert {-file cert_file} {-v} {-Jjavaoption}
```

```
C:\Arquivos de programas\Java\jre6\bin>keytool -list -alias filipe -keystore c:\ks\filipe.keystore
Enter keystore password:
filipe, 19/06/2008, PrivateKeyEntry,
Certificate fingerprint (MD5): AE:DF:80:7B:AB:83:1D:BF:A4:86:DB:B8:4C:71:19:D2
C:\Arquivos de programas\Java\jre6\bin>_
```

Lê o certificado contido no arquivo `cert_file` e imprime seu conteúdo.

12.4 Comandos de gerenciamento da keystore

```
-keyclone {-alias alias} [-dest dest_alias] [-keypass keypass] [-new new_keypass] {-storetype storetype} {-keystore keystore} [-storepass storepass] [-provider provider_class_name] {-v}
```

Realiza uma cópia das chaves de uma entidade representada por `alias` para uma entidade destino representada por `dest_alias`.

```
-storepasswd [-new new_storepass] {-storetype storetype} {-keystore keystore} [-storepass storepass] [-provider provider_class_name] {-v}
```

Gera uma nova senha para um determinado `keystore`. A nova senha é representada pelo parâmetro `new_storepass`

```
-keypasswd {-alias alias} [-keypass old_keypass] [-new new_keypass] {-storetype storetype} {-keystore keystore} [-storepass storepass] [-provider provider_class_name] {-v} {-Jjavaoption}
```

Modifica a senha da chave privada de uma entidade. É necessário informar a senha atual, representada por `old_keypass`, e a nova senha representada por `new_keypass`.

```
-delete [-alias alias] {-storetype storetype} {-keystore keystore} [-storepass storepass] [-provider provider_class_name] {-v}
