Casa do Código

MUNDOJ

MundoJ

Segurança com Java

mini LIVRO

Edição

Adriano Almeida
Vivian Matsui

Revisão

Bianca Hubert
Vivian Matsui

[2017]

Casa do Código

Livros para o programador

Rua Vergueiro, 3185 - 8º andar

04101-300 – Vila Mariana – São Paulo – SP – Brasil

www.casadocodigo.com.br

Sumário

1 Defendendo aplicações web de ataques **1**

1.1 Natureza de uma aplicação web 2

1.2 Alteração de parâmetros 6

1.3 Descobrindo senhas de usuário 11

1.4 SQL Injection 18

1.5 Script Injection e Cross Site Scripting 23

1.6 Cookie Poisoning 27

1.7 Servlet Filters 28

1.8 Considerações finais 29

1.9 Referências 31

2 Evitando acessos automatizados com CAPTCHA **32**

2.1 Teste de Turing 33

2.2 JCaptcha 35

2.3 SimpleCaptcha 38

2.4 Kaptcha 41

2.5 Quebrando o CAPTCHA 43

2.6 Considerações finais 44

2.7 Referências 45

3 Certificação digital com Java Cryptographic Architecture **46**

3.1 A certificação digital 47
3.2 Criptografia 48
3.3 Assinatura digital 53
3.4 Certificado digital 58
3.5 Segurança na plataforma Java 64
3.6 Padrões de assinaturas 70
3.7 Considerações finais 74
3.8 Referências 74

Versão: 20.5.2

Capítulo 1

DEFENDENDO APLICAÇÕES WEB DE ATAQUES

por Eduardo Guerra

A segurança de uma aplicação web é muito mais do que colocá-la atrás de um firewall e configurar o web server para utilizar SSL. Durante a implementação da aplicação, deve-se estar sempre atento para entradas que podem fazer com que a aplicação aja de forma inesperada. Existem ataques que exploram aplicações web obtendo o acesso a dados e funcionalidades aos quais não se possui permissão. Neste capítulo, será visto o funcionamento desses ataques e o que pode ser feito durante o desenvolvimento para garantir a proteção de uma aplicação web.

Segurança é um requisito não funcional muito sério! Existe uma responsabilidade por parte dos desenvolvedores ao implementar uma aplicação web que será disponibilizada na internet. Uma aplicação que trabalha com informações importantes para uma organização não pode ter seus dados corrompidos ou acessados por quem não possui permissão.

Muitos desenvolvedores, provavelmente por desconhecimento, não se dão conta dessa responsabilidade e não dão a devida atenção para a segurança durante a implementação. Da mesma forma, quem é responsável pela segurança da rede, também por desconhecimento, acaba ignorando as aplicações web como um possível ponto vulnerável. Como nenhuma corrente é mais forte que seu elo mais fraco, aquela aplicação web acaba sendo um alvo preferencial de pessoas mal intencionadas que desejam obter informações sigilosas da organização.

A proteção contra esses ataques é algo que normalmente é feito por iniciativa dos desenvolvedores, visto que é incomum encontrar um requisito não funcional como "A aplicação deve barrar tentativas de injeção de SQL". Requisitos não funcionais de segurança normalmente são relativos ao mecanismo de autenticação e de controle de acesso, mas raramente envolvem ataques, mesmo porque quem faz a especificação muitas vezes não sabe da existência deles.

Se todos esses fatores ainda se unirem com um cronograma apertado, tem-se um cenário perfeito para que a segurança da aplicação seja deixada de lado. Então, veremos aqui quais são os principais ataques que podem ser feitos contra uma aplicação web e que mecanismos podem ser criados para barrá-los.

Não será visto como é feita a segurança declarativa pelo *deployment descriptor* (o arquivo `web.xml`) ou como se configura um servidor web para usar o protocolo HTTPS. Apesar de esses aspectos também serem importantes, este capítulo dará mais ênfase a questões de implementação, que normalmente são deixadas de lado, do que a configurações para segurança.

1.1 NATUREZA DE UMA APLICAÇÃO WEB

Grande parte dos problemas de segurança que uma aplicação web pode apresentar tem relação com o seu mecanismo de funcionamento. O funcionamento básico do protocolo HTTP é um cliente, normalmente um navegador, enviar uma requisição a um servidor e este enviar uma resposta com a informação solicitada.

Após isso, nenhuma conexão é mantida entre o cliente e o servidor, de forma que, para o servidor manter informações da sessão do usuário entre as requisições, é preciso que algum identificador seja enviado a cada requisição. Normalmente, é usado um *cookie* ou a reescrita de URL. De posse desse identificador, é possível, de qualquer máquina, entrar dentro da sessão desse usuário sem a necessidade de autenticação.

Em uma aplicação web, o servidor simplesmente responde às requisições recebidas e o navegador é responsável pela exibição da página em formato HTML e pela execução do JavaScript. O que não se pode esquecer é que o navegador é uma variável fora de controle e que os parâmetros enviados pelas requisições podem ser qualquer coisa.

Por exemplo, o parâmetro relativo a um campo de um formulário que limita a 50 o número de caracteres pode chegar ao servidor com uma quantidade muito maior que isso. Esse fato ocorre, pois não se pode garantir o comportamento no navegador, sendo que os parâmetros podem ser alterados livremente por uma outra aplicação, como, por exemplo, usando o **SPIKE Proxy** (veja quadro).

Em relação ao JavaScript, o usuário pode simplesmente optar por não executar e, até mesmo, alterar o código, o que faz com que não seja confiável a utilização de JavaScript para a implementação de qualquer regra de negócio importante. É recomendado usar JavaScript para tornar a interface com o usuário mais interativa e mais responsiva. Porém, se uma validação é feita no cliente para

aumentar o desempenho, a mesma validação deve ser feita no servidor por questões de segurança.

Como se não bastassem esses problemas, ainda existem ataques de força bruta, de injeção de SQL e até mesmo de injeção de JavaScript. O maior problema de todos esses é que, muito provavelmente, eles serão considerados pelo firewall como um acesso legítimo à sua aplicação. A única opção que resta é colocar na própria aplicação mecanismos de segurança para impedir esses ataques, e é justamente o que será mostrado aqui.

SPIKE Proxy

O SPIKE Proxy é um proxy `http` para realização de testes de segurança em aplicações web. Ele armazena as páginas visitadas e permite que, depois, os mesmos requests sejam reenviados, porém com parâmetros modificados.

A ferramenta suporta testes de força bruta, SQL *injection* e *Cross-Site-Scripting*. A versão mais recente do SPIKE Proxy pode ser encontrada na URL:

http://www.immunitysec.com/resources-freesoftware.shtml.

Após o download da aplicação, o arquivo deve ser descompactado no diretório `SPIKEProxy` e, na versão Windows, é executado pelo arquivo `runme.bat` . Depois de iniciado, deve ser configurado o navegador para usar como proxy o IP 127.0.0.1 e a porta 8080. A interface gráfica pode ser acessada a partir da URL http://spike/. Veja a figura a seguir.

Navegando pelos diretórios, pode-se ter acesso às URLs já acessadas e os requests podem ser reescritos a partir de um formulário HTML. Essa ferramenta será utilizada para testar a aplicação dos exemplos mostrados neste capítulo.

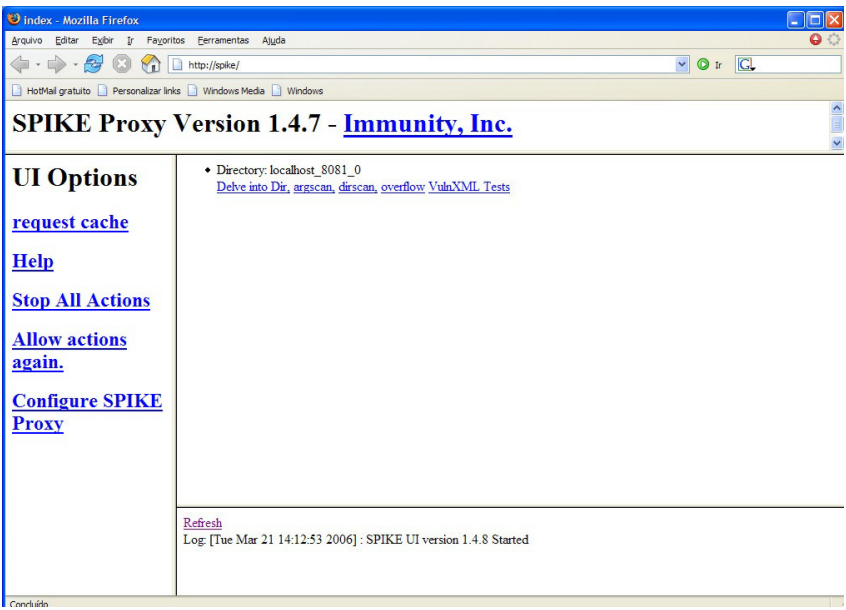

Figura 1.1: Interface do SPIKE Proxy

1.2 ALTERAÇÃO DE PARÂMETROS

A forma mais simples de burlar o funcionamento normal de uma aplicação web é através da alteração de parâmetros de uma forma maliciosa. Com essa alteração, o usuário pode ignorar validações feitas por JavaScript, acessar informações sem possuir permissão, corromper dados e até mesmo contornar regras de negócio. Para alterar os parâmetros de uma requisição, ela não precisa enviá-los via *query string*, usando o método `GET`. Com o método `POST`, os parâmetros podem ser facilmente alterados com o auxílio de uma ferramenta como o SPIKE Proxy.

O controle de permissões declarativo feito por meio do *deployment descriptor* em uma aplicação web define diversos perfis e a autorização de acesso a cada um dos *servlets* do sistema. Um usuário autenticado utilizando a API `JAAS` só pode acessar um servlet caso possua um dos perfis que tenham autorização de acesso

à URL mapeada para aquele recurso.

Na listagem a seguir, pode ser vista parte de um *deployment descriptor* com a configuração de segurança. Observe que estão ressaltadas a definição de um papel e a autorização de acesso desse papel para recursos da aplicação web.

Normalmente, em aplicações que não usam o JAAS para autenticação e controle de permissões, a autorização de acesso costuma ser verificada no nível de funcionalidade (por exemplo, inserir usuário, excluir usuário) ou no nível de recurso (como `listaUsuario.jsp` , servlet mapeado para `criarUsuario.do`). Muitas vezes, esse tipo de controle não é suficiente para as regras de negócio de controle de permissão da aplicação.

