UM SISTEMA DE DETECÇÃO DE CÓDIGOS JAVA HOSTIS NA REDE

Tantravahi Venkata Aditya
Antonio Montes

RESUMO

Visando uma maior proteção aos usuários da Internet e de aplicações Java, este artigo propõe um método para a detecção de códigos Java hostis. Após uma discussão sobre as características do tráfego de rede associado a aplicações Java, propõe-se um módulo para a detecção de códigos Java hostis, para ser incorporado num sistema de detecção de intrusão de redes.

ABSTRACT

In order to achieve a better protection for Internet and Java application users, this article proposes a method for detecting hostile Java codes. After discussing the network traffic characteristics associated with Java applications we present a module that detects hostile Java codes, that has been incorporated in a intrusion detection system.

1 INTRODUÇÃO

Java (http://www.javasoft.com) é uma linguagem multiplataforma, de sintaxe simples, desenvolvida pela Sun Microsystems e com vários recursos úteis para diferentes áreas de aplicação, entre elas a Internet.

Sua popularidade vem crescendo sistematicamente, devido a suas aplicações e também ao fato de permitir uma grande mobilidade de código. Em vista disso a segurança desta linguagem vem se tornando um tema importante e indispensável [1].

1.1 Problema da segurança de aplicações Java

Ao acessar um site na Internet que contém um *applet* (um aplicativo Java específico para páginas da Internet) que é executado instantâneamente, uma boa parte dos usuários não tem idéia dos riscos aos quais estão sujeitos. Códigos móveis auto-executáveis são poderosos, práticos e úteis, mas podem também ser perigosos e inseguros.

Falhas na implementação de navegadores (*browsers*) que habilitam Java vem sendo descobertas com razoável frequência. Essas falhas são tanto no âmbito do sistema, como no âmbito da aplicação [1]. O resultado é uma série de vulnerabilidades que permitem que *applets* Java apaguem ou modifiquem arquivos, capturem informações importantes do sistema, podendo até chegar a corrompê-lo.

Nas referências consultadas, são descritos vários tipos de vulnerabilidades e *applets* maliciosos. Em [2] são mostrados vários exemplos de vulnerabilidades, entre eles, um exemplo de tentativa de DoS (Denial of Service) usando Java. Em [3] encontram-se os *applets* hostis mais comuns da Internet. Livros que tratam de segurança de código móvel consideram Java uma linguagem de risco para o usuário quando não se tomam as devidas precauções [4], [5].

Existem alternativas para impedir que *applets* hostis causem problemas. Uma delas seria fazer uma política de segurança que permita apenas a execução de *applets* certificados por partes confiáveis. Pode-se também evitar navegar em *sites* desconhecidos ou em última instância desabilitar a Internet. Infelizmente, poucos sabem lidar com a política de segurança do Java. Isto deve ser levado em conta, assim como deve-se também considerar aqueles que querem navegar livremente pela rede sem se preocupar com a confiabilidades do *site* que desejam acessar.

Existem algumas soluções para remediar esses problemas. Uma delas seria melhorar o modelo de segurança do Java [6] (este modelo será discutido mais adiante), uma outra seria aplicar as *patches* (atualizacões) de segurança. Uma terceira seria proteger os usuários por meio do bloqueio de *applets* hostis no firewall, como está descrito em [7], que pode ser uma boa solução para o problema.Uma outra solução elegante seria usar mecanismos para se fazer uma auditoria na máquina virtual Java aliada a uma ferramenta de detecção de intrusão [8].

Existem grupos de pesquisas trabalhando que visam a segurança de Java e o levantamento e análise de suas vulnerabilidades; entre eles, destaca-se o Grupo SIP da Universidade de Princeton [9].

1.2 Objetivos e estrutura do artigo

Visando uma solução para o problema da segurança de aplicações Java na Internet, é discutido aqui o uso de um sistema de detecção de intrusão para detectar *applets* hostis, baseado em vulnerabilidades conhecidas, e seu uso para gerar alertas ou interromper a conexão associada. A partir dessas vulnerabilidades conhecidas foram geradas assinaturas baseadas em padrões hostis no conteúdo dos pacotes trafegando na rede que se adequam à ferramenta de detecção de intrusão SNORT [10].

