José Duarte de Queiroz

MICÆL: Uma Arquitetura para Detecção de Atividades Intrusas Baseada em Agentes Móveis de Software

Instituto de Matemática – Mestrado em Informática

Luci Pirmez

D.Sc. - COPPE/UFRJ - Brasil –1996

Luiz Fernando Rust da Costa Carmo

Docteur, Université Paul Sabatier/LAAS - França - 1995.

Rio de Janeiro

2001

MICÆL: Uma Arquitetura para Detecção de Atividades Intrusas Baseada em Agentes Móveis de Software

José Duarte de Queiroz

Dissertação (Tese) submetida ao corpo docente do Instituto de Matemática da Universidade Federal do Rio de Janeiro - UFRJ, como parte dos requisitos necessários à obtenção do grau de Mestre.

Aprovada por:

_____

Profª. Luci Pirmez,
DSc. COPPE/UFRJ

_____

Prof. Luiz Fernando Rust da Costa Carmo,
Dr. UPS., France

_____

Prof. Oswaldo Vernet de Souza Pires,
D.Sc. COPPE/UFRJ

_____

Prof. José Ferreira de Rezende,
Ph.D. Université Pierre et Marie-Curie, Paris - France

Rio de Janeiro - RJ
Março de 2001

FICHA CATALOGRÁFICA

QUEIROZ, JOSÉ DUARTE DE.

MICÆL: Uma Arquitetura para Detecção de Atividades Intrusas Baseada em Agentes Móveis de Software [Rio de Janeiro] 2001

xii, 85 p., 29,7 cm (IM/NCE/UFRJ, MSc., Informática, 2001)

IM/NCE/UFRJ, 2001.

Dissertação - Universidade Federal do Rio de Janeiro, IM/NCE

1. Segurança 2. Redes 3. IDS 4. Agentes Móveis

I. IM/NCE/UFRJ II. Título (série)

À minha esposa Ana Paula e meu filho Gabriel,

e

em memória do Prof. Júlio Salek Aude.

"Houve peleja no céu. Miguel e seus anjos pelejaram contra o dragão. Também pelejaram o dragão e seu anjos. Todavia não prevaleceram, nem mais se achou no céu o lugar deles. E foi expulso o grande dragão, a antiga serpente, que se chama Diabo e Satanás, o sedutor de todo mundo, sim, foi atirado para a terra e, com ele, seus anjos.". (Apocalipse 12:7-9)

AGRADECIMENTOS

A Deus, pela permissão de vencer este desafio.

A São Miguel Arcanjo, padroeiro e inspirador deste trabalho.

Aos meus pais, que com seu esforço e dedicação permitiram que eu chegasse até aqui.

À minha esposa Ana Paula, e meu filho Gabriel, que souberam compreender a quase ausência do marido e pai, e com seu carinho me deram forças para prosseguir nos momentos em que o cansaço e o desânimo tentaram me dominar.

Aos amigos do curso de mestrado, que sempre estiveram por perto nas horas em que deles precisei.

Aos meus orientadores, pela atenção e carinho recebidos no desenvolvimento deste trabalho, e aos professores do curso de Mestrado em Informática do Instituto de Matemática da UFRJ, pelas informações valiosas recebidas durante o curso.

À direção superior do Ministério Público Federal, pelo apoio recebido na forma da flexibilização do horário de expediente.

Ao Núcleo de Computação Eletrônica da Universidade Federal do Rio de Janeiro, pela magnífica infra-estrutura disponibilizada durante o curso.

Resumo da Tese apresentada ao IM/NCE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

MICÆL: Uma Arquitetura para Detecção de Atividades Intrusas Baseada em Agentes Móveis de Software

José Duarte de Queiroz

Março/2001

Orientadores: Prof[a]. Luci Pirmez
Prof. Luiz Fernando Rust da Costa Carmo

Departamento: Informática

Neste documento será descrito um projeto de pesquisa onde se objetiva a implantação de um Sistema de Detecção de Intrusos com o uso de Agentes de Software Autônomos. Um Sistema de Detecção de Intrusos é uma ferramenta de administração de sistemas que identifica e reage às tentativas de intrusão e uso não autorizado. Esses agentes usarão mecanismos de mobilidade, possibilitando uma melhor distribuição dos recursos utilizados com degradação mínima do desempenho percebido pelos usuários.

Abstract of Thesis presented to IM/NCE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

MICÆL: An Architecture to Detect Intrusive Activities based on Mobile Software Agents

José Duarte de Queiroz

March/2001

Advisors: Luci Pirmez, D.Sc.
Luiz Fernando Rust da Costa Carmo, Dr, UPS

Department: Informatics

Here we present a research project to create and deploy an Intrusion Detection System based on Autonomous Mobile Software Agents. An Intrusion Detection System is an administration/management tool that identifies and reacts to intrusion and unauthorized use attempts. These agents will use mobility facilities, allowing an efficient use of resources, by dynamically distributing processing tasks, with a minimal degradation of the performance perceived by users.

Sumário

CAPÍTULO 1 - INTRODUÇÃO 1

CAPÍTULO 2 - CONCEITOS DE SEGURANÇA 3

2.1 SEGURANÇA 4

2.2 ATAQUES 4

2.3 INVASÕES 6

2.4 RISCOS E MEDIDAS DE SEGURANÇA 7

2.5 AMEAÇAS INTERNAS E EXTERNAS 8

2.5.1 Ameaças às Máquinas 9

2.5.2 Ameaças às Comunicações 9

2.6 MECANISMOS DE SEGURANÇA 10

2.6.1 Mecanismos de Proteção às Máquinas 11

2.6.2 Mecanismos de Proteção às Comunicações 12

2.7 MECANISMOS DE DETECÇÃO DE ATAQUES 12

2.8 SISTEMAS DE DETECÇÃO DE INTRUSOS (IDS) 13

2.9 O IDS COMO ALVO DE ATAQUES 15

2.10 CONSIDERAÇÕES FINAIS DO CAPÍTULO 15

CAPÍTULO 3 - CONCEITOS BÁSICOS 17

3.1 AGENTES 18

3.2 AGENTES MÓVEIS 19

3.3 O PARADIGMA DE ORIENTAÇÃO A OBJETOS 21

3.4 TRABALHOS RELACIONADOS 22

3.5 CONSIDERAÇÕES FINAIS DO CAPÍTULO 23

CAPÍTULO 4 - ARQUITETURA PROPOSTA 25

4.1 COMPONENTES DO SISTEMA 27

4.1.1 Quartel General 27

4.1.2 Sentinelas 29

4.1.3 Destacamentos 32

4.1.4 Auditores 33

4.1.5 Agentes Especiais 35

4.2 CARACTERÍSTICAS DE MOBILIDADE 36

4.3 COMUNICAÇÕES ENTRE AGENTES 37
4.4 CONSIDERAÇÕES FINAIS DO CAPÍTULO 38

CAPÍTULO 5 - IMPLEMENTAÇÃO 39

5.1 LINGUAGEM DE PROGRAMAÇÃO 40
5.1.1 O problema do Desempenho 40
5.1.2 Heranças Múltiplas 41
5.1.3 Estrutura Hierárquica 41
5.1.4 Objetos Predefinidos 42
5.2 AMBIENTE DE MOBILIDADE 43
5.2.1 ASDK Versões 1.1b3, 1.1 e 1.2 43
5.2.2 Conceitos básicos do ASDK 44
5.3 O AMBIENTE OBJETO 46
5.3.1 O Ambiente Windows NT 46
5.3.2 O Ambiente Linux 47
5.4 ESTRUTURA DO CÓDIGO 48
5.5 COMUNICAÇÃO ENTRE OS AGENTES 50
5.6 DETECÇÃO DE TRÁFEGO DE REDE 50
5.6.1 O Pacote `jdq.libpcap2` 50
5.6.2 O pacote `jdq.NetAnalyser` 51
5.6.3 O Pacote `jdq.AntiScan` 53
5.6.4 O Pacote `jdq.IPGen` 53
5.7 EXEMPLO DE DESTACAMENTO: DESTACAMENTO ANTIFLOOD 53
5.8 CONSIDERAÇÕES FINAIS DO CAPÍTULO 54

CAPÍTULO 6 - ANÁLISE DOS RESULTADOS 56

6.1 GERAÇÃO DA MASSA DE DADOS 56
6.2 CONFIGURAÇÃO DO AMBIENTE DE TESTE 58
6.3 EXECUÇÃO DOS TESTES 58
6.4 RESULTADOS DOS TESTES 59
6.5 CONSIDERAÇÕES FINAIS DO CAPÍTULO 61

CAPÍTULO 7 - CONCLUSÕES FINAIS 63

APÊNDICE A - A LINGUAGEM JAVA 65

A.1 INDEPENDÊNCIA DE PLATAFORMA 65

A.2 A MÁQUINA VIRTUAL JAVA (JVM) 66
A.3 MULTI-THREADING 67
A.4 O MODELO JAVA DE GERENCIAMENTO DE MEMÓRIA 68
A.5 CONTROLE DE ACESSO E SEGURANÇA 68
A.6 A INTERFACE JAVA – NATIVO 69

APÊNDICE B - PRINCIPAIS TÉCNICAS DE ATAQUE 70
B.1 FORÇA BRUTA 70
B.2 ENGENHARIA SOCIAL 70
B.3 FALSA IDENTIFICAÇÃO E NEGAÇÃO DE AUTORIA 71
B.4 EXPLORAÇÃO DE ACESSOS OCULTOS (*BACKDOORS*) 71
B.5 PERSONIFICAÇÃO DE ENDEREÇO (*ADDRESS SPOOFING*) 71
B.6 CAPTURA DE CONEXÕES (*HIJACKING*) 72
B.7 BLOQUEIO OU PRIVAÇÃO DE SERVIÇO (DOS) 72
B.8 ALTERAÇÃO, INSERÇÃO OU REMOÇÃO DE MENSAGENS 73
B.9 CRIPTOANÁLISE 73

APÊNDICE C - EXEMPLOS DE SISTEMAS DE AGENTES MÓVEIS 74
C.1 TELESCRIPT 74
C.2 ODISSEY 74
C.3 CONCORDIA 74
C.4 VOYAGER 75
C.5 AGENT TCL 75
C.6 ARA 75
C.7 TACOMA 75
C.8 ASDK 76

APÊNDICE D - TÉCNICAS DE PORT SCANNING 77
D.1 PING SCAN 77
D.2 TCP CONNECT SCAN 77
D.3 TCP SYN STEALTH SCAN 78
D.4 UDP SCAN 78
D.5 TCP FIN STEALTH SCAN 78
D.6 OS SCAN 79

REFERÊNCIAS 80

Índice de Figuras

Figura 4.1– Exemplo do Sistema Micæl para uma rede com três *Hosts*. 25
Figura 4.2- Domínios de Controle definidos pelo Quartel General 29
Figura 4.3 - O Auditor em Funcionamento 33
Figura 5.1 - Trecho de código extraído do protótipo exemplificando a herança múltipla 41

Índice de Tabelas

Tabela 2.1 – Exemplos de pessoas com motivos para invadir um sistema (adaptado de [30]) 7
Tabela 6.1 - Teste 1: Tempo de Carga do Sistema (em Segundos) 60
Tabela 6.2 - Teste 2: Carga de um destacamento (em milissegundos) 60

Capítulo 1 - Introdução

O advento de computadores, que a cada dia ficam mais rápidos e mais baratos, permitiu (e incentivou) o uso de quantidades cada vez maiores de informações. Ao mesmo tempo, colocou essas informações ao alcance de mais pessoas. Essa facilidade de acesso às informações cobrou seu preço: a perda do controle sobre elas. Foi necessário, então, criar mecanismos que reforçassem e garantissem o controle sobre as informações armazenadas nos computadores, e por eles veiculadas.

Com o advento das redes de computadores, e em especial da Internet, o acesso a fontes de informações longínquas ficou muito mais fácil. E um novo problema foi criado: os mecanismos de controle que antes existiam se tornaram insuficientes. Novos mecanismos se tornaram necessários. Porém, o que se verifica hoje é que muitas vezes as paredes da fortaleza construída para a defesa das informações estão construídas sobre alicerces fracos, e que embora a porta seja trancada com sete cadeados, as janelas estão abertas. Acontece que muitas vezes os programas e sistemas usados nos computadores apresentam falhas, tanto na sua construção como no seu uso, permitindo que pessoas não autorizadas tenham acesso aos dados nele armazenados. O problema na verdade é maior, uma vez que os recursos disponíveis no computador são limitados. Deixar uma pessoa qualquer utilizá-los é, no mínimo, desperdício.

Quando se leva em conta que boa parte desses computadores está em atividade 24 horas por dia, e que é muito difícil para um operador manter vigilância constante sobre todas as atividades da máquina, ainda mais quando estas se desenrolam a taxas de milhares ou mesmo milhões por segundo, percebe-se a necessidade de ferramentas de vigilância e apoio à administração. Quando essas ferramentas compõem um sistema dedicado à prevenção do uso da máquina por pessoas não autorizadas, têm-se os chamados *Sistemas de Detecção de Intrusos* (IDS, do inglês *Intrusion Detection System*).

A abordagem clássica para a construção deste tipo de ferramenta prevê dois tipos de IDS: o HIDS (*Host* IDS), que trabalha com informações locais, internas à máquina, e é capaz de detectar tentativas de ataque oriundas dos seus usuários; e o NIDS (*Network* IDS), que trabalha com informações da rede a que os computadores estão ligados, e é capaz de identificar tentativas de ataque oriundas de outras máquinas.

Este trabalho propõe uma arquitetura que pretende unir as qualidades dos dois tipos de IDS distribuindo pequenos módulos independentes de código – os agentes – por todas as máquinas pertencentes a um determinado domínio de rede. Aliado com a capacidade de mobilidade, torna-se possível uma configuração simples e flexível deste sistema, com uma maior disponibilidade. Os agentes, agindo de forma coordenada, possibilitam a utilização racional dos recursos das máquinas da rede, de forma que esses recursos são alocados apenas no momento em que são realmente necessários.

Os capítulos 2 e 3 expõem as premissas de segurança e a base teórica que nortearam o desenvolvimento da arquitetura. No capítulo 4 é definida a arquitetura propriamente dita. O capítulo 5 contém a descrição dos passos envolvidos na implementação do protótipo de avaliação da arquitetura. No capítulo 6 estão os resultados obtidos do protótipo. Finalmente, o capítulo 7 discorre sobre a experiência do projeto, com as conclusões obtidas. Completando, quatro apêndices: o apêndice A explica alguns aspectos da linguagem Java, que é usada para implementar o protótipo da arquitetura descrita neste trabalho; o apêndice B exemplifica técnicas de ataque freqüentemente utilizadas; o apêndice C lista alguns exemplos de sistemas de agentes móveis, além do utilizado para a implementação do protótipo da arquitetura; e o apêndice D explica o processo de *port scan* (varredura de portas TCP/UDP). Para simplificar a consulta, uma cópia dos programas desenvolvidos no projeto foi disponibilizada na mídia anexa.

Capítulo 2 - Conceitos de Segurança

Este capítulo apresenta conceitos iniciais de segurança. As soluções adotadas na implementação da arquitetura proposta neste trabalho são baseadas nestes conceitos. Esses conceitos possibilitam também a definição de critérios para avaliar a qualidade da arquitetura proposta como ferramenta de segurança e de sua implementação.

De maneira geral, a segurança de um sistema de computadores é composta pela confiabilidade e disponibilidade do *hardware* e do *software* utilizado nesse sistema, bem como o controle de acesso aos recursos fornecidos pelo sistema. Além disso, em ambientes multi-usuário, deve haver a garantia de que as tarefas executadas por um usuário não atrapalhem a execução das tarefas dos outros usuários. Entretanto, a arquitetura proposta neste trabalho trata apenas das questões de uso não autorizado, assumindo que as outras questões já estão resolvidas.

As preocupações de segurança para um computador isolado são diferentes das para outro, ligado a uma rede. No computador isolado, é necessário controlar apenas o acesso local aos seus recursos. Já no ligado à rede é necessário, além de controlar o acesso local, controlar o acesso remoto e proteger os meios de comunicação usados.

Invasores e atacantes são personagens dos incidentes de segurança. Várias classificações de invasores e atacantes são possíveis, e as medidas de segurança a serem adotadas dependem dessas classificações. Fatores como a experiência do atacante, a sua origem, ou os meios usados para desferir o ataque devem ser levados em conta.

A dificuldade em manter todos esses fatores sob controle gera a necessidade de ferramentas automatizadas de manutenção da segurança dos sistemas de computadores. Um grupo dessas ferramentas é conhecido como Sistemas de Detecção de Intrusos. A arquitetura proposta neste trabalho é um exemplo de Sistema de Detecção de Intrusos.

Este capítulo inicia definindo conceitos básicos de segurança e definindo termos como *ataque*, *invasão*, *hacker*, *política de segurança*, *backdoor*, e outros. Em seguida são avaliadas as diferenças entre ameaças internas e externas, bem como os riscos de segurança. Define-se mecanismos de proteção e detecção, para depois discorrer sobre características de Sistemas de Detecção de Intrusos. Por último, são citados alguns exemplos de técnicas de ataque.

2.1 Segurança

Segurança de um Sistema de Computadores é uma medida de quanto se pode confiar nele. Um Sistema de Computadores é seguro se consegue garantir as características definidas a seguir.

- **Disponibilidade** – o sistema deve estar disponível para uso quando seus usuários precisam dele. Sistemas críticos precisam estar permanentemente disponíveis.
- **Utilidade** – o sistema e os recursos disponíveis devem ser úteis para o propósito designado pelos usuários.
- **Integridade** – os dados armazenados devem ser recuperados integralmente, sem alterações ou perdas.
- **Autenticidade** – o sistema deve ser capaz de verificar a identidade de seus usuários antes de lhes conceder acesso aos recursos, e estes devem ser capazes de verificar a identidade do sistema antes de lhe confiar seus dados.
- **Privacidade** – dados confidenciais devem ser visíveis apenas pelos seus donos, ou por pessoas escolhidas por eles.
- **Posse e Controle** – os donos de um sistema devem ser capazes de controlá-lo. Perder o controle de um sistema para um usuário malicioso afeta a segurança do sistema para todos os usuários.

Os responsáveis pela administração do ambiente, ao decidirem que tipos de operações podem ser executadas, e quem pode acessar quais recursos, definem uma **política de segurança**. A política de segurança pode ser *restritiva* (tudo que não é explicitamente permitido é proibido) ou *permissiva* (tudo que não é explicitamente proibido é permitido). A partir da definição da política de segurança, é possível caracterizar **ataques** e **invasões**.

2.2 Ataques

As tentativas de violação da política de segurança são conhecidas como a**taques**. Uma das tarefas do administrador de segurança é perceber quando esses ataques ocorrem e tomar as medidas necessárias, seja fechando as brechas que estão sendo exploradas, seja identificando o atacante e acionando as autoridades competentes ou mesmo contra-atacando, de forma a eliminar a capacidade de ação do atacante. Um resultado

comum de um ataque é o bloqueio do alvo, ou o impedimento do uso de algum de seus recursos pelos seus usuários legítimos. Os ataques que produzem este efeito são denominados de **negação de serviços** (DoS, do inglês *Denial of Services*). Os ataques recebem diferentes classificações conforme suas características, que são listadas abaixo.

- **Intencionalidade – i**ndica se o ataque foi desferido intencionalmente ou não. Ataques intencionais tendem a provocar prejuízos maiores para o ambiente atacado.
- **Objetivo** – o objetivo de um ataque pode ser o roubo de informações, ganhar o controle sobre a máquina atacada ou impedir o seu uso. Pode ser também que o ataque faça parte de um ataque maior, coordenado. Outra possibilidade é do ataque ser puro vandalismo, não tendo nenhum objetivo em particular.
- **Formato** – característica que possibilita que ele seja identificado durante a sua execução. Normalmente os ataques geram um padrão de eventos. Este padrão pode ser reconhecido por ferramentas automatizadas.
- **Origem** – indica se o ataque tem origem local ou remota. O responsável pelo ataque também pode ser classificado como interno ou externo em relação à entidade responsável pelo ambiente.
- **Risco** – indica o grau de comprometimento provocado no sistema caso o ataque tenha sucesso.
- **Meio de acesso** – indica o meio utilizado pelo atacante para desferir o ataque. Pode ser a rede local, um link de médio/longo alcance, o console, uma linha serial (terminal serial), ou alguma mídia magnética.

Uma outra classificação possível dos ataques é o nível de conhecimento exigido de seus autores para a sua realização, conforme abaixo.

- **Nível principiante** – são ataques que não exigem conhecimento profundo para serem realizados. Normalmente são de baixo risco, sendo detectados facilmente pelo próprio sistema operacional. O praticante contumaz deste tipo de ataque é conhecido na comunidade *hacker* como "*lammer*".
- **Nível básico** – são ataques em que o atacante precisa possuir algum conhecimento, tal como técnicas de ataque, ferramentas, vulnerabilidades comuns, etc. Ainda assim, é de baixo risco, e apesar de normalmente não ser detectado pelo SO, é facilmente detectável por um sistema de detecção de intrusos (IDS). O praticante deste tipo de ataque é conhecido como "*script kiddy*".

- **Nível Experiente** – são ataques em que o atacante precisa possuir um conhecimento extenso tanto das técnicas de ataque como das vulnerabilidades e das ferramentas de ataque. O risco envolvido é bem maior, mas ainda é detectável por IDS.
- **Nível Avançado** – são ataques em que o autor possui um conhecimento profundo de características do SO e da arquitetura da máquina alvo. As ferramentas utilizadas para desfechar esses ataques são muito elaboradas, e o seu autor utiliza variadas técnicas para enganar os sistemas de defesa e disfarçar suas atividades. Por essa razão, são de difícil detecção pelo IDS, o que multiplica o seu risco. O atacante que tem conhecimento para desferir este ataque é muito respeitado na comunidade *hacker*, sendo conhecido como "*master*" ou "*wizard*".

2.3 Invasões

Quando uma pessoa não autorizada consegue acesso aos recursos de um sistema, contrariando a política de segurança em vigor, diz-se que ocorreu uma **invasão**. Os **invasores** aproveitam-se de **vulnerabilidades** e de **brechas de segurança** para entrar no Sistema. **Vulnerabilidades** são falhas no ambiente operacional (máquina, sistema operacional e protocolos de rede) que impedem o cumprimento das determinações da política de segurança, possibilitam o uso do sistema por usuários sem a devida autorização. Já as brechas de segurança normalmente se originam de má administração. Entre os pontos que podem ser explorados por atacantes, temos:

- erros na configuração das medidas de segurança;
- falhas na implementação de software ou hardware;
- não aplicação de correções (*patches*) eliminando as vulnerabilidades conhecidas;
- entradas deixadas propositalmente abertas para facilitar o acesso (*backdoors*);
- entradas criadas por programas com código malicioso (cavalos de tróia).