```

Remove o keystore representada por `alias`.

13 POLICY TOOL

A *policy tool* é uma ferramenta disponibilizada pela SUN para editar as políticas de segurança de um aplicativo Java. As políticas de segurança são definidas através das permissões de acesso aos recursos do sistema.

A ferramenta está contida dentro do diretório de instalação *java/bin*, por exemplo, */jdk1.6.0_10/bin*. Para iniciar a ferramenta execute o comando *policytool*.

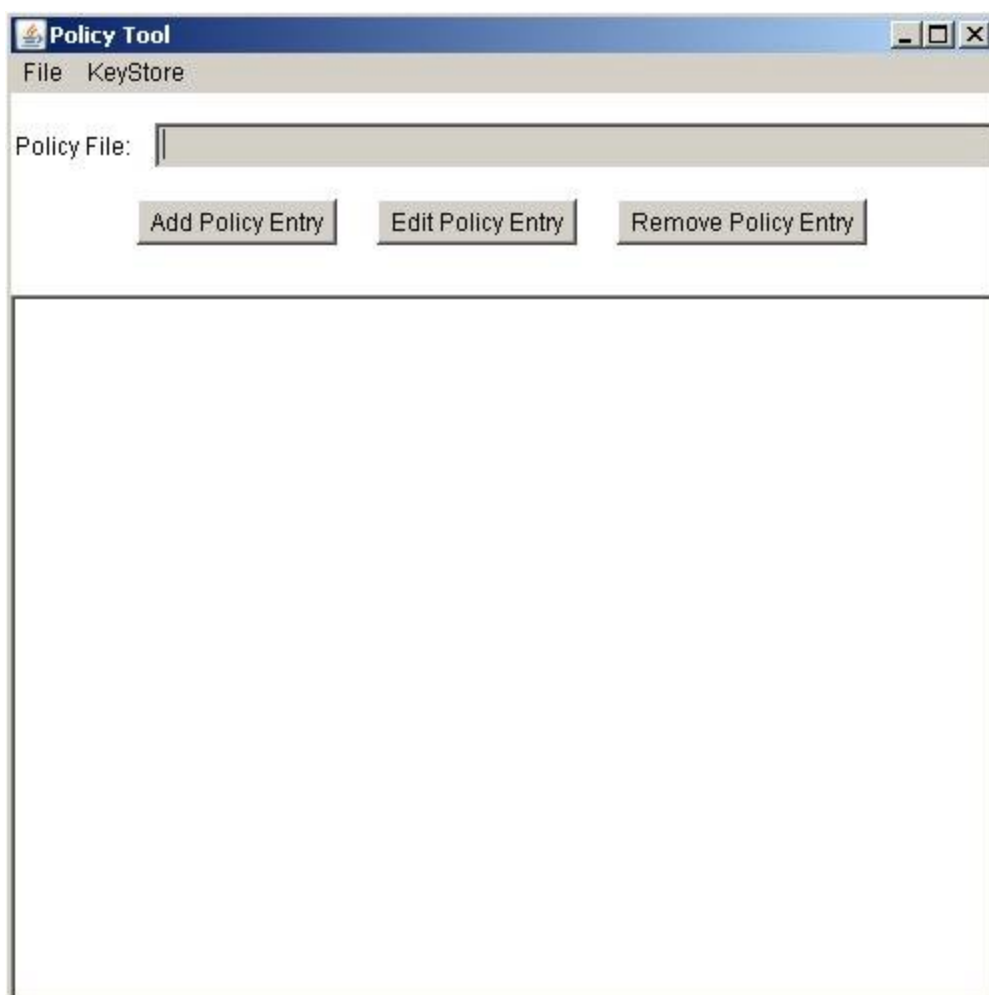


Figura 13.1 Policy Tool

Para a criação de um arquivo de políticas, selecione a opção `File ->New`. Esta opção encerrará o arquivo atual e um novo arquivo será aberto. A opção `File->Open` permite editar um arquivo de políticas existente.

Tendo o arquivo de políticas, pode-se criar as permissões através da opção `Add Policy Entry` ou especificar um *keystore* a ser utilizado caso alguma entrada contida no arquivo de políticas utilize um *alias* que referencie um *keystore*. Para informar o *keystore* utilizado seleciona a opção `KeyStore->Edit`.

Selecionando a opção `Add Policy Entry`, a seguinte tela será aberta:

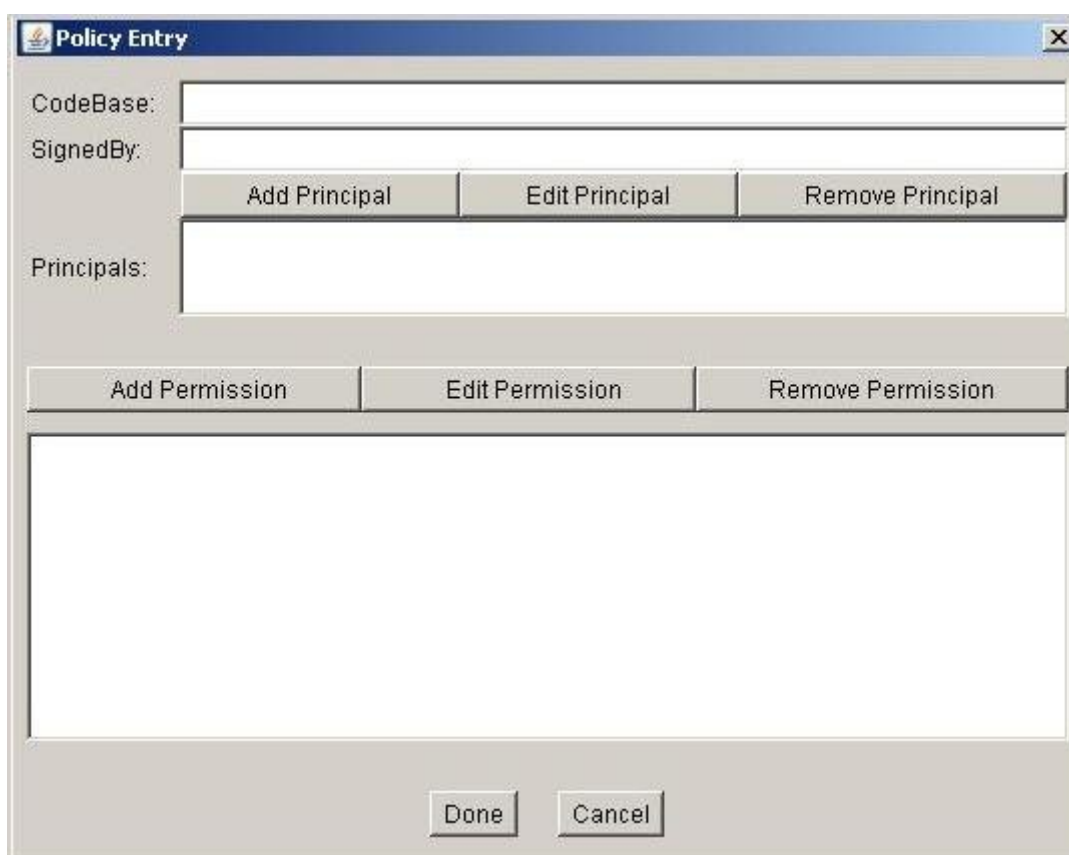


Figura 13.2 Policy Entry

- A opção `CODEBASE` informa a localização do `CodeSource`. Ex: `c:\workspace\`;
- A opção `SignedBy` indica o *alias* do responsável pela chave privada utilizada para assinar o `CodeSource`;
- A opção `Principals` refere-se a uma lista de entidades. Um `Principal` pode ser adicionado pela opção `Add Principal`.

