DESENVOLVIMENTO DE UM MÓDULO PARA DETECÇÃO DE CÓDIGOS JAVA HOSTIS NA REDE

Tantravahi Venkata Aditya
LAC INPE
aditya@tantravahi.com

Antonio Montes
LAC INPE
montes@lac.inpe.br

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.

1 INTRODUÇÃO

Java (*http://www.java.sun.com)* é uma linguagem multi-plataforma, de sintaxe simples, desenvolvido pela Sun Microsystems e com vários recursos úteis para diferentes áreas de aplicação, entre elas a Internet.

Sua popularidade vem crescendo a cada ano, 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 Problema da segurança de aplicações Java

Ao acessar um site na Internet que contém um applet que é executado instantaneamente, uma boa parte dos usuários não tem idéia dos riscos aos quais estão sujeitos. Códigos móveis auto-executáveis são poderosos, práticos e úteis , mas são também perigosos e inseguros.

Falhas na implementação de *browsers* que habilitam Java vem sendo descobertas com razoável freqüência. Essas falhas são tanto no âmbito do sistema, como no âmbito da aplicação. O resultado é uma série de vulnerabilidades que permitem que *applets* Java apaguem ou modifiquem arquivos, capturem informações importantes do sistema e corrompam o sistema do usuário.

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 o Java. 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 confiabilidade do *site* que desejam acessar.

Existem algumas alternativas para remediar esses problemas. Uma delas seria melhorar o modelo de segurança do Java [10] (este modelo será discutido mais adiante), uma outra seria aplicar os *patches*(corretivos) de segurança. Uma terceira seria proteger os usuários por meio do bloqueio de *applets* hostis no firewall, como esta descrito em [6], que pode ser uma boa solução para o problema. Neste trabalho discute-se a possibilidade de se usar um sistema de detecção de intrusão para detectar *applets* hostis, baseado em vulnerabilidades conhecidas, e usar este sistema para interromper a conexão associada.

Nas referências consultadas, são descritos vários tipos de vulnerabilidades e *applets* maliciosos. Em [1], é mostrado um ataque de *DoS (Denial of Service)* e são descritas as informações às quais os *applets* Java tem acesso e outras vulnerabilidades que aparecem por falhas no DNS. Em [2] temos um exemplo de como se pode capturar informações importantes do sistema . Em [3] temos um outro tipo de *DoS*, alem de outros exemplos interessantes. Em [4] temos um exemplo de *honeypot*. Em [5] temos outros dois exemplos de vulnerabilidades do Java. Em [7] temos um exemplo de uma vulnerabilidade que permite que se leia recursos protegidos do sistema de um usuário. Vários outros exemplos podem ser encontrados em [13] e [14]. Em [15] é

discutido o problema da segurança de Java e outros códigos móveis maliciosos.

2 FUNDAMENTAÇÃO TEÓRICA

O Java é uma linguagem simples, distribuída, portável, e interpretada, mas deveria ser também uma linguagem de alto desempenho, robusta, inerte á arquitetura e segura. Infelizmente, esse não é bem o caso. Ela não é segura, não é inerte á arquitetura, pois algumas vulnerabilidades ocorrem am certas arquiteturas e não ocorrem em outras, o desempenho é bom, mas não tanto quanto se esperava. O Java é considerado robusto e, de fato é mais robusto que outras linguagens, mas ainda tem muitas vulnerabilidades.

Para permitir que um código Java seja executado é necessário que o *browser* da máquina esteja habilitado para isso. O código é executado da seguinte forma: o navegador requisita um arquivo que contém um Java *bytecode,* o servidor envia o código que, em seguida, é executado por uma máquina virtual Java que está instalada em sua máquina. Os códigos que são executados na máquina do cliente são conhecidos como *applets*, que foram especialmente desenvolvidos para trabalhar com páginas *Web* da *Internet*. Na seção seguinte explicaremos o que uma aplicação Java pode fazer em uma máquina cliente.