Existem também os casos em que o ambiente não está preparado para identificar um acesso como não autorizado. No ambiente *WinTel*[1], por exemplo, não há diferencia-

[1] Nome como é conhecida a onipresente combinação Windows 95 ou 98 com processadores Intel Pentium

ção entre usuário comum e supervisor e nem entre usuários diferentes. Assim, não há como determinar se o usuário pode ou não acessar um determinado arquivo. Essa situação permite que qualquer usuário modifique arquivos importantes, tais como os componentes do Sistema Operacional, tornando a máquina extremamente sujeita a funcionamento incorreto. Este caso não caracteriza propriamente uma vulnerabilidade, porém representa um ponto fraco na definição de uma política de acesso.

2.4 Riscos e Medidas de Segurança

Risco é uma medida do potencial de prejuízo que pode ser causado por um incidente. Os sistemas de computadores interligados através de redes estão sujeitos a variados níveis de risco devido a ataques, que podem ser classificados de acordo com a origem do ataque, com a área afetada, com o método usado e com o objetivo do atacante.

Adversário	Alvo	Motivação	Nível de Risco
Usuário Legítimo	Sistema em Execução	Erro de operação	Baixo
Estudante	Ler mensagens de correio eletrônico e arquivos pessoais alheios	Diversão; Curiosidade	Baixo
Hacker	Atacar os computadores de uma rede, aparentemente sem provocação	Conhecer os mecanismos de segurança usados	Médio
Representante de Vendas	Ler mensagens de correio eletrônico alheias	Roubar clientes da concorrência	Médio
Ex-funcionario	Bloquear o Sistema Central ou Destruir o Banco de Dados Corporativo	Vingança por ter sido demitido	Médio/Alto
Funcionário	Ler mensagens de correio eletrônico alheias	Chantagear para ascender profissionalmente	Médio/Alto
Corretor de Valores	Alterar ou ler mensagens de correio eletrônico	Negar promessas de vendas; conseguir "*insights*".	Médio/Alto
Espião militar	Bloquear Sistemas de Vigilância militar	Impedir a reação a um ataque inimigo	Alto
Criminoso	Sistemas Policiais e Jurídicos	Impedir investigações ou processos criminais	Médio/Alto

Tabela 2.1 – Exemplos de pessoas com motivos para invadir um sistema (adaptado de [30])

A classificação do nível de risco é feita com base na análise do tipo atacante, isto é, do seu nível de conhecimentos, sua tenacidade, e dos recursos disponíveis para ele. A importância dos sistemas atacados é um fator multiplicador do risco. Por exemplo, um ataque a um sistema escolar é considerado de baixo risco; já um ataque a um sistema de controle de vôo é considerado de risco altíssimo. Independentemente do tamanho e da finalidade do sistema, o risco de segurança é uma parte considerável do Custo Total de Propriedade (TCO - *Total Cost of Ownership*).

2.5 Ameaças Internas e Externas

Ambientes de Rede têm a fama de serem porta de entrada para invasões e ataques, já que normalmente atribui-se a culpa pelos incidentes de segurança a agentes externos. As estatísticas provam, porém, que a maioria das violações de segurança partem, na verdade, de usuários diretamente relacionados com a própria empresa. Estudos recentes [31] indicam uma modificação nesse perfil; a tendência atual é de crescimento das violações de segurança provocadas por agentes externos, principalmente prestadores de serviço (o que não é nenhuma surpresa, uma vez que esses indivíduos têm acesso às instalações da empresa, sem ter um vínculo mais profundo com ela).

Mesmo assim, não é possível negligenciar o grande risco de segurança gerado pelo uso de um ambiente de rede público, não controlado, como a Internet. A disseminação massiva do acesso à Internet, aliada ao pouco conhecimento de suas características pelos usuários, terminou por criar novos meios de acesso e novas brechas de segurança. Essas brechas, como já foi dito, são oriundas principalmente de má administração, mas também são influenciadas pelas características da Internet.

A Internet é uma rede internacional baseada na família de protocolos TCP/IP, interconectando dezenas de milhões de computadores espalhados por todo o mundo (e a cada ano essa quantidade aumenta, numa progressão quase exponencial). A Internet foi concebida num ambiente cooperativo, logo não houve preocupação na especificação dos protocolos com a criação de mecanismos de segurança. O sucesso da Internet fez da família de protocolos TCP/IP o padrão *de facto* para a construção de redes locais, de médio ou longo alcance. Porém esses protocolos apresentam numerosos pontos que facilitam o seu uso em ataques.

1. **Não há autenticação de origem e de destino nos datagramas IP** – neste caso, o destinatário e a origem podem apenas assumir que o datagrama foi realmente enviado pela origem especificada, ou que chegou no destino verdadeiro.
2. **O datagrama IP trafega sem proteção de privacidade para seus dados** – como o TCP/IP se baseia no modelo armazenar e repassar (*store & forward*), durante o trajeto o conteúdo do pacote pode ser facilmente copiado e inspecionado.
3. **Há poucas verificações de integridade e coerência no conteúdo dos campos dos cabeçalhos** – conseqüentemente, é possível enviar datagramas com

campos incoerentes, que provocam resultados imprevisíveis.

4. **Embora exista um padrão de implementação, ele é muito flexível, dando margem a implementações muito diferentes do protocolo**. Isso potencializa os efeitos de (3).

Podemos também dividir as ameaças de segurança de rede em dois grupos principais: os ataques às máquinas ligadas à rede, e os ataques às comunicações realizadas entre elas.

2.5.1 Ameaças às Máquinas

Um ataque a uma máquina que logre resultado pode ter como conseqüências o uso não autorizado de recursos, a divulgação de arquivos sigilosos, a alteração ou a destruição, total ou parcial, de arquivos, ou o impedimento de uso de recursos por usuários legítimos. O uso não autorizado de recursos é a primeira e mais comum das conseqüências de uma invasão. Tempo de CPU, espaço em disco e banda passante são alguns exemplos de recursos consumidos pelo invasor. Arquivos de conteúdo sigiloso, tais como programas fonte, registros de vendas ou de cartões de crédito, listas de nomes de agentes secretos, ou quaisquer outros tipos de arquivo cuja divulgação cause prejuízo ao dono do sistema atacado, são alvos constantes dos invasores. Em muitos casos, inclusive, o acesso a esses arquivos é a razão do ataque.

Há casos em que o intruso, entre outras ações, altera ou destrói arquivos. Um dos alvos preferidos dos atacantes, nesse caso, é o registro de atividades do sistema, para encobrir suas "pegadas". O intruso pode também, às vezes, impedir que usuários legítimos usem recursos do sistema, seja pela modificação de arquivos de autorização ou pela ocupação desses recursos.

2.5.2 Ameaças às Comunicações

As mensagens que trafegam nas redes de computadores, sejam elas locais, de médio ou longo alcance, estão sujeitas a vários tipos de ataque, tais como falsa identificação, escuta não autorizada, alteração ou destruição, total ou parcial, dos dados das mensagens, impedimento de uso dos computadores, ou a duplicação de mensagens válidas. Fica evidente a analogia direta com as ameaças às máquinas, conforme visto acima.

Quando uma mensagem é transmitida numa rede, ela normalmente transporta consigo uma identificação de quem a originou e a quem ela se destina. Na maioria das

vezes, no entanto, não há como identificar se: (a) a mensagem se origina realmente do lugar especificado; (b) se quem a recebe é realmente o destinatário especificado. Erros de roteamento, transmissão, ou mesmo de endereçamento podem acontecer; além disso, um usuário mal-intencionado pode subverter o sistema para enviar mensagens como se fosse outro, ou fazer o sistema entregar-lhe mensagens de outrem.

Boa parte das tecnologias usadas para a transmissão de dados são passíveis de escuta. Algumas, como a transmissão por rádio, são inerentemente abertas; a escuta pode acontecer até involuntariamente. Já nas transmissões por fio, é possível a instalação de "grampos", gerando acesso a todos os dados em tráfego pelo canal de comunicação. Nas redes locais por difusão (e.g. Ethernet, IEEE 802), a escuta das transmissões é possível de ser feita pelas próprias interfaces de comunicação[2].

A escuta pode ser **passiva** (quando o oponente apenas registra o conteúdo das comunicações) ou **ativa** (quando o oponente altera o fluxo das mensagens). De um modo geral, não há como impedir a escuta de qualquer comunicação[3]. O que se pode fazer é impedir que os dados sejam interpretados.

Uma pessoa que tenha acesso às comunicações pode, propositalmente, alterar o conteúdo de mensagens ou mesmo destruí-las. Essa alteração pode ser feita através da mudança, da inclusão ou da supressão de trechos.

Uma invasão pode causar, além dos problemas listados acima, outros prejuízos à empresa atacada. Entre vários, podemos citar:

- perda da confiança dos seus clientes;
- responsabilidade civil pelos prejuízos indiretamente causados;
- obrigação de arcar com os custos de reconstrução e reconfiguração dos sistemas afetados, bem como seus dados.

2.6 Mecanismos de Segurança

Uma das primeiras medidas que o administrador deve tomar é implantar mecanismos de segurança que protejam máquinas e comunicações, de forma a minimizar os

---

[2] Esse modo de operação é conhecido como *modo promíscuo* e é usado para manutenção.

[3] [30] descreve um mecanismo para impedir escuta, baseado em hardware: os cabos de comunicação são colocados dentro de dutos lacrados, preenchidos com argônio a alta pressão. Para acessar o cabo, o atacante precisa furar o duto, o que faz o gás vazar; sensores de pressão nas extremidades do duto dão o alarme. Num esquema mais cruel, o gás nobre pode ser substituído por outro venenoso.

riscos a que o sistema está exposto. Esses mecanismos devem ser escolhidos como base de uma política de segurança bem definida. Existe uma relação unívoca entre os riscos a que os componentes de um sistema estão expostos e as medidas de segurança a serem adotadas. Assim como os riscos são separados em riscos às máquinas e às comunicações, também os mecanismos são divididos em mecanismos de proteção às máquinas e às comunicações.

2.6.1 Mecanismos de Proteção às Máquinas

Os mecanismos de proteção às máquinas envolvidas são divididos em mecanismos de identificação, de autorização e de isolamento. Eles também podem ser locais, quando tratam cada máquina de uma rede isoladamente, ou distribuídos, quando trabalham coordenadamente em duas ou mais máquinas ligadas à rede.

Nos mecanismos de identificação, o usuário só recebe acesso ao recurso desejado após informar sua identificação. Essa identificação deve ser confirmada através do uso de um esquema de desafio–resposta (o sistema apresenta ao usuário uma pergunta tal que só o verdadeiro usuário saberia responder) ou de senha (o sistema pede que o usuário forneça uma senha de acesso).

Nos mecanismos de autorização, o usuário (já identificado) recebe ou não acesso a determinados recursos a partir de tabelas de acesso mantidas pelo administrador do sistema. Essas tabelas podem ser definidas por recurso, por sistema, por grupo de recursos, por usuário, por grupo de usuários, etc.

Nos mecanismos de isolamento objetiva-se aumentar a segurança do sistema, como um todo, pelo isolamento do sistema das origens do risco. Esse isolamento pode ser total (no caso de sistemas críticos, que devem ser totalmente isolados, lógica e fisicamente) ou parcial. Um mecanismo de segurança bastante famoso hoje em dia é o *Firewall*. Um *firewall* [30] é uma estrutura formada por 3 elementos: um filtro de saída de pacotes, um filtro de entrada de pacotes, e uma máquina chamada *gateway de aplicações*. O *Firewall* fica no meio do caminho entre duas redes isoladas, selecionando e inspecionando todo o tráfego entre os dois lados. A idéia é a mesma dos fossos dos castelos medievais. O *firewall* define um **perímetro de segurança** em torno da rede interna. As regras de filtragem garantem que somente mensagens autorizadas circulam dentro do perímetro de segurança. Para entrar ou sair é necessária uma permissão especial. A principal desvantagem do uso do *firewall* clássico é que caso ele seja invadido, toda a rede fica comprometida. Além disso, esse esquema é considerado pouco versátil

por não haver comunicações automáticas entre o lado de dentro e o mundo. Na prática, ele é substituído pelo uso de *Firewalls Transparentes* e *Proxies* [30,27].

2.6.2 Mecanismos de Proteção às Comunicações

Como foi dito anteriormente, não há maneiras práticas de impedir que uma mensagem que esteja trafegando pela rede seja escutada ou adulterada por uma pessoa não autorizada. A solução para este problema é fazer com que, caso a escuta ocorra, a informação capturada seja ininteligível, e que a adulteração da mensagem seja visível para o receptor. Existem vários mecanismos para isso, quase todos baseados em alguma forma de **criptografia**. A teoria e os conceitos de criptografia fogem ao escopo deste trabalho; mais informações podem ser encontradas em [30] e [27].

Um dos principais mecanismos usados para a proteção das comunicações é o uso de **protocolos seguros**. Um protocolo seguro é um protocolo de comunicação que garante a identificação das partes envolvidas, bem como a integridade e privacidade do fluxo de mensagens entre elas. Podemos citar como exemplos os protocolos HTTPS e SSL [7].

2.7 Mecanismos de Detecção de Ataques

Quase sempre um ataque é precedido de algum sinal, algum tipo de atividade suspeita. O administrador deve ter à sua disposição ferramentas que o alertem para tais indícios. Alguns eventos são característicos de uma tentativa de invasão, como os a seguir.

Caso 1 – um usuário tenta entrar no sistema e seguidamente erra a senha. A suspeita aumenta se as tentativas são sucessivas e muito rápidas (pode ser um computador tentando fazer um ataque de força bruta);

Caso 2 – um usuário (externo à área de suporte técnico) tenta acessar um arquivo crítico do sistema, tal como as definições das permissões de acesso;

Caso 3 – um usuário começa a errar digitação de comandos de forma anormal (p.ex. um usuário que entra em sessão num sistema Unix e digita seguidamente `DIR`, ao invés de `ls` para obter o diretório, pode ser um intruso acostumado ao MS-DOS).

Caso 4 – um computador recebe vários pedidos de conexão seguidos, em portas TCP ou UDP que não estão em uso, oriundos de um grupo restrito de máquinas (pode estar sendo vasculhado à procura de pontos vulneráveis em sua implementação – *port scan*).

Caso 5 – um computador acessa portas não definidas ou passa parâmetros estranhos para portas conhecidas (pode estar procurando por *backdoors*);

Caso 6 – um pacote chega numa rede por um caminho diferente do habitual (pode ser um caso de falsa identificação);

Caso 7 – um pacote com endereço de origem da rede local chega de fora (pode ser falsa identificação).

Existem dois enfoques para a detecção de atividades intrusas e ataques. Em ambos os casos, a ferramenta de detecção procura por padrões de atividades nos fluxos de informação. No primeiro caso, a ferramenta estabelece um padrão de normalidade e classifica como ataque ou intrusão qualquer atividade que se afaste significativamente desse padrão (detecção por anomalia). No segundo caso, a ferramenta é alimentada com padrões que identificam atividades consideradas impróprias, gerando um alarme a cada vez que os padrões procurados sejam encontrados no fluxo de informação (detecção por padrões de ataque).

As ferramentas de segurança podem ser classificadas em vários grupos, conforme seu objetivo. Existem os sistemas *analisadores de vulnerabilidades*, que identificam se os computadores de um domínio apresentam alguma vulnerabilidade conhecida (ex.: SAINT, SATAN, COPS), os *monitores de rede*, que possibilitam acompanhar e registrar os pacotes que trafegam por um segmento de rede e identificar as operações que eles representam (ex.: tcpdump [15], SNORT), os *protetores de integridade*, que evitam que arquivos do sistema operacional sejam substituídos por cavalos de tróia (ex.: TRIPWIRE), os *controladores de acesso*, que definem com precisão quem pode e quem não pode utilizar os recursos do sistema, e finalmente, os *sistemas de detecção de intrusos*, que serão vistos com mais detalhe abaixo. A arquitetura proposta neste trabalho é um exemplo de sistema de detecção de intrusos.

2.8 Sistemas de Detecção de Intrusos (IDS)

Um Sistema de Detecção de Intrusos (IDS, do inglês *Intrusion Detection System*) é um sistema destinado a impedir que um determinado ambiente computacional seja utilizado de maneira indevida ou não autorizada.

Quanto ao escopo de sua proteção, ele pode ser classificado como IDS de Máquina (ou HIDS, do inglês *Host IDS*) e IDS de Rede (NIDS, do inglês *Network IDS*).

Um HIDS destina-se a detectar tentativas de uso não autorizado ou indevido de

recursos de uma máquina específica a partir de processos ou usuários localizados dentro da própria máquina. Normalmente o HIDS trabalha com informações disponibilizadas pelo próprio Sistema Operacional (SO) da máquina defendida, tais como *logs* de sistema, ou então *system calls* privilegiadas. Em alguns casos, também podem ser utilizados *system hooks* interceptando o acesso a arquivos chave dentro do sistema de arquivos da máquina defendida. Os conhecidos programas antivírus são bons exemplos de HIDS, apesar de concentrarem sua atuação apenas nos processos em execução na máquina, mais particularmente nos acessos destes processos aos arquivos sediados nessa máquina, sem se preocupar com as atitudes dos usuários.

Já um NIDS destina-se a detectar as tentativas de uso autorizado ou indevido de recursos de uma ou mais máquinas dentro de uma sub-rede (ou domínio de rede), que sejam originadas de processos ou usuários de fora desse domínio de rede. Para isso, o NIDS se vale da análise do tráfego presente na rede, obtido através de *network taps*. Os *network taps*, para serem funcionais, devem ser instalados em um ponto da sub-rede onde eles tenham acesso a todo o tráfego presente na sub-rede.

A qualidade de um IDS é medida pela sua eficiência em identificar corretamente a ocorrência de um ataque.

Existem dois vícios que devem ser evitados na detecção: o **falso positivo** e o **falso negativo**. O **falso negativo** é a situação em que um ataque aconteceu mas o IDS não o identificou. Essa situação é ruim pois a presença de um sistema de defesa pode fazer com que medidas de segurança adicionais necessárias não sejam tomadas, aumentando o potencial de prejuízo de um ataque bem sucedido. O **falso positivo** é inconveniente pois tende a minar a confiança dos operadores no sistema de defesa. A ocorrência constante de falsos positivos pode fazer com que alertas reais passem desapercebidos.

Existem outros pontos que são positivos na avaliação de um IDS: o tempo de resposta, a carga adicional sobre o sistema, e a compreensividade das mensagens de alerta geradas. As mensagens de alerta geradas pelo sistema devem ser completas e claras.

Embora não seja obrigatório, é conveniente que um IDS funcione em tempo-real, com um tempo de resposta mínimo (medido entre o início do incidente e a sinalização do alerta), de forma a minimizar a possibilidade de geração de prejuízos.

Como o IDS é executado em uma ou mais máquinas do sistema, e também gera tráfego com configurações e mensagens internas ou de alerta, ele tende a aumentar a carga de utilização do sistema defendido. É conveniente que esse aumento seja mínimo,

para não causar uma impressão negativa sobre o usuário final.

2.9 O IDS como alvo de Ataques

Uma situação que não pode ser negligenciada no projeto de um IDS é o fato de que ele é um alvo de grande valor dentro de um sistema.

Um ataque que provoque o bloqueio do sistema de defesa de uma máquina ou rede deixaria livre o caminho para ataques subseqüentes. Outro objetivo de um ataque pode ser confundir o sistema de defesa, fazendo com que este deixe de detectar um ataque em curso, ou então acuse falsos ataques, ou mesmo identifique erroneamente o tipo e/ou a origem de um ataque em curso, aumentando a taxa de falsos positivos.

Um erro muito comum no projeto de NIDS é não implementar identicamente o funcionamento da pilha IP das máquinas defendidas, fazendo que o NIDS interprete datagramas recebidos de maneira diferente da máquina alvo [22]. Com isso, o NIDS fica vulnerável aos dois tipos de ataques a seguir.

- **Inserção**: o atacante gera padrões completos ou incompletos de falsos ataques, de forma a fazer com que o NIDS gere falsos positivos. Com isso, ele pode desviar a atenção do controlador do sistema de outra atividade, ou minar a confiança deste no NIDS.
- **Evasão**: o atacante gera padrões de tráfego que, apesar de fazer parte de um ataque, não são reconhecidos pelo NIDS como tal.

A solução para este tipo de problema se torna mais complexa quando se tem uma rede heterogênea, formada por máquinas de arquiteturas diferentes, sistemas operacionais diferentes, ou mesmo versões diferentes do mesmo SO.

2.10 Considerações Finais do Capítulo

Neste capítulo foram descritos conceitos básicos de segurança. Fica claro que a ligação de um sistema de computadores a uma rede local expõe os recursos desse sistema a vários riscos, tanto como de uso indevido como de impedimento de uso desses recursos por usuários legítimos. Da mesma forma, informações trafegando em um canal de comunicação estão sujeitas a serem captadas por entidades não autorizadas, que podem usar essas informações contra sua origem, seu destino, ou ambos, seja utilizando essas informações para obter vantagem, seja alterando, inserindo ou removendo partes delas.

O risco envolvido na operação de sistemas de computadores é uma componente importante do Custo Total de Propriedade, o que justifica o uso de ferramentas de maior complexidade, tais como sistemas especialistas e Sistemas de Detecção de Intrusos, para reduzir o risco associado.

A necessidade de ferramentas automatizadas para a detecção de tentativas de abuso, ataque ou invasão é visível, pois a vigilância humana nesses casos, além de muito pouco eficaz, é de custo muito elevado.

Este trabalho descreve uma ferramenta voltada principalmente para a detecção e reação de tentativas de abuso, invasão e ataque.

Capítulo 3 - Conceitos Básicos

Na análise e codificação de sistemas para a resolução de problemas, é comum encontrar-se situações onde apesar dos passos para a solução serem descritos em seqüência, nem todos passos são temporalmente dependentes dos passos anteriores. Essa situação é conhecida como paralelismo implícito e quando descoberta e aproveitada, com a execução simultânea de tarefas, possibilita um ganho no tempo de processamento proporcional ao número de tarefas executadas simultaneamente. Para a execução de cada uma dessas tarefas é necessária uma unidade de processamento, que pode ser uma CPU, um micro-controlador, ou até um computador completo. Programas que são preparados para execução de forma a utilizar várias unidades de processamento ao mesmo tempo são chamados programas distribuídos.

Quando um trecho de programa atua como se fosse um operador ou usuário, ele é chamado de agente (esta definição será explicada com mais detalhe à frente). O conceito de agente é conveniente para a criação de sistemas distribuídos, devido a características tais como autonomia, modularidade e sociabilidade. A topologia de distribuição dos agentes componentes do sistema é um fator determinante para a qualidade do sistema resultante. Agentes podem ser fixos, mantendo a posição determinada na topologia durante todo o processamento, ou móveis, podendo assumir novas posições conforme a vontade ou a necessidade do sistema.

