UFPR – UNIVERSIDADE FEDERAL DO PARANÁ

FERNÃO FABRÍCIO IKOMA
RAFAEL TAIZO YAMASHITA
MARCOS VENÍCIO SCHEFFLER

CORRELACIONAMENTO DISTRIBUÍDO PARA ALERTAS GERADOS POR SISTEMAS DE DETECÇÃO DE INTRUSÕES

CURITIBA
OUTUBRO 2006

FERNÃO FABRÍCIO IKOMA
RAFAEL TAIZO YAMASHITA
MARCOS VENÍCIO SCHEFFLER

CORRELACIONAMENTO DISTRIBUÍDO PARA ALERTAS GERADOS POR SISTEMAS DE DETECÇÃO DE INTRUSÕES

Trabalho apresentado à disciplina CI083 Trabalho de Graduação em Organização e Arquitetura de Computadores II da Universidade Federal do Paraná – UFPR.

Orientador: Prof. Dr. Roberto A. Hexsel

CURITIBA
OUTUBRO 2006

AGRADECIMENTOS

Agradecemos aos nossos familiares por toda a estrutura e apoio incondicional, sem eles não seria possível. As nossas namoradas pelo apoio, carinho e compreensão.

Agradecemos também ao nosso orientador, professor Dr. Roberto A. Hexsel e a Thiago Mello, pela orientação, paciência, e por nos ensinar não só sobre o que a matéria exigia, mas também pelos valores que nos acompanharão para sempre.

A todos os colegas e professores que ajudaram direta ou indiretamente.

SUMÁRIO

RESUMO

Com o aumento da interconexão entre redes de computadores, novos tipos de ataques e suas quantidades têm aumentado. Sistemas de detecção de intrusão são ferramentas essenciais para fornecer segurança a redes de computadores. Esses sistemas geram alertas para o administrador através da análise de trafego da rede, quanto maior o trafego na rede maior é a quantidade de alertas gerados. Torna-se portanto necessário um sistema que sintetize o número de alertas, e ao mesmo tempo aumente o conteúdo semântico daqueles. Este trabalho descreve alguns de testes de desempenho e modificações no código do Nariz, um sistema que efetua o correlacionamento de alertas, gerados por sistemas de detecção de intrusão, de forma distribuída, possibilitando assim o emprego de sensores que executam em máquinas de menor custo e/ou capacidade, reduzindo o custo da detecção de ataques. O mecanismo de correlacionamento distribuído do Nariz é baseado na distribuição de tráfego para vários pares sensor-correlacionador, e na comunicação entre aqueles para produzir poucos alertas com elevado conteúdo semântico, para serem analisados pelos administradores da rede. Os experimentos realizados mostraram que o Nariz pode reduzir o número de alertas gerados para o administrador, bem como pode ser usado em uma rede de trafego intenso.

Palavras-chave: Detecção de Intrusão Distribuída, Correlacionamento de Alertas em Redes de Alta Velocidade, Correlacionamento Paralelo de Alertas.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

CA	– Correlacionador de alertas.
CERT	– Centro de Estudos, Resposta e Tratamento de Incidentes de Segurança no Brasil.
CSN	– Comunicador Sensor/Nariz.
NdS	– Negação de Serviços (“Denial of Service”).
PoP	– Ponto de Presença (“Point of Presence”).
RA	– Receptor de alertas.
RAL	– Receptor de alertas locais.
RAR	– Receptor de alertas remotos.
RNP	– Rede Nacional de Ensino e Pesquisa.
SCD	– Sistema de Correlacionamento Distribuído.
HTTP	– Hypertext Transfer Protocol.
ICMP	– Internet Control Message Protocol.
IP	– Internet Protocol.
OSSIM	– Open Source Security Information Management.
PCPS	– Primeiro a Chegar, Primeiro a Sair.
SDI	– Sistema de Detecção de Intrusão (“Intrusion Detection System”).
SEC	– Simple Event Correlator.
SO	– Sistema Operacional.
TCP	– Transmission Control Protocol.
URI	– Uniform Resource Identifier.

INTRODUÇÃO

A busca pela segurança é muito freqüente por pessoas e organizações. A segurança está relacionada com a proteção existente ou necessária sobre tudo (pessoas, objetos, dados ou informações) que possua valor para alguém ou para uma organização. Tais valores possuem aspectos básicos como confidencialidade, integridade e disponibilidade que nos ajudam a entender as necessidades de sua proteção.

A busca pela segurança é histórica e está ligada ao crescimento do número de grandes redes, com constante troca de dados e informações, fazendo com que pessoas e organizações busquem formas de se proteger de qualquer tipo de ataque. Em se tratando de redes de computadores, uma das formas de proteção são os Sistemas de Detecção de Intrusão, ou SDI's (*"Intrusion Detection System"*), que são meios técnicos de se descobrir quando uma rede está sofrendo acessos não autorizados que podem indicar a ação de *hackers*, funcionários mal intencionados ou qualquer outro tipo de risco de intrusão.

Com a constante evolução tecnológica, que tem como um dos resultados o aumento quase que continuo das velocidades das redes, alguns especialistas tem questionado a funcionalidade dos SDI's. Entre as questões discutidas está a geração de um grande volume de evidencias de tentativas de ataques, devido a rede ser de alta velocidade, dificultando muito o trabalho de análise por um humano.

O objetivo deste trabalho é estudar e aprimorar um sistema de correlacionamento distribuído de alertas. O correlacionamento consiste na agregação de dados contidos em vários alertas e visa uma redução no número de avisos que chegam para o administrador, filtrando aqueles que não representam ameaças.

1 REVISÃO DA BIBLIOGRAFIA

Amoroso [01] define detecção de intrusão como o processo de identificar e responder a atividades maliciosas dirigidas a computadores e recursos de rede. Proctor [09], por sua vez, define que detecção de intrusão é a tarefa de coletar informações de uma variedade de fontes – sistemas ou redes – e então analisá-las buscando sinais de intrusão e de mau-uso. É importante considerar alguns detalhes em ambas as definições. No primeiro caso, o termo “detecção de intrusão” é usado tanto no sentido de detecção propriamente dito como na reação a essa atividade. Isso amplia a funcionalidade dos SDI’s, impondo a eles a difícil tarefa de reagir aos ataques detectados. Proctor, por sua vez, cria uma distinção entre os termos intrusão e mau-uso, sendo o primeiro compreendido por ataques originados externamente, e o segundo, por ataques originados de dentro da organização.

Detecção de ataques seria o termo mais correto para ser usado nesse contexto, que é o de tentar identificar ações maliciosas que levem o sistema a um resultado não autorizado. Essas ações podem ser caracterizadas como variando desde uma destruição de informações até uma varredura de portas.

As varreduras de portas consistem em dados enviados por um *hacker* através de uma conexão para localizar um computador ou uma rede e descobrir se há portas abertas que aceitem a conexão, explorando vulnerabilidades existentes no sistema.

Para evitar uma confusão maior com a criação de um novo termo, “detecção de intrusão” será utilizado no restante do texto, englobando incidentes já concretizados, tentativas de ataques, obtenção de informações, ameaças internas e externas. Não será considerado nessa definição, no entanto, a fase de reação, porque consideramos que os SDI’s não têm necessariamente essa funcionalidade.

Meios para detectar e reagir a possíveis incidentes são os passos necessários na busca por uma segurança mais efetiva. Com o crescimento da interconexão de computadores em todo o mundo, materializado pela Internet, verifica-se o conseqüente aumento nos tipos e no número de ataques a sistemas, gerando um nível de complexidade muito elevado para a capacidade dos tradicionais mecanismos de prevenção. A FIGURA 1 mostra que o número de ataques reportados ao Centro de Estudos, Respostas e Tratamento de Incidentes de

Segurança no Brasil (CERT), que é o grupo responsável por receber, analisar e responder a incidentes de segurança em computadores envolvendo redes conectadas à Internet brasileira. O CERT recebe notificações de quaisquer atividades que sejam julgadas como um incidente de segurança pelas partes envolvidas. Estes incidentes podem ser varreduras (*scans*), tentativas de conseguir acesso não autorizado a sistemas ou dados, modificações em um sistema, sem o conhecimento, instruções ou consentimento prévio do proprietário do sistema, ataques de engenharia social, entre outros. O número de incidentes deste tipo este ano (2006) dobrou em relação a 2005.

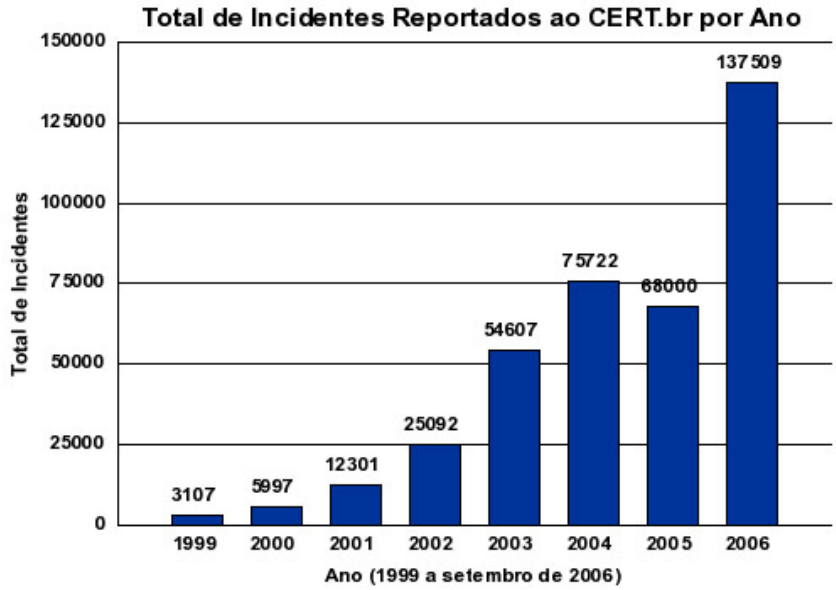

FIGURA 1 - Total de Incidentes reportados ao CERT desde 1999.
FONTE: CERT.br

Campello [02] afirma que, devido à diversidade de mecanismos de ataque, é praticamente impossível a concepção de um sistema de segurança baseado somente em travas, sem a existência de nenhum tipo de alarme. Da mesma forma, para a maioria das aplicações atuais, desde redes corporativas simples até sistemas de *e-commerce* ou aplicações bancárias, é praticamente inviável a simples utilização de mecanismos que diminuam a probabilidade de eventuais ataques. Um ataque força, em casos extremos, a interrupções totais dos serviços para que um lento e oneroso processo de auditoria e de posterior restauração manual seja efetuado.

Uma caracterização quantitativa das atividades de intrusão efetuadas na Internet, efetuada com base na análise de *logs* de *firewalls*, foi realizada por Yegneswaran em [10]. O trabalho envolveu a coleta, durante um período de 4

meses, de mais de 1.600 *logs* de *firewalls* e sistemas de detecção de intrusão distribuídos pelo mundo. Os resultados permitiram caracterizar diversos tipos de varreduras e sua relação com a disseminação de vírus e *worms*. Pesa sobre o trabalho o fato de ter sido realizado sem o apoio de ferramentas (o que compromete a sua utilização contínua). Além disso, a abordagem é exclusivamente quantitativa, o que dificulta o entendimento de algumas situações em que os eventos precisam ser analisados mais de perto para se poder confirmar uma atividade suspeita. As análises sobre os resultados obtidos apresentam mais algumas implicações. Com a popularização da internet, as atividades de intrusão atingiram uma escala gigantesca e os administradores de rede devem estar preparados para lidar com a quantidade de alertas gerados. Outra implicação diz respeito à diversidade dos tipos de ataques, foi constatado que uma pequena coleção de *logs* pode não ser suficiente para identificar os piores ataques e algumas vezes até mesmo os ataques mais comuns.

Em relação à análise de eventos, técnicas da Inteligência Artificial têm sido aplicadas para relacionar eventos gerados por sistemas de detecção de intrusão (e não por *firewalls*) [03], [07] e [08]. Ning em [07] apresenta um método que correlaciona pré-requisitos e conseqüências de alertas gerados por sistemas de detecção de intrusão a fim de determinar os vários estágios de um ataque.

Os autores Debar [03], Ning [07] e Porras [08] sustentam o argumento de que um ataque geralmente tem diferentes estágios e não acontece isoladamente, e que cada estágio do ataque é pré-requisito para o próximo. Por exemplo, uma varredura de portas pode identificar os computadores que possuem serviços vulneráveis. Com base nisso, o atacante pode explorar essas máquinas para executar código arbitrário com privilégios do sistema local ou causar uma negação de serviço.

Esta pode ser caracterizada como uma atividade maliciosa que busca paralisar um serviço de um servidor, fazendo com que a vítima simplesmente pare de oferecer o seu serviço aos clientes legítimos, enquanto tenta lidar com o tráfego gerado pelo ataque [06]. Um serviço pode ser o uso de um buscador de páginas, a compra de um determinado produto ou simplesmente a troca de mensagens entre duas entidades. O resultado de um ataque de negação de serviço pode ser o congelamento ou a reinicialização do programa da vítima que presta o serviço, ou ainda o esgotamento completo de recursos necessários para prover o serviço.

As abordagens descritas por Debar e Porras em [03] [08] propõem a análise de alertas gerados por dispositivos de segurança dispersos geograficamente. Ambas propõem algoritmos para agregação e correlação de alertas. Debar [03] define um modelo de dados unificado para representar alertas associados à detecção de intrusão e um conjunto de regras para processá-los. O algoritmo de detecção é capaz de identificar (i) alertas reportados por diferentes dispositivos, mas que estão relacionados com o mesmo ataque (duplicatas), e (ii) alertas que estão relacionados, e, portanto, são gerados juntos (conseqüências).

A abordagem de Porras [08] utiliza estratégias como análise de topologia, priorização e agregação de alertas que possuem atributos comuns. As duas abordagens mencionadas não lidam bem com a detecção de cenários que se diferenciam (mesmo que sutilmente) daqueles que foram descritos através de regras de fusão e agregação.

O trabalho apresentado por Mello [05] serviu como base para este trabalho. Aquele descreve um sistema de correlacionamento distribuído de alertas, chamado Nariz. O Nariz baseia-se em duas fases de correlacionamento, com pré-processamento local e pós-processamento distribuído. O sistema Nariz visa correlacionar alertas de forma distribuída em uma rede de alta velocidade, através de subsistemas de correlacionamento que podem ser executados em computadores com custo menor do que em sistemas centralizados. O correlacionamento distribuído utiliza troca de mensagens entre seus correlacionadores, que estão espalhados pela rede. Um alerta é encaminhado ao administrador da rede quando o sistema apresenta vários indícios de uma tentativa de ataque. A configuração do Nariz é bastante flexível, para permitir ajustes finos na detecção de atividades suspeitas. A cada classe de alarme são associados a parâmetros de configuração que determinam a freqüência com que alarmes serão enviados ao administrador da rede. Estes parâmetros também determinam o grau de cooperação entre os correlacionadores.