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 do sistema. Cada plataforma tem a sua própria JVM e, em razão disto, os programas precisam ser interpretados antes de serem executados. Isso afeta a linguagem em seu desempenho já que ela se torna mais lenta, mas em compensação ela é executável em qualquer plataforma.

2.2 Modelo de segurança de Java

Permitir a uma linguagem o recurso de poder baixar código móvel executável da rede e ao mesmo tempo torna-la portável, flexível e garantir a ela recursos poderoso, não é trivial. Segurança é um paradigma de custo/benefício, ou seja, faz-se uma segurança altamente rígida e compromete-se a flexibilidade.

Portabilidade e flexibilidade são características necessárias, e Java as possui, mas aumentam o risco de uso malicioso.

No modelo de segurança atual foi desenvolvido um controlador de acesso que verifica o acesso aos recursos do sistema antes do código ser executado. Nesse novo modelo tanto *applets* locais quanto *applets* remotos são obrigados a passar pelas etapas de verificação, ao contrário do modelo clássico que permitia acesso completo aos *applets* locais.

Foram criados zonas de segurança ou domínios desde o mais restrito até o mais acessível. Todo programa Java dentro de um mesmo domínio tem as mesmas permissões.

2.3 Bytecode Java

A especificação da maquina virtual Java representa o arquivo "*.class*" que contém o *bytecode* Java da seguinte forma:

```
Classfile {
        u4 magic;
        u2 minor-version;
        u2 major_version;
        u2 constant_pool_count;
        cp_info constant_pool
[constant_pool_count –1];
        u2 acces_flags;
        u2 this_class;
        u2 super_class;
        u2 interfaces_count;
        u2 interfaces{interfaces_count];
        u2 methods_count;
        method_info methods[methods_count];
        u2 attributes_count;
        attribute_info
attributes[attributes_count];
        }
```

Os 4 bytes iniciais do arquivo "*.class*" (u4 magic) que contém o *bytecode* Java formam uma seqüência numérica denominado número mágico e é através desta seqüência que pode se reconhecer o arquivo Java na rede. Em hexadecimal, esta seqüência é 0xCAFEBABE e não há maneira de se burlar um arquivo Java alterando este numero mágico, pois a JVM inicia a interpretação do código Java apenas após verificar a legitimidade deste número. Na Figura 1 tem se um exemplo de um *bytecode*.

```
0000:0000   CA FE BA BE 00 03 00 2D  00 17 0A 00 05 00 0E 08
0000:0010   00 0F 0A 00 10 00 11 07  00 12 07 00 13 01 00 06      ................
0000:0020   3C 69 6E 69 74 3E 01 00  03 28 29 56 01 00 04 43      <init>...()V...C
0000:0030   6F 64 65 01 00 0F 4C 69  6E 65 4E 75 6D 62 65 72      ode...LineNumber
0000:0040   54 61 62 6C 65 01 00 05  70 61 69 6E 74 01 00 16      Table...paint...
0000:0050   28 4C 6A 61 76 61 2F 61  77 74 2F 47 72 61 70 68      (Ljava/awt/Graph
0000:0060   69 63 73 3B 29 56 01 00  0A 53 6F 75 72 63 65 46      ics;)V...SourceF
```

Figura 1 Exemplo de um bytecode Java

Os 4 bytes seguintes *minor_version* e *major_version* representam a versão do compilador Java que gerou o *bytecode.* O *constant_pool* é uma tabela de estruturas que contém os nomes das classes, dos métodos, das constantes e dos *strings* contidas na classe. Um outro campo interessante é o *method_info* que, como pode ser observado na descrição acima do *Classfile*, não apresenta o número de bytes, pois ele é variável. Neste campo é que está contido o *bytecode* que será executado pela máquina virtual Java, que incorpora informações sobre o tamanho do *bytecode* entre outras.

2.4 Applets Hostis

São aqueles que prejudicam o sistema local de alguma forma. Existem dois tipos de *applets* hostis: os *applets* maliciosos e os *applets* de ataque.