As vantagens do uso de agentes para o desenvolvimento de sistemas distribuídos fizeram com que vários ambientes de desenvolvimento de agentes surgissem. Esses ambientes, em sua maioria, são baseados em linguagens que usam o paradigma de orientação a objetos. Além disso, nos casos dos ambientes de agentes móveis, é interessante que a linguagem adotada possibilite o uso de ambientes de execução heterogêneos. Essas características, mais a necessidade de um desempenho razoável, difícil de ser alcançado com linguagens de *script*, pesaram na escolha da linguagem Java para o desenvolvimento do projeto deste trabalho.

Este capítulo inicia descrevendo os conceitos de agente, sistemas multi-agente, e mobilidade. O texto prossegue apresentando o Paradigma de Orientação a Objetos. Por fim, são apresentados alguns trabalhos que estão de alguma forma relacionados com a arquitetura apresentada neste trabalho. A proposta de utilizar sistemas multi-agentes para controle e administração de atividades intrusivas foi objeto de outros trabalhos, po-

rém sem o uso de mobilidade. Outros trabalhos procuram alcançar escalabilidade em larga escala usando a mesma técnica.

3.1 Agentes

Apesar de ainda não existir consenso quanto às definições de agentes, é possível sintetizar o termo agente como sendo uma entidade de software que realiza um conjunto de operações em nome de uma outra entidade, com um grau arbitrário de independência e autonomia.

Todo agente possui características que possibilitam a sua classificação. Estas podem ser: autonomia; busca de metas; flexibilidade; iniciativa; comunicatividade; sociabilidade; inteligência; mobilidade; e continuidade temporal. Cada uma dessas características aparece num determinado grau em todos os agentes, e tem um peso específico na avaliação da qualidade do agente para o cumprimento de uma determinada tarefa.

Duas características têm significado especial neste trabalho: a *autonomia* e a *mobilidade*. A autonomia é a capacidade do agente em tomar decisões e executar ações de maneira independente de controle de operadores externos. Já a mobilidade é a capacidade de um agente de transportar seu estado de execução atual de um ambiente de execução para outro.

O motivo do interesse particular nessas características é a influência delas na qualidade do resultado da aplicação de um sistema multi-agentes como um Sistema de Detecção de Intrusos Distribuído operando em tempo real.

Um sistema multi-agentes é um sistema constituído por um conjunto não unitário de agentes, que colaboram para o cumprimento das tarefas para que foram designados. Esses agentes podem ou não pertencer a um mesmo usuário. As funções desempenhadas por esses agentes também podem ser diferentes. O grau de colaboração de um agente com outros é definido como a *sociabilidade* desse agente.

Um sistema distribuído caracteriza-se por ser composto por módulos que são executados em diferentes unidades de processamento. Note-se que essa não é uma condição necessária para um sistema multi-agentes, embora seja muito comum. A grande vantagem do sistema distribuído é o aproveitamento do paralelismo entre as unidades de processamento para aumento no desempenho. Os sistemas multi-agentes podem também ser classificados quanto às unidades de processamento, que podem ser

homogêneas ou *heterogêneas*. O ambiente reunido das unidades de processamento pode ser de *granularidade grossa* ou *fina*. As unidades de processamento podem também compartilhar ou não recursos como memória, disco ou meios de comunicação (caso dos ambientes de rede local).

Uma das maiores vantagens do uso de sistemas distribuídos multi-agentes na solução de um problema em particular é a fácil extração do paralelismo desse problema, associada com a sociabilidade dos agentes. Além disso, o desenvolvimento se dá de forma modular, facilitando o processo de codificação.

Outras vantagens do uso desta abordagem são a maior tolerância a falhas e a maior facilidade de reconfiguração. A tolerância a falhas vem do fato de que no caso da falha de um dos agentes, os outros podem continuar funcionando independentemente, sendo apenas prejudicados nos pontos que dependem de intervenção do agente em falha. A facilidade de reconfiguração vem da independência e do isolamento entre os agentes.

Uma característica inerente aos agentes componentes de um sistema multi-agentes é que eles definem, entre si, uma *interface* de dados e serviços. Essa interface define a estrutura dos dados trocados e os serviços que podem ser requisitados uns aos outros. A geração de novas versões dos agentes é simples, desde que a nova versão mantenha a mesma interface.

3.2 Agentes Móveis

Um agente que tem a capacidade de transportar o seu estado de execução de uma unidade de processamento para a outra é chamado *agente móvel*. Este tipo de agente traz vantagens tais como o aproveitamento remoto de recursos que estariam disponíveis apenas localmente e a diminuição do uso da banda de transmissão. Suas desvantagens são, entre outras, uma latência maior de ativação, devido à necessidade de transferência de código e estado, e uma sensibilidade de segurança, devido à possibilidade de uso não autorizado de recursos. Um estudo mais detalhado sobre agentes móveis pode ser encontrado em [6].

Uma característica do sistema baseado em agentes móveis é que neste tipo de sistema o intercâmbio de informações é feito principalmente a nível local, o que pode ser ajustado para economizar recursos de banda de transmissão. O agente móvel, então, procura acumular informação e processá-la localmente, para retornar com a informação

já processada.

Um requisito comum para os sistemas de agentes móveis é a possibilidade de execução em ambientes heterogêneos, ou seja, onde cada um dos *hosts* do domínio apresente um ambiente operacional – arquitetura de hardware e sistema operacional – diferente. Esse requisito faz dos sistemas móveis candidatos ao uso de linguagens interpretadas, tais como Tcl e Java.

Vários elementos fazem parte do sistema com agentes móveis. Entre eles, temos o código do agente, o seu estado de execução, o contexto de execução, o mecanismo de deslocamento, e as permissões de acesso. O *código* é o programa que define o comportamento do agente. O *estado* define o valor dos dados internos do programa do agente, bem como o ponto de execução do programa do agente. O *contexto de execução* de um agente é a unidade de processamento que serve de sede para a execução do agente. Dentro de cada unidade de processamento é necessário que haja um serviço de mobilidade, possibilitando o *deslocamento* do agente de um contexto para o outro. O mecanismo de deslocamento está intrinsecamente associado ao serviço de mobilidade.

Como agentes e contextos podem pertencer a *domínios administrativos* diferentes, é necessário definir quais são as *permissões de acesso* associadas a um agente dentro de um contexto. Essas permissões servem para disciplinar o acesso dos agentes aos recursos disponíveis no contexto.

Devido à possibilidade de agentes e contextos poderem pertencer a domínios administrativos diferentes, há um risco de segurança associado ao uso de agentes móveis. Esse risco existe em dois níveis: do agente para o contexto e do contexto para o agente.

Do lado do contexto, é preciso assegurar que o agente tenha as permissões para acessar os recursos solicitados. Essa verificação é feita levando-se em conta o domínio administrativo a que pertence o agente e sua identidade. Com isso, é necessário que se usem mecanismos que autentiquem os agentes em termos de identidade e origem.

Já do lado do agente, é necessário que ele possa, de alguma forma, confiar no contexto, pois o agente pode carregar dados sensíveis que podem ser acessados indevidamente pelo contexto ou por outros agentes dentro do mesmo contexto. O contexto pode também negar indevidamente um acesso, ou mesmo alterar dados de forma a prejudicar o funcionamento do agente. Esta questão é mais sensível, por não haver um consenso quanto aos mecanismos de proteção ao agente. Como já foi dito, o agente móvel, ao invés de enviar informações ao seu controlador, processa e compila essas

informações localmente para depois retornar à sua origem. É possível que o contexto, ou outros agentes, tenham acesso a essas informações, causando prejuízos ao proprietário do agente.

Esta última questão pode ser minimizada com o uso de uma interface obrigatória, bem definida, de acesso aos dados mantidos pelo agente. Essa interface deve prover controle de acesso, garantindo que apenas entidades autorizadas tenham acesso aos dados mantidos pelo agente. Esse controle é de certa maneira semelhante ao conceito de *encapsulamento*, defendido na abordagem de orientação a objetos, explicando o motivo pelo qual várias das implementações de sistemas de agentes móveis são baseadas em linguagens orientadas a objeto.

O apêndice C lista alguns exemplos de sistemas de agentes móveis disponíveis na internet.

3.3 O Paradigma de Orientação a Objetos

A abordagem de programação Orientada a Objetos (OO) consiste em visualizar o sistema como sendo uma coleção de objetos independentes, onde cada objeto define a estrutura interna e o comportamento dos seus dados. A abordagem estruturada convencional, ao contrário, visualiza o sistema como uma coleção de funções e procedimentos, que manipulam os dados e definem a inter-relação entre eles. Como o comportamento dos dados, por diversos motivos, tem uma tendência forte de volatilidade, fica difícil modelar convenientemente os dados do caso real em sistemas desenvolvidos na abordagem tradicional. Na abordagem orientada a objetos, ao contrário, a estrutura e o inter-relacionamento entre os dados ficam representados nos objetos, e a partir disso é modelado o comportamento. Com isso, mesmo com uma mudança no comportamento, a estrutura do objeto é mantida. O objeto define também o *método* de acesso aos seus dados, disciplinando o acesso conforme a vontade de seu criador.

Ao enfocar na estrutura e relacionamento entre os dados, a abordagem OO cria conceitos como identidade, classificação, hierarquia, polimorfismo e herança. A *identidade* é a propriedade dos dados em se dividirem em entidades discretas e distintas, que recebem o nome de objetos. A *classificação* é a propriedade dos objetos em compartilharem a mesma estrutura e comportamento, possibilitando o seu agrupamento em *classes*. Assim, cada objeto é dito ser uma *instância* de sua classe.

A *hierarquia* é a característica que permite o agrupamento de classes de objetos

em classes mais genéricas, chamadas *superclasses*, ou a divisão das instâncias de uma classe em classes mais especializadas, chamadas *subclasses*. Um objeto, ao ser instância de uma subclasse, também o é das superclasses dessa subclasse. O comportamento das instâncias de uma superclasse normalmente possui procedimentos comuns, embora diferentes no detalhe. Essa característica (instâncias de uma superclasse apresentarem funcionamento diferente para o mesmo procedimento) é chamada de *polimorfismo funcional*. Uma subclasse, ao especializar a superclasse, pode incluir, modificar ou excluir componentes. Os componentes aproveitados fazem parte da chamada *herança* da superclasse. É esta característica que dá ao modelo OO a reutilização de código.

Como resultado da aplicação dos conceitos de OO à modelagem das entidades envolvidas no sistema, temos que algumas características se destacam. A primeira delas, a *abstração* de dados, consiste na visualização da entidade em seus aspectos fundamentais, ignorando as propriedades não relevantes para a solução do problema. Outra é o *encapsulamento*, que é a capacidade do objeto de manter detalhes da estrutura de seus dados e do funcionamento de seus métodos ocultos do mundo exterior, aumentando a independência entre os diversos módulos e minimizando a propagação de alterações e erros.

A linguagem Java, que foi utilizada no desenvolvimento do protótipo do sistema objeto deste trabalho, é fortemente baseada no paradigma OO. Essa linguagem é descrita com mais detalhes no apêndice A.

3.4 Trabalhos Relacionados

Um dos primeiros trabalhos a propor o uso de agentes autônomos para o desenvolvimento de Sistemas de Detecção de Intrusos foi [3]. Nessa proposta, os agentes seriam fixos, desenvolvidos sob medida para a tarefa para que foram designados. O desenvolvimento usaria técnicas de *inteligência artificial* por *programação genética*. A programação genética diferencia-se dos *algoritmos genéticos* por ao invés de trabalhar em cima de uma população de soluções possíveis de um problema, trabalha em uma população de **códigos de programas** para a solução do problema, escolhendo aqueles que apresentem as melhores soluções segundo os critérios escolhidos.

Como prosseguimento da proposta acima, surgiu o projeto AAFID [33]. A arquitetura proposta neste projeto organiza os agentes (fixos) numa estrutura hierárquica, e atribui a estes incumbências diversas. Um sistema construído segundo a arquitetura

AAFID pode ser distribuído em qualquer quantidade pelas máquinas de uma rede; cada uma dessas máquinas pode conter uma quantidade arbitrária de *agentes*, monitorando eventos de interesse nelas. Todos os agentes localizados dentro de uma máquina reportam seus resultados a um único *transceptor*, que é responsável pela operação dos agentes dentro daquela máquina, tendo a possibilidade de disparar agentes, interrompê-los, ou enviar-lhes comandos de configuração. Eles podem também filtrar os dados enviados pelos agentes. Finalmente, os transceptores reportam-se a um ou mais *monitores*. Cada monitor controla a operação de um ou mais transceptores, tendo acesso a dados acerca da rede como um todo. Estes, portanto, são capazes de extrair correlações de alto nível e detectar invasões que atinjam várias máquinas. Eles podem, também, ser organizados hierarquicamente, de forma que um monitor reporte-se a outro monitor, ou trabalhar em paralelo, com redundância (para maior confiabilidade). Por fim, uma *Interface com o Usuário* é definida, possibilitando aos operadores acompanhar o funcionamento do Sistema e controlá-lo.

Outro projeto de IDS que é voltado para a busca de escalabilidade é o GrIDS [29]. Neste projeto, o ambiente a ser defendido de atividades intrusas é modelado como um grafo, e as operações de detecção e distribuição de alertas são vetorizadas. Assim, os autores do GrIDS esperam poder lidar com populações da ordem de milhares de máquinas.

Existem várias outras pesquisas em curso que usam um enfoque parecido, que seja o de dividir a tarefa de proteger a rede ou as máquinas-alvo de invasão em pequenas tarefas simples, tais como vigiar conexões, vasculhar arquivos de *log*, etc; uma lista mais completa dessas pesquisas pode ser encontrada na base de documentos do Projeto CERIAS[4].

Este trabalho é um desenvolvimento da proposta apresentada pelo autor nas publicações [23], [24] e [25].

3.5 Considerações Finais do Capítulo

Apresentamos neste capítulo conceitos básicos sobre agentes móveis e a linguagem Java. A partir destes conceitos, é possível perceber que um sistema distribuído multi-agentes apresenta várias vantagens sobre sistemas monolíticos, tais

[4] https://www.cerias.purdue.edu/papers/

como modularidade, aproveitamento de paralelismo, facilidade de reconfiguração e tolerância a falhas.

Conforme foi visto, percebe-se o ganho adicional presente nos sistemas baseados em agentes móveis, graças à redução da comunicação entre os componentes do sistema. Apresentamos alguns exemplos de ambientes de desenvolvimento de sistemas de agentes móveis, com destaque para o ambiente ASDK, que foi escolhido como infra-estrutura de mobilidade deste trabalho.

Vimos também que a linguagem Java apresenta grandes vantagens em seu uso em geral, e especialmente no desenvolvimento de sistemas multi-agentes. Seguindo estritamente o paradigma OO, ela permite com grande agilidade o desenvolvimento de sistemas, devido a cuidados para evitar construções que sejam origem de erros de programação, e a uma alta taxa de reaproveitamento de código. Sendo *multi-threaded*, apresenta várias facilidades para extração de paralelismo local, mas também exige cuidados adicionais, especialmente no trato com métodos nativos.

Capítulo 4 - Arquitetura Proposta

Neste capítulo é proposta uma nova arquitetura para um sistema distribuído, multi-agentes, para a detecção e reação a ataques e tentativas de invasão. Esta proposta é baseada na proposta original de [3] e será chamada doravante de Sistema Micæl. Os elementos da arquitetura e a topologia usados foram escolhidos de forma a aproveitar ao máximo a principal característica do sistema – a mobilidade dos agentes.

A nomenclatura utilizada é baseada em expressões militares, tais como *quartel general* e *sentinela*. O objetivo de tal escolha de nomes é enfatizar a posição e a utilidade de cada um dos agentes dentro da arquitetura.

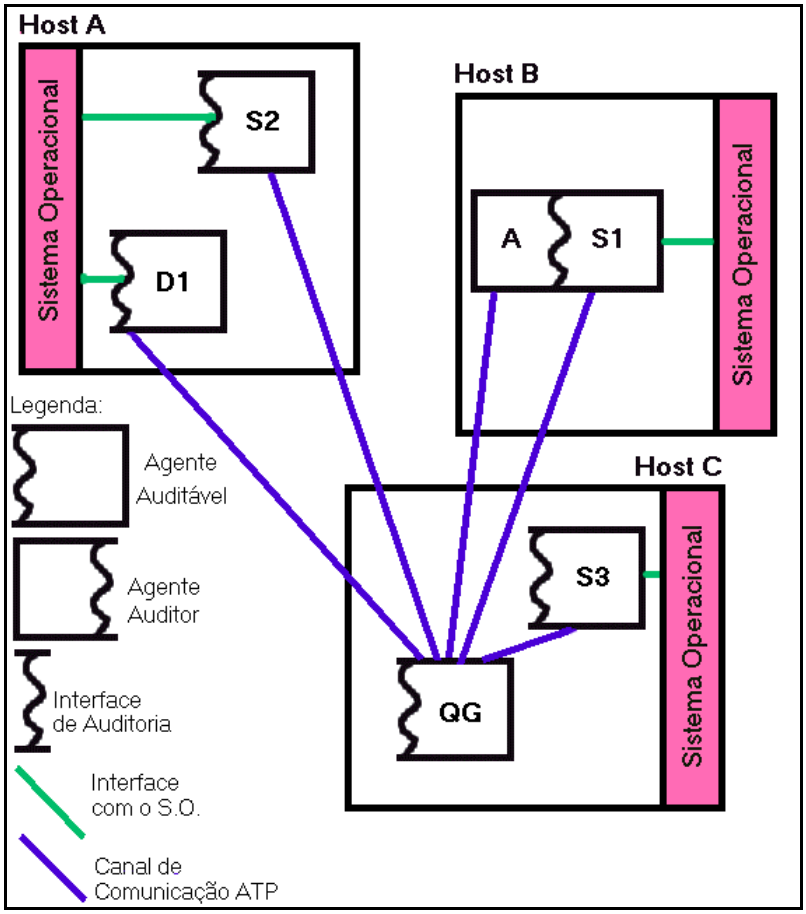

Figura 4.1– Exemplo do Sistema Micæl para uma rede com três *Hosts*.

A arquitetura foi projetada com o objetivo de manter a cada momento o mínimo de recursos computacionais em uso em um determinado ponto da rede, sem prejudicar as tarefas de detecção e reação a tentativas de ataque e/ou invasão.

O sistema Micæl foi concebido tendo em mente um domínio de rede composto por um ou mais segmentos de rede local, ao qual são interligados vários computadores

de arquitetura e funções heterogêneas. Espera-se que, desses computadores, alguns funcionem como servidores, e os outros como estações de trabalho (*workstations*). Como o sistema é concebido para manter a alocação de recursos num nível mínimo, tanto servidores como *workstations* podem participar do Sistema Micæl, aumentando a segurança do sistema. As únicas exigências feitas quanto a esse segmento de rede são que ele aceite tráfego IPv4 Unicast e Multicast. Fatores tais como banda passante, atraso de transmissão e QoS (qualidade de serviço) devem ser irrelevantes para o funcionamento do sistema.

A possibilidade de um sistema de segurança se tornar alvo de ataques é muito alta. Por isso, é necessário prover mecanismos para garantir que os agentes componentes do sistema não sejam subvertidos por um atacante, ou então evitar que um agente mal-comportado ponha a perder todo o sistema. Também é necessário fazer com que o sistema seja capaz de se recuperar após a ocorrência de falhas. Estas considerações levaram à criação do agente auditor e da interface de auditoria. O agente **Auditor** tem a função de verificar a integridade interna dos agentes em execução, e recuperar partes do sistema que estejam fora de operação. Para a verificação de integridade, o Auditor se vale da **interface de auditoria**, que é uma parte obrigatória de todos os agentes componentes do sistema Micæl. Usando os métodos definidos nessa interface, o Auditor pode acessar variáveis internas, calcular *check-sums*, ou tomar atitudes mais drásticas, como encerrar o agente auditado.

A estrutura de armazenamento dos dados coletados deve ser planejada para possibilitar a geração de vários relatórios, em suporte a uma das tarefas mais difíceis da gerência de segurança – a identificação e punição dos culpados. Além disso, o código dos agentes deve estar disponível para carga em um tempo curto. Essas são duas das funções designadas ao agente **Quartel General**.

A tarefa de detecção das tentativas de ataque e invasão são desempenhadas pelos agentes de detecção **sentinela** e **destacamento**. O destacamento, além de realizar detecção, também reage aos ataques. Os sentinelas realizam uma detecção genérica, simplificada, garantindo apenas que a incidência de *falsos negativos* seja mínima. Ao ser detectada uma atividade suspeita é convocado um agente destacamento, que realiza uma detecção especializada, preparada para minimizar a ocorrência de *falsos positivos*. Como o agente destacamento é especializado por ataques, a convocação deve anexar informação suficiente para identificar o ataque e a versão de destacamento apropriada.

4.1 Componentes do Sistema

Esta arquitetura divide a tarefa de detecção de intrusos e ameaças entre vários agentes, que são denominados conforme sua especialização: o *quartel general* (*QG*), os *sentinelas*, os *destacamentos*, os *auditores*, e os *agentes especiais*.

Podemos classificar os agentes também conforme o tipo de tarefas desempenhadas por eles, como abaixo.

- **agentes de detecção** – são agentes que trabalham diretamente com a detecção das atividades intrusas. Ex.: *destacamento*, *sentinela*.
- **agentes de reação** – são agentes que produzem algum tipo de reação à tentativa de ataque ou invasão, interna ou externamente. Ex.: *destacamento*.
- **agentes administrativos** – são agentes que realizam tarefas administrativas ou auxiliares ao trabalho de proteção. Ex.: *QG*, *auditor*.

4.1.1 Quartel General

O *QG* é o agente que centraliza as funções de controle do sistema. Ele é responsável pela criação dos outros agentes, mantendo por isso uma base de dados com os códigos executáveis dos agentes. O *QG* não realiza nenhuma tarefa de detecção. Alem disso, os agentes de detecção precisam estar em contato constante com ele. Essas características fazem com que a movimentação do QG de uma localidade para outra se torne muito cara, devendo ser minimizada. Foram identificadas duas situações em que é necessário que o QG se mova, listadas a seguir.

- Quando a carga de utilização do hospedeiro do *QG* aumenta (por exemplo, quando um usuário se "loga" nele) a um ponto em que a execução das tarefas de controle atrapalhe as tarefas do usuário, o *QG* precisa migrar para uma máquina com carga menor;
- Se o hospedeiro do *QG* é invadido ou infectado, o *QG* precisa migrar para evitar que seu código seja subvertido, ou o seu funcionamento impedido.

Nas situações em que o QG decide por migrar para um novo hospedeiro, os agentes ativos devem ser avisados, para reconstruírem suas referências de comunicação. Também é preciso considerar, no momento da migração, que o QG mantém uma série de bases de dados, algumas delas em disco; essas bases devem continuar acessíveis, independentemente do hospedeiro escolhido. Convém considerar, para tanto, o uso de

listas de hospedeiros possíveis e usar mecanismos semelhantes a URLs para identificar os arquivos das bases de dados.

