Modelo de Segurança da Linguagem Java

Claudio Masanori Matayoshi – cmmatayo@larc.usp.br
Dr. Wilson Vicente Ruggiero – wilson@larc.usp.br

gSeg – Grupo de Estudos sobre Segurança
LARC - Laboratório de Arquitetura e Redes de Computadores
EPUSP - Escola Politécnica da Universidade de São Paulo

Av. Prof. Luciano Gualberto Travessa 3 – 158, sala C1-46
Cidade Universitária – Cep: 05508-900
São Paulo – SP
Telefone: (011) 818-5280

Resumo : Nos dias de hoje é inegável a disseminação do uso da Internet como meio para disponibilizar os mais variados serviços e informações para usuários em qualquer parte do mundo. Para permitir determinadas aplicações, o uso de simples páginas estáticas já não é suficiente. Entre as inúmeras tecnologias que surgiram para permitir adicionar processamento aos browsers, a linguagem Java tem se destacado, com seu objetivo de permitir uma total independência de plataforma. Entretanto, ao possibilitar a execução no computador do usuário de um código de origem externa, inúmeras questões referentes à segurança devem ser analisadas. Este artigo tem como objetivo apresentar o modelo de segurança adotado pela linguagem Java e sua correspondente máquina virtual, ressaltando a sua importância e as respectivas implementações para identificação de possíveis características que possam vir a ser utilizadas em ataques, assim como, propor as soluções associadas.

Abstract : Nowadays the Internet is one of the main media to spread services and informations for everyone around the world. Simple static html pages are not enought for some kind of applications. Among the technologies that enable client-side processing in browsers, the Java language proposes an universal system for distributing applications. However, this ability to distribute executables over the network raises many concerns about security. This paper describes the Java security model, discusses the importance of studying this model and the necessity of analysing the Java Virtual Machine implementations, searching for features that could be exploit in attacks and suggesting solutions.

1. Introdução

A abertura da Internet para uso comercial, aliada ao surgimento da World Wide Web (WWW) com sua interface gráfica de fácil uso, geraram um enorme crescimento na quantidade de usuários e variedade de serviços disponíveis nestes últimos anos. Atualmente, grande parte das instituições públicas e privadas fornecem algum tipo de informação ou serviço através da Internet.

A interação básica do usuário com a Internet ocorre através do programa cliente do protocolo http (hyper-text transfer protocol). Tal programa cliente é denominado navegador (ou "browser").

A primeira geração de programas navegadores poderia ser considerada um "terminal burro" gráfico, pois apenas apresentavam textos e imagens, sem realizar nenhum processamento. Por exemplo, caso uma página apresentasse um formulário com alguns campos a serem preenchidos, a consistência dos valores fornecidos só poderia ser realizada pelo servidor, quando o navegador enviasse os dados. Apesar do usuário possuir um computador com considerável poder de processamento, este acabava sendo sub-utilizado, pois todo o processamento ficava a cargo do servidor WWW.

Para utilizar este potencial de processamento disponível nas máquinas dos usuários e possibilitar a criação de aplicações até então tecnicamente inviáveis (tais como VRML) surgiram inúmeras tecnologias : Java, JavaScript, ActiveX, Plug-Ins etc.

De uma maneira geral, estas tecnologias permitem que o navegador realize um processamento de informações na máquina do usuário, executando um código que é enviado junto com as páginas WWW.

Neste artigo serão abordados os aspectos de segurança da linguagem Java, e por meio de um exemplo de ataque será ilustrada a importância do estudo deste modelo de segurança e de suas implementações para identificar possíveis características que possam ser utilizadas em ataques e propor soluções.

O artigo está estruturado da seguinte forma : a Seção 2 apresenta as características gerais da linguagem Java, a Seção 3 descreve o seu modelo de segurança e a Seção 4 discute uma forma possível de ataque e propõe uma solução. Leitores já familiarizados com a linguagem Java e o seu modelo de segurança poderão prosseguir a leitura a partir da Seção 4.

2. A Linguagem Java

Um dos principais usos desta linguagem na Internet é a distribuição de aplicações embutidas em páginas WWW (os chamados "*applets*") para serem executadas nas máquinas dos usuários que as acessem.

Além dos *applets*, a linguagem Java também permite a criação de aplicações "convecionais", ou seja, que são instaladas diretamente num computador, desvinculadas de qualquer página WWW (também chamadas de aplicações *stand-alone*).

A linguagem Java foi baseada na linguagem C++, tanto que inúmeras estruturas, como desvios condicionais e laços foram mantidas inalteradas.

Conforme a experiência dos projetistas da linguagem Java, visando criar uma linguagem que facilitasse a geração de código robusto, confiável, facilmente utilizável em plataformas diversas, foram realizadas escolhas que adicionaram novas características ou suprimiram outras do C++.

Estas escolhas levaram alguns a chamarem Java de "C++ ++ --", onde o segundo ++ representaria as inovações e o -- indicam as restrições.

Exemplificando, temos :

- ++ : Garbage Collection, multithreading
- -- : Supressão da aritmética de ponteiros

A linguagem Java é inteiramente orientada a objetos, de tal maneira que apenas os números não são objetos. Todas as demais representação de entidades são feitas por meio de objetos.

O paradigma de programação orientada a objetos já se consagrou adequado para aplicações dos mais variados portes, principalmente aplicações de maior complexidade como os encontrados em ambientes de rede / programação distribuída. A facilidade para o reaproveitamento de código também colabora para o aumento de produtividade.

A linguagem Java fornece inúmeras bibliotecas de classes, tornando possível o desenvolvimento de aplicações completas apenas utilizando estas classes básicas (que formam o chamado Java Core).

Como exemplo destas bibliotecas de classes, temos :
- interface com redes
- interface gráfica
- interface com banco de dados (JDBC)

Características obscuras e pouco exploradas do modelo de objetos do C++ que não representavam um benefício indispensável foram retiradas, como por exemplo a herança múltipla.

A linguagem Java já traz por definição o suporte a multithreading incorporado. Esta é uma característica necessária para aplicações que devam interagir intensamente com o usuário, e especialmente em aplicações multimidia.

2.1 Independência de Plataforma

Esta é provavelmente a característica mais marcante da linguagem Java.

A independência de plataforma se desdobra em independência da arquitetura de hardware e independência do sistema operacional onde o programa deve ser executado.

Nos primórdios da Internet, quando os usuários eram restritos ao meio acadêmico, existia uma relativa independência de plataforma ao se distribuir o próprio código fonte do software, para que este fosse compilado na plataforma destino. Como no meio acadêmico o padrão eram sistemas UNIX com linguagem C, esta solução funcionava relativamente bem.

As primeiras versões do Mosaic eram distribuídas num arquivo que continha os fontes a serem compilados. O usuário adequava apenas alguns parâmetros de compilação para indicar a plataforma onde seria executado o software e após a compilação obtinha-se o Mosaic para a plataforma desejada.