Os *applets* maliciosos são aqueles que atacam a disponibilidade do sistema local de uma destas formas:

- *Denial of Service*
- Invasão de privacidade
- *Applets* Irritantes

Os *applets* maliciosos são encontrados na rede bem mais freqüentemente que os *applets* de ataque, estes últimos são mais perigosos e também mais difíceis de serem implementados. Os *applets* de ataque são aqueles que usam vulnerabilidades do JVM, ou da caixa de contenção, ou seja, vulnerabilidades no modelo de segurança. Estas incluem enganar o verificador, forjar carregadores de classes, travar o gerenciador de segurança, passar por *firewalls,* em resumo explorar erros de implementação. Existe uma boa quantidade desses *applets*, a grande maioria tendo sido desenvolvida por laboratórios que trabalham na correção de erros e usam esse *applets* apenas para realizar testes.

3 METODOLOGIA E RECURSOS

3.1 Sistemas de detecção de intrusão

Intrusão pode ser definida como o uso indevido ou impróprio de um sistema computacional. Detecção de intrusão é uma tecnologia de segurança que permite identificar e isolar intrusões em computadores ou redes. Na atualidade detecção de intrusão é uma ferramenta muito importante para a segurança de redes, como parte de um sistema de defesa em camadas, mas não é a única e nem é o bastante. São necessárias outras ferramentas para garantir um bom nível de segurança, como por exemplo, um *Firewall*.

Sistemas de Detecção de Intrusão em Redes (*SDIR*), são sistemas que monitoram o tráfego de rede e procuram detectar determinados tipos de intrusões, analisando padrões suspeitos de atividade na rede.

3.2 Criação de um filtro para o SNORT

Neste trabalho, resolveu-se utilizar o SNORT como ferramenta de detecção de intrusão. O SNORT é um detetor de intrusão, que realiza análise de tráfego em tempo real e registra pacotes trafegando em redes TCP/IP usando uma linguagem de descrição de regras que é simples, flexível e poderosa. Ele funciona em qualquer sistema que tenha o *libpcap* (biblioteca de captura de pacotes que permite fazer uma análise de tráfego de rede em baixo nível). O SNORT registra pacotes no formato binário do *tcpdump* ou em seu próprio formato ASCII. Ele possui um modo detetor de intrusão em redes de computadores, um de registro de pacotes e um *sniffer.*

Foi implementado um módulo de detecção de *applets* Java hostis que usa o SNORT para fazer uma

análise do tráfego na rede, buscando pacote que contenham arquivos de extensão "*.class*" e examinando o conteúdo desses pacotes. Em seguida é feita uma comparação com um conjunto de padrões associados a códigos Java hostis conhecidos. Caso algum desses padrões coincida, o pacote será registrado e a execução deste código poderá ser interrompida.

O módulo detecta um programa Java através do numero mágico, 0xCAFEBABE, como foi discutido na seção 2.2.3. Isto será feito capturando apenas o trafego TCP que destinado à porta definida para o serviço httpd, na maioria dos casos a porta 80. Cria se uma variável local $HTTP_PORTS com as portas que se deseja monitorar.

A seguir é mostrado um pequeno trecho de código que executa uma atividade hostil:

```
   {
    public void
actionPerformed(ActionEvent evt)

       {
            for (int i = 0; i < 105;
i++)
              {
              Ball b = new Ball(canvas,
Color.red);

b.setPriority(Thread.NORM_PRIORITY + 2);
                    b.start();
              }
        }
```

Este trecho de código executa 105 *threads* consumindo por completo o processador da máquina e conseqüentemente fazendo-a travar. O *bytecode*

correspondente é o seguinte:

```
56 01 00 10 6A 61 76 61  2F 6C 61 6E 67
2F 54 68
72 65 61 64 01 00 0B 73  65 74 50 72 69
```

Esse *bytecode* contém o método *start()* que inicia uma *thread* em Java e o *loop* que executa esta *thread* 105 vezes.