Como o *QG* reúne informações recolhidas pelos agentes de detecção, ele é o ponto apropriado para a emissão de relatórios e compilação de estatísticas. Entretanto, o sistema não prevê nenhuma interface com o usuário, já que esta interface é de grande complexidade e não interfere em nada nas tarefas de detecção. Isto não quer dizer, absolutamente, que as implementações deste sistema não conterão uma interface com o usuário; apenas que ela não é de interesse para o projeto, e que, por isso, seu formato é livre. Pode-se pensar, inclusive, no uso de um formato padronizado de intercâmbio de informações com outros IDSs, tal como o proposto em [4].

Como já foi dito, o *QG* não participa das tarefas de detecção; essas tarefas são desempenhadas exclusivamente pelos *sentinelas* e *destacamentos*, agentes de detecção. A pedido destes agentes, o *QG* pode disparar novos agentes ou enviar alertas para o operador. Periodicamente, o *QG* envia agentes *auditores* para verificar a integridade do sistema, como veremos à frente.

Para cada ataque tratado, deve ser desenvolvida uma versão específica de destacamento. Esta versão deve ser associada aos padrões detectáveis que identifiquem o ataque. Essa associação gera uma base de dados, que é consultada pelo QG a cada solicitação de envio de destacamento. Essa associação pode ser criada manualmente, ou então inferida com o auxílio de ferramentas de inteligência artificial. Porém, o uso de inteligência artificial nos agentes, por diversos motivos, não será explorado neste trabalho.

Pode-se pensar também no QG como o ponto central de controle do sistema para a rede corporativa, formando um domínio de controle. Visto por esse lado, é fácil extrapolar o conceito do QG e imaginar malhas compostas por domínios adjacentes, por exemplo uma para cada filial da empresa (Figura 4.2). Esta facilidade, porém, não será explorada neste trabalho.

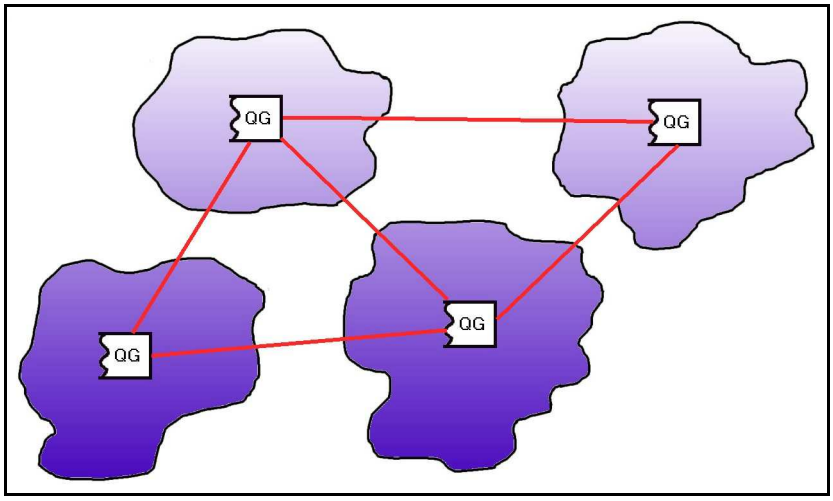

Figura 4.2- Domínios de Controle definidos pelo Quartel General

4.1.2 Sentinelas

Os *sentinelas* são agentes que ficam residentes em cada um dos hospedeiros do sistema, coletando informações relevantes e informando ao *QG* sobre eventuais anomalias para registro.

Quando um *sentinela* detecta um certo nível de anomalia, ele ajusta o **nível de alerta** associado ao hospedeiro. Esse nível de alerta pode variar desde um nível basal (sem alerta) até o máximo (alerta total, ou alerta vermelho). A lista a seguir define os níveis intermediários de alerta, num total de oito, num nível crescente de risco associado, e decrescente de visibilidade para o Sistema Operacional hospedeiro.

- **Sem alerta**: nenhuma atividade suspeita foi detectada num período significativo.
- **Alerta violeta**: uma atividade suspeita foi detectada, porém essa atividade está em um nível tal que pode ser controlada trivialmente pelo sistema operacional hospedeiro. Ex.: tentativa de abertura de sessão sem senha.
- **Alerta Azul**: uma atividade suspeita foi detectada, tal que o sistema operacional hospedeiro só pode tratá-la com o auxílio de utilitários específicos, de forma que a atividade é mapeada num risco maior. Ex.: acesso a um serviço interno a partir de uma máquina localizada na rede externa.
- **Alerta Ciano**: a atividade suspeita que ocasiona a entrada neste nível de alerta é detectável pelo SO hospedeiro, mas este não a trata. A partir

deste nível, destacamentos devem ser convocados. Ex.: aquisição de direitos de supervisor por usuário interno.

- **Alerta Verde**: atividades suspeitas a partir deste nível não são detectadas pelo sistema operacional. Ex.: Tentativa de conexão às portas usadas pelo BackOriffice, mesmo que este não tenha infectado a máquina alvo.
- **Alerta Amarelo**: atividades deste nível são tais que o atacante pode conseguir executar comandos na máquina alvo, ***sem*** privilégios de supervisão. Ex.: ataque PHF, onde o servidor HTTP não executa em modo supervisor.
- **Alerta Laranja**: atividades deste nível são tais que o atacante pode conseguir executar comandos na máquina alvo, ***com*** privilégios de supervisão. Ex.: ataque Bind ou LPR.
- **Alerta Vermelho**: atividade suspeita com risco de bloqueio de serviços. Ex.: SynFlood.

Para cada transição de nível de alerta, o sentinela adota uma atitude. Ao passar de um nível inferior para qualquer nível igual ou superior ao alerta verde, o sentinela solicita ao *QG* a criação de um *destacamento* para verificar com mais detalhe a anomalia detectada. Dentre os vários códigos executáveis disponíveis, o mais apropriado para lidar com a anomalia verificada é escolhido e disparado. Se após a entrada num determinado nível de alerta nenhuma atividade seja detectada que justifique a manutenção desse nível de alerta dentro de um tempo predeterminado, ou caso a contramedida aplicada seja eficaz, o alerta passa ao nível imediatamente inferior em que haja atividades suspeitas detectadas. Pode-se dizer, inclusive, que diversos níveis de alerta podem estar ativos simultaneamente, e todos eles devem ser tratados; os níveis mais altos, porém, devem receber maior prioridade no tratamento. Uma vez que a anomalia seja controlada, e nenhuma outra seja detectada durante um intervalo de tempo predeterminado, o sentinela desativa o nível de alerta correspondente.

O ciclo de vida do sentinela pode ser descrito como a seguir.

1. O *QG* cria em seu hospedeiro um agente sentinela;
2. O sentinela recebe ordem de migrar para o seu hospedeiro destino;
3. O sentinela ativa-se no hospedeiro de destino.
4. O sentinela consulta as bases de informações disponíveis em busca de anomalias ou padrões de invasão.

5. Caso uma anomalia seja encontrada, o sentinela ajusta o nível de alerta associado ao hospedeiro. Níveis mais altos de alerta tem prioridade sobre níveis inferiores.
6. Caso o nível de alerta transicione para qualquer nível igual ou superior ao alerta verde, o sentinela solicita ao QG o envio de um destacamento, que faz uma detecção mais sofisticada e toma as providências apropriadas.
7. Caso o nível de alerta transacione para o nível de alerta vermelho (risco de bloqueio), o sentinela deve adotar atitudes que garantam que os serviços de comunicação mantenham-se operantes até a ativação do destacamento convocado.
8. Se a partir da detecção de uma anomalia que ocasione uma transição de nível de alerta nenhuma outra anomalia do mesmo nível for detectada dentro de um período de tempo, o nível de alerta desce para o nível inferior em que haja atividade suspeitas.
9. Periodicamente o sentinela contacta o QG, e registra lá as informações recolhidas, em prevenção contra quedas repentinas da máquina hospedeira.
10. O processamento continua até que a máquina hospedeira seja desligada (em processo normal, ou seja, por *shutdown*) ou o sistema receba ordem de se desligar; nesses casos, a sentinela migra de volta para o hospedeiro do QG, onde grava um resumo das informações recolhidas e é terminado.

Embora não seja uma característica necessária, é conveniente que o sentinela seja dotado de um grau razoável de inteligência, manifestada na capacidade de aprendizado. Com essa capacidade, o sentinela pode ser codificado para detectar atividades com uma sensibilidade alta, recebendo re-alimentação do operador para distinguir os falsos positivos dos verdadeiros.

Existem situações em que pode ser necessária uma reação rápida, antes mesmo de ser possível convocar um destacamento. É o que acontece quando um ataque de bloqueio é desferido contra as comunicações da máquina hospedeira de um sentinela. Nesse caso, o próprio sentinela deve prover para que o bloqueio não aconteça, garantindo que o destacamento consiga chegar e ser ativado.

Apesar do requisito multi-plataforma, é aceitável que o sentinela seja ligeiramente especializado para o ambiente operacional em que vai operar. Isso acontece pois as ameaças que estão presentes num ambiente operacional podem não estar em outro. Assim evita-se que o procedimento de detecção se torne excessivamente generalizado, em prejuízo da clareza e concisão do código gerado.

4.1.3 Destacamentos

Os *destacamentos* são agentes especializados que são criados a pedido dos sentinelas para fazer frente a uma possível ameaça surgida. Quando um sentinela identifica uma anomalia, ou um padrão semelhante a uma invasão, ele solicita ao QG a criação e o envio de um destacamento ao ponto da anomalia. Os destacamentos possuem mecanismo de detecção mais elaborado e mais especializado para determinadas situações, podendo tomar medidas de defesa e contra-ataque contra a ameaça, caso ela seja confirmada.

O ciclo de vida dos destacamentos pode ser descrito como a seguir.

1. O sentinela de um hospedeiro identifica uma anomalia e recolhe as informações relevantes. Essas informações serão usadas para escolher qual o código mais apropriado para o destacamento.
2. O sentinela repassa essas informações ao QG, solicitando a criação e o envio de um destacamento.
3. O QG cria um novo agente em seu hospedeiro, com o código escolhido.
4. O novo destacamento recebe ordem de migrar para o hospedeiro sob ameaça.
5. Ao ativar-se no hospedeiro sob ameaça, o destacamento inicia uma avaliação sobre a real situação, podendo decidir por confirmar ou não a ameaça. Esta decisão é repassada ao sentinela local e ao QG, para registro.
6. Confirmada a ameaça, o destacamento inicia as contramedidas. Estas podem incluir desinfecção de programas, encerramento forçado de sessões de usuário, desligamento da máquina (*shutdown*) ou até mesmo contra-ataques para bloqueio da máquina agressora (no caso de agressões oriundas de fora).
7. Quando a ameaça é controlada, e o nível de alerta retorna ao normal, o destacamento migra de volta ao hospedeiro do QG, grava seu estado para referência futura, e se desliga.

Pode acontecer do destacamento decidir que não é apropriado para lidar com uma ameaça em potencial, e solicitar a criação de outro destacamento. Vários destacamentos podem estar em ação simultaneamente num mesmo hospedeiro. Essa sobreposição pode fazer com que a carga de utilização do hospedeiro suba até um ponto que o

funcionamento do sistema de defesa interfira na execução das suas tarefas normais. Por isso, um nível máximo de carga deve ser estabelecido, de forma que o(s) usuário(s) das máquinas sob ameaça recebam o serviço mínimo definido pelo administrador do sistema. Ao ultrapassar esse nível, o operador desse sistema deve ser consultado para escolher entre desligar a máquina, interromper o sistema, ou prosseguir a reação com uso de procedimentos remotos (uma possibilidade que não será desenvolvida neste trabalho).

A produção de novos destacamentos é feita com base na análise dos dados coletados pelos sentinelas e destacamentos durante o seu funcionamento. O procedimento de análise pode ser manual, ou automatizado com ajuda de ferramentas especiais. Essas ferramentas podem utilizar técnicas de inteligência artificial e/ou KDD (*Knowledge Discovering in Databases*), mas não serão objeto deste trabalho.

Os destacamentos exibem uma especialização para os ambientes operacionais usados ainda mais marcante do que a dos sentinelas. Isto acontece pois os destacamentos são desenvolvidos especificamente para os tipos de ataque, e os ataque são específicos para cada ambiente operacional.

Há outras situações em que o destacamento pode ser convocado sem que haja nenhum ataque sendo desferido no momento. É o caso quando se identifica um sistema em particular mais vulnerável a ataques, tal como um servidor de páginas ou de correio eletrônico. Assim, ao identificar a presença de um sistema desse tipo, o sentinela pode convocar o destacamento especializado apropriado (na verdade, um sentinela especializado). Deste momento em diante, o sentinela do hospedeiro não mais precisa preocupar-se com o sistema crítico. A mesma solução pode ser adotada para especializar os sentinelas para cada um dos ambientes operacionais.

4.1.4 Auditores

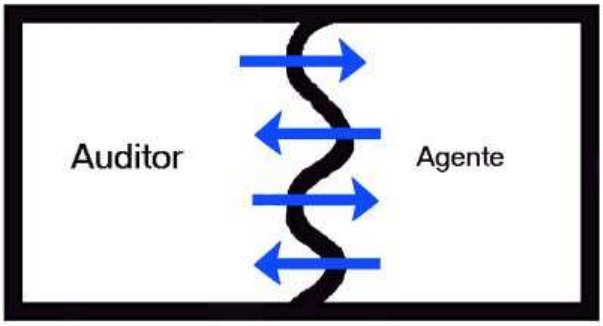

Figura 4.3 - O Auditor em Funcionamento

O *auditor* é o agente que tem como função evitar que um agente cujo código tenha sido subvertido prejudique a segurança do sistema. O auditor é criado pelo QG no início da operação, e migra entre os hospedeiros conhecidos pelo sistema, fazendo a verificação do correto funcionamento de cada agente.

O auditor é o único agente, além do QG, que tem a permissão de criar novos agentes. Essa permissão é usada para recriar o QG, caso este aborte suas funções por qualquer motivo. Caso o auditor perceba, em qualquer hospedeiro, que o sentinela não está ativo, ele solicita ao QG que o sentinela seja recriado.

Para possibilitar ao auditor um controle mais efetivo sobre os outros agentes, ele usa uma interface de auditoria que é parte obrigatória de todos os agentes. Essa interface deve oferecer funções básicas como autenticação e autorização, controle de execução e verificação.

O processo de auditoria é totalmente automático. Sendo a checagem fornecida pelo próprio agente auditado (por meio da interface de auditoria), ele não precisa ser reconfigurado nem recompilado quando da geração de novos agentes.

O ciclo de vida do auditor pode ser descrito como a seguir.

1. O QG cria o auditor, como parte do procedimento de inicialização do sistema.
2. O auditor obtém uma referência para cada um dos agentes ativos no hospedeiro corrente.
3. Para cada uma das referências, o auditor inicia o processo de auditoria obtendo a autorização de acesso. Essa autorização serve para evitar que a interface de auditoria seja usada para prejudicar o funcionamento dos agentes.
4. Uma vez autorizado, o auditor inicia a verificação da integridade do agente.
5. Caso a verificação mostre que o agente está corrompido, o auditor ordena o encerramento deste, e solicita ao QG a sua substituição.
6. Encerrada a verificação, o auditor passa à próxima referência de agente.
7. Uma vez que todos os agentes tenham sido verificados a partir de suas referências obtidas pelo auditor, este migra para o próximo hospedeiro da lista de hospedeiros ativos.

8. Antes de migrar ao próximo hospedeiro, o auditor contacta o QG e reporta a este o final da auditoria do contexto, retornando relatórios estatísticos.
9. Caso o QG perceba que, após um determinado período de tempo, o auditor não retornou nenhuma informação, ele inicia uma checagem de funcionamento neste por meio de mensagens sucessivas e periódicas.
10. Caso o auditor não responda às mensagens de checagem de funcionamento, o QG conclui que o auditor teve seu funcionamento abortado, e cria uma nova instância do auditor.
11. O itinerário do auditor é interrompido quando do desligamento do sistema. Nesse momento, o QG ordena ao auditor o seu encerramento. Ao receber esta ordem, o auditor, ao final da checagem do hospedeiro, retorna para o hospedeiro do QG ao invés de seguir para o próximo na seqüência.

Caso o auditor não encontre o agente que ele deseja auditar, ele conclui que esse agente abortou sua operação, e solicita ao QG que esse agente seja recriado. Caso o agente abortado seja o próprio QG, o auditor toma a iniciativa de recriá-lo, no mesmo hospedeiro em que ele esperava encontrá-lo.

4.1.5 Agentes Especiais

Além dos agentes descritos acima, o Sistema Micæl prevê a existência de agentes especiais, que realizam outras tarefas. Um deles é o agente de controle e monitoração de rede, que será visto mais à frente. Qualquer tipo de agente especial que seja necessário pode ser criado e incluído, desde que obedeça às regras de conduta definidas para os agentes Micæl.

Pode ser necessário, inclusive, prover para que o usuário final, ou o administrador do sistema alvo, desenvolva seus próprios agentes; para isso, ele deve se reportar aos modelos de agentes e às bibliotecas de ações predefinidas.

Como já foi dito, existem também serviços e aplicativos presentes nos hospedeiros que são especialmente vulneráveis a ataques, tais como servidores de páginas HTTP ou de correio eletrônico, ou então de terminal virtual. Para cada caso, pode ser desenvolvido um agente especial, que é ao mesmo tempo destacamento (pois é criado sob demanda, ao se perceber a presença do serviço crítico, e é específico para a situação) e sentinela (pois fica ativo não apenas durante a ocorrência de uma anomalia ou

ataque, mas durante todo o funcionamento do sistema). Outros agentes especiais, com a mesma característica mista de destacamento e sentinela podem ser imaginados, como a seguir.

- **agente Monitor de Rede** - A documentação do SNMP [28,26,32] chama a atenção para o fato de que, apenas com sua infra-estrutura básica, não é possível reunir informações relativas à rede. Por isso, foi criado o RMON, que é uma infra-estrutura para o monitoramento da rede em si. O RMON usa equipamentos especiais, chamados de *probes*, para recolher dados diretamente da rede (em especial, de redes de difusão ou barramento central) e compilá-los numa MIB. Assim, definimos um agente especial, chamado Monitor de Rede. Esse agente trabalha em parceria com os *probes* RMON, podendo descobrir ataques de bloqueio da rede (*flooding*, *spoofing*, DoS, etc.) e falhas de funcionamento que coloquem em risco a rede (por exemplo, uma placa de rede monopolizando o barramento).
- **agente Majordomo** - O agente Majordomo é um agente estático, que é carregado antes da inicialização do sistema, e é responsável pela ativação inicial do *aglets server*. Ele também provê a disponibilidade local do código das classes de cada um dos agentes, possibilitando a atualização automática das versões dos agentes.

4.2 Características de Mobilidade

O Sistema Micæl utiliza um enfoque diferenciado da mobilidade de seus agentes, com relação ao enfoque utilizado em outros sistemas de agentes móveis. Nesses outros sistemas, o agente migra para uma localização, realiza algum processamento, reúne os resultados desse processamento, e migra para outra localização, repetindo esse procedimento até que todas as localizações desejadas sejam visitadas e processadas. O agente então retorna à localização inicial, e entrega os dados resultantes à entidade desejada (o usuário ou outro agente).

Os agentes de Micæl, no entanto, não obedecem a esse padrão de comportamento; eles são criados numa origem, migram para a localização desejada, e realizam lá o processamento, porém mantêm contato constante com uma entidade centralizadora (o QG), para quem repassam dados compilados (relatórios de detecção). Ao final do

processamento, outros dados também são entregues (relatórios de comportamento) ao centralizador. Mas o objetivo final do processamento não é a obtenção desses dados, mas sim a detecção de padrões de ataques, e a reação a estes. Por isso, pode-se dizer até que a única função da mobilidade nesta arquitetura é possibilitar configuração dinâmica da distribuição dos agentes. O que não é uma situação ruim, uma vez que graças a esta facilidade é possível disponibilizar recursos no ponto onde eles são realmente necessários, no momento em que eles se façam necessários.

Além disso, o uso de um agente auditor, migrando entre as localidades componentes do domínio de controle desejado, resolve o problema de garantir a integridade do funcionamento dos agentes. Se esta solução não fosse adotada, seria necessário incluir em cada agente um módulo completo de auditoria, o que causaria um consumo adicional de recursos. Caso um agente malicioso fosse inserido no sistema, ele simplesmente não faria o controle de integridade desejado. Um agente subvertido poderia ter o seu módulo de auditoria desativado. Essa possibilidade não existe se o módulo de auditoria faz parte de um outro agente.

Uma outra possibilidade, que não será explorada neste trabalho, é a de utilizar a mobilidade dos agentes para rastrear ataques remotos. É comum que atacantes procurem mascarar sua verdadeira localização disparando sessões remotas a partir de outras sessões remotas. Um agente rastreador poderia seguir a trilha de sessões remotas até a localização real do atacante, produzindo provas importantes para efeitos legais.

4.3 Comunicações entre Agentes

Existem 3 tipos básicos de mensagens trocadas entre os agentes: *mensagens de controle*, *de alerta* e *de relatório*. De maneira geral, mensagens de controle são usadas entre um agente em trânsito e o QG. São usadas para alimentar o agente com sua configuração inicial, para ordenar seu retorno à localização do QG para encerramento do sistema, e para avisar aos agentes de que o QG ou o Auditor mudaram de localização.

Mensagens de alerta são trocadas entre os destacamentos e sentinelas ativos numa determinada localização, informando aos outros que atividades suspeitas foram detectadas por um agente. A mensagem de alerta contém informação suficiente para que os receptores possam correlacionar sua ocorrência com os dados coletados e decidir por algum procedimento de reação.

Já as mensagens de relatório têm função informativa, e são enviadas pelos agentes ao QG, que as reúne e compila. A partir destas mensagens é possível levantar estatísticas e coletar informação para a geração de novos agentes ou o aperfeiçoamento dos existentes.

Este trabalho não contempla a construção de ferramentas para a análise dos relatórios gerados pelo QG.

4.4 Considerações Finais do Capítulo

Apresentamos aqui a arquitetura de um sistema multi-agente para detecção e reação a atividades intrusivas e ataques, baseado em agentes de software autônomos e móveis. Esta arquitetura, por distribuir dinamicamente a carga de utilização entre as diversas máquinas hospedeiras componentes do sistema, possibilita o uso do sistema em ambientes heterogêneos compostos por máquinas servidoras, *workstations* e computadores pessoais (PCs). Por estar presente em todos os níveis da rede, o sistema oferece uma proteção mais efetiva contra ataques e tentativas de invasão.

Alguns pontos que podem ser objeto de trabalhos futuros são o uso de inteligência nos agentes, a criação de ferramentas de análise das massas de dados coletados, e a federação dos quartéis generais.

Os agentes podem ter o seu desempenho melhorado com o uso de mecanismos de inteligência e aprendizado. Em especial, o Quartel General pode otimizar a escolha do código numa solicitação, e o sentinela e os destacamentos podem melhorar o procedimento de detecção.

Com o uso de ferramentas de análise das massas de dados coletados pelos agentes detectores, o processo de criação de novas versões de destacamentos específicos para cada anomalia encontrada se tornará mais simples.

