Capítulo 1

Uma Proposta de Extensão da UMLintr para Especificação de Linguagens de Ataque

Emerson Alberto Marconato, Valter Vieira de Camargo, Luciana Andréia Fondazzi Martimiano, Kalinka R. L. J. Castelo Branco

Abstract

Attacks are frequently executed against hosts and computer networks, such as: virus, worms, trojan horses, port scanning, DoS (Denial of Service), DDoS (Distributed Denial of Service), and spoofing DNS (Domain Name System). An effective way to protect from these attacks is to use IDS (Intrusion Detection System). Several researches have been carried out aiming to improve the protection of the host connected to the Internet. Some of the IDS tools work with attacks signatures to detect them. However, to model attacks scenarios that generate these signatures is not easy to the IDS tools developers. This chapter proposes to use UML (Unified Modeling Language) and AOP (Aspect-Oriented Programming) to model scenarios that can generate attacks signatures to be used by IDS tools, using the benefits of better structuring, maintenance and legibility. As security is a transversal subject, it is suitable to be modeled using Aspects.

Resumo

Ataques são freqüentemente executados a hosts e redes de computadores, sendo alguns destes: vírus, worms, cavalos de tróia, port scanning - varredura de portas, DoS (Denial of Service) – negação de serviço, DDoS (Distributed Denial of Service) – negação de serviço distribuída, spoofing DNS (Domain Name System). Uma das formas efetivas de defesa contra os ataques é a utilização de IDS (Intrusion Detection System) – Sistema de Detecção de Intrusão. Várias pesquisas têm sido feitas para aprimorar as ferramentas IDS, visando sempre melhorar a proteção dos hosts ligados à Internet. Algumas das ferramentas IDS trabalham com assinaturas de ataques a fim de detectá-los. No entanto, a modelagem de cenários de ataques que gerem assinaturas não é simples para os desenvolvedores. Este capítulo propõe a utilização de UML (Unified Modeling Language) e de POA (Programação Orientada a Aspectos) na modelagem de cenários que gerem assinaturas de ataques a serem utilizadas pelas ferramentas IDS

com os benefícios de melhor estruturação, manutenção e legibilidade. O tema segurança é um interesse transversal, de modo que sua modelagem fazendo uso de Aspectos é adequada.

1.1 Introdução

Atualmente, muitas organizações conectadas à Internet tiveram um considerável aumento no orçamento para tratar de forma mais eficaz e eficiente a segurança de sua rede (Xiang, Zhou e Li, 2006). Para se proteger de atividades maliciosas, as organizações necessitam monitorar as brechas de segurança (Raihan e Zulkernine, 2005). Dentre as ferramentas mais utilizadas para prover essa segurança estão os Sistemas de Detecção de Intrusão (*Intrusion Detection System - IDS*) –. Várias pesquisas têm sido realizadas para aprimorar essas ferramentas, tais como desenvolvimento de linguagens e formas de modelagem, visando sempre aumentar a proteção dos *hosts* ligados à *Internet* (Raihan e Zulkernine, 2005). Esses esforços são de grande contribuição ao meio acadêmico e comercial, porém, é necessário testar a eficiência e eficácia dessas ferramentas, sendo que vários são os trabalhos que abordam esta visão da segurança de redes, dentre esses estão os trabalhos de Raihan e Zulkernine (2005) e de Hussein e Zulkernine (2006).

Segundo Hussein e Zulkernine (2006), a modelagem de *software* e a especificação de seus requisitos de segurança, como a especificação de cenários de ataques, são problemáticas para os desenvolvedores em conseqüência da necessidade de linguagens específicas para ambas as atividades. Nesse sentido, pode-se fazer presente a programação paradigma orientada a aspectos (POA), proposto por Kiczales (1997), visa a modularizar mais adequadamente um sistema de software. O objetivo é isolar em módulos distintos os interesses transversais de um sistema dos interesses funcionais, resultando em melhores níveis de manutenção e reúso.. Dentro da terminologia da POA um "interesse transversal" é um interesse cuja implementação torna-se espalhada e entrelaçada com os outros módulos funcionais do sistema. Um exemplo clássico de interesse transversal é Segurança. A UML (*Unified Modeling Language*), estendida para UMLintr (*UML for intrusion specifications*) (Hussein e Zulkernine, 2006), auxilia na modelagem de cenários de ataque, no entanto, essa linguagem ainda necessita ser estendida para contemplar outros tipos de ataques conhecidos. Dessa forma, a união da UMLintr com POA (Programação Orientada a Aspectos) alia os benefícios tanto de uma quanto de outra, uma vez que segurança é um interesse transversal adequado para ser implementado com Aspectos.

Um estudo que merece atenção, nesse contexto, é prover facilidade na escrita de cenários de ataques que gerem assinaturas para serem utilizadas em ferramentas IDS (Raihan e Zulkernine, 2005; Hussein e Zulkernine, 2006; Wang, Zhi-tang e Yao, 2006), e uma das formas de abordar este assunto é utilizar POA e UMLintr para especificar esses cenários.

Este capítulo está organizado da seguinte maneira: a Seção 1.2 descreve os principais conceitos relacionados a um IDS; a Seção 1.3 apresenta os principais tipos de ataques às redes de computadores; a Seção 1.4 apresenta as linguagens de especificação UML e UMLintr; a Seção 1.5 descreve os conceitos básicos de programação orientada a Aspectos; a Seção 1.6 apresenta a proposta de utilização da UMLintr e conceitos de

POA para exemplificar um ataque *Smurf*; e por fim, a Seção 1.7 apresenta as considerações finais.

1.2 Sistema de Detecção de Intrusão

Segundo Balasubramaniyan *et al.* (1998), detecção de intrusão pode ser definida como sendo o ato de revelar qualquer conjunto de ações que comprometa a integridade, confidencialidade ou a disponibilidade do sistema, identificando agentes que estejam utilizando um sistema computacional sem autorização, e identificando pessoas que têm acesso legítimo ao sistema, mas estão abusando de seus privilégios.

Sistemas de detecção de intrusão constituem aplicativos que monitoram a rede e são configurados para detectar qualquer comportamento que não seja considerado normal, sendo capaz de evitar um ataque, ou uma intrusão. Como exemplo típico, tem-se o *buffer overflow*, que é caracterizado pela ação do invasor em sobrescrever dados em um *buffer* não testado de um programa com seus próprios dados maliciosos. Dependendo de quais dados são sobrescritos, o programa pode parar de funcionar ou pode mudar sua execução fazendo o que o invasor deseja. À medida que os IDS fornecem meios de inferir sobre o conteúdo das conexões permitidas e detectar as que apresentem um comportamento suspeito ou não condizente com a política de segurança, eles tornam-se importantes (Ambrósio, 2002).

As principais características de um IDS segundo Balasubramaniyan *et al.* (1998) são:

- Executar continuamente sem interação humana;
- Ser seguro o suficiente de forma a permitir sua operação em *background*, entretanto não deve constituir uma caixa preta;
- Ser tolerante a falhas de forma a não ser afetada por uma queda do sistema (acidental ou por ações humanas);
- Resistir às tentativas de subversão (mudança), ou seja, deve monitorar a si próprio de forma a garantir sua segurança;
- Ter o mínimo de impacto no funcionamento do sistema, ou seja, gerar uma sobrecarga mínima no sistema computacional que está sendo executado;
- Poder detectar mudanças no funcionamento normal;
- Permitir fácil configuração: cada sistema possui padrões diferentes o IDS deve ser adaptado de forma fácil aos diversos padrões (de acordo com a política de segurança);
- Adaptar-se às mudanças no sistema e ao comportamento dos usuários;
- Ser escalável para dar suporte à carga de monitorar várias estações e ainda produzir resultados confiáveis em tempo hábil para a tomada de decisões;
- Ser difícil de ser enganado.

Muitos IDS realizam suas operações a partir da análise de padrões do sistema operacional e da rede (utilização de CPU, I/O de disco, uso da memória, número de tentativas de *login*, número de conexões, entre outros). Estes dados formam uma base de

informação sobre a utilização do sistema no tempo, ou permitem geração de bases com padrões de ataque previamente montadas. A partir dessas informações, o IDS pode identificar as tentativas de intrusão e até mesmo registrar a técnica utilizada.

Existem diferentes classificações dos IDS. No entanto, as principais são feitas em termos de:

1. **Como o sistema aborda o problema de detectar intrusão**

   - **Baseado em Rede**: Coletam os pacotes que trafegam na rede monitorada. Normalmente capturam dados nos pontos de estrangulamento da rede (roteadores ou *firewalls*), seja executando diretamente nesses elementos ou através de cópia de dados utilizando dispositivos de rede como *hubs* e *switches*. Examinam os dados que trafegam pela rede por meio de monitoração *on-line* dos pacotes, procurando por indícios de um ataque que possa estar acontecendo. Essa monitoração é realizada por meio de sensores espalhados pela rede que capturam todos os pacotes endereçados ao segmento de rede nos quais se encontram. Geralmente, esses sensores são dispositivos passivos que têm pouco impacto no desempenho da rede. Por outro lado, esses sistemas têm dificuldade em processar pacotes em uma rede grande de tráfego intenso ou isolada por *switch* e, não são capazes de analisar pacotes criptografados;
   - **Baseado em Nós da Rede**: Esses sistemas constituem um tipo híbrido de IDS que conseguem superar algumas das limitações dos IDS baseados em rede. Assim como os IDS baseados em rede, esses sistemas também capturam pacotes à procura de indícios de ataques. No entanto, os sensores, ou micro-sensores, só se preocupam com os pacotes endereçados ao nó de rede no qual se encontram. Uma vez que esses sensores não capturam todos os pacotes da rede, eles podem ser mais rápidos e consumir menos recursos, causando um baixo *overhead* no sistema. Além disso, são adequados para segmentos de tráfego intenso ou isolados por *switch*;
   - **Baseados em *Host***: Utilizam informações coletadas nos sistemas operacionais tais como registros de auditoria (*logs*), saída de programas de monitoração e informações de estado do sistema operacional, permitindo uma análise das atividades com mais confiabilidade e precisão. Essa análise determina quais processos e usuários estão envolvidos em uma invasão específica. Ao contrário dos IDS baseados em rede, esses sistemas podem saber o resultado de uma tentativa de invasão, uma vez que eles têm acesso direto e monitoram os arquivos e processos do sistema utilizados como alvos pelos invasores. Geralmente, esses sistemas utilizam informações de duas fontes: auditoria do sistema operacional (mais protegido e com informações mais detalhadas) e *logs* do sistema (menores e mais claros, sendo mais fáceis de analisar). Esses sistemas são mais difíceis de gerenciar, uma vez que as informações precisam ser configuradas e gerenciadas para cada estação monitorada. Além disso, eles não podem detectar invasões direcionadas a toda a rede. Sistemas baseados em Aplicações constituem um subgrupo dos sistemas

baseados em *Host* que utilizam os *logs* das aplicações para analisar os eventos ocorridos durante a execução dessas aplicações.

2. **Quanto ao método de análise**

   - **Por Assinatura**: Utilizam características comuns de ataques bem-conhecidos para classificar os dados coletados em normais ou maliciosos;
   - **Por Anomalias**: Constroem padrões de uso normal dos sistemas e detectam desvios significativos desses padrões como indicativo de possíveis atividades maliciosas.

Pode-se ainda classificar os sistemas de detecção de intrusão com base em três critérios: método de detecção, arquitetura e comportamento após a detecção. A Figura 1.1 esquematiza e exemplifica essa classificação.

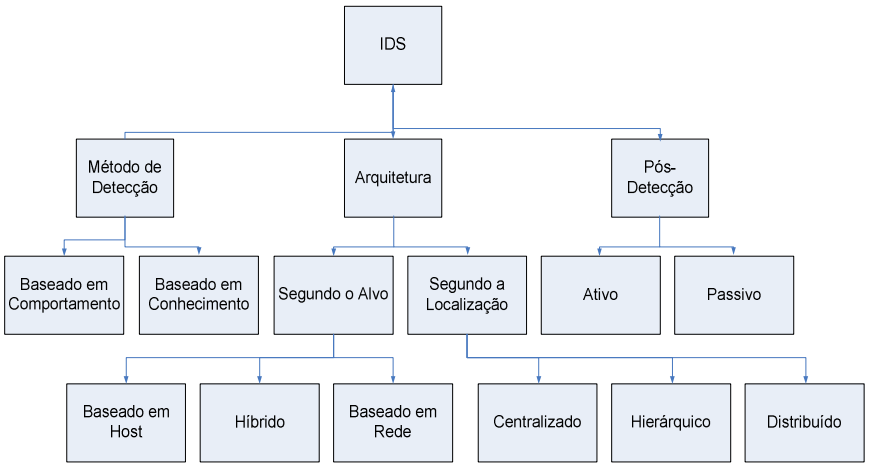

Figura 1.1 – Classificação de um IDS (Silva, 2002)

Independentemente do tipo de IDS, sua funcionalidade pode ser logicamente distribuída em três componentes: sensores, analisadores e interface. Os sensores são os responsáveis por coletar os dados para análise. As possíveis entradas para um sensor podem ser pacotes de redes ou arquivos de *logs*. Os sensores recolhem e transferem esses dados para os analisadores. Os analisadores, por sua vez, recebem dados dos sensores (ou de outros analisadores) e determinam se uma intrusão ocorreu. A saída de um analisador é a indicação de uma intrusão e, ainda, constituem evidências que apóiem essa indicação. Analisadores ainda podem fornecer uma lista de ações que podem ser tomadas quando houver uma intrusão. Já a interface permite que um usuário analise a saída do sistema ou controle o seu comportamento.

Ocorre que atualmente os ataques se "disfarçam" ou utilizam a criptografia para não serem detectados. Isto é resolvido pelas chamadas "assinaturas de ataques", que são comportamentos de cenários de ataques conhecidos (Raihan e Zulkernine, 2005). Portanto, a maioria das pesquisas em IDS deve focar no projeto de representação prática e eficiente de padrões para representação de situações complexas, exigindo uso, inclusive, de estruturas de dados como árvores. Em resumo, um sistema ideal de detecção de intrusão deve possuir um mecanismo de detecção eficiente e prover uma

boa representação de conhecimentos de padrões de intrusão os quais devem ser de fácil entendimento e manutenção.