Exemplo de parte de um deployment descriptor com a configuração de papéis e permissões:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app …>
    …
    <security-role>
       <role-name>Admin</role-name>
    </security-role>
    <security-constraint>
       <web-resource-collection>
          <web-resource-name>Funcionalidade</web-resource-name>
          <url-pattern>/insere.do</url-pattern>
          <url-pattern>/exclui.do</url-pattern>
          <http-method>POST</http-method>
       </web-resource-collection>
       <auth-constraint>
          <role-name>Admin</role-name>
       </auth-constraint>
    </security-constraInt>
     …
</web-app>
```

Vamos supor uma aplicação de *e-commerce* em que o usuário tem acesso apenas a suas ordens de compra, mas não a de outros usuários. Dessa forma, mesmo o usuário tendo acesso à

funcionalidade de consulta de ordens, não é a todas as ordens que ele possui permissão de acesso. Nesse caso, a permissão deixa de ser no nível da funcionalidade e passa a ser no nível da informação.

Imagine que nessa aplicação o usuário acesse uma página com a listagem de suas ordens e, em cada linha, exista um link para acessar os dados detalhados da ordem. Ao acessá-lo, será feita uma requisição a uma outra página, para a qual será passado como parâmetro o identificador da ordem selecionada.

Se a aplicação não verificar se o usuário logado possui permissão de acesso à ordem que está tentando abrir, será possível o acesso a qualquer ordem do sistema simplesmente alterando o parâmetro passado na URL. Na figura a seguir, está ilustrada a situação descrita neste parágrafo.

O fato de uma ordem não estar listada na tabela não significa que não seja possível obter o acesso alterando o parâmetro. Por mais que isso possa parecer inofensivo nesse caso, imagine que é possível construir uma aplicação que realize diversos acessos a esse sistema, criando relatórios que compilam informações com todas as ordens de sua empresa.

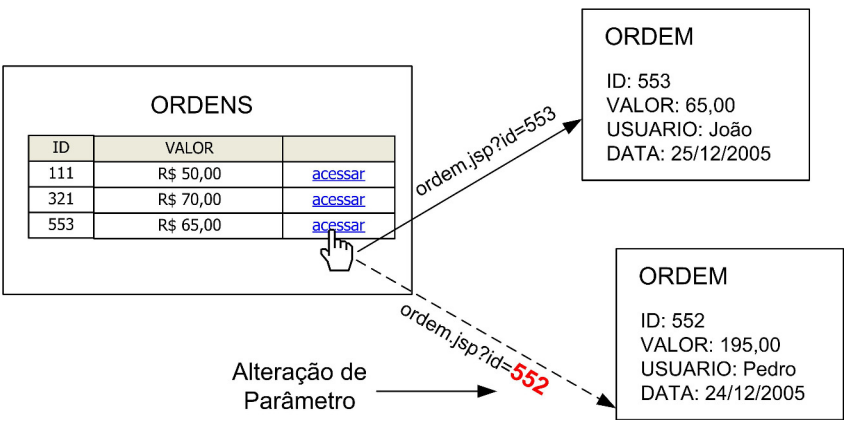

Figura 1.2: Exemplo do uso da alteração de parâmetro para o acesso a uma informação à qual não se tem autorização

Os campos ocultos e desabilitados em uma página HTML normalmente possuem esse comportamento para impedir a sua alteração por parte do usuário. Porém, como já foi dito anteriormente, esses parâmetros podem ser facilmente alterados usando uma ferramenta como o SPIKE Proxy. No desenvolvimento de uma aplicação web, esse fator deve ser levado em consideração.

Por exemplo, imagine que um formulário de edição de informações de usuário armazene o identificador em um campo oculto, e use esse identificador para localizar a entidade no banco de dados que deve ser modificada. Quando o parâmetro que carrega o identificador for alterado maliciosamente e o formulário for submetido, caso a aplicação não possua alguma proteção para esse tipo de situação, podem ser corrompidos dados aos quais o usuário sequer tinha acesso.

Ao localizar a entidade no banco de dados pelo identificador, como esse foi alterado, serão feitas as modificações na entidade errada, o que pode violar diversas regras de negócio da sua aplicação. A figura seguinte ilustra esse exemplo.

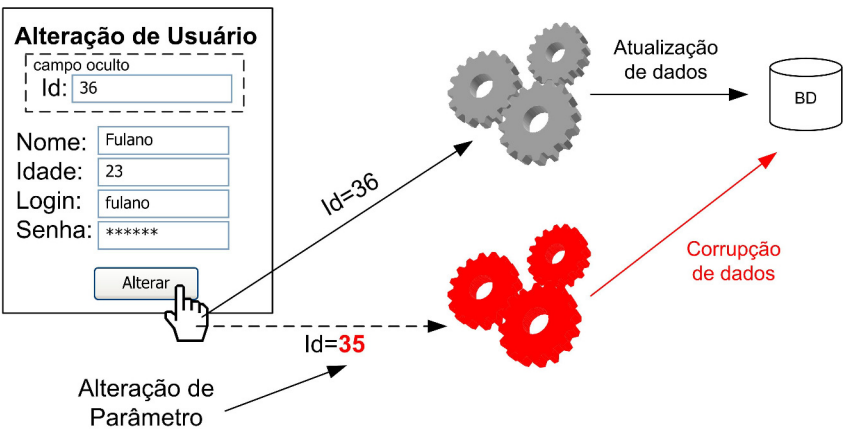

Figura 1.3: Exemplo do uso da alteração de parâmetro para a corrupção de dados

Sabendo da facilidade de alterar qualquer parâmetro da

aplicação, não é confiável implementar qualquer regra de negócio em JavaScript. Um tipo muito comum de regra de negócio implementada em JavaScript é a validação de campos de um formulário.

Essa prática é bastante comum, muitas vezes até por questões de performance, pois o usuário não precisaria ir e voltar ao servidor para dizer que o formato de um dos campos está inválido, ou que um campo não preenchido é obrigatório. Por outro lado, essas validações podem ser completamente ignoradas e dados inválidos podem acabar sendo aceitos pela aplicação.

O Javascript deve ser usado para tornar mais amigável a interação do usuário com o sistema. Nesse caso, é para que não seja necessário esperar as informações irem e voltarem do servidor para que uma simples validação seja realizada. Porém, é preciso validar essas informações também no servidor, para que requisições maliciosas não consigam executar uma funcionalidade com parâmetros inválidos.

A seguir, há uma pequena lista de recomendações para que sua aplicação não se torne vulnerável a uma alteração de parâmetros:

- Valide sempre as permissões do usuário ao acessar uma funcionalidade na qual existam regras de negócio para restringir o acesso do usuário baseado nos dados e não somente no recurso.
- Evite guardar em campos ocultos informações críticas ao funcionamento da aplicação, que o usuário não poderia modificar normalmente, como identificadores, preços e status.
- Caso o uso de campos ocultos seja realmente necessário, faça uma validação que os verifique assim que a requisição chegar ao servidor.
- Valide as informações sempre no servidor, inclusive o

tamanho dos campos, independentemente de existir ou não uma validação no navegador usando JavaScript.
- Não utilize JavaScript para implementar regras de negócio importantes e essenciais para segurança. Se for realmente necessário, lembre-se de replicar a lógica e validar as informações no servidor.

1.3 DESCOBRINDO SENHAS DE USUÁRIO

Um dos objetivos mais comuns de um ataque a uma aplicação costuma ser a obtenção de senhas de acesso. Uma das ideias mais simples de se fazer isso, e que muitas vezes funciona muito bem, é tentar várias possibilidades de senha até acertar.

Isso pode ser feito com a utilização de todas as combinações possíveis, também conhecido como ataque de força bruta, ou a partir de uma lista predefinida de possíveis senhas, o que também é chamado de ataque de dicionário. No ataque de dicionário, o ataque costuma ser direcionado ao usuário, incluindo na lista de possíveis senhas: datas importantes, nomes de familiares, nomes de animais de estimação, filmes favoritos etc.

Esses ataques se baseiam no fato de que é possível realizar várias tentativas falhas de autenticação no sistema. Uma das formas de impedir esse tipo de ataque é evitando que isso seja possível, bloqueando o acesso depois de um número máximo de tentativas. Esse bloqueio, a princípio, poderia ser feito no *firewall*, porém muitas vezes essa não é uma variável sob o controle dos desenvolvedores. Dessa forma, a melhor opção seria criar um mecanismo na própria aplicação que bloqueasse o acesso temporariamente com várias tentativas falhas de autenticação.

Uma forma de inserir essa verificação sem interferir na parte funcional do sistema é com a utilização de um filtro (para maiores

detalhes sobre filtros, veja a seção **Servlet Filters** no final desse artigo). Na próxima listagem, está um filtro que guarda informações das autenticações falhas de usuários no contexto na aplicação web. São parâmetros de configuração, o número máximo de tentativas e o tempo em milissegundos que o usuário fica bloqueado.

Filtro para impedir ataques de força bruta:

```
public class ForcaBrutaFilter implements Filter {

    private long tempoMilisegundos;
    private int numeroTentativas;

    public void init(FilterConfig config)
            throws ServletException {
        tempoMilisegundos = Long.parseLong(
            config.getInitParameter("tempoMilisegundos"));
        numeroTentativas = Integer.parseInt(
            Config.getInitParameter("numeroTentativas"));
    }
    public void destroy() {    }

    public void doFilter(ServletRequest request,
        ServletResponse response,
        FilterChain chain) throws IOException, ServletException {

        ServletContext context = ((HttpServletRequest)request)
            .getSession().getServletContext();

        ForcaBrutaControl control = null;

        if(context.getAttribute("ForcaBrutaControl") == null){
            control = new ForcaBrutaControl(
                tempoMilisegundos,numeroTentativas);
            context.setAttribute("ForcaBrutaCoNtrol", control);
        } else {
            control = (ForcaBrutaControl)
                context.getAttribute("ForcaBrutaControl");
        }

        if(!control.podeLoggar(request)){
            throw new ServletException("Seu IP está bloqueado.");
        }

        try {
            chain.doFilter(request, response);
```

```
        } catch (ServletException e) {
            control.registraFalha(request);
            throw e;
        }
    }
}
```

O filtro primeiramente busca uma instância da classe `ForcaBrutaControl`, apresentada na listagem a seguir, no contexto. E se a instância ainda não existir, ela é criada. Antes de ser chamado o servlet que processará a lógica de login da aplicação, é verificado se o usuário se encontra bloqueado.

Em caso positivo, o acesso é barrado nesse ponto, com o lançamento de uma exceção. Com o usuário não bloqueado, o servlet responsável é chamado e, caso ocorra alguma exceção em sua execução, é registrada uma falha de autenticação.

Filtro para impedir ataques de força bruta:

```
public class ForcaBrutaControl {

    private Map<String, RegistroFalha> tentativaPorIP =
        new HashMap<String, RegistroFalha>();

    private long tempoMilisegundos;

    private int numeroTentativas;

    public ForcaBrutaControl(long tempoMilisegundos,
            int numeroTentativas) {
        this.tempoMilisegundos = tempoMilisegundos;
        this.numeroTentativas = numeroTentativas;
    }

    public void registraFalha(ServletRequest request) {
        if (possuiIP(request) && !isTempoExpirado(request)) {
            RegistroFalha falha = tentativaPorIP.get(
                request.getRemoteHost());
            falha.setTentativa(falha.getTentativa() + 1);
            falha.setUltimaTentativa(System.currentTimeMillis());
        } else {
            RegistroFalha falha = RegistroFalha
                .criarRegistroFalha(request);
```

```
            tentativaPorIP.put(request.getRemoteHost(), falha);
        }
    }

    public boolean podeLoggar(ServletRequest request) {
        return !possuiIP(request) || isTempoExpirado(request)
                || !atingiuMaximoTentativas(request);
    }

    private boolean possuiIP(ServletRequesT request) {
        return tentativaPorIP.containsKey(
            request.getRemoteHost());
    }

    private boolean atingiuMaximoTentativas(
                ServletRequest request) {
        return tentativaPorIP.get(
            request.getRemoteHost()).getTentativa()
                > numeroTentativas;
    }

    private boolean isTempoExpirado(ServletRequest request) {
        return tentativaPorIP.get(
            request.getRemoteHost()).getTempo()
                > tempoMilisegundos;
    }
}
```

A classe mostrada na listagem anterior é que possui realmente a regra de negócios para registrar uma falha e verificar se o usuário não se encontra bloqueado. A classe `ForcaBrutaControl` possui um `Map` para armazenar as tentativas falhas de login, sendo que a chave é o IP de onde o acesso foi realizado, e o objeto é uma instância da classe `RegistroFalha`, representada na próxima listagem. A classe `RegistroFalha` é um *bean* que representa as informações de uma tentativa de autenticação sem sucesso.

Classe que armazena informações sobre falhas de autenticação na aplicação:

```
public class RegistroFalha {

    private String ip;
    private int tentativa;
```

```
    private long ultimaTentativa;

    public static RegistroFalha criarRegistroFalha(
            ServletRequest request) {
        RegistroFalha falha = new RegistroFalha();
        falha.setIp(request.getRemoteHost());
        falha.setTentativa(1);
        falha.setUltimaTentativa(System.currentTimeMillis());
        return falha;
    }

    public String getIp() {
        return ip;
    }
    public void setIp(String ip) {
        this.ip = ip;
    }
    public int getTentatIva() {
        return tentativa;
    }
    public void setTentativa(int tentativa) {
        this.tentativa = tentativa;
    }
    public long getUltimaTentativa() {
        return ultimaTentativa;
    }
    public void setUltimaTentativa(long ultimaTentativa) {
        this.ultimaTentativa = ultimaTentativa;
    }
    public long getTempo(){
        return System.currentTimeMillis() - ultimaTentativa;
    }
}
```

A próxima listagem mostra um exemplo de como configurar o filtro `ForcaBrutaFilter` para executar antes do servlet de autenticação em uma aplicação web.

Exemplo de configuração do filtro da listagem 2 em uma aplicação web:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app …>
    <display-name>AplIcacaoWeb</display-name>
    <servlet>
        <servlet-name>LoginServlet</servlet-name>
        <servlet-class>
```

```
            org.mundojava.servlet.LoginServlet
        </servlet-class>
    </servlet>
    <filter>
        <filter-name>ForcaBrutaFilter</filter-name>
        <filter-class>
            org.mundojava.filter.ForcaBrutaFilter
        </filter-class>
        <init-param>
           <param-name>tempoMilisegundos</param-name>
           <param-value>1200000</param-value>
        </init-param>
        <init-param>
           <param-name>numeroTentativas</param-name>
           <param-value>3</param-value>
        </init-paRam>
    </filter>
    <filter-mapping>
        <filter-name>ForcaBrutaFilter</filter-name>
        <servlet-name>LoginServlet</servlet-name>
    </filter-mapping>
    <servlet-mapping>
        <servlet-name>LoginServlet</servlet-name>
        <url-pattern>/Login.do</url-pattern>
    </servlet-mapping>
</web-app>
```

O exemplo apresentado faz distinção de usuário pelo IP do qual foi originada a requisição. Um filtro parecido poderia ser criado para realizar essa distinção pelo nome de login ou pela senha. Dessa forma, ficará realmente complicado de realizar um ataque de força bruta no sistema, mesmo se ele for sofisticado e alterar maliciosamente o IP de origem da requisição.

Além dessa proteção, existem algumas outras recomendações para dificultar ainda mais ataques desse tipo ao sistema:

- Impor um número mínimo de caracteres na senha é uma boa prática, pois o número de possibilidades que precisam ser testadas aumenta exponencialmente com o número de caracteres da senha.
- Obrigar o usuário a possuir números, letras e caracteres

especiais diminui a possibilidade de ele usar uma senha fraca, o que também reduz o risco de sucesso de um ataque do tipo dicionário.
- Exigir trocas de senha regularmente faz com que o usuário acabe não utilizando a senha da sua aplicação para outros sistemas, e reduz também o risco em ataques de dicionário.

Vale ressaltar que proteções utilizadas em ataques de força bruta podem ser usadas para ataques de negação de serviço (*DoS*). Em outras palavras, o atacante bloqueia de forma proposital usuários para impedir que eles acessem o sistema.

Por esse motivo, alguns sistemas evitam bloquear usuários. Em vez disso, estabelecem um tempo mínimo entre tentativas falhas de login. Dessa forma, além de não abrir brecha para um ataque de negação de serviço, evita os de força bruta nos quais milhares de tentativas são feitas por segundo.

Além desses ataques que exploram sua aplicação, também existem os que exploram o fator humano envolvido na utilização de seu sistema. Nesse caso, tenta-se adquirir a senha tentando se passar por uma pessoa ou um sistema, de forma a induzir o usuário a fornecê-la voluntariamente. A essa prática se dá o nome de **engenharia social**, e a sua utilização se torna mais comum a cada dia.

É frequente, por exemplo, receber um e-mail falso de alguém se passando por um banco, com um link que direciona a uma página que solicita a senha do usuário. Apesar de não estar no escopo deste capítulo, esse tipo de ataque não poderia deixar de ser citado. O investimento na educação dos usuários é essencial para que o risco desse tipo de ataque diminua.

1.4 SQL INJECTION

Esse não é um ataque exclusivo de aplicações web, e qualquer aplicação na qual os comandos SQL são construídos com concatenação de strings pode sofrer esse ataque. O SQL Injection consiste no envio de parâmetros de forma a alterar as instruções SQL, fazendo a aplicação se comportar de forma inesperada.

Os parâmetros são enviados de forma que, ao serem concatenados com uma instrução SQL, resultem em um comando que executará uma instrução maliciosa no banco de dados. A figura a seeguir representa o funcionamento de uma aplicação que usa a concatenação de strings para a formação de um comando SQL para autenticação de um usuário. O *container* web recebe a requisição com os parâmetros, a aplicação os concatena com a instrução SQL, e essa é enviada para execução no banco de dados.

Na figura adiante, tem-se o exemplo de SQL Injection na mesma aplicação. O parâmetro enviado como login (que pode ser visto na figura *Representação do uso de SQL Injection para obter a autenticação em uma aplicação*) primeiramente fecha as aspas que delimitaria o parâmetro. Depois acrescenta uma instrução condicional que torna a cláusula `WHERE` sempre verdadeira e, por fim, adiciona dois traços para comentar o resto do comando original. Com isso, consegue-se a autenticação na aplicação sem identificador e senha válidos.