A federação dos quartéis-generais permitirá a cooperação entre domínios de controle, o aproveitamento de códigos, e a possibilidade de substituição do quartel-general responsável por uma região da rede que por acaso fique isolada do quartel-general original.

Neste trabalho, o mecanismo de comunicação inter-agentes escolhido foi o uso de mensagens ATP. Uma outra sugestão de trabalhos futuros é a avaliação de outros mecanismos de comunicação, ou mesmo outros ambientes de mobilidade.

Capítulo 5 - Implementação

Este capítulo discorre sobre a experiência da implementação do protótipo de um sistema conforme a arquitetura descrita no capítulo anterior.

A primeira decisão tomada no início do desenvolvimento de um sistema é a escolha da linguagem a ser utilizada na programação. Para este trabalho, que pretende-se ser executado em diversos tipos de arquiteturas de *hardware* e sistemas operacionais interligados num ambiente de rede, vemos que a linguagem Java é a que associa estas facilidades com o melhor desempenho. A característica de modularidade e encapsulamento de Java torna-a muito apropriada para o desenvolvimento de sistemas multi-agentes. A escolha do ambiente de mobilidade, disponível em Java, veio solidificar a escolha de Java como linguagem de programação dos agentes do Sistema Micæl.

O ambiente de mobilidade, como foi dito no capítulo 3, precisava prover, além das primitivas básicas de movimentação, facilidades de comunicação com um nível mínimo de segurança. Para o desenvolvimento do protótipo, o ambiente precisava ainda ser de fácil acesso e razoavelmente estável. O ambiente de mobilidade que conjugou a maior quantidade dos requisitos foi o ASDK (*Aglets Software Development Kit*). As dificuldades enfrentadas na operação do ASDK são descritas mais à frente.

Uma vez decididos a linguagem de programação e o ambiente de mobilidade, passou-se à decisão do ambiente objeto do protótipo. A arquitetura não prevê qualquer restrição ao ambiente objeto, mas na prática limitações de disponibilidade do ambiente para testes e de compatibilidade dos *softwares* utilizados reduziram o leque de opções. Sistemas operacionais como Novell Netware e HP-UX não dispunham de máquina virtual Java[5]. O Laboratório de Redes e Multimídia do NCE, por sua vez, não dispunha de equipamentos com Solaris ou MacOS, apenas Windows NT e Linux. Assim, decidiu-se incluir no protótipo apenas agentes voltados para estes dois últimos. Problemas encontrados posteriormente forçaram a exclusão também do Linux.

Com todas as necessidades iniciais equacionadas, iniciou-se a implementação propriamente dita. Nesta fase foi criada a estrutura utilizada no protótipo, e que servirá para o desenvolvimento de versões futuras, mais completas. Nessa estrutura, os agentes

---

[5] O S.O. Novell Netware 5, que tem uma máquina virtual Java integrada aos programas de sistema, não estava disponível quando do início das pesquisas.

são separados em três grupos ou *pacotes*: o grupo básico, imutável, o grupo dos destacamentos especializados, e o grupo dos sentinelas especializados. Essa decisão facilita o desenvolvimento de novos agentes por terceiros, uma vez que tanto o grupo dos destacamentos especializados conta com interfaces e agentes-modelo que servem de base para o desenvolvimento.

A solução adotada no protótipo para a comunicação entre os agentes será detalhada no seu formato, na seção 5.5. Já a seção 5.6 descreve as bibliotecas desenvolvidas para capturar e analisar o tráfego da rede. Estas bibliotecas são portáteis, compactas e de desempenho mais do que razoável, levando-se em conta que estão sendo executadas num ambiente interpretado. Antes de passar às conclusões, a seção 5.7 descreve os exemplos de destacamentos desenvolvidos. Os códigos-fonte dos agentes desenvolvidos neste protótipo estão disponíveis na mídia anexa.

5.1 Linguagem de Programação

Como já foi dito no capítulo 3, Java foi a linguagem escolhida para a implementar o protótipo. A especificação de Java torna possível que programas compilados sejam executados em diferentes plataformas de forma eficiente. Essas características, entre outras, tornam esta linguagem uma das mais apropriadas para o desenvolvimento de programas que de alguma forma utilizarão ambientes de rede para sua execução.

O uso de POO (Programação Orientada a Objetos) também soluciona uma questão delicada, a de como delimitar o contorno de um agente no código fonte. Nesse modelo, um agente é mapeado diretamente em uma classe de objetos Java.

No desenvolvimento do protótipo foi utilizada a versão do JDK (*Java Development Kit*) 1.1.8, que foi a última versão publicada antes do advento da plataforma Java2.

5.1.1 O problema do Desempenho

O fato de Java ser uma linguagem interpretada impõe uma limitação severa ao desempenho do sistema. Mesmo com o uso de JITs (*Just In Time Compilers*)[6] ainda há um grande *overhead* associado ao modelo de memória e de acesso aos dados. Por esse motivo, em alguns pontos críticos optou-se por não utilizar Java, e sim C++ (compilado). Essas bibliotecas externas são acessadas por meio da interface JNI.

---

[6] Programa que traduz o código binário Java em linguagem de máquina do processador hospedeiro.

5.1.2 Heranças Múltiplas

Java define duas maneiras de geração de subclasses. Na primeira, a subclasse estende a superclasse (figura 1); a subclasse pode, assim, acrescentar métodos e variáveis aos membros da superclasse, ou então alterar o funcionamento de algum método desta, sobrecarregando-o. A limitação desta maneira de herança é que não é permitida a herança múltipla, ou seja, a subclasse só pode estender uma única superclasse. Essa limitação é contornada pelo uso de *interfaces*, que são módulos Java que definem variáveis e métodos, mas que não incluem a implementação destes métodos. A classe que implementa uma interface deve obrigatoriamente implementar todos os métodos descritos na interface. Essa implementação pode ser feita pela própria classe ou pela sua superclasse. Estas duas maneiras de implementar herança múltipla foram amplamente utilizadas na programação dos agentes do protótipo.

```
package jdq.Micael;

public final class DestacamentoAgente extends MicaelAglet
                                        implements Auditable {
...
```

Figura 5.1 - Trecho de código extraído do protótipo exemplificando a herança múltipla

5.1.3 Estrutura Hierárquica

A linguagem Java permite a organização das classes em pacotes (*packets*). Uma classe pertencente a um pacote tem alguns privilégios junto às outras classes, em especial no acesso aos seus membros. Além disso, pacotes podem incluir sub-pacotes, que criam a estrutura hierárquica. Note-se, porém, que um sub-pacote é considerado um pacote separado, e não parte do pacote original. Estes pacotes e subpacotes criam uma árvore, cujas folhas são as classes.

Uma classe de objetos é unicamente identificada pelo seu nome e seu pacote. Duas classes podem ter o mesmo nome, desde que estejam em pacotes diferentes. A qualificação completa de uma classe é feita escrevendo-se os nomes dos pacotes encontrados na ramificação da árvore de pacotes, do nível mais alto para o mais baixo, terminando no nome da classe. Os nomes são separados por pontos. Abaixo, alguns exemplos de pacotes predefinidos.

- `java.lang` – é o pacote básico da linguagem Java. Nele estão todas as classes e interfaces básicas (`Object`, `String`, `Integer`, `System`, `Serializable`, etc).
- `java.net` – contém todas as classes relacionadas à manipulação de dados em redes. Ex.: `URL`, `InetAddress`, `Socket`, etc.
- `java.io` – contém todas as classes relacionadas com a entrada e saída de dados em arquivos. Ex.: `File`, `Reader`, `PrintStream`, etc.
- `java.awt` – contém classes relacionadas com a apresentação de gráficos em terminais apropriados. Ex.: `Label`, `Window`, `TextArea`, etc.
- `javax.swing` – contém classes destinadas a implementar algumas facilidades presentes em outras classes, de forma mais eficiente. Ex.: `Timer`, `JLabel`, `JFrame`, etc.

Qualquer programador pode criar sua própria ramificação da árvore de nomes; porém, é recomendado que seja usado para nomear os pacotes o nome de domínio da entidade que o publica, de forma a evitar coincidência de nomes. Assim, o ambiente ASDK usa, para o nome dos seus pacotes, o prefixo "`com.ibm`". Ex.: `com.ibm.aglet`, `com.ibm.awt`, `com.ibm.agletx`, etc.

Escrever o nome qualificado da classe de um objeto a cada vez que se faz referência a ele é tremendamente aborrecido, além de contraproducente; assim, a linguagem permite que o programador *importe* o espaço de nomes de um determinado pacote, de forma que, não hávendo duplicidade de alguma classe, ele possa usar apenas o nome da classe, sem usar o pacote qualificado. Não é necessário, porém, qualificar as classes pertencentes do pacote `java.lang`, nem importá-lo.

Neste texto os nomes de classe componentes dos pacotes `java.lang`, `java.net`, `java.util`, e `com.ibm.aglet` serão abreviados, quando conveniente. Ex.: `String` em vez de `java.lang.String`, `URL` ao invés de `java.net.URL`, `AgletProxy` ao invés de `com.ibm.aglet.AgletProxy`. Além dessas, a classe `javax.swing.Timer` será escrita somente como `Timer`[7].

5.1.4 Objetos Predefinidos

A biblioteca básica da linguagem Java inclui uma série de objetos predefinidos, de grande utilidade para o programador. Alguns exemplos são listados a seguir.

---

[7] Não confundir com a classe `java.util.Timer` de Java2.

- `java.lang.Object` – Superclasse de todos os objetos de Java. Define métodos básicos de sincronização e comparação.
- `java.lang.String` – Define uma seqüência de caracteres, bem como métodos para extração de trechos dessa seqüência, comparação com outras *strings*, impressão na console e apresentação em janelas, leitura a partir do teclado, conversão de formatos internacionais, etc.
- `java.net.URL` – Define métodos para lidar com URLs (*Universal Resource Locators*). Um URL é descrito com uma *tupla* (protocolo,endereço,porta,arquivo), e costuma ser escrita no formato `protocolo://endereço:porta/arquivo`. Com o objeto `java.net.URL`, é possível abrir conexões com máquinas remotas e acessar diretamente o recurso desejado.
- `java.util.Hashtable` – Define métodos para lidar com a estrutura de armazenamento conhecida como *hash*, que é a matriz associativa. Com o *hash*, é possível armazenar e recuperar informações com base em qualquer tipo de

5.2 Ambiente de Mobilidade

A escolha do ambiente de mobilidade foi feita levando-se em conta principalmente a disponibilidade da solução escolhida, o ambiente ASDK (*Aglets Software Development Kit*). A arquitetura, porém, pode ser implementada em qualquer outro ambiente de mobilidade que forneça funcionalidade suficiente para tal. Outro fator importante para a escolha é o fato do ASDK ser escrito em Java.

5.2.1 ASDK Versões 1.1b3, 1.1 e 1.2

Foram utilizadas duas versões do ASDK no desenvolvimento do protótipo. Na primeira fase, foi utilizado o ASDK 1.1beta3, que ainda era proprietário da IBM. Essa versão, em fase de beta teste, tinha expiração para dezembro de 2000. Nesse meio tempo, o projeto ASDK foi adotado por uma equipe própria, e tornado de código aberto (***Open Source***). Com isso, foi liberada a versão 1.1, que é basicamente a mesma que a 1.1beta3 (sem a checagem de data) com a remoção de alguns *bugs*. A versão 1.1 foi adotada para o desenvolvimento do protótipo a partir de setembro de 2000.

A respeito do projeto ASDK Open Source, atualmente está disponível o ASDK 1.2, que permitirá o uso de Java2 (JDK 1.2 e 1.3). Porém, como a mudança de Java para

Java2 implica em alterações profundas no esquema de criptografia e reconhecimento de identidade de usuários, o ASDK 1.2 apresenta problemas de robustez. Por isso, foi descartado o seu uso neste projeto.

5.2.2 Conceitos básicos do ASDK

A unidade de execução do ASDK é o *Aglet Server*. Cada *aglet server* contém um ou mais *contextos*, que agrupam os *aglets* (agentes móveis). Para criar um *aglet* a partir de outro, o programador usa o método `AgletRuntime#createAglet(URL codebase, String classname, Object init)`. Esse método retorna ao programador um objeto `AgletProxy` que é um ponteiro que dá acesso às facilidades de mobilidade do novo *aglet*. Para provocar a migração do aglet para uma nova localização, há duas formas, dependendo de quem é a iniciativa da migração: `Aglet#dispatch()` e `AgletProxy#dispatch()`. A primeira forma é usada quando a iniciativa é do próprio agente, enquando na segunda a iniciativa é de outro agente. Em ambas, há duas possibilidades de informar o destino: `#dispatch(URL dest)` e `#dispatch(Ticket tck)`. A primeira é intuitiva; na segunda, mais elaborada, há a possibilidade de especificar um nível de confidencialidade e qualidade de transmissão desejados. Via de regra, todas as operações de mobilidade possíveis ao *aglet* por iniciativa própria estão disponíveis de forma análoga a outros *aglets* por meio do objeto `AgletProxy`. Entretanto, esse acesso é feito de forma controlada (usando autorizações de segurança) e independentemente da localização do objeto alvo.

Quando o *aglet* é iniciado ele executa o método `Aglet#run()`. Como Java não permite acessar o estado de execução, não é possível restaurá-lo diretamente após uma movimentação; assim, sempre que o *aglet* é reativado numa nova localização o método `Aglet#run()` é reexecutado. Caso seja necessário preservar de alguma forma o estado de execução, ele deve ser de alguma forma codificado em variáveis membro não transientes.

Existem ainda métodos que podem ser executados imediatamente em momentos especiais do ciclo de vida do agente: inicialização [`Aglet#onCreation(Object init)`], finalização [`Aglet#onDisposing()`], saída do ambiente [`MobilityAdapter#onDispatching(MobilityEvent ev)`], chegada a um novo ambiente [`MobilityAdapter#onArri-`

val(MobilityEvent ev)] e reversão[8] [MobilityAdapter#onReverting(MobilityEvent ev)].

Para enviar uma mensagem (*unicast*), o *aglet* usa o método AgletProxy#sendMessage(Message msg). Uma vez enviada a mensagem, o método #handleMessage(Message msg) é ativado no *aglet* receptor. Esse método pode aceitar a mensagem ou recusá-la. O receptor pode também gerar como resposta uma nova mensagem, ou uma exceção. Caso a mensagem seja recusada, uma exceção do tipo NotHandledException é gerada para o agente remetente. Exceções enviadas em resposta a mensagens são recebidas como exceções do tipo MessageException. Uma vez capturada essa exceção, a exceção original pode ser recuperada.

Ao enviar uma mensagem, o *aglet* remetente fica bloqueado até que a mensagem seja processada ou recusada, recebendo como retorno da chamada do método a mensagem resposta. Alternativamente, é possível enviar uma mensagem que não admite resposta [sendOneWayMessage(Message msg)], ou então que não bloqueie o remetente [sendAsyncMessage(Message msg), sendFutureMessage(Message msg)]. Dentro do receptor, as mensagens são tratadas de forma seqüencial, a menos que o método Aglet#exitMonitor() seja invocado. Esse método, quando chamado de dentro de #handleMessage(), permite que uma nova mensagem seja processada (em outra *thread*). Nesse caso, o programa do agente deve prover sincronização.

O *aglet server* é capaz de distribuir mensagens *multicast* dentro de um contexto. Os *aglets* interessados em receber algum tipo de mensagem *multicast* precisam apenas inscrever-se junto ao *Runtime* (núcleo de execução do *aglet server*) indicando a identificação (definida como um objeto *String*) da mensagem desejada. Quando outro *aglet* envia uma mensagem *multicast* com uma determinada identificação, todos os agentes dentro do mesmo contexto que tenham se inscrito para receber as mensagens com essa identificação recebem a mensagem, como se o outro *aglet* tivesse enviado uma mensagem *unicast* diretamente a este.

O objeto AgletProxy, uma vez gerado, só tem validade enquanto o *aglet* mantenha-se na mesma localidade. Quando a movimentação é de iniciativa de outro agente, o método AgletProxy#dispatch() retorna um novo *proxy* validado para a nova

---

[8] *Reversão* é quando um agente captura um outro agente em outro contexto e o desloca para o seu próprio contexto. É usado quando a configuração da máquina local não permite que conexões sejam estabelecidas a partir da máquina remota, como por exemplo quando a máquina local está atrás de um *firewall*.

localização. Porém quando a iniciativa é do próprio agente, todos os objetos *proxy* que referenciam este agente se tornam inválidos. Outros agentes que necessitem interagir com este precisam, então, de uma forma de revalidar suas referências. Esta tarefa pode ser realizada utilizando-se um serviço de registro

Para que seja possível recuperar *proxies* a partir do *AgletID*, é necessário que um outro módulo de serviços esteja disponível, o serviço `MAFFinder`. O serviço `MAFFinder` é implementado como um serviço RMI, e deve ser executado em uma máquina acessível às máquinas onde o ambiente ASDK está sendo executado. Para ativar o uso do serviço `MAFFinder`, o *aglet server* deve ser informado de sua localização.

5.3 O Ambiente Objeto

Um dos objetivos deste projeto é desenvolver um mecanismo de proteção contra acessos não autorizados que possa ser utilizado de maneira simultânea e coordenada em todos os computadores presentes em um determinado domínio de rede local, independente de serem servidores de rede ou simples estações de trabalho. Por isso, planejava-se originalmente incluir no protótipo agentes destinados a trabalhar em vários ambientes distintos, entre eles o Windows NT e o Linux, que são vistos abaixo.

5.3.1 O Ambiente Windows NT

O Windows NT é o equivalente corporativo do conhecido Sistema Operacional Windows 95. Ele está presente em duas modalidades, idênticas no funcionamento: o NT Servidor e a o NT Workstation. A diferença entre as duas modalidades se manifesta apenas na forma de configuração, sendo indiferente para o programador qual delas será utilizada.

O Windows NT teve na versão 3.5 a sua primeira versão operacional. Essa versão apresentava muitos problemas, tais como travamentos, incompatibilidades e falhas de segurança. Além disso, essa versão trazia muitos fragmentos de código de 16 bits, originários ainda do MS-DOS e do Windows 3.1. Esses códigos, além de obsoletos, não utilizavam plenamente a capacidade dos processadores presentes na época, todos capazes de processamento em 32 bits.

A versão 4 do Windows NT apresentou um impressionante melhoramento quando comparado à 3.5. Entre outras coisas, o *kernel* foi totalmente redesenhado, de forma que os fragmentos de código de 16 bits foram praticamente eliminados. Os mecanismos de segurança também foram redesenhados e reforçados. Essas mudanças

porém esbarraram no problema da retrocompatibilidade: muitos programas desenvolvidos para as versões anteriores usavam características de projeto que a experiência já tinha provado serem vulneráveis em termos de segurança (o principal exemplo é a pilha de protocolos utilizada pela Rede Microsoft, que é a base dos serviços de rede do Windows NT). Vários pacotes de correções, conhecidos como *Service Packs* (SPs) foram publicados, mas ainda assim perdura a fama do Windows NT ser uma verdadeira peneira de furos de segurança. Apenas para exemplificar, o conhecido programa nMap, ao realizar uma varredura de reconhecimento em uma máquina com o Windows NT 4 e o *Service Pack* 5 instalado (a versão corrente é a 6a), informou que a previsibilidade dos números de seqüência de segmento e *acknowledge* do protocolo TCP/IP era do nível "*Trivial Joke*" (brincadeira de criança). Para efeito de comparação, uma máquina Novell Netware 4.12 com o *Support Pack* 8 é classificada como "*Worthy challenge*" (desafio instigante). Quanto maior a previsibilidade dos números de seqüência, mais fácil é de desferir ataques de *Hijack* sobre as conexões dessa máquina. Em resumo, o ambiente Windows NT é um cliente potencial para qualquer produto de segurança ativa.

O Windows NT é um ambiente multi-programado com *time sharing*, com 7 níveis de prioridade distintos. A unidade de escalonamento é a *thread*. Cada *thread* NT, ao ser escalonada para execução, recebe o controle da CPU por um intervalo de tempo. Quando o intervalo passa, o *kernel* retoma o controle (*preempção*). Essa característica afeta o funcionamento da máquina virtual Java, que é definida para ser executada sobre *threads* não preemptivas (conforme a especificação das *Green threads* do sistema operacional Solaris).

5.3.2 O Ambiente Linux

O ambiente Linux[9] é um Sistema Operacional semelhante ao Unix, e que é conhecido por ser totalmente construído em código aberto. Ele está disponível para cópia gratuita na internet, e recebe contribuições de entusiastas de todas as partes do mundo. Como toda a família Unix, o Linux apresenta uma estabilidade muito superior aos sistemas operacionais da família Windows. Mas também é significativamente mais complexo, tanto em sua configuração como na operação. O controle de segurança dentro do Linux é muito mais elaborado do que na família Windows. A separação entre

[9] http://www.linux.org, http://www.kernel.org, http://www.ldp.org.

os modos de operação de usuário e supervisor é total. Mas, como toda peça de software, também está sujeito a falhas e vulnerabilidades, que podem ser utilizadas por agressores para acessos indevidos, ou então provocar o bloqueio do sistema. Poucas dessas vulnerabilidades estão dentro do *kernel* do Linux; a maioria está associada aos programas utilitários do sistema.

O *kernel* do Linux também é multi-processado com *time-sharing*. A unidade de escalonamento é o processo. É possível, com o uso de bibliotecas a nível de usuário, implementar *Green Threads*, porém sem preempção. Essa característica afeta o funcionamento da máquina virtual, em especial nas interações do funcionamento dos subsistemas de vídeo (AWT) e execução remota de métodos (RMI). Isso ocorre porque o RMI define *threads* que realizam *live polling*, ou seja, ficam checando continuamente uma condição externa. Como o AWT faz a atualização do conteúdo das janelas e a checagem dos eventos de entrada com *threads* da mesma prioridade usada pelo RMI, a atualização das janelas fica bloqueada até que o RMI receba alguma solicitação (quando finalmente a *thread* do RMI devolve o controle do sistema). Nessa situação, o programa fica como que travado (e também todas as outras janelas Java ativas). Esse problema não é verificado no ambiente Windows (que, fugindo à especificação, usa *threads* com preempção), nem no Java2 (que usa prioridades diferentes para os dois subsistemas).

Um problema foi detectado na execução do ambiente ASDK no Linux. Ocorre que o serviço MAFFinder, essencial para o funcionamento do protótipo, utiliza tanto o subsistema RMI como o AWT, provocando a situação descrita acima. Várias alternativas foram tentadas, por exemplo alterar a prioridade das threads RMI, porém nenhuma funcionou a contento. Como não foi possível garantir o funcionamento do ambiente para os testes, decidiu-se pela exclusão do ambiente Linux.