2 DESCRIÇÃO DO NARIZ

Com o aumento da velocidade das redes ao longo dos anos e o conseqüente aumento do tráfego através das mesmas, o número de tentativas de ataques também tem crescido. Os Sistemas de Detecção de Intrusão, que são sistemas eficientes de monitoramento de trafego em redes, podem tornar-se ineficientes em redes de alta velocidade, devido à enorme quantidade de alertas gerados, alguns provindos de falsos positivos. O objetivo do correlacionamento de alertas é gerar informações que sejam relevantes, de uma forma direta e resumida, para que o administrador possa analisar as informações contidas em cada evento.

Sensores, ou sensores SDIRD (Sistema de Detecção de Intrusão para Redes Distribuído), são Sistemas Detectores de Intrusão distribuídos que analisam o tráfego da rede destinado especificamente a um sensor, ao contrario de um SDIRC (SDIR Centralizado) que examina o tráfego em toda rede. O tráfego é analisado através de uma interface de rede à procura de qualquer tipo de evidência de tentativa de ataque. Se for encontrada alguma evidência, o sistema de detecção de intrusão cria um registro, em arquivo log, na forma de um alerta. Sensores distribuídos devem ser utilizados em redes de alta velocidade, pois um SDIR centralizado não é capaz de analisar o trafego em toda rede, tornando então necessária a divisão do trafego em porções sendo cada porção analisada por um sensor. O sensor utilizado em nosso trabalho é o Snort [15].

O Snort é um Sistema de Detecção de Intrusão para Redes muito utilizado. Tem seu código fonte aberto, é simples e eficiente e permite a inclusão de regras de detecção. Essas regras servem para fazer a filtragem e a análise do tráfego na rede. O Snort pode emitir, para o correlacionador, vários alertas de tentativas de ataques em tempo real e em vários formatos.

A FIGURA 2 mostra a organização de um Sistema de Correlacionamento Distribuído (SCD) que funciona dividindo o trafego da rede em porções e enviando essas porções para vários sensores. Cada sensor analisa o tráfego de acordo com as regras pré-definidas e se uma tentativa de ataque for identificada, é gerado um alerta. Em seguida esse alerta é enviado para o correlacionador que está ligado ao sensor. O correlacionador compara o alerta que acabou de receber do sensor com os alertas já existentes em sua base de dados. Se este for um novo alerta, ele é

inserido na base de dados local, senão ele é correlacionado com os alertas já existentes. Essa é a fase de correlacionamento local.

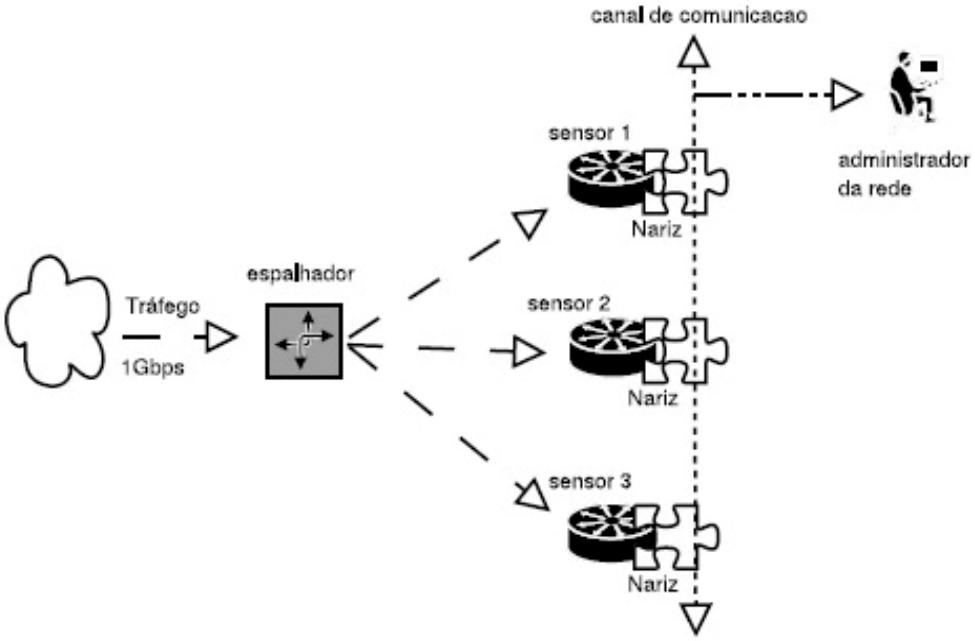

FIGURA 2 - Arquitetura de sensores com o sistema Nariz
FONTE: MELLO, 2004 [05].

Quando o número de alertas recebido pelo correlacionador atinge um valor especificado para cada classe de ataque identificado, então o correlacionador local envia uma mensagem para todos os correlacionadores remotos na rede. Essa troca de mensagens obedece a um protocolo sobre soquetes e serve para avisar os correlacionadores remotos sobre uma possível tentativa de ataque à rede. Essa é a fase de correlacionamento distribuído. O correlacionador distribuído em questão é o Nariz, que foi estudado e melhorado em nosso trabalho. A FIGURA 3 mostra com um pouco mais de detalhe o correlacionamento em sua fase local e em sua fase distribuída.

A Base de dados utilizada para armazenar os alertas é o SQLite [16], que é *open-source*. A biblioteca SQLite disponibiliza um conjunto de classes para gerenciamento e acesso aos dados na base, e é executada no mesmo espaço de endereçamento da aplicação, dispensando a comunicação entre processos ou comunicação através da rede. A base de alertas é mantida na memória principal para aumentar a velocidade do correlacionamento de alertas.

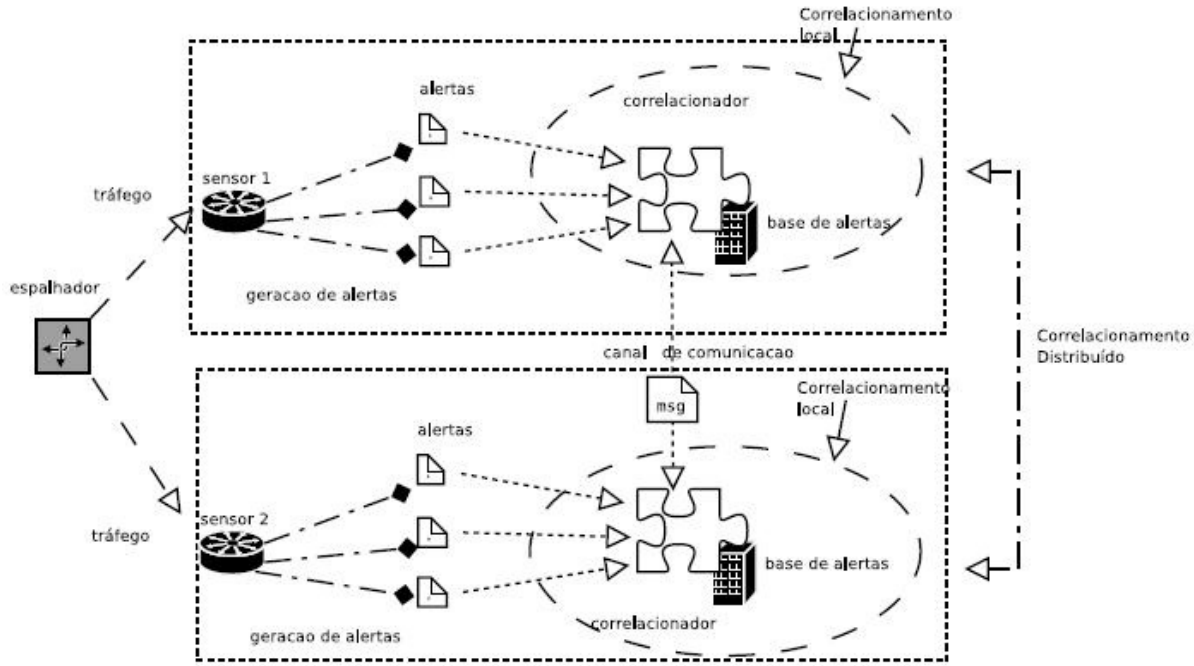

FIGURA 3 - Esquema dos sensores e correlacionadores
FONTE: MELLO, 2004 [05].

2.1 VARIÁVEIS DE CORRELACIOMANETO

O correlacionamento dos alertas é realizado através da aplicação de regras de correlacionamento sobre variáveis de correlacionamento que são extraídas na análise de trafego efetuada pelos sensores. As variáveis utilizadas pelo Nariz são provindas dos alertas emitido pelo Snort com a opção de alerta resumido (opção "-A fast") e tem os seguintes campos: (1) endereço IP origem, (2) endereço IP destino, (3) porta destino, (4) classe do ataque, (5) hora do ataque, (6) data. Abaixo é mostrado um exemplo de alerta resumido emitido pelo Snort:

```
12/29-17:28:40.729658  [**] [119:18:1] (http_inspect) WEBROOT DIRECTORY
TRAVERSAL [**] {TCP} 198.2.147.212:57437 -> 198.234.72.150:80
```

O alerta acima mostra:

(1) endereço IP origem: *198.2.147.212*

(2) endereço IP destino: *198.234.72.150*

(3) porta destino: *80*

(4) classe do ataque: *(http_inspect)*

(5) hora do ataque: *17:28:40*

(6) data: *12/29*

Existem mais três variáveis além dessas. Duas são obtidas através dos endereços da rede destino e origem, removendo, respectivamente, o ultimo octeto dos endereços IP destino e origem. A terceira variável é o limiar, uma variável numérica, do tipo inteiro, não negativo, iniciada com valor zero e que é incrementada a cada novo alerta que satisfaça a uma das regras de correlacionamento. Essa variável demonstra o grau de coincidência dos alertas e determina ações a serem executadas pelo Nariz, como o correlacionamento distribuído e o envio de mensagens para o administrador da rede.

2.2 REGRAS DE CORRELACIONAMENTO

As regras de correlacionamento consistem em operações binárias que comparam alertas com características semelhantes e eliminam informações redundantes diminuindo a quantidade de informação a ser analisada pelo administrador da rede. Existem dois tipos de regras: as regras de endereços e as regras de classe de ataque.

As regras de endereço (REs) contem cinco cláusulas, REipD, REipO, REpD, RErD e RErO, baseadas nas variáveis *ipDestino*, *ipOrigem*, *portaDestino*, *redeDestino* e *redeOrigem*, respectivamente. As RE's são avaliadas como verdadeiras se o conteúdo de uma das variáveis de um alerta previamente armazenado for igual ao conteúdo da mesma variável em um alerta novo. Note que tanto o endereço da máquina quanto o endereço da rede são necessários para que se possa distinguir “ataques contra máquinas distintas na mesma rede” de “ataques originados de máquinas distintas na mesma rede”.

A regra de classe de ataque (RCA) depende da variável *classe do ataque*, e é avaliada como verdadeira quando o conteúdo da variável classe de um alerta previamente armazenado é igual ao conteúdo desta mesma variável num novo alerta. A cada avaliação verdadeira de uma das RE's ou da RCA, o valor da variável *limiar* é incrementado. Caso um alerta satisfaça a todas as cláusulas, o limiar desse alerta será incrementado seis vezes: 5×RE's + RCA.

Os resultados da avaliação das regras RE e RCA são agregados na regra REC, que avalia como verdadeira quando um alerta de certa classe já existe na

base e algum dos endereços fonte ou destino do novo alerta é o mesmo que no alerta pré-existente. A regra REC é utilizada na união de alertas e é formalizada na Equação 1.

$$REC = \left(\left(REipD \vee REipO \vee REpD \vee RErD \vee RErO \right) \wedge RCA \right) \quad \textbf{(1)}$$

Para eliminar alertas duplicados, o sistema de correlacionamento avalia a chamada Regra de Duplicata (RD). A avaliação da regra RD é verdadeira, se todas as avaliações das RE's e RCA forem verdadeiras, como mostra a Equação 2. As regras RD e REC são utilizadas nos correlacionamentos local e distribuído.

$$RD = \left(REipD \wedge REipO \wedge REpD \wedge RErD \wedge RErO \wedge RCA \right) \quad \textbf{(2)}$$

2.3 CORRELACIONAMENTO LOCAL E DISTRIBUÍDO

No correlacionamento local as regras RD e REC são analisadas sempre que um novo alerta é enviado pelo sensor. Caso a análise das regras resulte em falso, o novo alerta é inserido na base local de alertas, pois é um novo alerta. Se a análise da regra RD for verdadeira, então a variável limiar do alerta na base de dados que satisfez a regra RD é incrementada e o novo alerta é descartado por já existir. Se a avaliação da regra REC for verdadeira, então a variável limiar do alerta é incrementada, mas o alerta não é descartado e sim unido ao alerta que já existe na base, concatenando-se os valores das variáveis que diferem. A data e a hora são atualizadas para aquelas do alerta mais recente.

No correlacionamento distribuído, a informação sobre possíveis ataques é distribuída a todos os correlacionadores vizinhos e algumas vezes são enviadas mensagens de alerta para o administrador da rede. O modelo de correlacionamento distribuído tem atribuição de pesos distintos para alertas tratados localmente e para mensagens recebidas de outros correlacionadores. Além dessa distinção de pesos, dois valores de gatilho são usados para indicar quando um correlacionador alertará os correlacionadores vizinhos e quando enviará uma mensagem para o administrador da rede.

2.4 LIMIARES E GATILHOS

Quando um alerta novo é inserido na base de dados de um correlacionador, seu limiar tem valor zero. Quando novos alertas vão sendo correlacionados, o limiar vai sendo incrementado. Esse incremento pode ser maior se o alerta for recebido de um correlacionador vizinho ao invés do sensor local. O limiar pode atingir dois valores de gatilho: *conversa* ou *pânico*. Esses valores são definidos pelo usuário no arquivo *"limiar.conf"* para cada classe de ataque.

A FIGURA 4 mostra um diagrama de tempo com a variação nos valores do limiar à medida que novos alertas são correlacionados. Quando um limiar atinge o valor de *conversa* definido (ponto A), o correlacionador local envia uma mensagem para todos os correlacionadores vizinhos. Quando o limiar atinge o valor de *pânico* (ponto C) uma mensagem é enviada para o administrador. Então o valor do limiar é decrementado, de acordo com a equação "*limiar(t+1) = limiar(t) / 3"*, fazendo com que este fique abaixo do gatilho *conversa*, evitando o envio de mensagens repetidas ao operador. Essa equação foi escolhida com base nos experimentos realizados durante o desenvolvimento do Nariz.