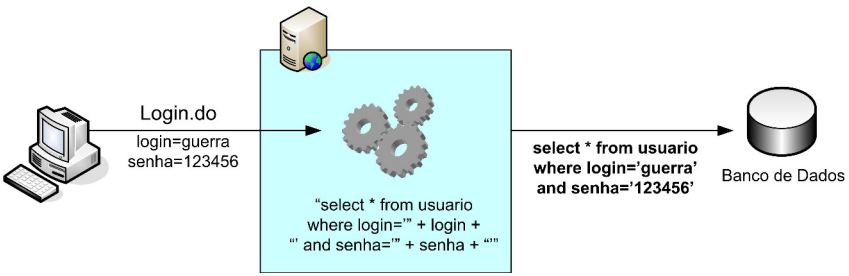

Figura 1.4: Representação de uma aplicação web que usa a concatenação de strings para formar comandos SQL

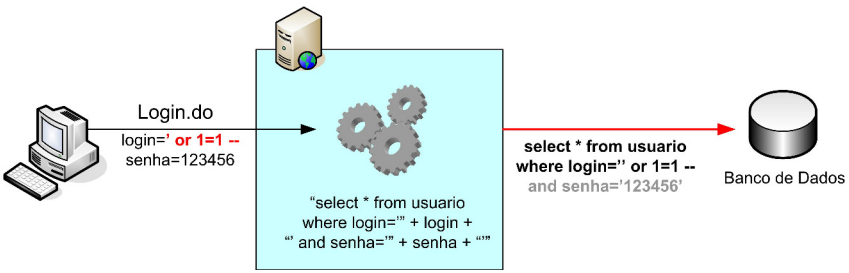

Figura 1.5: Representação do uso de SQL Injection para obter a autenticação em uma aplicação

O tamanho do estrago que pode ser feito por um ataque desse tipo depende bastante do banco de dados e do driver JDBC usado. Existem drivers que, por exemplo, não aceitam que sejam enviadas duas instruções SQL separadas por ponto e vírgula. Outros não aceitam que sejam inseridos dois traços para comentar parte do comando.

O banco de dados utilizado também pode influenciar nos danos causados pelo ataque. O uso do SQL Server, por exemplo, por esse possuir *stored procedures* que acessam diretamente funcionalidades do sistema operacional, aumenta o potencial de danos causados por um ataque desses. O SQL Injection pode ser usado de diversas formas e com objetivos distintos. Nas *Referências*, existe um artigo que detalha bastante as várias formas de SQL Injection.

A seguir, está apresentada uma breve lista de diferentes

maneiras de se fazer o SQL Injection, lembrando de que alguns dos itens podem não funcionar em virtude do banco de dados e do driver JDBC utilizado:

- Inserir uma parte do SQL dentro de um parâmetro de forma a criar uma condicional que seja sempre verdadeira para que uma autenticação seja realizada com sucesso.
- Completar comando SQL da forma desejada e comentar o restante.
- Colocar ponto e vírgula, e inserir um novo comando adicional como criar ou excluir uma tabela.
- Inserir uma cláusula `UNION` para que, em uma consulta, sejam retornados resultados adicionais aos da consulta original, como a lista de usuários e senha.

A utilização do mecanismo da classe `PreparedStatement` para a inserção dos parâmetros resolve esse problema, pois os caracteres são codificados de forma a não serem interpretados como parte do comando SQL. A listagem a seguir mostra um trecho de código de como a mesma instrução SQL poderia ser criada usando `Statement` (mais vulnerável a SQL Injection) e `PreparedStatement` (com a tradução de caracteres especiais).

Diferentes abordagens para execução de uma consulta com JDBC:

```
//Com Statement
String query = "select * from USUARIO where LOGIN='"+login+
    "' and SENHA='"+senha+"' "
Statement stmt = c.createStatement();
ResultSet rSet = stmt.executeQuery(query);

//Com PreparedStatement
String query = “select * from USUARIO where LOGIN=? and SENHA=? "
PreparedStatement  stmt = con.prepareStatement(updateString);
stmt.setString(1, login);
stmt.setString(2, senha);
```

```
ResultSet rSet = stmt.executeQuery();
```

Os frameworks de acesso a dados, como o Hibernate e o iBatis, e APIs como o JPA também eliminam o risco de ataques utilizando SQL Injection. Porém, deve-se ter em mente que, algumas vezes, surgem consultas com vários parâmetros que devem ser inseridos condicionalmente e a concatenação de strings acaba sendo a melhor alternativa para a sua criação. E, mesmo tendo grande parte da aplicação protegida, se não houver o devido cuidado, essa consulta em que as strings são concatenadas pode acabar se tornando um ponto vulnerável do sistema.

Mesmo com a utilização de frameworks e APIs que protegem a aplicação desse tipo de ataque, é interessante saber quando alguém está tentando realizar um ataque à sua aplicação. Na listagem a seguir, está um filtro que procura indícios de uma tentativa de SQL Injection em todos os parâmetros de uma requisição e lança uma exceção quando encontra algo.

Apesar de poder haver falsos positivos (ou seja, serem identificados ataques que não realmente o são), esse filtro já é uma pequena dica de como esse tipo de ataque pode ser identificado sem que essa verificação se misture com a lógica de negócios. São filtrados os caracteres aspas simples (usados na delimitação de parâmetros em formato texto), dois traços (que indicam o início de um comentário), e o ponto e vírgula (utilizado para a separação de dois comandos).

Filtro que busca indícios de SQL Injection nos parâmetros de uma aplicação web:

```
public class SQLInjectionFilter implements Filter{

    public void init(FilterConfig arg0) throws ServletException{}
    public void destroy() {}

    public void doFilter(ServletRequest request,
            ServletResponse response, FilterChain chain)
```

```
            throws IOException, ServletException {

        for(Object paramName : request.getParameterMap()
                                    .keySet()){
            String[ ] params =
                (String[ ])request.getParameterMap()
                .get(paramName);
            for(String param : params)
                if(param.indexOf("'")>=0
                    || param.indexOf("--")>=0
                    || param.indexOf(";")>=0){
                    throw new ServletException("SQL Injection");
                }
        }
        chain.doFilter(request, response);
    }
}
```

Segue uma lista de recomendações que ajudam a evitar o SQL Injection e a diminuir os danos caso um ataque ocorra:

- Utilizar a classe `PreparedStatement` ou frameworks de persistência que já fazem a tradução de caracteres-chave para que eles não sejam considerados parte do comando SQL.
- Filtrar e validar os parâmetros no servidor na chegada das requisições para rejeitá-la ou eliminar o risco de SQL Injection.
- Alterar algoritmo de autenticação de forma a recuperar somente a senha do usuário, e depois compará-la com a senha recebida como parâmetro.
- Não exibir na tela mensagens de exceções vindas diretamente do banco de dados, pois essas podem dar pistas de como está a estrutura do banco e qual o comando que está sendo executado.
- Não usar a senha de administrador para acessar o banco de dados, pois o atacante poderá ter acesso irrestrito às informações.
- Gravar as senhas de usuário no banco de dados

utilizando um algoritmo de *hashing*, como o SHA1 ou MD5, pois mesmo se um atacante obtiver acesso a essa informação, as senhas estarão protegidas (o terceiro capítulo deste livro mostra como gerar o hash de uma `String` em Java).

1.5 SCRIPT INJECTION E CROSS SITE SCRIPTING

Esse tipo de ataque explora a forma como o conteúdo HTML é gerado pela aplicação e como ele é interpretado pelo navegador. O atacante envia na requisição para a aplicação web parâmetros com pedaços de HTML e scripts embutidos. Com isso, é possível executar código no browser do usuário da aplicação, sem que ele tenha conhecimento. Uma das possíveis consequências é o sequestro de sessão, que é realizado através da obtenção do id de sessão pelo atacante, por meio do qual o atacante consegue acessar a aplicação usando a sessão do outro usuário.

Na figura adiante, pode ser vista uma representação de um ataque que usa o *Script Injection* para alterar o comportamento de uma página no navegador dos usuários. Em algum momento da aplicação, imagine que um atacante insira na base de dados um texto que será visualizado por outros usuários em uma página HTML.

No exemplo mostrado na figura, esse texto é exibido em um alerta quando um determinado botão é pressionado. Enviando um texto malicioso, o atacante modifica o JavaScript que era para ser executado originalmente e consegue incluir outros comandos para serem executados. No exemplo, o código injetado provoca a execução de outros alertas, incluindo um que exibe o cookie com o identificador de sessão do usuário.

Da mesma forma como o valor do identificador da sessão pode ser exibido na tela, ele também pode ser enviado para um outro site controlado pelo atacante, o que é conhecido como *Cross Site Scripting*. Substituindo o valor do cookie `JSESSIONID` em uma requisição, pode-se entrar na sessão desse usuário. Essa prática é conhecida como sequestro de sessão e faz com que a aplicação reconheça o atacante como um usuário já autenticado.

Existem inúmeras formas de se injetar código em uma página HTML fazendo com que ela se comporte de forma inesperada. No caso de um texto que será simplesmente exibido na página, seria possível injetar tags HTML com figuras, elementos de *script* e até mesmo *applets*. Existe um artigo nas *Referências* que oferece mais detalhes de como esse tipo de ataque pode ser realizado.

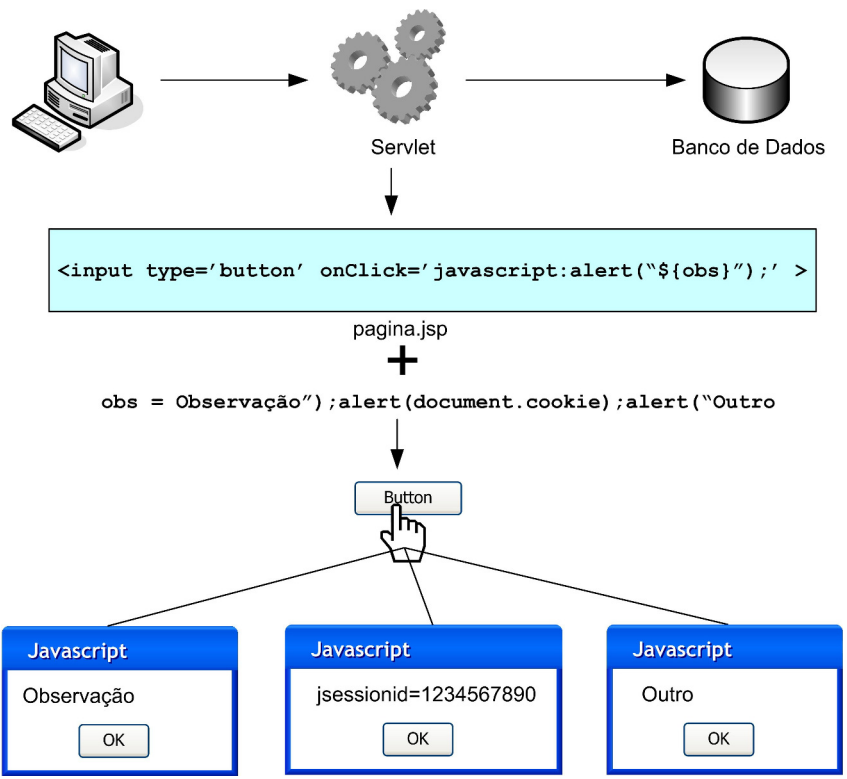

Figura 1.6: Representação do uso de SQL Injection para obter a autenticação em uma aplicação

Para prevenir que essa vulnerabilidade seja explorada em uma aplicação, existem basicamente duas abordagens. A primeira é bloquear qualquer requisição com algum parâmetro de que se suspeite que possua script injetado. A implementação desse bloqueio pode ser feita com um filtro equivalente ao usado para bloquear ataques de SQL Injection mostrado na seção anterior.

A única mudança seria os caracteres procurados, que neste caso incluiriam `<` e `>`, por exemplo. A desvantagem dessa abordagem é a possibilidade de detecção de falsos positivos.

A segunda abordagem seria trocar os caracteres-chave por entidades HTML, de forma que o conteúdo da variável seja sempre considerado texto, e nunca uma tag ou parte de um script. A desvantagem é que não seria possível identificar a tentativa de um ataque, e esse seria simplesmente evitado.

Para usar essa segunda maneira, uma opção é a utilização da tag `<out>` da "Core" library do JSTL. Essa tag possui a função de simplesmente imprimir um valor na página. Porém, com o atributo `escapeXML` igual a `true`, é feita a tradução dos caracteres especiais. No exemplo da figura anterior, poderíamos substituir o uso da EL `${obs}` por:

```
<c:out escapeXml="true" value='${obs}'></c:out>
```

O sequestro de sessão é uma das piores consequências dos ataques de *Script Injection*. Mesmo utilizando técnicas para evitar a injeção de scripts, ainda é possível sequestrar a sessão capturando o identificador de alguma outra forma, como interceptando um pacote na rede ou invadindo a máquina do usuário.

Uma prática de segurança usada em algumas aplicações é a implementação de um mecanismo dentro da aplicação que impeça o uso da mesma sessão por dois IPs diferentes. Com isso, caso o identificador da sessão seja capturado, na tentativa de se fazer o

acesso a partir de uma outra máquina, a sessão será encerrada e o acesso será bloqueado.

O problema dessa abordagem, que a torna nem sempre adequada, é que, em redes móveis de operadoras de celular e ao mudar de rede sem fio, o IP pode ser alterado, e um acesso legítimo seria considerado inválido, gerando a necessidade de o usuário realizar novamente a autenticação.

Na listagem a seguir temos o exemplo de um filtro que realiza essa tarefa. Inicialmente, o filtro recupera o objeto que representa a sessão do usuário, porém, sem criar uma nova, caso essa ainda não exista. Na sessão, é armazenado um atributo `host` que contém o IP de onde foi feito o primeiro acesso.

No caso de o atributo `host` ser diferente da propriedade `remoteHost` do request, isso significa que aquela sessão está sendo acessada de uma máquina diferente de onde ela foi criada. Quando isso acontecer, será invalidada e lançada uma exceção. Se a sessão for criada durante o processamento da requisição, no final do filtro é inserido o atributo `host` para armazenar o IP para o qual ela foi criada.