1.3 Principais tipos de ataques à rede de computadores

Um ataque é uma tentativa de acesso não autorizado, ou seja, é o que acontece quando uma ameaça[1] tenta levar vantagem sobre as vulnerabilidades[2] de um determinado sistema. Um ataque (ou intrusão) pode ser formalmente definido, segundo Heady *et al.* (1990) como: "qualquer conjunto de ações que tentem comprometer a integridade, confidencialidade ou disponibilidade dos dados e/ou do sistema". Pode-se ainda definir um ataque, de forma mais abrangente, como qualquer violação à política de segurança de um sistema.

Os ataques podem ser divididos em duas categorias: ataques ativos e ataques passivos (Kurose e Ross, 2005). Ataques ativos correspondem às ameaças que alteram o sistema na tentativa de levarem vantagem sobre uma vulnerabilidade do mesmo, enquanto que os ataques passivos consistem em ameaças que simplesmente observam informações que trafegam no sistema. Contrariamente, não são inseridas informações que explorem as vulnerabilidades desse sistema.

Segundo Kurose e Ross (2005), os principais tipos de ataques à rede de computadores são: vírus, *worms*, *port scanning* – varredura de portas, DoS (*Denial of Service*) – negação de serviço, DDoS (*Distributed Denial of Service*) – negação de serviço distribuído. Esses ataques são descritos nas seções que seguem.

1.3.1 Vírus, *Worms* e Cavalos de Tróia

Vírus, *worms* (vermes) e cavalos de tróia (*trojans horses*), típicos exemplos de códigos maliciosos, são considerados os maiores problemas dos administradores de redes.

Os vírus são programas de computadores (ou fragmentos de programas), geralmente maliciosos, que se propagam infectando o computador, gerando cópias de si mesmo e tornando-se parte de outros programas de computador. Eles dependem da execução do programa hospedeiro para serem ativados e continuarem o processo de infecção (Cert.br, 2006). Além de serem capazes de se reproduzirem, eles podem também corromper arquivos e sistemas.

A propagação dos vírus se dá através de disquetes, CD-ROM, documentos infectados que são executados, de e-mails (com anexos infectados), de programas piratas, da execução de *downloads* de procedência duvidosa (*orkut, msn, Skype*), entre outros. Em todos esses exemplos, faz-se necessária a presença e atuação do ser humano (Moura *et al.*, 1999).

Alguns vírus são pré-programados para danificar o computador de forma a corromper os programas, excluir arquivos ou até mesmo simplesmente construídos para transmitir mensagens cujo intuito é chamar a atenção de um número grande de pessoas.

---

[1] Ameaça a um sistema, neste texto, consiste em querer atentar contra a segurança desse sistema.

[2] Vulnerabilidade pode ser entendida como ponto fraco de projeto, implementação ou configuração de sistema que pode ser explorado com más intenções.

Independentemente do intuito, constituem, normalmente, situações que acabam por culminar na queda ou parada temporária do sistema.

Os *worms* podem, por outro lado, ser considerados uma espécie de vírus, que devido a sua forma de reprodução são considerados mais desastrosos que os vírus normais (Moura *et al*., 1999).

Por definição os *worms* são programas capazes de se reproduzirem de um computador para o outro, assim como os vírus, mas, diferentemente dos vírus, os *worms* não utilizam um programa como hospedeiro. Eles possuem a capacidade de se propagarem automaticamente no próprio computador ou de computador para computador, realizando assim a infecção (Cert.br 2006).

Um exemplo de propagação de *worms* é o do ***Melissa*** (CERT 1999-04), que após infectar um computador ele mesmo procura os endereços no programa cliente de *e-mail* da vítima e transmite, de forma transparente para o usuário, o documento infectado para todos os *e-mails* cadastrados no contato da vítima.

Os *worms* são considerados mais perigosos que os vírus comuns, pois, além da não necessidade da intervenção humana, eles fazem uso de situações normais de operação do sistema (por parte do usuário) para efetuarem sua propagação.

Os *trojans*, ou mais conhecidos como cavalos de tróia, são um exemplo de código malicioso que aparentemente se mostra inofensivos. Eles vêem escondidos em cartões virtuais, protetores de tela, fotos. Esses programas além de executarem funções para as quais foi aparentemente projetado, também executam outras funções normalmente maliciosas e sem o conhecimento do usuário (Cert.br 2006). Esse código malicioso costuma executar automaticamente diversas funções como captura de dados, alterações de características e configurações de um sistema ou permitir acesso remoto em uma base cliente/servidor.

Os métodos utilizados para a instalação desses programas no sistema são muito variados, principalmente porque os programas estão tipicamente escondidos no meio do código de outro programa ou são inseridos por meio de acesso não autorizado.

De um modo geral, as formas de se combater os vírus, *worms* e cavalos de tróia são:

- Utilizar (e manter atualizado) antivírus para detectar a presença dos mesmos;
- Não abrir *e-mail*s de procedência duvidosa;
- Não efetuar *download* de *softwares* e documentos de sites duvidosos;
- Estar atento sempre para as novas ameaças.

Uma vez que as ameaças, atualmente, são produzidas de formas variadas, fazendo uso de técnicas cada vez mais sofisticadas e audaciosas, existe a preocupação em se prover técnicas mais eficientes no combate aos diferentes tipos de ataques.

1.3.2 *Port Scanning* – Varredura de Portas

Port scanning é uma técnica de varredura de portas TCP (*Transmission Control Protocol*) comum a *crackers* para reconhecimento de sistemas alvo. Esse ataque consiste em testar as portas de um *host*, ou mesmo de um grupo de *hosts*, a fim de determinar quais dessas portas estão em condições de aceitar conexões. Dessa forma, um programa de *port scan* é aquele que averigua conexões cujos números de porta são bem conhecidos para detectar informações e serviços em execução no sistema alvo. Assim, com base na lista de portas que estão aguardando conexões, o *cracker* pode escolher entre um ou outro método de invasão (Kurose e Ross, 2005).

O *port scanning*, eventualmente, também é capaz de revelar outras informações de interesse do atacante tais como o sistema operacional em execução, a partir da análise de como o alvo reage aos eventos gerados durante a varredura.

Podem ocorrer ligeiras diferenças na forma de como é realizado o *port scanning*, mas ele consiste, basicamente, em enviar uma série de requisições, seja via TCP ou UDP (*User Datagram Protocol*), para um *range* de portas de um determinado *host* e verificar as respostas. Essas respostas podem ser utilizadas para determinar quais serviços estão ativos, por exemplo: *Web* (*HTTP - Hiper Text Transfer Protocol* – porta 80), transferência remota de arquivos (*FTP - File Transfer Protocol* – porta 21), ou *e-mail* (*SMTP – Simple Mail Tranfer Protocol* – porta 25) (Kurose e Ross, 2005).

Dentre os diferentes tipos de varredura podem-se destacar:

- **Varredura padrão**: a varredura é iniciada com o envio de um pacote TCP SYN (*Synchronize*) para o alvo. O *three-way handshake* é executado de forma completa e a conexão entre o *host* que executa o *port scanner* e o alvo é estabelecida, possibilitando a obtenção de informações mais abrangentes. Entretanto, facilita que a varredura seja percebida e sua origem detectada.
- **Varredura TCP SYN**: a varredura também é iniciada com o envio de um pacote TCP SYN para o alvo, entretanto a conexão não é completada. Uma vez que o alvo responde à solicitação de abertura de conexão (através de um pacote TCP, ACK (*Acknowledgement*) ou RST (*Reset*)), o *port scanner* já é capaz de avaliar se a porta está em estado de escuta e interrompe o processo de conexão. Por esse motivo, esse tipo de varredura é muitas vezes conhecido também como varredura semi-aberta. A vantagem desse método é que ele dificulta a detecção da origem da varredura.
- **Varredura *Stealth Scanning***: a varredura é, nesse caso, iniciada com um pacote que simula uma conexão já existente, e não mais com um pacote TCP SYN, e desse modo, a forma de resposta ao pacote revela não só o estado da porta, mas consegue fornecer informações extras sobre o estado do alvo. O propósito dessa varredura é evitar que filtros de pacote (bloqueio de pacotes TCP SYN) inviabilizem a varredura.

Independentemente do tipo de varredura, de um modo geral, elas são consideradas técnicas de ataque que se aproveitam do fato de que o protocolo TCP/IP

(*Internet Protocol*) gerencia as conexões de forma automatizada e com um nível de crítica muito simplista. Sendo assim, um *host* responde a toda solicitação de abertura de conexão endereçada a suas portas, sem avaliar a origem do pedido (se esta é confiável ou não). Além disso, o uso de portas conhecidas associadas a serviços padrão constitui outra característica explorada por esse tipo de ataque. Um ataque desse tipo é feito pelo uso de *softwares* específicos capazes de: enviar pacotes com destino às portas do *host* alvo; monitorar respostas emitidas pelo alvo e; gerar relatórios de análise.

Dentre as ferramentas empregadas para a realização do *port scanning* a mais popular é o *nmap* (http://www.insecure.org/nmap/), tendo sua popularidade advinda do fato de reunir as mais diversas técnicas de varredura. A Figura 1.2 apresenta um exemplo de um *port scanning* em ambiente *Linux* utilizando o *nmap*.

```
# Nmap 4.11 scan initiated Wed Aug 29 01:11:20 2007 as: nmap -v -sS -sU
-O -oN nmapTCPUDP.txt 200.XXX.XXX.XXX
Interesting ports on 200.XXX.XXX.XXX:
Not shown: 3151 closed ports
PORT      STATE         SERVICE
22/tcp    open          ssh
25/tcp    open          smtp
53/tcp    open          domain
80/tcp    open          http
111/tcp   open          rpcbind
443/tcp   open          https
631/tcp   open          ipp
901/tcp   open          samba-swat
951/tcp   open          unknown
953/tcp   open          rndc
53/udp    open|filtered domain
111/udp   open|filtered rpcbind
631/udp   open|filtered unknown
945/udp   open|filtered unknown
948/udp   open|filtered unknown
32768/udp open|filtered omad
No exact OS matches for host (If you know what OS is running on it, see
http://www.insecure.org/cgi-bin/nmap-submit.cgi).
TCP/IP fingerprint:
SInfo(V=4.11%P=i686-redhat-linux-gnu%D=8/31%Tm=46D8126F%O=22%C=1)
TSeq(Class=RI%gcd=1%SI=366EDF%IPID=Z%TS=1000HZ)
TSeq(Class=RI%gcd=1%SI=366DE0%IPID=Z%TS=1000HZ)
TSeq(Class=RI%gcd=1%SI=366D9C%IPID=Z%TS=1000HZ)
T1(Resp=Y%DF=Y%W=8000%ACK=S++%Flags=AS%Ops=MNNTNW)
T2(Resp=N)
T3(Resp=Y%DF=Y%W=8000%ACK=S++%Flags=AS%Ops=MNNTNW)
T4(Resp=Y%DF=Y%W=0%ACK=O%Flags=R%Ops=)
T5(Resp=Y%DF=Y%W=0%ACK=S++%Flags=AR%Ops=)
T6(Resp=Y%DF=Y%W=0%ACK=O%Flags=R%Ops=)
T7(Resp=Y%DF=Y%W=0%ACK=S++%Flags=AR%Ops=)
PU(Resp=Y%DF=N%TOS=C0%IPLEN=164%RIPTL=148%RID=E%RIPCK=E%UCK=E%ULEN=134%
DAT=E)
Uptime 0.007 days (since Wed Aug 29 01:01:16 2007)
TCP Sequence Prediction: Class=random positive increments
                         Difficulty=1346678 (Good luck!)
IPID Sequence Generation: All zeros

# Nmap run completed at Wed Aug 29 01:11:30 2007 -- 1 IP address (1
host up) scanned in 9.893 seconds
```

Figura 1.2 – *Port scanning* utilizando o *nmap* em um ambiente *Linux*.

Existem outras ferramentas que realizam *port scanning* a exemplo do *pscan* escrito por Pluvius em 1995, podendo o código fonte ser encontrado em *http://packetstormsecurity.org/UNIX/scanners/pscan.c*.

A Figura 1.3 apresenta um exemplo de um *port scanning* em ambiente *Linux* utilizando o *pscan*.

```
[root@notebook scan]# ./pscan -t 201.XXX.XXX.XXX
scanning host 201.XXX.XXX.XXX's tcp ports 1 through 10240
port 22 (ssh) open
port 80 (http) open
port 111 (sunrpc) open
port 443 (https) open
port 901 (swat) open
port 946 (UNKNOWN) open
```

Figura 1.3 – *Port scanning* utilizando o *pscan* em um ambiente *Linux*.

1.3.3 DoS (*Denial of Service*) – Negação de Serviço

Outra categoria de ataque que constitui uma ameaça à segurança das redes de computadores é o classificado ataque de negação de serviço.

Esta categoria de ataque tem como finalidade, como sugere o próprio nome, tornar impossível a utilização dos recursos de uma rede ou de um determinado *host*. Normalmente esse tipo de ataque sobrecarrega a infra-estrutura sob ataque, ficando esta impossibilitada de realizar os trabalhos legítimos (Kurose e Ross, 2005). Constitui um ataque baseado na sobrecarga da capacidade ou em uma falha não prevista.

Esse tipo de ataque tem como objetivo esgotar os recursos do sistema alvo, forçando uma interrupção total ou parcial dos serviços. A capacidade de processamento, de armazenamento de dados e a largura de banda são alguns dos recursos visados pelas técnicas de negação de serviços.

O problema principal está focado no protocolo IP, que é altamente vulnerável a ataques DoS. Além disso, muitas ferramentas de ataques estão disponíveis para o acesso público e são relativamente fáceis de utilizar.