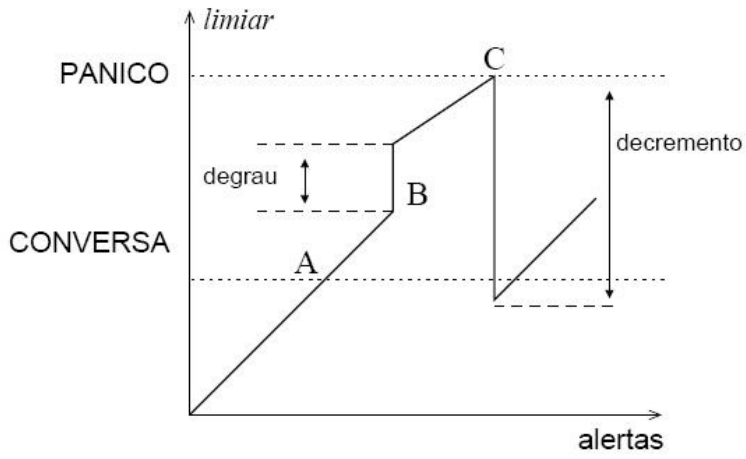

FIGURA 4 - Gráfico do limiar x alertas
FONTE: MELLO, 2004 [05].

O ponto B da FIGURA 4 representa o "salto" no valor do limiar, que é o acréscimo causado pelo recebimento do mesmo alerta de algum correlacionador vizinho. Esse salto é proporcional ao valor do degrau para o correlacionamento distribuído.

3 TRABALHOS REALIZADOS

3.1 COLETA DE ALERTAS

Para que o desempenho do correlacionador pudesse ser testado em um ambiente próximo ao real, um trabalho de coleta de alertas foi realizado com o apoio do Ponto de Presença da Rede Nacional de Ensino e Pesquisa no Estado do Paraná (PoP-PR). O PoP-PR foi escolhido por ser justamente o objeto principal para o qual o correlacionamento distribuído foi proposto, pois é uma rede de tráfego intenso de dados. A FIGURA 5 mostra o tráfego de entrada e saída de dados no enlace PoP - RNP em um período de 24h e a FIGURA 6 mostra o número de pacotes transmitidos pelo enlace no mesmo período de tempo.

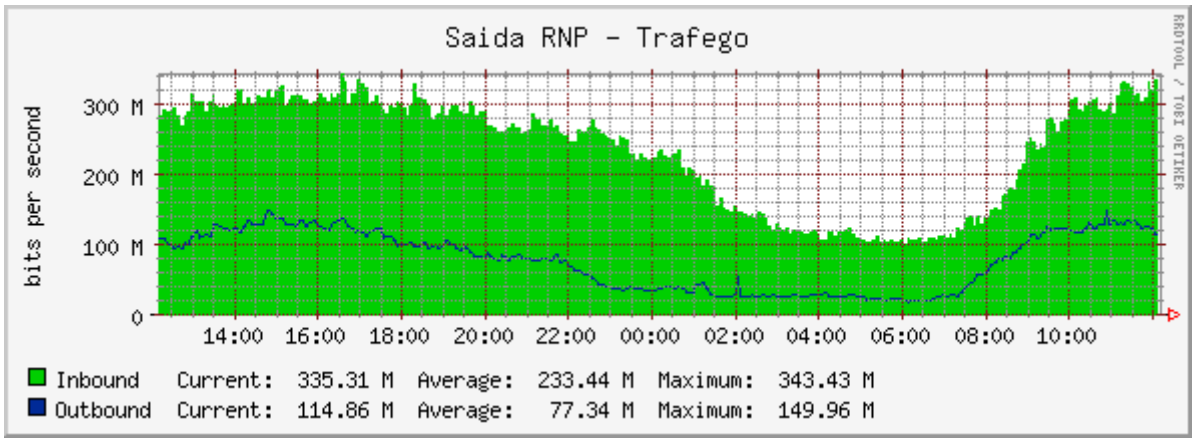

FIGURA 5 - Tráfego diário no enlace de saída da RNP
FONTE: PoP-PR

Para monitorar o tráfego de entrada e saída, um computador utilizando Linux foi configurado com o SDI Snort versão 2.4.2 e conectado à rede. A biblioteca de regras de detecção usada pelo Snort para a coleta de alertas foi a versão 2.4 encontrada em www.snort.org e a configuração completa é mostrada no ANEXO I - ARQUIVO DE CONFIGURAÇÃO DO SNORT.

A coleta, realizada em um período de 36 dias, gerou um total de 38.808.991 alertas, o que significa uma média de aproximadamente 1.080.000 alertas diários e 45.000 alertas por hora. Esse número comprova a afirmação que “um administrador de rede humano não consegue acompanhar uma grande quantidade de eventos, devido ao processo manual que o humano teria que realizar e às limitações físicas” [05].

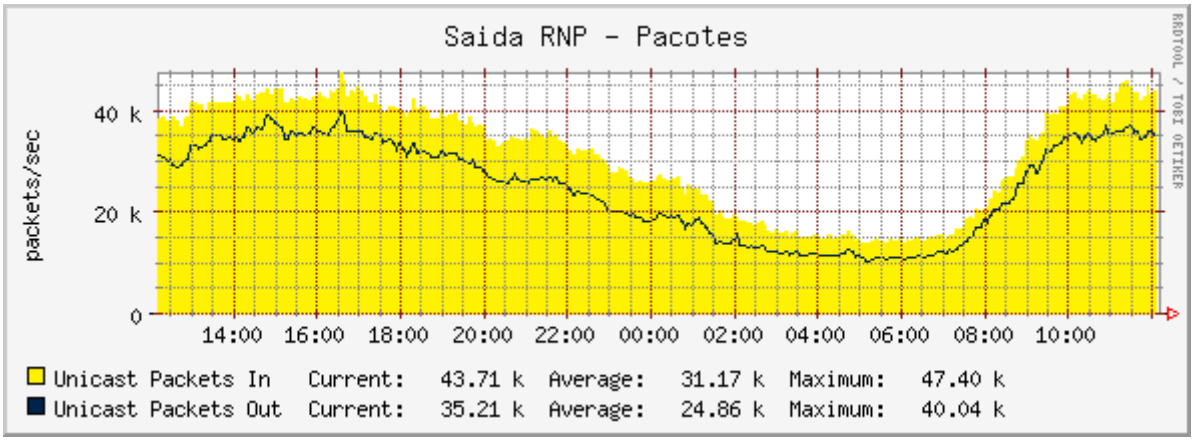

FIGURA 6 - Número de pacotes transmitidos pelo enlace de saída da RNP
FONTE: PoP-PR

Por motivos de sigilo e segurança, os endereços IP de origem e destino contidos nos alertas coletados foram substituídos por endereços gerados aleatoriamente através de um *script* desenvolvido utilizando o gerador de programas de análise léxica LEX. O *script* é mostrado no APÊNDICE I.

3.2 ALTERAÇÕES DO CÓDIGO FONTE

O ponto de partida adotado para o início dos trabalhos, é a versão original do *Nariz* publicada por Thiago Mello em 14 de Novembro de 2004, disponível em http://nariz.sourceforge.net. Algumas alterações no código fonte foram realizadas para melhorar o desempenho e a eficiência do correlacionador, corrigindo erros e alterando profundamente alguns algoritmos.

3.2.1 A Arquitetura Original

O sistema de correlacionamento Nariz original é dividido em dois processos. O primeiro é responsável por obter os alertas emitidos pelo sensor SDIR e enviá-los ao segundo, que realiza o correlacionamento dos novos alertas. Para melhorar o desempenho, cada uma dessas partes é executada de forma concorrente através de linhas de execução (*"threads"*). A FIGURA 7 representa a arquitetura original do sistema de correlacionamento Nariz e os relacionamentos entre as linhas de execução, que está dividida em 4 partes: Comunicador Sensor/Nariz,

Correlacionador de alertas, Receptor de alertas locais e Receptor de alertas remotos, descritas abaixo.

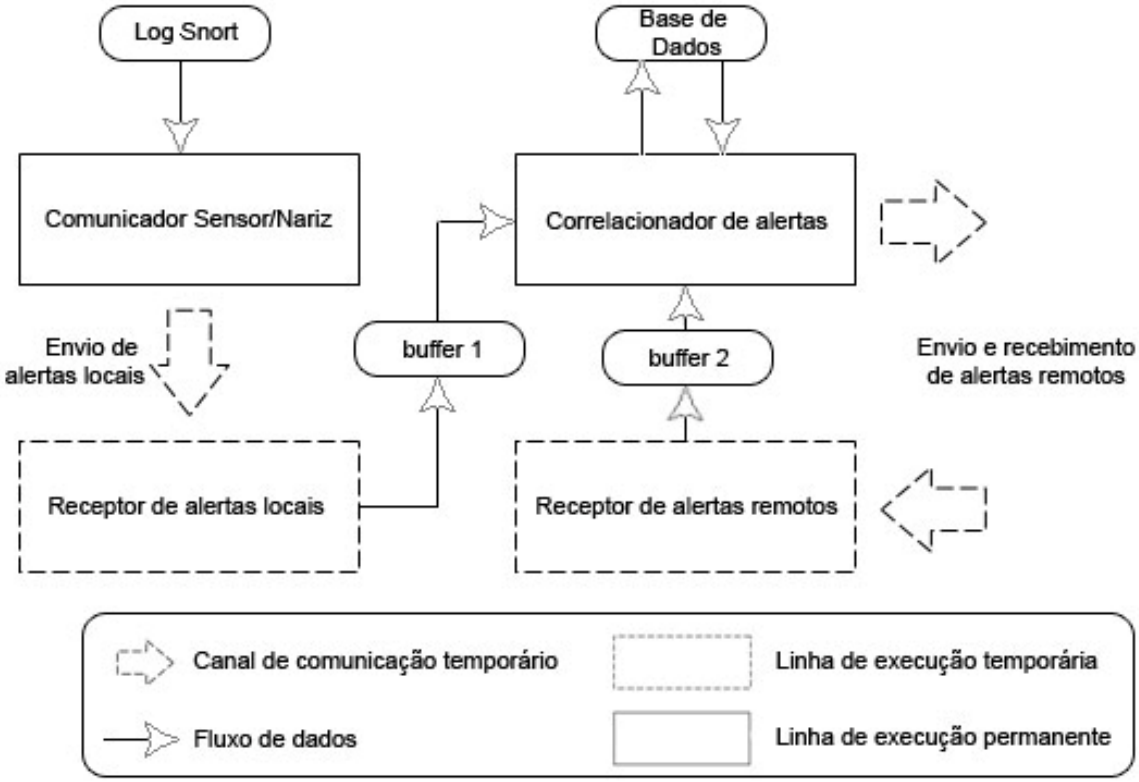

FIGURA 7 - Arquitetura original do sistema de correlacionamento Nariz

- Comunicador Sensor/Nariz (CSN): faz a leitura dos alertas armazenados no arquivo gerado pelo Snort e os transmite ao Receptor de alertas locais;
- Correlacionador de alertas (CA): correlaciona todos os alertas recebidos (locais e remotos), através das regras de correlacionamento, com os alertas já existentes na base de dados local;
- Receptor de alertas locais (RAL): recebe as mensagens enviadas pelo Comunicador Sensor/Nariz e as coloca em espera para serem correlacionadas;
- Receptor de alertas remotos (RAR): recepciona os alertas que foram enviados pelos outros correlacionadores que analisam porções diferentes do tráfego da rede.

O Comunicador Sensor/Nariz fica em espera por alertas gerados pelo Snort. Quando o sensor grava um novo alerta no arquivo de *log* este é lido pelo CSN, que o

envia ao Receptor de alertas locais. O envio e recebimento de alertas locais e remotos são efetuados via canais de comunicação (*"sockets"* TCP/IP).

Na arquitetura original esses canais são temporários porque, para cada alerta transmitido, um novo canal de comunicação é criado e faz-se necessária a realização de um novo processo de conexão para só então a mensagem ser enviada e a conexão finalizada.

Quando a *thread* do Receptor de alertas locais é criada, um *socket* é também inicializado o qual é colocado em modo de espera por conexões (*"listen"*) em uma determinada porta TCP. A linha de execução permanece "congelada" até que o Comunicador Sensor/Nariz realize uma conexão nesta porta e envie uma mensagem de alerta. Ao ser recebido, o alerta é copiado para uma área compartilhada de armazenamento em memória, ou *"buffer"*, e então a *thread* é finalizada.

O Receptor de alertas remotos segue o mesmo algoritmo do RAL, porém, a porta TCP usada pelo *socket* para esperar por conexões e o *buffer* para o qual o alerta recebido é copiado são distintos.

O Correlacionador de alertas é a parte principal do sistema Nariz, pois é ele que realiza todo o trabalho de avaliação das regras de correlacionamento para os alertas e também os acessos à base de dados. O algoritmo em pseudocódigo abaixo, demonstra de maneira geral o funcionamento do CA:

```
enquanto (verdade) {
    Cria_thread(RAL);
    Cria_thread(RAR);
    Espera termino de RAL;

    Correlaciona(buffer1);

    Se (buffer2 != VAZIO)
        Correlaciona(buffer2);
}
```

Primeiramente as duas linhas de execução dos receptores de alertas locais e remotos são criadas para que um novo alerta possa ser recebido. O correlacionador de alertas permanece parado até o RAL receber um alerta do Comunicador Sensor/Nariz.

Então o alerta que foi armazenado na área compartilhada de armazenamento em memória, representado por *"buffer1"* é correlacionado. Se o

“buffer2”, que é a área de armazenamento em memória do receptor de alertas remotos, não estiver vazio um alerta vindo de outro correlacionador remoto foi recebido e é também correlacionado.

Ao fim deste ciclo o correlacionador de alertas cria novamente as *threads* para os receptores RAL e RAR e é por isso que estas são consideradas linhas de execução temporárias, pois a cada alerta recebido a linha é finalizada e recriada posteriormente.

O correlacionamento dos alertas é um ponto crítico em relação ao desempenho do sistema, pois vários acessos de leitura e escrita à base de dados são realizados. Para cada uma das 6 variáveis de correlação o seguinte algoritmo é executado:

a) Todos os alertas da base de dados cujo conteúdo da variável seja igual ao conteúdo da mesma variável do novo alerta são selecionados;
b) Para cada alerta encontrado, o limiar é verificado. Caso tenha atingido o nível de CONVERSA definido, o alerta que está sendo correlacionado é enviado a cada um dos Narizes em execução na rede. Se o nível de PÂNICO é atingido, o alerta é enviado ao Administrador e o limiar é atualizado segundo a fórmula de corte em uso;
c) O limiar é incrementado e as atualizações nas outras variáveis são realizadas.

Caso não exista na base de dados nenhum alerta que satisfaça a igualdade de pelo menos uma das variáveis, o novo alerta é armazenado.