Será criada uma regra que ao enxergar o número mágico identificara que o programa é Java. Para tornar estas regras mais eficientes é necessário ter conhecimento do comportamento da rede quando se acessa um sítio que contenha um Java. Usando um *sniffer* para capturar o trafego na rede pode-se entender melhor este comportamento e a partir dele configurar e otimizar as regras que irão compor o módulo de detecção do código hostil Java .

```
12 22.202738  200.152.7.142  150.163.96.55  TCP   1263 > http [SYN] Seq=2463941143 Ack=0 Win=16384 Len=0
13 22.252888  150.163.96.55  200.152.7.142  TCP   http > 1263 [SYN, ACK] Seq=2681032563 Ack=2463941144 Win=5840  Len=0
14 22.253071  200.152.7.142  150.163.96.55  TCP   1263 > http [ACK] Seq=2463941144 Ack=2681032564 Win=17520 Len=0
15 22.253425  200.152.7.142  150.163.96.55  HTTP  GET /HelloWorldApp.class HTTP/1.1 200 ok
16 22.320504  150.163.96.55  200.152.7.142  TCP   http > 1263 [ACK] Seq=2681032564 Ack=2463941364 Win=6432 Len=0
17 22.345015  150.163.96.55  200.152.7.142  HTTP  HTTP/1.1 200 OK
```

Figura 2 Captura de trafego de rede usando o SNORT

Observando o tráfego mostrado na figura 2 notamos que é uma conexão TCP comum, ou seja, usando *three way handshaking* com os *flag*s normais. Em seguida se faz uma requisição para o *http* para que o arquivo "*.class*" possa ser baixado e executado, feito por meio do comando "`GET /HelloWorldApp.class`" e, em resposta a essa requisição, o servidor envia um pacote contendo a seqüência "`HTTP/1.1 200 OK`". Na figura 2, o endereço IP do servidor e' 200.152.7.142 e o do cleinte e' 150.163.96.55. Ao examinar o conteúdo deste pacote constatamos que é nele que se encontra o número mágico 0xCAFEBABE indicando assim que é um programa Java.

Visto isto, observamos que o Snort pode executar cinco 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 – alertar e acionar uma regra dinâmica (dynamic).

dynamic – Aguardar até ser ativada por uma regra de ativação (activate).

Assim, usando o SNORT foi criada a seguinte regra:

```
activate tcp any any ->
200.152.7.68 $HTTP_PORTS (content:"|CAFEBABE|"; content:"|560100106A6176612F6C616E672F548|";
 activates: 1; msg:"HOSTILE JAVA";)

  dynamic tcp any any -> 200.152.7.68 (activated_by:1; count:5;)
```

Figura 3 Regra de detecção de um trecho de código hostil de rede usando o SNORT

Essa é uma regra dinâmica que é ativada toda vez que é detectado um conteúdo com o hexadecimal 0xCAFEBABE associado ao padrão hostil 560100106A6176612F6C616E672F548. A busca pelo padrão hostil em questão é feita nos 5 pacotes seguintes. Para se ter uma maior eficiência pode se acrescentar na regra um campo que faça com que apenas tráfego HTTP seja examinado.

3.3 Resultados

Abaixo é mostrado o alerta gerado pela regra descrita na seção anterior.:

```
[**] [1:0:0] HOSTILE JAVA [**]
07/21-16:13:02.755267 0:50:BF:5D:1:A0 -> 0:10:B5:6:DE:9
type:0x800 len:0x2E7
200.222.209.204:80 -> 200.152.7.68:32863 TCP TTL:53 TOS:0x0
ID:9501 IpLen:20 DgmLen:729
***AP*** Seq: 0xCD15FFF7  Ack: 0x15A6850  Win: 0xFFFF
TcpLen: 32
TCP Options (3) => NOP NOP TS: 322440 343113
```

O SNORT também gera outro arquivo com o conteúdo deste pacote, que contem o código malicioso.

Foi criado um modulo que contem uma lista de padrões de códigos hostis em Java conhecidos. Isto foi feito usando se a chave *content-list* do SNORT, esta função chama o arquivo criado contendo os padrões hostis.

Content-list: [!] ”Hostile Java”

Um outro filtro foi criado que contem nomes de classes hostis em Java conhecidos como *NoisyBear.class* (*applet* que usa recursos de multimídia para irritar o usuário), *Consume.class* (*applet* que consome a memória) e *DiskHog.class* (*applet* que consome o disco rígido).