Neste artigo, é mostrada a criação de uma destas regras a partir de uma vulnerabilidade comum, examinando desde a teoria de funcionameto do Java até a geração do alerta correspondente à regra. Inicialmente são introduzidos alguns princípios da linguagem Java e de sua arquitetura, em seguida o seu modelo de segurança atual, tendo assim a fundamentação necessária para a compreensão da metodologia de detecção utilizada e da implementação do filtro. Foi constatado também que somente o uso de filtros no SNORT para detectar tráfego Java hostil pode sobrecarregar o sensor, principalmente em redes com grande volume de tráfego. Isso se deve, em parte, à complexidade dos padrões, como será mostrado adiante. Assim, foi desenvolvido um módulo pré-proces-

sador para separar apenas os pacotes contendo código Java, aos quais serão aplicados os filtros de conteúdo.

2 FUNDAMENTAÇÃO TEÓRICA

O Java é uma linguagem simples, distribuída, portável, e interpretada, mas deveria ser também uma linguagem robusta e inerte à arquitetura, portanto, segura. Infelizmente, esse não é bem o caso. Apesar de ser mais segura e robusta que as linguagens concorrentes ela não é nem tão segura, nem totalmente inerte á arquitetura.

Para permitir que um código Java seja executado é necessário que o navegador instalado na máquina esteja habilitado para isso. O código é executado da seguinte forma: o navegador requisita um arquivo que contém um *bytecode* Java, o servidor envia o código que, em seguida, é executado por uma máquina. Os códigos que são executados na máquina do cliente são conhecidos como *applets*, que foram especialmente desenvolvidos para trabalhar com páginas de Internet.

2.1 Máquina virtual Java

A máquina virtual Java (JVM) é um ambiente virtual baseado em software onde aplicações Java podem existir e manipular recursos [11]. Cada plataforma tem a sua prória JVM, que manipula e traduz requisições feitas ao sistema operacional por programas Java que necessitem de recursos. Em razão disto, os programas precisam ser interpretados antes de serem executados. A portabilidade do Java se deve ao fato de ser uma linguagem pseudo-interpretada, em outras palavras, não é só interpretado durante sua compilação mas tambem durante seu tempo de execução. Essa é a principal razão da lentidão desta linguagem em relação às suas concorrentes como o C++ que é totalmente compilada em linguagem de máquina.

Por outro lado a JVM não requer que os programadores Java se preocupem com detalhes intrincados de como um determinado sistema operacional acessa memória e outros recursos, encarregando-se desta tarefa. É um único processo de múltiplos *threads* ou, em outras palavras, é um processo de múltiplas tarefas que compartilham o mesmo conjunto de dados [12].

3 MODELO DE SEGURANÇA DO JAVA

O modelo de segurança original foi lançado com o JDK 1.0 (Java Development Kit) [13]. A primeira distribuição para desenvolvimento Java disponível para usuários, possuia um ambiente protegido de múltiplas camadas denominado *sandbox*, ou seja, uma caixa de contenção. Esta caixa era severamente limitada no que se refere a acessos remotos, enquanto que applets lançados localmente tinham acesso completo ao sistema. Os componentes que consituem o *sandbox* são o **verificador de bytecode**, o **carregador de classe** e o **gerenciador de segurança**. Cada um destes componentes atua em camadas distintas da caixa de contenção, como pode ser visto na **Fig. 1**.

A linguagem Java foi projetada para prevenir erros em ponteiros de memória e verificação dos limites de *strings* e *arrays*. Possui um gerenciamento de memória incorporado à máquina virtual, inexistente na maioria das linguagens, permitindo a manipulação de memória pelo programador. Em C e C++ os ponteiros de memória são declarados em tempo de compilação e podem requisitar endereços específicos na memória. Em Java as locações de memória são gerenciadas usando-se locações simbólicas de memória que são resolvidas pela JVM durante a execução.