Suponha os 2 alertas seguintes. O primeiro já está presente no banco de dados e o segundo acaba de ser recebido pelo receptor de alertas locais:

```
1. Classe: http_inspect
   IP Origem: 200.201.123.76
   IP Destino: 200.153.9.24
   Rede Origem: 200.201.123
   Rede Destino: 200.153.9
   Porta Destino: 8080
   Data: 29/12
   Hora: 16:40:00
   Limiar: 3
```

```
2. Classe: http_inspect
   IP Origem: 198.2.147.212
   IP Destino: 198.234.72.150
   Rede Origem: 198.2.147
   Rede Destino: 198.234.72
   Porta Destino: 80
   Data: 29/12
   Hora: 17:00:00
```

Ao comparar a variável *“Classe“* do segundo alerta, o primeiro seria selecionado da base de dados. Suponha que seu limiar igual a 3 ainda não tenha atingido nenhum dos níveis de CONVERSA ou PÂNICO, logo seria apenas incrementado e as outras variáveis atualizadas tendo o seguinte resultado:

```
Classe: http_inspect
IP Origem: 200.201.123.76, 198.2.147.212
IP Destino: 200.153.9.24, 198.234.72.150
Rede Origem: 200.201.123, 198.2.147
Rede Destino: 200.153.9, 198.234.72
Porta Destino: 8080, 80
Data: 29/12
Hora: 17:00:00
Limiar: 4
```

As atualizações na base são realizadas à medida que os resultados da busca são retornados. No SQLite, os dados de uma mesma tabela não podem ser escritos enquanto uma leitura é realizada. Uma tabela temporária é criada para armazenar os alertas que tenham satisfeito a condição de busca e então, todos os alertas dessa tabela são selecionados e as atualizações são realizadas na tabela principal.

Para cada alerta correlacionado seis tabelas temporárias são criadas e excluídas, operações estas de alto custo para o sistema, além da execução de 12 comandos de leitura a dados.

3.2.2 A Nova Arquitetura

A arquitetura original do Nariz tem 2 problemas críticos. O primeiro é que o sistema nunca correlacionará alertas remotos enquanto não estiver recebendo

alertas locais. O segundo problema advém da sobrecarga causada ao sistema operacional quando, a cada ciclo de correlacionamento, duas linhas de execução são finalizadas e recriadas.

A FIGURA 8 representa a nova arquitetura do sistema de correlacionamento Nariz. Note que o Receptor de alertas locais e o Receptor de alertas remotos foram unidos em um único Receptor de alertas (RA) e as duas áreas de armazenamento em memória foram substituídas por apenas um *buffer* implementado na forma de fila circular com espaço para múltiplos alertas.

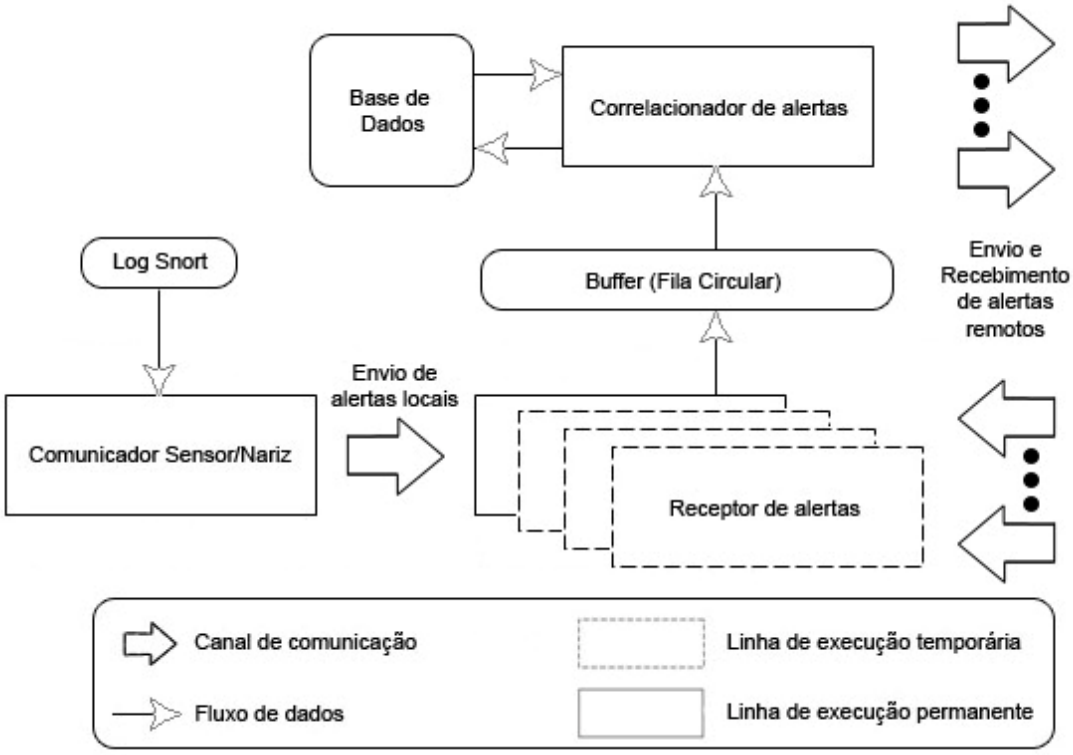

FIGURA 8 - Nova arquitetura para o sistema Nariz

A *thread* do Receptor de alertas é inicializada uma única vez pelo Correlacionador de alertas que passa a verificar o *buffer* em busca de alertas aguardando para serem correlacionados, sem precisar esperar pelo termino da linha de execução do receptor, aproveitando-se melhor da propriedade de paralelismo.

Assim como na arquitetura original, quando o RA é inicializado, um *socket* é criado e colocado em modo de espera por conexões em uma determinada porta IP, porém, para cada pedido de conexão uma nova *thread* é criada pelo receptor.

Essas *threads* de caráter temporário permanecem abertas até que a conexão seja finalizada e recepcionam os alertas enviados, armazenando-os no

buffer. Isso possibilita que o canal de comunicação usado entre o Comunicador Sensor/Nariz e o Receptor de alertas e os usados entre o Correlacionador de Alertas e os Narizes remotos permaneçam abertos, evitando a sobrecarga causada pelas requisições de abertura de conexão TCP/IP.

Geralmente os Narizes remotos são inicializados uma única vez durante o funcionamento normal, então não há sobrecarga ao SO por causa da inicialização excessiva de linhas de execução.

Nessa nova arquitetura, várias *threads* podem acessar o *buffer* ao mesmo tempo, portanto o uso de um mecanismo de sincronismo tornou-se necessário. O mecanismo adotado foi o modelo do produtor – consumidor ilustrado na FIGURA 9, utilizando semáforos. Como o sistema tem apenas um consumidor (Correlacionador de alertas) o uso de variável de exclusão mútua só foi necessário para os produtores (*threads* de recepção de alertas).

```
semaforo cheio = 0, vazio = N;
semaforo lock_prod = 1, lock_cons = 1;
```

Produtor:

```
while (true)
   item = produz();
   wait(vazio);
   wait(lock_prod);
   f = (f + 1) % N;
   buffer[f] = item;
   signal(lock_prod);
   signal(cheio);
```

Consumidor:

```
while (true)

   wait(cheio);
   wait(lock_cons);
   i = (i + 1) % N;
   item = buffer[i];
   signal(lock_cons);
   signal(vazio);
   consome(item);
```

FIGURA 9 - Modelo do produtor - consumidor utilizando semáforos

O que proporcionou o aumento mais significativo em desempenho para o sistema foi a mudança na forma de acesso ao banco de dados. Foi implementado um sistema de “atualizações tardias”, pelo qual os alertas do banco de dados são selecionados diretamente da tabela principal e as atualizações de variáveis e limiares necessárias são colocadas em uma fila de espera.

Após todos os resultados da consulta à base de dados serem processados, as atualizações são executadas na ordem “primeiro a chegar, primeiro a sair”

(PCPS). Esse mecanismo elimina a necessidade de criação de tabelas temporárias melhorando significativamente o desempenho do programa, principalmente quando o tamanho da base torna-se muito grande.

3.3 TESTES COM OUTROS CORRELACIONADORES

Durante a elaboração deste trabalho foram testados alguns sistemas de correlacionamento disponíveis na internet. O objetivo inicial era comparar a quantidade de alertas correlacionados, a velocidade de correlacionamento entre cada correlacionador e a do o nariz, mas como cada sistema testado possui uma arquitetura diferente e o seu próprio conjunto de regras para definir o que pode caracterizar um ataque, a tarefa de comparar os correlacionadores disponíveis com as variáveis descritas tornou-se inviável. Assim, utilizamos a pesquisa para conhecer a anatomia dos correlacionadores, com o intuito de aprender novas técnicas e modelos que pudessem melhorar o Nariz e também pela necessidade de avaliar melhor as diferentes soluções disponíveis.

3.3.1 Simple Event Correlator

O *Simple Event Correlator* (SEC) [12] é uma ferramenta de código aberto, multiplataforma que foi desenvolvida para fazer correlação de eventos que são gerados por soluções comerciais ou até eventos gerados por pequenos scripts. Depois de inicializado o SEC lê a partir de arquivos, ou da entrada padrão, e faz o cruzamento das linhas com padrões (expressões regulares) para reconhecer eventos de entrada e correlacionar de acordo com as regras no arquivo de configuração. Para efetuar os correlacionamentos as regras são lidas sequencialmente na mesma ordem em que foram escritas no arquivo de configuração, regras seqüenciais de diferentes arquivos são utilizadas virtualmente em paralelo. O SEC possui uma lista de ações muito extensa podendo ser realizada praticamente qualquer processamento sobre os eventos.

3.3.2 Open Source Security Information Management

Outra ferramenta de código aberto disponível para sistemas GNU/linux é o *Open Source Security Information Management* (OSSIM) [14] que é uma distribuição de produtos de código aberto que tem por objetivo integrar e prover uma infra-estrutura para monitoramento de redes. Suas premissas são centralizar, organizar, prover um sistema de detecção ajudando na organização dos eventos de segurança.

O sistema incluiu as seguintes ferramentas de monitoramento:

- Painel de controle;
- Monitor de atividades e riscos;
- Console forense e *network monitor* para o baixo nível;
- Sistema de correlação;
- Priorização;
- Avaliação de riscos.

O pós-processamento por sua vez, contém um conjunto de detectores e monitores que são incluídos na distribuição:

- SDI
- Detectores de Anomalias
- *Firewalls*

O OSSIM é uma ferramenta que tem como objetivo ser a estrutura que permite ao administrador da rede gerenciar os recursos, a topologia, a política de segurança, as regras de correlacionamento e integrar todas estas ferramentas de forma mais ou menos homogênea.

3.3.3 Idsa 0.96.2

Outro correlacionador de código aberto investigado foi o Idsa 0.96.2 [13], disponível para sistemas Unix. O Idsa é um conjunto de rotinas e padrões estabelecidos por um programa para utilização de suas funcionalidades (API) acessíveis somente por programas aplicativos para que outras aplicações não dependam exclusivamente dos modelos de segurança tradicionais como *firewalls* e antivírus. A tentativa do projeto do Idsa é ajudar aos programadores adicionarem mecanismos de segurança as suas aplicações de uma forma simples. Idsa faz isto fornecendo um monitor integrado de referência, um log e SDI que está acessível às

aplicações através de um API simples. A aplicação pode usar esta infra-estrutura para delegar o controle de acesso e a detecção do intruso ao Idsa.

4 ESTUDO DE CASO

O registro do Snort, coletado junto ao PoP-Pr, contém alertas de várias classes de ataques. Destacam-se deste grupo três classes, mostradas na TABELA 1, devido ao número elevado de ocorrências registradas. Somadas elas representam aproximadamente 94% do total de alertas.

TABELA 1 - Classes de ataques destaques entre as encontradas nos alertas coletados.

Classe	**Nº. de ocorrências**
ICMP	33.160.986
http_inspect	2.930.646
portscan	473.678
Outras classes	2.243.681
Total de Alertas Coletados	**38.808.991**

O ICMP (*"Internet Control Message Protocol"*) é um protocolo associado ao protocolo IP. As mensagens ICMP são geradas tipicamente em resposta aos erros em datagramas IP ou para fins de diagnóstico ou roteamento.

O Snort pode ser configurado para gerar um alerta sempre que identificar uma mensagem desse tipo na rede. Um número elevado de alertas ICMP pode significar uma tentativa de ataque a um computador da rede através da técnica de negação de serviços (NdS) ou uma tentativa de burlar o SDI. Na maioria dos casos, representa apenas uma falha de configuração da rede caracterizando-se como um falso-positivo.

O *"portscan"* é uma varredura de múltiplas portas de um computador em uma rede procurando por serviços que estejam esperando por conexões. Geralmente é o ponto de partida para ataques, pois procura identificar vulnerabilidades que possam ser exploradas.

A identificação positiva de *portscans* no Snort é feita quando certo número de portas distintas de um computador são acessadas em um intervalo inferior a certo tempo definido, gerando assim um alerta.

A classe escolhida para direcionar os testes, porém, foi a "http_inspect". O http_inspect é um pré-processador do Snort que detecta anomalias e pré-processa e

normaliza requisições do Protocolo HTTP (*"Hypertext Transfer Protocol"*) para facilitar as regras que pesquisam por conteúdo nesse tipo de pacote.

O pré-processador detecta vários tipos de possíveis ataques que geralmente passam despercebidos pelo mecanismo de detecção baseado em padrões. O processo de normalização traduz vários conjuntos de caracteres, tais como Unicode ou HEX, para caracteres que o Snort reconhece. O http_inspect trabalha especificamente com as *"strings"* de URI (*"Uniform Resource Identifier"*) de uma requisição HTTP.

A TABELA 2 apresenta todos os tipos de alertas da classe http_inspect encontrados no *log* coletado e as suas respectivas quantidades. Três dentre esses sete tipos foram destacados: *Double Decoding Attack*, *Oversize Request-Uri Directory* e *Webroot Directory Traversal*.

TABELA 2 - Tipos de alertas da classe http_inspect encontrados no log coletado

Tipo	**Nº. de ocorrências**
Bare Byte Unicode Encoding	2.463.760
Double Decoding Attack	230.923
IIS Unicode Codepoint Encoding	96.651
Oversize Request-Uri Directory	88.232
Webroot Directory Traversal	37.927
Oversize Chunk Encoding	12.152
U Encoding	1.001
Total	**2.930.646**

O motivo para que os três casos fossem destacados para os nossos testes é porque muitos *"Worms"* se utilizam desses tipos de ataques para explorar vulnerabilidades nos sistemas atacados. O *"The Philippine Honeynet Project"* [11] registrou no dia 06/10/2005 a atividade de um *worm* desconhecido que causou a geração de inúmeros eventos *Webroot Directory Traversal* e *Double Decoding Attack* pelo SDI usado.

No ano de 2005, 25% dos incidentes reportados ao CERT foram causados por *worms*, como podemos ver na FIGURA 10. Esse número aumentou para 52% no período de janeiro a setembro de 2006.

