Módulo 7

Segurança

Lição 3

Gerenciadores de Segurança

Versão 1.0 - Jan/2008

Autor
Aldwin Lee
Cheryl Lorica

Equipe
Rommel Feria
John Paul Petines

Necessidades para os Exercícios

Sistemas Operacionais Suportados

NetBeans IDE 5.5 para os seguintes sistemas operacionais:

- Microsoft Windows XP Profissional SP2 ou superior
- Mac OS X 10.4.5 ou superior
- Red Hat Fedora Core 3
- Solaris™ 10 Operating System (SPARC® e x86/x64 Platform Edition)

NetBeans Enterprise Pack, poderá ser executado nas seguintes plataformas:

- Microsoft Windows 2000 Profissional SP4
- Solaris™ 8 OS (SPARC e x86/x64 Platform Edition) e Solaris 9 OS (SPARC e x86/x64 Platform Edition)
- Várias outras distribuições Linux

Configuração Mínima de Hardware

Nota: IDE NetBeans com resolução de tela em 1024x768 pixel

Sistema Operacional	Processador	Memória	HD Livre
Microsoft Windows	500 MHz Intel Pentium III workstation ou equivalente	512 MB	850 MB
Linux	500 MHz Intel Pentium III workstation ou equivalente	512 MB	450 MB
Solaris OS (SPARC)	UltraSPARC II 450 MHz	512 MB	450 MB
Solaris OS (x86/x64 Platform Edition)	AMD Opteron 100 Série 1.8 GHz	512 MB	450 MB
Mac OS X	PowerPC G4	512 MB	450 MB

Configuração Recomendada de Hardware

Sistema Operacional	Processador	Memória	HD Livre
Microsoft Windows	1.4 GHz Intel Pentium III workstation ou equivalente	1 GB	1 GB
Linux	1.4 GHz Intel Pentium III workstation ou equivalente	1 GB	850 MB
Solaris OS (SPARC)	UltraSPARC IIIi 1 GHz	1 GB	850 MB
Solaris OS (x86/x64 Platform Edition)	AMD Opteron 100 Series 1.8 GHz	1 GB	850 MB
Mac OS X	PowerPC G5	1 GB	850 MB

Requerimentos de Software

NetBeans Enterprise Pack 5.5 executando sobre Java 2 Platform Standard Edition Development Kit 5.0 ou superior (JDK 5.0, versão 1.5.0_01 ou superior), contemplando a Java Runtime Environment, ferramentas de desenvolvimento para compilar, depurar, e executar aplicações escritas em linguagem Java. Sun Java System Application Server Platform Edition 9.

- Para **Solaris**, **Windows**, e **Linux**, os arquivos da JDK podem ser obtidos para sua plataforma em http://java.sun.com/j2se/1.5.0/download.html
- Para **Mac OS X**, Java 2 Plataform Standard Edition (J2SE) 5.0 Release 4, pode ser obtida diretamente da Apple's Developer Connection, no endereço: http://developer.apple.com/java (é necessário registrar o download da JDK).

Para mais informações: http://www.netbeans.org/community/releases/55/relnotes.html

Colaboradores que auxiliaram no processo de tradução e revisão

Aécio Júnior
Alexandre Mori
Alexis da Rocha Silva
Angelo de Oliveira
Bruno da Silva Bonfim
Denis Mitsuo Nakasaki
Emanoel Tadeu da Silva Freitas
Guilherme da Silveira Elias
Leandro Souza de Jesus
Lucas Vinícius Bibiano Thomé
Luiz Fernandes de Oliveira Junior
Maria Carolina Ferreira da Silva
Massimiliano Giroldi
Paulo Oliveira Sampaio Reis
Ronie Dotzlaw

Auxiliadores especiais

Revisão Geral do texto para os seguintes Países:

- **Brasil** – Tiago Flach e Vinícius G. Ribeiro (Especialista em Segurança)
- **Guiné Bissau** – Alfredo Cá, Bunene Sisse e Buon Olossato Quebi – ONG Asas de Socorro