Para tornar a segurança mais flexível foi desenvolvido um novo modelo [14]. Este é baseado num controle de acesso granular que verifica o acesso aos recursos do sistema antes que o código seja executado. Nele tanto *applets* locais quanto *applets* remotos são obrigados a passar pelas etapas de verificação.

Foram criadas zonas de segurança, denominadas domínios de proteção, desde o mais restrito ao mais acessível, como pode ser visto na **Fig. 2** . Todo programa dentro do mesmo domínio possui as mesmas permissões. Este novo modelo não é absoluto e ainda existem grupos de pesquisas que buscam aperfeiçoar este modelo ou novas soluções [6].

3.1 O formato do arquivo .class do Java

Um código-fonte Java compilado resulta em um arquivo chamado de arquivo de classe ou *.class*, pois esta é a extensão do arquivo. O arquivo de classe, seguindo a especificação do JVM, tem uma estrutura como a mostrada abaixo [12].

```
Classfile
{
  u4 magic;
  u2 minor_version;
  u2 major_version;
  u2 constant_pool_count;
  cp_info_constant_pool[constant_pool_count − 1];
  u2 access_flags;
  u2 this_class;
  u2 super_class;
  u2 interfaces_count;
  u2 interfaces[interfaces count];
  u2 fields-count;
  field_info_fields[fields_count];
  u2 methods_count;
  method_info_methods[method_count];
  u2 attributes_count;
  atribute_info_attributes[attribute_count];
}
```

Figura 3: **Estrutura de um arquivo .class**

O *constant_pool* é uma tabela de estruturas que representa um agrupamento de nomes de classes, campos e métodos, e também mostra as constantes usadas dentro do arquivo de classe. O *constant_pool_count* especifica quantas entradas existem no *constant_pool*, cada estrutura *cp_info* representa um dos 11 tipos que podem fazer parte do *constant_pool*. O campo *access flags* é uma máscara de modificadores usada para