Embora existam técnicas de negação de serviços destinadas a atacar *hosts* e computadores pessoais, a maioria dos ataques costuma ser direcionada contra sistemas de maior porte, os quais oferecem serviços a um grande número de usuários, como é o caso de servidores *WEB*, ou que desempenham funções críticas para o funcionamento de redes e sistemas distribuídos.

Outros motivos para existirem esse tipo de falha nos sistemas é um erro básico de programadores, falhas na implementação e *bugs* (erros), além de outras peculiaridades dos sistemas operacionais, na medida em que podem oferecer oportunidades para comprometer o funcionamento do sistema (pela incapacidade de tratar erros). Desse modo, o invasor inicia da premissa de que erros existem e que deve ficar efetuando diversos tipos de testes de falhas, até acontecer um erro e o sistema parar sua execução.

Esse tipo de ataque não causa perda ou roubo de informações, mas é um ataque preocupante, pois os serviços do sistema atacado ficarão indisponíveis por um tempo

indeterminado. Cabe ainda salientar que quando um computador/*site* sofre ataque DoS, ele não é invadido, mas sim sobrecarregado. Isso independentemente do sistema operacional utilizado.

Um ataque DoS pode ainda ser visto como um *worm* que se prolifera entre servidores infectados procurando novos computadores para se proliferar, entretanto, como o programa não se autodetecta ele se re-instala consumindo recurso das máquinas já infectadas exaurindo assim os recursos das máquinas.

Ataques de DoS podem ser lançados contra roteadores de borda, *bastion hosts*, e *firewalls*, sendo que roteadores, servidores DNS (*Domain Name Service*) e *firewalls* costumam ser os alvos preferenciais. Em ataques direcionados aos equipamentos de redes, o objetivo principal é tirar uma rede ou sub-rede inteira do ar e não somente algumas máquinas específicas. Existem *bugs* em vários tipos de roteadores e em outros equipamentos de conectividade que podem facilitar esses ataques. Ataques aos *firewalls* fazem com que esses percam a função de filtro, deixando passar conexões TCP para qualquer porta, sendo que um ataque DoS pode, e normalmente é, utilizado para facilitar uma invasão.

É importante ainda salientar que as técnicas de negação de serviços são freqüentemente adotadas como uma etapa intermediária de métodos de ataque mais complexos. Dessa maneira, elas servem como uma *armadilha* para deixar um *host* (sistema ou servidor) fora do ar a fim de que outro *host* assuma sua identidade ou, até mesmo, interrompa o funcionamento de um sistema que execute funções de segurança e controle da rede.

Existem três tipos principais de ataques de negação de serviço:

- **Exploração de falhas**: exploram as vulnerabilidades no *software* do sistema alvo causando falhas em seu processamento ou extinguindo seus recursos.
- ***Flooding***: enviam ao sistema alvo mais informação do que ele é capaz de manipular. Mesmo que a capacidade de processamento do sistema não seja totalmente absorvida o atacante pode ser capaz de monopolizar a conexão da rede do alvo, bloqueando assim qualquer tipo de uso deste recurso.
- **Ataques de negativa de serviço distribuído (DDoS)**: possuem uma estrutura previamente montada onde diversas máquinas lançam um ataque baseado no ataque *flooding* sobre o alvo. Esses ataques fazem uso de várias máquinas, sendo essas consideradas *zumbis*, para que o número de requisições de conexão ao servidor seja bem grande. São ataques mais complexos e eficientes, sendo que sua detecção também se torna mais complexa, chegando a algumas vezes se tornar até mesmo impossível.

As formas de ataque do tipo DoS mais conhecidas são:

- **SYN *Flooding*:** constitui um ataque de inundação SYN. Neste tipo de ataque um computador tenta estabelecer uma conexão com um servidor através de um sinal do TCP conhecido por SYN. Se o servidor atender ao pedido de conexão, será enviado ao computador solicitante um sinal

chamado ACK. O problema é que em ataques desse tipo, o servidor não consegue responder a todas as solicitações e então passa a recusar novos pedidos. Neste ataque, os pacotes TCP SYN/ACK enviados pelo alvo em resposta aos falsos pedidos de conexão não são respondidos. Isso normalmente se dá porque as solicitações são geradas com um endereço IP falso ou inválido no lugar do endereço verdadeiro da máquina que originou o ataque. Desse modo, a fila de conexões em andamento atinge seu limite configurado muito rápido e passa a descartar novas solicitações de conexão, tornando indisponíveis os serviços residentes no computador alvo. A vulnerabilidade de um sistema a este tipo de ataque pode ser minimizada se forem adotadas configurações especiais, como por exemplo, reduzir o limite de tempo após o qual uma conexão solicitada e não estabelecida é eliminada.

- **UDP *Packet Storm*:** neste tipo de ataque um computador faz solicitações constantes para que uma máquina remota envie pacotes de respostas ao solicitante. A máquina fica tão sobrecarregada que não consegue executar as funções para as quais foi previamente designada.
- **LAND:** neste tipo de ataque o foco são os datagramas IP. Ele baseia-se no efeito do recebimento de um datagrama IP pode ter sobre determinado sistema (os endereços de origem e destino são iguais). Isso pode produzir um *loop* que pode esgotar os recursos do computador, levando também a uma queda ou travamento do sistema. Pode ainda ocorrer variações, nas quais se tem a alteração dos campos do cabeçalho IP do pacote inválido (portas ou *bits* de controle).
- **Ataques baseados em ICMP (*Internet Control Message Protocol*):** neste tipo de ataque aproveita-se das funcionalidades do protocolo ICMP para criar eventos capazes de afetar o funcionamento de alvos específicos. O ICMP é empregado em tarefas de controle e verificação das comunicações entre *hosts* e roteadores. Ele utiliza mensagens padronizadas que são enviadas com o propósito de checar a possibilidade de comunicar-se com um *host* de destino. As implementações padrão do TCP/IP reagirão às mensagens ICMP recebidas executando as ações apropriadas a cada caso. Elas responderão a pedidos de *ICMP_ECHO* ou poderão encerrar conexões estabelecidas a partir do recebimento de mensagens do tipo *Destination Unreachable* ou *Time to Live Exceeded*. Normalmente, não será executada qualquer modalidade de autenticação dos pedidos ou de crítica de características especiais, como repetições excessivas. Assim, uma seqüência ininterrupta de mensagens ICMP é enviada ao *host* alvo que se ocupa em responder a todas elas, consumindo desnecessariamente seus recursos. Outro ataque desse mesmo tipo, denominado *Pong,* envia mensagens ICMP a um grande número de *hosts* endereçando-as em *broadcast*. Nas mensagens o endereço de resposta informado é o endereço do alvo. Sendo assim, quando todos os *hosts* respondem, o alvo recebe uma grande quantidade de mensagens ICMP simultaneamente e com isso suas comunicações são afetadas ou interrompidas. O ataque *smurf* consiste em fazer com que *hosts* inocentes

respondam requisições *echo* de pacotes ICMP para um determinado número IP falsificado pelo atacante. Constitui-se, então, em um aperfeiçoamento do ataque do tipo *Pong*, ampliando o número de *hosts* que enviarão mensagens ICMP ao alvo pelo envio da requisição falsa não apenas a um, mas a vários endereços de *broadcast.* O resultado é o envio de um grande número de pacotes ICMP de resposta ao *host* cujo número de IP foi falsificado (Kurose e Ross, 2005). Uma variante dessa técnica, denominada FRAGGLE, utiliza como protocolo de transporte o UDP em lugar do TCP.

- ***Teardrop*:** neste tipo de ataque é explorado o processo de remontagem do datagrama IP fragmentado, adulterando assim informações no cabeçalho IP de forma a produzir uma situação inadequada ao processamento. Esse processo leva a falhas ou instabilidade do sistema.
- ***Ping o'Death*:** neste tipo de ataque a técnica consiste do envio ao computador alvo de um datagrama com tamanho além do limite de 65535 *bytes*. Não constitui um ataque baseado em ICMP, apesar das primeiras versões serem baseadas no *ping*. Uma vez que o datagrama ultrapassa o limite de tamanho, ele é fragmentado por não poder ser roteado, chegando à origem na forma de vários datagramas (fragmentos do datagrama original). Quando o sistema começa o processo de remontagem acaba por ter queda de desempenho no sistema ou a paralisação do mesmo.
- **Ataques de dessincronização:** neste tipo de ataque é utilizada a fragilidade existente na conexão inicial do TCP. O atacante monitora os envios de pedido de abertura de conexão, e envia um sinal de RST para o *host* que foi solicitado fechando a conexão prematuramente. Logo após esse procedimento o atacante reenvia um sinal de solicitação de nova abertura de conexão. Desta forma ambos os *hosts* permanecem com as portas abertas sendo utilizadas e acreditando que estão em uma conexão válida. O resultado disso é que ambos terão uma conexão dessincronizada que não pode ser utilizada para transferências de dados, mas consome recurso de ambos.

1.3.4 DDoS (*Distributed Denial of Service*) – negação de serviço distribuído

Ataques DDoS são problemas sérios que afetam os usuários de *Internet*, uma vez que consomem os recursos de um *host* ligado à rede que prestam serviços, tais como *e-mail* (SMTP) e páginas *Web* (HTTP).

Basicamente, os ataques DDoS são coordenados por um atacante que de posse de *hosts* dedicados, conhecidos como *zumbis*, lança um ataque coordenado sobre uma rede ou *host* denominado vítima (Kurose e Ross, 2005). Esse tipo de ataque conquistou fama no início do ano 2000.

O DoS realiza o ataque através de um único computador. Mas, com o passar dos tempos observou-se que a idéia poderia ser expandida, utilizando uma série de computadores atacantes ao mesmo tempo poder-se-ia obter resultados mais desastrosos

e eficientes. Em verdade esse ataque potencializa os danos causados pelos ataques de negação de serviço.

Para que o ataque consiga sucesso faz-se necessário que sejam empregados *software*s específicos que visam organizar esse ataque. Tem-se como exemplo o *TFN*, o *Stacheldraht* e o *Trinoo*. Essas ferramentas são instaladas em alguns *hosts* que atuarão como servidores (mestres). Paralelamente, outros *hosts* recebem também componentes de *software*, passando por sua vez a representar o papel de clientes (escravos).

As instalações tanto dos servidores quanto dos módulos clientes são feitas de forma não autorizada, ou seja, os componentes são embutidos em outros programas supostamente inofensivos. Ao comando do atacante, os servidores se comunicam com os clientes, determinando o início do ataque, seguidos pelos *hosts* que executam o módulo cliente que lançam ao mesmo tempo uma série de ataques contra o alvo ou os alvos especificados.

Tanto para efetuar como para tentar evitar um ataque DDoS fazem-se necessárias ferramentas com um alto nível de sofisticação, integrando recursos avançados que vão desde mecanismos de distribuição automatizada dos módulos clientes até comunicações criptografadas entre os servidores e os clientes.

Dessa forma, os ataques DDoS merecem especial atenção, não apenas pela eficácia, mas também por estabelecer um novo modelo de ataque distribuído.

1.4 UML (*Unified Modeling Language*)

A UML ou *Unified Modeling Language* surgiu em junho de 1996 e trouxe recursos adequados para modelar cada fase do ciclo de vida do *software* de maneira estrutura e padronizada. A UML é baseada em técnicas orientadas a objetos e destina-se a visualização, especificação, construção e documentação dos artefatos de um sistema de *software* (Booch *et al.*, 2000).

Com a UML é possível modelar qualquer sistema implementado com linguagens orientadas a objetos (*Java*, C++, *Smalltalk*) e a partir dessa modelagem pode-se, por intermédio de ferramentas especializadas, gerar código-fonte em uma linguagem alvo. É possível também desenvolver o processo inverso dentro de algumas ferramentas (Monteiro, 2004).

O meta-modelo da UML possui blocos de construção (abstrações, relacionamentos e diagramas) que podem ser estendidos para a criação de um perfil (*Profile*). Um perfil é uma extensão da UML que permite modelar detalhes específicos de um domínio. As abstrações representam detalhes estruturais e comportamentais de um sistema; os relacionamentos declaram como as abstrações se relacionam e diagramas apresentam um resumo do conjunto de abstrações e seus relacionamentos (Stein, 2002).

1.4.1. A diferença entre Perfil UML e Notação UML

A linguagem de modelagem UML é a consolidação de outras técnicas de modelagem descritas por Booch e outros (Booch *et al.*, 2000) e possui alguns mecanismos de extensão que permitem definir conjuntos de elementos notacionais específicos de um determinado domínio, ferramenta ou projeto. Os mecanismos de extensão fornecidos pela UML são os estereótipos (*Stereotypes*), as etiquetas valoradas

(*Tagged Values*) e os adendos (*Advices*). Os estereótipos são usados para estender o vocabulário da UML introduzindo novos tipos de elementos. Cada estereótipo é um novo meta-elemento que estende alguma classe base (*Base class*) do meta-modelo da UML e que pode ser instanciado em algum modelo de projeto. Um estereótipo pode possuir um conjunto de propriedades (etiquetas valoradas), que geralmente são materializadas na forma de atributos. Uma etiqueta é basicamente um par que consiste de um nome e um valor e são representadas entre chaves, por exemplo: *{interface = nameOfInterface}*. As restrições são condições que devem ser mantidas verdadeiras para que a semântica do estereótipo fique correta.

Essas extensões podem ser definidas e agrupadas em um determinado perfil. Perfis visam a fornecer elementos de notação específicos de determinados domínios de aplicação com o objetivo de melhor representar suas particularidades e assim melhorar o entendimento da especificação. A própria OMG possui várias iniciativas para a criação de perfis UML.

A UML é composta por diversos diagramas que permitem descrever as particularidades mais relevantes dos sistemas que são implementados utilizando a abordagem orientada a objetos. Cada um dos diagramas foca uma dada visão do sistema e propositadamente enfatiza algumas particularidades e negligencia outras. A notação UML é independente da linguagem de programação e do processo de desenvolvimento adotados.