Filtro que impede o sequestro de sessão não deixando que uma mesma sessão seja acessada de um IP diferente.

```
public class SessionHijackingDetectionFilter implements Filter {

    public void init(FilterConfig arg0) throws ServletException {
    }
    public void destroy() {
    }

    public void doFilter(ServletRequest request,
            ServletResponse response, FilterChain chain)
            throws IOException, ServletException {

        HttpSession session = ((HttpServletRequest)request)
            .getSession(false);
```

```
        if(session != null){
            String sessionHost =
                (String)session.getAttribute("host");
            String requestHost = request.getRemoteHost();
            if(!sessionHost.equals(requestHost)){
                session.invalidate();
                throw new ServletException(
                    "Possível sequestro de sessão!");
            }
        }

        chain.doFilter(request, response);

        session= ((HttpServletRequest)request).getSession(false);

        if(session != null
            && session.getAttribute("host") == null){

            String requestHost = request.getRemoteHost();
            Session.setAttribute("host",requestHost);
        }
    }
}
```

1.6 COOKIE POISONING

Como foi visto nos ataques anteriores, grande parte deles é feita a partir da alteração de parâmetros de uma requisição à aplicação web. Como esses parâmetros estão sob o controle do usuário, ele pode modificá-los de forma maliciosa visando explorar alguma vulnerabilidade que exista na aplicação.

Não podemos esquecer que os cookies também são um tipo de parâmetro que é enviado com a requisição e que estão sob o controle do usuário. O *Cookie Poisoning* consiste na alteração maliciosa de cookies, que podem visar à execução de qualquer um dos ataques anteriores, como o SQL Injection, por exemplo. Portanto, os mesmos cuidados que sua aplicação possuir com parâmetros do request ela também deve ter com as informações contidas nos cookies.

1.7 SERVLET FILTERS

Um filtro na API de Servlets é um componente que intercepta uma requisição, e pode executar comandos antes e depois de ela ser tratada por um Servlet. Um filtro precisa implementar a interface `Filter`, que possui os métodos `init()`, `destroy()` e `doFilter()`. O método `doFilter()`, que é chamado quando uma requisição é interceptada, possui um parâmetro do tipo `FilterChain`, que representa a cadeia de filtros que está processando esta requisição.

Vários filtros podem ser executados em sequência em uma mesma requisição, conforme mostrado no diagrama da figura adiante. Ao receber uma requisição, o filtro pode fazer algum pré-processamento e, em seguida, ao chamar o método `doFilter()` da instância da classe `FilterChain`, invocar o próximo filtro ou o Servlet que vai tratá-la. Quando esse método `doFilter()` retorna, o filtro retoma o controle e pode realizar algum pós-processamento na requisição.

O uso de filtros é uma opção interessante quando se está implementando uma aplicação web. Características não-funcionais, ou seja, que não são diretamente relacionadas com regras de negócio, são excelente candidatas para serem implementadas dentro de filtros. Dentro dessa categoria, estão incluídas verificações de segurança para procurar parâmetros maliciosos em uma requisição.

Neste capítulo, foram vistos exemplos de filtros que fazem verificações de segurança e como podem ser configurados para interceptar a execução de um Servlet no *deployment descriptor* de uma aplicação web. Segue uma lista de vantagens do uso de filtros:

- Desacopla os aspectos não funcionais das regras de negócio.
- Evita duplicação de código, visto que um filtro pode

interceptar requisições de diversos Servlets.

- Pode ser reutilizado em várias aplicações, pois sua execução é transparente para os Servlets.
- Pode ser habilitado e desabilitado com configurações no *deployment descriptor*.

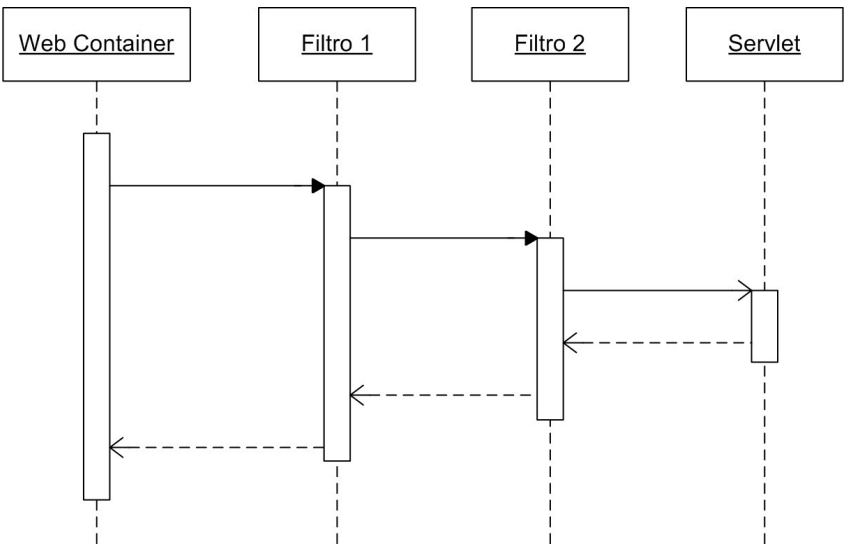

Figura 1.7: Diagrama de sequência demonstrando como uma cadeia de filtros é chamada antes de um Servlet

1.8 CONSIDERAÇÕES FINAIS

Gostaria muito de poder terminar dizendo que, com essas recomendações, sua aplicação web estará totalmente segura. Infelizmente, isso não é verdade, pois quem protege uma aplicação deve se preocupar em tapar todos os buracos. Porém, quem faz o ataque só precisa descobrir um.

Neste capítulo, foram mostradas várias técnicas para eliminar os principais tipos de vulnerabilidade que podem existir em uma aplicação web. O que posso afirmar é que, com certeza, com esses cuidados você aumentará bastante o nível de segurança da sua

aplicação. Segue uma lista com exemplos de outros fatores que devem ser levados em consideração para a segurança de uma aplicação web:

- Segurança da rede em que a aplicação está localizada, bloqueando portas de administração, por exemplo.
- Modelo de controle de acesso que permita bloquear o acesso às informações às quais um usuário não possuir permissão (observar a possibilidade da alteração de parâmetros).
- Comunicação com outras aplicações ou camadas remotas, como uma camada EJB ou o acesso a um Web Service.
- Gerenciamento das exceções de segurança que forem lançadas na aplicação, para que se possa ter conhecimento dos ataques que estão sendo feitos contra sua aplicação.
- Auditoria das ações dos usuários do sistema, tornando possível rastrear ações de usuários e encontrar responsáveis por ações indevidas que forem executadas no sistema.

Além de questões relativas à segurança, também foi visto como isolar o tratamento de segurança do resto da aplicação. Como esses problemas com segurança de aplicações web são recorrentes, é muito bom poder isolar a lógica de detecção de ataques para que ela possa ser reutilizada em várias aplicações.

Neste capítulo, usamos os filtros da API de Servlets para isolar esse tipo de funcionalidade. Porém, pode-se utilizar qualquer tipo de componente que intercepte requisições, como os *Interceptors* do *Struts*, ou até mesmo um aspecto. Isolando as regras de negócio do código que faz o tratamento de segurança, é possível obter não só uma aplicação mais segura, como também uma aplicação com uma

modelagem flexível e de qualidade.

1.9 REFERÊNCIAS

TAYLOR, Art; BUEGE, Brian; LAYMAN, Randy. *Segurança contra Hackers:* J2EE e Java. Editora Futura, 2003.

SCAMBRAY, Joel; SHEMA, Mike. *Segurança contra Hackers:* Aplicações Web. Editora Futura, 2003.

FORRISTAL, Jeff. *Site Seguro – Aplicações Web*. Alta Books, 2002.

Capítulo 2

EVITANDO ACESSOS AUTOMATIZADOS COM CAPTCHA

por Mateus Bahia e Rodrigo Cunha de Paiva

CAPTCHA (Completely Automated Public Turing test to tell Computers and Humans Apart) é um recurso utilizado para determinar se um usuário é humano ou não. A sua implementação mais conhecida são imagens distorcidas em que o usuário precisa lê-la e digitá-la para prosseguir com a operação. É um recurso cada vez mais usado em sites da internet e tem por principal objetivo evitar que ferramentas automatizadas acessem um determinado recurso do site. O objetivo deste capítulo é apresentar o conceito por trás de CAPTCHA, e também mostrar algumas alternativas de código aberto para implementá-lo em aplicações Java.

A internet é um ambiente bastante hostil. Ataques a um determinado site vêm de todos os lados, com armas cada vez mais poderosas, na tentativa de degradar performance, comprometer dados, derrubar servidores etc. Você pode se proteger com *patches* de segurança, validação de campos, criptografia e outras técnicas,

mas sempre existem novos tipos de ataques e é necessário estar preparado.

Uma das principais ferramentas de ataque é o poder computacional. A velocidade com que um computador pode fazer milhares de requisições para milhares de sites é incrível. Frequentemente, isso é usado ofensivamente, mas se usado com moderação, pode não parecer um ataque ao site. Um exemplo clássico ocorre com sites que oferecem endereço de e-mail gratuito. É possível construir um programa que registre milhares de contas de e-mail, com objetivo de fazer spam.

Para esses sites, é difícil, se não impossível, diferenciar um browser controlado por um ser humano ou por um programa. Qual a solução? Como um site pode diferenciar requisições iniciadas por humanos de requisições geradas por um programa?

Neste capítulo, você vai conhecer a teoria sobre o CAPTCHA, a melhor forma para resolver o problema apresentado. Além disso, você vai saber por que ele foi criado e algumas alternativas para implementá-lo em Java, usando JCaptcha, SimpleCaptcha e Kaptcha. No final, vamos apresentar algumas técnicas que podem ser usadas na tentativa de se quebrar um CAPTCHA.

2.1 TESTE DE TURING

Alan Turing apresentou a ideia para máquinas de Turing. Ele acreditava que um dia computadores teriam inteligência comparáveis à humana. Ele propôs um jogo de imitação. Coloque um computador em uma sala, e um humano em outra, ambos separados por um interrogador humano (que não sabe quem está em cada sala). Este interrogador faz perguntas para ambos e, quando satisfeito, ele adivinha em qual sala está o computador e em qual sala está o humano.

Se o interrogador não consegue acertar a resposta correta mais da metade das vezes, o computador passa no teste e é considerado tão inteligente quanto o ser humano. O Teste de Turing é baseado no problema de um humano diferenciar um computador de outro humano. No nosso problema, nós precisamos do contrário. Um computador precisa diferenciar entre um humano e outro computador, o que é muito mais complexo. Este cenário é conhecido como **Teste de Turing Reverso**.

Para resolver esse problema, em 2000, cientistas criaram o CAPTCHA. CAPTCHA é um acrônimo da expressão *Completely Automated Public Turing test to tell Computers and Humans Apart* (Teste público de Turing completamente automatizado para separar computadores e humanos). É um tipo de teste "desafio-resposta" usado para garantir que a resposta não é gerada por computador.

Como outros computadores são incapazes de gerar a resposta, qualquer usuário que entre com a solução correta é presumidamente humano. Um tipo comum de CAPTCHA requer que o usuário identifique as letras de uma imagem distorcida. Mas existem outras opções, incluindo sons (usados para deficientes visuais), reconhecimento de animais ou até de rostos humanos.

As figuras seguintes mostram alguns exemplos de CAPTCHA. A primeira figura é um tipo mais complexo de CAPTCHA em que o usuário precisa identificar o que está sendo mostrado na imagem. A segunda figura é um tipo mais comum, no qual o usuário precisa identificar as letras que estão exibidas na imagem. Nas próximas seções, serão apresentadas algumas ferramentas para se implementar CAPTCHA em Java.

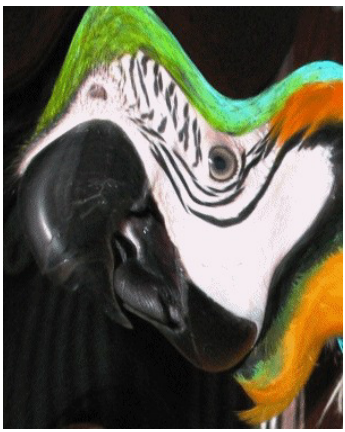

Figura 2.1: Exemplo de CAPTCHA mais complexo: Qual o tipo de animal?

Figura 2.2: Exemplo de CAPTCHA mais comum: Digite os caracteres exibidos

2.2 JCAPTCHA

Um dos frameworks Java mais tradicionais para implementação de CAPTCHA é o JCaptcha, distribuído sob *GNU Lesser General Public License*. Uma grande vantagem dele é ser *multi-type challenge*, o que significa que ele pode gerar desafios com imagens, textos ou sons. Isso é um diferencial se você estiver implementando um site largamente usado, ou um site em que você precisa se preocupar com acessibilidade e deficientes visuais estariam impossibilitados de passar no teste das imagens.

Veja os passos necessários para integrar o JCaptcha à sua aplicação:

Passo 1: adicione `jcaptcha-all.jar` e `commons-collection-3.2` ou superior no diretório `WEB-INF/lib` do seu

site.

Passo 2: implemente o `CaptchaService` (precisa ser um *singleton*), mostrado na próxima listagem na classe `CaptchaServiceSingleton`.

Implementação do `CaptchaService` para o JCaptcha:

```
import com.octo.captcha.service.image.ImageCaptchaService;
import com.octo.captcha.service.image
    .DefaultManageableImageCaptchaService;

public class CaptchaServiceSingleton {

    private static ImageCaptchaService instance =
        new DefaultManageableImageCaptchaService();

    public static ImageCaptchaService getInstance(){
        return instance;
    }
}
```

Passo 3: implemente o servlet gerador da imagem do CAPTCHA, mostrado na listagem seguinte no método `doGet()`.