5.4 Estrutura do Código

O código da implementação do protótipo foi dividido em duas partes: a parte básica, contendo o ambiente de mobilidade e controle, bem como os agentes básicos, e a parte extendida, contendo os agentes avançados. Esses agentes, cada um correspondendo a uma classe de objetos Java, foram reunidos num pacote Java nomeado `jdq.Micael`. Este pacote contém os agentes/objetos listados abaixo.

- Interface `jdq.Micael.MicaelAglet` – Define o comportamento básico necessário de um aglet (agente móvel dentro do ambiente ASDK) para o

ambiente Micæl. Todos os agentes, básicos ou avançados, implementam esta interface.

- Interface `jdq.Micael.Auditable` – Define o comportamento de um agente para o agente auditor. Por meio dessa interface, o auditor pode controlar o agente, checar o seu estado, forçar o seu encerramento, etc. Todos os agentes, básicos ou avançados, implementam esta interface.
- Classe `jdq.Micael.Micael` – Define o programa inicial do ambiente. É responsável pela carga e configuração do QG.
- Classe `jdq.Micael.QuartelGeneral` – Contém o código do agente Quartel General (QG). Várias threads são disparadas, inclusive uma responsável pela movimentação do QG para uma nova localidade, no caso do QG ser notificado de que o *host* atual não é mais seguro.
- Classe `jdq.Micael.Auditor` – Contém o código do agente auditor. Dispara várias threads, sendo uma delas responsável pela rotina de auditoria do contexto corrente. Nessa rotina, os agentes ativos são avaliados como instâncias da interface `jdq.Micael.Auditable`. Caso a avaliação seja verdadeira, o auditor identifica-se para a
- Classe `jdq.Micael.Sentinela` – Contém o código de um sentinela básico. O sentinela básico realiza duas operações ao se ativar: primeiro ele verifica qual o ambiente operacional presente, e convoca um sentinela especializado nesse ambiente operacional. Em seguida, ele verifica se o *host* corrente é o mesmo do QG. Caso afirmativo, ele convoca o Sentinela especializado na defesa do QG.

Outros dois pacotes importantes para o protótipo são `jdq.Micael.Sentinelas` e `jdq.Micael.Destacamentos`. O primeiro contém os agentes/objetos sentinelas especializados, e o último, os destacamentos desenvolvidos. A propósito, todos os destacamentos devem ser desenvolvidos como pertencendo ao pacote `jdq.Micael.Destacamentos`, e implementar, além das duas interfaces mencionadas acima, a interface `jdq.Micael.Destacamentos.Destacamento`. Além disso, o desenvolvedor pode aproveitar o código já desenvolvido, estendendo a classe abstrata `jdq.Micael.Destacamentos.DestacamentoBasico`, que contém métodos úteis para qualquer destacamento.

Similarmente, qualquer sentinela especializado deve implementar as interfaces `jdq.Micael.Auditor` e `jdq.Micael.MicaelAglet`, mas como os sentinelas especializados atuam como destacamentos, devem implantar também a interface `jdq.Micael.Destacamentos.Destacamento`. Finalmente, a interface `jdq.Micael.Sentinelas.Sentinela` define o comportamento mínimo dos sentinelas especializados. Boa parte dos métodos necessários a este grupo de agentes pode ser aproveitado estendendo-se a classe abstrata `jdq.Micael.Sentinelas.SemtinelaBásico`.

Atualmente estão desenvolvidos os agentes sentinelas especializados listados abaixo.

- `SentinelaQG` – faz o controle de segurança do QG.
- `SentinelaNT` – especializado na verificação do ambiente Windows NT
- `SentinelaLinux` – especializado na verificação do ambiente Linux.

5.5 Comunicação entre os agentes

Um agente Sentinela especializado, ao ser iniciado em um contexto, inscreve-se para receber mensagens *multicast* do tipo "Alerta". Quando ele encontra uma anomalia, ou identifica um ataque, ele gera uma mensagem desse tipo, avisando aos outros sentinelas ativos. Como resultado, os agentes podem alterar o seu nível de alerta. O sentinela básico também se inscreve para receber tal tipo de mensagem; ao recebê-lo, repassa-a ao QG.

5.6 Detecção de Tráfego de Rede

Para possibilitar a captura e análise do tráfego de rede destinado à máquina sendo defendida, foram desenvolvidos três pacotes: `jdq.libpcap2`, `jdq.NetAnalyser`, `jdq.AntiScan` e `jdq.IPGen`. Esses quatro pacotes trabalham de forma complementar,

5.6.1 O Pacote `jdq.libpcap2`

O pacote `jdq.libpcap2` contém classes que implementam uma adaptação da biblioteca Libpcap. A biblioteca Libpcap [15] proporciona ao programador uma forma simples e padronizada de acesso ao tráfego da rede local. Foi desenvolvida inicialmente para Unix BSD, e posteriormente adaptada para vários sistemas operacionais, inclusive Windows 9x e NT (esta última tradução conduzida pela equipe do Politecnico de Torino [21]). Como todas estas versões da biblioteca foram desenvolvidas tendo como alvo

programas escritos em linguagem C, foi necessário estabelecer uma interface JNI que adaptasse as funções definidas na biblioteca libpcap para uso em Java.

Foram desenvolvidas, assim, duas classes Java com essa ligação: a classe `jdq.libpcap2.PCap` e a classe `jdq.libpcap2.CaptureStream`. A primeira funciona como um gerador de eventos, onde o programador deve registrar um objeto que defina a interface `jdq.libpcap2.PCapHandler`. A cada pacote capturado, o método `PCapHandler#callback()` desse objeto é chamado, possibilitando ao programador acessar o seu conteúdo. A segunda classe trata os pacotes recebidos como um fluxo contínuo. O programador chama o método `CaptureStream#readPacket()` para obter o próximo pacote.

Ambas as classes possuem facilidades tais como:

- gravar o tráfego capturado em um arquivo, que pode ser lido posteriormente (o arquivo tem o mesmo formato usado pelo programa tcpdump [15]);
- analisar o conteúdo de um arquivo de tráfego;
- trabalhar opcionalmente em modo promíscuo;
- definir filtros de pacotes, reduzindo a quantidade de pacotes analisados.

5.6.2 O pacote `jdq.NetAnalyser`

O pacote `jdq.NetAnalyser` contém classes que realizam a análise dos cabeçalhos dos pacotes capturados com a classe `jdq.libpcap2.CaptureStream`. A classe `jdq.NetAnalyser.NetAnalyser`, principal classe do pacote, define um gerador de eventos, onde o programador interessado em analisar o tráfego capturado da rede deve registrar objetos derivados de `jdq.NetAnalyser.Event.Listener`. Vários grupos de eventos estão definidos:

- Eventos genéricos: **Novo Pacote**, gerado a cada novo pacote que é lido, antes dele ser analisado, e **Protocolo de Transporte Genérico**, gerado para pacotes que não sejam oriundos de redes Ethernet.
- Eventos Ethernet: **Protocolo Ethernet** é gerado assim que é identificado que o pacote é oriundo de rede do tipo ethernet; **Pacote Ethernet Genérico** é gerado ao constatar que o pacote não é IPv4 ou ARP/RARP.
- Eventos ARP: Pacote ARP/RARP, Resposta ARP/RARP
- Eventos IPv4 Genérico: Remontagem, checagem de campos, uso de opções IP, carga útil.

- Eventos TCP: Checagem de campos, teste de *flags*, conexões, uso de opções TCP, carga útil.
- Eventos de Conexão: Abertura e fechamento de conexão, chegada de segmento, acompanhamento de números de seqüência, checagem de protocolo.
- Eventos UDP: Checagem de campos, acesso, uso de opções UDP, carga útil.
- Eventos ICMP: Checagem de campos, funcionalidade, uso de opções ICMP, carga útil.

Apenas o protocolo IPv4 e seus derivados são analisados. Mas o evento "carga útil", que é gerado para todos os protocolos, a partir do nível IP, torna possível implantar extensões ao analisador.

A análise do tráfego é conduzida em *thread* própria, estando associada a uma interface de rede única. Quando uma situação de interesse é encontrada, é gerado um evento equivalente. Nesse evento, a *thread* do analisador desvia para o método `#actionPerformed(jdq.NetAnalyser.event.Event ev)` do *listener* pertinente. O usuário do analisador pode registrar os seguintes tipos de *listener*:

- *Listener* geral (eventos gerais de pacote);
- *Listener* Ethernet (eventos de pacote ethernet);
- *Listener* ARP (eventos de pacotes ARP/RARP);
- *Listener* IPv4 (eventos de pacotes IPv4 em geral);
- *Listener* UDP (eventos de pacotes UDP);
- *Listener* TCP (eventos de pacotes TCP em geral);
- *Listener* de Conexão (eventos de conexão);
- *Listener* ICMP (eventos de pacotes ICMP).

A última funcionalidade do pacote `jdq.NetAnalyser` é a geração de eventos periódicos de estatística. Tal como na análise do pacote, é possível registrar um *listener* para receber os eventos de estatística.

Como a *thread* do analisador (e, por conseqüência, todo o objeto) está associada a uma interface de rede única, máquinas *multi-homed*[10] devem criar uma instância do analisador para cada interface que deva ser analisada. É possível escolher entre o uso ou não do modo promíscuo na interface de rede.

---

[10] Que possuem mais de uma interface de rede em atividade simultânea.

5.6.3 *O Pacote* `jdq.AntiScan`

O pacote `jdq.AntiScan` realiza a checagem do tráfego, auxiliado pelo pacote `jdq.NetAnalyser`, em busca de padrões de *Port Scan* (varredura de portas TCP/UDP). O Apêndice D apresenta os padrões mais conhecidos de *port scan*. Assim como o anterior, ele funciona em esquema de geração de eventos com registro de objeto *listener*, mas não possui *thread* própria; ele utiliza a *thread* do analisador de tráfego. Caso haja mais de uma interface de rede em atividade, novas instâncias do analisador de tráfego devem ser criadas. Este pacote não utiliza o modo promíscuo das interfaces de rede.

Como o processo de descoberta de padrões de *port scan* é muito semelhante ao de busca de sinais de *SynFlood* (inundação por pacotes TCP SYN), o pacote `jdq.AntiScan` foi alterado para contemplar essa detecção. Quando há indícios de estar havendo um *SynFlood*, é gerado um evento próprio, que pode ser aproveitado por um Sentinela.

5.6.4 O Pacote `jdq.IPGen`

O pacote `jdq.IPGen` é utilizado para injetar pacotes na rede, de forma a interferir no funcionamento da rede. Assim como o `jdq.libpcap`, ele é implementado com o uso de métodos nativos (JNI), baseado na biblioteca LibNet [19]. A biblioteca LibNet, desenvolvida em C, permite a programas gerar pacotes IP ou Ethernet, sem o uso do *kernel*, de forma simples e padronizada. Por meio do pacote `jdq.IPGen` o tráfego pode ser gerado de forma simples ou repetitiva, e pode ser diretamente especificado pelo usuário ou lido de um arquivo gerado com o pacote `jdq.libpcap2`. Este pacote não implementa nenhum controle de recebimento do pacote pelo destinatário da mensagem. Para o desenvolvimento do protótipo foi utilizada uma versão da biblioteca modificada para uso em ambiente Windows 95/98/NT, a LibnetNT [5].

5.7 Exemplo de Destacamento: Destacamento AntiFlood

O destacamento `jdq.Micael.Destacamentos.AntiFlood` contém código preparado para minimizar os efeitos de um ataque *SynFlood*. Seu funcionamento consiste em gerar pacotes TCP RST à máquina local, cancelando as solicitações de abertura de conexão pendentes. Para diminuir o impacto da medida corretiva no sistema, este destacamento gera rajadas de pacotes até que o risco de bloqueio seja afastado. Após o fim da

atividade de agressão, o destacamento permanece ativo no *host* por um período de 5 minutos, prevenindo um possível retorno do ataque.

Para ter efetividade, o destacamento AntiFlood precisa ser convocado antes das tabelas de conexões pendentes estourarem sua capacidade. Como já era esperado, isso provoca um aumento nos alarmes falsos, mas diminui o risco do próprio agente ser bloqueado pelo ataque.

É conveniente que, durante o ataque, o Quartel General seja deslocado para evitar o seu bloqueio.

5.8 Considerações Finais do Capítulo

Foram apresentadas neste capítulo as experiências obtidas na implementação do protótipo da arquitetura Micæl, descrita no capítulo 4. As razões pendentes para a escolha da linguagem Java e do ambiente ASDK foram explicitadas. Adicionalmente, uma pequena explicação da operação do ASDK e do JDK foi feita.

Os módulos Java do protótipo foram desenvolvidos com o produto Borland JBuilder, versões 3.0 Standard, 3.5 Foundation e 4.0 Enterprise (variando comforme o *host*).

Os módulos nativos foram desenvolvidos em C++ com o produto Microsoft Visual C++, versão 6.0.

Os pacotes `jdq.libpcap2`, `jdq.NetAnalyser`, `jdq.AntiScan` e `jdq.IPGen`, que servem de base para a captura e análise de tráfego de rede, são subprodutos desta fase do projeto que podem ser burilados para se tornarem ferramentas de uso geral em pesquisas de rede local. Como são escritos em sua maior parte em Java, possuem a característica de portabilidade desta linguagem. Os destacamentos exemplo gerados são bastante simples, graças ao uso dos pacotes descritos acima.

Uma sugestão de trabalho futuro é a extensão destes pacotes para o tratamento do protocolo IPv6, ou então de capsulas de redes ativas. Também seria interessante transformar o analisador de tráfego num *singleton*, facilitando o seu uso por múltiplos agentes detectores.

Outra sugestão seria o desenvolvimento de um módulo de inicialização automática. Atualmente é necessário disparar o *aglets server* manualmente em cada uma das máquinas a serem defendidas, preencher um arquivo de configuração com a lista dessas máquinas, e por fim, disparar o agente de carga (`jdq.Micael.Micael`).

No próximo capítulo são apresentados os resultados dos testes com o protótipo.

Capítulo 6 - Análise dos Resultados

A execução do protótipo da arquitetura, conforme descrito no capítulo anterior, apresentou resultados que são analisados a partir de agora. Esses resultados são avaliados em relação à correção de detecção e ao tempo de reação.

Foram aplicados três testes ao sistema. No primeiro, procurou-se verificar o tempo de configuração inicial, com relação ao número de *hosts* ativos no sistema. Esse tempo inclui a criação e o posicionamento do Quartel General, a criação e o posicionamento dos sentinelas de cada um dos *hosts*, e a criação e o posicionamento do auditor. O tempo não inclui a criação de sentinelas especializados, uma vez que esses são considerados destacamentos e portanto são criados com o sistema em funcionamento (após a fase de configuração inicial). Verificou-se nesse teste também qual a influência da variação do número de *hosts* em uso pelo sistema.

No segundo teste, procurou-se verificar o tempo necessário para um destacamento ser convocado e ativado. Esse tempo inclui a requisição do destacamento, a sua seleção pelo QG, a criação e o posicionamento desse destacamento. Note-se que este tempo é uma estimativa básica, pois com o crescimento da base de destacamentos o tempo de seleção deve crescer consideravelmente.

No terceiro teste, foram aplicadas ao sistema massas de dados simulando os ataques para os quais foram desenvolvidos destacamentos: *port scan* (varredura de portas) e *syn flood* (inundação por pacotes TCP SYN). O objetivo desse teste é determinar se os agentes detectores são capazes de identificar e sinalizar os ataques.

6.1 Geração da massa de dados

As massas de dados dos testes foram geradas em um ambiente composto por uma máquina IBM PC/AT, com processador Pentium II 350 MHz, 128 MB de memória e placa de rede de 100 Mbps (Milhões de Bits Por Segundo), e com o S.O. Linux, funcionando como origem dos ataques. Também foram utilizados o utilitário *nmap* [8] e um programa gerador de *syn flood*.

O utilitário *nmap* é uma ferramenta de descoberta de vulnerabilidades por meio da execução de *port scan*. Ao ser acionada, ela gera um padrão de tráfego destinado à máquina que se deseja que seja analisada. O *nmap* pode realizar quatro tipos de *port*

scanning: *TCP Connect Scan* (abertura de conexão TCP), *TCP SYN Stealth Scan* (varredura furtiva por meio de envio de pacotes TCP SYN, também conhecida como *Half Open*, por não completar o protocolo de abertura de conexão TCP), *TCP FIN Stealth Scan* (varredura furtiva por envio de pacotes TCP FIN) e *UDP Scan* (varredura por meio de envio de pacotes UDP), bem como realizar *OS Fingerprinting*[9] (identificação do Sistema Operacional com base na forma como este implementa o protocolo TCP/IP). No apêndice D encontramos uma descrição de como funcionam esses tipos de varredura. O *nmap* também é capaz de operar em 6 taxas de geração de pacotes: *Insane* (envia pacotes até inundar a rede, sendo de fácil detecção), *agressive* (envia pacotes tão rápido quanto possível sem inundar a rede), *normal* (aguarda por respostas antes de prosseguir), *polite* (idem ao *normal*, porém evitando alguns padrões que causam pane em alguns SOs), *sneaky* (intervalo de 15 segundos entre cada pacote, mais difícil de detectar), e *paranoid* (intervalo de 5 minutos entre cada pacote, sendo dificílimo correlacionar cada pacote detectado a um único ataque). Para dificultar a detecção, é possível gerar falsas origens do tráfego (*address spoofing*).

O programa gerador de *syn flood* utilizado foi modificado a partir de um exemplo fornecido com a biblioteca Libnet, na versão Unix [19]. As modificações consistiram, basicamente, em melhorias de desempenho e de controle, passando a ser possível gerar pacotes até uma taxa aproximada de 10.000 pacotes/segundo (embora o programa aceite taxas maiores, de até 1.000.000 pacotes/segundo, o subsistema de rede não é capaz de produzir pacotes nessa taxa). A cada ativação do programa é gerada uma seqüência de rajadas de pacotes. O usuário do programa pode especificar o número de pacotes por rajada, o número de rajadas produzidas, e o intervalo entre rajadas. Cada pacote gerado consiste num pacote TCP SYN com endereço de origem, identificação IP, número de seqüência TCP e tamanho de janela TCP aleatórios.

As massas de dados geradas procuraram simular situações de *port scan* e *Syn Flood*, com as seguintes situações:

- *port scan*: fonte única, sem *address spoofing*, intervalo entre testes normal;
- *syn flood*: fonte única, rajada única de 1500 pacotes, 100 pacotes por segundo.

Em ambos os casos, as massas de dados foram geradas automaticamente no momento do seu uso, para garantir independência entre os resultados.

6.2 Configuração do ambiente de teste

O ambiente de teste utilizado foi composto de quatro máquinas IBM PC/AT com Windows NT 4.0 (Service Pack 6a), sendo duas delas com processador Pentium II 350 MHz e 128 MB de RAM, e as outras duas com processador AMD K6-II 500 e 64 MB de RAM. Das quatro máquinas, três contam com placas de rede de 100 Mbps e uma com placa de rede de 10 Mbps. O meio de rede utilizado é uma rede ethernet baseada em um Hub 10/100 Mbps.

6.3 Execução dos testes

O primeiro teste realizado foi a avaliação do tempo de configuração do sistema. Nesse teste procurou-se determinar o tempo de carga do sistema em diferentes configurações. Foram realizados testes com 1, 2, 3 e 4 máquinas. Os testes foram repetidos 10 vezes para cada configuração. Cada rodada de teste consiste em iniciar o sistema, a partir da janela do *aglets server*, sem nenhum outro agente ativo. O programa de carga foi modificado para isso, de forma a (1) marcar o momento de início da sua execução, considerado como o ponto de início da fase de configuração; e (2) aguardar por uma mensagem do agente Quartel General (que também foi modificado para isso) sinalizando o fim da configuração, marcar o intervalo de tempo decorrido entre as duas marcações, e imprimí-lo. O método usado para marcação do instante é `Date#getTime()`, que retorna a quantidade de milissegundos decorridos entre o tempo representado pelo objeto `Date` e a meia noite do dia 1/01/1970. Após a carga completa do sistema, o mesmo era totalmente descarregado, sem reinicializar o *aglets server* (para não interferir nas caches de classes, que como será visto mais adiante, influenciam fortemente os resultados). O resultado impresso foi anotado e compilado na Tabela 6.1. Da compilação dos resultados foram calculados a média e o desvio padrão dos valores.

O segundo teste realizado foi a avaliação do tempo de carga de um destacamento a partir da sua convocação, sem tráfego na rede. O teste foi realizado em duas situações: (a) carregando o destacamento no mesmo *host* do QG; e (b) carregando o destacamento em outro *host* que não o do destacamento. O teste foi repetido 10 vezes para cada modalidade. Para efeito dos testes, tomou-se o tempo de carga do agente especializado SentinelaNT. Cada rodada de teste consiste em provocar a carga do destacamento escolhido (por exemplo, recarregando o sistema); por isso, o agente Sentinela (geral) foi modificado, de forma a enviar uma mensagem ao módulo de carga, marcando o momento de

início do processo de carga do destacamento. Quando o destacamento solicitado se ativa no *host* de destino, o Quartel General sinaliza o agente Sentinela, como parte do processo normal de ativação de um destacamento. Ao receber essa sinalização, o agente sentinela envia uma nova mensagem ao módulo de carga, que marca o intervalo de tempo entre as duas mensagens, e o imprime. Usou-se aqui o mesmo método de marcação de tempo descrito para o teste 1. Os resultados impressos foram anotados e compilados na Tabela 6.2, onde se apresenta também a média e o desvio padrão dos resultados.

O terceiro teste, de detecção e reação, foi realizado com o sistema sendo executado em duas máquinas, sendo realizado em duas modalidades: ataque ao *host* do QG (que será chamado de primário), e ataque ao *host* secundário. Cada rodada era considerada bem sucedida se o ataque fosse sinalizado e o sistema adotasse alguma reação. Este teste também foi repetido 10 vezes, com massas de dados diferentes. Nenhuma medida de tempo foi tomada neste teste. As rodadas de teste consistiram em, com o protótipo pré-carregado no *aglets server*, executar os programas geradores da massa de dados. A avaliação do resultado consistiu na observação do comportamento do protótipo, sendo considerado sucesso quando pelo menos uma dessas condições se verifica:

(i) o protótipo sinaliza alguma mensagem relativa ao tipo de ataque desferido;
(ii) o protótipo sinaliza a ocorrência de uma alteração no nível de alerta; ou
(iii) o protótipo inicia a carga do destacamento específico para o ataque.

6.4 Resultados dos testes

Os resultados do teste 1 (tempo de configuração do sistema) estão na Tabela 6.1. Constata-se pelos resultados que o tempo de configuração tem pouca alteração com relação ao número de máquinas utilizadas no sistema, sendo de aproximadamente 200 mS por *host*. Uma outra constatação é de que os tempos apresentam grande variação, especialmente da primeira rodada para as subseqüentes. Isso acontece devido ao uso de *caches* para os códigos das classes. Com a adição de um *host* novo ao sistema, um *aglets server* recém inicializado passou a ser utilizado. Na primeira rodada, as *caches* de classe desse *aglets server* ainda não continham os códigos necessários, o que aumentou consideravelmente o tempo de configuração do sistema. Nas demais rodadas, o código já estava em *cache*, diminuindo sensivelmente o tempo de execução. A presença da *cache* de classes não afeta o sistema em nenhum outro aspecto. O resultado da primeira

execução, devido ao efeito da *cache*, foi descartado do cômputo da média e do desvio padrão.