3.3.1 Filtros em fase de teste

Outros filtros estão sendo testados e também estamos estudando o uso da facilidade do SNORT de se incluir um TCP RESET, por meio da chave *resp* na regra de filtragem para interromper a conexão. Esse sistema tem a vantagem de só interromper a conexão corrente, ao contrário de outros sistemas[6] que implementam filtros em *firewalls* que bloqueiam o endereço IP de origem e podem se prestar a ataques de DoS. Uma outra alternativa de bloqueio de conexões , com o SNORT, seria o uso da chave *react* que possui argumentos para bloqueio e para alerta, esta sendo testado um filtro com esse recurso.

Os padrões de código Java considerads visivelmente hostis são aqueles intencionalmente criados para prejudicar outros sistemas, por exemplo o que esta descrito na seção anterior. Devem ser interrompidos antes de serem executados. Existem outros que não, por exemplo uma aplicação que escreve em disco, em alguns casos isto se torna ferramenta útil por isso não deve ser interrompido, deve-se apenas gerar um alerta. O intuito de fazer isso é minimizar os falsos-positivos.

Usando o *content-list* foi criado um arquivo com padrões visivelmente hostis e está sendo criado outro com aqueles que podem ser hostis, assim com a chave *react* ou TCP RESET pode se bloquear os visivelmente hostis e gerar alertas para aqueles que são aparentemente hostis.

4 CONCLUSÃO

As regras criadas até o presente momento estão funcionando como previstos, mas como um dos objetivos desse trabalho é a detecção de *applets* hostis em tempo real, um outro cuidado que estamos tomando é o de buscar os menores padrões possíveis, visando um melhor desempenho. Há uma necessidade de se otimizar as regras, estudando melhor o comportamento rede, para melhorar o seu desempenho. Espera-se que estas regras permitam a detecção de códigos Java hostis em tempo real minimizando o número de falso-positivos.

Neste trabalho apenas são detectados padrões conhecidos de códigos hostis Java, podem existir

padrões hostis que não sejam detectados pelas regras contidas no módulo, pois a *Sun Microsystems* oficializa novos pacotes Java com freqüência. Para amenizar este problema deve-se sempre atualizar as regras com eventuais novos padrões hostis.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] DEAN, D. ; FELTEN, E. W. ; WALLACH D. S.; *Java security: From Hotjava to Netscape and Beyond, IEEE Symposium on Security and Privacy, 1996*

[2] OAKS, S.; *Segurança de dados em Java , Editor ciência Moderna,1999*

[3] FELTEN, E. W.; *Securing Java, www.securingjava.com, 1999*

[4] SANTOS, A. L.; *Two Java bugs,* UCSB, 1998

[5] BILLON, J.; *Java Security Weakness and Solutions, 1997*

[6] MARTIN, Jr, D. M., RAJAGOPALA , S., RUBIN D. A.; *Blocking Java Applets at the Firewall, 1997*

[7] CERT Advisories, www.cert.org/advisories/CA-2000-15.html , 2000

[8] Writing Snort Rules, www.snort.org

[9] CORNELL, G; HORSTMANN, C. S.; Core Java 2, Prentice Hall, 1999

[10] von DOOM, L.; *A Secure Java Virtual Machine,* 9th USENIX Security Symposium, 2000

[11] ROBERTS, S; *Complete Java 2 Certification Guide,* Sybex , 2000

[12] HAROLD, E. R.; *Java Network Programming*, O'Reilly, Second Edition, 2000

[13] Hostile Java, www.megasecurity.org/Hostile_java_, 2002

[14] Hostile Applets Home Page www.cigital.com/hostile-applets , 2002

[15] GRIMES, R. A. *Malicious Mobile Code,* 0'Reilly , 2001

[16] Digicrime site www.digicrime.com , 2002

[17] KOLAWA, Adam, *To Java or not to Java,* www.unixreview.com/print/documentID=15876, 2001

[18] Snortsam plugin www.snortsam.net, 2002