1.4.3 UMLintr (*UML for intrusion specifications*) – Uma proposta de UML para especificação de intrusão

Hussein e Zulkernine (2006) propuseram uma extensão para UML para modelar diversos tipos de ataques, sendo os principais DoS (*Denial of Service*), *remote to user, user to super-user* e *probe*. Essa extensão facilita a modelagem de cenários de ataques baseados em máquinas de estado. Porém, não atende a outros tipos de ataque como o DDoS.

Os diagramas UML utilizados por Hussein e Zulkernine são: diagramas de classe, diagrama de casos de uso e diagrama de máquina de estados. Cada um deles mostra o ataque em um diferente nível de abstração. O diagrama de caso de uso mostra o ataque no nível de abstração mais alto, podendo ser utilizado para as explicações aos consumidores. Os diagramas de classe apresentam a estrutura do ataque. O diagrama de máquina de estados apresenta o comportamento de um ataque.

Hussein e Zulkernine (2006) propuseram Estereótipos e Etiquetas Valoradas, que são valores para especificar um modelo UML que satisfaça um domínio particular, neste caso um cenário de ataque (ou intrusão). Estereótipos utilizados em casos de uso possuem seus próprios ícones e notações. Estereótipo e etiquetas valoradas nas classes são utilizadas como diretivas para ferramentas UML, podendo ser utilizado pelo IDS.

As Tabelas 1.1 e 1.2 apresentam, respectivamente, os Estereótipos e as etiquetas valoradas propostos por Hussein e Zulkernine (2006).

Tabela 1.1 – Estereótipos UMLintr

	Estereótipo	Classe Base	Etiquetas Valoradas	Descrição
1	<<DOS>>	Package		*Denial of service attacks*
2	<RemoteToUser>>	Package		*Remote to user attacks*
3	<<UserToRoot>>	Package		*User to root attacks*
4	<<Probe>>	Package		*Probe attacks*
5	<<attacker>>	Actor		*Attacker actor*
6	<<victim>>	Actor		*Victim actor*
7	<<abuse feature>>	Use-case		*Methods of exploitation*
8	<<misconfiguration>>	Use-case		
9	<<social engineering>>	Use-case		
10	<<masquerading>>	Use-case		
11	<<implementation bug>>	Use-case		
12	<<attacker>>	Class	initial-privilege, gained-privilege, skill, action	*Attacker class*
13	<<victim>>	Class	security	*Victim class*
14	<<intrusion>>	Class	severity, method, listeners	*Intrusion class*
15	<<deny>>	Connector		*Actions performed by attacker*
16	<<alter>>	Connector		
17	<<probe>>	Connector		
18	<<use>>	Connector		
19	<<intercept>>	Connector		

Tabela 1.2 - UMLintr etiqueta valorada (propriedades)

	Etiqueta Valorada	Descrição
1	initial-privilege	*Represents the initial access level of the attacker (e.g., remote network, local network, local user, root, or physical access)*
2	gained-privilege	*Represents the gained access level of the attacker (e.g., remote network, local network, local user, root, or physical access)*
3	skill	*The skill of the attacker (e.g., novice, intermediate, or experienced)*
4	action	*The action of the attacker (e.g., probe, deny, intercept, alter, or use)*
5	security	*The security level of the victim (e.g., low, medium, or high)*
6	severity	*The severity level of the attack (e.g., low, medium, or high)*
7	method	*The exploitation method of the attack (e.g., abuse feature, masquerading, implementation bug, misconfiguration, or social engineering*
8	listeners	*Events need to detect attacks (e.g., Windows audit, Unix log, Solaris BSM, or network packets)*

Os Estereótipos 1, 2, 3, e 4 da Tabela 1.1 representam quatro categorias de ataques. Os próximos (5, 6, 7, 8, 9, 10, e 11) representam estereótipos que classificam

atores e casos de uso que representam atacantes, vítimas e métodos. Os estereótipos (12, 13 e 14) representam as entidades que são utilizadas nos diagramas de classe. E, dessa forma os demais (15, 16, 17, 18 e 19) são conectores utilizados para especificar o efeito do ataque sobre a vítima nos diagramas de caso de uso.

Segundo Hussein e Zulkernine (2006) casos de uso são facilmente entendidos pelos usuários e auxiliam os desenvolvedores a pensarem em todas as possibilidades de ataque. Além disto, com estes diagramas, usuários que tentam danificar o sistema podem ser categorizados de acordo com sua experiência, meta e recursos atingíveis. O uso de caso constitui a base dos diagramas de classe e de máquina de estado.

Cada caso de uso envolve e um ator atacante *<<attacker>>* e um ator vítima *<<victim>>* conectados por meio de um relacionamento de intrusão.

Os estereótipos *<<intercept>>*, *<<alter>>*, *<<use>>*, *<<probe>>* e *<<deny>>* mostram o efeito da intrusão na vítima; *<<intercept>>* destina-se a ler arquivos, pacotes ou eventos em um sistema ou em uma rede; *<<alter>>* manipula dados ou *logs* da vítima; *<<use>>* caracteriza a ação de utilizar os recursos da vítima; *<<probe>>* incluem ações de coletar informações sobre usuários em um sistema ou sobre máquinas em uma rede; *<<deny>>* pára ou degrada um serviço.

Os estereótipos *<<social engineering>>*, *<<masquerading>>*, *<<misconfiguration>>*, *<<implementation bug>>* e *<<abuse feature>>* descrevem os métodos de exploração; *<<social engineering>>* representa métodos de exploração em que o atacante convence o usuário a fornecer o *username* e/ou senha; *<<masquerading>>* é a classe de métodos em que o atacante ganha o acesso para um serviço fornecendo usuário e senha legítimos; *<<misconfiguration>>* representa o atacante explorando uma falha de configuração na política de segurança; *<<implementation bug>>* representa o atacante explorando um *bug* na implementação de um serviço; *<<abuse feature>>* representa o atacante utilizando um serviço como uma forma de danificar o sistema.

Os pacotes *stereotyped* para UMLintr são: *<<DOS>>*, *<<RemoteToUser>>*, *<<UserToRoot>>* e *<<Probe>>*. Estes pacotes representam quatro tipos de ataques: negação de serviço, *remote to user*, *user to root* (ou super usuário) e ataques de supervisão (*surveillance*) / sondagem (*probe*), respectivamente.

DoS é o tipo de ataque no qual o atacante visa a parar ou degradar um serviço pelo abuso de sua utilização. *Remote to user* são os ataques nos quais o atacante com acesso remoto a um sistema, ganha o privilégio de usuário. *User to root* são os ataques em que o atacante que já possui acesso ao sistema como um usuário local, ganha o privilégio de *root*. *Probe* é coleta de informações sobre máquinas em uma rede para encontrar suas vulnerabilidades.

Após capturar os cenários de ataques nos diagramas de caso de uso, o próximo passo é representar esses ataques como classes. Três estereótipos são propostos por Hussein e Zulkernine (2006) para representar classes: *<<attacker>>*, *<<victim>>* e *<<intrusion>>*. As classes estereotipadas com esses estereótipos representam as entidades do atacante, vítima e intrusão, respectivamente. Cada estereótipo possui suas próprias etiquetas valoradas que apresentam informações relacionadas à segurança.

- *<<attacker>>* possui quatro *tags*: *initial-privilege*, *gained-privilege*, *action* e *skill*.
- *<<victim>>* possui uma *tag*: *security*.

- <<*intrusion*>> possui três *tags*: *severity*, *method* e *listeners*.
- *initial-privilege* é o privilégio com o qual o ataque começa.
- *gained-privilege* é o privilégio que com o qual o ataque termina.
- *action* é a operação que o atacante realiza após ganhar o acesso ao serviço.
- *skill* é utilizado para classificar atacantes baseado em suas habilidades e ferramentas.
- *security* representa como é a segurança da vitima.
- *severity* é o nível de danos que o ataque causa na vitima.
- *method* representa a estratégia que o atacante utilizar para lançar o ataque.
- *listeners* são os eventos em que o escritor de assinaturas necessita para detectar o ataque.

Exemplo de utilização da UMLintr para um Ataque - problemas existentes no programa ***Xterm*** e na biblioteca ***Xaw*** permitem ao usuário fornecer dados que causem *buffer overlfow* em ambos programa e biblioteca. Com este *bug* o usuário ganha o privilégio de *root*. O diagrama de classes para um ataque *Xterm* pode ser observado na Figura 1.4, na qual a classe *Nov_attacker* representa o atacante, a vitima é representada pela classe *Xaw_Library*. A implementação do *bug* pode ser observada na classe *Xterm*, mais especificamente com a *tag method implementation bug*.

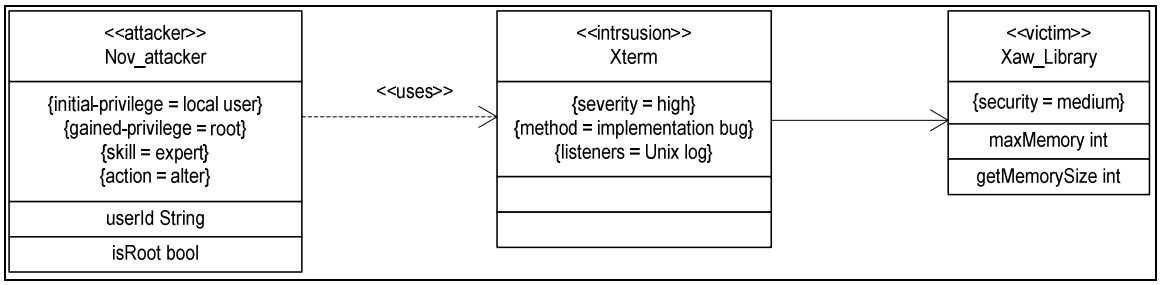

Figura 1.4 - Diagrama de Classe para ataque *Xterm*

1.5 Conceitos Básicos de POA – Programação Orientada a Aspectos

A Programação Orientada a Aspectos (Gregor Kiczales *et al*., 1997) e a linguagem *AspectJ* (Kiczales *et al*., 2001) surgiram no final da década de 90 como uma importante contribuição para modularizar interesses transversais, que até então ficavam misturados e espalhados pelo código orientado a objetos, sem que fosse possível organizá-los em módulos independentes. A POA fornece novas abstrações que contribuem para a separação de interesses (*separation of concerns*) (Dijkstra, 1976). Com a POA é possível implementar separadamente os interesses-base e os interesses transversais, o que até então era dificultado somente com orientação a objetos. Os interesses-base referem-se à funcionalidade principal do sistema e os transversais referem-se a restrições globais e a requisitos não-funcionais, como, por exemplo: persistência, distribuição, autenticação, controle de acesso e concorrência.

Tar e Ossher (2002) comentam que os paradigmas de desenvolvimento de *software* convencionais sofrem da Tirania da Decomposição Dominante, que consiste na

existência de apenas um tipo de módulo – o módulo dominante – para se encapsular mais de um tipo de interesse. Alguns exemplos de módulos dominantes são as classes na programação orientada a objetos, as funções nas linguagens funcionais e as regras na programação baseada em regras. Um paradigma que permite a decomposição em apenas um tipo de módulo é chamado de unidimensional, pois há apenas uma dimensão para a modularização dos interesses, não importando quantos tipos existem de interesses (Tar e Ossher, 2003).

A existência de apenas um tipo de módulo para se encapsular mais de um tipo de interesse faz com que aqueles interesses que envolvem restrições globais – chamados *interesses transversais* (Kiczales *et al.*, 1997) – não fiquem adequadamente encapsulados e acabam misturando-se com os interesses-base. Kiczales *et al.* (1997) foi quem cunhou o termo *interesses transversais* (*crosscutting concerns*), explicando que podem variar de requisitos de alto nível como segurança e qualidade de um serviço a noções de baixo nível como sincronização e manipulações de *buffer* de memória.

Esses *interesses* podem ser tanto funcionais, como regras de negócio, quanto não-funcionais, como gerenciamento de transações e persistência de dados. Alguns exemplos são: sincronização, interação de componentes, persistência, distribuição, adaptabilidade, gerenciamento de registros (*logging*), controle de acesso e de segurança. Alguns são dependentes de um domínio específico, como definição de produtos financeiros, enquanto outros são mais gerais, como por exemplo, sincronização e fluxo de trabalho(*work-flow*).

O termo *interesse transversal* faz analogia com o fato de que sua implementação com técnicas tradicionais de programação entrecorta transversalmente os módulos do sistema, afetando vários módulos e causando entrelaçamento (*tangling*) e espalhamento (*spreading*) de código de diferentes interesses. O entrelaçamento ocorre quando o código de um determinado interesse encontra-se misturado com o código de um outro interesse dentro de um mesmo módulo. O espalhamento ocorre quando o código de um interesse encontra-se em vários módulos do sistema. O entrelaçamento e o espalhamento de código de diferentes interesses causam problemas de manutenção, reúso e evolução.

Elrad et al. (2001) ilustram por meio da Figura 1.5 a natureza de um interesse transversal. O diagrama mostrado é de um editor de figuras que deve ser evoluído com a inclusão de um novo interesse, que consiste em notificar o mecanismo de exibição (classe *Display*) toda vez que um elemento do tipo *FigureElement* for movimentado.

Realizar essa operação de manutenção usando programação orientada a objetos consiste em adicionar em todos os métodos que alteram a posição de um ponto ou de uma linha (os métodos `set*()`) uma chamada ao método `updating()` da classe `Display`, como é mostrado em negrito na Figura 1.6. Note-se que o código desse novo interesse ficou entrelaçado com o código de cada um dos métodos e espalhado pelos vários métodos do sistema.