Implementação do `CaptchaService` para o JCaptcha:

```
protected void doGet(HttpServletRequest httpServletRequest,
            HttpServletResponse httpServletResponse)
            throws ServletException, IOException {
    byte[] captchaChallengeAsJpeg = null;
    // saída do stream para escrever a imagem captcha dentro
    ByteArrayOutputStream jpegOutputStream =
        new ByteArrayOutputStream();
    try {

        //Obtém Id da sessão que vai identificar o captcha gerado
        //O mesmo id precisa ser usado para validar a resposta
        String captchaId = httpServletRequest.
                getSession().getId();

        // Chama o método ImageCaptchaService getChallenge
        BufferedImage challenge = CaptchaServiceSingleton
            .getInstance().getImageChallengeForID(captchaId,
                httpServletRequest.getLocale());
```

```
            // jpeg encoder
            JPEGImageEncoder jpegEncoder = JPEGCodec
                .createJPEGEncoder(jpegOutputStream);
            jpegEncoder.encode(challenge);
        } catch (IllegalArgumentException e) {
            httpServletResponse.sendError(
                HttpServletResponse.SC_NOT_FOUND);
            return;
        } catch (CaptchaServiceException e) {
            httpServletResponse.sendError(
                HttpServletResponse.SC_INTERNAL_SERVER_ERROR);
            return;
        }

        captchaChallengeAsJpeg = jpegOutputStream.toByteArray();

        // resposta da requisição
        httpServletResponse.setHeader("Cache-Control", "no-store");
        httpServletResponse.setHeader("Pragma", "no-cache");
        httpServletResponse.setDateHeader("Expires", 0);
        httpServletResponse.setContentType("image/jpeg");
        ServletOutputStream responseOutputStream =
            httpServletResponse.getOutputStream();
        responseOutputStream.write(captchaChallengeAsJpeg);
        responseOutputStream.flush();
        responseOutputStream.close();
    }
```

Passo 4: adicione no `web.xml` a referência ao servlet criado anteriormente, como mostrado na listagem seguinte.

Alteração do `web.xml` para o JCaptcha:

```
<servlet>
    <servlet-name>jcaptcha</servlet-name>
    <servlet-class>ImageCaptchaServlet</servlet-class>
    <load-on-startup>0</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>jcaptcha</servlet-name>
    <url-pattern>/jcaptcha</url-pattern>
</servlet-mapping>
```

Passo 5: crie o formulário contendo o desafio, como na listagem seguinte.

Formulário com o desafio para o JCaptcha:

```
<img src="/jcaptcha">
<input type='text' name='resposta_captcha' value=''>
```

Passo 6: crie a rotina de validação, mostrada na listagem a seguir.

Rotina de validação para o JCaptcha:

```
Boolean respostaCorreta = Boolean.FALSE;

//Lembre-se que usamos um ID, no servlet?
//Aqui usaremos o mesmo para recuperar.
String captchaId = httpServletRequest.getSession().getId();

//Recupera a resposta
String response =
    httpServletRequest.getParameter("resposta_captcha");

// Chama o serviço
try {
    respostaCorreta = CaptchaServiceSingleton.getInstance()
        .validateResponseForID(captchaId, response);
} catch (CaptchaServiceException e) {
    //Este erro não deve ocorrer,
    //a menos que você tenha usado um id inválido
}
```

A figura a seguir apresenta exemplos de imagens geradas pelo JCaptcha.

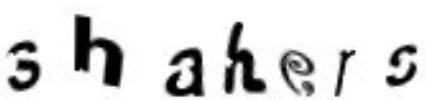

Figura 2.3: Exemplos de imagens geradas pelo JCaptcha

2.3 SIMPLECAPTCHA

O SimpleCaptcha é uma implementação Java open source (distribuído sob *BSD License*), com o propósito de gerar bons

CAPTCHAs com o mínimo de programação e configuração. A implementação padrão permite configurar cores do background, fontes, bordas, geradores de caracteres, entre outras opções.

A configuração é feita por meio do `web.xml` . Todos os valores podem ser deixados em branco, e a implementação padrão será usada. O servlet do CAPTCHA colocará na sessão uma chave que pode ser obtida por um *controller*.

Basicamente, o que você precisa fazer é configurar um *Servlet Mapping* no `web.xml` , e pronto. O servlet mapeado é responsável por receber a requisição, colocar o valor do CAPTCHA na sessão e retornar a imagem para o browser.

Para integração com sua aplicação, basta seguir os passos adiante:

Passo 1: copie o `simplecaptcha.jar` dentro do `WEB-INF/lib` de sua webapp.

Passo 2: crie um mapeamento de servlet no `web.xml` , como na listagem a seguir.

Alteração a ser feita no `web.xml` para o SimpleCatpcha:

```
<servlet>
    <servlet-name>jcaptcha</servlet-name>
    <servlet-class>ImageCaptchaServlet</servlet-class>
    <load-on-startup>0</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>jcaptcha</servlet-name>
    <url-pattern>/jcaptcha</url-pattern>
</servlet-mapping>
```

Passo 3: no seu `jsp` , crie uma tag de imagem e uma caixa de texto para obter a entrada do usuário, como no trecho mostrado a seguir.

Imagem e caixa de texto no JSP para o SimpleCaptcha:

```
<img src="Captcha.jpg">  <input type=”text” name=”captcha”/>
```

Passo 4: no seu *controller*, compare o valor inserido pelo usuário com o valor da chave de CAPTCHA armazenada na sessão, como na listagem a seguir.

Rotina de validação para o JCaptcha:

```
String kaptchaExpected = (String)session.getAttribute(
    nl.captcha.servlet.Constants.SIMPLE_CAPCHA_SESSION_KEY);
String kaptchaReceived = request.getParameter("kaptcha");
if (kaptchaReceived == null ||
        !kaptchaReceived.equalsIgnoreCase(kaptchaExpected)) {
    setError("kaptcha", "Invalid validation code.");
}
```

Pronto. Agora faça o tratamento enviando o usuário para a página de sucesso, ou mostrando uma mensagem de erro.

Você ainda pode fazer diversas configurações no `web.xml` , tais como cor, borda, fonte, tamanho da imagem etc. Acesse o site do projeto (vide referências) para mais detalhes sobre como fazer essas configurações. A figura seguinte mostra alguns exemplos de CAPTCHAs gerados pelo SimpleCaptcha.

Figura 2.4: Exemplos de CAPTCHAs gerados pelo SimpleCaptcha

Uma observação importante sobre o SimpleCaptcha é que o projeto foi descontinuado e não está mais sendo mantido. Além disso, ele só funciona para JDK 1.6. Mesmo sendo muito simples de

usar, esses fatores podem inviabilizar o seu uso.

2.4 KAPTCHA

O Kaptcha é uma versão moderna do projeto SimpleCaptcha, distribuído sob *Apache License 2.0*. O uso básico do Kaptcha na sua webapp é muito simples e muito parecido com o SimpleCaptcha. Tudo o que você precisa fazer é adicionar o `jar` para seu projeto, fazer referência ao `kaptcha servlet` no seu `web.xml` , e comparar o valor CAPTCHA gerado na sessão com o que o usuário submeteu no seu formulário.

> **Nota**
>
> Se estiver usando JCaptcha na mesma aplicação que o Kaptcha, você terá conflitos com a biblioteca de geração de imagem.

Veja a seguir os detalhes de como integrar o Kaptcha à sua aplicação:

Passo 1: coloque o `.jar file` no diretório `WEB-INF/lib` da sua aplicação.

Passo 2: coloque a referência do servlet no seu `web.xml` , como na listagem a seguir.

Alteração a ser feita no `web.xml` para o Kaptcha:

```
<servlet>
    <servlet-name>Kaptcha</servlet-name>
    <servlet-class>
        com.google.code.kaptcha.servlet.KaptchaServlet
    </servlet-class>
</servlet>
<servlet-mapping>
```

```
    <servlet-name>Kaptcha</servlet-name>
    <url-pattern>/kaptcha.jpg</url-pattern>
</servlet-mapping>
```

Passo 3: coloque a tag de imagem e caixa de texto na sua página, como na próxima listagem.

Imagem e caixa de texto no JSP para o Kaptcha:

```
<form action="submit.action">
    <img src="kaptcha.jpg" />
    <input type="text" name="kaptcha" value="" />
</form>
```

Passo 4: no seu *controller*, compare o valor inserido pelo usuário com o valor da chave de CAPTCHA armazenada na sessão, usando um código como o da listagem a seguir.

Filtro para impedir ataques de força bruta:

```
String kaptchaExpected = (String)request.getSession()
    .getAttribute(
        com.google.code.kaptcha.Constants.KAPTCHA_SESSION_KEY);
String kaptchaReceived = request.getParameter("kaptcha");
if (kaptchaReceived == null ||
        !kaptchaReceived.equalsIgnoreCase(kaptchaExpected)) {
    setError("kaptcha", "Invalid validation code.");
}
```

> **Nota**
>
> Tenha certeza de que você inicializou o `jdk` com `-Djava.awt.headless=true`.

Da mesma forma como no SimpleCaptcha, você pode fazer diversas configurações no `web.xml`, tais como cor, borda, fonte, tamanho da imagem etc. Acesse o site do projeto (vide referências) para mais detalhes sobre como fazer essas configurações. A figura mostra alguns exemplos de CAPTCHAs gerados pelo Kaptcha:

Figura 2.5: Exemplos de imagens gerados pelo Kaptcha

2.5 QUEBRANDO O CAPTCHA

É ilusão achar que o CAPTCHA vai eliminar 100% das chances de alguém criar um programa que possa desvendar o desafio e conseguir se passar por um ser humano. Não há como garantir que apenas humanos vão passar no teste. Na verdade, há algumas vulnerabilidades conhecidas na maioria das implementações.

Hoje, já existem serviços na internet que se propõem a desvendar desafios dos CAPTCHAs (conhecidos por *Software as a Service* — Saas). Existem servidores que aceitam uma imagem CAPTCHA de computadores e tentam resolvê-las (por uma taxa cobrada). A taxa de sucesso dos serviços é muito boa — uma em cinco soluções retornadas.

Quebrando com uso de OCR

Uma abordagem para quebrar CAPTCHA é a mesma usada para reconhecimento de objetos, como as aplicações que reconhecem pessoas em um vídeo. A essência dos problemas é similar.

No nosso caso, queremos encontrar as letras "A", "B", "C" em uma imagem. A distorção da imagem é suficiente para confundir o OCR (*Optical Character Recognition*), mas usando técnicas de computação, é possível identificar corretamente cerca de 90% das imagens. Isso tudo em poucos segundos.

Quebrando sem uso de OCR

A maioria das implementações de CAPTCHAs não destrói a sessão quando a frase correta é inserida. Logo, reutilizando o id da sessão de uma imagem CAPTCHA conhecida, é possível automatizar requisições a uma página protegida por CAPTCHA.

Portanto, uma boa prática é sempre remover o valor do CAPTCHA da sessão, depois que a frase correta é inserida. Isso nem sempre é feito pela API e, portanto, deve ficar a cargo do desenvolvedor.

2.6 CONSIDERAÇÕES FINAIS

Este capítulo apresentou o conceito de CAPTCHA e mostrou algumas opções de APIs Java para sua geração. Além disso, também comentamos sobre como um desafio CAPTCHA pode ser quebrado.

Das APIs apresentadas, se você for usar a implementação padrão, a melhor opção é a API Kaptcha, pois ela é muito fácil de configurar e produz um teste difícil de ser quebrado por máquina. Se você quiser alterar a aparência da imagem, existem várias opções de configurações que podem ser facilmente alteradas no `web.xml`.

O SimpleCaptcha também é uma ótima solução. Contudo, parece que não está sendo mantido e também há erros ao executar o projeto com JDK 1.5 e o header HTTP de saída do servlet. Existem alguns erros reportados, *patches* e mensagens de fórum notificando problemas.

O JCaptcha também é um grande produto, mas é mais uma biblioteca do que uma solução rápida. As imagens CAPTCHA padrão produzidas são muito difíceis para um ser humano ler, ou fáceis de serem quebradas por computador. Além disso, sua API

suporta desafios não apenas com imagens, mas também com sons.

2.7 REFERÊNCIAS

- Wikipédia — http://en.wikipedia.org/wiki/Captcha
- JCaptcha — http://jcaptcha.sourceforge.net/
- SimpleCaptcha — http://simplecaptcha.sourceforge.net/
- Kaptcha — http://code.google.com/p/kaptcha/
- Configuração do Kaptcha — http://code.google.com/p/kaptcha/wiki/ConfigParameters

Capítulo 3

CERTIFICAÇÃO DIGITAL COM JAVA CRYPTOGRAPHIC ARCHITECTURE

por Aman Rathie e Diego Augusto Rodrigues Gomes

A certificação digital tem sido bastante comentada nos últimos tempos e em diferentes esferas do governo e empresas privadas. Ela tem por objetivo assegurar a identidade de uma pessoa através de procedimentos lógicos e matemáticos considerados, até então, seguros. O presente capítulo tem por objetivo apresentar os conceitos básicos envolvidos na tecnologia, bem como algumas características e funcionalidades disponibilizadas pela plataforma Java que agregam confiabilidade nas aplicações que exigem alto grau de segurança.

A internet, desde os primórdios, é uma forma colaborativa de compartilhar informações e de facilitar a comunicação entre diferentes pontos do globo. Entretanto, suas funcionalidades não se limitam apenas a isso: elas foram sendo incrementadas com o desenrolar do tempo e com o desenvolvimento da tecnologia.

Com o crescimento da internet, cada vez mais as pessoas realizam suas transações bancárias e efetuam compras sem sair de casa, tudo pela web. Isso faz surgir novas oportunidades de negócio. Neste contexto, foi necessário criar mecanismos de segurança mais eficientes que cobrissem tais transações.

Crimes virtuais se tornaram comuns e perigosos; pragas virtuais se disseminam facilmente; pessoas começaram a se passar por outras para aplicar golpes. Enfim, a segurança deste ambiente se tornou um ponto bastante questionável e crítico. A troca de documentos sigilosos, por exemplo, se feita de maneira insegura, permite que sejam interceptados por terceiros, causando danos imprevisíveis.

A certificação digital surgiu como uma forma de realizar uma troca segura de informações entre diferentes pontos não confiáveis, de forma que o destinatário da mensagem soubesse quem é o remetente e vice-versa. Tendo em vista a crescente utilização do meio eletrônico na realização de operações que exigem maior grau de segurança, as empresas que prestam tais serviços prezam cada vez mais pela confiabilidade em suas aplicações, e orientam os seus clientes a se protegerem de fraudes através da certificação digital.

Na primeira parte do capítulo, serão abordados os conceitos teóricos de criptografia, assinatura e certificados digitais. Mais adiante, mostraremos um pouco da arquitetura e do suporte que a plataforma Java provê para o uso da certificação digital. Boa leitura!

3.1 A CERTIFICAÇÃO DIGITAL

A certificação digital tem como objetivo cobrir basicamente quatro aspectos de segurança no meio digital: autenticidade, integridade, privacidade e não repúdio. A seguir, descrevemos cada um desses objetivos.