	1 Máquina	2 Máquinas	3 Máquinas	4 Máquinas
1ª.	4,456	2,413	3,605	2,954
2ª.	1,783	1,773	1,993	2,063
3ª.	1,803	1,832	2,023	2,003
4ª.	1,532	1,552	2,173	2,223
5ª.	1,692	1,872	2,254	2,073
6ª.	1,623	1,692	2,543	2,353
7ª.	1,573	1,872	1,862	1,873
8ª.	1,593	1,823	2,183	2,193
9ª.	1,742	1,973	2,403	2,353
10ª.	1,642	1,902	2,093	2,203
Média	1,665	1,810	2,170	2,215
D.P	0,096	0,125	0,210	0,310

Tabela 6.1 - Teste 1: Tempo de Carga do Sistema (em Segundos)

Os resultados do teste 2 (tempo de carga de um destacamento) estão na Tabela 6.2. Constata-se aqui que o procedimento de carga de um destacamento no *host* local é muito rápido, chegando muito próximo à resolução máxima do temporizador utilizado (1 mS). Várias das medidas resultaram em 0 mS, o que indica que o tempo de carga foi menor do que 1 mS.

	Carga Local	Carga Remota
1ª.	10	511
2ª.	0	331
3ª.	10	461
4ª.	0	161
5ª.	10	60
6ª.	10	80
7ª.	0	41
8ª.	0	40
9ª.	10	40
10ª.	0	50
Média	5,000	177,500
D.P.	5,270	186,023

Tabela 6.2 - Teste 2: Carga de um destacamento (em milissegundos)

O efeito *cache* só foi percebido na carga em *host* remoto, embora tenha levado mais de uma rodada para estabilizar. Essa variação pode também resultar de consultas DNS, uma vez que o sistema de controle de acesso do *aglets server* exige a checagem do endereço IP da origem da mensagem contra um domínio. Essa checagem é feita por uma consulta de DNS reverso, que pode alterar o tempo da conexão conforme a carga do servidor de DNS utilizado.

Todas as rodadas do teste 3 foram bem sucedidas. Em todas as tentativas de *port scan*, tanto no *host* primário quanto no secundário, o protótipo sinalizou a ocorrência do evento. Um efeito do baixo desempenho da linguagem Java é que, apesar da detecção ter ocorrido, ela só se deu após um pequeno lapso de tempo. Como os pacotes capturados são enfileirados pelo subsistema de captura, e o seu processamento é seqüencial, ocorre esse atraso entre a ocorrência do evento e a sua detecção. Esse atraso, porém, não invalida o resultado. A mensuração desse atraso é uma tarefa muito complexa e que não traz nenhuma informação adicional, pois o tempo de atraso depende do número de pacotes trafegando na rede no momento da detecção, e da quantidade de pacotes que precisam ser processados para determinar com grau de certeza o ataque.

Nas rodadas de teste com ataque de *syn flood* também houve sucesso na detecção em todas as rodadas, tanto no *host* primário quanto no secundário. O mesmo efeito de atraso na detecção foi observado. Um efeito colateral digno de nota foi a geração, no módulo de detecção de *port scan*, de uma grande atividade. Isso ocorre porque os dois módulos trabalham com o mesmo tipo de informação, ou seja, pacotes TCP SYN. A diferença entre um *TCP SYN scan* e um *syn flood* é que no primeiro vários pacotes TCP SYN são recebidos, de uma ou mais origens, mas não há coincidência no número da porta TCP destinatária. Já no segundo, os pacotes TCP SYN recebidos são destinados em grande quantidade a uma única porta (ou a poucas portas). Outra diferença é que no primeiro caso as portas destino podem ou não estar em uso na máquina destino, enquanto no segundo caso a porta destino tem obrigatoriamente que estar ativa. Como as massas de dados, apesar de construídas do mesmo tipo de pacotes, são diferentes, não houve detecção errônea (falso positivo) de *port scan* durante a tentativa de *syn flood*.

6.5 Considerações finais do capítulo

Neste capítulo foram descritos os resultados dos testes realizados com o protótipo desenvolvido conforme a arquitetura descrita nos capítulos anteriores. Nestes testes, procurou-se verificar a correção e a efetividade do funcionamento do protótipo frente a tentativas de ataque.

Foram aplicados três grupos de testes, com o objetivo de testar, respectivamente, o tempo de configuração do sistema, o tempo de convocação de um destacamento, e a correção do sistema de detecção.

Por se tratar de um protótipo muito simplificado, não cabe comparar os resultados obtidos com outras ferramentas IDS, uma vez que estas cumprem funções muito mais complexas. Além disso, o fato do sistema ser baseado na linguagem Java implica num pesado *overhead* ao desempenho, sendo contraproducente compará-lo a outros sistemas construídos em linguagens mais voltadas para o desempenho.

Os resultados do primeiro teste mostraram uma grande influência do fator *cache* de classes no desempenho do sistema. Nas primeiras rodadas, devido ao código das classes ainda não estar em *cache*, houve um aumento significativo do tempo de execução, em todas as configurações.

Outra constatação do primeiro teste é que o tempo de configuração varia pouco em relação ao número de máquinas em uso no sistema. Isso ocorre pois o processo de configuração é composto por uma fase de preparação, que engloba a criação do Quartel General e do Auditor, e uma fase de distribuição, onde são criados e posicionados os sentinelas. Conclui-se que o tempo necessário para a primeira fase é significativamente maior do que o necessário para a segunda.

Quanto ao segundo teste, é interessante notar que o tempo de carga de um destacamento é razoavelmente pequeno, mesmo quando o seu destino é um *host* que não o mesmo do Quartel General. Esse tempo, porém, tende a aumentar à medida que a base de destacamentos cresça. Pode ser inclusive que o crescimento deste tempo determine a alteração dos mecanismos de busca, com a adoção de gerenciadores de bancos de dados e/ou inteligência artificial.

Quanto ao terceiro teste, ele provou a capacidade do protótipo em perceber as tentativas de ataque oriundas do subsistema de rede. Obviamente esse mecanismo pode ser bastante melhorado, em especial a detecção de *port scan*, que mesmo tendo identificado todas as tentativas, gerou atividade excessiva e desnecessária durante os ataques de *synflood*. Além disso, esse mesmo mecanismo não foi capaz de diferenciar um *TCP Scan* de um *TCP Syn Stealth Scan*. O mecanismo de detecção de *synflood* também pode ser melhorado, reduzindo a atividade gerada durante a rajada de ataque e concatenando alertas consecutivos.

De maneira geral, os testes foram satisfatórios e confirmaram a viabilidade do sistema. Vários pequenos erros de implementação foram encontrados e corrigidos durante o processo de preparação dos testes, aumentando a robustez da infra-estrutura básica do protótipo.

Capítulo 7 - Conclusões Finais

Foi apresentada neste trabalho a arquitetura de um sistema multi-agente, distribuído, baseado em agentes móveis, para a detecção e a reação a tentativas de intrusão e ataques. Essa proposta inova ao mesclar num único sistema de detecção de intrusos (IDS), características das duas vertentes principais deste tipo de ferramenta, que são os IDS baseados na máquina (HIDS) e os baseados na rede (NIDS). Graças ao uso de mobilidade, é possível realizar a configuração da distribuição dos agentes de maneira dinâmica.

A linguagem Java, escolhida para a implantação do protótipo, é bastante prática para o desenvolvimento dos agentes, permitindo uma codificação rápida com uma taxa significativamente alta de reutilização de código. Por sua característica de portabilidade irrestrita, é possível desenvolver agentes genéricos o suficiente para ser possível executá-los em máquinas das mais variadas arquiteturas e sistemas operacionais. Mas o seu uso impõe um considerável *overhead* à operação do sistema, tanto em termos de tempo de CPU quanto de ocupação de memória principal. O restrito conjunto de operações de rede possível aos programas não chegou a ser um empecilho, mas obrigou o uso de soluções alternativas (bibliotecas externas).

O desempenho das bibliotecas de captura e geração de pacotes de rede é resultado direto do uso misto de métodos java com bibliotecas nativas (externas). Esse desempenho, considerando as condições de observação, é bastante razoável, sendo que a rotina de captura consegue, nas máquinas usadas para teste, acompanhar o tráfego da rede de 100 Mbps, mesmo em modo promíscuo, sem perda de pacotes (apesar de apresentar atraso por enfileiramento). Através destas bibliotecas, é possível gerar e analisar tráfego de forma portátil e simples.

Uma característica interessante da proposta apresentada é que, apesar de ter sido concebida como um sistema de detecção de intrusos, ela pode ser facilmente adaptada para lidar com outros tipos de sistemas. Por exemplo, é possível imaginar sistemas de gerência e controle de sistemas, construídos de maneira análoga, usando sentinelas ou agentes equivalentes para identificar situações onde um tipo de reação automatizada é necessária, como por exemplo o controle de falhas em dispositivos. Nesse caso, agentes equivalentes aos destacamentos tomariam as atitudes necessárias.

O protótipo construído mostrou ser bastante efetivo. O fato de poder ser executado em quase todos os *hosts* pertencentes a uma rede faz com que sejam diminuídas as chances de que um atacante logre em conseguir uso não autorizado de qualquer recurso dentro do domínio de segurança, sem ser de alguma forma detectado. Assim, pode se dizer que a segurança do domínio será reforçada com o uso desta ferramenta.

Em termos de comparações com sistemas centralizados de segurança, não é possível confrontar os resultados de desempenho. O uso da linguagem Java no protótipo, apesar de trazer ganhos na facilidade de distribuição de processamento, gera uma perda de *performance*, já mencionada.

Como sugestões de prosseguimento dos trabalhos, deixamos, é claro, o desenvolvimento de novos agentes destacamento, para outros tipos de ataque além dos apresentados no capítulo 5. Além disso, há a possibilidade do desenvolvimento de ferramentas automatizadas de análise dos dados coletados, e o uso de inteligência computacional nos agentes. Também é possível extender a arquitetura para o uso de redes ativas, possibilitando a execução de agentes em equipamentos como roteadores.

Apêndice A - A Linguagem Java

A linguagem Java foi desenvolvida pela Sun Microsystems com o objetivo de criar uma linguagem orientada a objetos, dinâmica, para uso nos mesmos tipos de aplicações desenvolvidas em C e C++, mas sem as dificuldades e os erros mais comuns destas linguagens. A partir da experiência dos desenvolvedores, foram suprimidas algumas construções e acrescentadas outras da linguagem C++. O objetivo final era uma linguagem que facilitasse a geração de código robusto, confiável e facilmente utilizável em plataformas diversas. Java consegue essa independência de plataforma por meio da máquina virtual Java (JVM) e da especificação rígida da linguagem. Assim, novidades como *garbage collection* e *multi-lthreading* convivem com restrições como a supressão da aritmética de ponteiros e da conversão automática entre tipos, e um modelo de gerenciamento de memória próprio. Ao contrário de C++, Java segue estritamente o paradigma OO. Nem sempre é possível codificar totalmente um sistema em Java. Algumas vezes existem módulos ou programas compilados que precisam ser aproveitados. Outras vezes a codificação do algoritmo exige algum procedimento ou recurso não contemplado pela JVM. Para estes casos, a especificação da linguagem Java prevê a interface Java – Nativo. Essa interface define como um método escrito em linguagem nativa (não Java) pode ser chamado por métodos Java, ou vice-versa. O uso dessa facilidade, no entanto, traz alguns riscos, conforme será apresentado. Outra característica de Java é a existência de modelos de segurança, que definem políticas de acesso dos programas aos recursos da máquina hospedeira. Essas políticas de acesso são especialmente úteis quando se executa código Java desconhecido, proveniente de outras máquinas da rede.

A.1 Independência de Plataforma

Uma das mais marcantes características da linguagem Java é a *independência de plataforma*. Significa que um programa escrito em Java pode, teoricamente, ser executado sem alterações em qualquer ambiente operacional que implemente a JVM (*Java Virtual Machine*). Na prática, porém, esse objetivo é difícil de ser alcançado, pois existem dificuldades relacionadas a diferenças no ambiente operacional de base, além de pequenas falhas na implementação da JVM e/ou da biblioteca de classes básica em alguns dos ambientes operacionais, que fazem com que a execução dos programas tenha resultados diferentes nas diferentes plataformas utilizadas. Futuras versões da JVM e da

especificação da linguagem irão eliminando estes erros e eventualmente chegar-se-á a uma implementação da JVM que possibilite a verdadeira independência de plataforma. Por enquanto, para garantir essa independência, é necessário seguir uma série de procedimentos, que dão ao programa desenvolvido o rótulo ***100% Pure Java***.

A.2 A Máquina Virtual Java (JVM)

Para que fosse possível alcançar a independência de plataforma, decidiu-se que o compilador Java não geraria código para nenhum processador real específico. Ao invés disso, foi decidido que o compilador geraria código para um processador hipotético, ideal, que é o centro da ***Java Virtual Machine***. Esse código é conhecido como *bytecode* e reflete com coincidência quase total as estruturas da linguagem. A implementação da JVM consiste, portanto, de um programa interpretador do *bytecode*. Essa solução gera um grande *overhead* de processamento, que reduz em muito o desempenho das aplicações desenvolvidas em Java. Para diminuir esse *overhead* foram criados os *Just In Time Compilers* – JITs, que analizam o *bytecode* e o traduzem no código de máquina da arquitetura em uso. Apesar do ganho sensível, o desempenho final das aplicações Java ainda é bastante penalizado pela característica da linguagem de realizar uma estrita checagem das operações em *runtime*.

Esta última característica é fonte de polêmica entre os programadores. Como foi dito, Java realiza checagem estrita das operações em tempo de execução, o que gera um grande *overhead* de processamento. Por outro lado, a ausência dessa checagem é origem, em programas que ainda não tenham sido suficientemente depurados, de diversos erros, muitos deles de difícil solução. Em programas que já tenham sido considerados estáveis, porém, essa checagem representa apenas um peso extra.

Quando da identificação de um erro, a JVM responde com a geração de uma *exceção*. Essa exceção pode ser de dois tipos: ou a sua ocorrência pode ser prevista pelo programador, ou ela é resultado de uma situação imprevista. No primeiro caso, o programador avisa à JVM da possibilidade da geração da exceção com uma cláusula que compõe a *assinatura* do método, junto com a declaração dos parâmetros e dos valores de retorno desse método. Essas exceções são verificadas pelo próprio compilador, que exige que a chamada de métodos que se declarem geradores de exceções seja encapsulada numa construção especial. A documentação Java se refere a estas exceções como "***checked exceptions***".

Já as exceções imprevistas podem acontecer como resultado de erros na

codificação dos métodos ou então de situações fora do controle do programador, tais como falhas graves no sistema operacional hospedeiro ou intervenções do operador. Caso o programador identifique, em qualquer momento, uma probabilidade maior da geração de exceções imprevistas, ou então um trecho de código em que a geração de uma exceção imprevista possa gerar prejuízos para a disponibilidade do sistema, ele pode incluir no código intruções para tratá-las. Essas exceções são conhecidas como "***unchecked exceptions***".

Tanto no caso das *checked exceptions* quanto nas *unchecked exceptions*, a exceção é descrita por um objeto, sendo que atributos são usados para identificar o erro ocorrido e sugerir um procedimento de recuperação apropriado. O método que recebe a exceção pode escolher entre capturar a exceção (***catch***) ou repassá-la (***throws***). Esse modelo permite que o programador decida se que tratar a situação de erro localmente ao método ou deixar que o tratamento seja feito em um nível superior. Caso uma exceção não receba tratamento, o processamento da *thread* que a acusou é interrompido.

A.3 Multi-Threading

Existem duas formas de se trabalhar em ambientes multi-programados. Na primeira, um novo processo é criado para cada nova tarefa que precisa ser executada pelo *host*. Ao criar um novo processo, o sistema operacional aloca para ele uma nova área de código (***text***), uma nova área de dados (***data***) e uma nova área de trabalho e pilha de execução (***stack***). A cada vez que o processo é escalonado para execução, é necessária uma troca completa de contexto, que inclui o salvamento de todas as informações do processo anteriormente em execução nas estruturas de controle, e a troca dos controles de gerenciamento de memória, impondo um grande *overhead*, especialmente em ambientes com memória virtual.

Para minimizar esse *overhead* foi proposto um novo tipo de processo, mais leve, conhecido como ***lightweight process*** ou ***thread***. Uma ***thread*** representa um fluxo de processamento, mas é diretamente associada ao processo que a criou, compartilhando com ele pelo menos as áreas de código e dados. Há dois tipos de *thread*: as mantidas pelo sistema operacional (*kernel threads*) e as mantidas por bibliotecas de usuário (*user threads* ou *green threads*). Em ambos os casos, há vantagens sobre o uso de processos, tais como um menor *overhead* de chaveamento de contexto, mas o preço pago é alto: como existe acesso simultâneo a recursos da máquina, há a necessidade de controle do acesso e sincronização entre as *threads* que disputam os recursos.

A JVM faz uso intensivo de *multi-threading*, e por isso há primitivas de sincronização embutidas na linguagem. A princípio, qualquer objeto Java pode ser usado como elemento de sincronização. A especificação Java define o chamado ***monitor*** de acesso: uma vez que uma *thread* adquire o monitor, outras threads que tentem fazer o mesmo ficam com sua execução bloqueada até que o proprietário do monitor o libere total ou parcialmente. Nesse momento, uma das threads bloqueadas é escolhida e reativada, ganhando acesso ao monitor.

A.4 O Modelo Java de Gerenciamento de Memória

Um ponto fraco nas linguagens de alto nível que trabalham com referências dinâmicas (ponteiros) é o processo de alocação de memória. Uma referência não inicializada, ou mal inicializada, é fonte de múltiplos e variados erros para o programa. A alocação de novas regiões de memória é razoavelmente segura, porém a devolução dessas áreas ao sistema operacional normalmente é sensível, devido ao problema das referências externas. Se o programador decide por devolver manualmente as áreas alocadas, ele corre o risco de (i) esquecer de devolver uma ou mais áreas, gerando um consumo excessivo de memória; (ii) perder referência a uma área ainda alocada, impossibilitando a sua devolução; (iii) por manter múltiplas referências a uma área, voltar a acessar a área após a devolução; (iv) devolver mais de uma vez a mesma área. De qualquer forma, é consenso que a manipulação de áreas de memória de alocação dinâmica é um dos pontos mais complexos de qualquer linguagem de programação.

A solução da linguagem Java para a questão da alocação dinâmica foi separada em três partes. Em primeiro lugar, a alocação é feita por uma estrutura da própria linguagem, de forma controlada. Assim, o programador aloca objetos, não áreas de memória. Em segundo lugar, todas as variáveis de referência são inicializadas com o valor "*null*", que representa um acesso inválido. Para iniciar o uso da variável, o programa precisa inicializá-la explicitamente com o valor desejado. Por último, o programador Java não se preocupa em devolver áreas de memória; ele simplesmente invalida a referência para a área alocada. Para cada nova referência ao objeto, um contador é incrementado. Cada vez que uma dessas referências é destruída, o contador é decrementado. Quando esse contador chega a zero, o objeto torna-se elegível para *garbage collection*.

A.5 Controle de Acesso e Segurança

O controle de acesso em Java é feito em dois contextos: a nível de chamada de

métodos e a nível de operação de acesso a recursos. No primeiro nível, cada método ou atributo tem definida sua *visibilidade*, que pode ser *pública*, *restrita ao pacote*, *restrita à herança* ou *privada*. Na visibilidade pública, qualquer objeto pode, com seus métodos, acessar os componentes deste objeto (atributos e métodos). Na visibilidade restrita ao pacote apenas os objetos marcados como componentes do mesmo pacote podes acessar os componentes deste objeto. Na visibilidade restrita à herança, apenas objetos que forem instância de uma subclasse da classe deste objeto tem acesso aos componentes. E, finalmente, na visibilidade privada, apenas o próprio objeto tem acesso aos componentes. É prática comum definir todos os atributos do objeto como de visibilidade privada, enquanto os métodos são definidos como de visibilidade pública.

No nível de acesso aos recursos, Java define os chamados *modelos de segurança*. Um modelo de segurança é um conjunto de permissões que definem quais recursos podem ser acessados pelo programa Java. Esses recursos podem ser o sistema de arquivos do hospedeiro, a rede, estruturas de dados da JVM, ou mesmo outros programas Java em execução dentro da mesma JVM. O mais conhecido destes modelos de segurança é o utilizado pelos *applets* (pequenos aplicativos que são embutidos em páginas *web*), conhecido como *sandbox*. Este modelo, entre outras coisas, proíbe o acesso ao sistema de arquivos e à rede em geral (é permitido apenas contatar o *host* originário do *applet*).

A.6 A Interface Java – Nativo

A ***interface Java–Nativo*** (JNI) é um conjunto de regras que define como um objeto Java pode incluir e executar métodos desenvolvidos em linguagens nativas do ambiente hospedeiro, e vice-versa. Essa interface, de definição muito rigorosa, especifica com detalhe como os tipos básicos Java devem ser mapeados em tipos de dados nativos, ou como os parâmetros dos métodos Java são mapeados em parâmetros dos métodos nativos.

Uma característica que deve ser levada em conta no desenvolvimento de métodos nativos é o fato de Java ser uma linguagem eminentemente *multi-threaded*. Como não se está mais trabalhando dentro da JVM, não existem as ferramentas de sincronização disponibilizadas por esta. Além disso, cuidado redobrado deve ser tomado se métodos nativos de um mesmo objeto ou classe puderem ser acessados por *threads* diferentes.

Apêndice B - Principais Técnicas de Ataque

Como já foi dito, existe uma relação unívoca entre os mecanismos de proteção aos sistemas de computadores e os riscos a que estes são submetidos. Sendo assim, as técnicas usadas para desferir ataques são determinantes das características de operação das contra-medidas de segurança. Para melhor podermos entender quais devem ser as medidas de prevenção precisamos listar as principais formas utilizadas para atacar sistemas e comunicações.

A forma mais simples de invadir um sistema é conseguir um par conta/senha de acesso. Esses pares podem ser conseguidos, dentre outras formas, por meio de:

- **Roubo** – o par é simplesmente roubado do usuário original;
- **força bruta** – o invasor tenta todas as combinações possíveis de conta e senha até conseguir um par válido.

Em sistemas como o Unix, o uso de contas alheias é particularmente perigoso, especialmente se a conta tiver privilégios de superusuário. Há inclusive uma máxima de que, se um atacante experimentado consegue algum acesso à máquina, mais cedo ou mais tarde ele conseguirá acesso de super-usuário.

Nos parágrafos a seguir é apresentada uma lista com alguns dos principais tipos de técnicas de ataque.