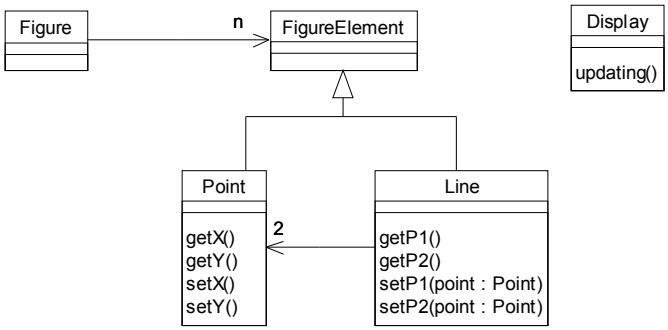

Figura 1.5 – Diagrama de Classes de um Editor Gráfico (Elrad et al., 2001)

```
class Line
{
   privatePoint p1, p2;

  Point getP1() {return p1;}
  Point getP2() {return p2;}

  void setP1(Point p1)
  {
      this.p1 = p1;
      Display.updating();
  }
  void setP2(Point p2)
  {
      this.p2 = p2;
      Display.updating();
  }
}
```

```
class Point
{
  int x, int y;

 int getX() { return x};
 int getY() { return y}

 void setX(int x)
 {
      this.x = x;
      Display.updating();
 }
 void setY(int y)
 {
      this.y = y;
      Display.updating()
 }
}
```

Figura 1.6 – Implementação Orientada a Objetos do Interesse de Atualização do *Display*

Com o objetivo de tratar esse problema, Kiczales et al. (1997) criou a (POA), que é baseada na idéia de que sistemas são programados de forma mais adequada distinguindo interesses que são *transversais* de interesses que podem ser considerados como *base*. Assim, os interesses transversais são projetados e implementados separadamente dos interesses-base. Os interesses transversais são implementados em módulos *aspectuais* e os interesses-base em classes normais.

A POA introduz novas abstrações de modularização e mecanismos de composição para melhorar a separação de interesses transversais. As novas abstrações propostas pela POA são: aspectos (*aspects*), pontos de junção (*join points)*, conjuntos de pontos junção (*pointcuts*), adendos e declarações intertipos (*inter-type declarations/introductions*).

Aspecto é o termo usado para denotar a abstração da POA que dá suporte a um melhor isolamento de interesses transversais. Em outras palavras, um aspecto corresponde a um interesse transversal e constitui uma unidade modular projetada para afetar um conjunto de classes e objetos do sistema.

Os pontos de junção são pontos bem definidos da execução de um sistema. Alguns exemplos de pontos de junção são: chamadas a métodos, execuções de métodos,

leitura de atributos e modificação de atributos. Por meio dos pontos de junção, torna-se possível especificar o relacionamento entre aspectos e classes. Por exemplo, um aspecto A afeta uma classe C1 no **instante da invocação de um método M1**. Em outro exemplo, o aspecto A afeta uma classe C2 **ao final da execução de um método M2**. O ***instante da invocação do método M1*** e o ***ponto final da linha de execução do método M2*** são exemplos de pontos de junção. Tais pontos de junção são os locais no programa onde os aspectos podem atuar. Eles podem ser especificados em uma linguagem orientada a aspectos através da definição de conjuntos de junção. Um conjunto de junção é o mecanismo que especifica os pontos de junção em que aspectos e classes se relacionam.

A implementação do comportamento transversal modularizado por um aspecto é feita em *adendos*. Adendo é um construtor semelhante a um método de uma classe que define o comportamento dinâmico executado quando são alcançados um ou mais conjuntos de junção definidos previamente. Existem três tipos adendos: os adendos anteriores (*before*), os posteriores (*after*) e os de substituição (*around*). Os adendos anteriores são executados sempre que os pontos de junção associados são alcançados e antes do prosseguimento da computação; os adendos posteriores são executados no término da computação, ou seja, depois que os pontos de junção forem executados e imediatamente antes do retorno do controle ao chamador; os adendos de substituição tomam o controle do código-base podendo ou não devolver o controle.

As declarações *intertipos* podem ser utilizadas quando se deseja introduzir atributos e métodos em classes-base do sistema. Ao contrário de adendos, que operam de forma dinâmica, as declarações *intertipos* operam estaticamente, em tempo de compilação.

São três as propriedades básicas da POA (Kiczales *et al*., 1997), (Elrad *et al*., 2001):

• **dicotomia aspectos-base**: diz respeito à adoção de uma distinção clara entre classes e aspectos. Os sistemas orientados a aspectos são decompostos em classes e aspectos. Os aspectos modularizam os interesses transversais e as classes modularizam os interesses-base;

• **inconsciência**: propriedade desejável da programação orientada a aspectos. É a idéia de que os componentes não precisam ser preparados para serem entrecortados por aspectos (Elrad *et al*., 2001). Pela propriedade de inconsciência, os componentes não percebem os aspectos que poderão afetá-los. Embora seja uma propriedade interessante, algumas pesquisas atuais começam a apontar vantagens quando o código-base possui consciência da existência dos aspectos (Kiczales e Mezini, 2005) (Griswold, *et al*., 2006);

• **quantificação**: capacidade de escrever declarações unitárias e separadas que afetam muitos pontos de um sistema (Elrad *et al*., 2001). Pela propriedade de quantificação, é possível fazer declarações do tipo *em programas P, sempre que a condição C for verdadeira*, *faça a ação A*.

As linguagens de programação orientadas a aspectos devem apoiar a definição de atributos, métodos, conjuntos de junção, adendos e declarações *intertipos* em

aspectos. A linguagem *AspectJ* (Kiczales *et al*., 2001), que é uma extensão orientada a aspectos da linguagem Java, oferece suporte à definição dessas abstrações.

Há várias abordagens lingüísticas para a implementação de sistemas orientados a aspectos, por exemplo, AspectC, AspectC#, AspectS e Appostle e AspectJ. Porém, como *AspectJ* é a linguagem mais difundida e a que foi utilizada para o desenvolvimento dos FTs, será a única descrita com detalhes.

1.5.1 A Linguagem *AspectJ*

Nesta seção, os conceitos básicos apresentados na seção anterior serão mostrados no contexto da linguagem *AspectJ*.

Um aspecto é uma unidade modular que encapsula interesses que são ortogonais a outras classes do programa (Gradecki e Lesiecki, 2003). Pode possuir atributos e métodos e participar de uma hierarquia de aspectos por meio da definição de aspectos especializados. Pode também ser combinado com algumas classes que ele entrecorta de acordo com especificações feitas no aspecto. Além disso, um aspecto pode introduzir métodos e atributos dentro de outras classes por meio do conceito de introduções/declarações *intertipos*.

Aspectos podem alterar a estrutura estática de um sistema adicionando membros (atributos, métodos e construtores) a uma classe, alterando a hierarquia do sistema. Esta característica de alterar a estrutura estática de um programa é chamada "entrecorte estático" (*static crosscutting*). Além de afetar a estrutura estática, um aspecto também pode afetar a estrutura dinâmica de um programa. Isto é possível por meio da interceptação de pontos no fluxo de execução, chamados pontos de junção, e da adição de comportamento antes ou depois dos mesmos, ou ainda por meio da obtenção de total controle sobre o ponto de execução.

Exemplos de pontos de junção são: invocação e execução, métodos, iniciação de objetos, execução de construtores, tratamento de exceções, acesso e atribuição a atributos, entre outros. Ainda é possível definir um ponto de junção como resultado da composição de vários pontos de junção.

Normalmente um aspecto define pontos de junção, os quais selecionam *pontos de junção* e valores nestes *pontos de junção* e adendos que definem o comportamento a ser tomado ao alcançar os *pontos de junção* definidos pelos conjuntos de pontos de junção.

A seguir serão explicados cada um dos principais conceitos básicos implementados pela linguagem *AspectJ*.

1.5.1.1 - Modelo de *Pontos de Junção*

Um *ponto de junção* é um ponto bem definido no fluxo de execução de um programa. A Figura 1.7 mostra um exemplo de fluxo execução entre dois objetos – cada um representado pelo círculo – e identifica alguns *pontos de junção* no mesmo – representados pelos pontos pintados em cinza.

O primeiro *ponto de junção* é a invocação de um método do objeto A, o qual pode retornar com sucesso, ou levantar uma exceção. O próximo *ponto de junção* é a execução deste método, que por sua vez também pode retornar com sucesso ou levantar uma exceção. Durante a execução do método do objeto A é invocado um método do

objeto B. A invocação e execução deste método são os *pontos de junção*, e da mesma forma que os do objeto A podem retornar com sucesso ou levantar uma exceção.

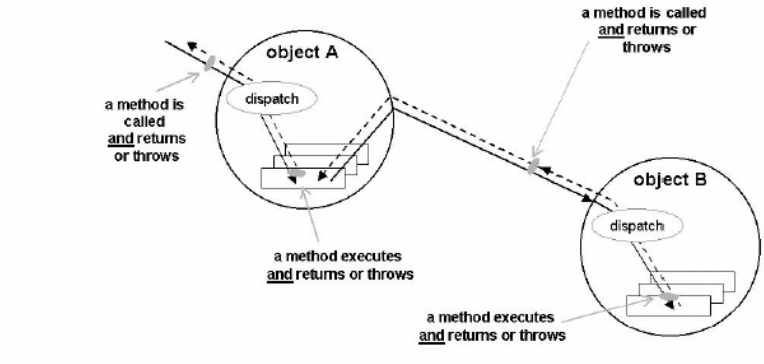

Figura 1.7 – Modelo de Ponto de Junção da Linguagem AspectJ (Elrad et al., 2001)

1.5.1.2 – Conjuntos de pontos de Junção

Conjuntos de pontos de Junção são formados pela composição de *pontos de junção*, através dos operadores (e), (ou), e (não). Utilizando esses conjuntos podem-se obter valores de argumentos de métodos, objetos em execução, atributos e exceções dos *pontos de junção*.

O trecho de código a seguir é a definição de um *conjunto de junção* que identifica as **invocações** de todos os métodos cujo nome inicia com `visualizarProdutos`, com qualquer tipo de retorno, nomes, ou parâmetros, devido ao uso dos *wildcards* * *e "..".*

```
public pointcut pt1(): call (* Pessoa+.visualizarProdutos*(..));
```

Além disso, este *conjunto de junção* restringe os *pontos de junção* àqueles os quais o código em execução pertencer aos subtipos (devido ao uso do *wildcard* +) da classe `Pessoa`. Para definir conjuntos de pontos de junção, identificando os *pontos de junção* a serem afetados, utilizamos construtores de AspectJ chamados designadores de conjuntos de pontos de junção (*conjunto de junção designators*), como os apresentados na Tabela 1.3.

Tabela 1.3 – Designadores de conjuntos de pontos de junção

`call`(*Assinatura*)	Invocação de método/construtor identificado por *Assinatura*
`execution`(*Assinatura*)	Execução de método/construtor identificado por *Assinatura*
`get`(*Assinatura*)	Acesso a atributo identificado por *Assinatura*
`set`(*Assinatura*)	Atribuição de atributo identificado por *Assinatura*
`this`(*PadrãoTipo*)	O objeto em execução é instância de *PadrãoTipo*
`target`(*PadrãoTipo*)	O objeto de destino é instância de *PadrãoTipo*
`args`(*PadrãoTipo, ...*)	Os argumentos são instâncias de *PadrãoTipo*
`within`(*PadrãoTipo*)	O código em execução está definido em *PadrãoTipo*

Padrão Tipo é uma construção que pode definir um conjunto de tipos utilizando *wildcards*, como * e +. O primeiro é um *wildcard* conhecido, pode ser usado sozinho

para representar o conjunto de todos os tipos do sistema, ou depois de caracteres, representando qualquer seqüência de caracteres. O último deve ser utilizado junto ao nome de um tipo para assim representar o conjunto de todos os seus subtipos.

1.5.1.3 – Adendos

Adendos são construções que definem código adicional que deverá executar nos *pontos de junção*.

O exemplo abaixo mostra dois adendos, um do tipo `before()` e o outro do tipo `after()`. Ambos atuam sobre o conjunto de pontos de junção `pt1()` definido anteriormente. Isso faz com que antes das chamadas aos métodos que iniciam com `visualizarProdutos`, o método `beginningMethod()` da classe `TracedClass` seja chamado, e o mesmo é feito depois dessas chamadas.

```
before() : pt1() {
     TracedClass.beginningMethod();
}
after() : pt1(){
     TracedClass.finishingMethod();
}
```

1.5.1.4 – Entrecorte Estático

Como mencionado anteriormente, a linguagem *AspectJ* permite alterar a estrutura estática de um programa por meio da adição de membros de classe, da alteração da hierarquia de classes ou da substituição de exceções checadas por não checadas.

O mecanismo que adiciona membros a uma classe é chamado de introduções ou declarações *intertipos* (*intertype declarations*). O aspecto mostrado no trecho de código abaixo introduz em uma classe chamada `Cliente` um atributo empresa, um método `setEmpresa()`, um método `getEmpresa()` e um novo construtor.

```
public aspect EmpresasConveniadas {

       private Empresa Cliente.empresa;

       public void Cliente.setEmpresa(Empresa empresa){...}

       public Empresa Cliente.getEmpresa(){...}

       public Cliente.new(...){...}
}
```

As introduções também podem ser utilizadas para alterar a hierarquia de classes. Por exemplo, o aspecto abaixo faz com que as classes `Hotel` e `Cliente` implementem a interface `PersistentRoot`. Dessa forma, todos os métodos existentes dentro dessa interface serão herdados pelas classes `Hotel` e `Cliente`. Este é um recurso bastante interessante quando não se deseja poluir as classes `Cliente` e `Hotel` com declarações `implements` dentro de seu código.

```
public aspect MyPersistentEntities extends PersistentEntities {
       declare parents: Hotel implements PersistentRoot;
       declare parents: Cliente implements PersistentRoot;
       ...
```

Na Tabela 1.4 são apresentadas as outras construções em *AspectJ* que alteram a estrutura estática de um programa.