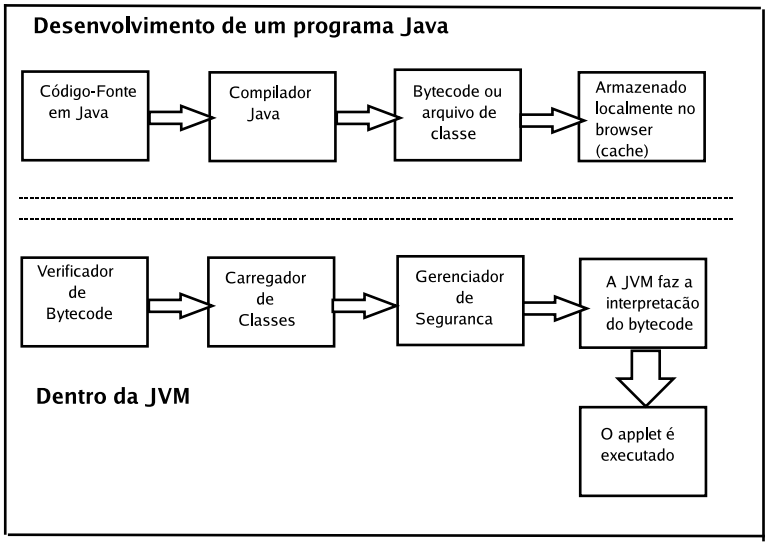

Figura 1: **Componentes da caixa de contenção**

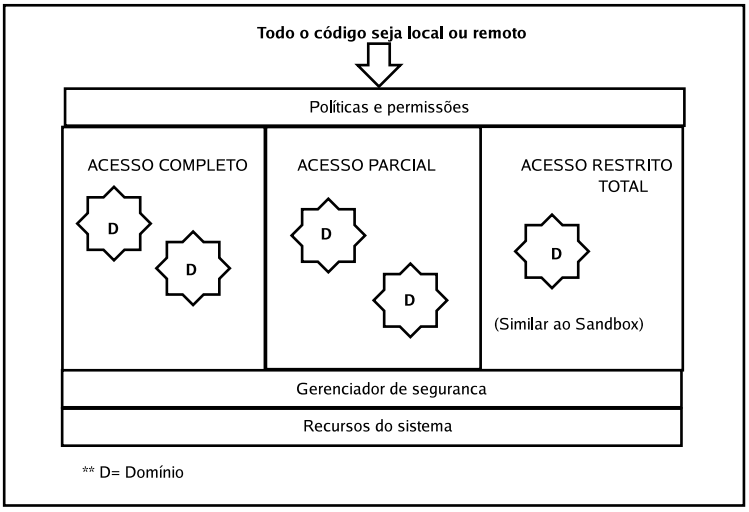

Figura 2: **Modelo de segurança atual**

especificar uma classe e a acessibilidade de uma interface. Os campos *this_class* e *super_class* contém referências à classe atual e à sua super classe. O restante do arquivo de classe consiste de quatro tabelas, uma para as interfaces, uma para os campos, uma para os métodos e uma para os atributos. Cada uma delas é antecedida por um campo de 2 bytes que especifica o número de entradas na tabela correspondente, por exemplo, o campo *methods_count* especifica o número de entradas da tabela de métodos. Todas estas tabelas fazem parte do arquivo de classe mas apenas a tabela de métodos contém o código a ser executado pela JVM.

A notação u*n* significa que o campo possui *n*-bytes. Os primeiros 4 bytes, *u4 magic*, correspondem a um número hexadecimal que serve como um identificador do arquivo de classe para a JVM, OxCAFEBABE. Em seguida tem-se 2 campos de 2 bytes (*minor_version* e *major_version*) que especificam qual a versão do compilador Java que produziu o arquivo.

Essas informações serão utilizadas a seguir no desenvolvimento dos padrões de filtragem.

4 METODOLOGIA

4.1 Sistemas de detecção de Intrusão

Intrusão pode ser definida como o uso indevido ou malicioso de um sistema computacional. Detecção de intrusão é uma tecnologia de segurança que permite identificar e isolar intrusões em computadores ou redes. Hoje em dia, é uma ferramenta muito importante para a segurança de redes de computadores, mas não é a única nem o bastante.

Sistemas de detecção de intrusão em redes, são sistemas que monitoram o tráfego de rede e procuram detectar determinados tipos de intrusões, analisando padrões suspeitos de atividade na rede.

Neste trabalho foi utilizado o SNORT [10] como

ferramenta de detecção de intrusão. Ele é um detetor de intrusão baseado em rede que realiza a análise de tráfego em tempo real e examina pacotes trafegando em redes TCP/IP usando uma linguagem de descrição de regras que é simples, flexível e poderosa. Tem como pré-requisito a biblioteca de captura de pacotes *libpcap*, que permite a análise do tráfego de rede em baixo nível. O SNORT armazena pacotes no formato do *tcpdump* ou em seu próprio formato. Possui a característica de permitir a adição novos módulos, por ser flexivel que, aliada à facilidade de seu uso, a tornam uma ferramenta atraente.

4.2 Metodologia de extração de trechos de código Java hostil

Nesta subseção será apresentado um método para a extração de trechos de código hostil Java. O resultado serão sequências de padrões a serem usados na elaboração de regras para o SNORT.

O arquivo de classe Java, ou seja, seu *bytecode* contém um número de identificação que, em hexadecimal é CAFEBABE, conhecido como número mágico. Este número garante que o arquivo seja uma classe compilada de um código-fonte de um programa em Java. O trecho a seguir mostra o inicio de um arquivo de classe Java.

```
CA FE BA BE 00 03 00 2D 00 17 0A 00 05 00 0E 08
00 0F 0A 00 10 00 07 00 12 07 00 13 01 00 06 3C
.................
```