Nota-se que esta era uma independência de plataforma "relativa", pois cabia ao usuário especificar o tipo de plataforma destino pelas diretivas de compilação, e caso não houvesse diretiva adequada para o sistema desejado, o usuário teria que aguardar o "porte" do software para a sua plataforma ou então realizar o "porte" por conta própria.

Com a explosão da Internet, este quadro sofreu uma grande alteração. É praticamente impossível saber qual arquitetura utilizada por um usuário. Como o único requisito para um computador se conectar na Internet é utilizar o protocolo TCP/IP, todas as combinações de arquitetura de hardware e sistema operacional são possíveis : PowerPC, Intel, Digital, Sun, SGI etc utilizando sistema operacional Windows NT, OS/2, UNIX, VMS etc.

A linguagem Java aborda a questão de independência de plataforma em diferentes níveis :

2.1.1 Tipos de Dados e Semântica

As linguagens C e C++ apresentam em sua definição formal alguns aspectos que deixam margem a interpretação, permitindo que fabricantes diferentes utilizem interpretações diferentes, levando a incompatibilidade do software desenvolvido numa plataforma em relação a outra.

Um exemplo é a precisão de números em diferentes arquiteturas. Uma simples pergunta como "Quantos bytes tem uma variável do tipo int em C" não pode ser respondida sem o conhecimento da plataforma na qual pretendemos executar o código C.

Na linguagem Java, qualquer que seja a plataforma, os tipos de dados tem um formato definido e único, como exemplificado na tabela abaixo :

Tipo de Dado	representação
Boolean	1 bit
Byte	8 bits
Char	16 bits (unsigned)
Short	16 bits
Int	32 bits
Long	64 bits
Float	32 bits IEEE-754

Outro possível ponto de conflito é na interpretação da semântica da linguagem, como por exemplo a resolução de argumentos de uma função ou a seleção de métodos sobrecarregados. Na definição da linguagem Java tais aspectos não deixam margem a interpretações conflitantes.

2.1.2 Java Virtual Machine

Este é o principal meio pelo qual é atingida a independência de plataforma na linguagem Java.

O alvo do compilador Java não é o assembler de um processador específico, pois isto limitaria o código gerado a uma determinada arquitetura. O compilador gera o código objeto (chamado *bytecodes*) para um processador hipotético, para a "Java Virtual Machine" (Máquina Virtual Java ou JVM).

As implementações da JVM nas diversas combinações de plataformas de hardware / software possibilitam a execução dos *bytecodes*. As primeiras JVM implementavam na prática um interpretador de *bytecodes*, acarretando conseqüentemente num pior desempenho na execução do código. Atualmente já estão disponíveis ambientes onde antes de serem executados, os *bytecodes* são compilados para o código objeto da plataforma de execução (são os chamados *Just in Time Compilers*). Desta maneira, as perdas na velocidade de processamento são consideravelmente reduzidas.

2.2 Robustez

Outro requisito desejável numa linguagem de programação é a sua robustez. Em linhas gerais, robustez pode ser definida como a capacidade do software executar corretamente a função para a qual foi programada, e nada além disso.

Num ciclo tradicional de desenvolvimento de software, metodologias adotadas pretendem garantir a robustez, além da obrigatória fase de testes. Mesmo com os mais rigorosos procedimentos, podem ocorrer casos em que algum erro acabe passando despercebido.

Estes pequenos erros podem ficar latentes no meio do código, sendo despertados apenas em condições excepcionais. Entretanto, ao serem ativados, podem levar ao funcionamento imprevisível do software e a conseqüências imprevisíveis.

A flexibilidade de C / C++ e sua rapidez se baseiam em parte na confiança de que o programador tem absoluto controle sobre o que o software estará realizando em tempo de execução. Apesar disso, boa parte dos programadores C / C++ já devem ter passado pela

situação onde horas / dias são gastos para determinar que o comportamento "estranho" do software era devido a um ponteiro mal comportado que acabava invadindo indevidamente uma área de memória.

Inúmeras das diferenças entre C / C++ e Java resultam de decisões que visaram tornar a linguagem Java mais robusta. Entre elas, as principais são as seguintes :

2.2.1 Ausência de Aritmética de Ponteiros

Ponteiros são uma das características da linguagem C / C++ que tem grande potencial "destrutivo" num eventual erro de programação. Isto por que ponteiros permitem que se acesse praticamente qualquer posição da memória, principalmente por engano ou ainda com segundas intenções. Para enfrentar este problema, a aritmética de ponteiros foi abandonada na linguagem Java. Todos os acessos às posições de memória (que na prática são as propriedades dos objetos) são realizados pelos métodos fornecidos pelos objetos, e não diretamente.

2.2.2 Verificações em Tempo de Compilação

As restrições da linguagem Java são verificadas na compilação do código. Entre outras podemos destacar a checagem das classes de objetos manipulados, pois em Java não é realizado o "cast" de tipos automaticamente.

2.2.3 Verificações em Tempo de Execução

Outra deficiência da linguagem C / C++ é quanto a verificações em tempo de execução. Nestas linguagens, fica a cargo do programador a responsabilidade de realizar tais verificações, possibilitando o funcionamento incorreto do sistema e até paralisações caso alguma situação errônea não tenha sido devidamente tratada. Um exemplo seria o acesso a um vetor além dos limites originalmente determinados. A linguagem Java realiza uma série de verificações em tempo de execução automaticamente, sendo que situações de erro causam exceções. O tratamento de tais exceções pode ser definido pelo programador, realizando as ações adequadas, ou deixadas a cargo do ambiente Java. A contrapartida para este nível maior de controle é o sacrifício no desempenho (menor velocidade de processamento).

2.2.4 Garbage Collection

Outra parcela significativa de problemas em aplicações são decorrentes de erros na alocação e desalocação de memória para os objetos. É comum liberar o espaço de um determinado objeto e posteriormente referenciar inadvertidamente tal objeto, quando isto já não deveria ser mais possível. Para contornar este problema, o ambiente Java se encarrega de determinar quando o objeto alocado pode ser liberado (processo conhecido como *Garbage Collection*), liberando o programador de tais preocupações (e eventuais erros).

2.3 Segurança

A segurança é um aspecto fundamental para a linguagem Java.

Na principal aplicação desta linguagem (distribuição de *applets* embutidas em páginas WWW para serem executadas nas máquinas dos usuários), o usuário estará recebendo um código através de uma rede de dimensões mundiais (a Internet). Neste momento surge a questão de como ter certeza de que não se está trazendo um vírus ou um cavalo de tróia[1] para dentro do computador ?

A linguagem Java especifica um modelo de segurança para enfrentar esta questão, que é detalhado a seguir.