Tabela 1.4 – Construções em AspectJ que alteram a estrutura estática de um programa

`declare parents` : *PadrãoTipo* `extends` *ListaTipos*;	Declara que os tipos em *PadrãoTipo* herdam dos tipos em *ListaTipos*
`declare parents` : *PadrãoTipo* `implements` *ListaTipos*;	Declara que os tipos em *PadrãoTipo* implementam os tipos em *ListaTipos*
`declare soft` : *PadrãoTipo*: *Pointcut*;	Declara que qualquer exceção de um tipo em *PadrãoTipo* que for lançada em qualquer *join point* identificado por *Pointcut* será encapsulada em uma exceção não checada

1.5.3.5 – Aspectos Abstratos

A linguagem *AspectJ* também permite a criação de aspectos abstratos, semelhantemente a uma classe abstrata em *Java*, porém, além de métodos abstratos o aspecto também pode possuir conjuntos de junção abstratos.

Um conjunto de junção abstrato não tem conhecimento dos pontos de junção que serão afetados, pois esses pontos devem ser informados em um aspecto concreto que especializar o aspecto abstrato. Como um adendo pode ser definido sobre um conjunto de junção abstrato, pode-se implementar um comportamento transversal no aspecto abstrato sem saber de antemão quais são os pontos de junção, e conseqüentemente, qual é o código-base em que esse comportamento irá atuar. Aspectos abstratos são a base do desenvolvimento orientado ao reúso e, conseqüentemente, ao desenvolvimento de *frameworks* orientados a aspectos.

Na Figura 1.8 é mostrado um aspecto abstrato cuja tarefa é estabelecer conexões com o banco de dados. Há um conjunto de junção abstrato denominado `doConnection()` e um adendo do tipo `after()` que atua sobre ele. O método `ConnectDB()` sempre será chamado após (*after*) os pontos de junção que serão informados no aspecto concreto que especializar esse aspecto abstrato.

Assim, um aspecto concreto deve ser criado para especializar esse aspecto abstrato e concretizar o conjunto de junção `doConnection()` com algum ponto de junção em que a conexão deve ser estabelecida, como está sendo exibido na Figura 1.9.

Esse aspecto concreto indica que sempre que um método *init()* for executado a conexão deve ser estabelecida.

```
public abstract aspect AbstractAspectConnection
{
       ...
       abstract pointcut doConnection();

       after() : doConnection()
       {
           ConnectDB();
       }
}
```

Figura 1.8 – Aspecto Abstrato

```
public aspect AspectConnection extends AbstractAspectConnection
{
   pointcut doConnection(): execution (void *.init(..));
}
```

Figura 1.9 – Aspecto Concreto

Na Figura 1.10 é mostrada uma possível implementação que visa solucionar o problema da atualização do gerenciador de exibição mostrado anteriormente na Figura 1.6.

Criou-se um aspecto que possui um conjunto de junção denominado *move()*, o qual agrupa dois pontos de junção: o primeiro são chamadas a todos os métodos que iniciam com as letras *set* da classe *Line* e o segundo faz o mesmo para a classe *Point*. Esses dois pontos de junção estão destacados com retângulos tracejados no quadro rotulado com a letra (c). Quando há uma chamada a algum método *set** da classe *Line* ou da classe *Point*, o adendo *after* executa o método *updating()* da classe *Display* após o término da execução desses métodos chamados.

O símbolo | | entre os dois pontos de junção destacados com retângulo significa um operador *or*. A declaração *returning* garante que a sugestão *after* só será executada se os pontos de junção do conjunto de junção *move* retornarem com sucesso de suas execuções.

Vale ressaltar que as linhas em negrito na Figura 1.6 (*Display.updating()*) devem ser retiradas quando esse aspecto for implementado.

```
aspect DisplayUpdating
{
    pointcut move():
       call(void Line.set*(Point)) || call(void Point.set*(int))
     after() returning: move() {
       Display.updating();
     }
}
```

Figura 1.10 – Implementação do Mecanismo de Atualização do *Display*

Dessa forma, as classes `Line` e `Point` conterão apenas os atributos e métodos pertinentes a suas funcionalidades, eliminando completamente chamadas ao método `updating()` da classe `Gerenciador`.

1.5.1.6 Propostas de modelagem para Programação Orientada a Aspectos

Pawlak *et al.* (Pawlak *et al*., 2002), propuseram um perfil UML para a programação orientada a aspectos utilizando o estereótipo <<*aspect*>> (para representar os aspectos) e <<*pointcut*>> (para representar aonde eles afetam).

Na Figura 1.11, pode-se ver um exemplo que utiliza a notação desses autores. Esse diagrama ilustra um simples sistema de venda de produtos. As classes Cliente e Produto formam a aplicação e *RastreiaOperação* e *Rastrear* formam o comportamento transversal.

O aspecto *Rastrear*, afeta as classes *Cliente* e *Produto*. Isso pode ser observado pelo relacionamento de ligação que existe entre o aspecto e a classe *Cliente* e também pelo aspecto e a classe *Produto*. Pode-se notar que em ambos os relacionamentos é possível ver em anexo o estereótipo <<*pointcut*>> (no aspecto, o conjunto de junção foi definido como *methodToBeTraced())*. Pode-se notar também que há quatro métodos ao lado da classe *Cliente* e um método ao lado da classe *Produto*. São esses métodos que serão afetados pelo aspecto *Rastrear* ao serem chamados pelas respectivas classes (todos eles pertencem ao conjunto de junção methodToBeTraced(), portanto cada um deles é um ponto de junção).

Na classe *Cliente*, os quatro métodos são afetados no contexto de execução. Isso pode ser observado pelo símbolo "?". Caso o método seja afetado na chamada, ele deverá possuir o símbolo "!" antes do seu nome. Os métodos afetados são: *?Cliente.new(..)*, *?adicionarProduto(..)*, *?comprarProduto(..)* e *?removerProduto(..)*. Toda vez que algum desses métodos for chamado pela classe Cliente, antes do método começar a sua execução (visualize o *<<before>>methodToBetraced( ))*, o aspecto chama o método *beginningMethod()* que está na classe *RastreiaOperação* para ser executado. Após a execução do *beginningMethod()*, o método afetado continua executando normalmente. Quando o método afetado estiver para terminar a sua execução, o aspecto novamente chama a classe *RastreiaOperação* e o método *finishingMethod()* é executado. Após isso, o método afetado termina a sua execução.

No exemplo não existem atributos dentro do método *methodToBeTraced()*. Porém se existisse, nesse perfil proposto, antes do nome do atributo poderia haver um sinal de “>” ou “<” *(exemplo: >nome)* que seriam atributos de entrada e saída respectivamente.

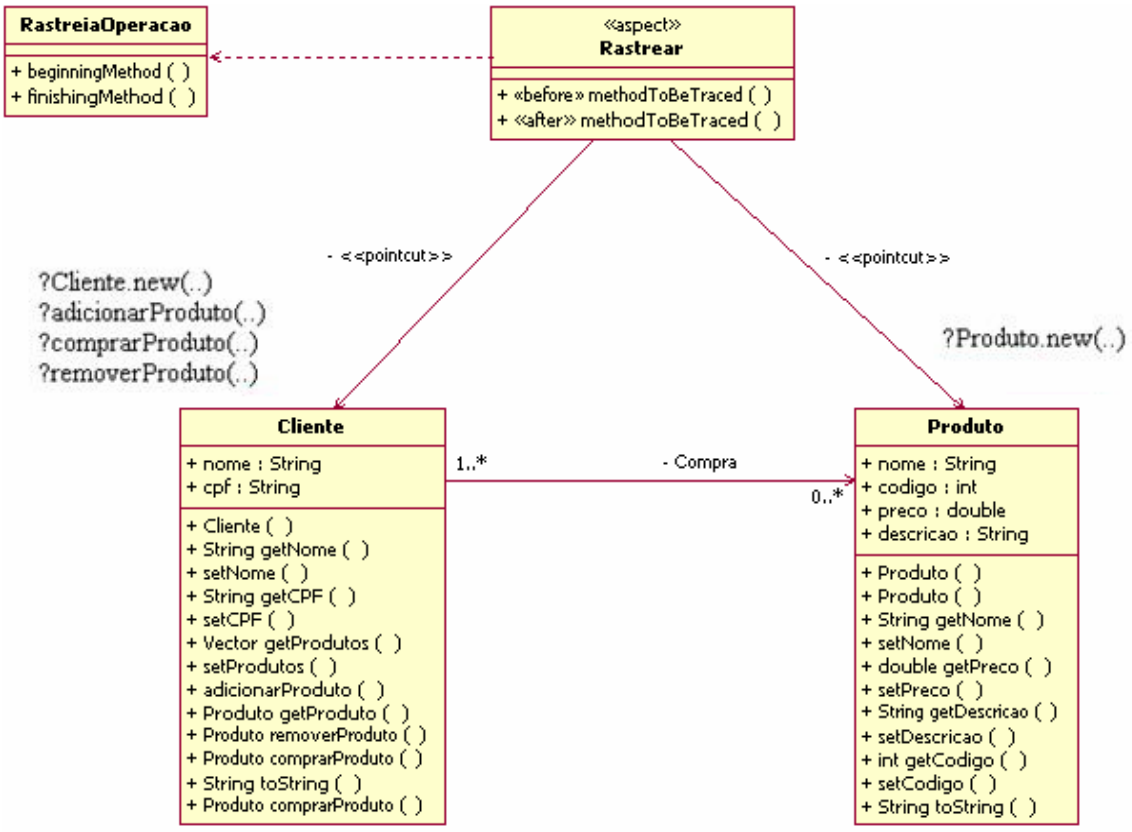

Figura 1.11 – Exemplo de uma modelagem utilizando <<*pointcut*>> e <<*aspect*>>.

1.6. Proposta de utilização da UMLintr e conceitos de POA para exemplificar um ataque *Smurf*

Esta seção descreve um exemplo de ataque *smurf* utilizando UMLintr e conceitos POA, destacando os benefícios aos programadores na modelagem de um cenário de ataque.

1.6.1 Um cenário de ataque *smurf* utilizando UMLintr e POA

A versão original do ataque smurf é baseada no ICMP. Este ataque consiste em fazer *spoofing*, ou seja, se fazer passar por um número IP – o da vítima, por exemplo, e enviar mensagens *echo request* para um endereço broadcast de uma ou várias sub-redes. As respostas dos equipamentos que estão escutando na rede à requisição para o IP da vítima causam a negação de serviço.

Após a divulgação do código original variantes deste ataque apareceram na Internet, inclusive uma baseada no protocolo UDP. No aniversário do lançamento do

smurf o autor lançou uma versão melhorada, com código mais limpo e abrangendo todas as variantes do ataque, sendo conhecida como *smurf5* ou *papasmurf*.

O ataque *smurf* funciona da seguinte forma:

1. O atacante cria uma lista de endereços *broadcast* de rede;
2. Pacotes *spofados* com o IP da vítima são enviados para cada uma das redes;
3. Todas as máquinas de cada rede que recebem os pacotes respondem, enviando pacotes para o computador da vítima e não do atacante.

Segundo Northcutt *et al.* (2002) *spoofing* significa endereçar um pacote com uma informação falsa de modo que pareça vir de um lugar diferente de onde realmente veio. Um pacote pode ser endereçado como se viesse de um *host* interno na rede de destino, de um dos intervalos de endereços privados ou mesmo de outra rede totalmente diferente.

A Figura 1.12 ilustra como é o comportamento do ataque *smurf*

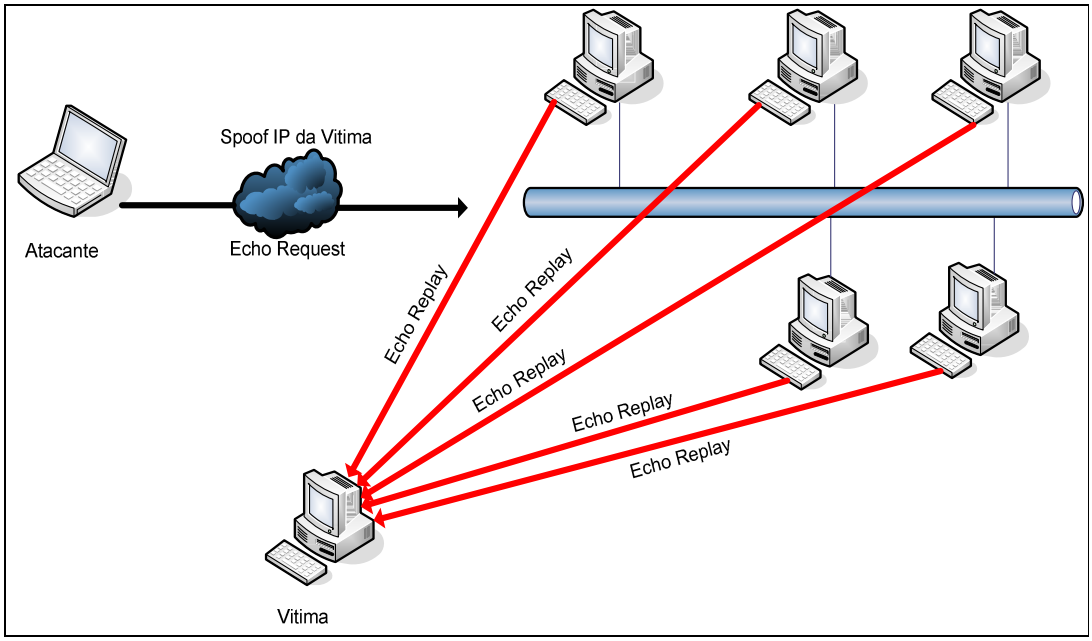

Figura 1.12 – Funcionamento do *Smurf*

Caso a rede onde seu computador está instalado seja vulnerável a ataques do tipo *smurf*, ou seja, é uma rede amplificada, basta efetuar um comando *ping* no *broadcast* (Figura 1.13) da rede e monitorar o resultado com o *TCPDump* (Figura 1.14), como mostra o exemplo a seguir.

```
root@notebook ~]$ ping -b 192.168.0.255
WARNING: pinging broadcast address
PING 192.168.0.255 (192.168.0.255) 56(84) bytes of data.
64 bytes from 192.168.0.7: icmp_seq=0 ttl=64 time=0.026 ms
64 bytes from 192.168.0.8: icmp_seq=0 ttl=255 time=0.163 ms (DUP!)
64 bytes from 192.168.0.9: icmp_seq=0 ttl=255 time=0.165 ms (DUP!)
64 bytes from 192.168.0.10: icmp_seq=0 ttl=255 time=0.167 ms (DUP!)
64 bytes from 192.168.0.11: icmp_seq=0 ttl=255 time=0.168 ms (DUP!)
64 bytes from 192.168.0.7: icmp_seq=1 ttl=64 time=0.032 ms
64 bytes from 192.168.0.8: icmp_seq=1 ttl=255 time=0.153 ms (DUP!)
64 bytes from 192.168.0.9: icmp_seq=2 ttl=255 time=0.155 ms (DUP!)
64 bytes from 192.168.0.10: icmp_seq=1 ttl=255 time=0.155 ms (DUP!)
64 bytes from 192.168.0.11: icmp_seq=1 ttl=255 time=0.157 ms (DUP!)
64 bytes from 192.168.0.7: icmp_seq=2 ttl=64 time=0.031 ms
64 bytes from 192.168.0.8: icmp_seq=2 ttl=255 time=0.169 ms (DUP!)
64 bytes from 192.168.0.9: icmp_seq=2 ttl=255 time=0.170 ms (DUP!)
64 bytes from 192.168.0.10: icmp_seq=2 ttl=255 time=0.172 ms (DUP!)
64 bytes from 192.168.0.11: icmp_seq=2 ttl=255 time=0.172 ms (DUP!)
```