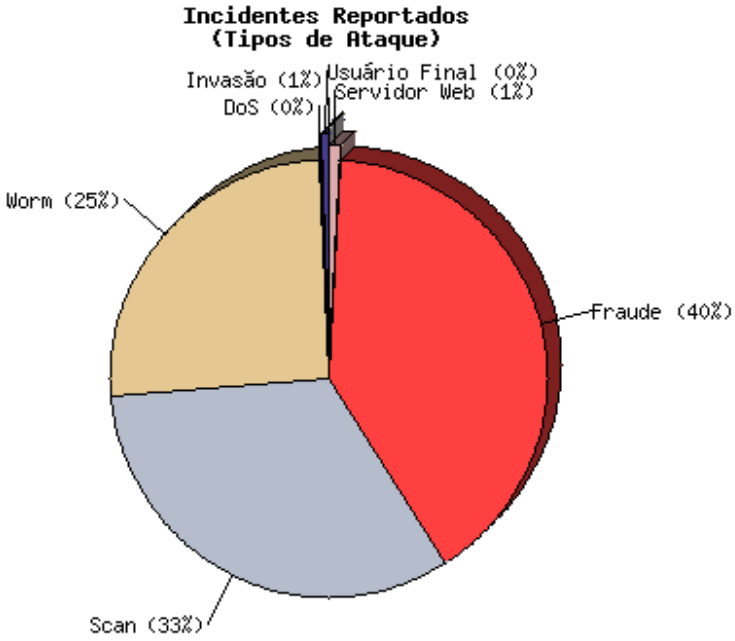

FIGURA 10 - Incidentes reportados ao CERT de Janeiro a Dezembro de 2005
FONTE: CERT.br

4.1 PREPARAÇÃO DOS TESTES

Baseado na média de alertas por hora encontrada no *log* coletado junto ao Pop-PR, um arquivo contendo 45mil alertas foi preparado de modo a conter uma boa quantidade de alertas dos três tipos “http_inspect” escolhidos além de alertas de outras classes para gerar “ruído” e tornar as medições mais próximas de um ambiente real.

A classe dos alertas dos tipos *Double Decoding Attack*, *Oversize Request-Uri Directory* e *Webroot Directory Traversal* foram alteradas para “http_inspect-2”, “http_inspect-15” e “http_inspect-18” respectivamente, para que pudessem ser diferenciados dos demais ataques “http_inspect” e para permitir a configuração de limiares personalizados para cada um.

A TABELA 3 mostra a quantidade de alertas de cada classe encontrada no arquivo preparado e também suas quantidades. Note que existe um número razoável de ataques do tipo Double Decoding e do tipo Webroot Directory Transversal, em analogia ao ataque de um worm desconhecido reportado pelo The Philippine Honeynet Project e citado anteriormente.

TABELA 3 - Classes de ataques presentes no arquivo log preparado

Classe	**Nº. de ocorrências**
http_inspect	20.446
http_inspect-2	5.160
http_inspect-15	359
http_inspect-18	3929
Outras	15.106
Total de Alertas	**45.000**

Os testes foram subdivididos em dois casos e em ambos a fórmula de corte utilizada foi *limiar = round(limiar/3)* e um degrau de valor 2 para alertas remotos correlacionados, conforme proposto por MELLO [05].

Para o primeiro caso, os limiares para as diversas classes de ataque foram definidos de forma que o nível do gatilho de CONVERSA fosse de 30% o valor do gatilho de PÂNICO, mantendo propositadamente o limiar acima do nível CONVERSA após o corte ser efetuado, diminuindo o número de alertas remotos gerados.

Para o segundo caso, os limiares foram definidos de maneira que o nível do gatilho de conversa fosse de 50% o valor do gatilho de PÂNICO. A TABELA 4 mostra os limiares usados em cada classe para ambos os casos.

TABELA 4 - Configuração de limiares usada nos testes

Classe	**Caso 1**	**Caso 2**
http_inspect-2	C = 30, P = 100	C = 50, P = 100
http_inspect-15	C = 30, P = 100	C = 50, P = 100
http_inspect-18	C = 18, P = 60	C = 30, P = 60
PADRAO	C = 60, P = 200	C = 100, P = 200

Foi definido um limiar mais baixo para os gatilhos da classe http_inspect-18 (Webroot Directory Transversal) para que esta classe de alertas fosse priorizada, pois os danos causados por esse tipo de ataque podem ser grandes já que o atacante tenta transpassar as barreiras do diretório raiz do servidor *web* para conseguir privilégios no computador atacado.

4.2 RESULTADOS

Foram efetuados experimentos com 1, 2 e 3 correlacionadores em ambos os casos de teste citados acima, utilizando um número de alertas igualmente divididos entre os Narizes: 45 mil alertas para um Nariz, 22.500 alertas para cada um de dois Narizes e 15 mil alertas para cada um de três Narizes. Cada correlacionador foi executado em um computador diferente de uma rede.

Após as alterações realizadas no código, o desempenho do sistema de correlacionamento nariz melhorou consideravelmente. O gráfico da FIGURA 11 mostra os tempos de execução em segundos para os testes realizados com 1, 2 e 3 correlacionadores.

Infelizmente, um erro que ainda não foi identificado impediu que o teste utilizando 3 narizes para o caso em que a maior quantidade de alertas remotos é enviada (caso 2) pudesse ser concluído.

Tomando como base os registros coletados, percebe-se que com apenas um correlacionador (sistema centralizado), o Nariz não é capaz de correlacionar 45.000 alertas em uma hora, demonstrando portanto, desempenho insuficiente para tratar todos os alertas recebidos. Os atrasos no processamento foram relativamente pequenos, de 11 e 10 minutos para o primeiro e segundo casos, respectivamente.

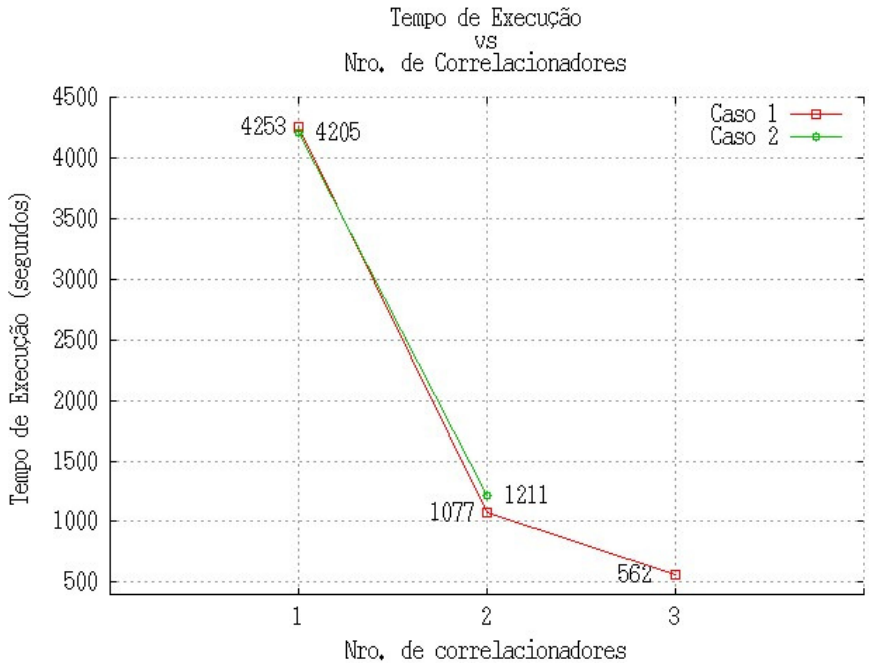

FIGURA 11 - Tempo de Execução vs Nro. de Correlacionadores

Aumentando o número de correlacionadores, observa-se uma melhora significativa no tempo de execução e na taxa de alertas correlacionados a cada segundo, como pode ser visto na FIGURA 12. Isso demonstra a eficiência do correlacionamento distribuído com o correlacionamento dos 45 mil alertas em aproximadamente 20 minutos com 2 correlacionadores e 10 minutos com três.

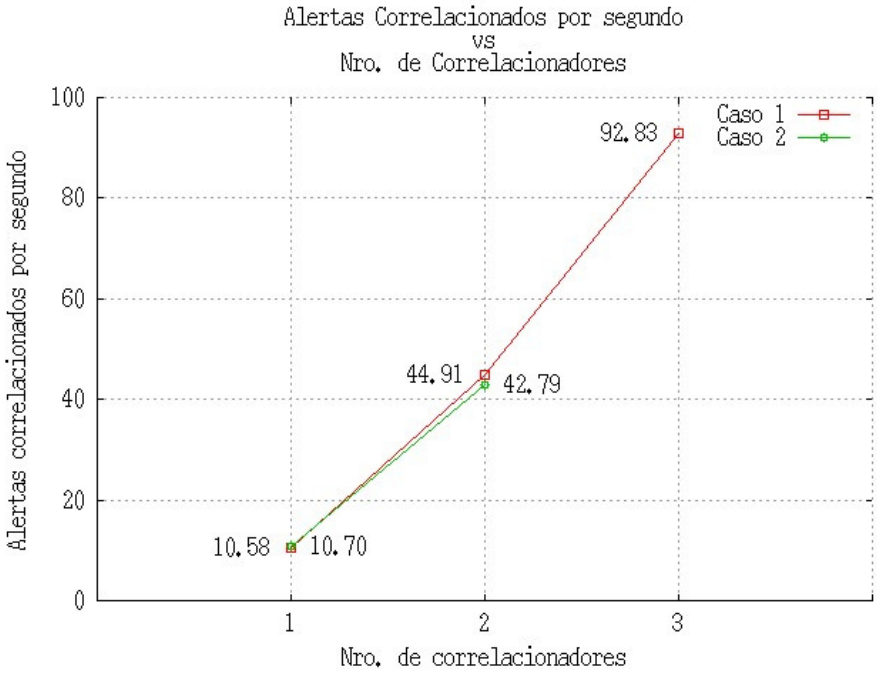

FIGURA 12 - Taxa de correlacionamentos por segundo

À medida que o número de correlacionadores aumenta, o número de alertas a serem correlacionados também aumenta (FIGURA 13) por causa das mensagens remotas do correlacionamento distribuído. Esse aumento no número de alertas remotos pode gerar um efeito cascata no correlacionamento distribuído, pois o limiar de cada alerta correlacionado aumenta com o dobro da velocidade, devido ao degrau de valor 2. Dependendo da configuração de limiares, o aumento excessivo do número de narizes pode prejudicar o desempenho do sistema.

A quantidade de alertas que serão repassados ao administrador, que é o efeito de filtragem desejado, também depende da configuração dos limiares. O gráfico da FIGURA 14 mostra a porcentagem do total de alertas correlacionados que são repassados ao administrador.

No caso em que menos alertas remotos são trocados entre os vários correlacionadores, a relação entre alertas correlacionados e alertas enviados ao

administrador mantêm-se menor. Já no outro caso, o número de mensagens repassadas ao administrador aumenta rapidamente.

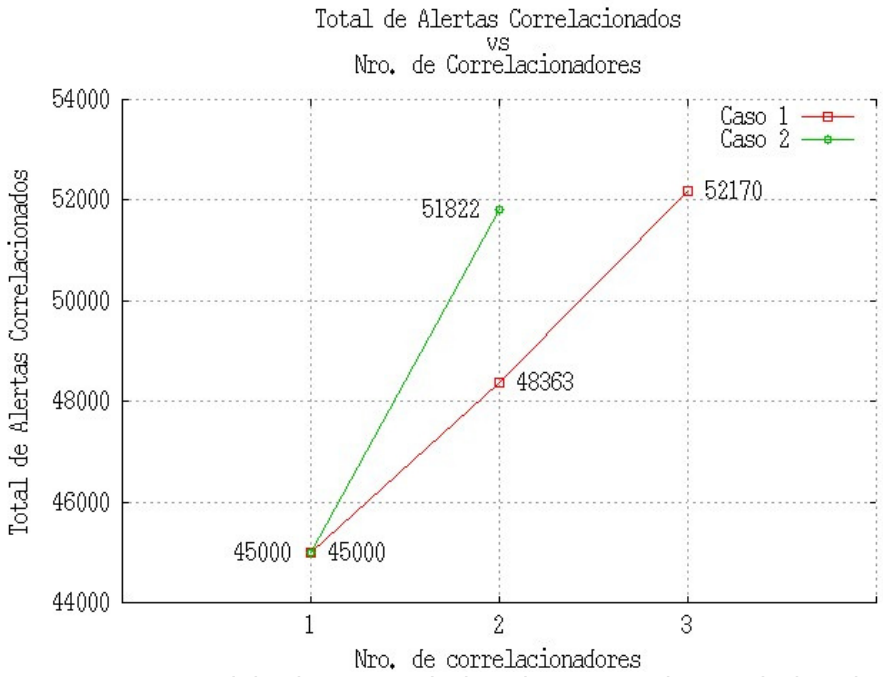

FIGURA 13 - Total de alertas correlacionados vs Nro. de Correlacionadores

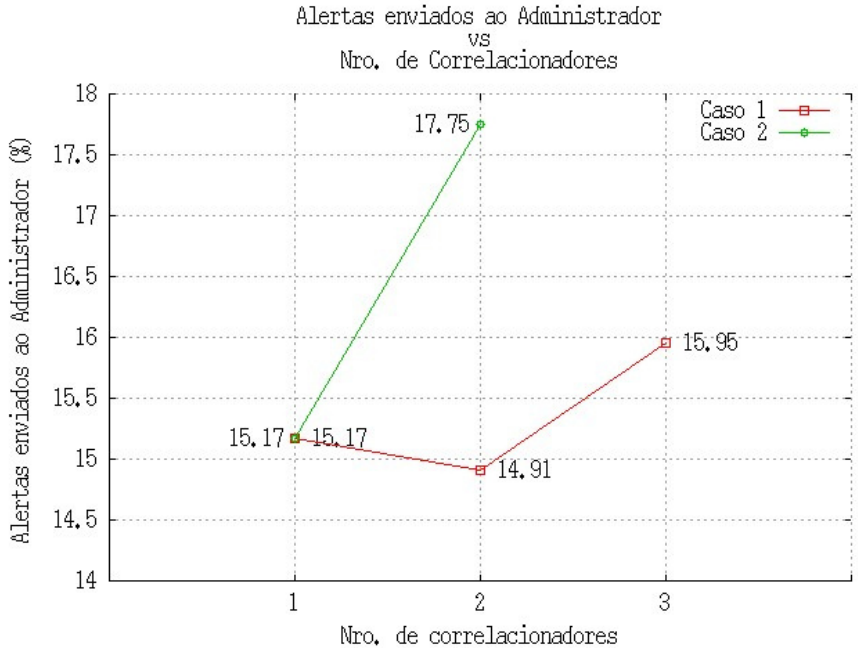

FIGURA 14 - Porcentagem do total de alertas correlacionados repassados ao administrador

As FIGURA 15 e FIGURA 16 mostram a quantidade de alertas enviados ao administrador separados por classe para o primeiro e segundo casos, respectivamente.