Em razão disto, para se detectar um *applet* na rede o primeiro passo é verificar se o pacote que está trafegando contém o número mágico. O segundo passo é saber se o programa é hostil: para isto, é necessário que se analise o *bytecode* e se extraiam as sequências de bytes correspondentes á ações hostis. Como exemplo, será feito a análise do seguinte trecho de código hostil:

```
private static void killOneThread(Thread t)
{
 if (t == null || t.getName().equals(¨ killer¨ ))
 {return;}
 else {t.stop()};}
}
```

Figura 4: **Trecho de código hostil que interrompe a execução de outros threads**

Este trecho pertence a um *applet* hostil que impede a execução de outros *applets*. Ele faz uso de um *thread* criado anteriormente chamado de **killer**, que executa com prioridade máxima, como pode ser visto na **Fig. 5**. Consequentemente os *threads* de prioridade menor irão perder o privilégio de execução para o **killer**. O trecho de código verifica se o *thread* é o killer ou se é um que já foi interrompido, nesses casos ele retorna ao programa. Caso seja um *thread* ainda operante, o método *stop()* é chamado para que este seja interrompido.

```
killer = new Thread(this, ¨killer¨);
killer.setPriority(Thread.MAX_PRIORITY);
killer.start();
```

Figura 5: **Thread de prioridade máxima** *Killer*

O *bytecode* correspondente ao trecho de código acima é o seguinte, 2A C6 00 0F 2A B6 00 0D 12 OE B6 OO OF 99 OO 04 B1 2A A6 B6 00 10 B1. Transformando esta sequência de bytes nos seus mnemônicos correspondentes por meio do comando javap [4] teremos:

```
Method void killOneThread(java.lang.Thread)
   0 aload_0
   1 ifnull 16
   4 aload_0
   5 invokevirtual #13
   <Method java.lang.String getName()>
   8 ldc #14 <String "killer">
  10 invokevirtual #15
  <Method boolean equals(java.lang.Object)>
  13 ifeq 17
  16 return
  17 aload_0
  18 invokevirtual #16
  <Method void stop()>
  21 return
```

Uma análise minuciosa deste trecho mostra que alguns bytes podem ser eliminados. As chamadas a métodos e variáveis, cujos *strings* e valores correspondentes são guardados no *constant_pool*, são feitas indicando a posição em que elas se encontram. Esta posição varia de acordo com o programa no qual o trecho de código hostil está presente. Então, é necessário eliminar os bytes correspondentes a estas posições. Os métodos são representados por *strings* em ASCII no arquivo de classe; por fim, levando-se isto em conta, o trecho hostil resultante é: "2A C6", "2A B6", "12", "B6","B1 2A A6 B6", e os *strings* **stop** e **equals**.

5 IMPLEMENTAÇÃO DO FILTRO E DO MÓDULO PARA O SNORT

Será mostrado como arquivo de regras que compõem o filtro do SNORT criado. Para isto será criado uma regra baseado no exemplo da seção anterior.

5.1 *Pré-processador para a captura de arquivos Java*

A detecção de assinaturas de ataques por padrões bem definidos, é feita regras de simples sintaxe efetuada pelo SNORT, torna esta uma ferramenta poderosa em relação às suas similares. Ainda assim, não é o bastante, pois a análise das regras é feita após o processamento do pacote, resultando numa sobrecarga de processamento. Para solucionar este problema, a equipe de desenvolvimento do SNORT criou *plugins* modulares chamados de preprocessadores. Estes têm o objetivo de gerenciar o conteúdo dos pacotes antes do mecanismo de detecção executar o processo de busca por padrões no conteúdo [15]. Uma outra razão para a necessidade de pré-processamento é o número de falsos-positivos que podem ser gerados devido à generalidade de alguns conjuntos de regras.

Mesmo possuindo um mecanismo de inspeção de múltiplas regras de alto desempenho [16], principalmente em relação à sua velocidade, foi necessário criar um pré-processador. As regras criadas possuem múltiplos conteúdos e com isso há um aumento no tempo de processamento e na probabilidade de o mecanismo de detecção deixar passar pacotes que gerariam alertas sem que isso aconteça (falsos-negativos) [17]. Para evitar isso, o pré-processador criado verifica se o pacote contém um *bytecode* Java, para que o conjunto de regras seja aplicado somente nestes casos.