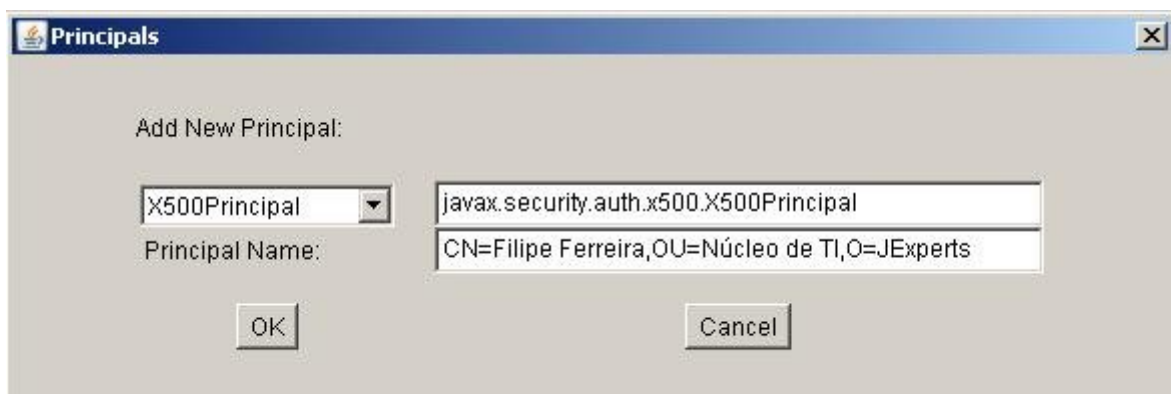


Figura 13.3 Adicionando um principal

No caso das permissões, a ferramenta permite adicionar, editar ou remover permissões. Para adicionar uma permissão selecione a opção `Add Permission`. Para esta opção, os seguintes parâmetros são obrigatórios:

- `Permission`: tipo da permissão ;
- `Target Name`: o diretório alvo;

- Actions: as ações permitidas para a Target Name .

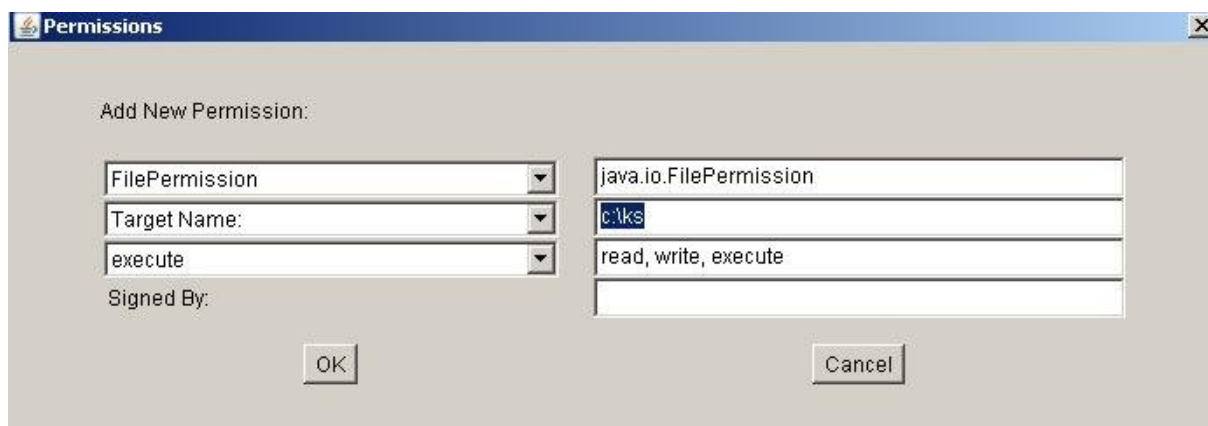


Figura 13.4 Adicionando uma permissão

CONSIDERAÇÕES FINAIS

Realizando o estudo da tecnologia Java, verificou-se que o Java foi modelado seguindo regras bem definidas de segurança, o que justifica a larga utilização da linguagem no mercado de software.