Coordenação do DFJUG

- **Daniel deOliveira** – JUGLeader responsável pelos acordos de parcerias
- **Luci Campos -** Idealizadora do DFJUG responsável pelo apoio social
- **Fernando Anselmo -** Coordenador responsável pelo processo de tradução e revisão, disponibilização dos materiais e inserção de novos módulos
- **Rodrigo Nunes -** Coordenador responsável pela parte multimídia
- **Sérgio Gomes Veloso -** Coordenador responsável pelo ambiente JEDI™ (Moodle)

Agradecimento Especial

John Paul Petines – Criador da Iniciativa JEDI™
Rommel Feria – Criador da Iniciativa JEDI™

1. Objetivos

De acordo com a perspectiva das API's do Java, existe um gerenciador de segurança que se encontra no controle do policiamento da segurança de uma aplicação. Existem também outras facetas que são de importância para a segurança em Java, mas a regra utilizada pelo gerenciador é vital na definição das rotinas de segurança que serão ativadas na execução de um programa em particular.

Numa visão simplificada, o gerenciador de segurança é responsável por determinar a maioria dos argumentos da Java *Sandbox*. Isto significa que todas as regras estabelecidas pelo gerenciador determinam quais operações são permitidas ou não. Se um aplicativo Java tenta abrir um arquivo, deseja se conectar com outra máquina em uma rede, ou se deseja alterar o estado de um serviço, o gerenciador de segurança decide se autoriza tais operações ou as rejeita, baseando-se nas regras definidas.

Ao final desta lição, o estudante será capaz de:

- Compreender a arquitetura dos Gerenciadores de Segurança
- Conhecer os métodos dos Gerenciadores de Segurança
- Construir um Gerenciador de Segurança customizado

2. Gerenciadores de Segurança de Aplicativos

Applets Java possuem uma segurança muito restritiva por padrão. Por outro lado, aplicativos Java não possuem estas definições de segurança por padrão. Então, é possível notar que o gerenciador de segurança é utilizado somente se for explicitamente instalado.

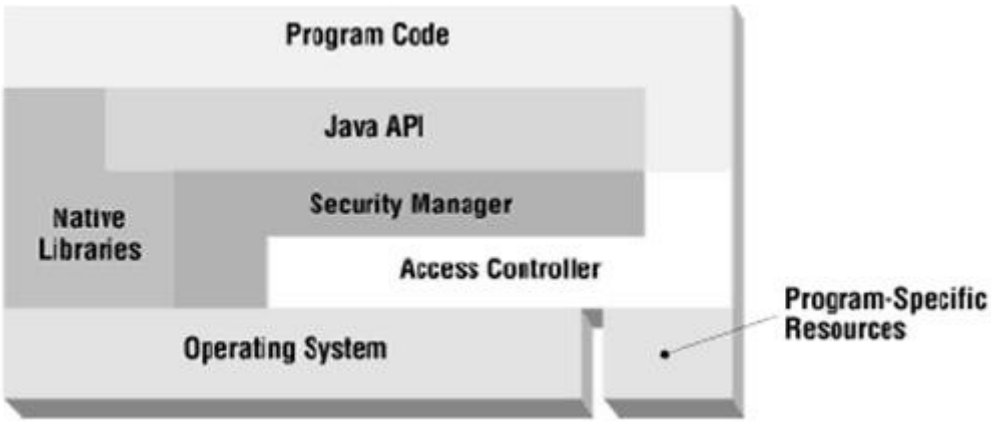

Figura 1: Estrutura do Gerenciador de Segurança Java

Para instalar o gerenciador de segurança especifica-se o parâmetro –D *java.security.manager* na execução de um aplicativo Java. Este gerenciador será automaticamente instalado pelo *Applet Viewer* ou um *plug-in* Java.

É possível conferir se uma aplicação já se encontra com o gerenciador de segurança carregado também é possível. Isto é realizado por meio do método *getSecurityManager*. Este método retorna uma referência relativa ao gerenciador de segurança ou *null*, se nenhum gerenciador de segurança foi carregado ou definido. Como exemplo:

```
SecurityManager secureApp = System.getSecurityManager();
```

Outro método que pode ser utilizado é o *setSecurityManager*. Permite que o usuário configure manualmente o gerenciador de segurança para o sistema. Como por exemplo:

```
System.getSecurityManager(new SecurityManagerImpl());
```

3. Métodos dos Gerenciadores de Segurança

Agora que conhecemos como o gerenciador de segurança funciona, mostraremos o que o gerenciador de segurança é capaz de fazer. As classes do gerenciador de segurança, *SecurityManager*, oferecem métodos para realizar a verificação de segurança. Estes métodos agem em conjunto com alguns aspectos do sistema Java, tais como: acesso à arquivos, à rede, à própria JVM, à serviços, à recursos do sistema e mesmo a aspectos de segurança.

