PONTIFÍCIA UNIVERSIDADE CATÓLICA DO PARANÁ
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA
MESTRADO EM CIÊNCIA DA COMPUTAÇÃO

THIAGO MATTOS ROSA

UMA ONTOLOGIA PARA SISTEMA DE DETECÇÃO DE INTRUSÃO EM WEB SERVICES

CURITIBA
2011

THIAGO MATTOS ROSA

UMA ONTOLOGIA PARA SISTEMA DE DETECÇÃO DE INTRUSÃO EM WEB SERVICES

Dissertação apresentada ao Programa de Pós-Graduação em Informática, da Pontifícia Universidade Católica do Paraná, como requisito parcial à obtenção do título de Mestre.

Orientador: Prof. Dr. Altair O. Santin

Co-orientadora: Profª. Drª. Andreia Malucelli

CURITIBA

2011

Dedico este trabalho aos meus amigos
que estiveram sempre presentes em
minha vida e aos meus pais e familiares
que sempre me apoiaram e aconselharam
ao longo de minhas conquistas.

AGRADECIMENTOS

Agradeço a todos que de alguma forma contribuíram para que eu alcançasse meus objetivos nos últimos dois anos.

Agradeço ao meu orientador Altair O. Santin por sempre me manter no caminho certo durante todas as etapas desta longa jornada. Agradeço também à minha co-orientadora Andreia Malucelli por sanar minhas dúvidas e prover ajuda sempre que necessário.

RESUMO

As tecnologias utilizadas por *web services* trazem vulnerabilidades conhecidas para esse novo ambiente. Sistemas de detecção de intrusão (IDS) clássicos, baseados em assinaturas e anomalias, podem ser utilizados para mitigar ataques contra *web services*. IDS baseados em anomalias tentam deduzir um ataque baseados em perfis de ataques conhecidos, mas quando erram geram falsos positivos. Já os baseados em assinaturas erram menos, porém só detectam ataques que já estejam em sua base ataques, propiciando ataques *zero-day*. Ataques *zero-day* introduzem altos níveis de exposição ao risco, pois acontecem quando uma vulnerabilidade de software se torna pública antes que sua detecção seja possível. Este trabalho propõe o uso de uma ontologia para construir uma base de conhecimento de ataques baseada em estratégia para o IDS. Ataques conhecidos são representados por instâncias na base de conhecimento da ontologia, que são detectados com baixas taxas de falsos positivos como na abordagem baseada em assinaturas. E variações de ataques podem ser deduzidas e derivadas por uma máquina de inferência que se baseia em classes e axiomas pré-definidos na ontologia, o que pode ser comparado com a habilidade da abordagem baseada em anomalias de detectar ataques desconhecidos. Assim, o IDS proposto adota uma abordagem híbrida de detecção, agregando as principais vantagens das abordagens clássicas baseadas em assinaturas e anomalias. Além disto, a abordagem é capaz de mitigar ataques *zero-day* para estas variações de ataques, já que as mesmas são automaticamente adicionadas como instâncias na base de conhecimento da ontologia logo após serem inferidas.

Palavras-chave: Ataque Zero-day, Ontologia, Sistema de Detecção de Intrusão, Web Services.

ABSTRACT

The underlying technologies used by web services bring known vulnerabilities to a new environment. The classical intrusion detection systems (IDS), signature and anomaly-based, can be used to mitigate attacks against web services. Anomaly-based IDS try to infer attacks based on known attack profiles, but when they are wrong they generate false positives. Signature-based IDS are much less often wrong, however, they only detect attacks that are already in the database, allowing for zero-day attacks. Zero-day attacks introduce high risk exposure levels, as they happen when a software flaw is publicly disclosed before it can be detected, due to its fix not being available in the signature database. This work proposes using an ontology to build a strategy-based knowledge attack database to assist intrusion detection systems. Known attacks are represented by instances in the ontology's knowledge database, which are detected with low false positive rates just like in a signature-based detection approach. And attack variations can be detected by a reasoner based on classes and axioms that are pre-set in the ontology, which can be compared to an anomaly-based approach's ability to detect unknown attacks. Thus, the proposed IDS adopts a hybrid intrusion detection approach, bringing together the main advantages of signature and anomaly-based classical approaches. Moreover, this hybrid approach is capable of mitigating zero-day attacks for these attack variations, since they are automatically added as instances to the ontology's knowledge database right after being inferred.

Key-words: Intrusion Detection System, Ontology, Web Services, Zero-day Attack.

LISTA DE TABELAS E ILUSTRAÇÕES

LISTA DE ABREVIATURAS E SIGLAS

ACK - *Acknowledgement code*
API - *Application Programming Interface*
CAPEC - *Common Attack Pattern Enumeration and Classification*
DAML - *DARPA Agent Markup Language*
DIG - *DL Implementation Group*
DoS - *Denial of Service*
F/IDS - *Firewall*/IDS
FTP - *File Transfer Protocol*
HIDS - *Host-based Intrusion Detection* System
HTTP - *Hypertext Transfer Protocol*
IDS - *Intrusion Detection System* (Sistema de Detecção de Intrusão)
IP - *Internet Protocol*
NIDS - *Network-based Intrusion Detection System*
OIL - *Ontology Inference Layer*
OWASP - *Open Web Application Security Project*
OWL - *Web Ontology Language*
PKI - *Public Key Infrastructure*
RDF - *Resource Description Framework*
SMTP - *Simple Mail Transfer Protocol*
SOA - *Service Oriented Architecture*
SOAP - *Simple Object Access Protocol*
SPARQL - *SPARQL Protocol and RDF Query Language*
SQL - *Structured Query Language*
SSL - *Secure Sockets Layer*
TCP - *Transmission Control Protocol*
UBR - *UDDI Business Registry*
UDDI - *Universal Description, Discovery and Integration*
UDP - *User Datagram Protocol*
VPN - *Virtual Private Network*
W3C - *World Wide Web Consortium*
WSA - *Web Services Architecture*

WSDL - *Web Services Description Language*

XML - *Extensible Markup Language*

SUMÁRIO

1 INTRODUÇÃO

1.1 CONTEXTUALIZAÇÃO E MOTIVAÇÃO

Web services vêm crescentemente sendo utilizados com a computação distribuída na internet porque permitem interoperabilidade entre aplicações web, que podem estar sendo executadas em plataformas e *frameworks* diferentes (BOOTH et al, 2004). *Web services* têm uma interface descrita em um formato processável por computadores, mais especificamente *Web Services Description Language* (WSDL).

Outros sistemas interagem com *web services* utilizando mensagens SOAP (*Simple Object Access Protocol*), por exemplo, que são transmitidas utilizando HTTP (*Hypertext Transfer Protocol*) como transporte e XML (*Extensible Markup Language*) para serialização. Estas tecnologias utilizadas por w*eb services* podem trazer vulnerabilidades aos mesmos, permitindo que ataques conhecidos sejam possíveis neste novo ambiente.

Uma das formas de se proteger w*eb services* destes ataques é utilizar um IDS (*Intrusion Detection System* - Sistema de Detecção de Intrusão). A maioria dos IDSs consegue identificar ataques através de detecção baseada em assinaturas ou de detecção baseada em anomalias (SCARFONE e MELL, 2007, p. 17).

Na detecção baseada em assinaturas, um conjunto de “padrões de ataque” (assinaturas) é previamente cadastrado em uma base que é usada pelo IDS para detecção de ataques conhecidos. A principal vantagem desta abordagem é que são produzidos poucos alarmes falsos (falsos positivos), que identificam um acesso autêntico como sendo um ataque. Porém, a desvantagem é que variações de um mesmo ataque não são detectadas, o que demanda constante atualização da base de assinaturas.

Na detecção baseada em anomalias o IDS utiliza uma base com o perfil de

uso normal do sistema sendo monitorado, e tudo o que desviar consideravelmente deste perfil será alertado como uma intrusão. A vantagem da detecção por anomalias é que ataques desconhecidos podem ser detectados, já que a detecção é baseada em perfis de “comportamentos”. Porém, nesta abordagem é comum a produção de um grande número de falsos positivos, devido à ampla gama de ações imprevisíveis (não modeladas) de usuários do sistema.

A técnica de detecção baseada em assinaturas é bastante utilizada, porém permite ataques *zero-day*, que ocorrem quando uma vulnerabilidade (falha de software) se torna publicamente conhecida antes que sua correção esteja disponível para ser inserida na base de assinaturas (GORELIK, 2007, p. 26). Independentemente de como uma vulnerabilidade se torna conhecida, a efetividade de um ataque *zero-day* pode variar de horas até meses (ZERO DAY INITIATIVE, 2011).

A capacidade de detectar novos ataques dá grande vantagem ao IDS baseado em anomalias. Porém, o alto número de falsos positivos desta abordagem, em relação ao IDS baseado em assinaturas, acaba fazendo com que os alertas deste IDS caiam em descrédito. Além disso, para se obter um bom modelo de perfis "normais" deve-se constantemente fazer coletas de dados para atualizar os mesmos.

Assim, o ideal seria que um IDS tivesse a capacidade de aprendizado da abordagem baseada em anomalias com a taxa de falsos positivos da abordagem baseada em assinaturas. Para isso, surge a hipótese de que uma ontologia poderia ser utilizada. A ontologia descreve conceitos e relacionamentos que são importantes para um determinado domínio, estabelecendo um vocabulário comum, além de prover uma especificação formal do significado dos termos utilizados neste vocabulário (NOY e MCGUINNESS, 2001, p. 1).

Aplicando ontologias ao problema de detecção de intrusão é possível representar os ataques e seus atributos, assim como expressar os relacionamentos entre os dados coletados, sendo possível utilizar axiomas para deduzir que uma determinada ação, ou sequência de ações, caracteriza um ataque (UNDERCOFFER et al, 2004, p. 48). Axiomas são declarações que envolvem conceitos e relacionamentos, e são expressos através de lógica de primeira ordem com o intuito de representar uma verdade reconhecida em um determinado domínio de conhecimento (GUARINO, 1998, p. 4).

Assim, substituindo-se a base de ataques/perfis dos IDSs clássicos por uma ontologia, pode-se obter uma forma dedutível de representação dos ataques, o que resultaria em um IDS híbrido contendo as principais vantagens de cada abordagem.

As principais vantagens de se utilizar ontologias para representar um domínio de conhecimento é o fato de as mesmas proverem uma semântica explícita e formal, serem reutilizáveis (adaptadas para contemplar outros objetivos) e poderem ser compartilhadas (entre vários sistemas escritos em linguagens diferentes). Ontologias também podem prover informações sobre um determinado domínio que não estejam explicitamente definidas na base de conhecimento, com o uso de inferência (KONSTANTINOU, SPANOS e MITROU, 2008, p. 6).

1.2 OBJETIVOS

O objetivo deste trabalho é modelar os ataques através de uma estratégia representada por classes e seus relacionamentos em uma ontologia. Acredita-se que conhecendo a estratégia de um ataque, que define o relacionamento semântico entre elementos do mesmo, pode-se facilmente detectar variações nos *payloads* e, como consequência, adicioná-los automaticamente à base de conhecimento da ontologia. Os objetivos específicos deste trabalho são, portanto:

a) Identificar e estudar um conjunto de ataques a *web services*;

b) Construir uma ontologia para os ataques resultantes do estudo anterior;

c) Propor um Sistema para Detecção de Intrusão considerando a ontologia;

d) Construir o protótipo do IDS proposto;

e) Avaliar o IDS proposto;

1.3 CONTRIBUIÇÃO

A contribuição deste trabalho consiste em prover uma abordagem de sistema de detecção de intrusão para *web services* baseada em estratégia que também ajude a mitigar ataques *zero-day* para as variações dos ataques contidos na ontologia.

Nesta abordagem ataques novos (desconhecidos) são derivados de estratégias conhecidas, neste caso com baixa taxa de falsos positivos na detecção. Para este propósito apresenta-se o sistema de detecção de intrusão baseado em estratégia como uma abordagem híbrida − suportando detecção baseada em anomalias, derivada da detecção baseada em assinaturas. A construção de uma ontologia baseada em uma taxonomia de ataques para *web services* também é considerada uma contribuição do trabalho.

1.4 ORGANIZAÇÃO DO TEXTO

O capítulo 2 deste documento apresenta a fundamentação teórica necessária para a compreensão do restante do trabalho. O capítulo 3 contempla os trabalhos relacionados, apresentando o estado da arte relevante ao tema. O capítulo 4 apresenta a proposta deste trabalho, e os resultados da sua avaliação são apresentados no capítulo 5. Por fim, no capítulo 6 o trabalho é concluído.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo serão abordados os seguintes temas: Sistemas de detecção de Intrusão (2.1), *Web services* (2.2), Ontologias (2.3), e Recursos (2.4) utilizados durante o desenvolvimento da proposta.

2.1 SISTEMAS DE DETECÇÃO DE INTRUSÃO

A detecção de intrusão é o processo de se monitorar os eventos ocorrendo em um sistema computacional ou em uma rede para identificar individuos que estejam utilizando estes recursos sem autorização, ou individuos que tenham acesso aos recursos mas que estejam excedendo seus privilégios (SNAPP et al, 1991, p. 167) Apesar de muitos incidentes de segurança serem maliciosos por natureza, muitos também não o são. Por exemplo, uma pessoa pode digitar errado o endereço de um computador e acidentalmente tentar conectar-se a um sistema sem autorização.

Um IDS é uma aplicação que automatiza este processo de monitorar eventos ocorrendo em um sistema ou rede, procurando por sinais de problemas de segurança (BACE, 2001, p. 5). IDSs estão primeiramente focados na identificação de possíveis incidentes, por exemplo, um IDS poderia detectar se um atacante conseguiu comprometer um sistema explorando uma vulnerabilidade no mesmo. Então, o IDS poderia reportar esse incidente aos administradores do sistema para que ações sejam tomadas.

As principais metodologias utilizadas pelos IDSs para detecção de intrusões são a detecção baseada em assinaturas e a detecção baseada em anomalias.

Uma assinatura é um padrão que corresponde a um ataque conhecido. A detecção baseada em assinaturas é o processo de se comparar assinaturas com

eventos observados para identificar possíveis ocorrências de incidentes (SCARFONE e MELL, 2007, p. 18). Um exemplo de assinatura pode ser um e-mail enviado com o assunto “Fotos!” e um arquivo anexo com o nome “Fotos.exe”, que são características de uma forma conhecida de *malware*.

A detecção baseada em assinaturas é muito eficiente para detectar ameaças conhecidas, mas muito ineficiente para detectar ameaças desconhecidas, ameaças disfarçadas por técnicas de evasão, e variações de ameaças conhecidas. Por exemplo, se um atacante modificasse o *malware* mencionado há pouco para utilizar um nome de arquivo “Fotos2.exe”, uma assinatura procurando por “Fotos.exe” não o identificaria. Detecção baseada em assinaturas é a metodologia mais simples de detecção de intrusão, pois apenas compara a atividade que está ocorrendo, como o envio de um pacote na rede, com uma lista de assinaturas através de operações de comparação de *strings*.

Detecção baseada em anomalias é o processo de se comparar definições de atividades consideradas normais com eventos observados, para identificar desvios significantes (SCARFONE e MELL, 2007, p. 18). Um IDS que usa detecção baseada em anomalias possui perfis que representam o comportamento normal, por exemplo, de usuários, *hosts*, conexões de rede ou aplicações. Esses perfis são obtidos monitorando-se as características das atividades normais durante um período de tempo.

Suponha-se que o perfil para uma rede pode mostrar que suas atividades consomem uma média de 13% da banda disponível para internet durante dias normais de trabalho. O IDS então usa métodos estatísticos para comparar as características da atividade corrente da rede com limites relacionados a esta métrica pertencente ao perfil da rede. Se alguma atividade consome mais banda do que o normal, o IDS pode alertar um administrador sobre a anomalia.

O maior benefício da detecção baseada em anomalias é que essa pode ser muito eficiente para detectar ameaças desconhecidas (SCARFONE e MELL, 2007, p. 18). Isto é, se um *malware* desconhecido infecta um computador, poderia consumir a capacidade de processamento, enviar vários e-mails, iniciar várias conexões e poderia ter vários outros comportamentos que desviariam bastante dos perfis normais para esse computador e, portanto, seria detectado.

Porém, um problema nessa metodologia de detecção de intrusão é que às vezes atividades maliciosas que ocorrem durante o treinamento dos perfis podem

ser incluídas como sendo atividades normais. Além disso, na detecção baseada em anomalias são gerados muitos falsos positivos por causa de atividades benignas que desviam muito do "perfil normal" de um sistema.

Uma variação da análise por anomalias é a análise de protocolos com monitoração de estado (*statefull*), processo que se utiliza de perfis de definições comumente aceitas como atividades de protocolo normais. Estes perfis (normais) são comparados com os eventos observados tentando identificar desvios. Diferentemente da detecção por anomalia, que usa perfis de rede ou de *host* específicos, a análise de protocolos com monitoração de estado utiliza perfis universais desenvolvidos por fabricantes, que especificam como determinados protocolos devem e não devem ser usados.

Por exemplo, quando um usuário inicia uma sessão de *File Transfer Protocol* (FTP), a sessão está em um estado não autenticado. Usuários não autenticados devem somente executar alguns comandos nesse estado. Depois que o usuário se autenticar a sessão passa para o estado autenticado e o usuário pode executar qualquer um de dezenas de comandos. Se o usuário executar a maioria destes comandos enquanto está no estado não autenticado, o IDS consideraria suspeito. A desvantagem no uso dessa análise é que ela consome muito recurso do sistema, por ser muito complexa ao ter que monitorar estados de muitas sessões simultâneas.

IDSs podem também ser classificados pelo tipo de tecnologia. Existem diversos tipos de tecnologia para IDSs, porém os mais conhecidos e utilizados são NIDS (*Network-based Intrusion Detection System*) e HIDS (*Host-based Intrusion Detection System*).

NIDS monitora o tráfego de uma rede ou de segmentos de uma rede e analisa suas atividades para identificar atividades suspeitas e vários tipos de eventos de interesse (SCARFONE e MELL, 2007, p. 20). Este tipo de IDS é geralmente instalado em uma fronteira entre redes, por exemplo, perto de firewalls ou roteadores, servidores de VPN (*Virtual Private Network*), servidores de acesso remoto e redes sem fio. NIDS é bastante útil para detectar acessos não autorizados de fora da rede (*hackers*), ataques de negação de serviço e aumentos significativos no consumo de banda. Porém, NIDS não é muito eficaz para detectar ataques locais em um *host*, por exemplo, se um usuário estiver usando seus privilégios de acesso de forma maliciosa. A figura 2.1 exemplifica a implantação de um NIDS.

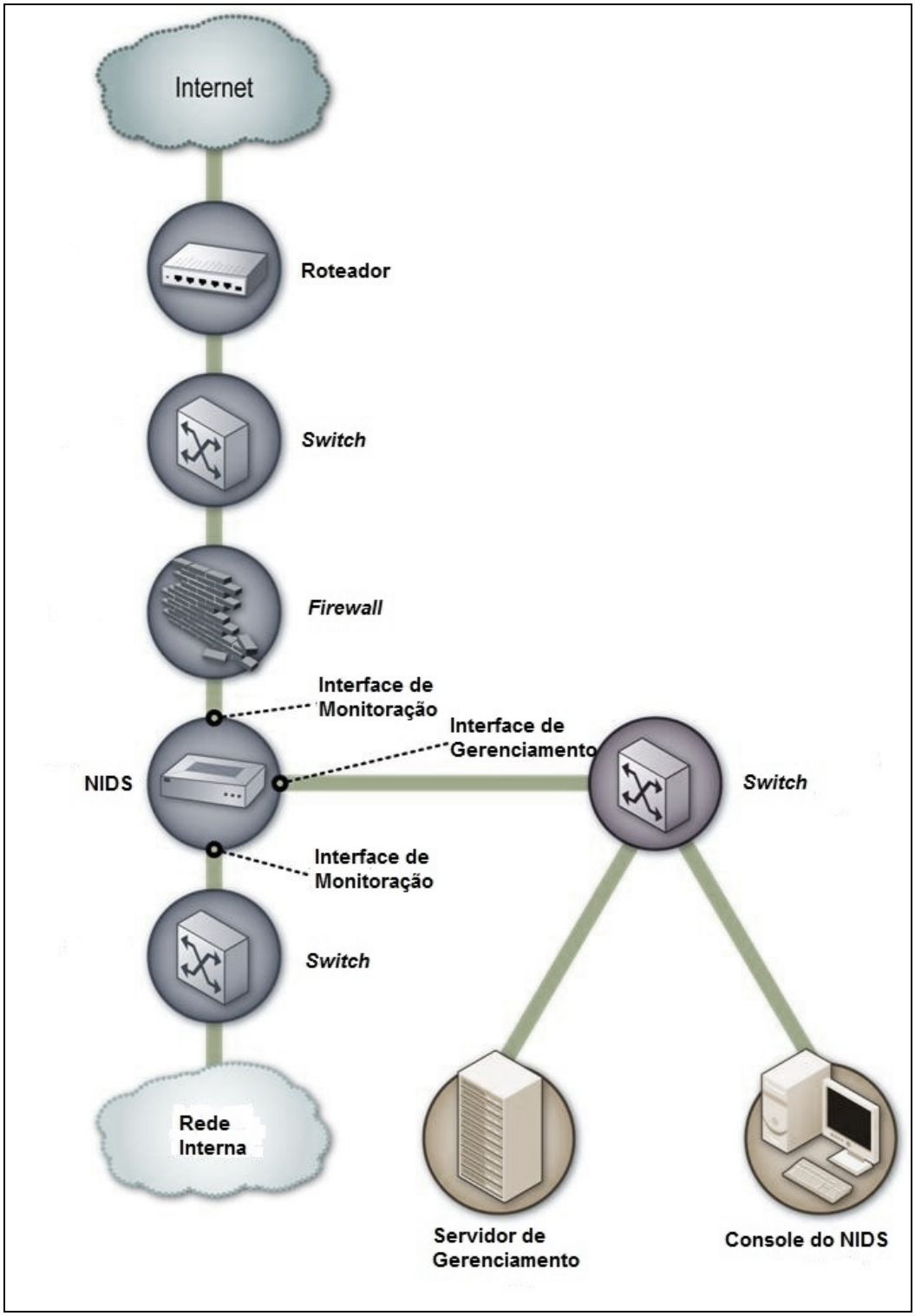


Figura 2.1 - Arquitetura NIDS adaptado de Scarfone e Mell (2007)

HIDS monitora as características de um *host* e dos eventos ocorrendo neste *host* para identificar atividades suspeitas (SCARFONE e MELL, 2007, p. 21). Exemplos das características que um HIDS pode monitorar são tráfego de rede

(somente para aquele *host*), logs de sistema, processos sendo executados, atividades de aplicações, modificação e acesso de arquivos, e modificações de configurações do sistema.

HIDS são geralmente instalados em *hosts* que são críticos, como servidores publicamente acessíveis e servidores contendo informações sensíveis, podendo detectar ataques que não venham necessariamente da rede. HIDS tem normalmente um tempo de resposta menor do que NIDS. Uma desvantagem, porém, é que o HIDS não é escalável, ou seja, à medida que novos *hosts* são adicionados ao ambiente de monitoração, a complexidade de gerenciamento aumenta pela adição de novos HIDSs para esses *hosts*.

2.2 WEB SERVICES

Web services são sistemas projetados para prover um padrão de interoperabilidade entre diferentes aplicações executando em uma variedade de plataformas e *frameworks* (BOOTH et al, 2004). *Web services* têm sua interface descrita em um documento WSDL (documento de descrição), sendo que outros sistemas interagem com os mesmos utilizando mensagens SOAP, tipicamente sobre HTTP com serialização XML.