Figura 1.13 – Exemplo de *ping* para *broadcast*

```
[root@notebook ~]# tcpdump -i eth0 | grep echo
tcpdump: verbose output suppressed, use -v or -vv for full protocol
decode
listening on eth0, link-type EN10MB (Ethernet), capture size 96 bytes
17:30:00 IP notebook.br > 192.168.0.255: icmp 64: echo request seq 0
17:30:00 IP 192.168.0.8 > notebook.br: icmp 64: echo reply seq 0
17:30:00 IP 192.168.0.9 > notebook.br: icmp 64: echo reply seq 0
17:30:00 IP 192.168.0.10 > notebook.br: icmp 64: echo reply seq 0
17:30:00 IP 192.168.0.11 > notebook.br: icmp 64: echo reply seq 0
17:30:01 IP notebook.br > 192.168.0.255: icmp 64: echo request seq 1
17:30:01 IP 192.168.0.8 > notebook.br: icmp 64: echo reply seq 1
17:30:01 IP 192.168.0.9 > notebook.br: icmp 64: echo reply seq 1
17:30:01 IP 192.168.0.10 > notebook.br: icmp 64: echo reply seq 1
17:30:01 IP 192.168.0.11 > notebook.br: icmp 64: echo reply seq 1
17:30:02 IP notebook.br > 192.168.0.255: icmp 64: echo request seq 2
17:30:02 IP 192.168.0.8 > notebook.br: icmp 64: echo reply seq 2
17:30:02 IP 192.168.0.9 > notebook.br: icmp 64: echo reply seq 2
17:30:02 IP 192.168.0.10 > notebook.br: icmp 64: echo reply seq 2
17:30:02 IP 192.168.0.11 > notebook.br: icmp 64: echo reply seq 2
```

Figura 1.14 – Exemplo de *tcpdum* gerado pelo *ping* para *broadcast*

No *site http://www.powertech.no/smurf/* é possível verificar uma lista de redes amplificadas (*Smurf Amplifier Registry - SAR*). Segundo este *site*, 447.685 redes foram verificadas e listadas no SAR, e atualmente 230 delas estão sem acesso, enquanto que 193.589 tiveram sua vulnerabilidade corrigida após serem listadas pelo site.

O ataque pode ser facilmente evitado, por exemplo, configurando o *firewall* da rede para não responder requisições ICMP. Entretanto, vários serviços de *Internet* utilizam portas UDP, como exemplo o DNS (*Domain Name System*), na qual um ataque com a variação do *smurf*, o *papasmurf*, pode ser utilizado. Nesse caso, o atacante tem que saber o número IP da vítima ou definir quais portas serão atacadas.

O atacante pode se valer de outros *softwares* como um *port scanner* para descobrir quais portas da vítima estão potencialmente suscetíveis a um ataque. Um *port scanner* pode varrer as portas de um *host* ou um *range* de IP, sendo possível realizar a

varredura em portas TCP ou UDP. Para este trabalho se restringe às varreduras de portas UDP, utilizadas pelo *papasmurf*.

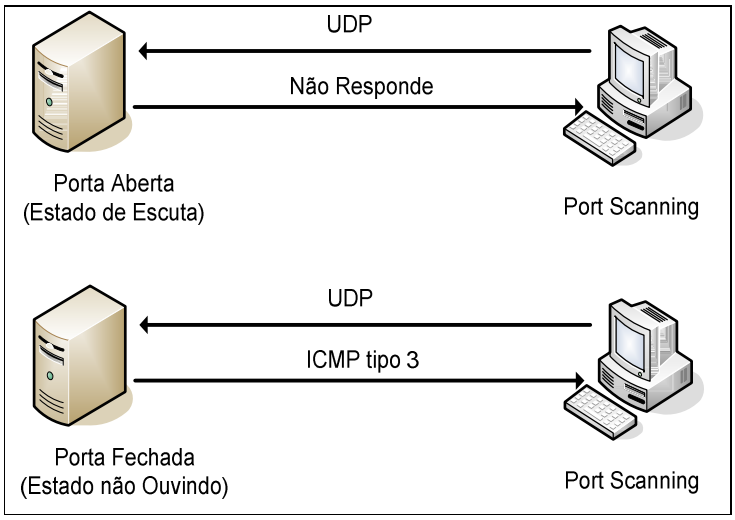

Figura 1.15 – Funcionamento do *port scanning* – Portas UDP

A Figura 1.15 exemplifica uma varredura de portas UDP. Durante a varredura, quando são enviados pacotes UDP e o *host* está ouvindo (existe um serviço ativo naquela porta) não é recebido nenhum tipo de resposta, significando que a porta está aberta. Caso contrário, o *port* scanner recebe um retorno com pacote ICMP tipo 3 (*Destination Unreachable*) significando que a porta está fechada. Dessa forma é possível a detecção da vulnerabilidade do *host* atacado.

A Figura 1.16 apresenta um exemplo do resultado de um *port scanner* UDP.

```
root@notebook scan]# ./pscan -u 200.2XX.XXX.XXX
scanning host 200.2XX.XXX.XXX's udp ports 1 through 10240
port 53 (domain) open
port 68 (bootpc) open
port 111 (sunrpc) open
port 631 (ipp) open
port 940 (UNKNOWN) open
port 943 (UNKNOWN) open
port 5353 (mdns) open
```

Figura 1.16 – *Port scanning* UDP utilizando o *pscan* em um ambiente *Linux*.

Com a utilização do *port scanning* e do ataque *papasmurf*, o atacante pode realizar um ataque com maior eficiência.

A UMLintr, por sua vez, propõe uma estrutura que contempla estes dois tipos de ataques, desde que utilizados separadamente. Na proposta aqui apresentada sara utilizada a UMLintr estendida para dar suporte a POA e para que possa ser utilizada para outros tipos de ataques mais eficientes.

A proposta da UMLintr cita que com os diagramas de classe é possível gerar o diagrama de máquinas de estados, sendo este último utilizado por algumas linguagens

de ataque, tal como STATL (Eckmann *et al.*, 2002), para conceber cenários que permitam a geração de assinaturas de ataques para serem utilizadas em ferramentas IDS.

Com a utilização de dois ataques integrados, o ataque funciona da seguinte forma (Figura 1.18):

1. O atacante dispara um ataque *port scanning*, classe ***Nov_attacker***, o qual pode ser para um host classe ***TCP_Protocol*** ou para um *range* de IP. Pode-se inclusive definir o *range* de portas UDP a serem varridas;
2. O *port scanner*, representado pela classe **PortScan**, verifica os *hosts* e portas que são potenciais vítimas de um ataque por meio do método *isReplayValid*. Para cada conjunto IP/Porta pode ser disparado um ataque papasmurf;
3. De posse do IP e da porta, o *papasmurf*, classe ***PapaSmurf***, realiza o ataque DoS contra a vitima, representada pela classe ***TPC_Protocol***;

Como mencionado anteriormente, estes dois ataques existem separadamente, porém, para utilizá-los em conjunto, os *softwares*, escritos em C++, teriam que ter uma dependência, a classe *PortScan* depende da classe *PapaSmurf*. Com a utilização de POA a dependência é isolada em um aspecto, fazendo com que essa dependência não fique "explicitamente" codificada no código. Este fato pode ser observado no diagrama pelo aspecto ***AutomatizaAtaque*** (Figura 1.18). Esse diagrama está sendo modelado de acordo com a proposta de Pawlak *et al*. (2002)

O objetivo deste aspecto é entrecortar a execução do método *isReplayValid()*, capturar informações de contexto e utilizar a classe *PapaSmurf*. Dessa forma, não se cria uma dependência explícita da classe *PortScan* com a classe *PapaSmurf*, como seria a implementação tradicional. O isolamento dessa dependência permite que a classe *PortScan* seja utilizada em outros contextos em que a classe *PapaSmurf* não seja necessária. Além disso, evita os problemas de entrelaçamento e espalhamento de código.

Outro ponto importante da decisão de projeto adotada aqui é não colocar no aspecto todo o código do *PapaSmurf*. Assim, também não se cria dependência desse módulo com *PortScan*. Essa estratégia consiste em considerar o interesse transversal como sendo composto pela classe *PapaSmurf* e pelo aspecto *AutomatizaAtaque*.

O código do aspecto *AutomatizaAtaque* pode ser observado na Figura 1.17.

Na Figura 1.17 é mostrado um aspecto abstrato chamado *AutomatizaAtaque*. Um aspecto abstrato não tem conhecimento dos pontos de junção que irão ser afetados mas já possui comportamento definido em seus adendos. A idéia principal de um aspecto abstrato como esse é propiciar bons níveis de reúso, já que os detalhes do código-base em que ele será acoplado ainda não são fornecidos.

Na Figura 1.18 é mostrado um aspecto concreto chamado *AutomatizaAtaquePapaSmurf* que estende o aspecto abstrato *AutomatizaAtaque* e que concretiza o conjunto de junção abstrato *ataca* com pontos de junção do código-base que deve ser afetado. A criação desse aspecto concreto pode ser vista como a criação de um conector entre o código-base e o interesse transversal.

```
public abstract aspect AutomatizaAtaque {
public abstract pointcut ataca();
after():ataca(){
   ... chamada a métodos da classe papasmurf com os parâmetros do ataque ...
   }
}
```

Figura 1.17 – Código do aspecto abstrato

```
public aspect AutomatizaAtaquePapaSmurf extends AutomatizaAtaque {
public pointcut ataca(): execution (* PortScan.isReplayValid(true));
```

Figura 1.18 – Código do aspecto concreto ilustrando o conjunto de pontos de junção.

Partindo da observação deste tipo de ataque, percebe-se que a utilização de conceitos de POA traz benefícios na modelagem, uma vez que se pode acoplar e desacoplar facilmente outros tipos de ataque a um *port scanning*, por exemplo, sem comprometer a utilização destes com ataques individuais.

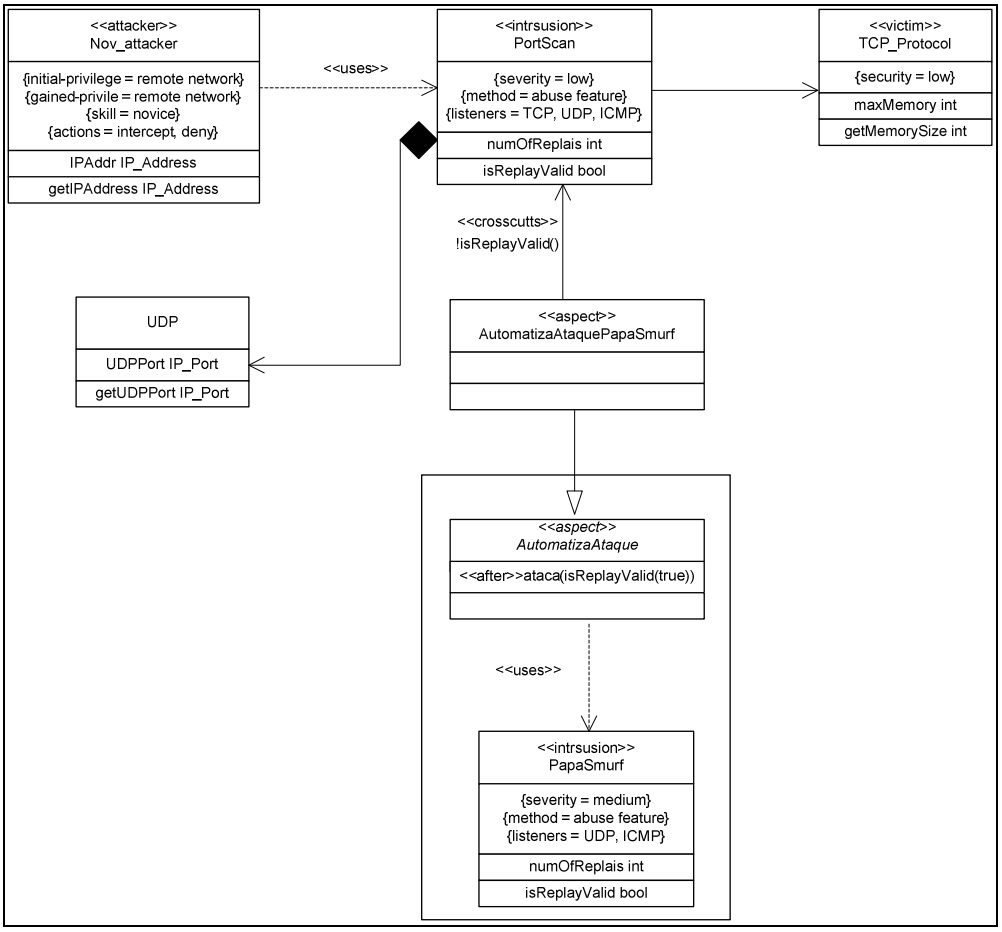

Figura 1.19 – Diagrama de Classe Representando o *Port Scanning* e o *PapaSmurf*

Na Figura 1.20 é mostrada a classe *PortScan* escrita em C++, destacando o método *isReplayValid()* que será entrecortado pelo aspecto *AutomatizaSmurf*.

```
…

class PortScan {

…

bool PortScan::isReplayValid(void)
{
  return ataqueValido;
}

…

bool PortScan::scan(PortScan scan)
{
…

      if ((rc < 0) && (scan.get_scan_type() == 2))
        continue;
      if (scan.get_scan_type() == 1)
         serv = getservbyport(htons(port),"tcp");
      else if (scan.get_scan_type() == 2) {
         serv = getservbyport(htons(port),"udp");
         rc = 0;        /* fake out below msg */
      }
      else
        {
          scan.set_ataqueValido(false);
          return false;
        }
     scan.set_ataqueValido(true);
     if (scan.isReplayValid())
        scan.exibir(port, serv, rc);
   }
   scan.set_ataqueValido(true);
   return true;
}
void PortScan::exibir(int port, struct servent *serv, int rc)
{
      fprintf(stdout,"port %d (%s) %s\n", port,
        (serv == NULL)?"UNKNOWN": serv->s_name,
        (rc == 0) ? "open" : strerror(errno));

}
```