3.1. Rastrear o Acesso

A seguir, temos os métodos utilizados pelo gerenciador de segurança para rastrear o acesso a arquivos.

```
public void checkRead(FileDescriptor fd)
public void checkRead(String file)
public void checkRead(String file, Object context)
```

Estes métodos verificam se o programa tem permissão para ler um arquivo. O primeiro destes métodos obtém sucesso quando a proteção de domínio permite, em tempo de execução, acesso ao arquivo por meio do *FileDescriptor*. Os próximos métodos obtém sucesso se a proteção de domínio possui permissão de leitura em um arquivo com um nome que coincida ao argumento contendo o nome do arquivo informado e a partir de um determinado contexto.

```
public void checkWrite(FileDescriptor fd)
public void checkWrite(String file)
```

Estes métodos verificam se o programa tem permissão para escrever em um determinado arquivo. O primeiro método obtém sucesso quando a proteção de domínio permite, em tempo de execução, acesso ao arquivo por meio do *FileDescriptor*. E o segundo obtém sucesso se a proteção de domínio possuir permissão de gravação em arquivo com o nome que coincida ao argumento contendo o nome do arquivo informado.

```
public void checkDelete(String file)
```

Este método verifica se o programa tem permissão para eliminar um determinado arquivo. Obtém sucesso se a proteção de domínio possuir uma permissão de exclusão de um arquivo com um nome que coincida ao argumento contendo o nome do arquivo informado.

O gerenciador de segurança utiliza os seguintes métodos para verificar o acesso à rede:

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

Estes métodos servem para verificar se o programa pode abrir uma conexão com outro computador em uma porta no computador *host*. Obtém sucesso se a proteção de domínio possuir uma conexão via *socket* com o mesmo nome do *host*, porta e em determinado contexto.

```
public void checkListen(int port)
```

Este método verifica se o programa pode criar um novo servidor em determinada porta. A proteção de domínio deve possuir uma permissão de monitoramento de *socket* onde o *host* seja *localhost*, o alcance da porta inclua a porta especificada aberta com a ação de escutar.

```
public void checkAccept(String host, int port)
```

Este método verifica se o programa pode aceitar (em um servidor existente) a conexão de um cliente que possa ser originária de um dado *host* e porta. A proteção de domínio deve possuir uma permissão de *socket* onde o *host* e a porta sejam iguais aos argumentos passados para o método.

```
public void checkMulticast(InetAddress addr)
public void checkMulticast(InetAddress addr, byte ttl)
```

Este método verifica se o programa pode criar um *multicast* em um determinado endereço IP.

Obtém sucesso se a proteção de domínio possuir uma permissão de se conectar e aceitar conexões *socket* com um *host* que seja igual ao endereço dado, o alcance da porta de todas as portas, e a ação.

```
public void checkSetFactory()
```

Este método verifica se o programa possui permissão para alterar a fábrica de *sockets*. Quando a classe *Socket* é utilizada para criar o *socket*, recebe um novo *socket* da fábrica, que fornece um *socket* padrão baseando-se no protocolo TCP. Entretanto, um programa poderia instalar uma fábrica de *sockets* que fizesse com que todos os *sockets* fornecidos possuam semânticas diferentes, tais como *sockets* que enviam informações sobre o rastreamento de algum dado. A proteção de domínio deve ter permissão durante a execução de um programa através do método *setFactory*.