Web services são uma noção abstrata que deve ser implementada por um agente concreto. O agente é uma peça de software ou hardware que envia e recebe mensagens, enquanto que o serviço é o recurso caracterizado pelo conjunto de funcionalidades abstratas providas (BOOTH et al, 2004). Para ilustrar, *web services* podem ser implementados utilizando um agente específico inicialmente (escrito em uma linguagem de programação específica) e depois este pode ser utilizado por outro agente (escrito em uma linguagem de programação diferente) com a mesma funcionalidade. Mesmo que o agente tenha mudado, a funcionalidade de *web services* continua a mesma.

O objetivo de *web services* é prover alguma funcionalidade em nome de seu

dono (que pode ser uma pessoa ou uma organização). A entidade provedora é uma pessoa ou organização que fornece um agente implementando um determinado serviço. Uma entidade solicitante é uma pessoa ou organização que deseja fazer uso de *web services* de uma entidade provedora. Esta primeira entidade utilizará um agente solicitante para trocar mensagens com o agente provedor da segunda entidade. Na maioria das vezes o agente solicitante é quem inicia a troca de mensagens.

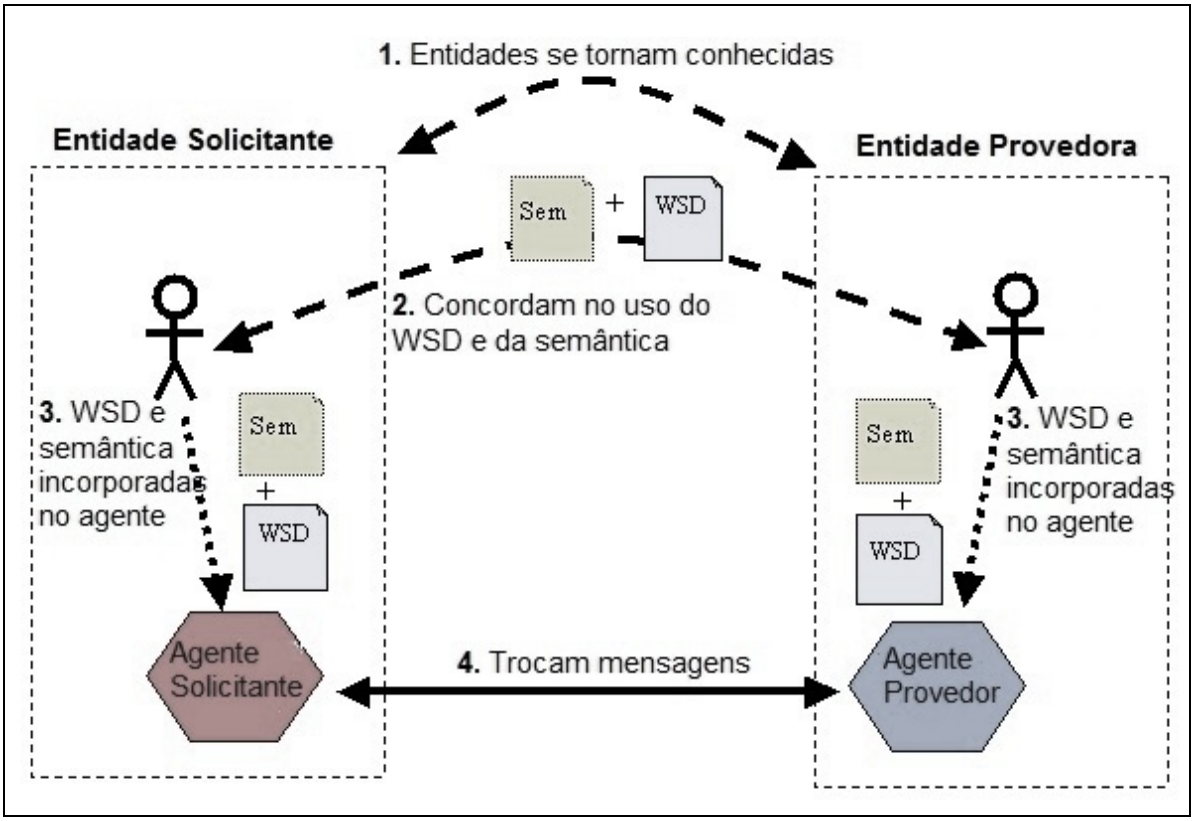


Figura 2.2 - Passos para troca de mensagens em *web services* adaptado de Booth et al (2004)

Para que haja sucesso na troca de mensagens entre *web services*, a entidade solicitante e a entidade provedora devem primeiro concordar com o uso da semântica e da mecânica da troca de mensagens. De acordo com Booth et al (2004), esse processo é feito em quatro passos.

No primeiro passo, a entidade solicitante precisa se tornar conhecida pela entidade provedora. No segundo passo, a entidade solicitante e a entidade provedora concordam no uso do documento de descrição (documento WSDL) e na semântica do serviço, que regrarão a interação entre o agente solicitante e o agente provedor. Isto não significa necessariamente que a entidade solicitante e a entidade

provedora devem se comunicar ou negociar entre si, simplesmente significa que ambos os lados devem ter o mesmo entendimento da descrição e semântica do serviço. No terceiro passo, a descrição e semântica do serviço são incorporadas, ou implementadas, no agente solicitante e no agente provedor. No quarto e último passo, os agentes solicitante e provedor trocam mensagens SOAP em nome de seus donos. Estes passos estão ilustrados na figura 2.2.

A semântica de *web services* é a expectativa (compartilhada) do comportamento do serviço, especialmente em relação às mensagens que são enviadas ao mesmo. Este é o “contrato” entre a entidade solicitante e a entidade provedora em relação ao propósito e às consequências da interação. Embora esse “contrato” represente o acordo entre as duas entidades, não é preciso que este seja escrito nem explicitamente negociado. O "contrato" pode ser explícito ou implícito, oral ou escrito, processável por um computador ou compreensível por um humano, e pode ser um acordo legal ou informal.

Enquanto a descrição do serviço representa um “contrato” gerindo a mecânica da interação com um determinado serviço, a semântica representa um “contrato” gerindo o sentido e o propósito desta interação. A fronteira entre esses dois “contratos” não precisa ser rígida, pois quanto mais se usa linguagens ricas em semântica para descrever a mecânica da interação, mais informação pode migrar da semântica informal para a descrição do serviço.

Existem diversas tecnologias que podem ser úteis com *web services*, porém as mais importantes são XML, SOAP e WSDL.

XML reduz significantemente a obrigação de implantação das várias tecnologias necessárias para garantir o sucesso de *web services*, pois oferece um formato padrão, flexível e inerentemente extensível (BOOTH et al, 2004). Os aspectos importantes de XML são a sintaxe em si, os conceitos do XML *Infoset* (um conjunto de informações e suas propriedades associadas para definir uma descrição abstrata de um documento XML bem formado), XML *Schema* (descrição formal do XML *Infoset*) e XML *Namespaces* (provê um método para classificar nomes de elementos e atributos usados em documentos XML). Um objeto é considerado um documento XML somente se for bem formado, de acordo com a especificação XML 1.0 (BRAY et al, 2000). Porém, mesmo sendo bem formado, não necessariamente será um objeto válido (precisaria obedecer outras restrições também).

SOAP provê um *framework* combinável, extensível e padrão para

empacotamento e troca de mensagens XML, que podem ser transmitidas através de diversos protocolos de rede, como HTTP, SMTP (*Simple Mail Transfer Protocol*) e FTP (BOOTH et al, 2004). O *framework* foi projetado para ser independente de qualquer modelo de programação. Uma mensagem SOAP possui dois elementos, o cabeçalho (opcional) e o corpo (obrigatório). O cabeçalho é um mecanismo de extensão que provê uma forma de transmitir informações que não são carregadas pela aplicação. Essas informações de controle podem ser, por exemplo, informações de contexto relacionadas ao processamento da mensagem, e podem ser alteradas pelas entidades à medida que a mensagem vai sendo repassada. O corpo é o elemento mandatório da mensagem SOAP e carrega a principal informação que será transmitida.

A WSDL provê um modelo e um formato XML para descrever *web services*. A WSDL permite que a descrição abstrata das funcionalidades oferecidas por um serviço seja separada dos detalhes da descrição do mesmo (BOOTH et al, 2004).

Em um nível abstrato a WSDL descreve *web services* em termos de mensagens que envia e recebe: uma operação associa um padrão de troca de mensagens com uma ou mais mensagens; um padrão de troca de mensagens identifica a sequência e a cardinalidade das mensagens enviadas e/ou recebidas assim como para quem elas são enviadas e/ou de quem são recebidas; e, finalmente, uma interface agrupa as operações, sem compromisso com nenhum formato de transporte.

Em um nível concreto a WSDL descreve *web services* em detalhe (como “onde” e ”de que forma” aquela funcionalidade é oferecida): um *binding* especifica detalhes do formato de transporte para uma ou mais interfaces; um *endpoint* associa um endereço de rede com um *binding*; e, finalmente, um serviço agrupa *endpoints* que implementam uma interface comum.

Ataques a *web services* envolvem ameaças ao sistema do *host*, à aplicação e à rede. Para garantir a segurança de *web services* uma variedade de mecanismos de segurança baseados em XML é necessária para resolver problemas relacionados à autenticação, autorização, políticas de segurança distribuída e segurança em nível de mensagens. Para fins deste trabalho, somente a segurança em nível de mensagens será detalhada.

Mesmo que tecnologias de segurança de rede tradicionais sejam usadas para *web services*, estas são insuficientes neste contexto, pois *web services* necessitam

de granularidade mais fina. Em geral, *web services* usam uma abordagem baseada em mensagens que permite interações complexas, incluindo o roteamento de mensagens entre diversos domínios de segurança. Uma mensagem pode passar por vários intermediários antes de chegar ao seu destino.

Exemplos de ameaças à segurança de mensagens em *web services* são ataques de *Probing*, XML *injection* e Manipulação de Protocolo.

Ataques de *Probing* tentam obter informações através do documento WSDL para ajudar no ataque a *web services*. Essas informações vão desde a localização do serviço até os tipos de funções e métodos que podem ser chamados, juntamente com os parâmetros necessários (CAPEC, 2011).

XML *injection* é uma técnica de ataque utilizada para manipular ou comprometer a lógica de uma aplicação ou serviço XML. A inserção de conteúdos e/ou estruturas XML não intencionados em uma mensagem XML pode alterar a lógica da aplicação, bem como causar a inserção de conteúdo malicioso no documento resultante (OWASP, 2011).

Ataques de manipulação de protocolo podem alterar maliciosamente os parâmetros de uma mensagem SOAP, podendo causar desde a alteração do destino da mensagem até ataques de SQL (*Structured Query Language*) *injection (SQL injection through SOAP parameter tampering)* e XML *injection* através do cabeçalho da mensagem (CAPEC, 2011).

2.3 ONTOLOGIA

Ontologia é “uma especificação de uma conceitualização” (GRUBER, 1993, p. 907). Em outras palavras, uma ontologia é a representação do conhecimento de um determinado domínio utilizando um formalismo declarativo. Os objetos descritos em uma ontologia, juntamente com os relacionamentos entre eles, podem ser refletidos no vocabulário representacional com o qual um programa representa conhecimento.

Para fins deste trabalho, uma ontologia representa uma série de definições

que descrevem conceitos de um determinado domínio, bem como uma série de axiomas formais que restringem a interpretação e uso destes conceitos (KONSTANTINOU, SPANOS e MITROU, 2008, p. 4). Além disso, a ontologia permite interoperabilidade entre sistemas baseando-se numa conceitualização compartilhada e permite ainda que inferências sejam feitas de acordo com os axiomas (verdades reconhecidas) previamente definidos (DOU, MCDERMOTT e QI, 2004, p. 35).

As características que uma ontologia deve ter são: clareza, coerência, extensibilidade e um mínimo compromisso ontológico (GRUBER, 1993, p. 908-909).

A **clareza** refere-se à definição dos termos, que deve ser objetiva e formal. Se uma definição pode ser feita através de axiomas lógicos, então esta deve ser feita. Sempre que possível, uma definição completa (uma classe/conceito definido por condições necessárias e suficientes) deve ser usada ao invés de uma definição parcial (definido somente por condições necessárias ou somente por condições suficientes).

Em relação à **coerência**, uma ontologia deve somente aceitar inferências que são consistentes com as definições e os axiomas de definição devem ser logicamente consistentes. Coerência também se aplica aos conceitos definidos informalmente, por exemplo, os que são descritos em linguagem natural para documentação.

A **extensibilidade** se refere à previsão dos usos do vocabulário comum provido pela ontologia. A ontologia deve ser construída de uma forma que possa ser sempre estendida e especializada, sem que os termos já existentes precisem ser reavaliados.

E finalmente, uma ontologia deve ter o **mínimo compromisso ontológico** possível que atenda as necessidades de compartilhamento. Ou seja, a ontologia deve afirmar o mínimo possível de verdades absolutas sobre o domínio sendo modelado. Desta forma, os sistemas utilizando a ontologia têm liberdade para especializar e instanciar a ontologia conforme necessário.

Dentre as características descritas acima, existem algumas contradições, por exemplo, entre clareza e mínimo compromisso ontológico, e é por isso que antes de se construir uma ontologia deve-se estudar o domínio que se deseja representar e o objetivo que a ontologia em questão deve atender.

Para facilitar o entendimento da aplicação de ontologias, apresenta-se um

exemplo prático onde uma ontologia é utilizada para compartilhar dados bibliográficos, como por exemplo, informações em um catálogo de uma biblioteca e informações de referências no final de um trabalho escolar (GRUBER, 1993, p. 918-921).

O objetivo desta ontologia é permitir uma tradução automática entre bases de dados de referências bibliográficas, para suportar a especificação de estilos de formatação bibliográficos independentemente do programa. Nesta ontologia, a bibliografia é feita de *referências*. Uma *referência* contém informações necessárias para se encontrar uma publicação. *Referências* contêm *dados* sobre publicações; *referências* não são as publicações em si. *Documentos* são as coisas criadas por *autores* que podem ser lidos, vistos, escutados, etc. Livros e revistas são *documentos*. Podem existir *referências* para várias publicações por *documento*, como em coleções. *Documentos* são criados por *autores*, que são *pessoas*. *Documentos* são publicados por *editoras* em datas de publicação, que são *pontos no tempo*. *Autores* e *editoras* têm *nomes*.

Todos os termos em *itálico* mencionados acima são classes (ou objetos) na ontologia. Por exemplo, referências bibliográficas são representadas por instâncias da classe *referência*. A definição desta classe inclui as condições necessárias de "ser associada com um *documento"* e de "ser associada com um *título*". Um documento, por exemplo, estaria associado com uma referência através de uma propriedade (ou relacionamento). Seguindo nesta linha de pensamento, a ontologia pode ser projetada até que todas as situações que se deseja contemplar tenham sido expressas.

A linguagem padrão para se representar ontologias, recomendada pela W3C (*World Wide Web Consortium*) desde fevereiro de 2004, é a OWL (*Web Ontology Language*) (MCGUINNESS e HARMELEN, 2004). A OWL é projetada para uso por aplicações que necessitem processar o conteúdo da informação ao invés de somente apresentar a informação a humanos. A OWL permite uma maior interpretação de conteúdo web por um computador, se comparada com XML e RDF (*Resource Description Framework*), por exemplo, pois provê vocabulário adicional juntamente com uma semântica formal. Esta linguagem possui três sublinguagens que aumentam o nível de expressividade de uma para a outra: OWL Lite, OWL DL (*Description Logics*) e OWL Full.

A OWL Lite permite uma classificação hierárquica de termos, com

simplicidade nas restrições oferecidas. Por exemplo, esta sublinguagem oferece suporte a restrições de cardinalidade, porém, só permite valores de cardinalidade 0 ou 1.

A OWL DL permite o máximo de expressividade sem perder a completude computacional (todas as inferências são computáveis) e decidibilidade (todas as computações serão completadas em um tempo finito). A OWL DL inclui todas as construções da linguagem OWL, que podem ser usadas com algumas restrições. Por exemplo, esta sublinguagem permite que uma classe seja subclasse de várias classes, mas não permite que uma classe seja instância de outra classe.

A OWL *Full* permite a máxima expressividade juntamente com a liberdade sintática do RDF, porém sem garantia de computabilidade. Por exemplo, na OWL *Full* uma classe pode ser tanto uma coleção de instâncias quanto uma instância de outra classe. Dificilmente um motor de inferência conseguirá inferir levando em conta todas as características desta sublinguagem.

O mecanismo que infere sobre o conhecimento contido em uma ontologia chama-se máquina (ou motor) de inferência (*reasoner*). Este tipo de mecanismo avalia e aplica as regras (axiomas) de acordo com as informações contidas na base de conhecimento da ontologia (RUSSEL e NORVIG, 1995).

2.4 RECURSOS

Esta seção apresenta os principais recursos disponíveis para utilização de ontologias e para o desenvolvimento de softwares visando segurança.

A ferramenta Protégé (STANFORD, 2011) é um editor de ontologias e um *framework* para bases de conhecimento que é gratuito e *open-source*. Permite exportar arquivos de ontologia em diversos formatos, incluindo RDF, OWL e XML *Schema*. O Protégé é baseado na linguagem Java, é extensível e provê um ambiente que o torna uma base flexível, permitindo rápida prototipação e desenvolvimento de aplicações. O Protégé é suportado por uma comunidade de

desenvolvedores e acadêmicos, usuários do governo e corporativos, que usam esta ferramenta para soluções em diversas áreas de conhecimento. Neste trabalho esta ferramenta foi utilizada para o desenvolvimento da ontologia baseada em estratégia.

O Jena (SOURCEFORGE, 2011) é um *framework* Java para desenvolvimento de aplicações que necessitam trabalhar com conteúdo semântico. O Jena provê funções para manipular arquivos RDF e OWL (arquivo da ontologia). Também provê suporte ao SPARQL, *SPARQL Protocol and RDF Query Language* (PRUD'HOMMEAUX e SEABORNE, 2008), que é uma linguagem para consultas em arquivos RDF e OWL. Juntamente com o Jpcap (SOURCEFORGE, 2011), biblioteca Java para captura de pacotes de rede, o Jena foi o principal recurso utilizado para o desenvolvimento do protótipo.

O Pellet (CLARK&PARSIA, 2011) é uma máquina de inferência para arquivos OWL, também baseado na linguagem Java. O Pellet pode ser utilizado por aplicações que precisem representar e inferir informações utilizando OWL. Neste trabalho o Pellet foi utilizado tanto para a construção da ontologia, sendo acessado pelo Protégé através da interface padrão DIG (*DL Implementation Group*) (BECHHOFER, 2006), quanto para testar o protótipo em tempo de execução. O Pellet já possui uma interface nativa do Jena, eliminando a necessidade de se utilizar uma interface padrão para inferência. A inferência direta é muito mais eficiente (e.g. não tem o atraso da comunicação por HTTP) e permite inferências mais poderosas do que utilizando o DIG, que é uma interface mais genérica.

O Wireshark (COMBS, 2011) é uma ferramenta de análise de protocolos de rede. Permite a captura e análise interativa do tráfego em uma rede de computadores. O Wireshark utiliza a biblioteca pcap para captura de pacotes sendo enviados e recebidos. Utilizou-se este *sniffing* para captura de pacotes na rede para estudar as características dos ataques sendo executados pelas ferramentas de teste de segurança e para comprovar (ou confrontar) as estratégias de ataque sugeridas pela CAPEC (*Common Attack Pattern Enumeration and Classification*).

A CAPEC (CAPEC, 2011) é uma taxonomia de padrões de ataque que descreve mecanismos para explorar falhas de software de acordo com a perspectiva do atacante. Foram utilizadas as informações contidas neste site para construir a estrutura inicial das classes de ataque e suas estratégias na ontologia.

O Metasploit (MOORE, 2011) provê informações e ferramentas para testes de segurança e para desenvolvimento de assinaturas para sistemas de detecção de

intrusão. Este projeto foi criado para prover informações úteis sobre técnicas de exploração de vulnerabilidades e para criar uma base de conhecimento funcional para profissionais da área de segurança. O Metasploit foi uma das ferramentas de teste de segurança utilizadas para estudar as características de ataques, neste caso o *XPath Injection*.

O *Open Web Application Security Project* (OWASP, 2011) é uma organização sem fins lucrativos focada em aprimorar a segurança de aplicações *web*. A OWASP mantém uma série de projetos, dentre eles um que tem como objetivo prover acesso a aplicações de teste de segurança a *web services*. Uma destas ferramentas, o WSFuzzer (ANDREU e BANCIU, 2011), foi utilizada para testar ataques de *XPath/XQuery Injection*, *XSS Injection*, *SQL Injection* e *External Entity Attack* contra *web services*. Outra ferramenta disponibilizada pelo OWASP é o WebScarab (DAWES, 2011), que foi utilizada principalmente como uma *Spidering Tool* para testar a proposta.

Outra ferramenta de teste de segurança utilizada foi o WSDigger (MCAFEE FOUNDSTONE, 2005). O WSDigger é um *framework* de testes para *web services* projetado para automatizar testes de segurança. Além da ferramenta que executa os ataques, este *framework* também provê um *web service* vulnerável a *XPath/XQuery injection*. Esta ferramenta foi utilizada para simular ataques de *XPath/XQuery Injection*, *XSS Injection* e *SQL Injection*.

3 TRABALHOS RELACIONADOS

Os trabalhos relacionados foram divididos em quatro seções, sendo que as três primeiras representam as abordagens utilizadas pelas propostas estudadas: IDS e Ontologia, IDS e *Web Services* e Firewall/Proxy e *Web Services*. A quarta e última seção contém considerações finais e um quadro com uma rápida comparação entre as propostas.

3.1 IDS E ONTOLOGIA

Segundo Vorobiev e Han (Vorobiev e Han, 2006), *web services* se tornaram uma parte importante da web pelos vários recursos que oferecem, como o fato de serem simples de se utilizar e serem independentes de plataforma. Porém, estes recursos fazem com que *web services* sejam vulneráveis a várias ameaças de segurança.

Web services semânticos, que podem publicar dados semânticos sobre suas propriedades funcionais e não funcionais, adicionam ainda mais problemas de segurança, já que para encontrar alvos um atacante precisa apenas consultar o UBR (*UDDI Business Registry*) para coletar toda a informação de que necessita para atacar *web services*.

Os autores apresentam informações sobre ataques a *web services* com vários exemplos, como o ataque *Mitnick* (relacionado a ataques do tipo *man-in-the-middle*), ataques de *probing* (escaneamento de documentos WSDL e procura de *web services* no UBR) e ataques de *CDATA field injection* (inclusão de caracteres ilegais em documentos XML, podendo ser seguido de ataques XML).

Os autores também mencionam ataques de DoS (*Denial of* Service, que é a tentativa de causar negação de serviço, por exemplo, pelo envio incessante de mensagens SOAP aos *web services* ou pelo envio de um documento XML com um número muito grande de elementos aninhados), ataques usando o SOAP, (criação de mensagens SOAP com cabeçalhos extremamente complexos) e ataques XML (alteração da estrutura de um documento XML para enganar o analisador, *parser*,

permitindo que código XML seja inserido no banco de dados, por exemplo).

De acordo com os autores, com a crescente necessidade do uso de *web services*, novos tipos de ataque estão surgindo, e alguns antigos estão reaparecendo. Portanto, firewalls e sistemas de detecção de intrusão (F/IDS) clássicos não são mais suficientes para proteger *web services*, pois eles normalmente são aplicados a um host somente e não se comunicam com outros.

A proposta dos autores é criar uma ontologia de ataques a *web services* que forneça um vocabulário comum para F/IDS, utilizando a OWL como linguagem. Os autores escolheram o uso de ontologia ao invés de taxonomia porque a primeira permite que entidades diferentes possam compartilhar um entendimento comum de informações que podem ser inferidas e analisadas automaticamente, permitindo que firewalls e IDSs distribuídos possam se comunicar mesmo sendo de diferentes fabricantes e com propósitos distintos. Além disso, as ontologias podem evoluir com o tempo. Desta forma, *web services* estariam mais protegidos como um todo, e não individualmente.