- **Autenticidade:** garante que o conteúdo da mensagem enviada é, realmente, do remetente que a enviou. Visa garantir que ninguém se "passou" pelo verdadeiro emissor.
- **Integridade:** garante que a mensagem não sofreu alterações no caminho entre o remetente e o receptor.
- **Privacidade ou confidencialidade:** impede que o conteúdo da mensagem seja lido por pessoas não autorizadas.
- **Não repúdio:** registro da prova da participação e da intenção. No mundo real, trata-se de um terceiro de confiança que registra compromissos (testemunhas, por exemplo).

Para entender como esses objetivos são aplicados na certificação digital, é importante conhecer alguns conceitos básicos sobre criptografia.

3.2 CRIPTOGRAFIA

Em períodos de guerra, é sempre vantajoso para um lado conhecer informações secretas do seu inimigo. No decorrer do tempo, muitas técnicas foram elaboradas para evitar que um lado do conflito conseguisse informações sobre o seu oponente. E, mesmo quando estas informações eram conseguidas, elas deveriam ser inúteis para tal, ou seja, não poderiam ser compreendidas por aqueles aos quais elas não se destinavam. Tais objetivos, hoje largamente utilizados em aplicações comerciais, são alcançados pela criptografia das informações.

Muito já se ouviu falar em criptografia, mas qual o conceito por trás desta palavra e quais são as técnicas usadas? A criptografia (do grego *kryptós*, "escondido", e *gráphein*, "escrever") nasceu desta

necessidade em esconder algo escrito para outras pessoas.

A mensagem é codificada de forma que apenas o seu destinatário tenha condições de compreendê-la, pois somente ele detém a chave que decodificará a informação. Em termos mais técnicos, a criptografia é o estudo de princípios e técnicas de se transformar a informação a partir de uma forma legível (*plaintext* ou *cleartext*) para uma forma codificada ou cifrada (*ciphertext* ou *codetext*). Esta informação em forma codificada geralmente tem um aspecto ilegível, ou seja, não faz o menor sentido para quem está lendo.

Criptografia simétrica

A criptografia simétrica, também conhecida como criptografia de chave secreta, é o tipo mais intuitivo dentre outros tipos. O método simétrico da criptografia tem se tornado bastante forte com o desenvolvimento da matemática e o crescimento do poder computacional. Isso tem possibilitado a criação de cifras que são praticamente inquebráveis com o poder de processamento das máquinas atuais.

A criptografia simétrica envolve o uso de uma chave secreta conhecida apenas pelos participantes da comunicação segura. O emissor criptografa a mensagem com esta chave, conhecida apenas por ele e por seu destinatário, e envia a mensagem criptografada para o destinatário. O destinatário, de posse da mesma chave, decodifica a mensagem e tem acesso ao seu conteúdo.

O uso da criptografia simétrica não está restrito apenas ao envio de mensagens por um meio de transmissão inseguro. Ela também pode ser usada para realizar uma autenticação mútua forte. Esta autenticação mútua forte é a garantia de que as duas pontas da comunicação são as pontas de que se espera que estabeleçam a comunicação e que estão seguras.

Por exemplo, 'João' e 'Maria' definem uma chave secreta para a criptografia das mensagens. Esta chave secreta pode ser uma palavra, uma frase, uma sequência de números, enfim, está a cargo deles em definir esta chave que se tornará um segredo compartilhado.

Para ter certeza de que 'Maria' e 'João' se comunicam de forma segura em um canal inseguro, eles podem realizar o seguinte procedimento:

- 'João' envia uma mensagem para 'Maria' dizendo: "Oi, meu nome é João";
- 'Maria' gera uma mensagem aleatória e envia para 'João';
- 'João' vai receber esta mensagem, vai criptografá-la utilizando a chave secreta compartilhada com 'Maria' e retornará essa mensagem criptografada para ela;
- 'Maria' vai receber a mensagem criptografada de 'João' e vai descriptografá-la com a chave compartilhada. Verificando que a mensagem recebida é a mesma mensagem enviada anteriormente. Verifica-se que, de fato, a pessoa do outro lado da conversa é o 'João'. Neste momento, 'Maria' sabe que a pessoa com quem se comunica possui conhecimento da chave secreta. Como se pressupõe que a chave é conhecida apenas pelos respectivos emissor e receptor, infere-se que o receptor é 'João';
- 'João' realiza o mesmo procedimento de envio de uma mensagem para 'Maria';
- 'Maria' criptografa a mensagem com a chave secreta e retorna a mensagem criptografada para 'João';
- 'João' decifra a mensagem e confirma que está se comunicando com 'Maria'.

Este é um procedimento básico de autenticação mútua. O leitor que tiver interesse pode conferir o projeto Kerberos do MIT (http://web.mit.edu/Kerberos/). O Kerberos é um protocolo de autenticação mútua que utiliza criptografia para garantir autenticidade e privacidade em um canal de comunicação inseguro.

Tendo em vista que existem outras formas de criptografia, é importante destacar algumas das vantagens e desvantagens da criptografia simétrica.

- **Vantagens:**
  - Os algoritmos de criptografia simétrica são geralmente mais rápidos.
  - São mais apropriados para o processamento de streams de dados grandes.
  - Chaves simples.

- **Desvantagens:**
  - Presume-se que ambas as partes tenham concordado na escolha de uma chave e estejam habilitadas a compartilharem esse segredo de forma segura.
  - Há o perigo de, no compartilhamento da chave, essas informações serem indesejavelmente interceptadas por terceiros.

Existem diversos algoritmos de criptografia simétrica que são amplamente usados nos protocolos de rede hoje em dia: DES, 3DES e AES são alguns deles. O AES (*Advanced Encryption Standard*) foi selecionado em um concurso realizado pelo NIST (*National Institute of Standards and Technology*), em meados de 2000, como o algoritmo padrão de criptografia simétrica.

Criptografia assimétrica

A criptografia assimétrica, também conhecida por criptografia de chave pública, utiliza duas chaves diferentes no processo de criptografia e descriptografia. Este sistema de criptografia de chave assimétrica foi publicado primeiramente em 1976, por Whitfield Diffie e Martin Hellman.

Os participantes de uma comunicação segura utilizando criptografia assimétrica devem possuir um par de chaves único, conhecido como chave pública e chave privada:

- **Chave pública**: que pode ser acessada por qualquer um. Como o próprio nome diz, é pública e todos têm direito a tê-la ou a acessá-la.
- **Chave privada**: deve ser apenas acessada e conhecida pelo seu proprietário. Não pode ser disponibilizada para terceiros.

Estas duas chaves são matematicamente dependentes uma da outra, sendo que uma mensagem criptografada com a chave privada só poderá ser descriptografada com sua chave pública correspondente, e vice-versa.

- **Vantagens:**
  - Maior segurança, pois a chave que compartilhamos é a chave pública. Não existe problema algum em sua distribuição.
- **Desvantagens:**
  - Geralmente mais lentos que os algoritmos de criptografia simétrica.

Alguns algoritmos assimétricos conhecidos são: os de curvas elípticas, El Gamal, DSA e RSA. O poder do algoritmo RSA (nome dado em homenagem aos criadores Ron Rivest, Adi Shamir e Len

Adleman) se baseia na lógica da fatoração do produto de dois números primos grandes, algo computacionalmente complexo para os processadores atuais calcularem. Por essa razão, quanto maior o número de bits das chaves, mais difícil será de quebrá-la.

Um estudo feito pela University of British Columbia mostrou que, para quebrar uma chave assimétrica de 1024 bits, considerando-se um computador que possa processar um milhão de instruções por segundo, seriam necessários 300 bilhões de anos. Note que o universo possui "apenas" 13,7 bilhões de anos.

3.3 ASSINATURA DIGITAL

Para melhor entender o processo de assinatura digital, podemos fazer uma analogia ao processo de assinatura convencional. Quando assinamos um documento, por exemplo, anexamos um tipo de identificação que é teoricamente única e pessoal, e que pode ser verificada por meio de outro documento, como, por exemplo, a carteira de identidade ou um passaporte. O problema é que a assinatura manual não é difícil de ser forjada, o que diminui sua confiabilidade.

A assinatura digital visa solucionar os problemas de autenticação e integridade dos dados sendo transmitidos digitalmente. Para garantir a autenticidade, precisamos de um código digital que seja único para o transmissor.

Como vimos anteriormente, este recurso é a chave privada, que será usada para "assinar" uma mensagem ou arquivo sendo enviado pela rede. Quando assinamos um documento digital, anexamos também nosso certificado digital, que conterá diversas informações, tais como nossa chave pública, autoridade certificadora que emitiu o certificado, validade do certificado, dados pessoais e outros (falaremos sobre certificados digitais mais adiante). De posse dessas

informações, o receptor da mensagem pode realizar diversos tipos de verificações, tais como se a autoridade certificadora é confiável, se o certificado expirou, ou até mesmo ele foi criado para aquele fim específico.

O usuário que deseja assinar um determinado documento utiliza sua chave privada para gerar outro documento, este contendo agora sua assinatura e o documento original. Existem casos em que a assinatura e o documento original não são colocados em um único arquivo, mas sim, enviados separadamente. Não existe nenhum problema nisso, pois no ato da verificação, como veremos mais a frente, qualquer modificação no documento inviabiliza sua validação.

Uma vez enviados os dados (o documento original e a assinatura digital), o receptor pode verificar a sua autenticidade por meio da chave pública do emissor, e sua integridade, garantindo que o documento não foi violado durante sua transmissão.

Processo de assinatura digital

Suponha que Diego deseja enviar uma mensagem para Roger, com a certeza de que se Roger a receber, saberá que foi Diego que a enviou. O que Diego faz inicialmente é gerar o *hash* de sua mensagem utilizando alguns dos algoritmos de hash conhecidos (para entender melhor, leia a seção sobre **hash** a seguir).

De posse de sua chave privada, Diego criptografa o hash da mensagem gerada, resultando no que conhecemos como a assinatura digital da mensagem. Note que, nesse caso, não estamos preocupados com a privacidade da mensagem.

Diego envia a mensagem e a assinatura digital para Roger. De posse dos dados, Roger usa a chave pública de Diego para descriptografar a assinatura digital recebida, resultando no hash

original da mensagem. Para verificar que o arquivo não foi corrompido ou alterado durante o envio, Roger calcula o hash da mensagem novamente utilizando o mesmo algoritmo de Diego e compara com aquele que acabou de descriptografar. Se os dois tiverem o mesmo valor, Roger terá a certeza de que os dados vieram de Diego e que não foram alterados.

Esse foi um exemplo básico de como a assinatura digital garante a autenticidade e a integridade da mensagem (ver a primeira figura). É importante notar que a assinatura por si só não garante a privacidade da mensagem.

No caso, para que Diego pudesse ter a certeza de que somente Roger vai ler a mensagem, ele teria de criptografar também a mensagem, e não somente o hash dela. Esta criptografia seria feita usando-se a chave pública de Roger. Assim sendo, somente Roger poderia descriptografá-la com sua chave privada (veja a segunda figura).

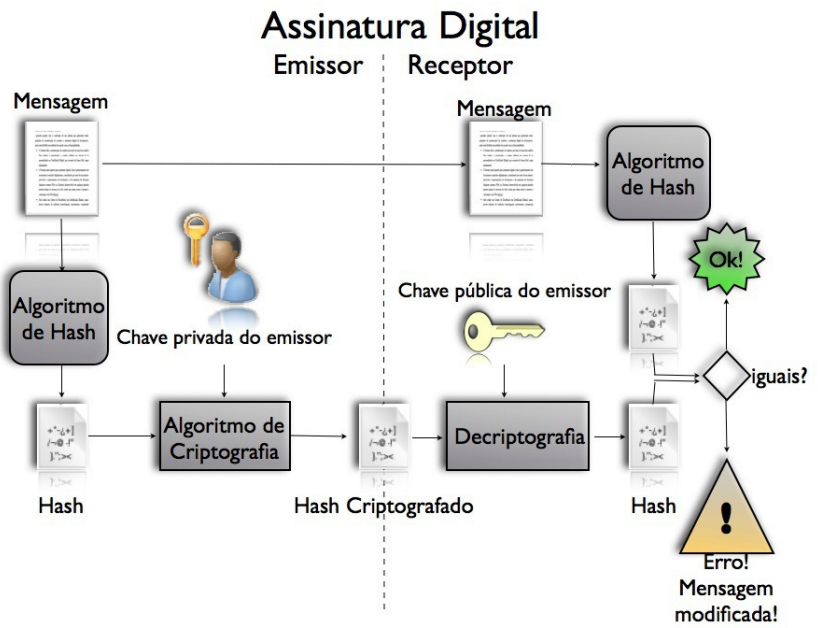

Figura 3.1: Assinatura digital sem privacidade

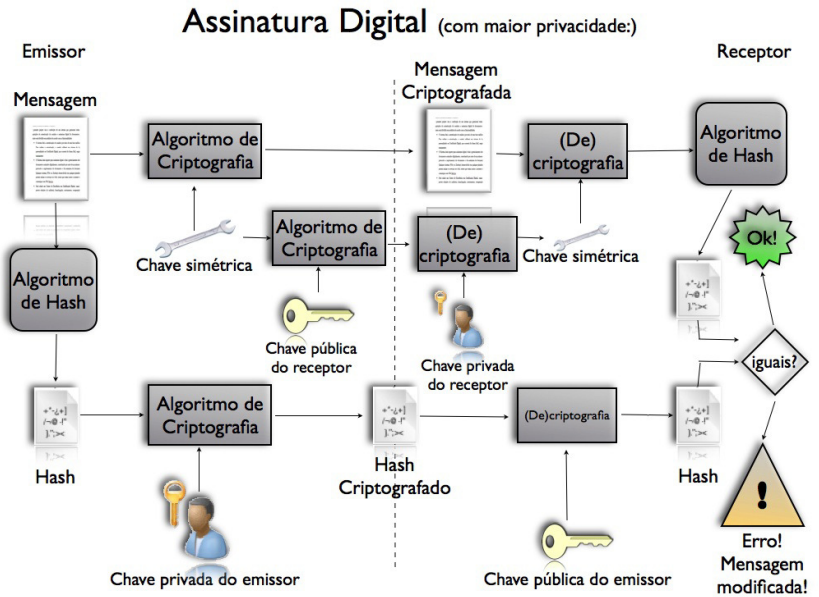

Figura 3.2: Assinatura digital com privacidade

Hash

Um hash nada mais é do que uma sequência "única" de caracteres gerados a partir de uma determinada entrada. Para assegurar a integridade da informação sendo transmitida entre duas entidades, utiliza-se uma função de hash.