---

[1] *Cavalo de Tróia : programa desenvolvido com o intuito de ao ser executado realizar ações ofensivas e/ou destrutivas no computador da vítima.*

3. O Modelo de Segurança da Linguagem Java

O modelo de segurança Java é uma característica chave da arquitetura da linguagem que a torna uma opção apropriada para ambientes de rede de computadores [BV97]. Segurança é importante porque estes ambientes permitem um potencial ataque a partir de qualquer computador que tenha acesso à rede. Estas preocupações se tornam especialmente fortes quando programas (software) são trazidos através da rede e executados localmente, como ocorre com *applets* Java. É provável que navegando pela rede o usuário encontrará *applets* de origem não confiável. Para enfrentar este ambiente potencialmente hostil, os mecanismos de segurança da linguagem Java estabelecem uma distinção de tratamento dependendo da origem do código a ser executado.

O modelo de segurança da linguagem Java se propõe a proteger o usuário de programas hostis trazidos pela rede de fontes não confiáveis. Para tanto, o ambiente Java se utiliza de uma *sandbox* dentro do qual o programa Java é executado. O programa pode realizar qualquer coisa dentro dos limites da *sandbox*, mas não consegue realizar nenhuma ação fora de seus limites. *Applets* Java trazidos a partir da rede não podem realizar inúmeras ações, entre elas :

- ler ou escrever no sistema de arquivos da máquina do usuário
- estabelecer conexões de rede para qualquer servidor, a não ser com o servidor do qual o *applet* foi carregado
- criar novos processos
- carregar dinamicamente novas bibliotecas e chamar diretamente métodos nativos

Ao evitar que o código trazido através da rede execute certas operações, o modelo de segurança Java pretende proteger o usuário de programas hostis.

3.1 "Sandbox"

Tradicionalmente, somente se instala um programa se este for de origem conhecida e confiável. Desta forma, pode se garantir a segurança utilizando-se apenas software de fontes confiáveis, sem dispensar a utilização regular de programas anti-virus para verificar possíveis ataques. Uma vez que um software obtenha acesso ao sistema, se este for um cavalo de tróia, os danos podem ser consideráveis, pois apenas as restrições existentes no ambiente de execução limitarão a sua ação (isso quando estas restrições existem). Portanto, em esquemas tradicionais de segurança, antes de tudo procura-se evitar que códigos mal-intencionados obtenham acesso ao computador.

O modelo de segurança baseado na *sandbox* facilita o tratamento de software originado de fontes não confiáveis. Em vez de exigir que o usuário se preocupe em evitar que um código de origem não confiável chegue ao seu computador, o modelo de *sandbox* permite que um código de qualquer fonte seja executado. Mas, durante a execução, a *sandbox* evita que o código de origem não confiável realize qualquer ação que possivelmente poderia comprometer a segurança do sistema.

Para assegurar o funcionamento adequado da *sandbox*, o modelo de segurança Java envolve diversos aspectos da arquitetura da linguagem. Se houver áreas na arquitetura Java na qual a segurança for fraca, um programador mal-intencionado potencialmente poderia explorar esta área para contornar as restrições da *sandbox*.

Os componentes fundamentais responsáveis pela segurança na linguagem Java são :

- características de segurança presentes na máquina virtual Java e na própria linguagem.
- o processo de carga de classes (*class loader*)

- o processo de verificação de classes (*class verifier*)
- o gerenciador de segurança (*security manager*) e a API Java

3.2 Características de Segurança da Máquina Virtual Java e da Linguagem

A máquina virtual Java (*Java Virtual Machine* - JVM) implementa inúmeros mecanismos para garantir a segurança durante a execução de um programa. Os principais mecanismos são :

- verificação de referências a objetos utilizando classes diferentes (*type safe reference casting*)
- acesso estruturado à memória (não existe aritmética de ponteiros)
- liberação automática de memória (*garbage collection*)
- verificação de limites de vetores
- verificação de referências nulas

Sempre que um objeto é referenciado, a JVM verifica se este ocorre de maneira correta. Por exemplo, ao se tentar "converter" uma referência a um objeto para uma classe diferente, a JVM avaliará a validade desta conversão (*type casting*). Ao acessar um vetor, é verificado se o elemento solicitado está dentro dos limites definidos. Qualquer acesso a uma referência a objeto que não tenha sido inicializada, gerará uma exceção.

Estas características da JVM fazem com que programas acessem a memória apenas de maneira segura. Isto torna os programas Java além de mais robustos também mais seguros, por duas razões :

Um programa que acesse indevidamente a memória apresentará problemas em sua execução, podendo afetar outros programas.

Além disso, o acesso irrestrito à memória permitiria tentativas de burlar os mecanismos de segurança. Por exemplo, poderia se tentar acessar a porção de memória utilizada pelo módulo de controle de carga de classes (*class loader*) para alterar o seu comportamento.

A proibição de acesso direto à memória é inerente ao próprio conjunto de instruções da JVM (também chamado de *bytecodes*). No conjunto de instruções não há maneira de se expressar um acesso direto à memória.

A característica da linguagem Java permitir o acesso a métodos nativos (isto é, chamada de funções escritas em outras linguagens como C ou C++ e conseqüentemente sem as características de segurança do Java) representaria uma brecha nestas restrições impostas pelos *bytecodes*. Por este motivo o gerenciador de segurança estabelece uma política para determinar quando um programa pode acessar métodos nativos. Por exemplo, *applets* não têm permissão para chamada de métodos nativos.

Ao se chamar um método nativo, tal método estará totalmente livre das restrições da *Sandbox*. Portanto o modelo de segurança para métodos nativos é o mesmo modelo "tradicional" : deve-se confiar no método nativo antes de invocá-lo.

Outra característica importante da JVM que contribui com a segurança é o tratamento de erros por meio de exceções. Devido a este tratamento, a JVM tem sempre uma ação a ser tomada quando uma violação de segurança ocorrer. Ao detectar uma violação, invés de instabilizar todo o sistema, apenas o thread causador pode ser parado. O compilador força o programador a tratar as exceções que os métodos possam gerar.

3.3 Processo de Carga de Classes (Class Loader)

Na JVM, o *class loader* é responsável por trazer os *bytecodes* que definem as classes. Na realidade, pode existir mais de um *class loader* numa implementação de JVM, permitindo aplicações Java a carregar classes de maneiras customizáveis [BV97a].

Uma aplicação Java pode utilizar dois tipos de *class loaders* : um *class loader* primordial e objetos que implementam *class loader*. Por exemplo, no caso de uma JVM implementada por meio de um programa C num determinado sistema operacional, o *class loader* primordial será uma parte deste programa C. O *class loader* primordial em geral carregará classes confiáveis, inclusive as classes da API Java do disco local.