B.1 Força Bruta

Um ataque de força bruta faz uso da principal característica do computador – a capacidade de executar a mesma operação milhões e milhões de vezes –, para forçar a entrada num sistema ou decodificar uma mensagem. A idéia é tentar todas as alternativas possíveis até o sucesso.

Bons sistemas de proteção devem ser capazes de resistir a ataques de força bruta, pelo menos por tempo suficiente para que, ou o atacante desista, ou o ataque seja descoberto, ou a informação atacada se torne irrelevante.

B.2 Engenharia Social

O ataque de engenharia social consiste em convencer a administração do sistema a conceder-lhe direitos de acesso a algum recurso mediante fraude. Apesar de ser uma das mais antigas e simples estratégias, é uma das mais eficientes.

B.3 Falsa Identificação e Negação de Autoria

Um usuário pode forçar outra máquina ou usuário a conceder-lhe acesso a recursos enviando uma identificação falsa. Ataques desse tipo são evitados confirmando a identificação através de algum tipo de teste (desafio-resposta, senha, etc).

Um usuário pode, por meio de falhas nos programas servidores ou usando máquinas subvertidas, enviar mensagens de correio eletrônico para outro como se fosse uma terceira pessoa. Para evitar isso, mensagens de importância devem ser autenticadas, ou seja, deve ser usado um mecanismo para comprovar a autoria da mensagem. Uma vez comprovada a identificação do usuário ou a autenticidade da mensagem, fica afastada também a possibilidade de negação de autoria. Negação de autoria ou repúdio ocorre quando o autor legítimo de uma operação ou mensagem nega a sua autoria, ou a atribui a outra pessoa.

B.4 Exploração de Acessos Ocultos (*BackDoors*)

Um *backdoor* (porta dos fundos) é um mecanismo pelo qual uma pessoa pode acessar um sistema sem passar pelos mecanismos normais de identificação. Os *backdoors* são a grande dor de cabeça dos administradores de sistema, uma vez que eles tanto podem ser implantados pelos invasores, como podem vir no próprio Sistema Operacional ou nos programas aplicativos.

Dois exemplos de *backdoors* são, por coincidência, programas de controle de correio eletrônico. Um é o **SendMail,** padrão do sistema Unix. Esse programa ficou conhecido por conter numerosas opções não documentadas, algumas delas permitindo que o invasor forçasse o sistema a executar código malicioso. O outro é o **Microsoft Outlook**, que, apesar de desenvolvido muito tempo depois, não aproveitou a experiência e repetiu os mesmos erros.

B.5 Personificação de Endereço (*Address Spoofing*)

A personificação consiste em gerar pacotes cujo endereço IP de origem não corresponde ao endereço correto da máquina onde ele foi gerado. Com isso, torna-se possível à máquina atacante passar por outra, a qual está autorizada ao uso de um certo recurso. Além disso, é mais difícil determinar a origem dos pacotes maliciosos. Devido à possibilidade do uso desta técnica, não é seguro revidar a um ataque baseado apenas no endereço de origem, já que ele pode ser falsificado.

Uma característica do uso de *address spoofing* é que os ataques que usam essa técnica normalmente são cegos, ou seja, não recebem nenhum *feedback* do sistema atacado durante o procedimento. Isso acontece pois o tráfego de retorno é roteado para a localização verdadeira do endereço usado. Por isso é comum este tipo de técnica aparecer combinada com outras, como *sniffing* (escuta), bloqueio por *flooding* (inundação, sobrecarga) e *source routing* (alteração de rota).

B.6 Captura de Conexões (*Hijacking*)

O ataque de captura de conexões consiste em fazer com que a máquina alvo abra uma conexão com a máquina atacante acreditando que se trata de outra máquina. A máquina alvo pode também ser forçada a aceitar a inserção de pacotes falsificados pelo atacante em uma conexão já aberta. O ataque de *hijacking* é dividido, a grosso modo, em três fases, que veremos a seguir.

Fase 1: O atacante identifica que duas máquinas tem uma relação de confiança mútua. Nesta fase ele realiza um *port scanning*, encontrando portas abertas e determinando o sistema operacional dos alvos[11].

Fase 2: O atacante inunda uma das duas com mensagens, de forma a fazer com que ela fique com suas filas de serviço lotadas, não podendo responder a solicitações legítimas;

Fase 3: O atacante espera que a outra máquina envie uma mensagem para a máquina bloqueada e responde como se fosse ela. Neste ponto, diz-se que a comunicação entre elas foi capturada.

Serviços TCP/IP, que mantém "conexões" abertas por longo tempo, tais como NFS (Sistemas de Arquivos Remoto) e TELNET (Terminal Virtual), são particularmente vulneráveis a este tipo de ataque.

B.7 Bloqueio ou Privação de Serviço (DoS)

A privação de serviço (DoS –*Denial of Services*) costuma ser usada para evitar

---

[11] A identificação do sistema operacional das máquinas alvo facilita a escolha de técnicas específicas de ataque. Convém impedir que essa identificação seja fornecida pelo *hostname* ou por *banners* de servidores de *telnet* ou *ftp*. Porém, uma possibilidade aterradora é o uso do *fingerprinting* (impressão digital), que usa características da implementação dos protocolos TCP/IP para determinar, com precisão espantosa, o ambiente operacional em uso na máquina alvo. Essa técnica é capaz de determinar até o nível de *patch* aplicado ao sistema, em alguns casos.

que o sistema cumpra sua função, ou então por vandalismo. Ela consiste basicamente em enviar mensagens tais que o receptor entre em falha ou bloqueio.

A primeira forma de bloquear um sistema já foi descrita acima: basta inundá-lo de mensagens. Outra forma é enviar uma mensagem grande demais para ser guardada em memória. Um ataque que ganhou o nome de "*Ping Of Death*" forçou a atualização de vários sistemas na Internet; nele, o atacante enviava uma mensagem ICMP Ping com tamanho de 65 Kbytes ou mais a um destino. Como alguns sistemas operacionais não reservavam espaço suficiente para armazenar a mensagem completa, ocorria estouro de *buffer*[12], provocando o travamento da máquina alvo e forçando o administrador a reinicializar o sistema.

Outro alvo possível de ataque de bloqueio são as comunicações entre as máquinas da rede. Esses ataques podem ser realizados de diversas maneiras:

- destruindo as mensagens que passam por um certo ponto;
- alterando as tabelas de roteamento de forma a impedir a entrega das mensagens;
- bloqueando o sistema emissor, o sistema receptor, ou ambos.

B.8 Alteração, Inserção ou Remoção de Mensagens

Um atacante pode alterar de várias formas uma mensagem válida, mesmo que autenticada. Ele pode suprimir alguns trechos, incluir outros, ou mudar a ordem em que eles apareçam no texto. Para evitar isso, o receptor deve ter mecanismos para verificar a integridade da mensagem. Outra prática comum é a retransmissão de mensagens válidas. Para garantir uma segurança efetiva, o receptor deve poder distinguí-las das mensagens verdadeiras.

B.9 Criptoanálise

O processo de descoberta de formas para decodificar mensagens criptografadas sem as chaves apropriadas é conhecido como criptoanálise. A teoria sobre criptografia e criptoanálise compõe uma linha de pesquisa muito extensa e que recebe muitos investimentos, envolvendo modelos matemáticos altamente complexos [30].

---

[12] Estouro de *buffer* é uma das vulnerabilidades mais comuns; existem vários exemplos de *exploits* que utilizam essa técnica. Sua origem está na má codificação das rotinas de entrada de dados.

Apêndice C - Exemplos de Sistemas de Agentes Móveis

Diversos sistemas foram desenvolvidos, ou estão em desenvolvimento, para a construção de sistemas baseados em agentes (alguns, inclusive, disponíveis para acesso público na internet). A linguagem de desenvolvimento utilizada é muito variada, indo desde o uso de linguagens proprietárias (como no caso do ***Telescript***), passando por C++ e Tk/TCL e terminando em Java.

Duas características comuns dos sistemas de agentes móveis baseados em Java são que todos eles usam a Máquina Virtual Java (JVM) padrão e o mecanismo de serialização de agentes oferecido pela própria linguagem.

C.1 Telescript

O ***Telescript*** [11] é o primeiro e mais conhecido sistema comercial de agentes móveis. Esse ambiente executa e gerencia aplicações baseadas em agentes em equipamentos servidores. Ele é baseado em uma linguagem proprietária, o que restringiu muito sua aceitação.

C.2 Odissey

O Sistema ***Odissey*** [10] foi resultado de um redesenho do ***Telescript*** levando em conta a popularidade da Internet e o sucesso da linguagem Java. Assim, ***Odissey*** implementa todas as funcionalidades do ***Telescript*** utilizando classes Java, o que resulta em uma biblioteca de classes que facilita o desenvolvimento de aplicações de agentes móveis.

C.3 Concordia

A plataforma ***Concordia*** [16] foi criada tendo em mente o desenvolvimento e gerenciamento de aplicações de agentes móveis. É composta de múltiplos componentes, todos escritos em Java, que ao serem combinados fornecem um ambiente completo para aplicações distribuídas. Este sistema é composto por uma JVM padrão, um servidor e um conjunto de agentes. Este sistema provê mecanismos de segurança para a execução de agentes, bem como a definição de *checkpoints* em suporte a tolerância a falhas.

C.4 Voyager

O ambiente ***Voyager*** [17] é uma plataforma voltada para a construção de sistemas distribuídos baseados em agentes. Ele disponibiliza mecanismos tradicionais de troca de mensagens e primitivas de mobilidade de agentes. Pode-se dizer que ***Voyager*** combina as propriedades de um *Object Request Broker* (ORB) baseado em Java com as de um sistema de agentes móveis. Assim, é possivel que programadores Java criem aplicações de rede utilizando ambas as técnicas.

C.5 Agent Tcl

O ambiente ***Agent Tcl*** [12] é um sistema de agentes móveis que provê mecanismos de comunicação e de segurança, ferramentas de depuração e auditoria. O principal componente deste sistema é um servidor que é executado em cada máquina do domínio, permitindo a migração do estado de execução completo, incluindo variáveis locais e ponteiro de instrução. Quando um agente deseja migrar de um *host* para outro, o servidor da origem executa a função *agent_jump()* que captura o estado completo do agente e envia para o servidor em execução no *host* de destino. Este servidor então inicia um novo processo Tcl, inicializando-o com a informação de estado recebida. Com isso, o agente é reiniciado no mesmo ponto em que estava ao deixar o contexto de origem.

C.6 Ara

O ***Ara*** [20] é outra plataforma baseada em Tcl que permite a execução segura de agentes móveis em redes heterogêneas. Este projeto está mais voltado, porém, ao suporte da execução segura e portável de sistemas de agentes móveis do que para as características de nível de aplicação, tais como padrões de cooperação entre agentes, comportamento inteligente e modelagem de usuários.

C.7 TACOMA

O sistema ***TACOMA*** [13] foi desenvolvido em linguagem C e é baseado em ambientes Unix e no uso do protocolo TCP. Ele suporta agentes escritos em C, Tk/Tcl, Perl, Python e Scheme. Essa plataforma oferece suporte ao desenvolvimento de sistemas de agentes móveis que podem ser usados na resolução de problemas tradicionalmente endereçados aos sistemas operacionais.

C.8 ASDK

O ***ASDK*** (*Aglets Software Development Kit*) [18] foi escolhido para o desenvolvimento deste trabalho por ser uma plataforma de uso simples, liberada para uso sem necessidade de licenciamento. Atualmente está disponível para *download* na Internet em duas versões (1.0.3, estável, e 1.1B3, de desenvolvimento) e recentemente seu código fonte foi disponibilizado, tornando-a *OpenSource*[13] (código aberto)

O ambiente ASDK foi desenvolvido em Java, e é constituído de um servidor de mobilidade com interface gráfica e bibliotecas de classes para o desenvolvimento de agentes. O modelo ASDK [14] é baseado no protocolo ATP (*Agents Transfer Protocol*), que é uma extensão do protocolo HTTP criando as primitivas básicas de mobilidade: criação, movimentação, suspensão, reativação, comunicação e desativação.

Uma limitação do ASDK (e de todos os ambientes baseados em Java) é que ele não possibilita o aproveitamento do estado completo do agente, uma vez que a definição da JVM padrão não permite o acesso ao ponteiro de execução nem aos *registros de ativação* das funções em execução.

[13] http://www.opensource.org

Apêndice D - Técnicas de Port Scanning

Neste apêndice serão vistas algumas técnicas de *port scanning* utilizadas. *Port Scanning* é o processo de identificar as portas TCP e UDP abertas em uma máquina para, assim, identificar os serviços oferecidos por ela. Com base nessa informação, o agressor pode planejar um ataque.

D.1 Ping Scan

O *Ping Scan* não é propriamente um *scan* de serviços, mas de presença. Consiste em enviar um pacote ICMP ECHO para o alvo, e aguardar pelo retorno. Caso o pacote ICMP ECHO_REPLY chegue dentro do tempo predeterminado, assume-se que o alvo está ativo e é alcançável.

Seqüência Enviada

a -> **b** ICMP ECHO

Respostas Possíveis

b -> **a** ICMP ECHO REPLY

- Host b está ativo e é alcançável

X -> **a** ICMP DEST UNREACH: HOST UNREACH (**X** roteador intermediário)

X -> **a** ICMP DEST UNREACH: NET UNREACH

- Host **b** não é alcançável.

(Sem resposta – timeout)

- Host **b** não está ativo (ou não é alcançável devido a congestionamento na rede)

D.2 TCP Connect Scan

O *TCP Connect Scan* consiste em tentar abrir conexões para cada uma das portas da máquina alvo. Caso a conexão seja aberta, ela é imediatamente fechada. Pode ser feita de duas maneiras: por meio de conexões normais (usando o *kernel*) ou por meio de bibliotecas externas.

Como a conexão é totalmente aberta, é passível de registro pelo *kernel*.

D.3 TCP SYN Stealth Scan

O *TCP SYN Stealth Scan* consiste em enviar pacotes TCP SYN para cada porta da máquina alvo e em seguida enviar um pacote TCP RST para essa mesma porta. Como a conexão não chega a ser aberta (pois o *3-Way Handshake* não se completa), o *kernel* da máquina alvo não a registra. Esta técnica exige o uso de bibliotecas especiais de usuário, pois o kernel não permite desrespeitar o *3-Way Handshake*. O usuário que executa este tipo de *port scan* precisa ter privilégios de supervisão junto ao SO da máquina origem.

Seqüência Enviada

a.p -> **b**.q TCP SYN (x,0)

Resposta Esperada:

b.q -> **a**.p TCP SYN,ACK (y,x+1) [2ª. Fase do *3-way handshake*]

- Conclui-se que a porta q está ativa no host **b**. Como o *kernel* da máquina a não enviou o TCP SYN, ele não reconhece o SYN,ACK e responde com um TCP RST.

D.4 UDP Scan

O *UDP Scan* consiste em enviar para cada uma das portas UDP da máquina alvo um pacote UDP. Pela definição da implementação do IPv4, as portas que não estejam ativas devem responder com uma mensagem ICMP DEST UNREACH/PORT UNREACH.

D.5 TCP Fin Stealth Scan

O *TCP Fin Stealth Scan* consiste em enviar um pacote TCP FIN para cada uma das portas da máquina alvo. Portas que estejam abertas (ativas) não irão responder, ao passo que portas fechadas (inativas) irão responder com TCP FIN. Como não há abertura de conexões, o *kernel* não registra essa atividade. O usuário precisa ter privilégios de supervisão junto ao SO da máquina origem para executar este tipo de *port scan*.

Seqüência Enviada

a.p -> **b**.q TCP FIN (0, 0)

Resposta Esperada

b.q -> **a**.p TCP FIN (0, 1) [se **a** porta q não estiver ativa no host **b**]

D.6 OS Scan

O *OS Scan*, ou *Finger Printing* (impressão digital), não é propriamente um *scan* de serviços, mas sim de identificação do Sistema Operacional utilizado na máquina alvo. Consiste em enviar pacotes com características patológicas à máquina alvo e esperar a resposta. De acordo com a resposta, pequenas idiossincrasias de implementação podem ser identificadas, e a partir de uma tabela de respostas, denunciar o sistema operacional em uso na máquina alvo.

Por exemplo, uma máquina Windows NT 4 usa, nos pacotes IP, o campo TTL sempre igual a 128, enquanto máquinas FreeBSD usam esse campo igual a 64 para pacotes TCP e 255 para pacotes ICMP. A busca pode chegar ao refinamento de identificar versões e *Service Packs* aplicados ao SO.

A seqüência de pacotes enviada é muito complexa, compondo um padrão de testes. Mais informações sobre *Finger Printing* podem ser encontradas em [9].

Referências

[1] Anzen Computing, *fragrouter*. Disponível para *download* a partir de http://www.anzen.com/ndisbench/

[2] Anzen Computing, *tcpreplay*. Disponível para *download* a partir de http://www.anzen.com/ndisbench/

[3] Crosbie, M., E.H. Spafford. *Defending a Computer System Using Autonomous Agents*. COAST Technical Report 95/22. COAST Laboratory – Purdue University. Março/1994.

[4] Debar, H., M.Y. Huang, D.J. Donahoo. *Intrusion Detection Exchange Format Data Model*. IETF, 2000. http://www.ietf.org/internet-drafts/draft-ietf-idwg-data-model-03.txt

[5] eEye: Digital Security, *LibnetNT*. Disponível para *download* em http://www.eeye.com/html/tools/LibnetNT/LibnetNT.zip

[6] Endler, Markus. *Novos Paradigmas de Interação usando Agentes Móveis*. IME/USP. 1998.

[7] Freier, A.O., P. Karlton, P.C. Kocher, *The SSL Protocol - Version 3.0*. Internet Draft, Netscape Communications, Março/1996. http://home.netscape.com/eng/ssl3/ssl-toc.html.

[8] Fyodor (pseudônimo). *Nmap -- Stealth Port Scanner For Network Security Auditing, General Internet Exploration & Hacking*. http://www.insecure.org/nmap/.

[9] Fyodor (pseudônimo). *Remote OS Detection via TCP/IP Stack Fingerprinting*. Outubro/1998. http://www.insecure.org/nmap/nmap-fingerprinting-article.html.

[10] General Magic. *Odissey*. Disponível para *download* a partir de http://www.genmagic.com/technology/odissey.html

[11] General Magic. *Telescript*. Disponível para *download* a partir de http://www.genmagic.com/telescript/index.html

[12] Gray, R.S., *AgentTCL: A transportable agent system*, Proc. 4th Intl. Conference on Information and Knowledge Management (CIKM'95), 1995. http://agent.cs.dartmouth.edu/papers/gray:agenttcl.pdf.

[13] Johansen, D., R. van Renesse, e F.B. Schneider. *An Introduction to the TACOMA Distributed System Version 1.0*. Technical Report 95-23, Department of Computer Science, University of Tromsø, Noruega, Junho/1995. http://www.cs.uit.no/forskning/rapporter/Reports/9523.html

[14] Lange, D.B., M. Oshima, *Programming and Deploying Java Mobile Agents With Aglets*, Addison-Wesley Publ. Co., Agosto de 1998.

[15] Lawrence Berkeley National Laboratory, *tcpdump*. Disponível para *download* em ftp://ftp.ee.lbl.gov/libpcap.tar.Z.

[16] Mitsubishi Eletric ITA. *Concordia*. Disponível para *download* a partir de http://www.meita.com/HSL/Projects/Concordia/

[17] Objectspace. *ObjectSpace Voyager Core Technology. 2.0 – User Guide.* Objectspace Inc., 1998. Disponível para *download* a partir de http://www.objectspace.com/voyager/index.html.

[18] Oshima, M., G. Karjoth. *Aglets Specification Version 1.0 (Draft)*. IBM, Abril de 1998. http://www.trl.ibm.com/documents.html.

[19] PacketFactory, *LibNet*. Disponível para *download* a partir de http://www.packetfactory.net/libnet/

[20] Peine, H. *An Introduction to Mobile Agent Programming and the Ara System*. ZRI report 1/97, Dept. of Computer Science, University of Kaiserslautern, Germany. http://wwwagss.informatik.uni-kl.de/Projekte/Ara/Doc/intro-prog.ps.gz.

[21] Politecnico di Torino, *WinPCap*. Disponível para *download* a partir de http://netgroup-serv.polito.it/winpcap/

[22] Ptacek, T. H, e T. N. Newsham, *Insertion, Evasion and Denial of Service: Eluding Network Intrusion Detection*, Secure Networks, 1988.

[23] Queiroz, J.D., L. Pirmez e L.F.R.C. Carmo. *Micæl: An Autonomous Mobile Agent System To Protect Networked Applications Of New Generation*. In memorials of Workshop RAID - Recent Advances in Intrusion Detection - Purdue University. Set/1999.

[24] Queiroz, J.D., L. Pirmez e L.F.R.C. Carmo. *Micæl: Um Sistema para Detecção de Intrusos com o uso de Agentes Móveis Autônomos*. Relatório Técnico do Núcleo de Computação Eletrônica da UFRJ. Julho/1999.

[25] Queiroz, J.D., L. Pirmez e L.F.R.C. Carmo. *Projeto e Desenvolvimento de um Sistema Baseado em Agentes Móveis Autônomos para Detecção de Intrusos*. In Memorias do XXV CLEI - Conferência Latino-Americana de Educação e Informática, pp. 1121 a 1132. Set/1999.

[26] Rose, M.T., *The Simple Book – An Introduction to Internet Management*. 2nd Ed. Prentice-Hall Intl. 1994.

[27] Soares, L.F, G.L. Souza e S. Colcher. *Redes de Computadores: Das LANs, MANs e WANs às Redes ATM*. Rio de Janeiro, Ed. Campus, 1997.

[28] Stallings, W., *SNMP, SNMPv2 and RMON – Pratical Network Management*. 2nd Ed. Addison-Wesley Publ. 1996.

[29] Staniford-Chen, S. *et al. GrIDS – A Graph Based Intrusion Detection System for Large Networks*. UC/Davis, 1996. http://olympus.cs.ucdavis.edu/arpa/grids/nissc96.ps.

[30] Tanenbaum, A. S., *Redes de Computadores*. Tradução da 3ª Edição Americana. Ed. Campus, 1997.

[31] Vários. *Detecting Malicious Insider Misuse (panel)*. Panel Chair: Richard Brackney (NSA), USA. **Participants**: Roy Maxion (CMU), Eugene Spafford (Purdue University), Mark Wood (ISS), Yves Deswarte (LAAS-CNRS & Microsoft Research). In: Proceedings of the 1999 RAID Workshop, http://www.cerias.purdue.edu/raid/raid1999.html

[32] Waldbusser, S., *Remote Network Monitoring Management Information Base*, RFC 1757, Fevereiro de 1995.

[33] Zamboni, D., J.S. Balasubramaniyan, J.O. Garcia-Fernandez, D. Isacoff, E.H. Spafford. *An Architecture for Intrusion Detection using Autonomous Agents*. COAST Technical Report. 98/05. COAST Laboratory – Purdue University. Junho/1998.