Primeiramente o pré-processador verifica se o pacote contém conteúdo HTTP. Para isto é usada a função **http-resp** do SNORT que verifica se o conteúdo é parte do protocolo HTTP e se veio como resposta á uma requisição feita ao servidor pelo cliente. Em seguida é verificado se o cabeçalho do protocolo é o da resposta ao protocolo HTTP desejado e então analisam-se os dados (*payload*) deste; caso contrário, o pacote é ignorado. Em seguida, é buscada nos dados a sequência que forma o número mágico Ox-CAFEBABE. Para isto, é utilizada a estrutura de dados fornecido pelo SNORT, que verifica os dados brutos (*rawbytes*). O preprocessador foi batizado com o nome de **httpjava**, e se adequou bem ao seu propósito.

O **httpjava** realiza apenas a seleção de pacotes contendo arquivos de classe Java. Não realiza remontagem de sessões; para isto foi utilizado o pré-processador **stream4_reassembly**. A remontagem de sessão é feita apenas nos pacotes vindos do servidor, uma vez que os *applets* são baixados para a máquina cliente por meio de um servidor. Este pré-processador requer o pré-processador **stream4** para executar.

5.2 *Criação do arquivo de regras*

Este filtro se baseia em conjuntos de padrões associados a códigos Java hostis conhecidos. O código é baixado para a máquina do cliente via uma conexão TCP comum, ou seja, usando *three way handshake* com os *flags* convecionais. Em seguida, é feita uma requisição HTTP para que o arquivo de classe possa ser baixado e executado, por meio do comando "GET" e, em resposta a essa requisição; o servidor envia um pacote contendo uma confirmação, ou seja, uma resposta "HTTP/1.1 200 OK".

O SNORT pode executar as seguintes ações default:

- alert: gera uma alerta através de um método criado e depois faz o registro do pacote.
- log: faz o registro do pacote
- pass: ignora o pacote
- activate: alerta e aciona uma regra dinâmica
- dynamic: aguardar até ser ativada por uma regra de ativação.

A chave *content* permite a busca de um conteúdo específico no *payload* do pacote. Múltiplos *contents* são lidos em sequência pelo *engine* do SNORT e são regidos por uma operação AND, ou seja, para ser verdadeiro, todos os conteúdos devem estar presentes na sua sequência. Por necessitar de remontagem de sessão, foram ativados os preprocessadores **stream4** e **stream4_reassembly**, como descrito na seção anterior. A seguir, um arquivo de regras baseado no exemplo da seção anterior:

```
preprocessor httpjava
preprocessor stream4: disable_evasion_alerts
preprocessor stream4_reassemble: both ports 80\
8080

alert tcp any any -> 200.152.19.83 any \
(content:"equals"; content:"stop";
content:"|2AC6|"; content:"|2AB6|"; \
content:"|12|"; content:"|B6|";\
content:"|B12AA6B6|"; msg:"Threadkiller";)
```

Essa regra gera um alerta toda vez que a sequência de múltiplos conteúdos é encontrada.

Foi criado também um outro filtro que contém nomes de classes hostis em Java conhecidos, como exemplo, *NoisyBear.class* (usa recursos de multimídia para irritar o usuário), *Consume.class* (consome a memória), entre outros. Neste filtro foram usadas regras de alerta.

6 RESULTADOS E CONCLUSÕES

A seguir é mostrado o alerta gerada pela regra descrita na seção anterior:

```
[**] [1:0:0] threadkiller [**]
[Priority: 0]
07/22-05:10:40.894496
0:E0:7D:CD:74:70 -> 0:10:B5:6:DE:9
type:0x800 len:0x5EA
64.65.61.213:80 -> 200.152.19.83:33680 TCP
TTL:40 TOS:0x0 ID:11755
IpLen:20 DgmLen:1500 DF
***AP*** Seq: 0xB5CAE996  Ack: 0xA5867E70
Win: 0x7D78  TcpLen: 32
TCP Options (3) => NOP NOP TS: 268618525 \
5326374
```

Este alerta nos mostra também o tipo de código hostil, nesse caso o *threadkiller*. O filtro criado com nomes de classes hostis conhecidos também trouxe resultados satisfatórios, apesar de ter gerado alguns falsos-positivos, pois o nome de algumas classes estavam expostos em páginas da Internet.