Em tempo de execução, uma aplicação Java pode instalar um objeto *class loader* que carrega classes de uma maneira particular, por exemplo trazendo classes através da rede. A JVM considera qualquer classe carregada pelo *class loader* primordial como sendo confiável, independentemente desta classe fazer parte ou não da API Java. Entretanto, as classes carregadas pelos objetos *class loader* são consideradas por definição não confiáveis.

A flexibilidade proporcionada pelos objetos *class loader* está na possibilidade de determinar em tempo de execução quais classes serão realmente necessárias e carregá-las sob demanda.

Para cada classe carregada, a JVM mantém um controle de qual *class loader* (primordial ou objeto) foi utilizado. Quando uma classe carregada referencia outra classe, a JVM utilizará o mesmo *class loader* que carregou a primeira classe. Devido a esta característica, classes podem apenas interagir com outras classes carregadas pelo mesmo *class loader*. Desta maneira, são criados múltiplos espaços de nomes (*name-spaces*) dentro de uma mesma aplicação Java. Um *name-space* é um conjunto de nomes únicos de classes corregadas por um determinado *class loader*.

Uma vez carregada uma classe num determinado *name-space*, é impossível carregar uma classe diferente que tenha o mesmo nome neste mesmo *name-space*. Por outro lado, se existirem três *name-spaces* distintos, cada uma delas poderá carregar classes diferentes que tenham o mesmo nome. Classes carregadas por *class loader* distintos se localizam em *name-spaces* diferentes e podem apenas interagir caso a aplicação explicitamente permita. Desta maneira, a arquitetura de *class loaders* pode ser utilizada para controlar a interação entre código carregado de origens distintas, evitando que um código hostil ganhe acesso e corrompa código confiável.

Um exemplo prático é o próprio navegador WWW, que utiliza objetos *class loader* para trazer classes que compõem um *applet* pela rede. O navegador inicia uma aplicação Java que instala um objeto *class loader* (em geral chamado de *applet class loader*) que sabe como requisitar arquivos contendo *bytecodes* de servidores HTTP. Quando esta aplicação Java é iniciada, não se sabe "a priori" quais classes o navegador irá solicitar, pois estas são determinadas em tempo de execução, a medida que o navegador encontre páginas contendo *applets* Java. Em geral, a aplicação Java iniciada pelo navegador cria um *applet class loader* distinto para cada ponto na rede a partir do qual ele carrega os arquivos de classes. Desta maneira, classes de diferentes origens estarão em *name-spaces* separados, evitando que um *applet* mal-intecionado interfiram em classes trazidas de outras fontes.

Freqüentemente, objetos *class loader* se apoiam em outros *class loaders*, ao menos no *class loader* primordial. Em geral, um *class loader* verifica a disponibilidade de uma classe junto ao *class loader* primordial, e apenas caso este não consiga atender a solicitação será utilizado o método específico do objeto *class loader* para obter a classe requisitada. Isso é particularmente interessante pois sempre são utilizadas as classes da API Java, fornecidas pelo *class loader* primordial.

A arquitetura do *class loader* constitui a primeira linha de defesa contra código mal-intencionado, pois ela além de previnir a interferência entre códigos de origens distintas, também mantém a integridade da biblioteca de classes confiáveis utilizadas pelo *class loader* primordial. Classes não confiáveis ficam impedidas de se passar por classes confiáveis.

3.4 Processo de Verificação de Classes (Class Verifier)

Toda JVM possui um módulo verificador de classes (*class verifier*) que garante que os arquivos de classes Java tenham uma estrutura interna correta. Caso seja detectado um erro no arquivo de classe Java, uma exceção será gerada. Esse processo é fundamental para permitir a execução de classes de origem externa. [BV97b]

O *class verifier* reforça a robustez do código, pois um erro no compilador ou um código mal-intencionado poderia por exemplo gerar um método cujos *bytecodes* incluem instruções para provocar um desvio além do final deste método, podendo eventualmente instabilizar o sistema.

O *class verifier* verifica a integridade do código antes que este seja executado. Em grande parte dos casos, a verificação dos *bytecodes* antes da execução é suficiente para garantir a robustez do código, evitando a verificação de cada instrução toda vez que esta é executada.

Entre as verficações realizadas pelo *class verifier* estão as instruções de desvio (condicional e incondicional) que devem atingir uma instrução válida dentro do método.

A atuação do class verifier pode ser dividida em duas fases : checagens internas e verificação de referências simbólicas.

3.4.1 Fase Um : Checagens Internas

Esta fase em geral é realizada logo após a carga dos *bytecodes* que definem a classe.
Nesta fase, a estrutura do arquivo da classe Java é analisada isoladamente, assegurando o formato correto, consistência interna, adequação às restrições da linguagem Java e *bytecodes* seguros para serem executados pela JVM.

Nesta fase também é realizada uma análise do fluxo de dados definido pelos *bytecodes* que representam os métodos das classes. Por essa análise, verifica-se que qualquer que seja o fluxo de execução tomado pelo código, ao se executar determinada instrução da JVM, a pilha utilizada para fornecer os operandos desta instrução sempre conterá o correto número e tipos de operandos.

Entre outras verificações realizadas estão :

- utilização de variáveis antes destas conterem um valor inicial;
- referências e atribuição de valores a campos através dos tipo corretos;
- códigos de operação (opcodes) válidos e com o número e tipos corretos de operandos

3.4.2 Fase dois : Verificação de Referências Simbólicas

Esta fase é realizada durante a execução do código. Nesta fase são verificadas as referências simbólicas, que são as referência a classes, campos ou métodos realizadas a partir de seus nomes (e informações adicionais quando necessárias).

Esta fase pode ser encarada como parte do processo de ligação dinâmica (*dynamic linking*). Quando uma classe é carregada, ela pode conter referências simbólicas a outras classes e a seus respectivos métodos e campos. A ligação dinâmica é o processo de converter estas referências

simbólicas em referências diretas. Esta conversão é realizada encontrando a classe sendo referenciada (carregando-a se necessário) e substituindo a referência simbólica por uma referência para a classe, método ou campo.

O *class verifier* assegura que a referência é válida. Por exemplo, pode ocorrer um erro na carga da classe referenciada ou a classe pode não conter o método ou campo referenciado. Nestes casos é gerada uma exceção.

3.5 Gerenciador de Segurança (Security Manager) e a API Java

O Gerenciador de Segurança define os limites da *SandBox* [BV97c]. O Gerenciador de Segurança é na prática qualquer classe derivada da classe java.lang.SecurityManager.

A API Java sempre verifica a permissão de executar uma ação potencialmente insegura consultando o Gerenciador de Segurança. Para cada ação potencialmente insegura, existe um método do Gerenciador de Segurança que define as condições na qual esta ação será permitida na *SandBox*, por exemplo, o método checkRead() define quando se pode executar uma leitura de arquivo. A implementação destes métodos define a política de segurança da aplicação.