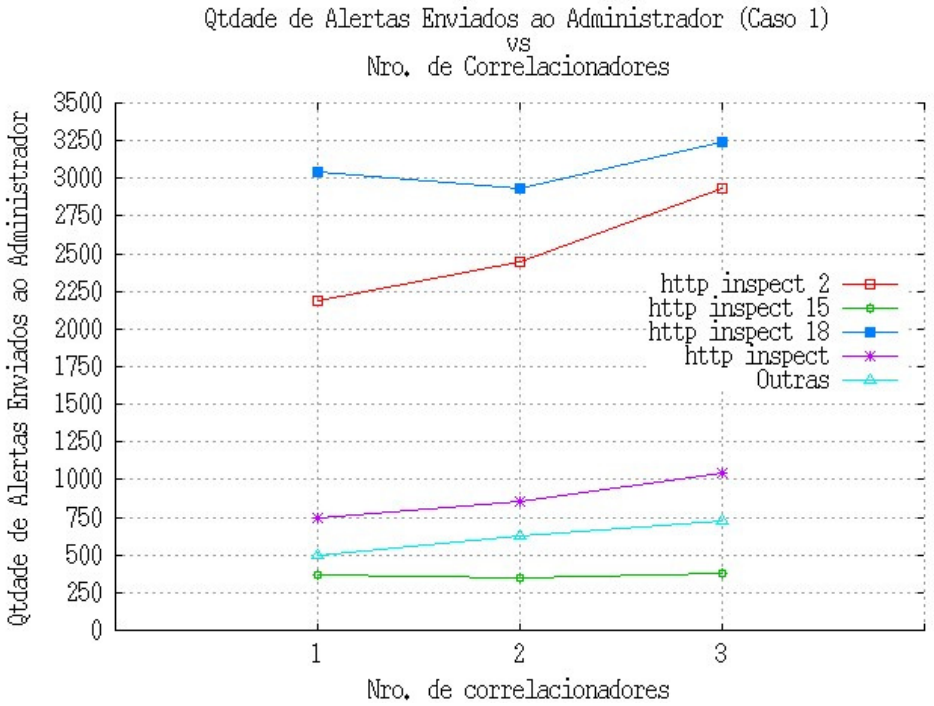

FIGURA 15 - Quantidade de alertas por classe enviadas ao administrador (Caso 1)

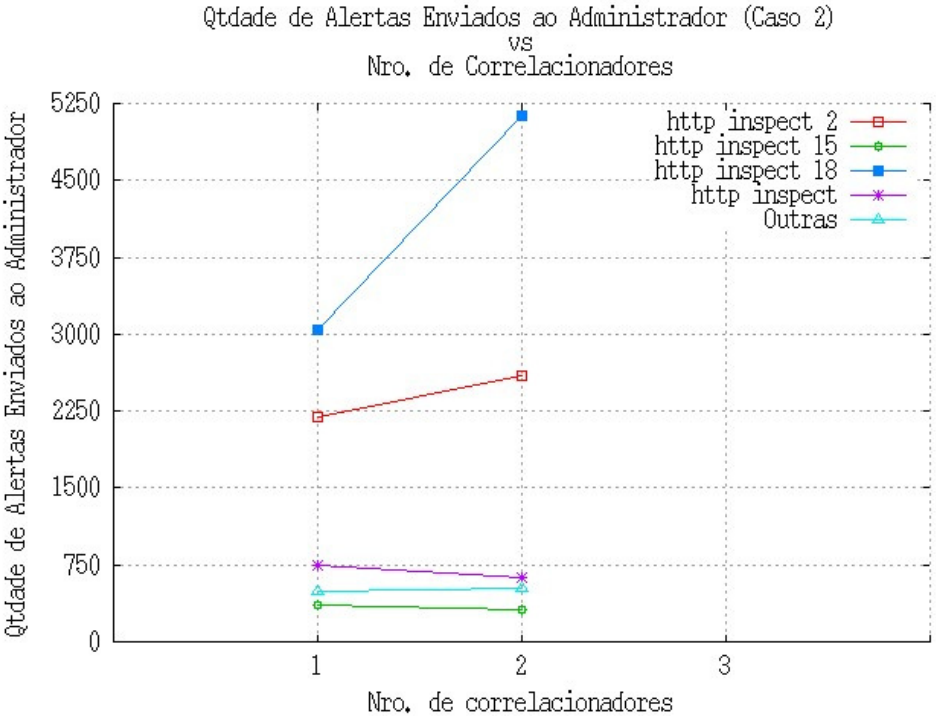

FIGURA 16 - Quantidade de alertas por classe enviadas ao administrador (Caso 2)

No segundo caso, o número de mensagens do tipo http_inspect-18 enviadas quando dois correlacionadores estão em funcionamento é maior do que o número inicial de alertas desse tipo existentes nos registros. Isso significa que o

correlacionador se comportou de maneira indesejada, devido à configuração de limiares utilizada que gera um número excessivo de correlacionamentos remotos fazendo com que o limiar dos alertas desta classe atinja rapidamente o gatilho de PÂNICO.

Para o primeiro caso, o número de mensagens aumenta de forma lenta quando aumentamos o número de correlacionadores. Outro comportamento, sendo este desejado, que pode ser visto nos gráficos é que a classe com menor limiar é também a que teve mais alertas do seu tipo enviadas ao administrador do sistema.

As modificações realizadas no código produziram resultado, pois eliminaram a sobrecarga que anteriormente era causada ao SO e melhoraram o desempenho do correlacionador. Embora o sistema centralizado tenha se mostrado ineficiente, o comportamento do correlacionador distribuído foi o desejado, com um efeito de filtragem bem caracterizado de acordo com a configuração dos limiares.

CONCLUSÃO

Com o aumento da interconexão entre redes de computadores a preocupação em fornecer segurança contra intrusões aos sistemas computacionais tornou-se fundamental. Sistemas de detecção de intrusão de redes (SDIR) são capazes de detectar atividades consideradas danosas através análise de trafego e quanto mais intenso o trafego, maior deve ser a capacidade de processamento do SDIR. Os sensores emitem alertas ao administrador da rede quando é detectada alguma atividade considerada maliciosa. Para reduzir o número de alertas, e ao mesmo tempo aumentar o conteúdo semântico, os alertas são correlacionados aos alertas previamente inseridos em uma base de dados.

Este trabalho descreve alguns de testes de desempenho e modificações no código do Nariz, um sistema que efetua o correlacionamento de alertas, gerados por sistemas de detecção de intrusão, de forma distribuída, possibilitando assim o emprego de sensores que executam em máquinas de menor custo e/ou capacidade, reduzindo o custo da detecção de ataques. O mecanismo de correlacionamento distribuído do Nariz é baseado na distribuição de tráfego para vários pares sensor-correlacionador, e na comunicação entre aqueles para produzir poucos alertas com elevado conteúdo semântico, para serem analisados pelos administradores da rede. A configuração do nariz é bastante flexível, permitindo ajustes finos na detecção de atividades suspeitas, a cada classe de alarme são associados quatro parâmetros de configuração que determinam a freqüência com que alarmes serão enviados ao administrador. Estes parâmetros também determinam o grau de cooperação entre os correlacionadores.

Durante a fase inicial do projeto foi realizada a coleta de alertas na rede de entrada do o PoP-PR, justamente por se tratar de uma rede de trafego intenso. Ao todo foram coletados 38.808.991 alertas durante o período de 36 dias, o que corresponde à aproximadamente 1.080.000 alertas diários. Paralelamente fez-se um estudo com alguns sistemas de correlacionamento disponíveis, com os objetivos de conhecer o funcionamento de outras soluções e também de aprender novas técnicas e abordagens que pudessem melhorar o desempenho do Nariz.

Para tornar o Nariz mais eficiente, melhorando o seu tempo de execução e ao mesmo tempo reduzindo a demanda por recursos computacionais necessários,

algumas alterações no código fonte original foram realizadas. No projeto original, para cada mensagem enviada entre os correlacionadores era criada uma conexão, a mensagem então era transmitida entre os pares de correlacionadores e a conexão encerrada consumindo muito tempo. O código foi alterado para que os Narizes passassem a utilizar conexões permanentes evitando a criação e o encerramento de conexões em excesso.

A forma como as diversas *threads* se relacionavam também foi aprimorada. Uma única área de armazenamento em memória é compartilhada entre as linhas de execução que fazem a recepção de alertas e o correlacionador, sendo seu acesso sincronizado através do modelo produtor-consumidor utilizando semáforos, eliminando assim a necessidade de criação e finalização de várias *threads* durante a execução do Nariz.

O método de atualização das tabelas do banco de dados foi aprimorado do modo que, após as mudanças no código, para cada alerta correlacionado sete tabelas temporárias deixam de ser criadas, tornando o Nariz mais rápido e mais eficiente. Paralelamente às melhorias no código alguns estudos sobre as características dos ataques foram realizados. Este estudo tinha como objetivo definir os limiares e os gatilhos de modo a melhorar o conteúdo semântico das mensagens enviadas ao administrador da rede..

Os resultados dos testes mostraram que um Nariz (sistema centralizado) é insuficiente para processar a demanda de uma rede de alto tráfego como a do PoP-PR. Aumentando o número de correlacionadores, tem-se uma melhora significativa no tempo de execução e na taxa de alertas correlacionados a cada segundo demonstrando a eficiência do correlacionamento distribuído. Considerando a média de 45 mil alertas gerados por hora, dois narizes correlacionaram os alertas em aproximadamente 20 minutos e 3 narizes supriram a demanda em 10 minutos. Alguns pontos da implementação do Nariz precisam de maiores testes, tais como a aferição de quantos pares de sensores-correlacionador são necessários para determinada intensidade de trafego e também uma comparação com SDIR centralizado.

Trabalhos futuros incluem a definição de regras de correlacionamento baseado em tempo, estudos sobre a estabilidade na geração de alarmes em sistemas com grande número de correlacionadores e correção do erro que não

permitiu que o teste utilizando 3 narizes para o caso em que a maior quantidade de alertas remotos é enviada fosse realizado.

REFERÊNCIAS BIBLIOGRÁFICAS

[01] AMOROSO, Edward. **Intrusion Detection: An Introduction to Internet Surveillance, Correlation, Trace Back, Traps, and Response**. Sparta, New Jersey: Intrusion.Net Books, 1999.

[02] CAMPELLO, Rafael Saldanha; WEBER, Raul Fernando. **Sistemas de detecção de intrusão**, In: SIMPÓSIO BRASILEIRO DE REDES DE COMPUTADORES, 19., 2001, Florianópolis. P.143.

[03] DEBAR, H. and WESPI, A. **Aggregation and Correlation of Intrusion-Detection Alerts**, In: RECENT ADVANCES IN INTRUSION DETECTION, 2001.

[04] HAINES, J.; RYDER, K. D.; TINNEL, L.; TAYLOR, S. **Validation of Sensor Alert Correlators**, In: INTRUSION ALERT CORRELATION, IEE Computer Society, 2003.

[05] MELLO, Thiago. Nariz: **Um sistema de correlacionamento distribuído de alertas**. Curitiba, 2004. 81p. Dissertação (Programa de Pós-Graduação em Informática) – Setor de Ciências Exatas, Universidade Federal do Paraná.

[06] MIRKOVIC, J.; DIETRICH, S.; DITTRICH, D.; REIHER, P. **Internet Denial of Service: Attack and Defense Mechanisms**. 1. ed. Prentice Hall PTR, 2004.

[07] NING, P.; CUI, Y. and REEVES, D. **Analyzing Intensive Intrusion Alerts via Correlation**, In: RECENT ADVANCES IN INTRUSION DETECTION, 2002.

[08] PORRAS, P. A.; FONG, M. W. and VALDES, A. **A Mission-Impact-Based Approach to INFOSEC Alarm Correlation**, In: RECENT ADVANCES IN INTRUSION DETECTION, 2002.

[09] PROCTOR, Paul. **Practical Intrusion Detection Handbook**. Upper Saddle River, New Jersey: Prentice Hall, 2001.

[10] YEGNESWARAN, V.; BARFORD, P. and ULRICH, J. **Internet Intrusions: Global Characteristics and Prevalence**, In: ACM SIGMETRICS PERFORMANCE EVALUATION REVIEW, v. 31, n. 1, 2003. p.138-147.

[11] The Philippine Honeynet Project. **Honeynet Activity Monitor Report Archive 2005-10-06, Webroot Directory Traversal, EXPLORE.EXE and bloated logs...** Disponível em: <http://www.philippinehoneynet.org/dataarchive.php?date=2005-10-06> Acesso em: 15 dez. 2006.

[12] Simple Event Correlator. Disponível em: <http://www.estpak.ee/~risto/sec/>

[13] IDSA. Disponível em: <http://www.net-security.org/software.php?id=184>

[14] Open Source Security Information Management. Disponível em: <http://www.ossim.net/>