Uma função de hash recebe como entrada uma informação, que pode ser desde um simples texto ou até mesmo um arquivo, e produz como saída a tal sequência de caracteres, conhecida também como *message digest* ou *digital fingerprint*. Ela pode ser produzida por diversos algoritmos de hash existentes, dentre os quais os mais conhecidos são o SHA-1 e o MD5.

O hash produzido contém um tamanho fixo de bits, definido pelo algoritmo. Qualquer alteração na mensagem, por mínima que seja, deve produzir um hash totalmente diferente.

Um bom algoritmo de hash deve seguir as seguintes propriedades:

- Dado `x`, deve ser difícil encontrar `m`, tal que `x = hash(m)`. Em outras palavras, não se pode encontrar a mensagem original a partir do seu hash (*Preimage Resistant*);
- Deve ser difícil encontrar duas entradas diferentes `m1` e `m2`, tal que `hash(m1) = hash(m2)` (*Collision Resistant*).

Dessa forma, qualquer alteração que houver em uma mensagem fará com que o hash gerado seja diferente do original, resultando na quebra da integridade da informação. Raramente, pode ocorrer de dois arquivos diferentes gerarem hashes iguais. Isso é conhecido como colisão de hash e pode ser minimizado utilizando-se algoritmos eficientes e que gerem sequências maiores de caracteres para o hash.

O trecho de código a seguir mostra como poderíamos obter o hash MD5 de uma string qualquer. Note a total diferença no hash quando alteramos um pequeno trecho da string.

```
String msg1 = "mensagem qualquer";
String msg2 = "mensagem qualquer 2";
MessageDigest md = MessageDigest.getInstance("MD5");
BigInteger hash1 = new BigInteger(1, md.digest(msg1.getBytes()));
BigInteger hash2 = new BigInteger(1, md.digest(msg2.getBytes()));
System.out.println("hash md5 msg1 = " + hash1.toString(16));
System.out.println("hash md5 msg2 = " + hash2.toString(16));
```

O resultado é:

```
hash md5 msg1 = c421076352a520329710f24772205b0b
hash md5 msg2 = 1af4017ba84dfbc3f1fb922906282f20
```

3.4 CERTIFICADO DIGITAL

Um certificado digital é um documento eletrônico que tem por finalidade identificar pessoas físicas, jurídicas e até mesmo servidores (de aplicação, web etc.). Quando assinamos um documento, "anexamos" nosso certificado digital à assinatura, para que este possa ser lido pelo receptor.

O certificado digital contém diversas informações que identificam a entidade em questão, tais como nome e CPF em um certificado de uma pessoa física. Outras informações identificam a validade e a revogação do certificado. Entretanto, duas informações merecem destaque: a **chave pública** e a **AC (Autoridade Certificadora)**.

A chave pública é usada pelo receptor de uma mensagem para validar a assinatura do emissor e certificar-se de sua identidade. Já uma AC é uma entidade de confiança que emite certificados digitais de diversos tipos, determina prazos de validade e cuida da renovação deles.

Quando uma AC emite um certificado, ela o assina, criando uma cadeia de confiança. Quando um usuário tem dúvida sobre a confiabilidade do certificado, ele olhará para a cadeia daquele certificado (além de sua validade), a fim de averiguar que ele foi emitido por uma AC no qual ele confia.

Uma AC pode ser assinada digitalmente por outra AC, e assim por diante, até que se chegue a uma AC raiz, que por sua vez é autoassinada. No Brasil, a AC raiz é representada pelo ITI (Instituto Nacional de Tecnologia da Informação), órgão que dita as regras da Infraestrutura de Chaves Públicas Brasileira (ICP Brasil).

Para a obtenção de um certificado válido pela ICP Brasil, é necessário que você entre em contato com uma das diversas autoridades certificadoras credenciadas existentes. A lista pode ser visualizada no seguinte endereço: http://acraiz.icpbrasil.gov.br/.

Validação de um certificado digital

A validação de um certificado é importante para assegurarmos sua confiabilidade. Para que um certificado seja considerado válido, levam-se em conta:

- **Data de expiração** – Cada certificado emitido possui uma data de validade. Geralmente, variam de um a três anos. Caso essa data já tenha sido ultrapassada, o certificado é considerado inválido.

- **CRL ou OCSP** – Tanto a CRL (*Certificate Revocation List*) quanto a OCSP (*Online Certificate Status Protocol*) são meios que disponibilizam a checagem de certificados que foram revogados pelas autoridades certificadoras. Um certificado pode ser revogado por diversos motivos, como perda ou roubo da chave privada (que estejam em dispositivos, como tokens ou smart cards, por exemplo), ou o uso impróprio/ilegal do certificado.

  A CRL é disponibilizada em um arquivo que é atualizado periodicamente pela AC. Já a OCSP, um modelo novo, possibilita a consulta on-line do status do certificado.

- **Cadeia de certificados** – Visa garantir que o certificado emitido faz parte de uma cadeia que é considerada confiável. Poderíamos, por exemplo, aceitar somente certificados cuja raiz da cadeia fosse o ITI, recusando todos os outros. A maioria dos browsers e outras aplicações já incorporam uma série de autoridades certificadoras na sua lista de confiabilidade. Por isso, existe um esforço contínuo para que a raiz da ICP-Brasil seja incoporada por padrão no maior número de

aplicações possíveis.

A figura adiante mostra o exemplo de uma *applet* que foi validada pelo browser e apresenta a mensagem "*The digital signature has been validated by a trusted source*". Quando você aceita que a applet seja executada, você dá permissões para que ela possa acessar vários recursos do seu sistema (inclusive o envio de dados para o servidor). Por essa razão, é importante que a applet seja validada (o que mesmo assim não impede que ela contenha algum código malicioso).

Vale a pena lembrar de que um certificado que não passe em um desses critérios ainda pode ser utilizado ou aceito, mas por conta e risco próprio. O leitor que estiver interessado na assinatura digital de applets deve dar uma olhada no comando `jarsigner` do Java. Veja as referências para mais informações.

Outro uso extenso de certificados digitais é feito no protocolo de segurança SSL. A próxima seção sobre SSL mostra um exemplo de criação de uma conexão segura usando certificados digitais. Além desses critérios, é necessário verificar informações constantes dentro do certificado como, por exemplo, se ele foi criado para a assinatura de documentos ou apenas para a autenticação. Estas informações geralmente estão presentes dentro do certificado em campos denominados *Certificate Policies* e *Certificate Extensions* (segundo o padrão X.509).

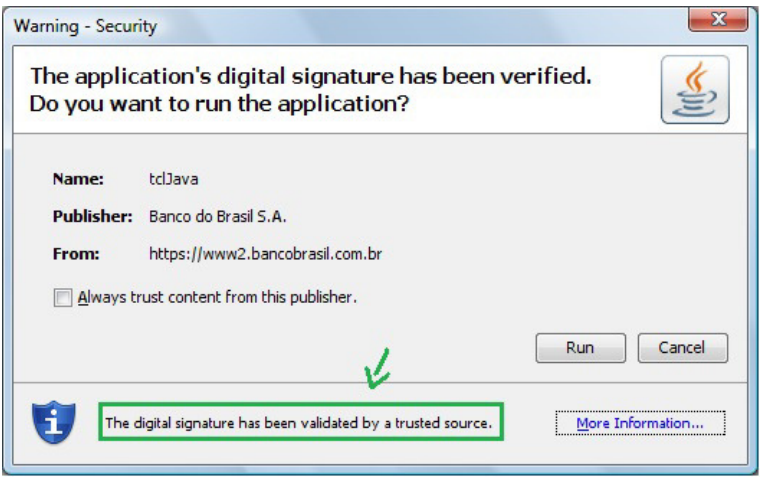

Figura 3.3: Applet com certificado confiável

SSL

O SSL é um protocolo criptográfico que provê comunicações seguras na internet (e-mail, acesso a páginas web, mensagens instantâneas) e que visa prover confiabilidade e privacidade através de um canal de conexão segura entre os participantes. Utiliza algoritmos de criptografia simétrica para a codificação dos dados transmitidos, e algoritmos de criptografia assimétrica para codificação da chave simétrica e para fins de assinatura digital.

De forma sucinta, as etapas envolvidas em uma conexão SSL são as seguintes (veja a figura adiante para um exemplo ilustrativo):

- O cliente envia uma mensagem para o servidor informando uma série de informações referentes ao protocolo suportado por ele (versão, algoritmos de criptografia e funções de hash).
- O servidor escolhe o algoritmo de criptografia e a função de hash mais forte informados pelo cliente e o informa àqueles que foram selecionados.

- O servidor envia seu certificado digital para o cliente. Esta é a sua identificação. Opcionalmente, o servidor pode solicitar o certificado do cliente para fins de autenticação.
- Para gerar as chaves de sessão usadas nesta conexão segura, o cliente criptografa um número aleatório com a chave pública do servidor e envia o resultado para o servidor.

A partir deste número aleatório, ambos os lados da comunicação realizam a criptografia e descriptografia da informação. Feito esse processo, conhecido também como *handshake*, cliente e servidor estabelecem uma conexão segura em que dados serão transmitidos de forma criptografada.

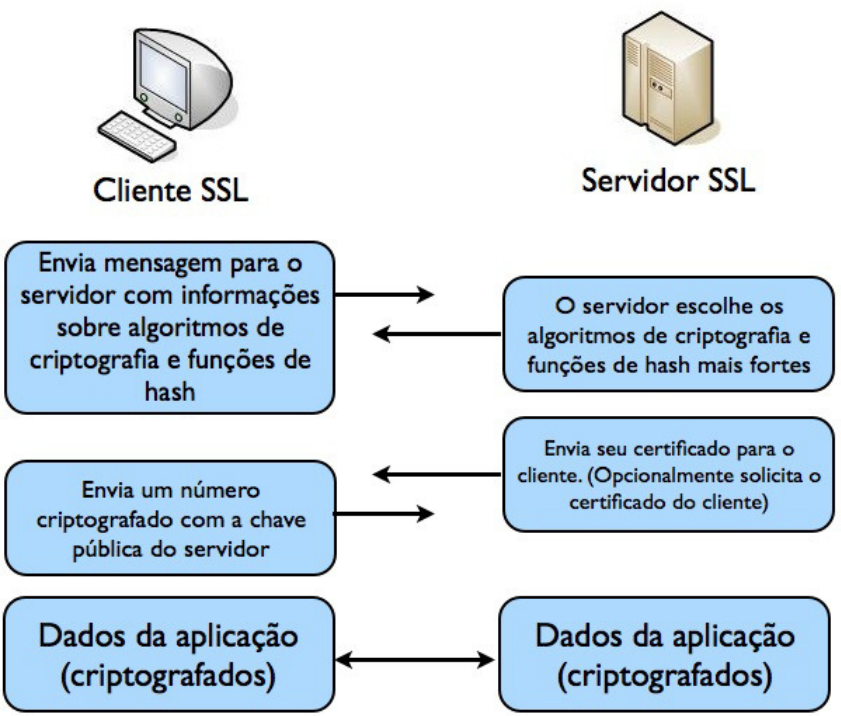

Figura 3.4: SSL Handshake

3.5 SEGURANÇA NA PLATAFORMA JAVA

A plataforma Java foi projetada com uma forte ênfase na segurança. A verificação e o carregamento de classes garante que apenas código Java que seja legítimo possa ser executado.

Atualmente, a arquitetura de segurança da plataforma Java inclui grandes conjuntos de APIs, ferramentas, implementações de algoritmos, protocolos e mecanismos de segurança. Dentre as várias frentes de segurança oferecidas, focaremos na parte da plataforma que trabalha com dispositivos de segurança e certificados digitais.

JCA/ Providers

Ao se trabalhar com dispositivos de segurança digital usando a plataforma Java, é importante conhecer os conceitos, as siglas e as APIs envolvidas. A primeira delas é a JCA (*Java Cryptography Architecture*) e, em seguida, serão vistos alguns conceitos relacionados aos *providers*.

JCA – Java Cryptography Architecture

A JCA é uma parte da plataforma Java que contém um conjunto de APIs utilizadas em processos de assinatura digital, hashes de mensagens, validações de certificados, criptografia, geração e gerenciamento de chaves, dentre outros serviços criptográficos. A JCA foi projetada segundo alguns princípios:

- Interoperabilidade e independência de implementação;
- Extensibilidade e independência de algoritmos.

A independência de implementação e de algoritmos são conceitos complementares, e significam dizer que o desenvolvedor não precisa conhecer os detalhes de implementação dos algoritmos que formam a base dos serviços criptográficos. Entretanto, a

completa independência de implementação não é possível.

Para resolver isso, a JCA provê a possibilidade de utilização de APIs específicas nas chamadas a estes serviços. Em outras palavras, a JCA permite que o desenvolvedor escolha a API que implementa o serviço desejado e que realizará efetivamente a execução do serviço criptográfico desejado.

A independência de algoritmos é alcançada por meio da definição de classes criptográficas chamadas de classes engines (também chamadas de serviços criptográficos). Alguns exemplos são: `Signature`, `KeyPairGenerator` e `Cipher`. As classes *engines* são classes definidas pela JCA, mas que são implementadas por um *provider*.

A interoperabilidade significa dizer que qualquer artefato gerado por um provider poderá ser usado em outro. A extensibilidade é a capacidade de se implementarem novos algoritmos que são suportados pelas classes *engine*.

Providers

O *provider* refere-se a um pacote ou conjunto de pacotes que implementam serviços criptográficos, como, por exemplo, os algoritmos de assinaturas digitais. São os reais fornecedores, provedores dos serviços criptográficos.

A classe `java.lang.Provider` é a principal para todos os providers. Cada provider, também conhecido como *Cryptographic Service Provider* (CSP), contém uma instância desta classe com o seu nome e os serviços disponibilizados por ele.

A instalação do JDK contém um ou mais providers-padrão configurados e instalados. A ordem de preferência em que os providers serão pesquisados pode ser definida no ambiente de

execução da plataforma Java. A utilização destes se dá por meio de chamadas às classes engines com o tipo de algoritmo ou serviço especificado.