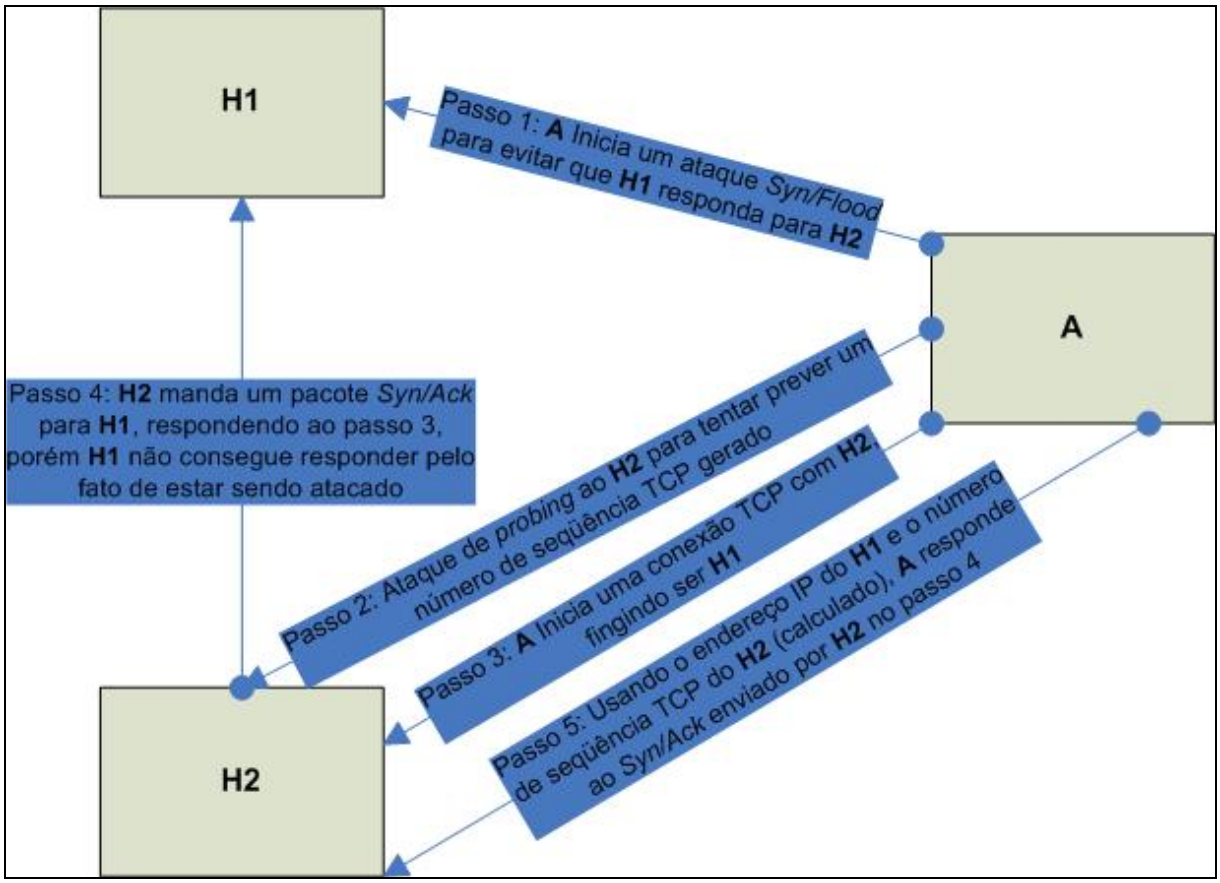


Figura 3.1 - Ataque *Mitnick* adaptado de Vorobiev e Han (2006)

Porém, o artigo somente utiliza um exemplo de como o uso da ontologia protegeria *web services*, no caso de ataques *Mitnick*, os quais não são ataques exclusivos a *web services*. A proposta também não menciona testes nem resultados concretos, descrevendo somente a teoria.O ataque *Mitnick* é um ataque distribuído e multi-fásico que explora fraquezas do design do protocolo TCP (*Transmission Control Protocol*) na hora de estabelecer uma conexão TCP entre dois *hosts*. Neste tipo de ataque, o atacante (**A**) faz com que o *host* 2 (**H2**) acredite que está se comunicando com o *host* 1 (**H1**) no qual **H2** confia, através de um ataque de *Syn/Flood* baseado no "aperto de mão triplo" (*three way handshake*) durante o estabelecimento de uma conexão TCP. O ataque é representado na figura 3.1.

Após o sucesso do ataque, **H2** acha que estabeleceu uma conexão confiável com **H1**, quando na verdade está se comunicando com um *host* malicioso, permitindo ao atacante causar danos a **H2**. Neste exemplo do artigo, é mencionado que o F/IDS de **H1** pode registrar um ataque de Syn/Flood curto, enquanto o F/IDS de **H2** pode detectar uma tentativa de prever o número de sequência TCP. Então se chega à conclusão de que se esses F/IDSs não agirem em conjunto, eles não conseguirão descobrir o ataque Mitnick, tornando a idéia do artigo viável.

Apesar de ser convincente na teoria, não existe uma implementação desta ontologia disponível para download e nenhum site para o projeto foi encontrado. Os autores ainda comentam que como trabalho futuro pretendem construir uma ontologia de defesa para *web services* e combinar com sua ontologia de ataques, para permitir desenvolver melhores mecanismos de defesa distribuídos.

Undercoffer e seus colegas (Undercoffer et al, 2004) produziram uma ontologia que especifica um modelo de ataques a computadores, baseada em uma análise de mais de 4.000 classes de intrusões e suas respectivas estratégias de ataque. Sua ontologia é categorizada principalmente de acordo com o componente alvo do ataque a um sistema, mas também considera os meios de ataque, as consequências do ataque e a localização do atacante.

A motivação argumentada pelos autores é a de que quaisquer características taxonômicas utilizadas para definir um ataque são limitadas em escopo e não abrangem tudo o que é observável e mensurável no alvo do ataque. Também mencionam que a classificação e descrição de ataques a computadores estavam limitadas a taxonomias até então. Além disso, enfatizam que a utilidade de uma ontologia não está na simples representação de atributos de um ataque, mas sim no

fato de que é possível expressar as relações entre os dados coletados e utilizar essas relações para deduzir que esse conjunto de dados representa um ataque.

Sua ontologia foi especificada utilizando DAML (*DARPA Agent Markup Language*) com OIL (*Ontology Inference Layer),* que é a linguagem antecessora à OWL, e foi prototipada utilizando DAMLJessKB (antecessora à OWLJessKB), que é uma máquina de inferência para DAML.

Para construir sua ontologia, os autores se basearam em uma análise empírica das características e atributos, e suas inter-relações, de mais de 4.000 classes de ataques e intrusões. A ontologia é representada em uma visão de alto nível pela figura 3.2.

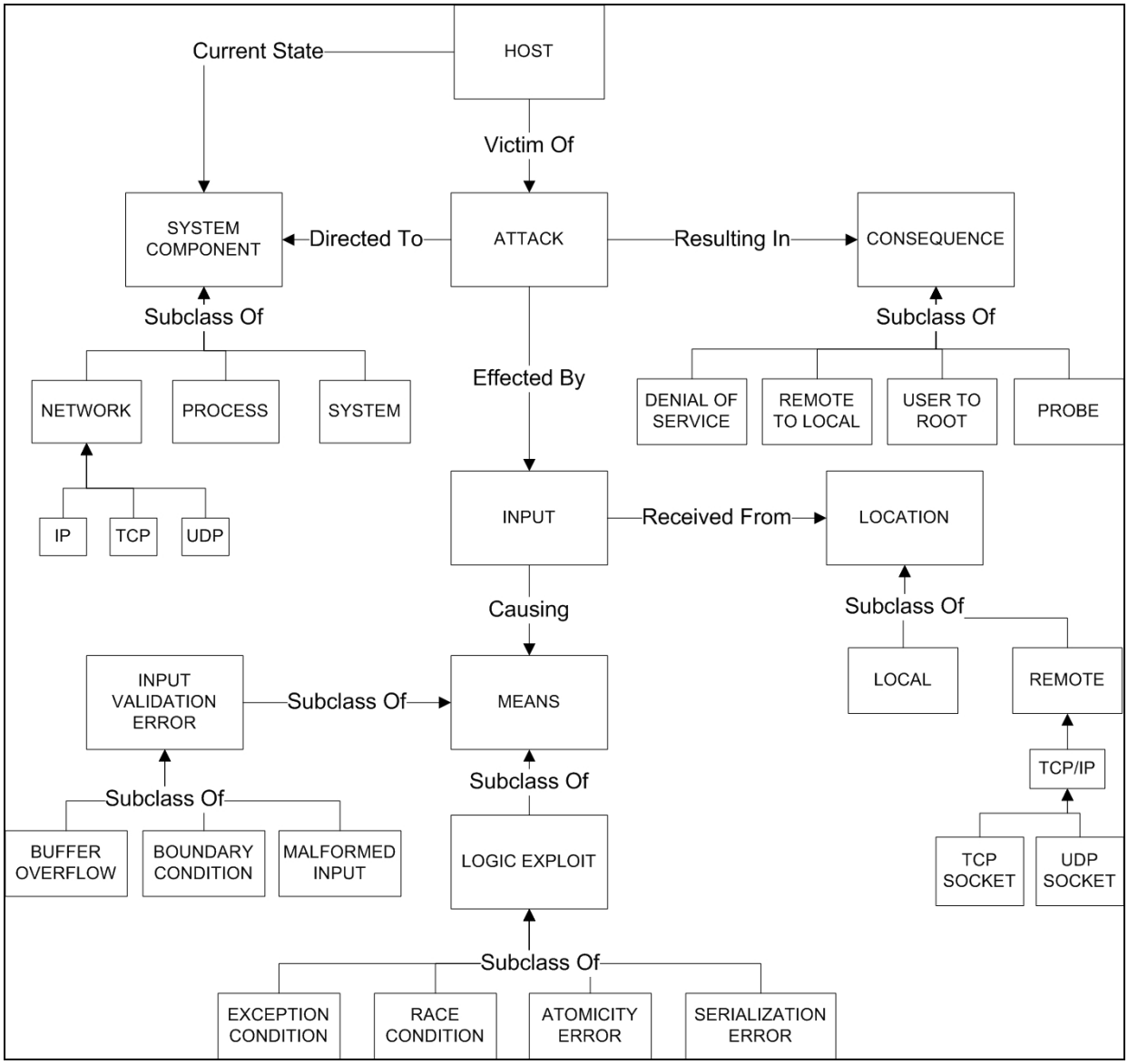


Figura 3.2 - *Target-centric ontology*, adaptado de Undercoffer et al (2004)

No topo da ontologia está a classe *Host*. Esta classe tem as propriedades *Current State*, que é definida pela classe *System Component*, e *Victim of*, definida pela classe *Attack*. A classe *System Component* possui as subclasses *Network*, *System* e *Process*. *Network* representa a camada de rede. Como os autores somente focam no protocolo TCP/IP, somente as subclasses IP (*Internet Protocol*), TCP e UDP (*User Datagram Protocol*) são consideradas abaixo de *Network*. A subclasse *System* representa o sistema operacional do *host*, possui atributos que representam o uso da memória e o uso do processador. Esta subclasse também possui atributos relacionados ao número de usuários simultâneos, utilização de disco, tabela de chamadas de sistema, etc. A subclasse *Process* possui atributos que representam os processos que devem ser monitorados. Estes atributos incluem o valor atual do apontador de instruções, o topo da pilha e o número de processos filhos.

A classe *Attack* possui as propriedades *Directed to*, *Effected by*, e *Resulting in*. Essa construção foi escolhida pelos autores pela noção de que um ataque consiste de algum conteúdo que é direcionado a algum componente do sistema e resulta em alguma consequência. Desta forma, as classes *System Component*, *Input*, e *Consequence* são os objetos correspondentes. Esta última possui diversas subclasses, entre elas *Denial of Service* (negação de serviço), *User Access* (acessos não privilegiados), *Root Access* (acessos privilegiados) e *Probe* (escaneamento, podendo obter informações de perfis do sistema).

Finalmente, a classe *Input* possui os atributos *Received from* e *Causing*, onde esta última define a relação entre as classes *Means* e *Input*. Os autores definem as subclasses *Input Validation Error* e *Logic Exploits* para a classe *Means*. *Input Validation Error* representa erros causados quando conteúdo malformado é recebido pelo sistema e não é tratado apropriadamente, e possui as subclasses a seguir: *Buffer Overflow* (utilização extensiva da memória através do envio de estruturas de dados complexas), *Boundary Condition Error* (um processo tenta ler ou escrever além do último endereço válido) e *Malformed Input* (quando um processo aceita conteúdo sintaticamente incorreto). A subclasse *Logic Exploits* representa vulnerabilidades de hardware ou software que podem ser exploradas, e possui as subclasses a seguir: *Exception Condition* (erro originado pela falha em manipular uma exceção gerada), *Race Condition* (erro que ocorre durante uma janela de

tempo entre duas operações), *Serialization Error* (serialização inapropriada de operações) e *Atomicity Error* (erro que ocorre quando uma estrutura de dados parcialmente modificada é utilizada por outro processo).

Após apresentar a estrutura da ontologia, os autores apresentam a forma de implementação, utilizando a máquina de inferência DAMLJessKB para inferir sobre instâncias do modelo de dados que são consideradas suspeitas e reajustá-las de acordo com as regras e axiomas definidos na ontologia. Nesta seção também exemplificam com ilustrações a estrutura de algumas das classes da ontologia.

E para ilustrar o uso da sua ontologia, os autores utilizam o mesmo exemplo do ataque multi-fásico *Mitnick* utilizado por Vorobiev e Han (2006), que consiste de ataques de negação de serviço, previsão do número de sequência TCP e mascaramento de endereço IP. Para evitar esse tipo de ataque, conjuntos de IDSs são formados onde cada um é responsável por partes específicas de um domínio. Por exemplo, um IDS pode ser responsável por um determinado *host*, enquanto outro é responsável por um grupo de *hosts*, e um terceiro IDS é responsável por monitorar o tráfego de rede. Como esses IDSs todos compartilham uma ontologia em comum, todos ficam sabendo de qualquer ataque detectado por um deles.

Os autores exemplificam e ilustram bem o uso da sua proposta, porém não apresentam testes nem resultados numéricos. Um ponto forte é que disponibilizam a ontologia para *download* na internet. Pelo fato de a DAMLJessKB possuir algumas limitações, os autores pretendem modificá-la ou utilizar outra máquina de inferência na construção da ontologia completa.

3.2 IDS E WEB SERVICES

Yee, Shin e Rao (2007) sugerem o uso de um framework de detecção e prevenção de intrusão (ID/IP) adaptável, para conter as ameaças que rondam *web services*. Iniciam sua proposta falando sobre diversos ataques a *web services* existentes hoje em dia, devido ao aumento do número de adeptos desta plataforma. No geral, os autores afirmam que as demais propostas presentes na literatura são muito específicas e contemplam uma faixa pequena de ataques, concluindo que há necessidade urgente de se chegar a uma abordagem mais genérica de detecção

que contemple a maioria dos tipos de ataques. A motivação do framework adaptável proposto também tenta compensar as diferenças entre as abordagens de detecção de intrusão por anomalia e por assinatura, através da integração de agentes, *data mining* (mineração de dados) e técnicas de *fuzzy logic* (lógica difusa), em uma abordagem híbrida.

No framework proposto pelos autores (figura 3.3), comportamentos normais de usuários como IP de origem, IP de destino, ID de usuário e requisições de serviço mais frequentes são previamente arquivados. Outros aspectos como parâmetros SOAP, tamanho de mensagem, tempo de requisição/resposta, número de mensagens em um intervalo de tempo para uma requisição/resposta, sintaxe XML válida e comandos/sintaxe SQL válidos também são capturados.

Agentes atuando como sensores são usados para detectar violações ao perfil normal de usuários utilizando técnicas de *data mining* como análise de cluster e análise de associações. Então as anomalias são analisadas utilizando *fuzzy logic* para determinar ataques genuínos e reduzir alarmes falsos. Itens normais que não sejam capturados pelos sensores continuarão com suas atividades normais, afetando desta forma o desempenho do sistema minimamente. Se um ataque está sendo detectado, um componente chamado “provedor de ações” irá agir para prevenir que o ataque aconteça. As ações tomadas por esse componente do framework podem ser de bloquear, rejeitar ou terminar a atividade.

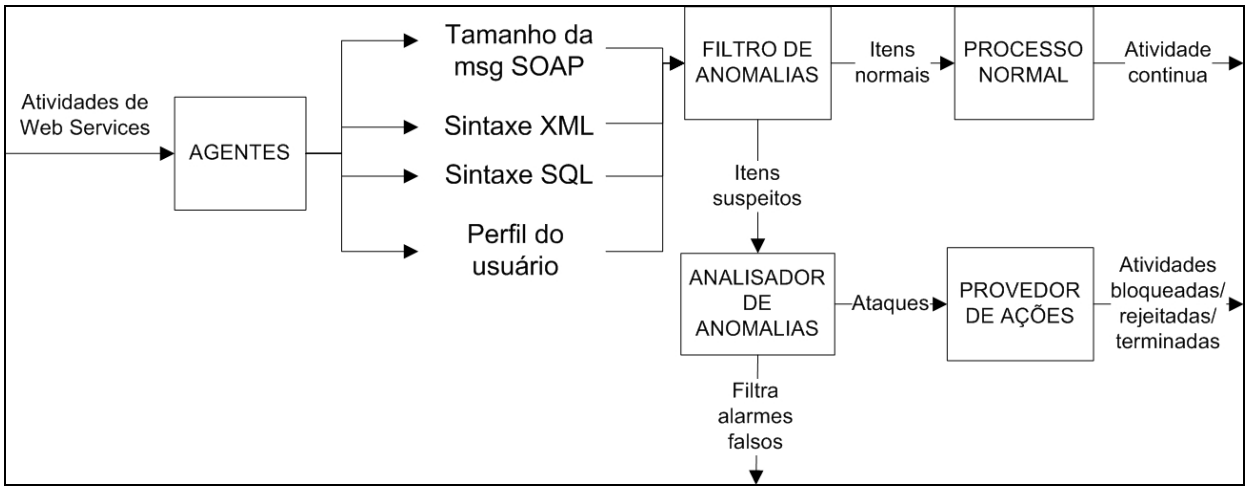


Figura 3.3 - Framework de ID/IP, adaptado de Yee, Shin e Rao (2007)

Para demonstrar como o framework opera, os autores ilustram a proposta utilizando três cenários.

No primeiro cenário a técnica baseada em análise de cluster é utilizada. Uma requisição específica de serviço com uma mensagem SOAP tem um tamanho normal (armazenado previamente na base de dados) igual a 1MB. Qualquer anomalia para essa requisição de serviço que contenha uma mensagem SOAP com o tamanho de 1 a 5% maior do que o normal será categorizada como pertencente ao cluster de 5%, e receberá um índice de cluster de 0,95 ou um indicador de ataque igual a LOW. Outra anomalia com uma mensagem SOAP de tamanho igual a 2MB pertencerá ao cluster de 50% e receberá um índice de cluster de 0,5 ou um indicador de ataque igual à HIGH. Quanto maior for o desvio em comparação com o valor normal, menor será o índice de cluster recebido e maior será o indicador de ataque, mostrando a chance de a atividade ser um ataque de *oversized payload*.

Em um segundo cenário a técnica baseada em análise de associação é utilizada. Os ataques podem ser detectados quando algumas de suas características são analisadas em conjunto. Por exemplo, se uma grande quantidade de uma mesma mensagem SOAP é transmitida por um endereço de origem específico em um intervalo de tempo pequeno, provavelmente é um ataque de *flood*. Por exemplo, o número de mensagens transmitidas permitidas em um intervalo de tempo específico está sendo violado e esta atividade recebe um índice de cluster 0,65 ou um indicador de ataque MEDIUM, e o número de ocorrências do IP de origem para a mesma mensagem é violado também e recebe um índice de cluster de 0,35 ou um indicador de ataque HIGH. Presumindo que os indicadores HIGH, MEDIUM e LOW carregam pesos de 0,5, 0,3 e 0,2 respectivamente, então o índice de associação daria igual a 0,37 para o indicador de ataque MEDIUM-HIGH. Quanto menor o índice de associação, maior a chance de o ataque ser genuíno.

No terceiro e último cenário a técnica baseada em sequência é utilizada. Algumas requisições/respostas de serviços operam seguindo uma sequência específica de atividades. Por exemplo, uma requisição para acessar uma base de dados precisará primeiro passar por uma autenticação da ID do usuário, fazendo uma requisição SQL no banco para validar se a ID é genuína. Depois da validação, a ID ganha acesso ao banco de dados. Essa série de atividades será capturada como um perfil normal. Se um usuário errar a senha da ID várias vezes, então a atividade receberá um índice de cluster de 0,3 ou um indicador de ataque HIGH. Se a sintaxe SQL for violada, receberá outro índice de cluster de 0,5 ou um indicador de ataque MEDIUM por uma suspeita de SQL *injection*. E se a ID do usuário não

passar na validação do banco de dados para este tipo de acesso, a atividade recebe um terceiro índice de cluster 0,8 ou um indicador de ataque LOW. O índice sequencial do indicador de ataque HIGH-MEDIUM-LOW então teria uma média ponderada de 0,46, tendo uma chance baixa de ser um alarme falso.

De acordo com os autores, os exemplos acima mostram que o uso de técnicas de *data mining* juntamente com *fuzzy logic* permite tomar decisões em ambientes incertos e imprecisos. Como trabalho futuro comentam que pesquisas serão feitas para poder conceitualizar o framework proposto, já que não apresentam testes nem resultados concretos no artigo, apenas teoria. Outra menção também é feita sobre web semântica, pretendendo estender o framework em questão para cobrir as ameaças à segurança encontradas especialmente nas camadas XML e RDF. Como já comentado, a proposta dos autores fica na teoria, e nenhuma implementação foi encontrada disponível para download.

Em outra abordagem, Siddavatam e Gadge (2008) propõem uma série de testes para detectar diversos tipos de ataques a *web services* e desenvolvem um sistema em Java, utilizando AXIS2 SOAP (um módulo de processamento de requisições SOAP), que captura requisições SOAP feitas em um determinado período de tempo. Em resumo, essas requisições SOAP passam pelos testes propostos para determinar se têm potencial para serem classificadas como ataques. Todas as requisições que não passarem no teste são separadas para que ações posteriores possam ser tomadas.

Os autores dizem que firewalls conseguem bloquear tráfego de rede direcionado ao nível de sistema operacional e de serviços, porém não conseguem bloquear ataques direcionados ao nível de aplicações web, já que o tráfego nas portas 80 e 443 são legítimos. E firewalls de aplicação normalmente protegem apenas contra ataques baseados em HTML e browser, mas não contra os que utilizam o fluxo de mensagens XML. Além destas afirmações, os autores também se motivam em dados de pesquisa (não há referência no artigo de onde esses dados foram retirados) que mostram que 70% dos caminhos que foram fechados para ataques por firewalls na última década, serão reabertos pelos *web services* XML.

A arquitetura do sistema (figura 3.4) proposto pelos autores é formada por três camadas e uma base de dados central.