[15] Snort. Disponível em: <http://www.snort.org>

[16] SQLite Home Page. Disponível em: <http://www.sqlite.org/>

APÊNCIDE I - SCRIPT PARA TROCA DE IP

```
%{
      #include <math.h>
      #include <stdio.h>
      #include <stdlib.h>

      int nips = 0;
      int linhas = 0;
      int ind;

      int tabClass [256]; // Classe A: 1 - 127; Classe B: 128 - 191; Classe
C: 192 - 223;
      int tabTroca [256]; // Troca geral dos octetos

      int octetos [4]; // octetos do IP
      char newIP [15];
      char auxIP [3];

      void splitIP (int oct[], char * ip) {
            int tam = strlen(ip);
            int i, j = 0;
            char *aux;

            for (j=0; j < 4; j++) {
                  for (i=0; i < tam; i++)
                        if (ip[i] == '.')
                              break;

                  aux = (char *)malloc(i * sizeof(char));
                  strncpy(aux, ip, i);
                  oct[j] = atoi(aux);
                  tam -= (i + 1);
                  if (j != 3)
                        ip += (i + 1);
            }
      }

      int retOct (int oct[], int n){
            return (oct[n]);
      }
%}

OCTETO      [0-9]?[0-9]?[0-9]
IP          {OCTETO}\.{OCTETO}\.{OCTETO}\.{OCTETO}

%%

{IP}  {
      //fprintf (yyout, "IP");
      nips++;
      splitIP (octetos, yytext);
      sprintf (newIP, "%d.%d.%d.%d",
            tabClass[retOct(octetos, 0)], tabTroca[retOct(octetos, 1)],
            tabTroca[retOct(octetos, 2)], tabTroca[retOct(octetos, 3)]
            );
      fprintf (yyout, "%s", newIP);
}
[\n]    { fprintf (yyout, "%s", yytext); linhas++; }
```

```
[^{IP}\n]   { fprintf (yyout, "%s", yytext); }

%%

int yywrap ()
{
      return 1;
}

int procura (int inicio, int fim, int tab[], int val) {
      int i;

      for (i = inicio; i <= fim; i++) {
            if (tab[i] == val)
                  return 1;
      }
      return 0;
}


int main (int argc, char *argv[])
{
      int r; // numero aleatorio
      int i;

      srand (time (NULL)); // hora como semente

      if (argc == 3) {
            yyin = fopen (argv[1], "r");
            yyout = fopen (argv[2], "w");
      }
      else {
            printf ("\n./ipswap [arquivo_fonte] [arquivo_saida]\n\n");
            exit(1);
      }

      // inicializa os vetores de troca
      tabClass[0] = 0;

      printf ("Inicializando tabela de troca do primeiro octeto (Classe A)
........");
      fflush (stdout);
      // inicializa classe A
      for (i = 1; i <= 127; i++){
            r = fmod(rand(), 127) + 1;
            if (procura(1, i, tabClass, r) == 0)
                  tabClass[i] = r;
            else
                  i--;
      }
      printf ("[ OK ]\n");
      printf ("Inicializando tabela de troca do primeiro octeto (Classe B)
........");
      fflush (stdout);
      // inicializa classe B
      for (i = 128; i <= 191; i++) {
            r = fmod(rand(), 64) + 128;
            if (procura(128, i, tabClass, r) == 0)
                  tabClass[i] = r;
            else
                  i--;
```

```
	}
	printf ("[ OK ]\n");
	printf ("Inicializando tabela de troca do primeiro octeto (Classe C)
........");
	fflush (stdout);
	// inicializa classe C
	for (i = 192; i <= 223; i++) {
		r = fmod(rand(), 32) + 192;
		if (procura(192, i, tabClass, r) == 0)
			tabClass[i] = r;
		else
			i--;
	}
	printf ("[ OK ]\n");
	printf ("Inicializando tabela de troca para os demai Octetos
................");
	fflush (stdout);
	// inicializa resto do vetor
	for (i = 224; i <= 255; i++){
		r = fmod(rand(), 32) + 224;
		if (procura(224, i, tabClass, r) == 0)
			tabClass[i] = r;
		else
			i--;
	}
	// inicializa vetor padrão para os demais Octetos
	for (i = 0; i < 256; i++){
		tabTroca[i] = 999; // só para não ficar igual a nenhum numero
entre 0 e 255
		r = fmod(rand(), 256);
		if (procura(0, i, tabTroca, r) == 0)
			tabTroca[i] = r;
		else
			i--;
	}
	printf ("[ OK ]\n\n");
	printf ("Trocando IP's do arquivo de entrada
................................");
	fflush (stdout);
	yylex ();
	printf ("[ OK ]\n");

	// imprime tabelas
//	for (i = 0; i < 256; i++)
//		printf ("tabs[%d] => \t%3d\t%3d\n", i, tabClass[i],
tabTroca[i]);

	printf ("\nLinhas Processadas = %d\nIP's encontrados e substituidos =
%d\n\n", linhas, nips);

	return 0;
}
```