3.2. Verificar a JVM

A verificação da JVM é realizada empregando-se os seguintes métodos:

```
public void checkCreateClassLoader()
```

A distinção realizada sobre as classes confiáveis e não confiáveis é baseada no local de onde a classe é carregada. Como resultado, o carregador de classe toma uma importante decisão, desde que o gerenciador de segurança esteja configurado para questionar ao carregador onde a classe se encontra. O carregador é também responsável por certificar de que determinadas classes são assinadas. Tipicamente, uma classe não confiável não pode ser permitida para realizar uma criação de um carregador de classes. Este método é chamado somente pelo construtor da classe através do *Class Loader*: se for possível realizar a criação de um objeto desta classe (ou obter previamente a referência de um objeto) para fazer uso deste. Para obter sucesso, a proteção de domínio deve possuir permissão durante a execução.

```
public void checkExec(String cmd)
```

Este método é utilizado para prevenir a execução arbitrária de comandos de sistema vindos de classes não confiáveis. Classes não confiáveis não podem, por exemplo, executar um processo distinto que remova todos os arquivos de seu disco rígido. Este processo necessitaria não ser escrito em Java, é claro, pois, caso contrário, não existiria gerenciador de segurança que pudesse restringir esta ação. Para obter sucesso, a proteção de domínio deve possuir uma permissão de execução com o mesmo nome do comando dado através do argumento passado.

```
public void checkPropertiesAccess( )
public void checkPropertyAccess(String key)
```

A JVM possui um conjunto de propriedades globais (de sistema) que contém informações sobre o usuário e a máquina: nome do usuário, diretório raiz, entre outras informações. Classes não confiáveis têm, geralmente, acesso negado a algumas dessas informações em uma tentativa de limitar a quantidade a informações (espionagem) que uma *applet* pode fazer. Normalmente, estes métodos apenas previnem acesso às propriedades de sistema; uma classe não confiável é livre para enviar suas próprias propriedades e compartilhá-las com outras classes se assim desejar. Para ter sucesso, o domínio corrente de proteção deve carregar uma permissão de propriedade. Se uma chave é especificada, então o nome da permissão de propriedade deve incluir determinada chave e ter definida uma ação de leitura. De outra forma, o nome da permissão de propriedade deve ser "*" e a ação deve ser de leitura e escrita.

```
public boolean checkTopLevelWindow(Object window)
```

Classes Java, sem levar em consideração se são confiáveis ou não confiáveis, normalmente tem permissão para criar janelas *top-level* no *desktop* do usuário. Entretanto, há uma preocupação acerca de uma classe não confiável apresentar uma janela que se pareça exatamente com outra aplicação no *desktop* do usuário e, dessa forma, guiar o usuário a fazer algum procedimento que não deveria ser realizado. Por exemplo, uma *applet* pode apresentar uma janela que se pareça exatamente com uma sessão *telnet* e então capturar a senha do usuário quando este responder a um pedido de informação da senha. Por essa razão, janelas *top-level* são criadas por classes não

confiáveis e possuem um *banner* de identificação. Esse método é diferente de outros no gerenciamento de segurança, pois possui três resultados:

- Retornar verdadeiro, a janela será criada normalmente;
- Retornar falso, a janela será criada com o *banner* de identificação.
- Lançar uma exceção de segurança (assim como todos os outros métodos da classe de gerenciamento de segurança) para indicar que a janela não deve ser criada; a maioria dos aplicativos criados realiza esta ação.

Para que este método retorne verdadeiro, o domínio de proteção corrente deve carregar uma permissão AWT denominada *showWindowWithoutWarningBanner*.

3.3. Aspectos Gerais de Segurança

Estes são os métodos cuidam de alguns aspectos de segurança:

```
public void checkMemberAccess(Class clazz, int which)
```

A API *reflection* é poderosa o bastante a tal ponto que, por inspeção, um programa pode determinar atributos de instância e métodos particulares de uma classe (embora não possa realmente acessar esses atributos ou chamar esses métodos). Todas as classes tem a permissão de inspecionar qualquer outra classe e descobrir seus atributos e métodos públicos. Todas as classes carregadas pelo mesmo *Class Loader* tem a permissão de inspecionar todos os atributos e métodos de outras. De outra forma, o domínio de proteção corrente deve carregar uma permissão em tempo de execução com o nome de *accessDeclaredMembers*. A implementação padrão desse método é bastante frágil. Diferente de todos os outros métodos que foram vistos, é um erro lógico sobrescrever esse método e depois chamar *super.checkMemberAccess*( ).

```
public void checkSecurityAccess(String action)
```