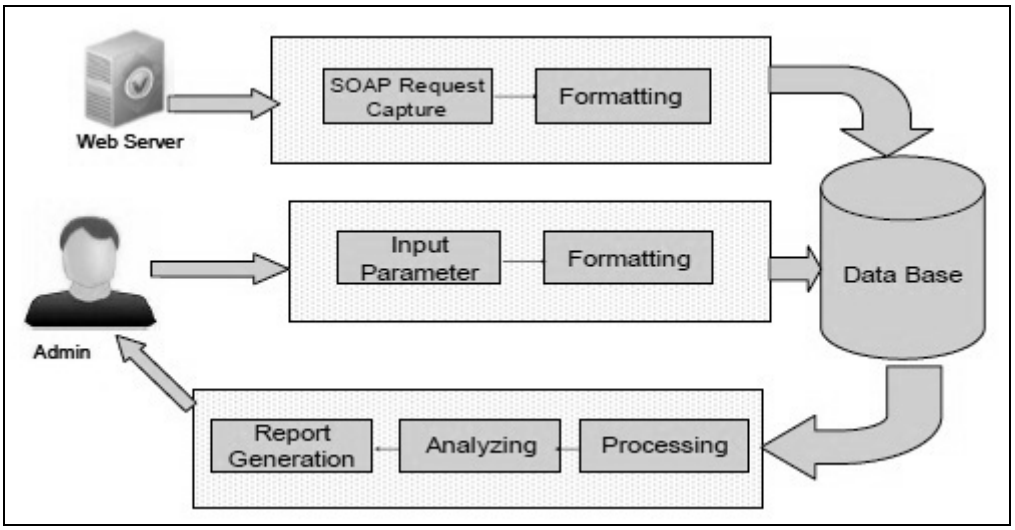


Figura 3.4 - Arquitetura do sistema proposto, Siddavatam e Gadge (2008)

A primeira camada recebe as requisições SOAP direcionadas aos *web services*. Um dos blocos da primeira camada é responsável por monitorar e capturar estas requisições. O segundo bloco dessa camada é responsável por formatar a requisição SOAP com parâmetros como o horário da requisição e o nome do serviço. Cada requisição é convertida em um arquivo XML e armazenada na base de dados. A segunda e terceira camadas são o coração do sistema e trabalham em sincronia.

A segunda camada recebe os valores permitidos para cada parâmetro, definidos pelo administrador, para validar as requisições SOAP e definir se caracterizam um possível ataque. Depois de coletadas todas essas informações, a terceira camada se encarrega de processar os testes através de diversos algoritmos para detectar os ataques.

Um dos algoritmos de teste é o *cipher test*. Neste teste informações sensíveis são cifradas e enviadas juntamente com a requisição SOAP para que possam ser verificadas nos servidores dos *web services*. Outro exemplo de algoritmo é o *replay attack test*, onde o objetivo é descobrir se o cliente está fazendo requisições dentro do intervalo de tempo permitido, visando evitar ataques de negação de serviço.

Para testar sua proposta, os autores desenvolveram um módulo de software que opera em sincronia com o módulo monitor e construíram diversos casos de teste para os ataques. Apresentam uma tabela com os resultados consolidados dos testes, porém não explicam como avaliar os dados da tabela. Os autores também ilustram no artigo de que forma os algoritmos de teste propostos funcionam, porém

nenhum deles é explicado em detalhes. Em geral, Siddavatam e Gadge (2008) passam muito rapidamente sobre cada ponto de sua proposta, ficando fácil de entender o objetivo de cada teste, porém difícil de entender o funcionamento do mecanismo de detecção dos ataques.

Zheng e Hu (2005) motivam seu trabalho na afirmação de que os ataques de negação de serviço distribuídos (DDoS) se tornaram a principal ameaça a *web services*, e que geralmente são precedidos de ataques de *Probing*. Um mecanismo de detecção de intrusão por anomalias baseado na técnica de quantização vetorial (*Vector Quantization*) é proposto para conter esses ataques. De acordo com os autores, o perfil de tráfego normal de rede pode ser modelado e representado pelo livro-código (*codebook*) da quantização vetorial, através do qual os desvios de um comportamento do tráfego TCP normal podem ser bem medidos quantitativamente.

Quantização vetorial é uma eficiente técnica de codificação, especialmente quando se fala em compressão de imagens e sinais de voz baseados nas medidas de similaridade entre vetores de atributos. A idéia básica da quantização vetorial é representar agrupamentos de dados através de um número finito de vetores que se denominam de protótipos. Os pontos são alocados em cada um dos agrupamentos de acordo com a técnica de vizinhos mais próximos. Deste modo, a idéia central é formar k grupos de tal forma que as distâncias entre os elementos dos dados de entrada $x = \{x_1 , x_2 , x_3 ,..., x_m \}$ e um dos valores de referência $w = \{w_1 , w_2 , w_3 ,..., w_k \}$ dos grupos seja mínimo. Então, cada elemento do conjunto x é classificado em um grupo.

A vantagem da aplicação de quantização vetorial para a análise de perfil de tráfego de rede é que se pode compartimentalizar o espaço do vetor de características do tráfego em grupos, comparando as similaridades dos vetores de características. Todos os perfis podem ser separados e recapitulados no livro-código. Consequentemente, através da organização do livro-código, pode-se obter os perfis de uso para caracterizar o tráfego de rede. Os autores decidiram treinar seu livro-código utilizando o algoritmo *Competitive Learning - Kohonen Learning Algorithm*. Depois de treinado, o livro-código serve como um dicionário de comportamentos normais do tráfego de rede, onde comportamentos parecidos ficam agrupados, e cada agrupamento possui sua palavra-chave.

Na fase de detecção de intrusões deste framework de quantização vetorial, os vetores de características recebidos (extraídos de atributos do tráfego TCP) serão

processados pelo livro-código, procurando a melhor palavra-chave para cada vetor. Para isso, nove atributos relacionados ao estabelecimento de uma conexão TCP são levados em conta: o endereço IP de origem, o endereço IP de destino, a porta de origem, a porta de destino, o tamanho médio dos pacotes do fluxo TCP, o número de bytes da origem, o número de bytes do destino, o estado do fluxo TCP, a freqüência de um determinado IP de origem em um intervalo de tempo e a freqüência de um determinado IP de destino em um intervalo de tempo.

O mais importante destes atributos é o estado do fluxo TCP, que é representado por um número de nove bits, cada bit representando uma *flag*. Para cada *flag*, seu bit correspondente é setado para 1 se ela for observada durante o processo de estabelecimento ou fechamento de uma conexão, senão o bit correspondente é setado para zero. O intervalo de tempo definido pelos autores para os últimos dois atributos foi de 2 segundos. A grande discrepância entre o comportamento normal e anormal de cada atributo é o que ajuda a detecção de intrusão baseada em anomalia a ser mais precisa.

Para testar e avaliar sua proposta, os autores utilizaram uma parte do conjunto de dados de 1999 para avaliação de detecção de intrusão da base DARPA. Com isso conseguiram estimar a quantização vetorial para detecção de intrusão por anomalia *offline*. O resultado dos testes é apresentado em tabelas e é avaliado pelos autores na conclusão. Segundo os mesmos, sua proposta permite atingir alta taxa de detecção com baixa taxa de falsos positivos, porém, não comparam a proposta com outras abordagens, além de terem usado uma base muito antiga.

A abordagem de Bravenboer et al (2010) sugere utilizar incorporação de sintaxe, de acordo com as linguagens utilizadas no *guest* e *host* (e.g. XPath com Java), para gerar automaticamente o código que irá prevenir vulnerabilidades para ataques de *injection* (e.g. adicionando funções para filtrar caracteres inválidos).

A motivação dos autores está no fato de que softwares desenvolvidos em uma determinada linguagem geralmente necessitam utilizar outras linguagens também, como queries SQL, XQuery e XPath, comandos Shell, etc. E isto é quase sempre feito sem muita preocupação, através de concatenação de strings para formar uma sentença, por exemplo. Isto propicia um tipo de falha de segurança bem conhecido, as *injections*. As *injections* constituem uma das maiores classes de problemas de segurança. O mais conhecido dos ataques é o SQL *injection*, porém *injections* acontecem nos mais diversos contextos.

De acordo com os autores, ataques de *injection* podem ser prevenidos filtrando conteúdo externo, por exemplo, tratando uma string para retirar caracteres inválidos para uma determinada linguagem. Porém, é fácil de esquecer-se de filtrar todo o conteúdo externo adequadamente. Uma solução melhor poderia ser utilizar uma API (*Application Programming Interface*) para fazer esse trabalho. Uma API poderia garantir que *injections* sejam prevenidas no momento da contrução de uma *query*. Entretanto, APIs específicas podem não existir ou serem diferentes para cada linguagem.

Baseando-se nas afirmações acima os autores apresentam sua proposta (*StringBorg*) para combinar a segurança de se utilizar uma API com a facilidade de manipular *strings* ao manipular sentenças (e.g *queries*). E fazem isto incorporando a sintaxe da linguagem *guest* na sintaxe da linguagem *host*. Por exemplo, abaixo apresenta-se um trecho de código representando SQL em Java:

```
SQL q = <| SELECT id FROM users WHERE name = ${userName} AND password = ${password} |>;
if (executeQuery(q.toString()).size() == 0) ...
```

A sintaxe SQL é incorporada diretamente no código Java utilizando “<|...|>” para construir código SQL. Da mesma forma, código Java é incorporado no SQL através de “${...}”, para permitir a composição do código SQL. Um preprocessador, chamado assimilador, traduz o código escrito nesta linguagem combinada para Java puro, que por sua vez chama uma API gerada da gramática da linguagem do *guest*. Esta API é responsável por garantir que qualquer entrada externa seja adequadamente filtrada.

A principal contribuição dos autores é que eles mostram que formalismos de análise modulares permitem que as incorporações sejam criadas genericamente. Ou seja, especificando a gramática de uma linguagem *guest*, é possível incorporar esta linguagem em todas as possíveis linguagens *host*. E especificando a gramática de uma linguagem *host* nova juntamente com um gerador de API, esta nova linguagem *host* imediatamente permite a incorporação de todas as linguagens *guest*.

O *StringBorg* utiliza um formalismo de definição de sintaxe modular para definir a sintaxe das linguagens *host* e *guest*. Este módulo primeiro define a sintaxe livre de contexto de expressões e *queries* simples, consistindo de *strings*, identificadores e comparações de igualdade, e depois define a sintaxe léxica das *strings*. Esta combinação da sintaxe livre de contexto e da sintaxe léxica é importante para a análise das incorporações.

A análise de arquivos que utilizam uma combinação de uma linguagem *host* e várias linguagens *guest* é um desafio para muitas técnicas de análise. Na abordagem proposta pelos autores, o analisador é gerado automaticamente da definição da sintaxe combinada. O problema de muitas técnicas de análise é que as gramáticas não podem ser compostas facilmente, por exemplo, porque uma linguagem *guest* geralmente possui palavras chaves e operadores diferentes da linguagem *host*. O *StringBorg* é baseado em um formalismo de definição de sintaxe que é implementado sem utilizar analisadores léxicos específicos, permitindo a generalização dos contextos e a incorporação de linguagens em outras linguagens de uma forma natural.

Um ponto positivo desta abordagem é que ela é genérica, podendo ser aplicada com facilidade a qualquer combinação de linguagens. De acordo com os autores este fato é importante para que a abordagem seja viável e prática para o uso. Testaram sua proposta utilizando as linguagens *host* Java e PHP e experimentaram com várias combinações de *guest* languages: SQL, LDAP, XPath, Shell e XML. Porém, fica uma discussão no artigo sobre a viabilidade da implementação desta abordagem em aplicações existentes, ou seja, da adoção desta nova abordagem por programadores. Uma limitação apontada pelos autores é o fato de nem todas as linguagens possuírem uma gramática livre de contexto, saindo do escopo atual do *StringBorg*. Por exemplo, algumas linguagens que possuem uma regra de identação (e.g. Haskell ou Python) não são suportadas.

3.3 FIREWALL/PROXY E WEB SERVICES

Bebawy e seus colegas (Bebawy et al, 2005) apresentam um firewall *open source* orientado a objetos (Nedgty), que tem como objetivo prevenir os novos ataques que surgiram com o crescente uso de *web services*. Este firewall previne contra ataques de negação de serviço (incluindo negação de serviço através de arquivos XML) e *buffer overflow*, além de possuir um mecanismo de autorização. Nedgty é um software *stand-alone* que fica no nível de aplicação do servidor, trabalhando como um *gateway* entre o cliente e o *pool* de *web services*. O sistema operacional utilizado para hospedagem é o Linux OS. Nedgty se comunica com o *IPTables*, um firewall

nativo do Linux, fazendo uso do módulo *QUEUE* para interceptar pacotes específicos de *web services*, validá-los e então reencaminhá-los ou rejeitá-los. Esse módulo no Linux serve para encaminhar qualquer pacote desejado ao espaço de usuário.

Uma interface é responsável por configurar as políticas de segurança do firewall e salvá-las em um repositório. Para que requisições passem pelo firewall, o administrador precisa registrar os *web services* que serão publicados e definir as políticas de segurança para cada um deles. Através da interface o administrador pode adicionar, editar e excluir os perfis dos *web services*. Em cada perfil o administrador especifica as operações permitidas, a localização do serviço e os formatos de mensagens SOAP que são aceitos, podendo fazê-los manualmente ou efetuando o *upload* do arquivo WSDL. Outros controles também podem ser especificados, como por exemplo, a taxa de requisições SOAP permitida pelos *web services*, prevenindo ataques de negação de serviço. O administrador escolhe as regras que se aplicam a cada um dos *web services*, permitindo uma maior proteção individual.

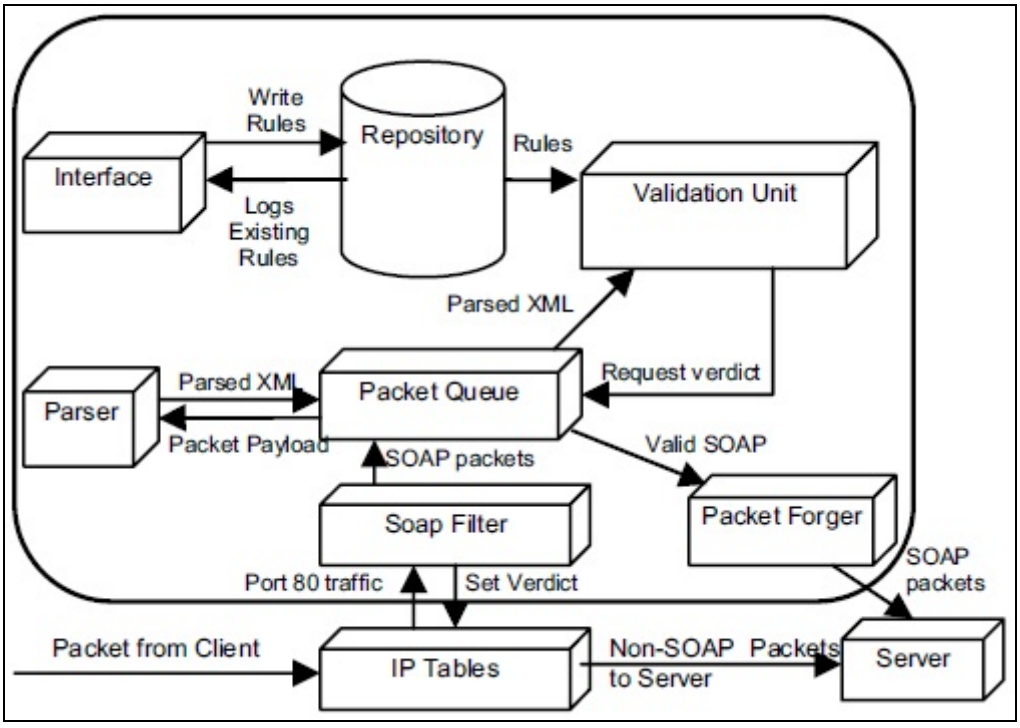


Figura 3.5 - Arquitetura do Nedgty, Bebawy et al (2005)

Além de uma interface e de um repositório, o Nedgty é composto pelos seguintes componentes: um subsistema de validação, um analisador (*parser*), uma

fila de pacotes, um filtro de mensagens SOAP e um “forjador” de pacotes. Depois que um perfil é configurado, ele é salvo (ou sobrescrito) no repositório em formato XML, permitindo a consulta do mesmo pelo subsistema de validação.

Para que o Nedgty seja transparente tanto para o cliente quanto para o servidor, os pacotes das requisições SOAP que foram validados passam pelo “forjador” de pacotes antes de serem re-enviados ao servidor. Assim o servidor acha que os pacotes foram enviados pelo remetente original.

Um ponto interessante na proposta dos autores é que pelo fato deste firewall ser orientado a objetos, ele pode ser estendido sem que a aplicação inteira necessite ser re-desenvolvida. Novos módulos podem ser adicionados ao subsistema de validação para proteger os *web services* de outros ataques não previstos na proposta inicial. Extensões também podem ser feitas no componente da interface, adicionando novos módulos que permitam ao administrador definir mais políticas de segurança.

Cada subsistema do protótipo do Nedgty foi testado separadamente utilizando casos de testes customizados durante a fase de desenvolvimento para ter certeza de que funcionavam corretamente. Depois de integrar todos os componentes, o sistema como um todo foi testado simulando ataques de negação de serviço e de *buffer overflow*. Também foi testada a política de autorização baseada na faixa de endereços IP permitidos. O firewall obteve sucesso em bloquear as mensagens indesejadas.

Como trabalho futuro os autores propuseram melhorias como a adoção de uma abordagem distribuída onde a aplicação possa executar em cada servidor e não somente em um *gateway*, evitando tráfego excessivo de rede concentrado em um ponto. Outra idéia futura não implementada pelos autores é a de vincular um sistema de detecção e prevenção de intrusão ao firewall, já que muita informação sobre as mensagens SOAP é armazenada em log e acaba não sendo utilizada. A implementação deste firewall está disponível na internet para *download* e também pode ser obtida enviando um e-mail aos autores.

Já Melzer e Jeckle (2003) propõem um proxy que pode ser adicionado a uma rede privada para prover segurança aos *web services* envolvidos. Os autores iniciam o artigo dando uma base sobre segurança de *web services*, mencionando o padrão no qual o proxy se baseia (*WS-Security*), encriptação e assinatura de documentos XML, para depois propor seu trabalho. A motivação de Melzer e Jeckle se dá no fato

de que geralmente *web services* não estão bem protegidos, e SOAP, o protocolo para troca de mensagens com os *web services* que é baseado em XML, não se preocupa muito com a segurança na comunicação e nem foi projetado para isso.

Este novo servidor que seria adicionado à rede (o proxy) seria responsável por assinar todas as mensagens SOAP que saem de um ambiente empresarial, por exemplo, com uma assinatura pertencente à empresa. Caso as mensagens passem por uma conexão não segura até chegar ao destino, o proxy também poderia encriptar alguns conteúdos das mensagens para que somente possam ser acessados pelo destinatário desejado (utilizando SSL, *Secure Sockets Layer*). A idéia então é que o firewall do destinatário (poderia ser uma empresa parceira) que estaria recebendo estas mensagens possa bloquear todas as chamadas SOAP que não contenham a assinatura apropriada ou que não consigam ser decriptadas com sucesso.

Em outras palavras, o firewall do receptor interceptaria as mensagens SOAP recebidas e, além dos passos usuais, autenticaria as mensagens utilizando a informação que foi adicionada aos cabeçalhos pelo proxy. Depois de permitida a passagem das mensagens pelo firewall, outra checagem é feita para verificar se o solicitante tem permissão para acessar o web service em questão.

A parte de autenticação é sugerida pelos autores, mas não fica claro se teria sido implementada ou se ficaria como um trabalho futuro. Essa parte seria feita utilizando PKI (*Public Key Infrastructure*) para gerar assinaturas pessoais para cada usuário autenticado na rede. Porém, logo em seguida já mencionam que esse requisito é perigoso, pois a confidencialidade de chaves privadas é muito importante e erros nesta parte da implementação poderiam comprometer a confiabilidade do sistema inteiro.

Um ponto forte da proposta dos autores é que poucas mudanças precisam ser feitas pelos usuários e administradores de uma rede. Os usuários apenas precisam alterar suas configurações de proxy para contatar o proxy de segurança de *web services* ao invés do proxy clássico. Os administradores precisam alterar as regras do firewall para forçar assinaturas válidas nos cabeçalhos das mensagens SOAP. Lembrando que a maioria das chamadas SOAP são simétricas, ou seja, recebem uma resposta. Portanto, essas alterações devem ser feitas em ambos os lados da comunicação.

Todas as aplicações então passam a participar automaticamente deste

ambiente seguro sem que precisem ter nenhuma parte dos seus códigos alterada. Porém, como ponto fraco, os autores não mencionam nenhum teste que tenha sido executado para validar a proposta, a qual também não se encontra disponível. Além disso, não especificam quais tipos de ataques a *web services* são previstos pelo proxy, apenas mencionam que ele deve protegê-los contra qualquer mensagem indesejada, o que torna a proposta um pouco genérica.

Cremonini e seus colegas (Cremonini et al, 2003) analisam os requisitos de segurança da WSA (*Web Services Architecture*), mencionando que o modelo de segurança existente não é suficiente. Mais especificamente, diz que muitos aspectos relacionados à segurança de redes e à integração de firewalls à WSA foram subestimados. A WSA é um documento definido pela W3C, com a intenção de prover uma definição comum do que são *web services* e de como ele deveria se encaixar em um framework maior de *web services*. Este documento identifica as características mínimas que devem ser comuns a todos os *web services*, e também características que são necessárias para a maioria deles, além de identificar os elementos globais de uma rede de *web services* que são necessários para a interoperabilidade entre os mesmos.

A WSA também representa uma instância específica da SOA (*Service Oriented Architecture*), que nada mais é que um sistema distribuído onde as entidades que interagem são serviços. Na WSA mensagens são trocadas com o intuito de requisitar e prover serviços, enquanto na Internet as mensagens têm o intuito de trocar informações.

O artigo se motiva no fato que apesar de já haver surgido padrões como *WS-Security* e SAML, que são padrões de segurança para *web services*, ainda existe muita preocupação na área de segurança, atrasando muitas vezes a adoção da tecnologia de *web services* em empresas. *WS-Security* define como inserir informação no envelope SOAP, e SAML define o que a informação de segurança será. Para resolver esse conflito, não adianta ficar discutindo a funcionalidade contra a segurança de *web services*, pois teríamos de um lado a WSA com excelentes padrões para segurança de serviços e mensagens, e do outro lado a comunidade de segurança de rede argumentando contra. Para construir uma WSA segura então se discutem medidas de segurança orientadas a serviços de um lado e segurança de rede do outro lado.

Service-Oriented Security é um modelo compatível com SOA e WSA, onde o

sistema global é visto como uma rede de pontos provendo serviços e fazendo requisições. Nesse modelo os itens de segurança focam em proteger as mensagens que transitam entre esses pontos e controlar a interação com esses pontos de acordo com suas definições. As tecnologias de segurança providas na WSA são perfeitamente coerentes com esse modelo, pois a entrega de mensagens através da rede pública é feita através de assinatura e criptografia de mensagens, garantindo a integridade e confiabilidade, e através do sequenciamento das mensagens utilizando IDs. Além disso, o acesso aos serviços é baseado em módulos de autorização que reforçam políticas de acesso.

Porém, no modelo de segurança orientado a serviços, as políticas de segurança apenas controlam o uso das funcionalidades de um serviço dentro do domínio definido na descrição deste serviço. Os autores defendem que existem outros itens a serem protegidos, como por exemplo, a rede privada da organização onde o provedor de serviço está localizado (visa evitar tráfego indesejado na rede que possa comprometer o serviço) e o ambiente onde o provedor de serviço está hospedado (visa proteger tudo que suporta e ajuda a prover o serviço, por exemplo, o sistema operacional).

Os autores então propõem um modelo integrado, unindo os dois modelos para obter uma infra-estrutura de segurança completa para *web services*. A política de autorização deste modelo seria formada por quatro elementos: sujeito, objeto, condição e sinal. O sujeito define a quem a política se aplica. O objeto define a qual serviço ou a quais serviços a política de aplica. A condição define restrições adicionais que devem ser satisfeitas pelo sujeito para que possa acessar o objeto. O sinal apenas define se a autorização define uma permissão (+) ou uma negação (-). Um exemplo de uma regra de autorização restringe uma sub-rede onde os *web services* provendo serviços devem estar, para evitar a instalação de serviços maliciosos na rede.