Estas regras são asseguradas por componentes da arquitetura Java, como o compilador, o verificador de *bytecodes* e a máquina virtual Java.

A linguagem, através dos provedores de segurança, que implementam os motores de criptografia, permite aplicar os conceitos largamente utilizados no mundo da segurança digital como criptografia assimétrica e simétrica, assinaturas digitais, resumos de mensagens e certificados digitais.

Além de suportar estes conceitos, o Java é uma tecnologia flexível. Permitindo assim, que sua arquitetura de segurança seja estendida por implementações próprias do gerenciador de segurança e provedores de segurança que possuam algoritmos proprietários.

Segurança é um tema delicado e exige cautela. A arquitetura de segurança Java fornece meios para o desenvolvimento de aplicativos com segurança, mas, para isto, é necessário ter conhecimento da linguagem Java, e principalmente dominar os conceitos de segurança antes de iniciar o desenvolvimento de um aplicativo.

REFERÊNCIAS BIBLIOGRÁFICAS

- [1] BURNETT, Steve; PAINE, Stephen. Criptografia e Segurança: O Guia Oficial RSA. 4. ed. Rio de Janeiro: Elsevier, 2002.
- [2] FLANAGAN, David. Java in a Nutshell. 3. ed. [S.l.]: O'Reilly, 1999.
- [3] OAKS, Scott. Segurança de Dados em Java. 1. ed. [S.l.]: O'Reilly, 1999.
- [4] OAKS, Scott. Java Security. 2. ed. [S.l.]: O'Reilly, 2005.
- [5] Java 2 Platform Security Architecture. [S.l.]: Sun Microsystems, 1997 – 2002. Disponível em: <<http://java.sun.com/javase/6/docs/technotes/guides/security/spec/security-spec.doc.html>>. Acesso em: 04 junho 2008.
- [6] Java Cryptography Reference Guide for Java Platform Standard Edition 6. [S.l.]: Sun Microsystems, 1995 – 2006. Disponível em: <<http://java.sun.com/javase/6/docs/technotes/guides/security/crypto/CryptoSpec.html>>. Acesso em: 04 junho 2008.
- [7] BONDAN, G. I. R. B.; COSTA, L. S. Autoridade Certificadora Otimizadora. Disponível em: http://projetos.inf.ufsc.br/arquivos_projetos/projeto_646/ACO.pdf.
- [8] CHAN, Patrick. The Java Developers Almanac 1.4. 1. ed. [S.l.]: Addison Wesley, 2002.
- [9] GONG, Li; ELLISON, Gary; DAGEFORDE, Mary. Inside Java 2 Platform Security: Architecture, API Design, and Implementation. 2. ed. [S.l.]: Addison Wesley, 2003.
- [10] Keytool – Key and Certificate Management Tool. [S.l.]: Sun Microsystems, 2002. Disponível em: <<http://java.sun.com/j2se/1.3/docs/tooldocs/win32/keytool.html>>. Acesso em: 04 junho 2008.
- [11] Policy Tool – Policy File Creation and Management Tool. [S.l.]: Sun Microsystems, 2002. Disponível em: <<http://java.sun.com/j2se/1.4.2/docs/tooldocs/windows/policytool.html>>. Acesso em: 02 junho 2008.