As principais atividades potencialmente inseguras são :

- aceitar uma conexão socket de um determinado computador numa determinada porta;
- modificar uma thread (mudar sua prioridade, pará-la etc);
- abrir uma conexão para um determinado computador numa deteriminada porta;
- criar um novo *Class Loader*;
- apagar um arquivo;
- criar um novo processo;
- parar uma aplicação;
- carregar uma biblioteca dinâmica que contem métodos nativos;
- esperar por uma conexão em uma determinada porta local;
- acessar ou alterar propriedades do sistema;
- ler um arquivo
- escrever um arquivo

Quando uma aplicação Java inicia, ela não possui nenhum Gerenciador de Segurança. Opcionalmente a aplicação pode instalar um.

Caso não seja instalado, nenhuma restrição será aplicada nas ações requisitadas à API Java. Por este motivo aplicações Java não apresentam restrições de segurança, em contraste com os *applets*.

Caso a aplicação instale um gerenciador de segurança, então ele atuará durante todo o ciclo de vida da aplicação, não podendo ser substituído, extendido ou alterado.

Geralmente, um método de verificação do Gerenciador de Segurança gera uma exceção se a atividade verificada não for permitida.

Um exemplo prático de Gerenciador de Segurança é aquele presente na JVM dos navegadores que implementam o *SandBox* para *applets*, mencionado no item 3.

4. Exemplo de Ataque e Proposta de Solução

Apesar de todas as precauções tomadas em seu modelo de segurança, desde o surgimento da linguagem Java uma série de ataques foram propostos.

Um possível ataque que poderia ser realizado explorando algumas características da linguagem Java e de sua implementação é descrito em [PSK97].

O ataque se baseia nas características das classes Applet e Thread, e também na capacidade de um *applet* abrir uma nova janela do navegador e manipular o conteúdo exibido nesta janela.

4.1 Applets e Threads

Cada *applet* é implicitamente associado a uma thread, e um *applet* pode acessar outros *applets* através de uma instância da classe *Thread*, da seguinte maneira :

A classe *Thread* possui o método *getThreadGroup* que retorna um objeto *ThreadGroup*. A partir deste objeto é possível obter uma relação de todas as threads (e conseqüentemente *applets*) que pertencem ao mesmo grupo ou subgrupo de threads. O método *getParent* pode ser recursivamente chamado para se chegar ao objeto *ThreadGroup* que é a raiz de todos os grupos de threads do navegador. A partir do objeto *ThreadGroup* raiz é possível obter uma relação de todos os *applets* mantidos pelo navegador.

A relação de threads mencionada anteriormente a rigor é uma relação de referência a objetos da classe *Thread*. De posse desta referência é possível tentar interferir no funcionamento de outros *applets*, por exemplo manipulando a sua prioridade de processamento ou até matando a thread. A extensão desta interferência dependerá da implementação de threads na máquina virtual.

Toda thread tem um nome associado, que é o nome da classe que define o *applet*. Tendo-se uma referência para o objeto da classe *Thread*, este nome pode ser obtido através de um método da classe.

4.2 Applets Interagindo com o Navegador

Um *applet* pode interagir com o navegador para que este apresente um novo documento, através do método *showDocument*.

O método showDocument pode apresentar o documento na janela atual do navegador, num frame da janela atual ou em uma nova janela (uma nova instância do navegador). No Netscape esta nova instância do navegador pode ser identificada por um nome único fornecido como parâmetro do método *showDocument*. Depois de criada, esta nova instância do navegador pode ser acessada através do nome único a partir de qualquer *applet* ativo, independente de seu contexto. Ou seja, qualquer *applet* que saiba o nome da instância do navegador pode solicitar que este apresente um novo documento, independentemente do usuário.

4.3 O Ataque "Man-in-the-middle"

O objetivo deste ataque é monitorar a atividade do usuário para determinar o instante em que ele visita um site-alvo (por exemplo, o site de um banco) e então desviá-lo para um site falso.

Para o funcionamento deste ataque, é necessário que duas instâncias do navegador estejam abertas : uma com um *applet* "espião" e a outra que será utilizada pelo usuário. Isto se deve ao fato do ataque se basear no método *showDocument* que apenas *applets* ativos podem executar.

Para iniciar o ataque, o usuário precisa acessar uma página que contenha o *applet* "espião". Ao ser carregado, este *applet* "espião" através do método *showDocument* abre uma nova instância do navegador, apresentando uma página com o conteúdo esperado pelo usuário. O objetivo destas duas instâncias do navegador é preservar a instância apresentando a página que contem o *applet* "espião", de maneira que este *applet* não tenha sua execução suspensa. Espera-se que o usuário utilize a partir deste momento apenas o nova instância do navegador.

O *applet* "espião" fica constantemente verificando o nome das threads sendo executadas (e conseqüentemente os *applets*), aguardando o usuário solicitar a página alvo do ataque. Para que o ataque seja bem sucedido, é necessário que a página alvo tenha um *applet* com um nome pouco comum (para que o ataque seja iniciado apenas na página alvo). Vamos supor que a página alvo tenha um *applet* xyz.

Quando o usuário acessar a página alvo do ataque, o navegador iniciará a execução do *applet* xyz. O *applet* "espião" irá então perceber que o usuário está acessando a página alvo (pois uma thread de nome xyz.class passou a constar da relação das threads em execução). Neste momento, o *applet* "espião" irá forçar a instância do navegador que o usuário está utilizando a apresentar uma página idêntica à pagina-alvo, só que originada de um site falso. Desta maneira, o usuário ao iniciar o acesso a um determinado site, é desviado para um site falso.

O usuário fornecerá as informações como número de cartão de crédito, senhas de acesso a serviços etc ao servidor falso pensando estar se comunicando com o servidor original. Para acobertar o desvio após coletar as informações de interesse, o site falso poderia apresentar uma mensagem de erro genérica (como por exemplo "Erro na conexão, por favor tente novamente") e retornar ao site real, onde o usuário conseguiria realizar o acesso corretamente e se esqueceria deste incidente.