Outro exemplo especifica que se uma requisição SOAP for enviada para determinado serviço, ela tem que obrigatoriamente conter uma URL HTTP específica e um método a ser executado. Isso ajuda a evitar tráfego desnecessário por mensagens ilícitas contendo métodos falsos de *web services*. Filtrando estas mensagens, pelo menos no ambiente onde o serviço está sendo executado, não se desperdiça poder computacional.

Os autores ficam mais na teoria, não apresentam uma proposta pronta

baseada em testes, apenas dizem que um protótipo de um firewall semântico está em desenvolvimento.

3.4 CONSIDERAÇÕES

O quadro 1 relaciona os principais aspectos de cada abordagem para efeito comparativo entre as mesmas:

Quadro 1 - Comparação das propostas estudadas

Proposta	Ataques contemplados	Técnicas aplicadas	Adota algum padrão
Security Attack Ontology for Web Services	Mitnick, Probing, UDDI scanning, CDATA field attack, Web Services DoS, SOAP attacks, XML attacks e Semantic Web Service attack	Ontologia para que firewalls e IDSs distribuídos possam trabalhar em conjunto	OWL (W3C)
A Target-Centric Ontology for Intrusion Detection	DoS, User/Root access, Probing, Buffer overflow	Ontologia para que IDSs distribuídos possam trabalhar em conjunto	DAML+OIL (W3C)
An Adaptive Intrusion Detection and Prevention (ID/IP) Framework for Web Services	Parameter tampering, Recursive/Oversized payload attack, Replay attack, SOAP header attack, SQL injection, XML injection e CDATA field attack	Agentes, data mining e fuzzy logic	Não
Comprehensive Test Mechanism to Detect Attack on Web Services	Recursive/Oversized Payload attack, Parameter tampering, Buffer overflow, Replay attack, Entity expansion e XML rewriting	Sistema Java utilizando AXIS2 SOAP *engine*	Não
Intrusion Detection of DoS-DDoS and Probing Attacks for Web Services	DoS, DDoS, Probing	Quantização vetorial	Não
Preventing injection attacks with syntax embeddings	Injections (SQL, XQuery, XPath, etc.)	Incorporação de sintaxe para gerar automaticamente o código que irá prevenir vulnerabilidades	Não

Proposta	Ataques contemplados	Técnicas aplicadas	Adota algum padrão
Nedgty: Web Services Firewall	Buffer overflow, DoS, XML DoS	Firewall e um mecanismo de autorização	Não
A Signing Proxy for Web Services Security	Não menciona	Addon para proxies regulares visando proteção de Web Services, autenticação e autorização	RFC 2616, RFC 2717 e RFC 3275 (IETF), XML Encryption (W3C), SAML (OASIS) e WS-Security (OASIS)
An XML-based Approach to Combine Firewalls and Web Services Security Specifications	DoS, Spoofing, Replay attack, Malformed packets/messages, Buffer overflow, SQL injection e Trojaned services	Semantic-aware firewall e autorização	WSA (W3C), WS-Security (OASIS), SAML (OASIS)

Grande parte das propostas encontradas na literatura técnica utiliza abordagens de detecção clássicas (SIDDAVATAM e GADGE, 2008; YEE, SHIN e RAO, 2007; ZHENG e HU, 2005; BRAVENBOER, DOLSTRA e VISSER, 2010) ou não trabalham com ataques a *web services* (UNDERCOFFER et al, 2004). Outras abordagens encontradas trabalham com proteção contra ataques a *web services* (BEBAWY et al, 2005; MELZER e JECKLE, 2003; CREMONINI et al, 2003), porém não utilizam IDS e ontologia para tal.

Vorobiev e Han (Vorobiev e Han, 2006) propuseram a abordagem que está mais próxima da proposta deste trabalho. Os autores aplicaram uma ontologia especificamente para representar o domínio de ataques a *web services*. Entretanto, a implementação da ontologia não foi encontrada e a proposta não utiliza inferência para detectar variações de ataques, a ontologia é utilizada principalmente como um vocabulário comum.

Desta forma, nenhum trabalho foi encontrado na literatura técnica que aborde os objetivos específicos da proposta deste trabalho.

4 USANDO UMA ONTOLOGIA NO SISTEMA DE DETECÇÃO DE INTRUSÃO PARA WEB SERVICES

Conforme comentado anteriormente, na abordagem clássica de detecção de intrusão por anomalias um modelo é extraído usando técnicas de aproximação que classificam e agrupam ações numa classe denominada normal. Em geral, na classe normal são colocados os mais variados tipos de ações consideradas corriqueiras. Desta forma, em tempo de detecção qualquer ação (nova) que não foi catalogada no modelo como normal será inferida como ataque. Ou seja, mesmo as ações que são consideradas normais, mas não estão no modelo porque não eram consideradas como tais na época da primeira classificação, serão alertadas como ataque – esta é a principal razão da geração de altas taxas de falsos positivos nesta abordagem.

Na detecção de intrusão por assinaturas só são catalogados os *payloads* (de ataques) e, portanto, os falsos positivos são nulos (considerando o catálogo correto dos *payloads*). Porém, nesta abordagem só são detectados ataques previamente catalogados.

Na abordagem deste trabalho, cada perfil de ataque é catalogado como uma classe e suas variações como instâncias. Deste modo, a detecção acontece analogamente à abordagem por assinaturas quando uma sequência bem definida de ações representa a assinatura da estratégia de uma instância de ataque já cadastrada na ontologia. Porém, há um diferencial importante na abordagem proposta, pois as estratégias das categorias de ataques são também catalogadas (através dos axiomas das classes) e possibilitam a dedução de variações de ataques (novas instâncias) por inferência.

Assim, a proposta deste trabalho é melhorar o mecanismo de detecção de ataques a web services com a construção de uma ontologia em que o domínio de conhecimento são estes ataques e suas estratégias (XMLInjection, ProtocolManipulation e suas subclasses para fins desta proposta). O objetivo é construir um mecanismo que capture pacotes da rede e que, usando a ontologia como uma base de ataques para IDSs, detecte ataques conhecidos e deduza novos ataques através de *engines* de detecção e inferência (figura 4.1). O principio da utilização de inferência é análogo a abordagem baseada em anomalias, mas a

proposta visa eliminar as altas taxas de falsos positivos (erros de avaliação que geram alertas) observados na detecção clássica deste tipo de IDS.

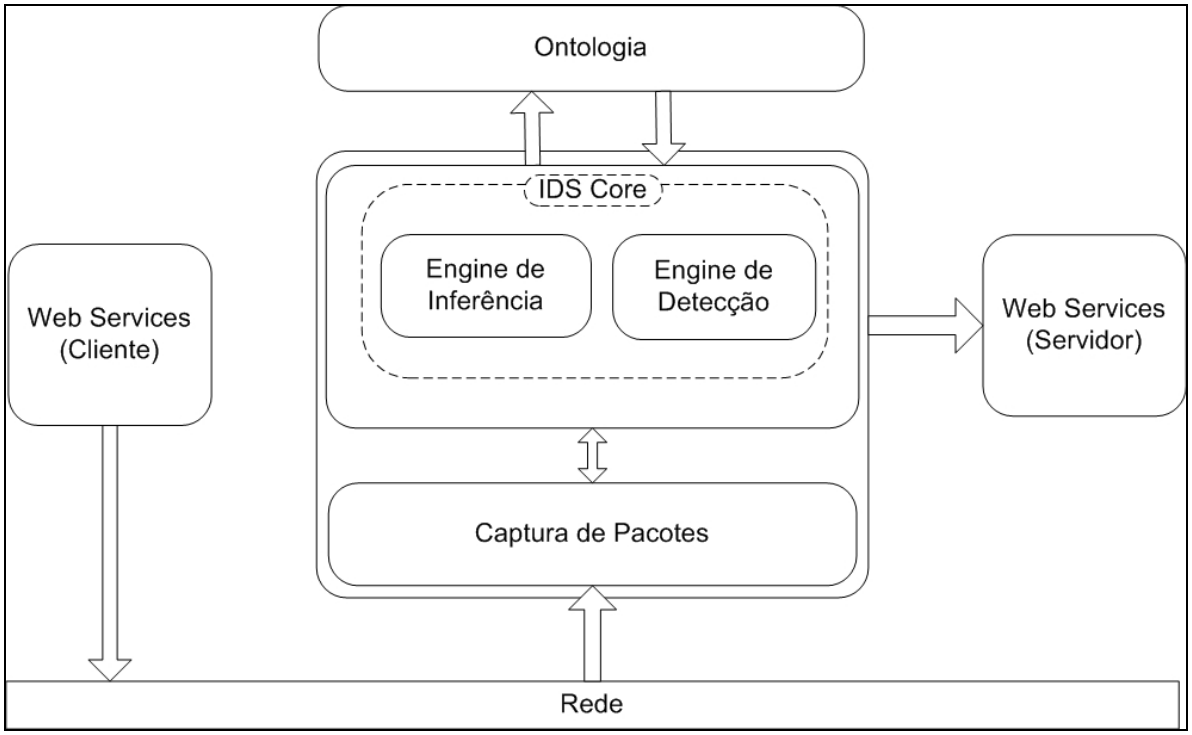


Figura 4.1 - Visão geral da proposta de IDS

Uma visão geral do funcionamento dos *engines* de detecção e de inferência do IDS (Figura 4.1) é mostrada na figura 4.2. É possível observar que quando uma ação é detectada (evento i, figura 4.2) em um pacote na rede (*payload*) pela primeira vez o IDS cria uma instância (evento ii) de ataque (por enquanto sem persistir na ontologia) e cria um relacionamento da instância com esta ação (evento iii). Ou seja, a estratégia de um possível ataque começa a ser perseguida. A forma como as ações são identificadas nos pacotes será descrita em detalhes na seção 4.1 e o algoritmo de detecção do ataque será descrito na seção 4.2.

A seguir, o IDS verifica se existe alguma instância na base de conhecimento da ontologia que seja idêntica à criada (evento iv), o que indicaria que o ataque é conhecido (v). Isto é feito através de uma busca na ontologia por uma instância de ataque que esteja relacionada com exatamente as mesmas ações encontradas na detecção até este ponto.

Quando não é encontrada uma instância idêntica à criada no evento ii, o IDS tenta inferir um novo ataque a partir das informações contidas na base de conhecimento da ontologia (evento vi), verificando se a instância pode ser

considerada uma variação de ataque. É através desta inferência, que leva em conta classes e axiomas na ontologia, que a abordagem tenta aprender um novo ataque e adicioná-lo na base de conhecimento (processo detalhado na seção 4.1), mitigando ataques *zero-day*. Espera-se que esta dedução de novos ataques não gere falsos positivos, já que é feita baseando-se em informações contidas na ontologia.

Não chegando a uma inferência conclusiva, o IDS continua analisando os próximos pacotes até encontrar uma nova ação. Esta nova ação então é adicionada à instância (que agora contém duas ações), e novamente verifica-se a possibilidade de um ataque ter ocorrido.

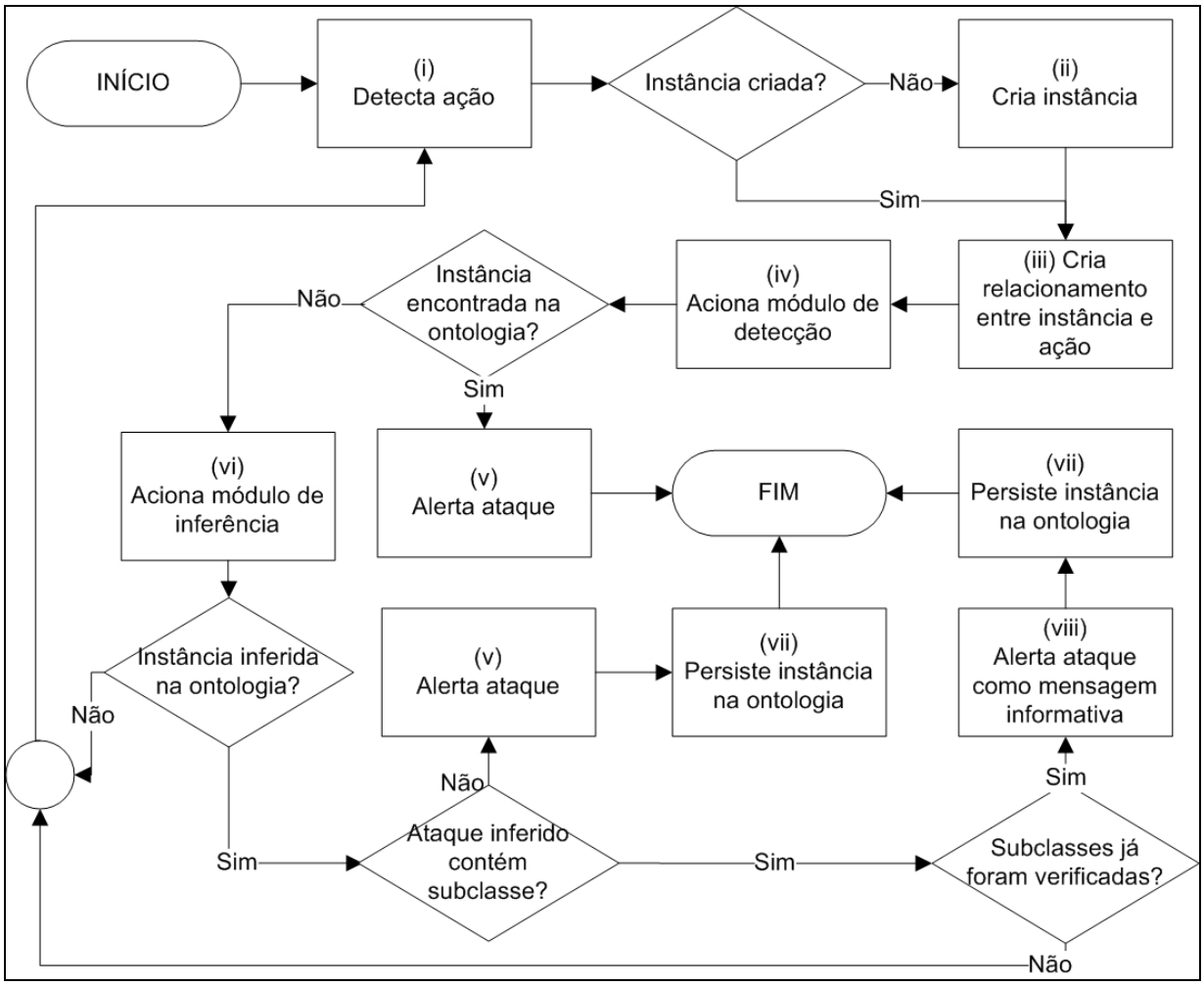


Figura 4.2 - Fluxograma de detecção do protótipo de IDS

Este ciclo de eventos ocorre até que uma instância seja apontada como um ataque, de acordo com o conjunto de ações detectado, quando então a sequência do algoritmo de detecção é reiniciada.

Um ataque pode ser alertado pelo IDS (evento v) se for encontrada uma instância idêntica na ontologia ou se for inferida uma instância como um novo ataque. Quando um ataque é inferido, antes de alertar o ataque, o IDS verifica se a classe da instância inferida contém subclasses, o que significa que há possibilidade de se tratar de um ataque mais específico. Caso encontre subclasses na ontologia, o IDS aguarda a próxima ação ser detectada e faz uma nova inferência. Se esta nova inferência não alertar nenhuma subclasse mais específica, o IDS alerta o ataque inferido inicialmente como uma mensagem informativa, o que significa que o ataque pode não estar completo ou que se trata de uma nova categoria (subclasse) de ataques. Em caso contrário o IDS alerta o ataque – porque a subclasse mais específica foi alcançada.

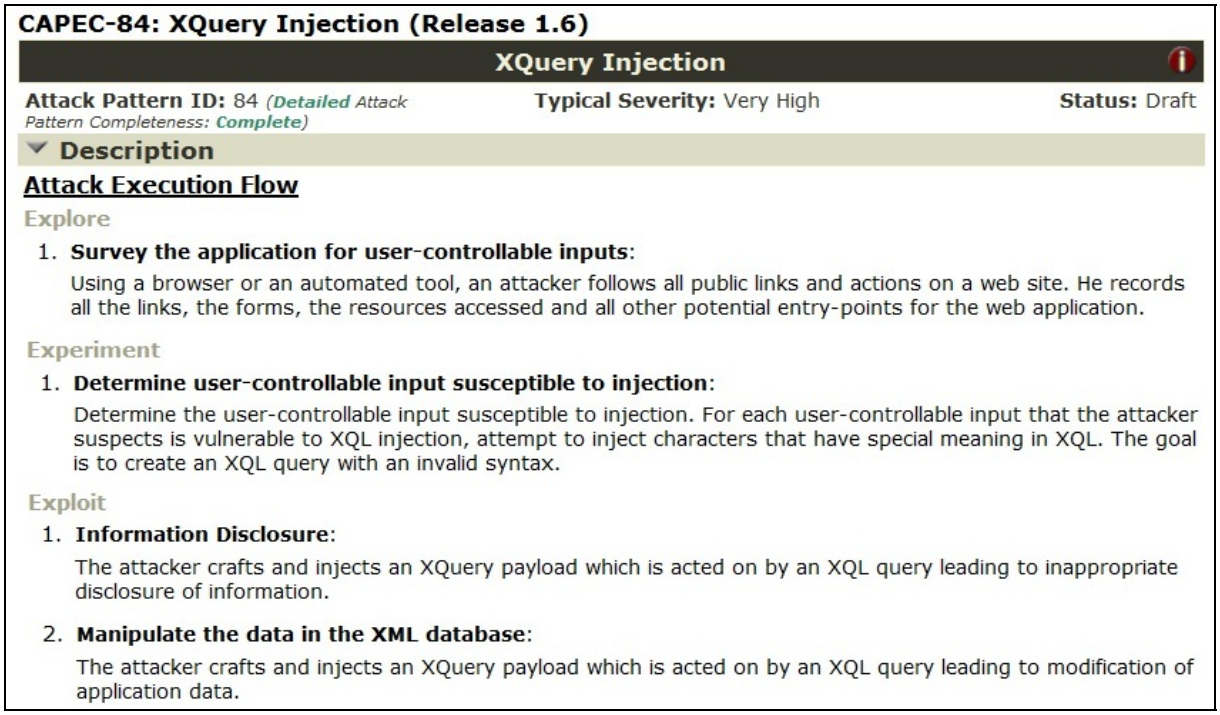
Além de emitir o alerta, o IDS adiciona esta nova instância à classe correspondente na base de conhecimento da ontologia (evento vii). Assim, o par protótipo de IDS e ontologia podem ser considerados um sistema de detecção com uma abordagem híbrida. A proposta permite detecção baseada em assinaturas através das instâncias (sequência bem definida de ações), mas também permite a detecção de variações de ataques através de inferência nas classes e axiomas, fazendo analogia à detecção clássica baseada em anomalias.

É bom ressaltar que a proposta não prevê possíveis ataques de negação de serviço ao IDS, que poderiam ser executados explorando as repetidas inferências e detecções na sequência do fluxograma (Figura 4.2). Pretende-se prevenir estas possíveis falhas em trabalhos futuros.

4.1 ONTOLOGIA BASEADA EM TAXONOMIA DE ATAQUES

Para satisfazer os critérios propostos por Gruber (GRUBER, 1993, p. 908-909) na construção da ontologia, utilizou-se inicialmente a taxonomia de ataques apresentada pelo *website* da CAPEC (CAPEC, 2011). A CAPEC descreve mecanismos utilizados para explorar falhas de software de acordo com a perspectiva do atacante. Esta descrição é feita em alto nível, como se pode ver na figura 4.3, por isto a ontologia foi refinada baseando-se também em ferramentas de teste de

segurança. A figura 4.3 mostra um fragmento da descrição da classe de ataque *XQueryInjection* na CAPEC.



CAPEC-84: XQuery Injection (Release 1.6)

XQuery Injection

Attack Pattern ID: 84 *(**Detailed** Attack Pattern Completeness: **Complete**)* **Typical Severity:** Very High **Status:** Draft

Description

Attack Execution Flow

Explore

1. **Survey the application for user-controllable inputs**:
   Using a browser or an automated tool, an attacker follows all public links and actions on a web site. He records all the links, the forms, the resources accessed and all other potential entry-points for the web application.

Experiment

1. **Determine user-controllable input susceptible to injection**:
   Determine the user-controllable input susceptible to injection. For each user-controllable input that the attacker suspects is vulnerable to XQL injection, attempt to inject characters that have special meaning in XQL. The goal is to create an XQL query with an invalid syntax.

Exploit

1. **Information Disclosure**:
   The attacker crafts and injects an XQuery payload which is acted on by an XQL query leading to inappropriate disclosure of information.
2. **Manipulate the data in the XML database**:
   The attacker crafts and injects an XQuery payload which is acted on by an XQL query leading to modification of application data.

Figura 4.3 - Ataque *XQueryInjection* parcial, CAPEC (CAPEC, 2011)

4.1.1 Definindo a ontologia

A ontologia proposta é composta de classes e propriedades (figura 4.4), instâncias (figura 4.5) e axiomas. As duas superclasses da ontologia são *AttackAction* e *WebServicesAttack*. *AttackAction* possui subclasses contendo instâncias que representam ações suspeitas (*payloads*) que podem ser capturadas na rede.

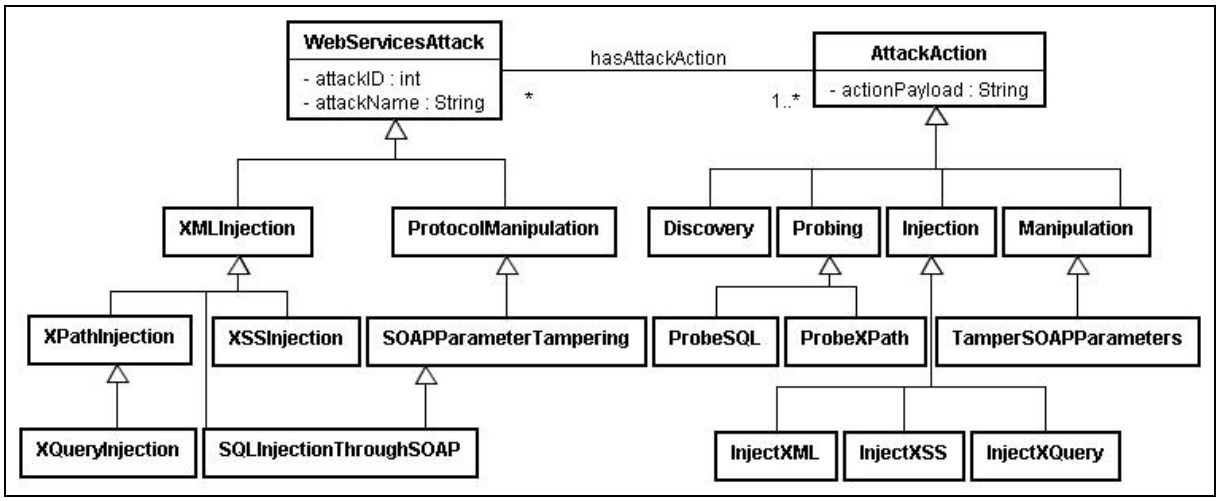


Figura 4.4 - Diagrama de classes da ontologia

A classe *WebServicesAttack* possui subclasses representando categorias de ataques a *web services*. Cada uma destas subclasses possui instâncias representando assinaturas (estratégias específicas) de ataques conhecidos. Na ontologia, a assinatura de uma instância de ataque é representada por relacionamentos − que a mesma tem com ações − através da propriedade *hasAttackAction*. Uma instância de ataque pode ter uma ou mais ações e uma ação pode ser parte de várias instâncias de ataque. Todas estas informações, juntamente com os axiomas das classes que serão descritos nesta seção, formam a base de conhecimento da ontologia.