Figura 1.20 – Fragmentos da Classe *PortScan*.cpp escrita em C++, baseada no programa *pscan.c*

Na Figura 1.21 é mostrada a Classe *PapaSmurf* que será instaciada dentro do adendo do aspecto *AutomatizaSmurf.* Essa instância será utilizada para invocar métodos dessa classe e efetivamente promover o ataque.

```
…
class PapaSmurf
{
…
void PapaSmurf::PapaSmurfUDP (struct PapaSmurf *sm, u_long dst, int n)
{   struct iphdr *ip;
    struct udphdr *udp;
    char *packet, *data;
    int pktsize = sizeof(struct iphdr) + sizeof(struct udphdr) + sm-
>psize;
    packet = (char *) malloc(pktsize);
    ip = (struct iphdr *) packet;
    udp = (struct udphdr *) (packet + sizeof(struct iphdr));
    data = (char *) (packet + sizeof(struct iphdr) + sizeof(struct
udphdr));
    memset(packet, 0, pktsize);
    if (*sm->padding)
        memcpy((char *)data, sm->padding, sm->psize);
    /* fill in IP header */
    ip->version = 4;
    ip->ihl = 5;
    ip->tos = 0;
    ip->tot_len = htons(pktsize);
    ip->id = htons(getpid());
    ip->frag_off = 0;
    ip->ttl = 255;
    ip->protocol = IPPROTO_UDP;
    ip->check = 0;
    ip->saddr = sm->sin.sin_addr.s_addr;
    ip->daddr = dst;
    /* fill in UDP header */
    if (sm->srcport) udp->source = htons(sm->srcport);
    else udp->source = htons(rand());
    if (sm->rnd) udp->dest = htons(rand());
    else udp->dest = htons(sm->dstport[n]);
    udp->len = htons(sizeof(struct udphdr) + sm->psize);
    udp->check = in_chksum((u_short *)udp, sizeof(udp));
    /* send it on its way */
    if (sendto(sm->s, packet, pktsize, 0, (struct sockaddr *) &sm-
>sin,    sizeof(struct sockaddr)) == -1)
    {perror("sendto()"); exit(-1); }
    free(packet);/* free willy! */
}
u_short PapaSmurf::in_chksum (u_short *addr, int len)
{
    register int nleft = len;
    register u_short *w = addr;
    register int sum = 0;
    u_short answer = 0;
    while (nleft > 1)
    {  sum += *w++;
        nleft -= 2;
    }
    if (nleft == 1)
    { *(u_char *)(&answer) = *(u_char *)w;
        sum += answer;    }
    sum = (sum >> 16) + (sum + 0xffff);    sum += (sum >> 16);
 answer = ~sum;
    return(answer);}
…
```

Figura 1.21 – Método *PapaSmurfUDP* da Classe *PapaSmurf.cpp* escrita em C++, baseada no program *papasmurf.c*

Os exemplos mostrados acima consistem em uma primeira iniciativa no sentido de averiguar as vantagens e desvantagens da utilização da POA na modularização de ataques, considerando esses como interesses transversais. Na iniciativa mostrada parte-se da modelagem do sistema e pretende-se dar continuidade ao longo das demais etapas do ciclo de vida de desenvolvimento de um sistema.

Pode-se também concluir que é possível efetuar a modelagem de ataques utilizando tanto os conceitos de UML quanto de POA.

A continuidade deste trabalho será baseada nos trabalhos de: Eckmann *et. al.* (Eckmann *et al.*, 2002) que propõem a utilização da STATL, uma linguagem de ataque para gerar assinaturas de ataques baseadas em máquinas de estado e; Hussein e Zulkernine (2006) propõem uma extensão da UML para intrusão, e com isso a possibilidade de sua utilização para representar diagramas de máquinas de estados. Sendo assim, este trabalho constitui o início para o desenvolvimento de um compilador que possa gerar código com a finalidade de ser importado pela STATL.

Este modelo poderá auxiliar o desenvolvedor permitindo a realização do desenvolvimento de forma mais rápida, uma vez que não haverá a necessidade deste conhecer profundamente as linguagens de ataque e mesmo assim modelar um ataque com a utilização da UML gerando suas assinaturas.

A utilização de conceitos de POA contribui no sentido de permitir que sejam modelados em classes os vários tipos de ataques de maneira independente e, conforme a funcionalidade de cada ataque seja possível a utilização destas classes previamente definidas e em várias combinações diferentes, podendo gerar assinaturas de ataque com diversos comportamentos diferentes e que permitam a detecção mais eficiente por parte de um IDS que as possua.

1.7. Considerações Finais

Devido à facilidade com que ferramentas automáticas de invasão podem ser encontradas, invadir um sistema computacional requer cada vez menos habilidades dos intrusos. Enquanto os intrusos mais experientes estão se tornando mais eficientes, o conhecimento requerido por parte de intrusos novatos para copiar e iniciar ataques está diminuindo (Allen *et al.*, 2000). Com isso, cada vez mais sistemas computacionais são invadidos a cada dia.

Para definir a segurança de um sistema computacional é importante compreender todo o ambiente em que o sistema está inserido e não somente as tecnologias particulares. A segurança em si é um sistema interconectado (uma cadeia), no qual diferentes componentes interagem trocando informações. Se uma das partes está fraca (pouco segura) todo o sistema está comprometido. Assim, segurança é um processo e não um produto. Sendo um processo, a segurança deve permear todo o sistema: seus componentes, tecnologias, níveis de complexidade e interações (Schneier, 2000). Assim, diferentes ferramentas podem e devem ser combinadas para garantir uma melhor segurança aos sistemas computacionais.

O constante aumento no investimento em segurança torna ainda mais desafiante o desenvolvimento de ferramentas e algoritmos que permitam a execução de ataques cada vez mais eficientes. Isso faz com que o desenvolvimento de novas ferramentas e a

utilização de novas metodologias se torne necessária para a evolução das ferramentas que detectam e que impeçam a ação de ataques maliciosos.

A POA, aliada a extensão da UML, a UMLintr, permite que assinaturas de ataques mais robustas e eficientes sejam geradas de forma mais intuitiva e fácil, fazendo com que essas possam ser inseridas nos IDS e esses, por sua vez, possam prover uma menor vulnerabilidade e uma maior eficiência na detecção e refuta de ataques.

Acredita-se que no futuro o DDoS tende a ser mais forte do que é hoje, desse modo o número de ferramentas para sua detecção devem crescer também. Assim, ao mesmo tempo em que novos ataques são feitos para tentar superar as defesas existentes, novas defesas têm muitos entraves para repelir esses ataques, e a POA visa auxiliar na superação desses entraves.

Este trabalho apresenta apenas um exemplo simples da especificação de um ataque em UMLintr estendida para aspectos. Espera-se como trabalhos futuros efetuar a modelagem de outros ataques como DDoS de modo a gerar a especificação e a assinatura desses ataques e prover uma interface amigável para que novas e diferentes assinaturas de ataque possam ser geradas incorporadas pela STATL, além de efetuar a modelagem de outros ataques como DDoS de modo a gerar a especificação e a assinatura desses ataques.

Referências

Allen, J.; Christie, A.; Fithen, W.; McHugh, J.; Pickel, J.; Stoner, E. (2000). *State of the practice of intrusion detection technologies*. Technical Report. Software Engineering Institute, Carnegie Mellon University, TR CMU/SEI-99-TR-028, ESC-99-028.

Ambrósio, D. R. (2002). *Métodos alternativos de reconhecimento de padrões para sistemas de detecção de intrusão*. Dissertação de Mestrado, Instituto de Ciências Matemáticas e de Computação - ICMC, Universidade de São Paulo - USP, São Carlos - São Paulo.

Balasubramaniyan, J. S.; Fernandez, J. O. G.; Isacoff, D.; Spafford, E.; Zamboni, D. (1998). *An architecture for intrusion detection using autonomous agents*. Technical Report, Department of Computer Sciences, Purdue University, COAST Laboratory TR 98/05.

Booch, G., Rumbaugh, J., Jacobson, I. (2000). UML – *Guia do usuário*. Rio de Janeiro. Editora Campus.

Cert.br (2006). *Cartilha de segurança para Internet*. Disponível em <http://cartilha.cert.br/malware/>. Acessado em julho de 2007.

Dijkstra, E. W (1976). *A Discipline of Programming*. Prentice-Hall.

Eckmann, S. T.; Vigna, G.; Kemmerer, R. A. (2002). STATL: An Attack Language for State-based Intrusion Detection. Journal of Computer Security, vol. 10, no. 1/2, pp. 71-104

Elrad, T., Filman R., Bader A. (2001). *Aspect-Oriented Programming*. Communications of the ACM, vol 44, pp 29-32.

Fowler, M. (2005). *UML essencial: um breve guia para a linguagem padrão de modelagem de objetos*. Bookman. 3ª Edição. Porto Alegre.

Gradecki, J.D. And Lesiecki, N. (2003). *Mastering AspectJ – Aspect Oriented Programming in Java*. Wiley Publishing.

Griswold, W.G., Shonle, M., Sullivan, K., Song, Y., Cai, Y., Rajan, H. (2006). *Modular Software Design with Crosscutting Interfaces*, IEEE Software, pp 51-60.

Heady, R.; Luger, G.; Maccabe, A., Servila, M. (1990). *The Architecture of a Network Level Intrusion Detection System*, Technical Report - Department of Computer Science, University of New Mexico, USA.

Hussein M.,Zulkernine M. (2006). *UMLintr: A UML for Specifying Intrusion*. 13th Annual IEEE International Symposium and Workshop on Engineering Based Systems.

Jacobson, I., Booch, G., Rumbaugh, J. (1999). *The Unified Software Development Process*. Reading, MA. Addison-Wesley.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, G. (2001). *Getting Started With AspectJ,* Communications of the ACM, vol 44, No. 10, pp.59-65.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., Irving, J. 1997). *Aspect Oriented Programming*. In: Proceedings of 11 ECOOP. pp. 220-242.

Kiczales, G., Mezini, M. (2005) *Aspect-Oriented Programming and Modular Reasoning*. In: Proceedings of International Conference on Software Engineering (ICSE'05), St. Louis, Missouri, USA, pp. 49-58.

Kurose, J. F.; Ross, K. W. (2005) *Redes de computadores e a Internet: uma abordagem top-down*. Pearson-Addison Wesley. 3ª edição. São Paulo.

Li, W.; Zhi-tang L.; Yao L. (2006) *A novel algorithm SF for mining attack scenarios model*. Proceedings of the International Conference on e-Business Engineering. pp. 55-61.

Monteiro, E.. (2004). *Um Estudo Sobre Modelagem Orientada a Aspectos Baseada em AspectJ e UML*. Monografia apresentada ao Centro Universitário Luterano de Palmas.

MOURA, José Antão Beltrão; et al. (1999). *Redes de computadores : serviços, administração e segurança*. São Paulo: Makron Books, 493p.

Nagamalai D.; Dhinakaran C.; Lee J. K. (2007). *Multi Layer Approach to Defend DDoS Attacks Caused by Spam*. Proceedings of the International Conference on Multimedia and Ubiquitous Engineering. pp. 97-102.

Pawlak, R. Duchien, L. Florin, G. Legond-Aubry, F., Senturier, L., Martelli, L. (2002). *A UML Notation for Aspect Oriented Software Design*.. França.

Raihan M. F.; Zulkernine M. (2005) *Detecting Intrusions Sepecified in a Software Specification Language*. Proceedings of the 29th Annual Internation Computer Software and Applications Conference. pp. 143-148.

Schneier, B.(2000). *Secrets & lies - digital security in a networked world*. John Wiley & Sons, 412 p.

Silva, A.R. (2002). *Um modelo representativo de assinaturas de ataque para sistemas detectores de intrusão*. Ttrabalho de conclusão de curso do Departamento de Ciências e Computação e Estatística do Instituto de Biociências, Letras e Ciências Exatas, São José do Rio Preto.

Stein, Dominik (2002). *An Aspect-Oriented Design Model Based on AspectJ and UML*. Dissertação de Mestrado (Mestrado em Gerenciamento de Sistemas de Informação) – Universidade de Essen, Germany.

Tar, P.; Osser, H.; Sutton, S. (2002). Hyper/JTM: Multi-domensional Separation of Concerns for Java. In: Proc. of the 24th Internation Conference on Software Engineering. Orlando, Florida

Xiang, Y.; Zhou, W.; Li Z. (2006). *An Analytical Model for DDoS Attacks and Defense*. Proceedings of the International Multi-Conference on Computing in the Global Information Technology. pp.66.