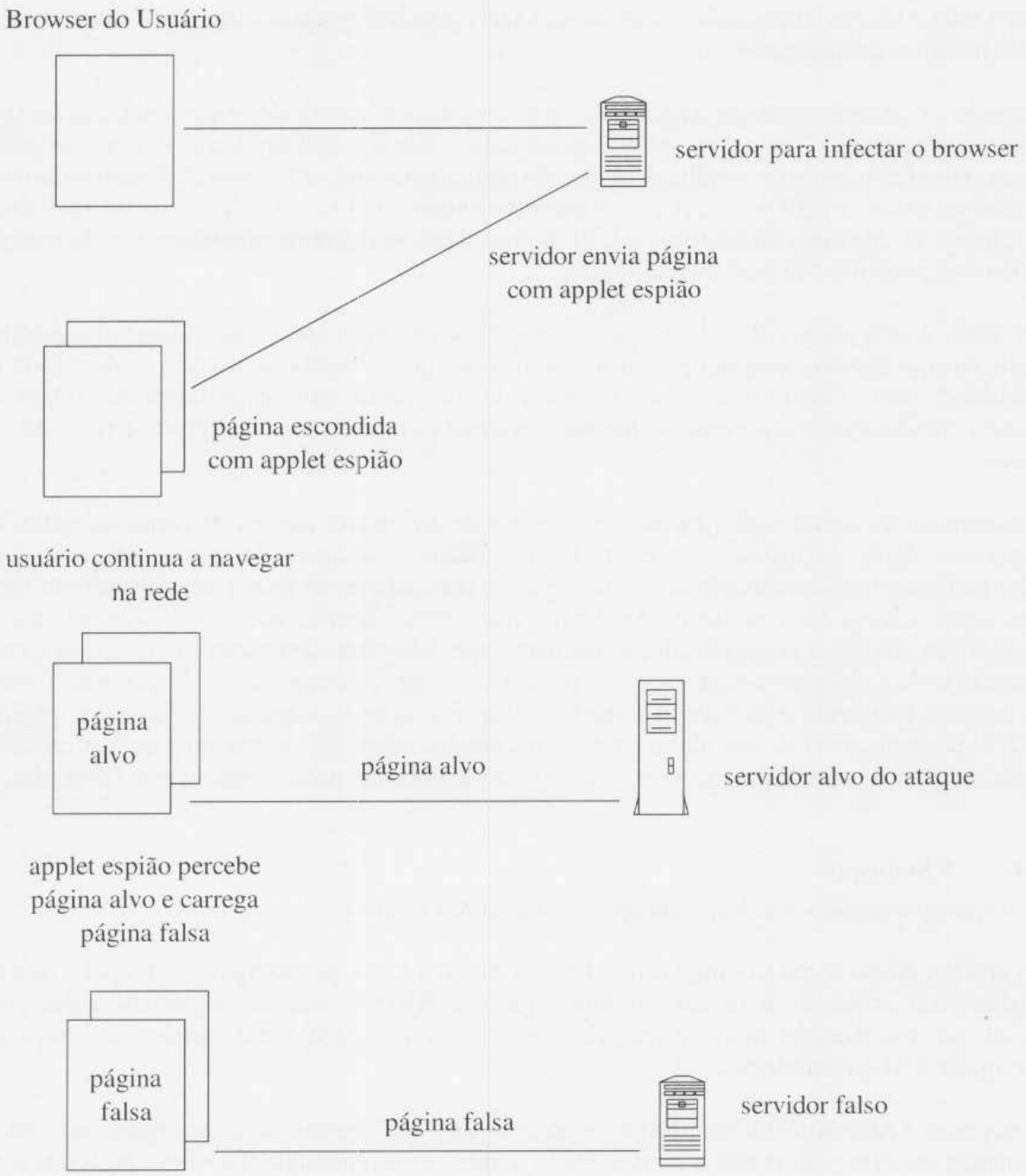

Figura 1. Ataque "man-in-the-middle"

Este tipo de ataque pode gerar algumas questões :

Primeiro, já que o ataque depende de manter duas instâncias abertas do navegador (uma com o *applet* "espião" e outra onde o usuário realiza a navegação), como garantir que o usuário não fechará a instância com o *applet* espião e utilizará a segunda instância para navegação ? Uma maneira de tornar isto mais provável seria adicionar na página que contem o *applet* "espião" informações que são atualizadas periodicamente (por exemplo: informações da bolsa de valores, notícias etc). Para aproveitar esta atualização das informações o usuário não fecharia esta instância do navegador. Outra maneira seria a nova instância do navegador encobrir totalmente a instância anterior que contem o *applet* espião, o que ocorre quando a janela do navegador

ocupa toda a tela do computador. Neste caso, é mais provável que o usuário acabe esquecendo a outra instância do navegador.

Segundo : o usuário pode conferir a URL apresentado pelo navegador e perceber que não é a URL do site original. Neste caso, a URL do site falso poderia ser idêntica com a do site original, exceto por algum pequeno detalhe. Um método seria, registrando um nome de domínio bastante similar ao site alvo [FBDW96], por exemplo substituindo um l (a letra *ele* minúsculo) por um 1 (o número 1). Mesmo mantendo uma URL completamente diferente, dependeria-se da atenção do usuário para que este perceba a alteração.

Terceiro : e se a página alvo do ataque for originada de um servidor que utilize criptografia ? Neste caso, o servidor original possui um certificado que assegura ao usuário a identidade da "entidade" com o qual ele está se comunicando, mas neste caso depende-se da cultura do usuário com relação a segurança na Internet para que este perceba que algo está errado em seu acesso.

A comunicação segura é identificada na interface do navegador através de pequenos símbolos como uma chave não quebrada ou um cadeado fechado. Caso queira ser mais cuidadoso, o site falso poderia ter um certificado de servidor válido, pois para se conferir a identificação do site é necessário acionar uma opção do navegador que provavelmente poucos usuários acionaram alguma vez. Existe a possibilidade de manter o site falso sem certificado, pois a chave ou o cadeado indicando comunicação segura são bem discretos, podendo passar despercebido o fato de estarem indicando conexões não seguras. Por fim, quando submete-se dados em páginas WWW por meio de conexões não seguras os navegadores em geral apresentam uma advertência sobre o fato, mas esta opção pode ser desabilitada ou o usuário pode simplesmente ignorá-la.

4.4 A Solução

O ataque apresentado anteriormente apresenta algumas limitações :

As versões do Netscape Communicator 4.x corrigiram a falha que permitia a um *applet* de uma página obter os nomes de threads em outras páginas [SKP97], não sendo portanto vulneráveis ao ataque mencionado neste artigo. O ataque é efetivo apenas nas versões do Netscape Navigator 3.01 ou anteriores.

O navegador Microsoft Internet Explorer ao criar uma nova instância de navegador através do comando *showDocument* não associa nenhum nome, impossibilitando o controle posterior desta instância pelo *applet* "espião" [PSK97].

Do ponto de vista dos responsáveis pelo site alvo do ataque, poderia se concluir que uma solução seria simplesmente não disponibilizar os serviços para os usuários que possuem as versões vulneráveis do programa navegador ou então sugerir que os usuários desabilitem o suporte a Java em seus navegadores.

Estas soluções podem não ser razoáveis em muitas situações. A negação do serviço para os usuários que possuem versões de navegadores vulneráveis ao ataque pode excluir um número considerável de usuários, podendo prejudicar a imagem do serviço prestado.

Quanto a desabilitar o suporte a linguagem Java, isto poderia inviabilizar o serviço prestado, pois a linguagem Java não é utilizada apenas para embutir animações em páginas WWW. Existem sites onde a linguagem Java realiza funções adicionais de segurança (como por exemplo criptografia) ou proporciona uma interface gráfica com o usuário impossível de ser implementada em páginas estáticas html.