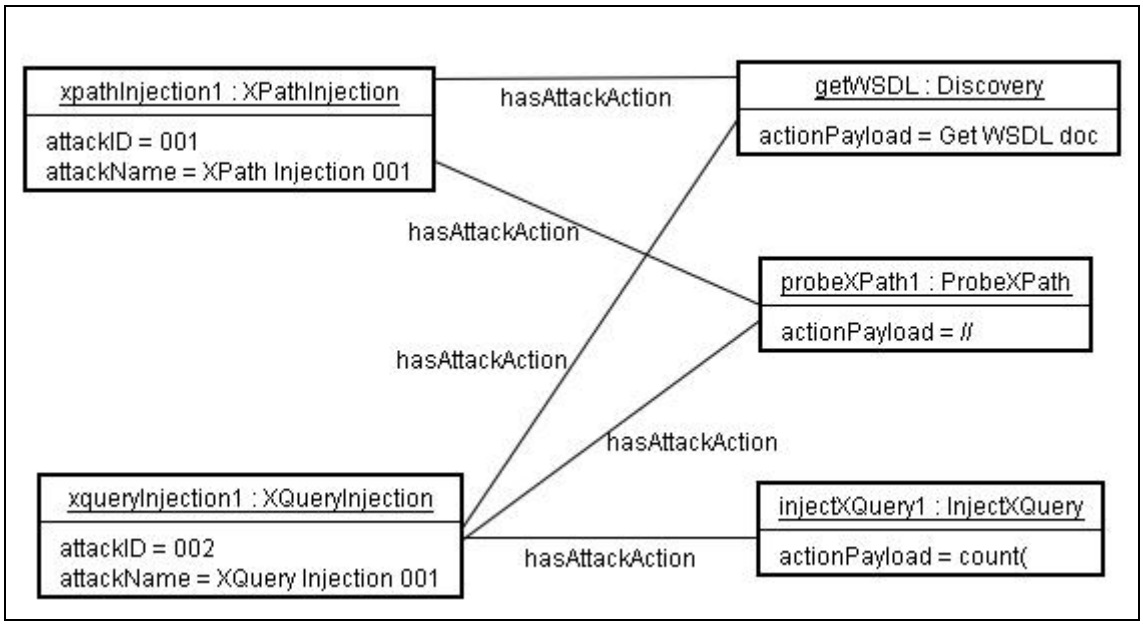


Figura 4.5 - Exemplos de instâncias da ontologia

A figura 4.5 apresenta dois exemplos de instâncias da ontologia, a *xpathInjection1* (instância da subclasse de ataques *XPathInjection*) e a *xqueryInjection1* (instância da subclasse de ataques *XQueryInjection*). A instância *xpathInjection1* tem relacionamentos (*hasAttackAction*) com as ações *getWSDL* (instância da subclasse de ações *Discovery*) e *probeXPath1* (instância da subclasse de ações *ProbeXPath*). Enquanto que *xqueryInjection1* tem relacionamentos com as mesmas ações de *xpathInjection1* e também com a ação *injectXQuery1* (instância da subclasse de ações *InjectXQuery*).

As classes que foram modeladas abaixo da superclasse *WebServicesAttack* representam tudo o que é considerado ataque para o protótipo de IDS, e tudo o que não satisfizer as restrições (axiomas) destas classes não é considerado um ataque. Esta consideração é feita porque o IDS baseado em anomalias considera no mínimo duas classes, uma de ataques e outra normal.

Tradicionalmente, a abordagem adotada para criar as classes em IDSs baseados em anomalias é a de criar a classe normal e considerar todo o restante ataque. Esta é uma razão pela qual IDSs baseados em anomalias geram muitos alertas falsos, pois havendo alterações nas ações normais, mesmo que pouco significantes, essas não serão mais consideradas normais e gerarão falsos alertas de ataque.

Na abordagem proposta os ataques foram modelados, somente considerando o que é conhecidamente ataque ou deriva de um. É importante mencionar que esta estratégia é considerada nos IDSs baseados em assinaturas, a diferença é que em tal abordagem não há inferência e, portanto, novos ataques não serão deduzidos.

Na ontologia, os axiomas definidos para uma classe (categoria de ataque) restringem o número e o tipo de ações que as instâncias daquela classe terão. Além disso, axiomas também podem ajudar máquinas de inferência a deduzir se um tipo de ataque ocorreu quando o padrão identificado (instância) ainda não está na base de conhecimento da ontologia, permitindo que este novo conhecimento seja adicionado à ontologia a partir desta dedução. A máquina de inferência utilizada neste trabalho pelo protótipo de IDS foi o Pellet (CLARK&PARSIA, 2011).

Um exemplo de axioma (1) criado na ontologia para a classe *XQueryInjection*, representado com lógica de primeira ordem, é apresentado a seguir.

“XQueryInjection ≡ ∃hasAttackAction.Discovery ⊓ ∃hasAttackAction.ProbeXPath ⊓ ∃hasAttackAction.InjectXQuery” (1)

Este axioma instrui a máquina de inferência a deduzir que qualquer ataque possuidor de uma ação do tipo *Discovery*, uma ação do tipo *ProbeXPath*, e uma ação do tipo *InjectXQuery* terá que obrigatoriamente ser uma instância da classe *XQueryInjection*. Isto, na lógica de detecção do IDS, significa que um ataque do tipo *XQueryInjection* ocorreu.

Os axiomas foram modelados para cada classe de ataque baseando-se nas estratégias de ataque propostas pela CAPEC, e foram validados/ajustados baseando-se em ferramentas de teste de segurança para *web services* (seção 4.2). A CAPEC descreve a estratégia da categoria de ataque *XQueryInjection* (figura 4.3) na perspectiva do atacante em três passos:

- ***Explore***: explorar a aplicação (neste caso *web services*) procurando por parâmetros controlados pelo usuário que possam servir de porta de entrada para um atacante. Este passo é refletido na primeira parte do axioma (1) e suas possíveis ações aparecem na classe *Discovery* na ontologia.
- ***Experiment***: determinar os parâmetros controlados pelo usuário que possam fazer parte de uma *query* XQuery e tentar determinar a estrutura destas *queries*. Isto pode ser feito inserindo caracteres inválidos para a linguagem XPath e observando as respostas do servidor. Este passo é refletido na segunda parte do axioma (1) e suas possíveis ações estão descritas na classe *ProbeXPath* na ontologia.
- ***Exploit***: submeter conteúdo malicioso a estes parâmetros contendo expressões XQuery que visam roubar ou modificar informações na base de dados da aplicação. Este passo é refletido na terceira parte do axioma (1) e suas possíveis ações estão representadas na classe *InjectXQuery* na ontologia.

Um exemplo do uso do axioma (1) é mostrado a seguir, quando os pacotes (2), (3) e (4) foram detectados pelo protótipo de IDS (seção 4.2) antes da instância de ataque *xqueryInjection1* ser adicionada na ontologia.

O pacote (2) representa um usuário acessando o documento WSDL de um *web service*, o que fez com que o protótipo criasse um relacionamento (através da propriedade *hasAttackAction*) com a ação específica *getWSDL* (figura 4.5) − uma das instâncias da classe *Discovery* na ontologia.

“*GET* /WSDigger_WS/WSDigger_WS.asmx*?wsdl* HTTP/1.0\r\n”	(2)

O pacote (3) contém caracteres (‘//’) que não deveriam ser digitados em um campo de usuário para uma operação XPath, o que fez com que o IDS criasse outro relacionamento, desta vez com a ação *probeXPath1* (figura 4.5) − instância da classe *ProbeXPath* que representa o *payload* “//”.

“<query>//*</query>” (3)

E finalmente, o pacote (4) contém o *payload* “count(”, que representa um usuário tentando obter a quantidade de algum elemento da estrutura da base de dados XML. Isto fez com que o IDS criasse um terceiro relacionamento, com a ação *injectXQuery1* (figura 4.5), instância da classe *InjectXQuery* na ontologia.

“<query>*count(/child::node())*</query>” (4)

O IDS então inferiu na ontologia, utilizando o Pellet, se este conjunto de ações poderia representar um ataque, já que nenhuma instância de ataque contendo relacionamentos com estas ações específicas havia sido encontrada na ontologia.

Foi possível observar que mesmo a base de conhecimento da ontologia não contendo esta instância de ataque pré-cadastrada, a máquina de inferência considerou o conjunto de eventos capturados como uma nova instância da classe *XQueryInjection* – de acordo com o axioma definido. Isto permitiu à ontologia aprender um novo ataque, pois em seguida o mesmo foi automaticamente adicionado à base de conhecimento na forma de uma instância da classe *XQueryInjection*. Ou seja, na próxima vez que este mesmo ataque ocorrer será detectado sem que a máquina de inferência precise ser acionada.

Abaixo estão listadas todas as classes que contêm instâncias de ações (*payloads*) que foram incluídas na base de conhecimento da ontologia.

- ***Discovery***: classe que contém instâncias de ações representando tentativas de mapear a estrutura de *web services* por um atacante, seja manualmente ou de forma automatizada. Exemplos de ações são o acesso ao documento WSDL de *web services* e o uso de uma ferramenta para buscar e armazenar todos os links disponíveis de forma automatizada. Esta segunda ação é detectada pelo protótipo caso o intervalo de tempo entre requisições *GET* seja muito curto ou estas requisições sigam um padrão de intervalo constante (e.g. requisições *GET* a cada 0.8 segundos), o que não deve acontecer quando um humano está acessando *web services*.

- ***Manipulation***: classe que contém instâncias de ações representando alguma manipulação do protocolo SOAP. Um exemplo seria a referência a uma entidade externa encontrada em um pacote XML. Neste caso o protótipo procura pelo padrão “<!ENTITY [entity_name] SYSTEM [uri_or_file_path]>”.
- ***TamperSOAPParameters***: subclasse da classe *Manipulation* que contém instâncias de ações representando mensagens SOAP com valores de campos de usuário diferentes do que o servidor está esperando (*XML Schema*), por exemplo, números em campos string e vice-versa. Para detectar esta ação o protótipo analisa pacotes de resposta do servidor buscando pelo padrão “<faultcode> ... </faultcode>” ou “<faultstring> ... </faultstring>”.
- ***ProbeXPath***: subclasse da classe *Probing* que contém instâncias de ações representando caracteres inválidos para XPath. Neste caso o protótipo procura por caracteres como "//", "|", "::", "[", "]", etc.
- ***ProbeSQL***: subclasse da classe *Probing* que contém instâncias de ações representando caracteres inválidos para SQL. O protótipo detecta estas ações quando encontra em campos de usuário caracteres como "--", ";", "/*", "*/", "<>", "<>", """, etc.
- ***InjectXML***: subclasse da classe *Injection* que contém instâncias de ações representando *strings* contendo meta-caracteres XML. Esta ação é detectada quando expressões contendo *strings* como “<!--”, “-->”, “<”, “>”, “</”, etc. são encontradas em campos de usuário.
- ***InjectXQuery***: subclasse da classe *Injection* que contém instâncias de ações representando *strings* contendo padrões exclusivos da linguagem *XPath/XQuery*, como por exemplo "count(", "node()", "text()", "comment()", "processing-instruction()", etc.
- ***InjectXSS***: subclasse da classe *Injection* que contém instâncias de ações representando *strings* utilizadas para ataques de XML *Cross-Site Scripting*. Por exemplo: "<![cdata[", "]]>", "%3cscript", "%3c%2fscript%3e", etc.

A seguir são apresentados, em lógica de primeira ordem, os demais axiomas que foram criados para as classes de ataque da ontologia.

O axioma (a) foi criado para a classe *ProtocolManipulation*, e significa que qualquer instância de ataque que possua relacionamento com ao menos uma ação do tipo *Manipulation* deverá ser instância da classe *ProtocolManipulation*.

$$\text{ProtocolManipulation} \equiv \exists \text{hasAttackAction.Manipulation} \quad \text{(a)}$$

O axioma (b) foi criado para a classe *SOAPParameterTampering*, e significa que qualquer instância de ataque que possua relacionamento com ao menos uma ação do tipo *Discovery* e possua relacionamento com ao menos uma ação da classe *TamperSOAPParameters* deverá ser instância da classe *SOAPParameterTampering*.

$$\text{SOAPParameterTampering} \equiv \exists \text{hasAttackAction.Discovery} \sqcap \exists \text{hasAttackAction.TamperSOAPParameters} \quad \text{(b)}$$

O axioma (c) foi criado para a classe *SQLInjectionThroughSOAP*. Este axioma instrui a máquina de inferência a deduzir que qualquer instância de ataque que possua relacionamento com ao menos uma ação do tipo *Discovery*, uma ação do tipo *TamperSOAPParameters* e uma ação do tipo *ProbeSQL*, terá que obrigatoriamente ser instância da classe *SQLInjectionThroughSOAP*.

$$\text{SQLInjectionThroughSOAP} \equiv \exists \text{hasAttackAction.Discovery} \sqcap \exists \text{hasAttackAction.TamperSOAPParameters} \sqcap \exists \text{hasAttackAction.ProbeSQL} \quad \text{(c)}$$

O axioma (d) foi criado para a classe *XMLInjection*. Este axioma permite a máquina de inferência deduzir que qualquer instância de ataque que possua relacionamento com ao menos uma ação do tipo *Probing*, ou com ao menos uma ação do tipo *Injection*, já pode ser considerada instância da classe de ataque *XMLInjection*.

$$\text{XMLInjection} \equiv \exists \text{hasAttackAction.Probing} \sqcup \exists \text{hasAttackAction.Injection} \quad \text{(d)}$$

O axioma (e) foi criado para a classe de ataque *XPathInjection*, e significa que qualquer instância de ataque que possua relacionamento com ao menos uma ação do tipo *Discovery* e uma ação do tipo *ProbeXPath* será uma instância da classe *XPathInjection*.

$$\text{XPathInjection} \equiv \exists \text{hasAttackAction.Discovery} \sqcap \exists \text{hasAttackAction.ProbeXPath} \quad \text{(e)}$$

E finalmente, o axioma (f) foi criado para a classe *XSSInjection*. Este axioma instrui a máquina de inferência a deduzir que qualquer instância de ataque que possua relacionamento com ao menos uma ação do tipo *Discovery* e ao menos uma

ação do tipo *InjectXSS*, terá que obrigatoriamente ser instância da classe de ataque *XSSInjection*.

$$XSSInjection \equiv \exists hasAttackAction.Discovery \sqcap \exists hasAttackAction.InjectXSS \quad (f)$$

4.1.2 Implementando a ontologia

A ontologia proposta foi criada utilizando a ferramenta Protégé (STANFORD, 2011) e representada utilizando a linguagem OWL (MCGUINNESS e HARMELEN, 2004). A OWL, uma das linguagens suportadas pelo Protégé, provê um vocabulário com uma semântica formal, permitindo que a ontologia desenvolvida possa ser compreendida por outras ferramentas e aplicações.

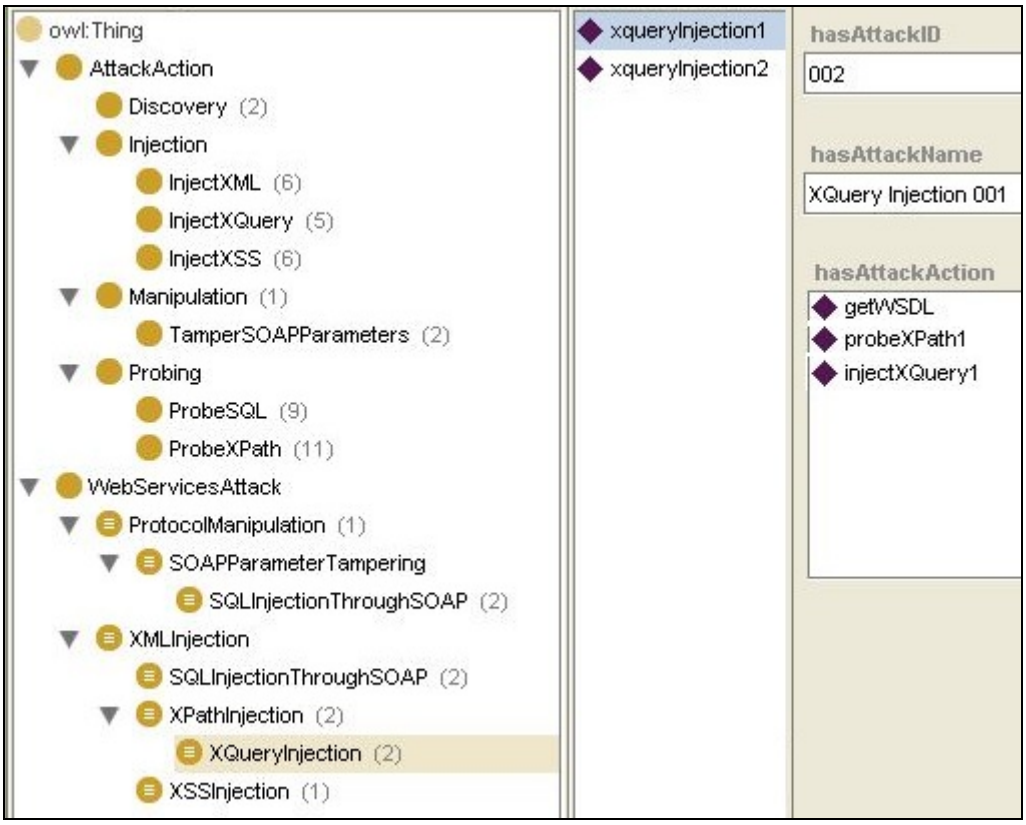


Figura 4.6 - Ontologia proposta no Protégé

O lado esquerdo da figura 4.6 mostra a estrutura das classes da ontologia no Protégé, criada baseando-se na figura 4.4. O restante da figura 4.6 mostra um exemplo de instância de ataque (*xqueryInjection1*) e suas propriedades e relacionamentos (lado direito da figura 4.6).

As instâncias de ataques e seus relacionamentos na ontologia podem ser comparados com os padrões de ataque conhecidos em uma abordagem de detecção baseada em assinaturas. Além disso, as classes e axiomas permitem que a máquina de inferência deduza que um ataque ocorreu mesmo que ele ainda não esteja na base de conhecimento da ontologia, fazendo analogia à abordagem de detecção baseada em anomalias.

Na fase de construção da ontologia a máquina de inferência pode sugerir mudanças estruturais e apontar inconsistências, baseando-se nos axiomas criados para as classes. Desta maneira, a máquina de inferência Pellet foi utilizada (BECHHOFER, 2006) no Protégé para avaliar a estrutura da ontologia construída.

Primeiramente, as classes *XQueryInjection* e *XPathInjection* estavam no mesmo nível (classes irmãs) abaixo da classe *XMLInjection*, como sugerido pela taxonomia da CAPEC. Entretanto, depois de construída a ontologia e executado o Pellet, este sugeriu que *XQueryInjection* deveria ser uma subclasse de *XPathInjection*. Depois de analisar tal inferência foi possível concluir que esta sugestão faz sentido, já que a *XQueryInjection* possui todas as restrições da *XPathInjection*. Também é possível encontrar fundamentação para isto no site da W3C (BOAG et al, 2011), que sugere que a linguagem XQuery seja uma extensão da XPath.

Outra mudança estrutural sugerida pelo Pellet foi que a classe *SQLInjectionThroughSOAP* (que é subclasse de *SOAPParameterTampering*, como sugerido pela CAPEC) também poderia ser uma subclasse de *XMLInjection*. Esta mudança também é coerente, já que os axiomas criados para a classe *SQLInjectionThroughSOAP* satisfazem tanto as restrições da classe *XMLInjection* quanto as da classe *SOAPParameterTampering*.

A inferência, nestes casos, ajudou a aperfeiçoar a organização das classes de ataque na ontologia e, por conseguinte, tornar os resultados da detecção mais efetivos.

4.2 PROTÓTIPO DE IDS

O protótipo de IDS, construído para avaliar a ontologia proposta, foi desenvolvido utilizando a tecnologia Java (ORACLE, 2011). A arquitetura deste protótipo é apresentada na figura 4.7. O protótipo implementado é um HIDS (*Host-based Intrusion Detection System*) e considera o tráfego para o *web service* sem criptografia.

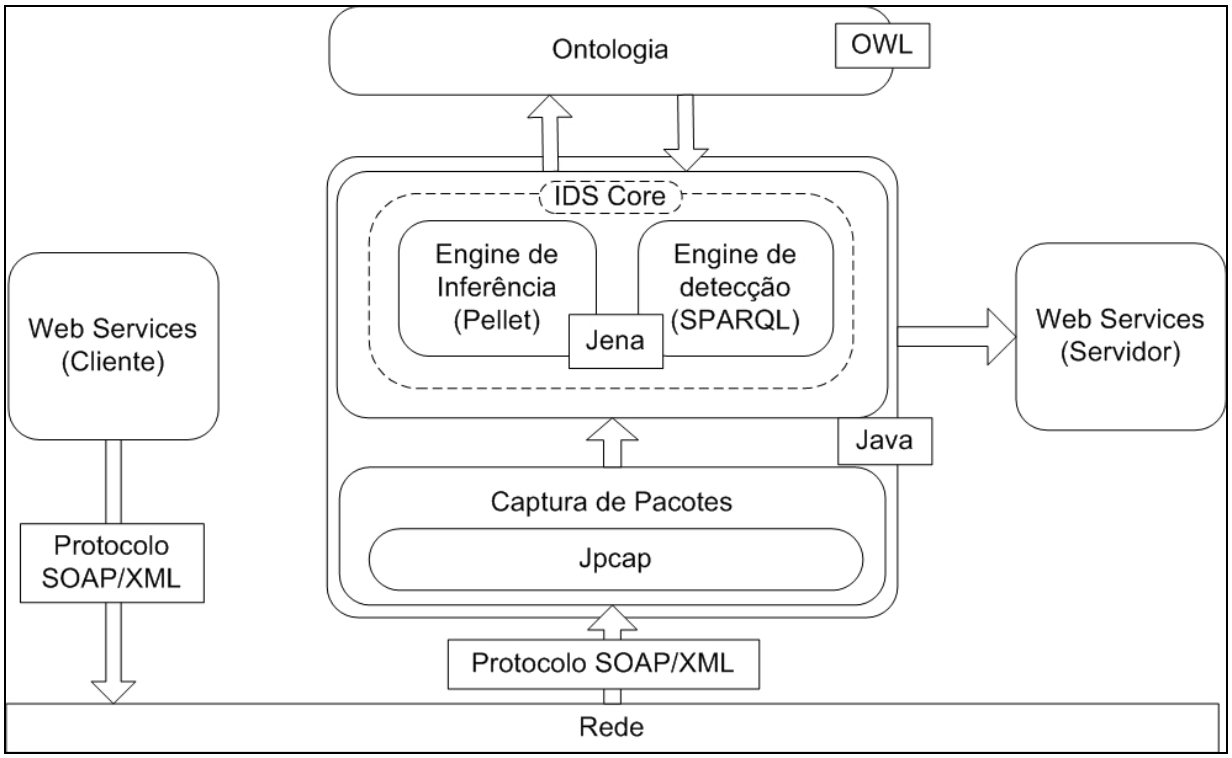


Figura 4.7 - Visão geral da arquitetura do protótipo de IDS

Para capturar pacotes na rede foi utilizada a Jpcap (SOURCEFORGE, 2011), uma biblioteca de captura de pacotes de rede para aplicações Java. A Jpcap pode filtrar pacotes IP e TCP, que são então transferidos para o módulo de detecção que analisa somente conteúdo relativo a *web services* – conteúdo XML, para fins de detecção do protótipo de IDS.