Nesta arquitetura, seguindo a ordem de preferência dos providers, o primeiro que implementar o serviço solicitado será selecionado. Entretanto, há a possibilidade de determinar um provider específico para fornecer o serviço desejado. Segue o exemplo de um `MessageDigest`, classe engine responsável por extrair o valor do hash de alguma informação segundo um algoritmo:

```
MessageDigest md = null;
md = MessageDigest.getInstance(“MD5”);
md = MessageDigest.getInstance(“MD5”, “ProviderC”);
```

No primeiro caso, o primeiro provider que implementa o algoritmo MD5 na classe `MessageDigest` será usado. No segundo, utiliza-se a implementação do provider `ProviderC`. A figura a seguir representa as duas chamadas de métodos mostradas antes.

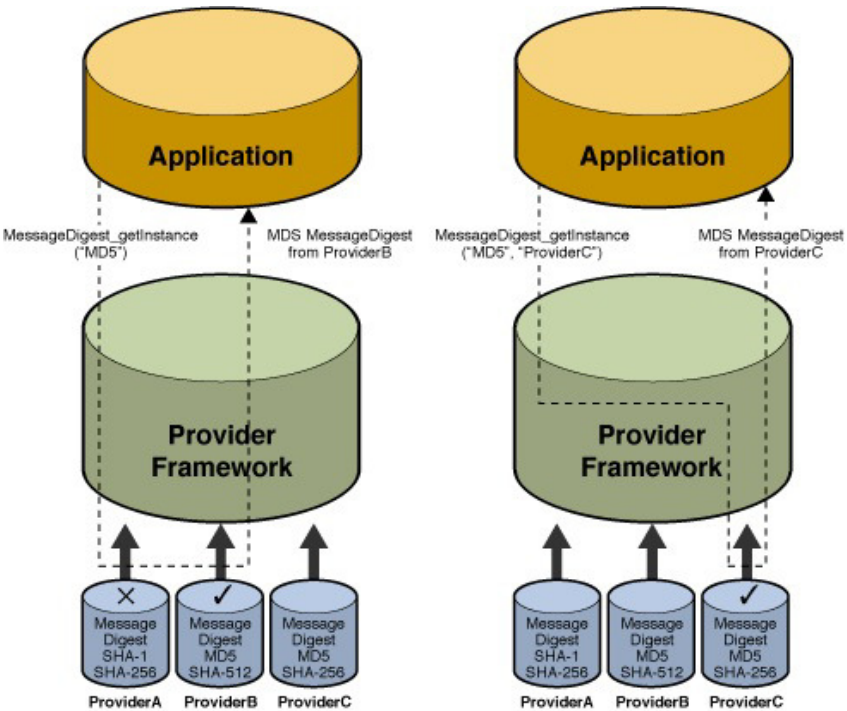

Figura 3.5: Exemplo de chamada de um serviço e disponibilizados por diferentes providers

O JDK da Sun contém providers que implementam alguns serviços criptográficos. Dentre eles, podemos destacar:

- SUN: contém serviços de hash (`MessageDigest`) e assinatura (`Signature`);
- XMLDSig: contém serviços de assinatura digital baseada em XML;
- SunMSCAPI: permite que aplicações usem a API padrão da JCA para acessar bibliotecas criptográficas nativas e repositório de chaves e certificados da plataforma Windows;
- SunPKCS11: estabelece a comunicação entre o ambiente Java e os providers PKCS11 nativos (não contém as funcionalidades criptográficas);

- SunRsaSign: contém a implementação do tipo de assinatura RSA.

Talvez ainda tenha ficado um pouco vaga a ideia do funcionamento dos providers em uma aplicação real. O exemplo a seguir (mostrado na figura seguinte) tenta apresentar de forma clara como tudo funciona.

Imagine uma aplicação na qual seja necessário verificar a integridade de uma mensagem qualquer. Como foi visto, funções de hash são utilizadas para fazer esta verificação. Para isso, é criada uma instância da classe `MessageDigest` com o algoritmo "SHA-1". Ao invocar o método estático `MessageDigest.getInstance("SHA-1")` que cria esta instância, a JCA busca em sua lista de providers aquele que contém a implementação do algoritmo "SHA-1".

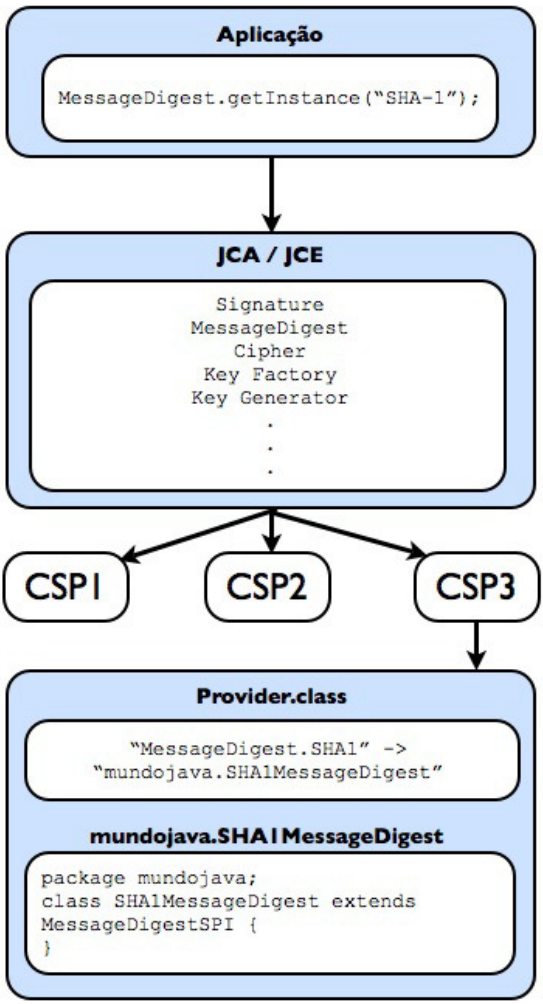

Figura 3.6: Representação dos providers

Para cada classe engine, existe uma classe abstrata SPI (*Service Provider Interface*) correspondente. A classe que implementa um serviço criptográfico em um provider é uma subclasse da SPI da respectiva classe engine. No desenvolvimento de projetos de segurança utilizando a plataforma Java, muitas vezes é necessário utilizar providers que implementam certa funcionalidade não

contemplada pelos providers padrão. Exemplos destes providers são o Bouncy Castle e o JSS, ambos gratuitos e de código aberto.

O Bouncy Castle possui toda uma estrutura para a manipulação de certificados X.509, uma biblioteca para a leitura e escrita de objetos codificados em ASN.1 (presente em alguns campos dentro de um certificado digital), classes para a criação e verificação de pacotes de assinaturas no formato CMS, além de outras funcionalidades que não são encontradas em providers presentes no JDK.

O JSS (*Network Security Services for Java*) é uma interface Java para o NSS (*Network Security Services* – projeto Mozilla composto por um conjunto de bibliotecas que implementam diversos padrões de segurança).

3.6 PADRÕES DE ASSINATURAS

O processo de assinatura digital permite que tais assinaturas sejam empacotadas em diferentes formatos. Dentre estes, destacamos os padrões CMS e XMLDSig.

CMS

O CMS (*Cryptographic Message Syntax*) é baseado na sintaxe do PKCS#7 e está especificado na RFC 3852. Pode ser utilizado em assinaturas digitais, hashes, autenticação ou criptografia de qualquer informação digital. Podemos dizer que o CMS é uma atualização do PKCS#7. O PKCS#7 (*Public Key Cryptography Standards #7*) é um padrão definido pela RFC 2315, e é usado na assinatura e criptografia de mensagens sob uma infraestrutura de chaves públicas.

Arquivos de assinaturas CMS (ou PKCS#7) possuem extensão

`.p7s` . Um exemplo de assinatura de uma informação qualquer em formato CMS é apresentado na listagem a seguir. Para este exemplo, foi utilizada a API do Bouncy Castle (ver referências).

O método cria um repositório de certificados a partir da cadeia de certificados recebida como um argumento da sua chamada. Este repositório é representado pela classe `java.security.cert.CertStore` . Em seguida, uma instância de `org.bouncycastle.cms.CMSSignedDataGenerator` é criada, representando um gerador de assinaturas CMS.

Após a criação de um objeto desta classe, informações do assinante, da cadeia de certificados e da lista de certificados revogados são adicionadas a ele. A informação que será assinada é encapsulada por uma instância da classe `org.bouncycastle.cms.CMSProcessableByteArray` . Depois, é gerada uma assinatura CMS e armazenada em uma entidade do tipo `org.bouncycastle.cms.CMSSignedData` .

Este método recupera o array de bytes referentes à assinatura gerada e o retorna no fim da execução do método. O sistema operacional Windows, por exemplo, possui um programa que identifica arquivos `.p7s` e apresenta a cadeia de certificados que participou do processo de assinatura (figura adiante).

Assinando no padrão CMS:

```
/**
 * Método responsável por assinar um array de bytes
 *
 * @param arquivo bytes que formam a informação/arquivo
 *        que será assinado
 * @param cadeiaDeCertificados cadeia de certificados do
 *        responsável pela assinatura
 * @param chavePrivada chave privada do responsável pela
 * assinatura
 * @param nomeDoProvider nome do provider que implementa
 * esta operação de assinatura
 */
```

```
public byte[] assinaturaAttached(byte[] arquivo,
            X509Certificate[] cadeiaDeCertificados,
            PrivateKey chavePrivada,
            String nomeDoProvider) {

    byte[] informacaoAssinada = null;
    // repositório de certificados
    CertStore certsStore = null;
    // classe geradora de uma assinatura PKCS#7
    CMSSignedDataGenerator signedDataGenerator = null;
    // interface contendo método que copia os dados
    CMSProcessable content = null;
    // classe geral para gerenciar um pacote de assinatura PKCS#7

    CMSSignedData cmsSignedData = null;
    List<X509Certificate> certList = null;

    try {
        certList = Arrays.asList(cadeiaDeCertificados);
        // é criado um repositório de certificados com a cadeia
        // de certificados do assinante
        certsStore = CertStore.getInstance("Collection",
            new CollectionCertStoreParameters(certList), "SUN");

        // cria uma classe geradora do pacote de assinatura CMS
        signedDataGenerator = new CMSSignedDataGenerator();

        // classe geradora adiciona um assinante, passando chave
        // privada, seu certificado (posição 0 da cadeia de
        // certificados) e o algoritmo de hash utilizado na
        // assinatura (SHA-1)
        signedDataGenerator.addSigner(chavePrivada,
            cadeiaDeCertificados [0],
            CMSSignedDataGenerator.DIGEST_SHA1);

        // classe geradora adiciona o repositório de certificados
        // e listas de certificados revogados
        signedDataGenerator.addCertificatesAndCRLs(certsStore);

        // criação da classe que contém os dados (arquivo)
        content = new CMSProcessableByteArray(info);

        // o gerador de assinaturas CMS gera um objeto assinado
        // O valor true significa que uma cópia da
        // informação assinada
        // será incluída na assinatura
        cmsSignedData = signedDataGenerator
            .generate(content, true, nomeDoProvider);
```

```
            informacaoAssinada = cmsSignedData.getEncoded();
        } catch (Exception e) {
            e.printStackTrace();
        }
        return informacaoAssinada;
}
```

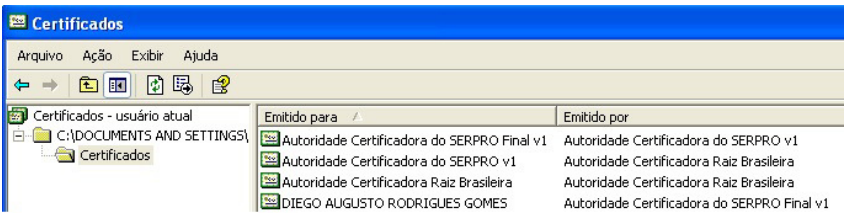

Figura 3.7: Cadeia de certificados do arquivo assinado no formato CMS

XMLDSig

A assinatura digital baseada em XML, além de ser uma recomendação da W3C, é um padrão largamente utilizado em processos de assinatura digital. O padrão pode ser usado para assinar qualquer formato de arquivo, não se limitando a assinatura de arquivos XML.

Assinaturas digitais XML habilitam uma pessoa que está enviando mensagens a assinar informações criptograficamente, e as assinaturas podem ser usadas como credenciais de autenticação, ou como forma de checar a integridade dos dados enviados. Com o advento do padrão XML para representação de dados e da interoperabilidade entre aplicações através do uso de Web services, o padrão XMLDSig é cada vez mais utilizado na certificação digital.

A partir da versão 6, o Java incorporou o suporte ao XMLDSig em sua API(JSR 105) nos pacotes `javax.xml.crypto` e `javax.xml.crypto.dsig` . Apesar disso, o suporte é limitado. Para o leitor que queira se aprofundar mais, recomendamos dar uma olhada na biblioteca Apache XML Security

(http://santuario.apache.org/).

3.7 CONSIDERAÇÕES FINAIS

A certificação digital vem agregando confiabilidade em transações realizadas no meio digital. Através da infraestrutura de chaves públicas, cria-se um ambiente capaz de agregar autenticidade, integridade, privacidade e não repúdio ao meio digital.

Sua confiança está diretamente atrelada ao poder das chaves criptográficas udadas, que em boa parte são impossíveis de serem quebradas com o poderio computacional existente. O modo tradicional de autenticação (baseado no envio de login e senha) é substituído pelo envio do certificado e da assinatura digital, de modo que interceptá-los não representaria, a princípio, um risco.

Obviamente, nem tudo é perfeito. A certificação digital ainda encontra algumas barreiras, como a falta de informação dos usuários a respeito dos benefícios da tecnologia e o custo razoável para a aquisição de certificados de entidades confiáveis. Como qualquer mudança de paradigma, sua adoção será gradual, e iniciativas como a declaração do imposto de renda por meio de certificados digitais tendem a disseminá-la.

3.8 REFERÊNCIAS

- XMLDSIG — http://java.sun.com/javase/6/docs/technotes/guides/security/xmldsig/XMLDigitalSignature.html
- E-Commerce Brasil — http://www.e-commerce.org.br/STATS.htm

- Java Cryptography Architecture (JCA) Reference Guide — http://java.sun.com/javase/6/docs/technotes/guides/security/crypto/CryptoSpec.html
- Bouncy Castle — http://www.bouncycastle.org/
- JSS (Network Security Services for Java) — http://www.mozilla.org/projects/security/pki/jss/
- Apache XML Security — http://santuario.apache.org/