O pacote de segurança depende desse método no gerenciador de segurança para decidir quais classes podem executar certas operações relacionadas a segurança. Como um exemplo, antes de uma classe ter permissão de uma chave particular, esse método é chamado com um argumento do tipo *String* indicando que uma chave particular está sendo lida. Como é esperado, apenas classes confiáveis têm permissão de executar quaisquer operações relacionadas a segurança.

```
public void checkPackageAccess(String pkg)
public void checkPackageDefinition(String pkg)
```

Esses métodos são usados em conjunto com um *Class Loader*. Quando um *Class Loader* é requisitado a carregar uma determinada classe com um nome de pacote em particular, primeiro perguntará ao gerenciador de segurança se é permitido realizar esta ação através da chamada ao método *checkPackageAccess*(). Isso permite ao gerenciador de segurança ter certeza que uma classe não autorizada não está tentando usar classes específicas da aplicação que ela não deveria ter conhecimento. Similarmente, quando um *Class Loader* realmente cria uma classe em um pacote em particular, pergunta ao gerenciador de segurança se é permitido fazê-lo, através da chamada ao método *checkPackageDefinition*(). Isso permite ao gerenciador de segurança prevenir que uma classe não confiável carregue uma classe a partir da rede e armazene-a dentro, por exemplo, do pacote java.lang.

Repare na distinção entre esses dois métodos: no caso do método checkPackageAccess(), a pergunta é se o *Class Loader* pode referenciar a classe como um todo, isto é, se podemos chamar uma classe no pacote da *Sun*. No método *checkPackageDefinition*(), os bytes da classe foram carregados e o gerenciador de segurança está sendo questionado se eles podem pertencer a um pacote em particular. Por padrão, o método *checkPackageDefinition*() nunca é chamado e o método *checkPackageAccess*() é chamado apenas para pacotes listados na propriedade *package.access* dentro do arquivo *java.security*. Para ter sucesso, o domínio de proteção corrente deve ter uma permissão em tempo de execução com o nome de *defineClassInPackage.+<pkg>* ou *accessClassInPackage.+<pkg>*.

4. Construir Gerenciadores Customizados

Para criar seu próprio gerenciador de segurança, é necessário estender a classe *SecurityManager*. Como visto na sessão anterior, a classe *SecurityManager* possui uma série de métodos que podem ser chamados pelas classes Java. Sobrescrevendo esses métodos, pode-se criar políticas de segurança definidas pelo usuário que, quando carregadas em uma aplicação, irão fornecer o nível de segurança desejado.

Há algumas considerações a serem observadas ao se construir um gerenciador de segurança. Aqui estão algumas questões que podem ajudar neste processo:

- Quais métodos devem ser implementados?
- Novos métodos precisam ser adicionados?
- Como os métodos serão implementados?
- Quão restritas as regras de segurança deverão ser?

Um exemplo de um gerenciador de segurança customizado é a classe PasswordSecurityManager, detalhada em:

```
http://java.sun.com/developer/onlineTraining/Programming/JDCBook/signed2.html
```

O gerenciador de segurança customizado para este programa solicita que o usuário entre com sua senha antes de permitir um FileIO para escrever um texto em um arquivo ou ler um texto a partir de um arquivo.

```
Import java.io.*;

public class PasswordSecurityManager extends SecurityManager{
  private String password;
  private BufferedReader buffy;

  public PasswordSecurityManager(String p, BufferedReader b){
    super();
    this.password = p;
    this.buffy = b;
  }
  // Solicitar ao usuário final uma senha, verifica a senha e retorna
  // verdadeiro se a senha estiver correta ou falso se não estiver.
  private boolean accessOK() {
    int c;
    String response;

    System.out.println("Senha, por favor:");
    try {
      response = buffy.readLine();
      if (response.equals(password))
        return true;
      else
        return false;
    } catch (IOException e) {
      return false;
    }
  }
  public void checkRead(String filename) {
    if((filename.equals(File.separatorChar + "home" +
          File.separatorChar + "monicap" +
          File.separatorChar + "text2.txt"))){

      if(!accessOK()){
        super.checkRead(filename);
        throw new SecurityException("Sem Chance!");
      } else {
        FilePermission perm = new FilePermission(
        File.separatorChar + "home" +
```

```
            File.separatorChar + "monicap" +
            File.separatorChar + "text2.txt", "read");
            checkPermission(perm);
          }
        }
      }
      public void checkWrite(String filename) {
        if((filename.equals(File.separatorChar + "home" +
                    File.separatorChar + "monicap" +
                    File.separatorChar + "text.txt"))){
          if(!accessOK()){
            super.checkWrite(filename);
            throw new SecurityException("Sem chance!");
          } else {
            FilePermission perm = new FilePermission(
                      File.separatorChar + "home" +
                      File.separatorChar + "monicap" +
                      File.separatorChar + "text.txt" ,
                      "write");
                  checkPermission(perm);
          }
        }
      }
    }
```