A base de conhecimento da ontologia foi manipulada pelo IDS utilizando o *framework* Jena (SOURCEFORGE, 2011), do qual o Pellet já possui uma interface nativa, e foi consultada utilizando SPARQL (PRUD'HOMMEAUX e SEABORNE,

2008), uma linguagem para consulta em arquivos RDF e OWL (arquivo da ontologia). Utilizou-se um arquivo de texto como base de dados (*payloads*) baseada em assinaturas para comparação de desempenho (seção 5.1), obtido da base de regras do Snort (SOURCEFIRE, 2011).

As estratégias dos ataques, representadas na ontologia com classes, relacionamentos e axiomas, foram estudadas e testadas utilizando o *framework* Metasploit (MOORE, 2011), a ferramenta WSDigger (MCAFEE FOUNDSTONE, 2005) e as ferramentas de teste de segurança WSFuzzer (ANDREU e BANCIU, 2011) e WebScarab (DAWES, 2011) sugeridas pelo OWASP (OWASP, 2011). Também foram utilizados scripts contidos no website ha.ckers (HANSEN, 2011) para gerar ataques de XML *Cross-Site Scripting* (*XSS Injection*). O uso destas ferramentas também auxiliou na validação do mecanismo de detecção do protótipo de IDS.

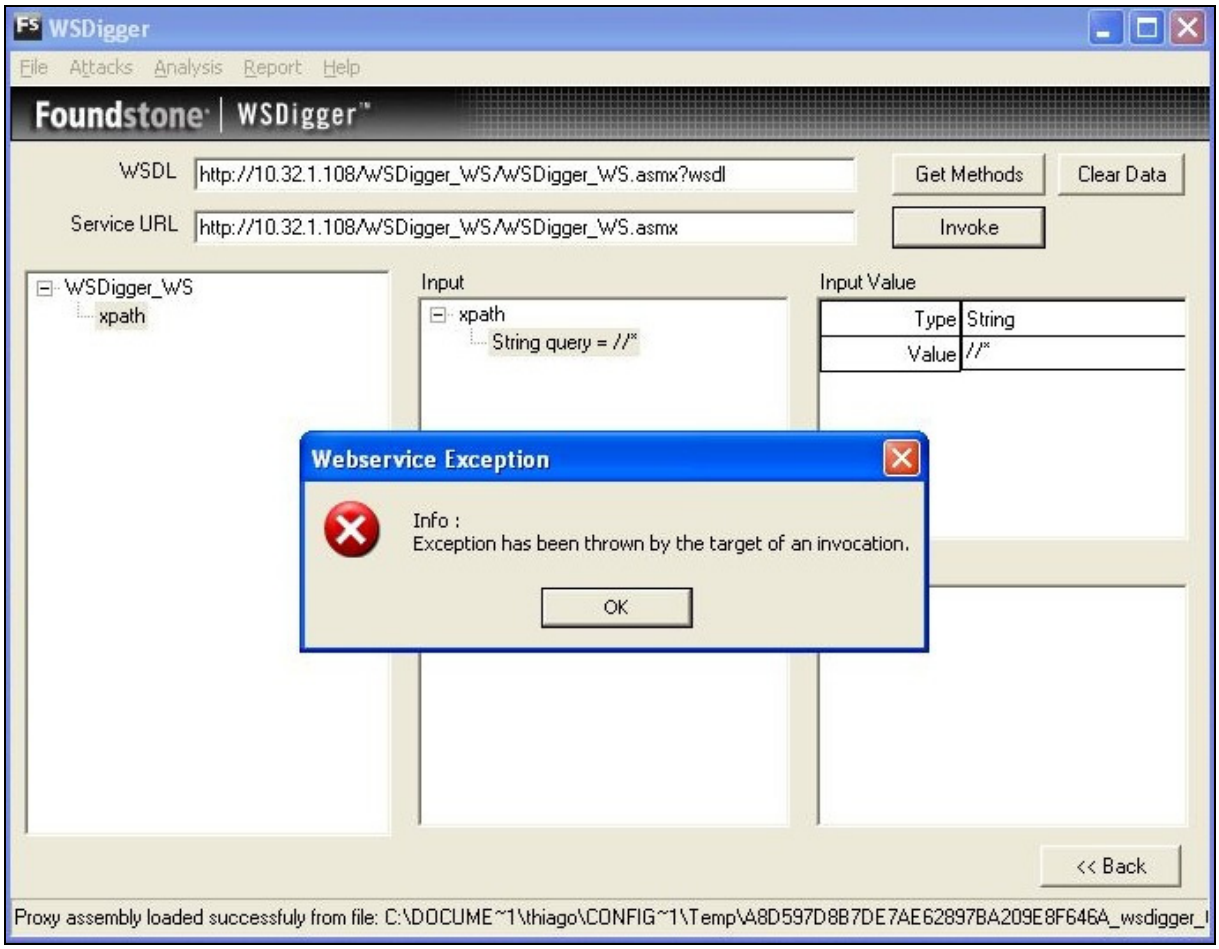


Figura 4.8 - Instância *xqueryInjection1* parcial no WSDigger

Por exemplo, a figura 4.8 mostra a interface da ferramenta WSDigger sendo usada para executar as duas primeiras ações da instância de ataque *xqueryInjection1* (instância utilizada como exemplo na seção 4.1.1). Primeiramente informou-se o endereço do documento WSDL para revelar os métodos e parâmetros disponíveis através do botão "*Get Methods*" (ação *getWSDL*). Em seguida a *string* "//*" foi submetida ao parâmetro *query* através do botão "*Invoke*" (ação *probeXPath1*). A mensagem de erro indica que o alvo gerou uma exceção após o ataque, ou seja, o conteúdo submetido ao parâmetro sendo testado não é corretamente validado antes de chegar ao servidor.

A terceira e última ação (*injectXQuery1*) foi capturada no *payload* "*count(/child::node())*", que foi submetido ao mesmo parâmetro *query* logo em seguida visando extrair informações da base de dados XML do *web service*. Esta sequência de três ações, conforme comentado na seção 4.1.1, fez com que o protótipo inferisse um novo ataque e o adicionasse à classe *XQueryInjection* na ontologia.

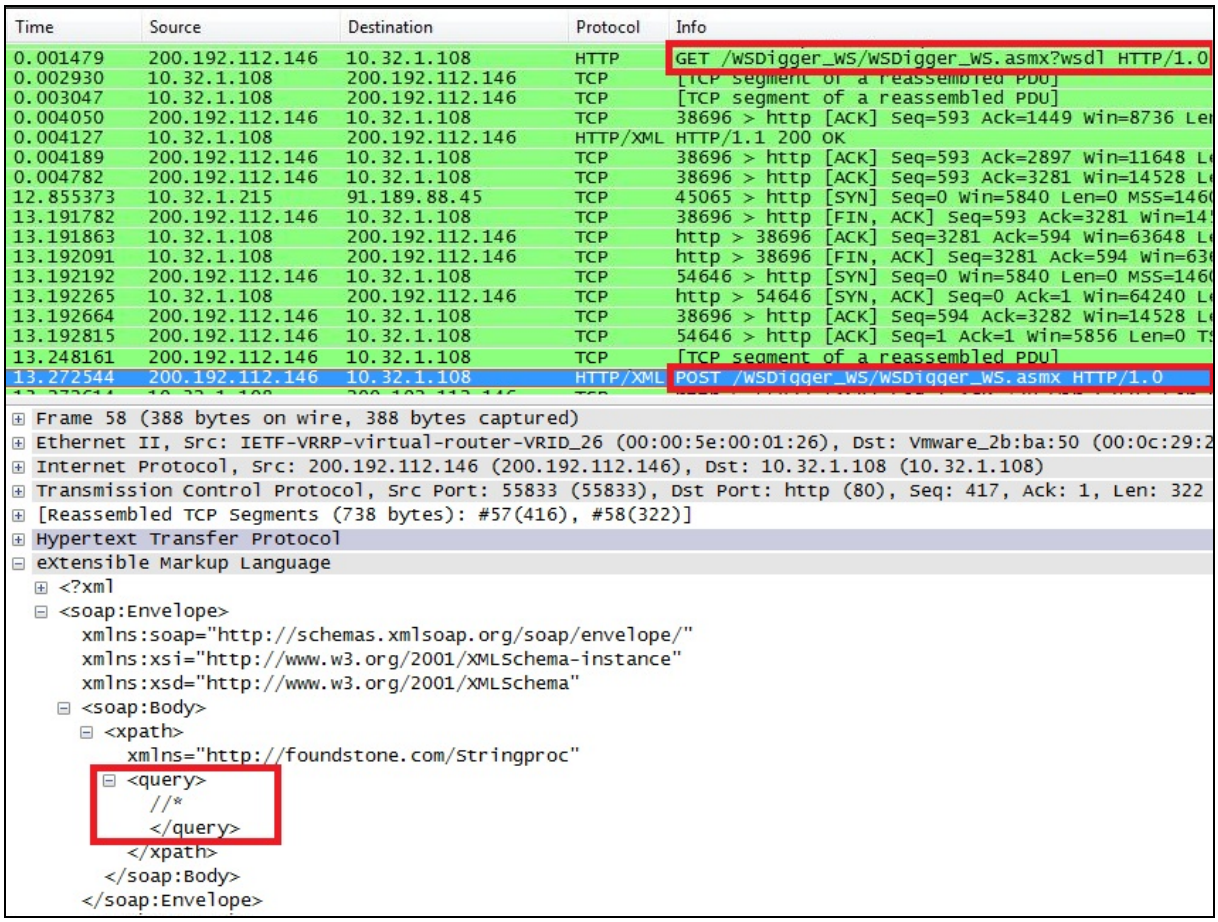


Figura 4.9 - Ações *getWSDL* e *probeXPath1* no Wireshark

O *sniffing* Wireshark (COMBS, 2011) foi utilizado para capturar pacotes na rede, visando definir corretamente as instâncias de ações na ontologia e analisar a forma como os pacotes chegam ao protótipo de IDS. As figuras 4.9 e 4.10 mostram fragmentos de captura no Wireshark representando as ações desta mesma instância *xqueryInjection1*.

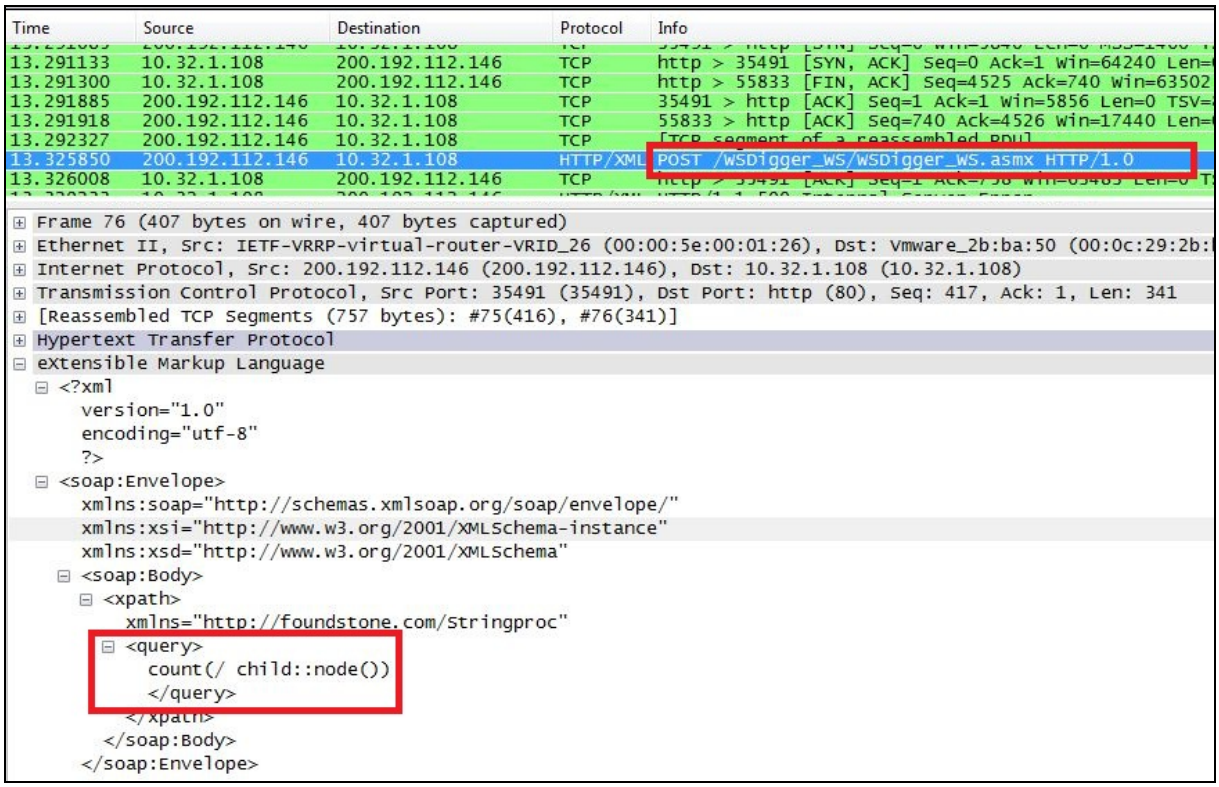


Figura 4.10 - Ação *injectXQuery1* no Wireshark

A seguir é apresentada a principal classe Java do protótipo de IDS, através de pseudocódigo:

1 **Variáveis**
2 jpcap: capturador de pacotes;
3 ataque: instância de ataque;
4 classe: classe de ataque;
5 ação: instância de ação capturada;
6 instânciaCriada, detectado: booleano;
7 analisador: módulo analisador de pacotes;
8 pellet: módulo Pellet;
9 sparql: módulo SPARQL;

```
10      ontologia: base da ontologia;
11 Início
12 instânciaCriada:=falso;
13 detectado:=falso;
14      Enquanto(verdadeiro)
15              ação:=analisador.analisar(jpcap.próximoPacote()); //captura de pacotes (fig.4.7)
16              Se(ação!=nulo)
17                      Se(!instânciaCriada)
18                              ataque:=novaInstância();
19                              classe:=novaClasse();
20                              instânciaCriada:=verdadeiro;
21                      Fim se
22                      ataque.adicionaAção(ação);
23                      detectado:=sparql.verifica(ataque); //engine de detecção (fig.4.7)
24                      Se(detectado)
25                              Escreva(‘Ataque detectado: ’+ataque);
26                              instânciaCriada:=falso;
27                      Senão
28                              detectado:=pellet.verifica(ataque); //engine de inferência (fig.4.7)
29                              Se(detectado)
30                                      Se(!ontologia.contemSub(ataque.Classe()))
31                                              Escreva(‘Ataque inferido: ’+ataque);
32                                              ontologia.adiciona(ataque);
33                                              instânciaCriada:=falso;
34                                      Senão
35                                              Se(ataque.Classe()==classe)
36                                                      Escreva(‘Ataque genérico inferido: ’+ataque);
37                                                      ontologia.adiciona(ataque);
38                                                      instânciaCriada:=falso;
39                                              Senão
40                                                      classe:=ataque.Classe();
41                                              Fim se
42                                      Fim se
43                              Fim se
44                      Fim se
45              Fim se
46      Fim enquanto
47 Fim
```

Como se pode observar, quando uma ação é detectada pela primeira vez em um pacote capturado (linha 15), o IDS cria uma instância de ataque e adiciona esta ação à mesma (linhas 18 e 22). Por enquanto esta instância ainda não foi persistida na base de conhecimento da ontologia.

A seguir o IDS chama o módulo SPARQL (linha 23) para verificar se existe alguma instância de ataque na base da ontologia que seja idêntica à criada, alertando este ataque (linha 25). Se não for encontrada uma instância idêntica, o IDS chama o módulo Pellet (linha 28) para verificar na ontologia se a instância pode ser inferida como uma variação de ataque.

Se nenhum ataque for detectado ou inferido, o IDS continua chamando os módulos SPARQL e Pellet à medida que novas ações vão sendo encontradas em outros pacotes e adicionadas à instância de ataque criada pelo IDS. Isto ocorre até que um ataque seja detectado e alertado através do SPARQL (detecção) ou do Pellet (inferência), quando então a instância e a detecção são reiniciadas (linhas 26, 33 e 38).

Se o ataque for detectado através de inferência, o IDS verifica na ontologia se a classe do ataque inferido contém subclasses (linha 30). Se não forem encontradas subclasses o IDS alerta o ataque (linha 31), pois não existem classes de ataque mais específicas na ontologia.

Caso sejam encontradas subclasses, o IDS irá aguardar a detecção da próxima ação para verificar se um ataque mais específico pode ser inferido. Isto é feito armazenando a classe do ataque que foi originalmente inferido (linha 40) e verificando se a instância de ataque mudou de classe (linha 35) após a adição da próxima ação.

Se não houve mudança na classe da instância o IDS informa o ataque inferido originalmente como um ataque mais genérico (linha 36), caso contrário verifica-se se a nova classe contém subclasses e assim por diante. Sempre que uma instância é inferida como um novo ataque, o IDS persiste esta nova instância na base de conhecimento da ontologia (linhas 32 e 37) antes de reiniciar a detecção.

5 AVALIAÇÃO

A avaliação da proposta foi dividida em avaliação quantitativa (seção 5.1) e avaliação qualitativa (seção 5.2).

5.1 AVALIAÇÃO QUANTITATIVA

Para avaliar a eficiência da proposta, três cenários foram desenvolvidos. No primeiro foi aplicado SPARQL, no segundo Pellet (máquina de inferência) e no terceiro a base de assinaturas Snort (arquivo texto).

O objetivo dos experimentos foi comparar a escalabilidade e o desempenho da base de conhecimento da ontologia com a da base de dados baseada em assinaturas, dado que os experimentos foram feitos em um ambiente controlado e que a eficácia das detecções foi verificada. A avaliação de desempenho foi executada para mostrar que a abordagem proposta, mesmo utilizando inferências (computacionalmente custosas), é viável no mundo real.

Para avaliação foi utilizada uma base composta de até 128 ataques catalogados no Protégé. Esta base iniciou com 4 classes de ataques pré-cadastradas (*XMLInjection, XPathInjection, XQueryInjection* e *XSSInjection*) e 4 instâncias de ataques (*xpathInjection1, xpathInjection2, xqueryInjection1 e xssInjection1*). Adicionando incrementos de 4/8/16 instâncias de ataques por vez a base foi sendo aumentada até totalizar os 128 ataques. Para compor a base, diversas instâncias de ataques e ações foram simuladas com o intuito de imitar as variações de ataques que vão sendo incorporadas à base, em uma aplicação real da proposta. As categorias de ataques *ProtocolManipulation*, *SOAPParameterTampering* e *SQLInjectionThrougSOAP* foram modeladas somente após a análise dos testes de performance descritos nesta seção.

O primeiro experimento testou a ontologia consultando-a com apoio do SPARQL. Esta abordagem analisou os pacotes da rede procurando por ações maliciosas, que já estavam pré-cadastradas na base de conhecimento da ontologia, relacionando as mesmas com instâncias de ataques que foram previamente introduzidas utilizando o Protégé.

O segundo experimento utilizou o Pellet para avaliar a ontologia em tempo de execução. Neste experimento não havia instâncias de ataques na ontologia quando a mesma foi consultada pelo SPARQL, portanto a máquina de inferência tentou derivar novos ataques baseando-se em axiomas pré-definidos para cada classe de ataques. Em outras palavras, já que o módulo SPARQL falhou, o módulo de inferência foi invocado para determinar se os conjuntos de ações sendo capturadas poderiam ser considerados ataques.

Toda vez que uma nova sequência de ações foi inferida como sendo um ataque (nova assinatura), uma nova instância para este ataque foi automaticamente adicionada à base de conhecimento da ontologia. Assim, não se desperdiçará tempo invocando o módulo de inferência novamente caso este ataque específico seja capturado no futuro, pois o SPARQL irá detectá-lo primeiro, além de eliminar a possibilidade de ataques *zero-day* para estas novas instâncias.

O terceiro experimento não utilizou a ontologia como base de conhecimento; foi utilizado o arquivo de texto contendo regras para detecção de assinaturas de ataque do Snort, sem nenhuma técnica de otimização nas consultas.

O terceiro experimento foi executado duas vezes. Na primeira vez o arquivo de assinaturas foi consultado para procurar *payloads* (aleatoriamente inseridos do início ao final do arquivo), com o objetivo de comparação com o desempenho do SPARQL (figura 5.1). Na segunda execução o arquivo de assinaturas foi consultado procurando por *payloads* que não estavam no arquivo, o objetivo foi comparar seu desempenho com o do Pellet (figura 5.2). Assim como no primeiro e segundo experimentos, a quantidade de *payloads* sendo procurados pelo protótipo de IDS variou de 4 a 128.

O gráfico da figura 5.1 compara o experimento de detecção baseada em assinaturas com o experimento que utilizou o SPARQL, em um cenário onde os ataques estavam pré-cadastrados no arquivo de texto e na base de conhecimento da ontologia.

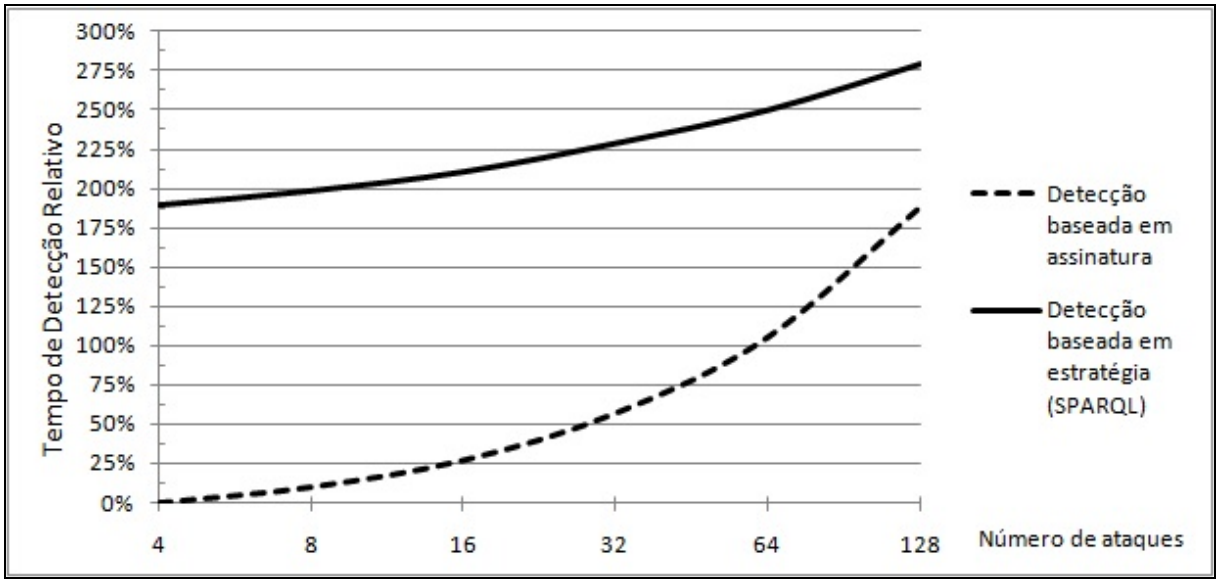


Figura 5.1 - Tempo de detecção relativo (Assinaturas x SPARQL)

O gráfico da figura 5.2 compara o experimento de detecção baseada em assinaturas com o experimento do Pellet, em um cenário onde as assinaturas sendo procuradas não estão no arquivo de texto e os conjuntos de ações sendo capturadas não correspondem a nenhuma instância de ataque na base de conhecimento da ontologia.