ANEXO I - ARQUIVO DE CONFIGURAÇÃO DO SNORT

```
#--------------------------------------------------
#   http://www.snort.org     Snort 2.4.0 config file
#     Contact: snort-sigs@lists.sourceforge.net
#--------------------------------------------------
# $Id: snort.conf,v 1.144.2.9.2.12 2005/07/27 15:56:37 roesch Exp $
#
##################################################
# This file contains a sample snort configuration.
# You can take the following steps to create your own custom configuration:
#
#  1) Set the variables for your network
#  2) Configure preprocessors
#  3) Configure output plugins
#  4) Add any runtime config directives
#  5) Customize your rule set
#
##################################################
# Step #1: Set the network variables:
#
# You must change the following variables to reflect your local network.The
# variable is currently setup for an RFC 1918 address space.
#
# You can specify it explicitly as:
#
# var HOME_NET 10.1.1.0/24
#
# or use global variable $<interfacename>_ADDRESS which will be always
# initialized to IP address and netmask of the network interface which you
# run snort at.  Under Windows, this must be specified as
# $(<interfacename>_ADDRESS), such as:
# $(\Device\Packet_{12345678-90AB-CDEF-1234567890AB}_ADDRESS)
#
# var HOME_NET $eth0_ADDRESS
#
# You can specify lists of IP addresses for HOME_NET
# by separating the IPs with commas like this:
#
# var HOME_NET [10.1.1.0/24,192.168.1.0/24]
#
# MAKE SURE YOU DON'T PLACE ANY SPACES IN YOUR LIST!
#
# or you can specify the variable to be any IP address
# like this:

var HOME_NET any

# Set up the external network addresses as well.  A good start may be "any"
var EXTERNAL_NET any

# Configure your server lists.  This allows snort to only look for attacks
# to systems that have a service up.  Why look for HTTP attacks if you are
# not running a web server?  This allows quick filtering based on IP
# addresses These configurations MUST follow the same configuration scheme
# as defined above for $HOME_NET.

# List of DNS servers on your network
var DNS_SERVERS $HOME_NET
```

```
# List of SMTP servers on your network
var SMTP_SERVERS $HOME_NET

# List of web servers on your network
var HTTP_SERVERS $HOME_NET

# List of sql servers on your network
var SQL_SERVERS $HOME_NET

# List of telnet servers on your network
var TELNET_SERVERS $HOME_NET

# List of snmp servers on your network
var SNMP_SERVERS $HOME_NET

# Configure your service ports.  This allows snort to look for attacks
# destined to a specific application only on the ports that application
# runs on. For example, if you run a web server on port 8081, set your
# HTTP_PORTS variable like this:
#
# var HTTP_PORTS 8081
#
# Port lists must either be continuous [eg 80:8080], or a single port [eg
# 80]. We will adding support for a real list of ports in the future.

# Ports you run web servers on
#
# Please note:  [80,8080] does not work.
# If you wish to define multiple HTTP ports,
#
## var HTTP_PORTS 80
## include somefile.rules
## var HTTP_PORTS 8080
## include somefile.rules
var HTTP_PORTS 80

# Ports you want to look for SHELLCODE on.
var SHELLCODE_PORTS !80

# Ports you do oracle attacks on
var ORACLE_PORTS 1521

# other variables
#
# AIM servers.  AOL has a habit of adding new AIM servers, so instead of
# modifying the signatures when they do, we add them to this list of
# servers.
var AIM_SERVERS
[64.12.24.0/23,64.12.28.0/23,64.12.161.0/24,64.12.163.0/24,64.12.200.0/24,2
05.188.3.0/24,205.188.5.0/24,205.188.7.0/24,205.188.9.0/24,205.188.153.0/24
,205.188.179.0/24,205.188.248.0/24]

# Path to your rules files (this can be a relative path)
# Note for Windows users:  You are advised to make this an absolute path,
# such as:  c:\snort\rules
var RULE_PATH ../rules

# Configure the snort decoder
# ============================
#
```

```
# Snort's decoder will alert on lots of things such as header
# truncation or options of unusual length or infrequently used tcp options
#
#
# Stop generic decode events:
#
# config disable_decode_alerts
#
# Stop Alerts on experimental TCP options
#
# config disable_tcpopt_experimental_alerts
#
# Stop Alerts on obsolete TCP options
#
# config disable_tcpopt_obsolete_alerts
#
# Stop Alerts on T/TCP alerts
#
# In snort 2.0.1 and above, this only alerts when a TCP option is detected
# that shows T/TCP being actively used on the network.  If this is normal
# behavior for your network, disable the next option.
#
# config disable_tcpopt_ttcp_alerts
#
# Stop Alerts on all other TCPOption type events:
#
# config disable_tcpopt_alerts
#
# Stop Alerts on invalid ip options
#
# config disable_ipopt_alerts

# Configure the detection engine
# ==============================
#
# Use a different pattern matcher in case you have a machine with very
# limited resources:
#
# config detection: search-method lowmem

# Configure Inline Resets
# =======================
#
# If running an iptables firewall with snort in InlineMode() we can now
# perform resets via a physical device. We grab the indev from iptables
# and use this for the interface on which to send resets. This config
# option takes an argument for the src mac address you want to use in the
# reset packet.  This way the bridge can remain stealthy. If the src mac
# option is not set we use the mac address of the indev device. If we
# don't set this option we will default to sending resets via raw socket,
# which needs an ipaddress to be assigned to the int.
#
# config layer2resets: 00:06:76:DD:5F:E3

##################################################
# Step #2: Configure preprocessors
#
# General configuration for preprocessors is of
# the form
# preprocessor <name_of_processor>: <configuration_options>
```

```
# Configure Flow tracking module
# ------------------------------
#
# The Flow tracking module is meant to start unifying the state keeping
# mechanisms of snort into a single place. Right now, only a portscan
# detector is implemented but in the long term,  many of the stateful
# subsystems of snort will be migrated over to becoming flow plugins. This
# must be enabled for flow-portscan to work correctly.
#
# See README.flow for additional information
#
preprocessor flow: stats_interval 0 hash 2

# frag2: IP defragmentation support
# ---------------------------------
# This preprocessor performs IP defragmentation.  This plugin will also
# detect people launching fragmentation attacks (usually DoS) against
# hosts.  No arguments loads the default configuration of the preprocessor,
# which is a 60 second timeout and a 4MB fragment buffer.

# The following (comma delimited) options are available for frag2
#    timeout [seconds] - sets the number of [seconds] that an unfinished
#                        fragment will be kept around waiting for
#                       completion, if this time expires the fragment will
#                        be flushed
#    memcap [bytes] - limit frag2 memory usage to [number] bytes
#                      (default:  4194304)
#
#    min_ttl [number] - minimum ttl to accept
#
#    ttl_limit [number] - difference of ttl to accept without alerting
#                         will cause false positves with router flap
#
# Frag2 uses Generator ID 113 and uses the following SIDS
# for that GID:
#  SID     Event description
# -----   -------------------
#   1       Oversized fragment (reassembled frag > 64k bytes)
#   2       Teardrop-type attack

#preprocessor frag2

# frag3: Target-based IP defragmentation
# --------------------------------------
#
# Frag3 is a brand new IP defragmentation preprocessor that is capable of
# performing "target-based" processing of IP fragments.  Check out the
# README.frag3 file in the doc directory for more background and
# configuration information.
#
# Frag3 configuration is a two step process, a global initialization phase
# followed by the definition of a set of defragmentation engines.
#
# Global configuration defines the number of fragmented packets that Snort
# can track at the same time and gives you options regarding the memory cap
# for the subsystem or, optionally, allows you to preallocate all the
# memory for the entire frag3 system.
#
# frag3_global options:
#   max_frags: Maximum number of frag trackers that may be active at once.
#              Default value is 8192.
```

```
#   memcap: Maximum amount of memory that frag3 may access at any given
#           time. Default value is 4MB.
#   prealloc_frags: Maximum number of individual fragments that may be
#                   processed at once.  This is instead of the memcap
#                   system, uses static allocation to increase performance.
#                   No default value. Each preallocated fragment eats ~1550
#                   bytes.
#
# Target-based behavior is attached to an engine as a "policy" for handling
# overlaps and retransmissions as enumerated in the Paxson paper. There are
# currently five policy types available: "BSD", "BSD-right", "First",
# "Linux" and "Last". Engines can be bound to bound to standard Snort CIDR
# blocks or IP lists.
#
# frag3_engine options:
#   timeout: Amount of time a fragmented packet may be active before
#            expiring. Default value is 60 seconds.
#   ttl_limit: Limit of delta allowable for TTLs of packets in the
#              fragments. Based on the initial received fragment TTL.
#   min_ttl: Minimum acceptable TTL for a fragment, frags with TTLs below
#            this value will be discarded. Default value is 0.
#   detect_anomalies: Activates frag3's anomaly detection mechanisms.
#   policy: Target-based policy to assign to this engine.  Default is BSD.
#   bind_to: IP address set to bind this engine to.  Default is all hosts.
#
# Frag3 configuration example:
#preprocessor frag3_global: max_frags 65536 prealloc_frags 262144
#preprocessor frag3_engine: policy linux \
#                           bind_to [10.1.1.12/32,10.1.1.13/32] \
#                           detect_anomalies
#preprocessor frag3_engine: policy first \
#                           bind_to 10.2.1.0/24 \
#                           detect_anomalies
#preprocessor frag3_engine: policy last \
#                           bind_to 10.3.1.0/24
#preprocessor frag3_engine: policy bsd

preprocessor frag3_global: max_frags 65536
preprocessor frag3_engine: policy first detect_anomalies


# stream4: stateful inspection/stream reassembly for Snort
#--------------------------------------------------------------------
# Use in concert with the -z [all|est] command line switch to defeat
# stick/snot against TCP rules. Also performs full TCP stream reassembly,
# stateful inspection of TCP streams, etc. Can statefully detect various
# portscan types, fingerprinting, ECN, etc.

# stateful inspection directive
# no arguments loads the defaults (timeout 30, memcap 8388608)
# options (options are comma delimited):
#   detect_scans - stream4 will detect stealth portscans and generate
#                  alerts when it sees them when this option is set
#   detect_state_problems - detect TCP state problems, this tends to be
#                           very noisy because there are a lot of crappy ip
#                           stack implementations out there
#
#   disable_evasion_alerts - turn off the possibly noisy mitigation of
#                            overlapping sequences.
#
#
```

```
#   min_ttl [number]       - set a minium ttl that snort will accept to
#                            stream reassembly
#
#   ttl_limit [number]     - differential of the initial ttl on a session
#                            versus the normal that someone may be playing
#                            games. Routing flap may cause lots of false
#                            positives.
#
#   keepstats [machine|binary] - keep session statistics, add "machine" to
#                          get them in a flat format for machine reading,
#                          add "binary" to get them in a unified binary
#                          output format
#   noinspect - turn off stateful inspection only
#   timeout [number] - set the session timeout counter to [number] seconds,
#                        default is 30 seconds
#   max_sessions [number] - limit the number of sessions stream4 keeps
#                          track of
#   memcap [number] - limit stream4 memory usage to [number] bytes
#   log_flushed_streams - if an event is detected on a stream this option
#                          will cause all packets that are stored in the
#                          stream4packet buffers to be flushed to disk. This
#                          only works when logging in pcap mode!
#   server_inspect_limit [bytes] - Byte limit on server side inspection.
#
# Stream4 uses Generator ID 111 and uses the following SIDS
# for that GID:
#  SID     Event description
# -----   -------------------
#   1       Stealth activity
#   2       Evasive RST packet
#   3       Evasive TCP packet retransmission
#   4       TCP Window violation
#   5       Data on SYN packet
#   6       Stealth scan: full XMAS
#   7       Stealth scan: SYN-ACK-PSH-URG
#   8       Stealth scan: FIN scan
#   9       Stealth scan: NULL scan
#   10      Stealth scan: NMAP XMAS scan
#   11      Stealth scan: Vecna scan
#   12      Stealth scan: NMAP fingerprint scan stateful detect
#   13      Stealth scan: SYN-FIN scan
#   14      TCP forward overlap

preprocessor stream4: disable_evasion_alerts

# tcp stream reassembly directive
# no arguments loads the default configuration
#   Only reassemble the client,
#   Only reassemble the default list of ports (See below),
#   Give alerts for "bad" streams
#
# Available options (comma delimited):
#   clientonly - reassemble traffic for the client side of a connection
#                only
#   serveronly - reassemble traffic for the server side of a connection
#                only
#   both - reassemble both sides of a session
#   noalerts - turn off alerts from the stream reassembly stage of stream4
#   ports [list] - use the space separated list of ports in [list], "all"
#                  will turn on reassembly for all ports, "default" will
#                  turn on reassembly for ports 21, 23, 25, 42, 53, 80,
```

```
#                  110, 111, 135, 136, 137, 139, 143, 445, 513, 1433, 1521,
#                  and 3306
#   favor_old - favor an old segment (based on sequence number) over a new
#               one. This is the default.
#   favor_new - favor an new segment (based on sequence number) over an old
#               one.
#   flush_behavior [mode] -
#           default      - use old static flushpoints (default)
#           large_window - use new larger static flushpoints
#           random       - use random flushpoints defined by flush_base,
#                          flush_seed and flush_range
#   flush_base [number] - lowest allowed random flushpoint (512 by default)
#   flush_range [number] - number is the space within which random
#                          flushpoints are generated (default 1213)
#   flush_seed [number] - seed for the random number generator, defaults to
#                         Snort PID + time
#
# Using the default random flushpoints, the smallest flushpoint is 512,
# and the largest is 1725 bytes.
preprocessor stream4_reassemble

# Performance Statistics
# ----------------------
# Documentation for this is provided in the Snort Manual. You should read
# it. It is included in the release distribution as doc/snort_manual.pdf
#
# preprocessor perfmonitor: time 300 file /var/snort/snort.stats pktcnt
10000

# http_inspect: normalize and detect HTTP traffic and protocol anomalies
#
# lots of options available here. See doc/README.http_inspect.
# unicode.map should be wherever your snort.conf lives, or given
# a full path to where snort can find it.
preprocessor http_inspect: global \
    iis_unicode_map unicode.map 1252

preprocessor http_inspect_server: server default \
    profile all ports { 80 8080 8180 } oversize_dir_length 500

#
#  Example unqiue server configuration
#
#preprocessor http_inspect_server: server 1.1.1.1 \
#    ports { 80 3128 8080 } \
#    flow_depth 0 \
#    ascii no \
#    double_decode yes \
#    non_rfc_char { 0x00 } \
#    chunk_length 500000 \
#    non_strict \
#    oversize_dir_length 300 \
#    no_alerts


# rpc_decode: normalize RPC traffic
# ---------------------------------
# RPC may be sent in alternate encodings besides the usual 4-byte encoding
# that is used by default. This plugin takes the port numbers that RPC
# services are running on as arguments - it is assumed that the given ports
# are actually running this type of service. If not, change the ports or
```

```
# turn it off.
# The RPC decode preprocessor uses generator ID 106
#
# arguments: space separated list
# alert_fragments - alert on any rpc fragmented TCP data
# no_alert_multiple_requests - don't alert when >1 rpc query is in a packet
# no_alert_large_fragments - don't alert when the fragmented
#                            sizes exceed the current packet size
# no_alert_incomplete - don't alert when a single segment
#                       exceeds the current packet size

preprocessor rpc_decode: 111 32771

# bo: Back Orifice detector
# -------------------------
# Detects Back Orifice traffic on the network.  Takes no arguments in 2.0.
#
# The Back Orifice detector uses Generator ID 105 and uses the
# following SIDS for that GID:
#  SID     Event description
# -----   -------------------
#   1       Back Orifice traffic detected

preprocessor bo

# telnet_decode: Telnet negotiation string normalizer
# ---------------------------------------------------
# This preprocessor "normalizes" telnet negotiation strings from telnet and
# ftp traffic. It works in much the same way as the http_decode
# preprocessor, searching for traffic that breaks up the normal data stream
# of a protocol and replacing it with a normalized representation of that
# traffic so that the "content" pattern matching keyword can work without
# requiring modifications. This preprocessor requires no arguments.
# Portscan uses Generator ID 109 and does not generate any SID currently.

preprocessor telnet_decode

# sfPortscan
# ----------
# Portscan detection module.  Detects various types of portscans and
# portsweeps.  For more information on detection philosophy, alert types,
# and detailed portscan information, please refer to the README.sfportscan.
#
# -configuration options-
#     proto { tcp udp icmp ip_proto all }
#    The arguments to the proto option are the types of protocol scans that
#    the user wants to detect.  Arguments should be separated by spaces and
#    not commas.
#  scan_type { portscan portsweep decoy_portscan distributed_portscan all }
#       The arguments to the scan_type option are the scan types that the
#       user wants to detect. Arguments should be separated by spaces and
#       not commas.
#     sense_level { low|medium|high }
#       There is only one argument to this option and it is the level of
#       sensitivity in which to detect portscans.  The 'low' sensitivity
#       detects scans by the common method of looking for response errors,
#       such as TCP RSTs or ICMP unreachables.  This level requires the
#       least tuning.  The 'medium' sensitivity level detects portscans and
#       filtered portscans (portscans that receive no response).  This
#       sensitivity level usually requires tuning out scan events from
#       NATed IPs, DNS cache servers, etc. The 'high' sensitivity level has
```

```
#       lower thresholds for portscan detection and a longer time window
#       than the 'medium' sensitivity level.  Requires more tuning and may
#       be noisy on very active networks. However, this sensitivity levels
#       catches the most scans.
#     memcap { positive integer }
#       The maximum number of bytes to allocate for portscan detection. The
#       higher this number the more nodes that can be tracked.
#     logfile { filename }
#       This option specifies the file to log portscan and detailed
#       portscan values to. If there is not a leading /, then snort logs to
#       the configured log directory. Refer to README.sfportscan for
#       details on the logged values in the logfile.
#     watch_ip { Snort IP List }
#     ignore_scanners { Snort IP List }
#     ignore_scanned { Snort IP List }
#       These options take a snort IP list as the argument.  The 'watch_ip'
#       option specifies the IP(s) to watch for portscan.  The
#       'ignore_scanners' option specifies the IP(s) to ignore as scanners.
#       Note that these hosts are still watched as scanned hosts.  The
#       'ignore_scanners' option is used to tune alerts from very active
#       hosts such as NAT, nessus hosts, etc.  The 'ignore_scanned' option
#       specifies the IP(s) to ignore as scanned hosts. Note that these
#       hosts are still watched as scanner hosts. The 'ignore_scanned'
#       option is used to tune alerts from very active hosts such as syslog
#       servers, etc.
#
preprocessor sfportscan: proto  { all } \
                         memcap { 10000000 } \
                         sense_level { low }

# arpspoof
#--------------------------------------
# Experimental ARP detection code from Jeff Nathan, detects ARP attacks,
# unicast ARP requests, and specific ARP mapping monitoring. To make use of
# this preprocessor you must specify the IP and hardware address of hosts
# on the same layer 2 segment as you. Specify one host IP MAC combo per
# line. Also takes a "-unicast" option to turn on unicast ARP request
# detection. Arpspoof uses Generator ID 112 and uses the following SIDS for
# that GID:

#  SID     Event description
# -----   ------------------
#   1       Unicast ARP request
#   2       Etherframe ARP mismatch (src)
#   3       Etherframe ARP mismatch (dst)
#   4       ARP cache overwrite attack

#preprocessor arpspoof
#preprocessor arpspoof_detect_host: 192.168.40.1 f0:0f:00:f0:0f:00

# X-Link2State mini-preprocessor
# ------------------------------
# This preprocessor will catch the X-Link2State vulnerability
# (www.microsoft.com/technet/security/bulletin/MS05-021.mspx).
#
# Format:
# preprocessor xlink2state: ports { <port> [<port> <...>] } [drop]
#
# "drop" will drop the attack if in Inline-mode.

#  SID     Event description
```

```
# -----   -------------------
#   1      X-Link2State length greater than 1024

preprocessor xlink2state: ports { 25 691 }

##################################################################
# Step #3: Configure output plugins
#
# Uncomment and configure the output plugins you decide to use.  General
# configuration for output plugins is of the form:
#
# output <name_of_plugin>: <configuration_options>
#
# alert_syslog: log alerts to syslog
# ----------------------------------
# Use one or more syslog facilities as arguments. Win32 can also optionally
# specify a particular hostname/port.  Under Win32, the default hostname is
# '127.0.0.1', and the default port is 514.
#
# [Unix flavours should use this format...]
# output alert_syslog: LOG_AUTH LOG_ALERT
#
# [Win32 can use any of these formats...]
# output alert_syslog: LOG_AUTH LOG_ALERT
# output alert_syslog: host=hostname, LOG_AUTH LOG_ALERT
# output alert_syslog: host=hostname:port, LOG_AUTH LOG_ALERT

# log_tcpdump: log packets in binary tcpdump format
# -------------------------------------------------
# The only argument is the output file name.
#
# output log_tcpdump: tcpdump.log

# database: log to a variety of databases
# ---------------------------------------
# See the README.database file for more information about configuring
# and using this plugin.
#
# output database: log, mysql, user=root password=test dbname=db
#                  host=localhost
# output database: alert, postgresql, user=snort dbname=snort
# output database: log, odbc, user=snort dbname=snort
# output database: log, mssql, dbname=snort user=snort password=test
# output database: log, oracle, dbname=snort user=snort password=test

# unified: Snort unified binary format alerting and logging
# -------------------------------------------------------------
# The unified output plugin provides two new formats for logging and
# generating alerts from Snort, the "unified" format. The unified format is
# a straight binary format for logging data out of Snort that is designed
# to be fast and efficient. Used with barnyard (the new alert/log
# processor), most of the overhead for logging and alerting to various slow
# storage mechanisms such as databases or the network can now be avoided.
#
# Check out the spo_unified.h file for the data formats.
#
# Two arguments are supported.
#    filename - base filename to write to (current time_t is appended)
#    limit    - maximum size of spool file in MB (default: 128)
#
# output alert_unified: filename snort.alert, limit 128
```

```
# output log_unified: filename snort.log, limit 128


# prelude: log to the Prelude Hybrid IDS system
# ------------------------------------------------
#
# output prelude: profile=snort
# profile = Name of the Prelude profile to use (default is snort).
# config  = Optional name of a specific prelude configuration file to use
#           for snort.
#
# Snort priority to IDMEF severity mappings:
# high < medium < low < info
#
# info   = 4
# low    = 3
# medium = 2
# high   = anything below medium
#
# These are the default mapped from classification.config.
#
# output alert_prelude


# You can optionally define new rule types and associate one or more output
# plugins specifically to that type.
#
# This example will create a type that will log to just tcpdump.
# ruletype suspicious
# {
#   type log
#   output log_tcpdump: suspicious.log
# }
#
# EXAMPLE RULE FOR SUSPICIOUS RULETYPE:
# suspicious tcp $HOME_NET any -> $HOME_NET 6667 (msg:"Internal IRC
# Server";)
#
# This example will create a rule type that will log to syslog and a mysql
# database:
# ruletype redalert
# {
#   type alert
#   output alert_syslog: LOG_AUTH LOG_ALERT
#   output database: log, mysql, user=snort dbname=snort host=localhost
# }
#
# EXAMPLE RULE FOR REDALERT RULETYPE:
# redalert tcp $HOME_NET any -> $EXTERNAL_NET 31337 \
#   (msg:"Someone is being LEET"; flags:A+;)
#
# Include classification & priority settings
# Note for Windows users:  You are advised to make this an absolute path,
# such as:  c:\snort\etc\classification.config
#

include classification.config

#
# Include reference systems
# Note for Windows users:  You are advised to make this an absolute path,
```

```
# such as:  c:\snort\etc\reference.config
#

include reference.config

##################################################################
# Step #4: Configure snort with config statements
#
# See the snort manual for a full set of configuration references
#
# config flowbits_size: 64
#
# New global ignore_ports config option from Andy Mullican
#
# config ignore_ports: <tcp|udp> <list of ports separated by whitespace>
# config ignore_ports: tcp 21 6667:6671 1356
# config ignore_ports: udp 1:17 53


##################################################################
# Step #5: Customize your rule set
#
# Up to date snort rules are available at http://www.snort.org
#
# The snort web site has documentation about how to write your own custom
# snort rules.

#========================================
# Include all relevant rulesets here
#
# The following rulesets are disabled by default:
#
#   web-attacks, backdoor, shellcode, policy, porn, info, icmp-info, virus,
#   chat, multimedia, and p2p
#
# These rules are either site policy specific or require tuning in order to
# not generate false positive alerts in most enviornments.
#
# Please read the specific include file for more information and
# README.alert_order for how rule ordering affects how alerts are
# triggered.
#========================================

include $RULE_PATH/local.rules
include $RULE_PATH/bad-traffic.rules
include $RULE_PATH/exploit.rules
include $RULE_PATH/scan.rules
include $RULE_PATH/finger.rules
include $RULE_PATH/ftp.rules
include $RULE_PATH/telnet.rules
include $RULE_PATH/rpc.rules
include $RULE_PATH/rservices.rules
include $RULE_PATH/dos.rules
include $RULE_PATH/ddos.rules
include $RULE_PATH/dns.rules
include $RULE_PATH/tftp.rules

include $RULE_PATH/web-cgi.rules
include $RULE_PATH/web-coldfusion.rules
include $RULE_PATH/web-iis.rules
include $RULE_PATH/web-frontpage.rules
```

```
include $RULE_PATH/web-misc.rules
include $RULE_PATH/web-client.rules
include $RULE_PATH/web-php.rules

include $RULE_PATH/sql.rules
include $RULE_PATH/x11.rules
include $RULE_PATH/icmp.rules
include $RULE_PATH/netbios.rules
include $RULE_PATH/misc.rules
include $RULE_PATH/attack-responses.rules
include $RULE_PATH/oracle.rules
include $RULE_PATH/mysql.rules
include $RULE_PATH/snmp.rules

include $RULE_PATH/smtp.rules
include $RULE_PATH/imap.rules
include $RULE_PATH/pop2.rules
include $RULE_PATH/pop3.rules

include $RULE_PATH/nntp.rules
include $RULE_PATH/other-ids.rules
# include $RULE_PATH/web-attacks.rules
# include $RULE_PATH/backdoor.rules
# include $RULE_PATH/shellcode.rules
# include $RULE_PATH/policy.rules
# include $RULE_PATH/porn.rules
# include $RULE_PATH/info.rules
# include $RULE_PATH/icmp-info.rules
 include $RULE_PATH/virus.rules
# include $RULE_PATH/chat.rules
# include $RULE_PATH/multimedia.rules
# include $RULE_PATH/p2p.rules
include $RULE_PATH/experimental.rules

# Include any thresholding or suppression commands. See threshold.conf in
# the <snort src>/etc directory for details. Commands don't necessarily
# need to be contained in this conf, but a separate conf makes it easier to
# maintain them. Note for Windows users:  You are advised to make this an
# absolute path, such as:  c:\snort\etc\threshold.conf
# Uncomment if needed.
# include threshold.conf
```