Como visto, os métodos *checkRead* e *checkWrite* foram sobrescritos. A implementação customizada para *SecurityManager.checkWrite* testa o caminho */home/monicap/text.txt*. Se o caminho existir, solicita ao usuário a senha. Se estiver correta, o método *checkWrite* executa a checagem de acesso através da criação de uma instância da permissão requerida e passando-a ao método *SecurityManager.checkPermission*. Essa checagem terá sucesso se o gerenciador de segurança encontrar um sistema, usuário ou arquivo de política de segurança com a permissão especificada.

Ao invés de adicionar a política de segurança no código fonte, pode-se optar por colocá-la em um arquivo separado, como um arquivo *.properties* ou *.xml*. Pode-se, então, solicitar ao gerenciador de segurança a leitura desse arquivo e estabelecer uma política de segurança dessa forma. Essa técnica faz com que a implementação das políticas de segurança sejam flexíveis, já que novas regras adicionadas ao código fonte podem requerer a modificação do código, recompilação e redistribuição da aplicação a cada vez que fossem modificadas.

Por exemplo, em *Enterprise Java Beans* (EJB), políticas de segurança podem ser especificadas no elemento *security-permission* do descritor *weblogic-ejb-jar.xml*. O exemplo seguinte concede acesso de leitura e escrita a um diretório temporário no *filesystem* do servidor para os *EJB*:

```
<weblogic-enterprise-bean>
 <!-- instruções webLogic enterprise bean vão aqui -->
</weblogic-enterprise-bean>
<security-role-assignment>
 <!-- As tarefas opcionais de segurança vão aqui -->
</security-role-assignment>
<security-permission>
 <description>
  concede permissão de leitura e escrita para a pasta ext no drive d:
 </description>
 <security-permission-spec>
  grant {
   permission java.io.FilePermission "d:${/}ext${/}-", "read,write";
  }
 </security-permission-spec>
<security-permission>
```

4.1. Instalando os Gerenciadores de Segurança

Uma vez que as regras de segurança foram estabelecidas e a classe gerenciadora de segurança que irá impor as regras feitas, deste modo, pode-se agora instalar o gerenciador de segurança em

uma aplicação. Isso é feito usando-se o método estático *setSecurityManager* da classe *System*, como apresentado a seguir.

```
try {
      System.setSecurityManager(new SecurityManagerImpl());
} catch (SecurityException se) {
      System.err.println("Gerenciador de Segurança já está instalado!!!");
}
```

Deve-se notar que um gerenciador de segurança pode ser instalado, mas a permissão para instalar um gerenciador diferente pode ser concedida. Assumindo que uma permissão não seja concedida, então uma *SecurityException* será disparada. Depois que o gerenciador de segurança foi instalado, o código da aplicação não precisa referenciar a este diretamente. Isso porque a JVM irá fazer todas as referências ao gerenciador depois da instalação, a menos que uma chamada explícita a um método seja necessária.

Parceiros que tornaram JEDI™ possível

Instituto CTS
Patrocinador do DFJUG.

Sun Microsystems
Fornecimento de servidor de dados para o armazenamento dos vídeo-aulas.

Java Research and Development Center da Universidade das Filipinas
Criador da Iniciativa JEDI™.

DFJUG
Detentor dos direitos do JEDI™ nos países de língua portuguesa.

Banco do Brasil
Disponibilização de seus *telecentros* para abrigar e difundir a Iniciativa JEDI™.

Politec
Suporte e apoio financeiro e logístico a todo o processo.

Borland
Apoio internacional para que possamos alcançar os outros países de língua portuguesa.

Instituto Gaudium/CNBB
Fornecimento da sua infra-estrutura de hardware de seus servidores para que os milhares de alunos possam acessar o material do curso simultaneamente.