Portanto, para os responsáveis pelo site potencialmente alvo do ataque é necessário uma solução que permita um acesso seguro mesmo para os usuários dos navegadores vulneráveis.

Neste artigo proporemos uma abordagem capaz de neutralizar o ataque descrito :

A monitoração pelo *applet* "espião" aguarda o surgimento de uma thread que tenha o nome de um *applet* específico. A solução se baseia em gerar aleatoreamente os nomes das classes *applets* utilizados no site a ser protegido, impossibilitando a detecção e desvio para o site falso pelo *applet* "espião".

Para permitir a implantação desta solução algumas questões devem ser resolvidas :

- geração dinâmica da página contendo o *applet*
- determinação do nome da classe
- adaptação dinâmica e envio do arquivo contendo os *bytecodes* (arquivo .class)

4.4.1 Geração Dinâmica da Página Contendo o Applet

Uma página html que apresenta um *applet* referencia o nome da classe que implementa este *applet*.

Como o nome da classe que implementa o *applet* da página deve ser aleatório, esta página não poderá ser mais uma página estática armazenada no servidor WWW. Ela deve passar a ser gerada dinamicamente através de algum mecanismo do gênero cgi, como scripts Perl, scripts ASP, ISAPI, NSAPI, servlets etc. No estudo realizado, foi utilizada a geração desta página através de ISAPI (Information Server API) no servidor Microsoft Internet Information Server.

O nome da classe deve seguir um algoritmo, conforme apresentado no próximo item.

4.4.2 Determinação do Nome da Classe

A determinação do nome da classe não pode ser totalmente aleatória, pois o navegador solicitará ao servidor que este envie o arquivo contendo os *bytecodes* da classe. O navegador solicitará ao servidor um arquivo com o nome :

```
<nome-da-classe>.class
```

Caso o servidor a ser protegido forneça mais de um *applet*, será necessário mapear o nome aleatório da classe do *applet* para o *applet* correto a ser enviado.

Sugere-se que o nome aleatório tenha o mesmo tamanho que o nome da classe original, e que este nome aleatório tenha uma propriedade que indique a qual classe se refere. Na implementação testada, o módulo 32 calculado sobre o checksum das letras que compõem o nome aleatório indica o *applet* referenciado pelo nome aleatório. O servidor poderia fornecer um total de 32 arquivos de *bytecodes* com nomes "aleatórios".

Obviamente as restrições quanto a formação dos nomes das classes da linguagem Java devem ser seguidas, como por exemplo, o nome de uma classe não pode começar por um número (caracteres de 0 a 9).

4.4.3 Adaptação Dinâmica e Envio do Arquivo Contendo os Bytecodes

Ao determinar aleatoreamente o nome da classe, como o servidor irá atender o pedido pelo arquivo contendo os bytecodes desta classe ?

Uma possibilidade seria alterar o arquivo fonte adaptando adequadamente o nome da classe para o nome aleatório, compilá-lo e disponibilizar o arquivo contendo os bytecodes. Todo esse processo realizado por demanda, a medida que os nomes aleatórios fossem sendo gerados.

Outra possibilidade seria compilar previamente uma quantidade considerável de classes mudando apenas os nomes. Os nomes "aleatórios" deverão ser sorteados entre estes arquivos previamente compilados.

Entretanto existe uma alternativa menos dispendiosa, que é apresentada a seguir :

Na estrutura de um arquivo bytecode, existe uma tabela de constantes (*constant pool*) onde são armazenadas entre outras coisas strings literais, valores de constantes, nomes de classes, interfaces, variáveis, métodos [BV96]. A diferença entre dois arquivos bytecodes gerados a partir do mesmo código fonte no qual se alterou apenas o nome da classe basicamente são apenas as entradas na tabela de constantes que representam o nome da classe e os nomes dos métodos construtores (que tem o mesmo nome da classe). Portanto para se obter um arquivo de bytecodes válidos de uma classe da qual se alterou o nome basta trocar as entradas na tabela de constantes referentes ao nome da classe e o nome dos métodos construtores.

Na implementação testada, adotou-se uma maneira simplificada para realizar esta adaptação dinâmica dos *bytecodes* das classes que passaram a ter o seu nome gerado "aleatoreamente". Como se assumiu a restrição do nome aleatório ter exatamente o mesmo tamanho do nome "original" da classe, o processo de adequação das entradas da tabela de constantes se resume a simplesmente substituir todas as seqüências de bytes equivalentes ao nome original da classe pelo nome aleatório. Desta maneira evita-se o ajuste da entrada que indica o tamanho da string armazenada na tabela de constantes e uma interpretação mais detalhada da estrutura da seqüência de *bytecodes*.

Este método simplificado impõe uma restrição ao código fonte, que é a de não permitir que ocorra nenhuma outra substring igual ao nome da classe a ser substituida, a não ser na definição do nome da classe e de seus construtores. Por exemplo, caso o nome da classe original fosse "abc" e o nome aleatório passasse a ser "123", supondo que no código fonte houvesse uma referência a um arquivo "testeabc.dat". Ao ser submetido ao método simplificado de adaptação dos *bytecodes*, o código resultante iria passar a referenciar um arquivo "teste123.dat", pois como este método aplica uma busca e substituição de todas as substrings, não apenas daquelas referentes ao nome da classe e dos construtores.

Um outro problema que poderia surgir com este método simplificado seria a eventualidade de uma seqüência de bytecodes coincidir com o nome aleatório, o que em geral deve ser pouco provável. Em todo caso, para uma solução mais robusta e sem tantas restrições deve-se realizar esta adaptação dinâmica dos *bytecodes* através de um método que interprete a estrutura dos *bytecodes*.

Desta maneira dispõe-se de um método para gerar os *bytecodes* correspondentes a uma classe com nome aleatório a partir dos *bytecodes* da classe original sem a necessidade de recompilar o código fonte.

Finalmente, esta adaptação dinâmica dos bytecodes pode ser realizada por mecanismos cgi de maneira que ao receber a solicitação de um arquivo de bytecodes Java com o nome aleatório, a

partir deste nome aleatório possa ser determinado qual a classe "original" e enviar os bytecodes devidamente adaptados.

4.5 Analisando a Solução Proposta

Com a solução apresentada anteriormente, obtem-se um mecanismo capaz de neutralizar o ataque descrito no item 4.3, pois o *applet* "espião" não tem como ser programado para iniciar o ataque, pois não existe mais um nome de thread bem determinado para indicar que o usuário está acessando a página alvo.