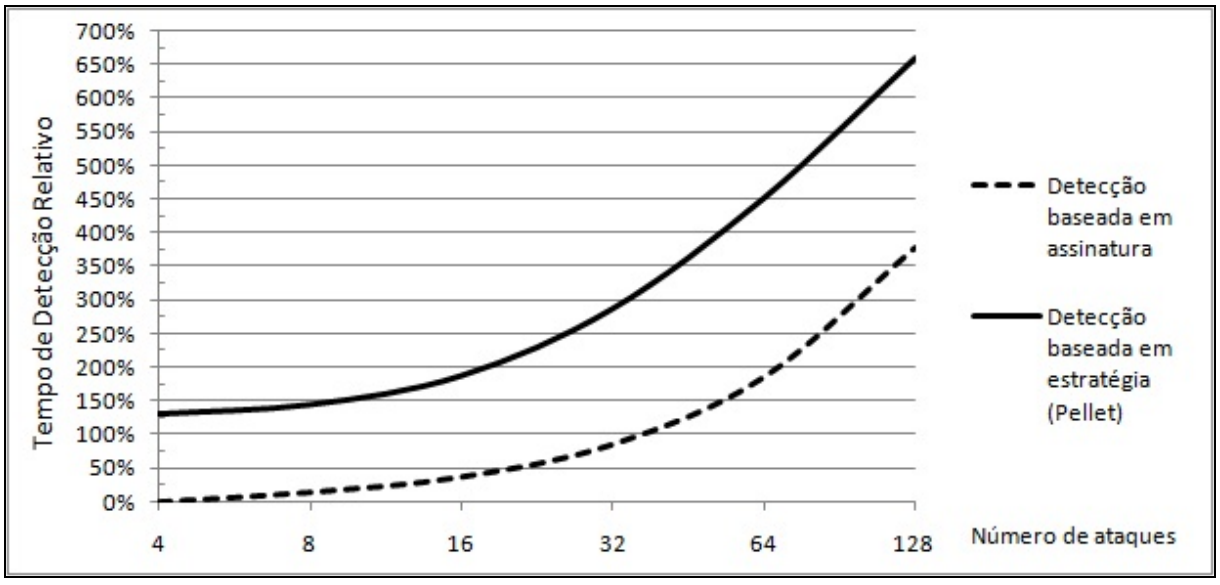


Figura 5.2 - Tempo de detecção relativo (Assinaturas x Pellet)

As figuras 5.1 e 5.2 mostram gráficos comparando o tempo de detecção relativo, tomando como referência o tempo para procurar 4 ataques através de detecção baseada em assinaturas. A motivação para tal escolha é que,

observando-se o ponto de partida da curva do Pellet na figura 5.2, nota-se que o mesmo gastou um tempo extra (se comparado com a abordagem baseada em assinaturas) necessário para derivar novas instâncias através dos axiomas das classes de ataques da ontologia. Foi observado que o tempo gasto para avaliar as classes de ataques sem sucesso utilizando Pellet e o tempo gasto para inferência e adição de uma nova instância abaixo de uma das classes é similar.

O gráfico da figura 5.2 mostrou o pior caso para detecções de ataques, pois as consultas resultam em perda de tempo de processamento porque o ataque não está na base, logo toda a base é consultada sem sucesso. Entretanto, mesmo o Pellet consumindo três vezes mais tempo do que a detecção baseada em assinaturas, sua abordagem ainda é vantajosa (em relação à detecção baseada em assinaturas) pelo fato de que esse módulo é executado uma única vez para cada nova variação de ataque – visando evitar ataques *zero-day* para as mesmas. Uma vez que o Pellet infere uma nova instância de ataque, este módulo não será mais executado no futuro se o mesmo ataque for capturado novamente, pois o SPARQL irá detectar o ataque primeiro da próxima vez. Já na detecção baseada em assinaturas este processamento sempre significará perda de tempo.

Observando a figura 5.1 constata-se uma tendência do desempenho do SPARQL ultrapassar a detecção baseada em assinaturas quando a base chegar a 512 ataques. Em aplicações reais as bases de ataques são muito amplas, logo a abordagem proposta teria a vantagem quantitativa de maior escalabilidade no tempo de detecção.

Baseando-se nos resultados relatados acima é possível concluir que a proposta deste trabalho, que mistura o primeiro e o segundo cenário, é vantajosa em relação ao terceiro cenário (abordagem clássica), obtendo a melhor relação custo benefício de detecção – baseada em assinaturas (SPARQL) *vs* baseada em conhecimento (Pellet).

5.2 AVALIAÇÃO QUALITATIVA

Em ontologias as definições de conceitos devem ser feitas através de axiomas lógicos (GRUBER, 1993, p. 909). Além disso, Gruber menciona que estas definições devem ser preferivelmente completas, ou seja, através de condições necessárias e suficientes. Isto porque se uma instância atende às condições necessárias e suficientes (definidas através de axiomas) de uma classe, ela obrigatoriamente será inferida como instância daquela classe.

Considerando as afirmações de Gruber, em um mundo perfeito não haveria a hipótese de alertas falsos serem gerados pelo protótipo de IDS, já que instâncias detectadas ou inferidas necessariamente atendem aos axiomas definidos para suas classes de ataque. Porém, Gruber (GRUBER, 1993, p. 916) também ressalta que se o resultado de uma inferência gerar um conhecimento que não corresponde à definição informal do domínio sendo representado, a ontologia pode estar incoerente. Ou seja, mesmo que os axiomas definidos na ontologia garantam que nada diferente do que foi definido será deduzido (inferido pelo IDS), sempre há a possibilidade de uma entrada incorreta na definição da ontologia por erro humano (assim como acontece em qualquer sistema automatizado – se a entrada está incorreta, o resultado será impreciso).

Em outras palavras, se o ataque está completamente descrito (contendo as condições necessárias e suficientes que refletem a estratégia do ataque no mundo real) a probabilidade de falso positivo é nula. Porém, se o conjunto de atributos (relacionamentos) e restrições não estiver precisamente descrito há possibilidade de ocorrência de falsos positivos.

A partir destas considerações, dois cenários foram criados para testar a taxa de falsos positivos da abordagem na detecção pelo Pellet (módulo de inferência que utiliza os axiomas para deduzir ataques no IDS), visando avaliar se a ontologia foi projetada de forma coerente.

Para o teste dos cenários foram criadas 128 instâncias (reais e simuladas) para testar os axiomas das sete classes de ataque da proposta (*XMLInjection, XPathInjection, XQueryInjection, XSSInjection, ProtocolManipulation*, *SOAPParameterTampering* e *SQLInjectionThrougSOAP*).

No primeiro cenário a lógica de detecção do IDS alertou ataques a cada inferência conclusiva, ou seja, assim que as restrições (axiomas) de qualquer classe

eram atendidas o ataque era alertado. Esta era uma das possibilidades consideradas inicialmente para a lógica de detecção da proposta.

No segundo cenário a lógica de detecção do IDS verificou se os ataques continham subclasses antes de alertá-los, verificando se um ataque mais específico poderia estar ocorrendo. Esta foi a abordagem de detecção escolhida para a proposta deste trabalho.

A avaliação dos cenários foi dividida em duas fases. Na primeira fase 64 dos 128 ataques criados foram utilizados, com o intuito de se fazer uma avaliação (treinamento) inicial da ontologia e ajustar as classes e axiomas caso necessário. Portanto, 50% da amostragem de instâncias já estava na ontologia ao término da primeira fase. Pequenos ajustes na programação do protótipo de IDS foram feitos após avaliar os resultados desta fase, já que o mesmo estava alertando erroneamente algumas ações de *TamperSOAPParameters* depois que ações das classes *Probing* e *Injection* geravam exceções na resposta do servidor. Nenhuma alteração foi feita na ontologia. Na segunda fase as 64 instâncias de ataque restantes foram simuladas para testar a porcentagem de acerto do IDS nas inferências feitas em tempo de execução, para ambos os cenários.

O resultado da avaliação foi que no primeiro cenário 7/64 instâncias de ataque não foram detectadas corretamente. Destas sete instâncias, quatro continham uma ação da classe *Discovery*, uma ação da classe *TamperSOAPParameters* e uma ação da classe *ProbeSQL*. Estas sequências de três ações deveriam alertar ataques do tipo *SQLInjectionThroughSOAP*, de acordo com o axioma definido para esta classe na seção 4.1.1. Porém, o IDS alertou para cada uma das quatro instâncias ataques de *SOAPParameterTampering* e de *XMLInjection*, respectivamente (totalizando 8 ataques alertados). Isto ocorreu porque a primeira parte destes ataques gerados (ação de *Discovery* e ação de *TamperSOAPParameters*) satisfaz o axioma da classe de ataque *SOAPParameterTampering*, e a parte restante (ação de *ProbeSQL*) satisfaz o axioma da classe genérica *XMLInjection*.

O mesmo erro de detecção foi identificado para as outras três instâncias de ataque que geraram alertas incorretos, porém envolvendo outras classes na ontologia. Cada uma destas três instâncias de ataque continha uma ação da classe *Discovery*, uma da classe *ProbeXPath* e uma da classe *InjectXQuery*. Desta vez as sequências de ações deveriam alertar ataques de *XQueryInjection*, também de

acordo com o axioma desta classe de ataque (seção 4.1.1). Porém, foram alertados para cada instância ataques de *XPathInjection* e *XMLInjection*, respectivamente (totalizando 6 ataques alertados). Da mesma forma, a classificação errônea ocorreu porque a primeira parte destes ataques gerados (ação de *Discovery* e ação de *ProbeXPath*) atende o axioma da classe de ataque *XPathInjection*, e a parte restante (ação de *InjectXQuery*) atende o axioma da classe genérica *XMLInjection*.

No primeiro cenário o resultado da dedução foi impreciso em alguns casos porque não foi considerado integralmente o conjunto de ações que atendem as restrições da classe mais específica. Isto é, a dedução considerou apenas um subconjunto de ações que satisfaziam as restrições (axiomas) de classes mais genéricas – primeiras a serem testadas na lógica de detecção.

No segundo cenário (abordagem escolhida), o protótipo foi programado para apenas informar um ataque mais genérico após verificar as classes mais específicas do ataque sendo inferido. Desta forma, todas as 64 instâncias de ataque foram inferidas com sucesso e adicionadas às classes corretas na ontologia. Como neste cenário o IDS aguardou a detecção da próxima ação antes de alertar um ataque, quando havia a possibilidade de um ataque mais específico estar ocorrendo, não houve imprecisões na detecção.

Possíveis imprecisões na abordagem do segundo cenário (ataque genérico inferido mesmo depois de verificadas as subclasses) não são consideradas como falsos positivos na abordagem proposta, porque falsos positivos são resultantes da classificação errônea de ações normais consideradas como ataques. Na abordagem proposta as imprecisões que são deduzidas em classes mais genéricas são alertadas como informativos, e não como ataques. Isto é, quando o nível de especificidade do ataque sendo deduzido não é suficiente para atingir um grau de precisão confiável (depois de verificadas suas subclasses na ontologia), o mesmo é alertado como uma mensagem informativa ao invés de ataque.

Um ataque inferido em uma classe genérica indica que nenhum dos conjuntos mais específicos de restrições (axiomas das subclasses) de ataques catalogados na ontologia foi satisfeito. Tal fato pode dar indícios de que o ataque inferido não está completo (alguma ação das estratégias das subclasses foi perdida na detecção), ou que uma nova categoria de ataque pode ter sido detectada. Esta informação pode então ser investigada por um administrador (especialista) para verificar se uma nova

subclasse teria que ser criada ou se a instância se encaixa em alguma subclasse existente.

Outra possibilidade de geração de falsos positivos seria a detecção de ações em pacotes de rede gerados por diferentes usuários, já que a proposta (em sua maioria) não verifica endereços IP na detecção das sequências de ações. Porém, se a proposta somente alertasse ataques cujas ações viessem de uma mesma origem, falsos negativos também poderiam ocorrer, já que um atacante poderia utilizar diferentes endereços IP para executar cada ação.

5.3 CONSIDERAÇÕES

Apesar de aplicar ontologia, a detecção utilizando SPARQL é similar à abordagem baseada em assinaturas, levando em conta que instâncias de ataques estão pré-cadastradas na base de conhecimento. Além disso, o Pellet trabalha inferindo na ontologia para derivar novas instâncias de ataque quando o SPARQL não encontra combinações exatas. A inferência neste caso mantém a taxa de falsos positivos na detecção similar à de abordagens baseadas em assinaturas, pois novos ataques só podem ser derivados de classes e axiomas pré-cadastrados na ontologia.

A falha encontrada no primeiro cenário da avaliação qualitativa, que gerou imprecisão na detecção do IDS, foi devida à adoção de uma estratégia de detecção que buscava apenas identificar as condições necessárias e suficientes na *engine* de inferência, nem sempre levando em conta os axiomas das subclasses mais específicas. No segundo cenário esta falha não ocorreu, já que os axiomas das subclasses eram sempre verificados e, caso não fossem atendidos, os ataques das classes mais genéricas poderiam ser alertados como informativos.

Outra vantagem da proposta é que ataques *zero-day* (derivados de estratégias de categorias de ataques) são eliminados, já que novas instâncias de ataque são automaticamente adicionadas à base de conhecimento da ontologia no momento em que são inferidas pela primeira vez.

A abordagem de IDS com detecção por anomalias também pode classificar novas ações como ataques, porém em tal abordagem a classificação é feita com base em um conjunto de ações isoladas umas das outras, que descrevem o modelo normal. Já na abordagem proposta, a inferência acontece a partir de classes e seus axiomas, que caracterizam a estratégia para cada categoria de ataque individualmente. Ou seja, a partir da catalogação de ataques em uma ontologia se pode ter “conhecimento” sobre o ataque e não apenas um *payload*. Isto diminui em muito a possibilidade de falsos positivos.

A inferência na ontologia pode ser utilizada tanto em tempo de execução (quando necessário) para aprender novos ataques, quanto na fase de modelagem para sugerir mudanças estruturais e encontrar inconsistências, otimizando a hierarquia de classes de ataque. Além disso, dependendo do tamanho da ontologia, redundâncias na sua definição que levariam horas para serem encontradas por um humano podem ser encontradas por uma máquina de inferência em alguns segundos.

6 CONCLUSÃO

Este trabalho apresentou uma abordagem baseada em ontologia para auxiliar IDSs na proteção de *web services* e para mitigar o problema dos ataques *zero-day* (para variações de ataque inferidas). As instâncias de ataque a *web services* que já estavam na base de conhecimento foram detectadas com sucesso utilizando SPARQL para consultar a ontologia. Adicionalmente, as variações nas estratégias dos ataques (sequências de ações) foram detectadas com sucesso através de inferência com uma baixa taxa de falsos positivos, já que novas instâncias de ataque são detectadas baseando-se somente em classes e axiomas de ataques pré-existentes. Estas novas sequências de ações foram automaticamente adicionadas à base de conhecimento da ontologia como instâncias abaixo das classes relacionadas, eliminando o problema dos ataques *zero-day* para as mesmas.

A proposta agrega as principais vantagens das abordagens clássicas de detecção, pois permite a detecção de ataques conhecidos (SPARQL), como na abordagem baseada em assinaturas, e permite a detecção de novos ataques (Pellet), como na abordagem baseada em anomalias. Além disso, a ontologia proposta pode ser considerada uma forma dedutível de representação de ataques, já que uma ação (*payload*) pode fazer parte de várias instâncias de ataque.

Em termos de desempenho a proposta é comparável à abordagem clássica de detecção baseada em assinaturas quando os ataques são conhecidos. Quando os ataques não são conhecidos a proposta deste trabalho perde em desempenho quando comparado à abordagem por assinaturas. Porém, ainda assim é vantajosa, já que neste caso podem-se detectar variações de ataques com uma baixa taxa de falsos positivos e mitigar ataques *zero-day*. Na abordagem de detecção por assinaturas esta consulta pela base inteira (procurando por um padrão de ataque não cadastrado) sempre significará desperdício de tempo.

Como trabalho futuro pretende-se estender a ontologia para que contemple também outras classes de ataques a *web services*, como por exemplo, ataques de negação de serviço (inclusive direcionados ao próprio IDS), e para que contemple outras ações (*payloads*) que possam ser detectadas pelo IDS. Desta forma, o

número de possibilidades de inferência aumentará proporcionalmente e a proposta de detecção de intrusão se tornará mais completa em termos de escopo.

REFERÊNCIAS

ANDREU, Andres; BANCIU, Cosmin. **OWASP WSFuzzer Project**. Disponível em: <http://www.owasp.org/index.php/Category:OWASP_WSFuzzer_Project>. Acesso em: Abril de 2011.

BACE, Rebecca; MELL, Peter. **Intrusion Detection Systems**. NIST Special Publication on Intrusion Detection Systems, 2001. 51 p.

BEBAWY, Ramy; SABRY, Hesham; EL-KASSAS, Sherif; HANNA, Youssef; YOUSSEF, Youssef. **Nedgty: Web Services Firewall**. In Proceedings of the IEEE International Conference on Web Services, ICWS'05, 2005.

BECHHOFER, Sean. **DL Implementation Group (DIG)**. Disponível em: <http://dl.kr.org/dig>. Acesso em: Abril de 2011.

BOAG, Scott; CHAMBERLIN, Don; FERNÁNDEZ, Mary F.; FLORESCU, Daniela; ROBIE, Jonathan; SIMÉON, Jérôme. **XQuery 1.0: An XML Query Language**. Disponível em: <http://www.w3.org/TR/xquery>. Acesso em: Abril de 2011.

BOOTH, David; HAAS, Hugo; MCCABE, Francis; NEWCOMER, Eric; CHAMPION, Michael; FERRIS, Chris; ORCHARD, David. **Web Services Architecture**. Disponível em: <http://www.w3.org/TR/ws-arch>. Acesso em: Abril de 2011.

BRAVENBOER, Martin; DOLSTRA, Eelco; VISSER, Eelco. **Preventing injection attacks with syntax embeddings**. In Science of Computer Programming archive, v. 75, p. 473-495, 2010.

CAPEC. **Common Attack Pattern Enumeration and Classification**. Disponível em: <http://capec.mitre.org/data/graphs/1000.html>. Acesso em: Abril de 2011.

CLARCK&PARSIA. **Pellet: OWL 2 Reasoner for Java**. Disponível em: <http://clarkparsia.com/pellet>. Acesso em: Abril de 2011.

COMBS, Gerald. **Wireshark - Go Deep**. Disponível em: <http://www.wireshark.org>. Acesso em: Abril de 2011.

CREMONINI, Marco; DAMIANI, Ernesto; VIMERCATI, Sabrina; SAMARATI, Pierangela. **An XML-based Approach to Combine Firewalls and Web Services Security Specifications**. ACM Workshop on XML Security, p. 69-78, 2003.

DAWES, Rogan. **OWASP WebScarab Project**.
Disponível em:
<http://www.owasp.org/index.php/Category:OWASP_WebScarab_Project>.
Acesso em: Abril de 2011.

DOU, Dejing; MCDERMOTT, Drew; Qi, Peishen. Ontology Translation on the Semantic Web. **In Journal on Data Semantics (JoDS) II**, p. 35-57, 2004.

GORELIK, Vlad. **One step ahead**. In ACM Queue, p. 24-31, 2007.

GRUBER, Thomas R. **Toward Principles for the Design of Ontologies Used for Knowledge Sharing**. In International Journal Human-Computer Studies 43, p. 907-928, 1993.

GUARINO, Nicola. **Formal Ontology and Information Systems**. Proceedings of FOIS'98, p. 3-15, 1998.

HANSEN, Robert. **XSS (Cross Site Scripting) Cheat Sheet**.
Disponível em:
< http://ha.ckers.org/xss.html>. Acesso em: Abril de 2011.

KONSTANTINOU, Nikolaos; SPANOS, Dimitrios-Emmanuel; MITROU, Nikolas. Ontology and Database Mapping: A Survey of Current Implementarions and Future Directions. **Journal of Web Engineering**, v. 7, p. 1-24, 2008.

MCAFEE FOUNDSTONE. **WSDigger**.
Disponível em:
<http://www.mcafee.com/br/downloads/free-tools/wsdigger.aspx>. Acesso em: Abril de 2011.

MCGUINNESS, Deborah L.; HARMELEN, Frank van. **OWL Web Ontology Language**.
Disponível em:
< http://www.w3.org/TR/owl-features>. Acesso em: Abril de 2011.

MELZER, Ingo; JECKLE, Mario. **A Signing Proxy for Web Services Security**. Berliner XML Tage 2003, p. 292–304, 2003.

MOORE, HD. **Metasploit - Penetration Testing Resources**.
Disponível em:
<http://www.metasploit.com>. Acesso em: Abril de 2011.

NOY, Natalya F.; MCGUINNESS, Deborah L. **Ontology Development 101: A Guide to Creating Your First Ontology**. Stanford Knowledge Systems Laboratory Technical Report KSL-01-05 and Stanford Medical Informatics Technical Report SMI-2001-0880, p. 1-25, 2001.

ORACLE. **For Java Developers**.
Disponível em:
<http://www.oracle.com/technetwork/java/index.html>. Acesso em: Abril de 2011.

OWASP. **The Open Web Application Security Project**.
Disponível em:
<http://www.owasp.org>. Acesso em: Abril de 2011.

PRUD'HOMMEAUX, Eric; SEABORNE, Andy. **SPARQL Query Language for RDF**.
Disponível em:
<http://www.w3.org/TR/rdf-sparql-query>. Acesso em: Abril de 2011.

RUSSELL, Stuart J.; NORVIG, Peter. **Artificial Intelligence: a Modern Approach**. New Jersey: Prentice Hall, 1995. 932 p.

SCARFONE, Karen; MELL, Peter. **Guide to Intrusion Detection and Prevention Systems (IDPS)**. Recommendations of the National Institute of Standards and Technology, National Institute of Standards and Technology, Gaithersburg, p. 1-127, 2007.

SIDDAVATAM, Irfan; GADGE, Jayant. **Comprehensive Test Mechanism to Detect Attack on Web Services**. 16th IEEE International Conference on Networks, p.1-6, 2008.

SNAPP, Steven R.; BRENTANO, James; DIAS, Gihan V.; GOAN, Terrance L.; HEBERLEIN, L. Todd; HO, Che-lin; LEVITT, Karl N.; MUKHERJEE, Biswanath; SMAHA, Stephen E.; GRANCE, Tim; TEAL, Daniel M.; MANSUR, Doug. **DIDS (Distributed Intrusion Detection System) - Motivation, Architecture, and An Early Prototype**. In Proceedings of the 14th National Computer Security Conference, p. 167-176, 1991.

SOURCEFIRE. **Sourcefire VRT Certified Rules - The Official Snort Ruleset**.
Disponível em:
< http://www.snort.org/snort-rules>. Acesso em: Abril de 2011.

SOURCEFORGE. **Network Packet Capture Facility for Java.**
Disponível em:
<http://sourceforge.net/projects/jpcap>. Acesso em: Abril de 2011.

SOURCEFORGE. **Jena – A Semantic Web Framework for Java**.
Disponível em:
<http://jena.sourceforge.net>. Acesso em: Abril de 2011.

STANFORD. **The Protégé Ontology Editor and Knowledge Acquisition System**.

Disponível em:
<http://protege.stanford.edu>. Acesso em: Abril de 2011.

UNDERCOFFER, Jeffrey; PINKSTON, John; JOSHI, Anupam; FININ, Timothy. **A Target-Centric ontology for intrusion detection.** Proceedings of the IJCAI Workshop on Ontologies and Distributed Systems, p. 47-58, 2004.

VOROBIEV, Artem; HAN, Jun. **Security Attack Ontology for Web Services**. Proceedings of the Second International Conference on Semantics, Knowledge, and Grid, paper 42 (6 pages), 2006.

YEE, Chan G.; SHIN, Wong H.; RAO, G.S.V.R.K.. **An Adaptive Intrusion Detection and Prevention (ID/IP) Framework for Web Services**. IEEE 2007 International Conference on Convergence Information Technology, p. 528-534, 2007.

ZERO DAY INITIATIVE. **Zero Day Initiative**.
Disponível em:
<http://www.zerodayinitiative.com/advisories/upcoming>. Acesso em: Abril de 2011.

ZHENG, Jun; HU, Ming-zeng. **Intrusion Detection of DoS/DDoS and Probing Attacks for Web Services**. Lecture Notes in Computer Science, Springer Berlin / Heidelberg, v. 3739/2005, p. 333-344, 2005.