Após a inserção do pré-processador no filtro houve uma melhora no desempenho. Os padrões hostis foram reduzidos ao máximo possível mas, ainda assim, eram relativamente grandes, confirmando assim a necessidade do pré-processador *httpjava*. Foram levantados padrões associados com todos os applets hostis conhecidos e esses padrões foram incorporados em regras que estão sendo testadas numa rede de alto volume de tráfego HTTP para determinar a taxa de falsos-positivos.

Neste trabalho apenas são considerados padrões conhecidos de códigos hostis Java e, desta forma, podem existir padrões não detectáveis pelos filtros. Novos pacotes são adicionados com frequência à arquitetura Java e, junto com estes, aparecem novas vulnerabilidades.

O número de classes hostis em Java ativos na rede é muito pequeno, a probabilidade de se encontrar algum é baixa. Isto provavelmente se da porque usuários mal intencionados ainda não descobriram o poder desta linguagem e desconhecem as vulnerabilidades da arquitetura. É apenas uma questão de tempo para que estes aprendam a usar Java para propósitos indevidos.

6.1 Trabalhos futuros

A metodologia usada neste trabalho consistiu em extrair padrões hostis baseados nas caracterísitcas do tráfego de rede associado à linguagem Java. O mesmo pode ser extendido para outras linguagens de código móvel.

REFERÊNCIAS

[1] Last Stage of Delirium Research Group, "Java and JVM Security: Vulnerabilities and their exploitation techniques," 2002. http://lsd-pl.net.

[2] D. Dean, E. Felton, and S. D. Wallach, "Java Security: From Hotjava to Netscape and Beyond," *IEEE Symposium on Security and Privacy*, 1996. Los Angeles, CA, USA.

[3] M. D. LaDue in *Hostile Applets Home Page*. http://www.cigital.com/hostile-applets.

[4] R. Grimes, *Malicious Mobile Code*. O'Reilly, 2001.

[5] J. Forristal and J. Traxler, *Site Seguro: Aplicações Web*. Alta Books, 2002.

[6] L. von DOOM, "A Secure Java Virtual Machine," *9th USENIX Security Symposium*, 2000. Denver, Colorado, USA.

[7] D. M. Jr., S. Rajagopala, and D. Rubin, "Blocking Java Applets at the Firewall," *1997 Symposium on Network and Distributed System Security*, 1997. San Diego, CA, USA.

[8] S. Soman, C. Krintz, and G. Vigna, "Detecting Malicious Java Code Using Virtual Machine Auditing," *12th USENIX Security Symposium*, 2003. Washigton, DC, USA.

[9] "Princeton's Secure Internet Programming Group," 2003. http://www.cs.princeton.edu/sip.

[10] M. Roesch and C. Green, *Snort Home Page*. 2003. http://www.snort.org.

[11] J. Gosling, B. Joy, and G. Steele, *The Java Language Specification*. Addison-Wesley, 1996.

[12] T. Lindholm and F. Yellim, *The JVM Specification*. Addison-Wesley, 1997.

[13] J. Developers, "The Javasoft Home Page," 2003. http://www.javasoft.com.

[14] L. Gong, *Inside Java 2 Platform Security*. Addison Wesley, 1999.

[15] J. Koziol, *Intrusion Detection with SNORT*. SAMS, 2003.

[16] Sourcefire, "SNORT 2.0 - High Performance Multi-Rule Inspection Engine," 2003. http://www.sourcefire.com/technology/whitepapers.html.

[17] J. Beal, J. Foster, and J. Posluns, *SNORT 2.0 Intrusion Detection*. Syngress, 2003.