Nos testes de implementação da solução proposta, o método de adaptação dinâmica dos bytecodes foi ampliado de maneira que não apenas o nome da classe que implementa o *applet* fosse gerado aleatoriamente, mas também os nomes das demais classes das quais o *applet* dependia passaram a ser gerados aleatoriamente (classes definidas pelo programador, que não fazem parte do conjunto básico de classes da máquina virtual Java). Para permitir esta ampliação, bastou alterar as referências simbólicas aos nome das classes a serem utilizadas de maneira similar à troca do nome da classe e construtores.

A versão final da solução proposta além de neutralizar o ataque proposto no item 4.3 dificulta ainda uma forma de ataque que afeta praticamente qualquer versão de navegador : a incorporação de classes de origem suspeita no CLASSPATH.

Antes de solicitar os *bytecodes* de uma classe ao servidor de onde foi carregada uma página html contendo um *applet*, a JVM implementada no navegador através do *class loader* primordial verificará se tal classe já não está localizada em seu CLASSPATH [MR97]. O CLASSPATH é um diretório situado na máquina do usuário onde estão todas as classes "básicas" da linguagem Java, o chamado "Java Core".

Para adicionar uma classe nova ao CLASSPATH a única ação a ser feita é copiar o arquivo contendo os *bytecodes* da classe neste diretório. Esta facilidade pode ser bastante útil, pois se permite acrescentar classes na máquina do usuário, dispensando a busca através da rede destas classes, tornando mais rápida a execução dos *applets*.

Entretanto, esta facilidade acrescenta uma vulnerabilidade ao sistema. É possível que um usuário seja enganado e acabe acrescentando classes mal intencionadas ao seu CLASSPATH. Por exemplo, poderia ser acrescentada uma classe com o mesmo nome da classe que implementa o *applet* (ou alguma outra classe utilizada por este) que recebe o número do cartão de crédito do usuário e o envia criptografado para ser processado no servidor de um site freqüentemente acessado pelo usuário. No próximo acesso a este site alvo, em vez de solicitar os *bytecodes* das classes ao servidor original, serão carregados os *bytecodes* adulterados que foram localizados no CLASSPATH. Além de enviar corretamente as informações ao servidor original, seria possível enviar as informações coletadas para o autor da fraude.

Com a geração aleatória do nome de todas as classes (exceto as classes do Java Core) utilizadas pelos *applets* relevantes do site a ser protegido, esta forma de ataque também deixa de ser efetiva, pois não existe mais uma determinada classe a ser instalada indevidamente no CLASSPATH da máquina do usuário.

5. Conclusões

Analisando o ataque "Man-in-the-middle", podemos observar que ele se tornou possível devido a problemas na implementação da máquina virtual Java quanto ao acesso de threads de uma determinada página a partir de um *applet* que está numa página diferente.

Já o ataque baseado na adição de classes de origem duvidosa no CLASSPATH do usuário é decorrente em parte da política de segurança adotada. Nesta política, as classes instaladas na máquina do usuário são confiáveis, pois como "não são importadas [diretamente] de fora da organização, e são (em teoria) apenas instaladas por indivíduos confiáveis, estas aplicações Java não acrescentam novas preocupações quanto a segurança" [FM96]. Entretanto, esta abordagem transfere uma grande responsabilidade para o usuário, que em muitos casos pode ser leigo em questões técnicas.

Uma sugestão para dificultar a instalação de classes mal-intencionadas no CLASSPATH da máquina do usuário seria aceitar no CLASSPATH apenas classes digitalmente assinadas. O conceito de assinatura digital pressupõe que a entidade que irá gerar as classes deverá obter um certificado junto a uma autoridade certificadora (*Certification Authority* ou CA) reconhecida pela máquina virtual Java. No processo de obtenção do certificado, esta entidade deverá se identificar junto a CA. Desta maneira, os arquivos contendo classes terão sua integridade garantida e a origem claramente indicada [ITUT93].

Desta maneira, caso alguém queira gerar uma classe mal-intencionada para ser instalada na máquina de algum usuário desavisado, ele terá que obter um certificado válido de uma CA da confiança da máquina virtual Java, o que retira o conforto do anonimato de suas ações.

O lançamento cada vez mais freqüente de novas versões de navegadores torna o teste sistemático das máquinas virtuais Java fundamental para detectar eventuais novas falhas de segurança decorrentes da nova implementação e constatar correções de problemas anteriores.

O modelo de segurança da linguagem Java não deve ser encarado como um modelo perfeito e intocável. Sugestões de aperfeiçoamento devem ser consideradas, discutidas e futuramente incorporadas para aumentar a confiança no ambiente de programação Java.

6. Bibliografia

[BV96] Bill Venners, "Under the Hood: The Java class file lifestyle -An introduction to the basic structure and lifestyle of the Java class file", Java World, Julho de 1996, http://www.javaworld.com/javaworld/jw-07-1996/jw-07-classfile.html

[BV97] Bill Venners, "Java's security architecture - An overview of the JVM's security model and a look at its built-in safety features", Java World, Agosto de 1996, http://www.javaworld.com/javaworld/jw-08-1997/jw-08-hood.html

[BV97a] Bill Venners, "Security and the Class Loader Architecture – A look at the role played by class loader in the JVM's overall security model", Java World, Setembro de 1997, http://www.javaworld.com/javaworld/jw-09-1997/jw-09-hood.html

[BV97b] Bill Venners, "Security and the Class Verifier – A look at the role played by class verifier in the JVM's overall security model", Java World, Outubro de 1997, http://www.javaworld.com/javaworld/jw-10-1997/jw-10-hood.html

[BV97c] Bill Venners, "Java Security : How to install the security manager and customize your security policy", Java World, Novembro de 1997, http://www.javaworld.com/javaworld/jw-11-1997/jw-11-hood.html

[FBDW96] Edward W. Felten, Dirk Balfanz, Drew Dean e Dan S. Wallach, "Web Spoofing : An Internet Con Game", Technical Report 540-96, Dept. of Computer Science,

Princeton University, Dezembro de 1996, http://www.cs.princeton.edu/sip/pub/spoofing.html

[FM96] J. Steven Fritzinger e Marianne Mueller, "Java Security", Sun Microsystems Inc., 1996

[ITUT93] ITUT-T Telecommunication Standardization Sector of ITU, "X.509 Data Networks and Open System Communications - Directory", Novembro de 1993

[MR97] Mark Roulo, "Reduce the launch time of your applets: Store them on client machines", Java World, Junho de 1997, http://www.javaworld.com/jw-06-plugins.html

[PSK97] Flavio de Paoli, Andre L. dos Santos e Richard A. Kemmerer, "Vulnerability of Secure Web Browsers", a ser publicado no 20th National Information System Security Conference, Outubro de 1997, http://www.cs.ucsb.edu/~andre/nissc97.ps

[SKP97] A. L. dos Santos, R. A. Kemmerer, F. de Paoli, "Secure Browsers ?", Computer Science Department, University of California Santa Barbara, 21 de Agosto de 1997