UNIVERSIDADE FEDERAL DA PARAÍBA

CENTRO DE INFORMÁTICA

PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

ESTRATÉGIAS PARA TRATAMENTO DE ATAQUES DE NEGAÇÃO DE SERVIÇO NA CAMADA DE APLICAÇÃO EM REDES IP

YURI GIL DANTAS

ORIENTADOR: PROF. DR. VIVEK NIGAM
CO-ORIENTADOR: PROF. DR. IGUATEMI E. DA FONSECA

João Pessoa – PB

Julho/2015

UNIVERSIDADE FEDERAL DA PARAÍBA

CENTRO DE INFORMÁTICA

PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

ESTRATÉGIAS PARA TRATAMENTO DE ATAQUES DE NEGAÇÃO DE SERVIÇO NA CAMADA DE APLICAÇÃO EM REDES IP

YURI GIL DANTAS

Dissertação apresentada ao Programa de Pós-Graduação em Informática da Universidade Federal da Paraíba, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação,
Área de concentração: Segurança da Informação
Orientador: Prof. Dr. Vivek Nigam

João Pessoa – PB

Julho/2015

Dedico este trabalho a minha tia, Maria Francileide Gomes, que infelizmente não viveu tempo suficiente para acompanhar essa jornada.

AGRADECIMENTOS

Aos meus pais por estarem sempre ao meu lado.

Aos meus irmãos pelas desavenças, amizade e carinho.

Ao meu tio, Wellington Gomes Dantas, pelos conselhos e por ser um exemplo a ser seguido.

Aos amigos Arnaldo Gualberto, Camilo Cabral, Glauco de Sousa, Hugo Neves, José Ivan e Raoni Azeredo que, ao longo da graduação e mestrado, demonstraram união, companheirismo e amizade.

A equipe do Grupo de Trabalho Ambiente Computacional para Tratamento de Incidentes com Negação de Serviço do LAR/UFPB pela dedicação no projeto e cooperação neste trabalho.

Ao tutor do PET Computação, o professor Dr. Leonardo Vidal Batista, por seus ensinamentos durante mais de três anos como PETiano.

Ao PET.Com pelas oportunidades e experiência adquirida.

Aos meus orientadores Dr. Vivek Nigam e Dr. Iguatemi E. Fonseca, por me orientarem e contribuírem para a minha formação acadêmica.

A todos que, de alguma forma, contribuíram para a concretização deste objetivo.

"As pessoas têm medo das mudanças, eu tenho medo que as coisas nunca mudem".

Chico Buarque

RESUMO

Ataques de Negação de Serviço Distribuídos (Distributed Denial of Service - DDoS) estão entre os ataques mais perigosos na Internet. As abordagens desses ataques vêm mudando nos últimos anos, ou seja, os ataques DDoS mais recentes não têm sido realizados na camada de transporte e sim na camada de aplicação. A principal diferença é que, nesse último, um atacante pode direcionar o ataque para uma aplicação específica do servidor, gerando menos tráfego na rede e tornando-se mais difícil de detectar. Tais ataques exploram algumas peculiaridades nos protocolos utilizados na camada de aplicação. Este trabalho propõe *SeVen*, um mecanismo de defesa probabilístico para mitigar ataques DDoS na camada de aplicação, baseada em *Adaptive Selective Verification* (ASV), um mecanismo de defesa para ataques DDoS na camada de transporte. Foram utilizadas duas abordagens para validar o *SeVen*: 1) Simulação: Todo o mecanismo de defesa foi formalizado na ferramenta computacional, baseada em lógica de reescrita, chamada *Maude* e simulado usando um modelo estatístico (*PVeStA*). 2) Experimentos na rede: Análise da eficiência do *SeVen*, implementado em C++, em um experimento real na rede. Em particular, foram investigados três ataques direcionados ao Protocolo HTTP: GET FLOOD, Slowloris e o POST. Nesses ataques, apesar de terem perfis diferentes, o *SeVen* obteve um elevado índice de disponibilidade.

Palavras-chave: Ataques de Negação de Serviço, Camada de Aplicação, Mecanismo de Defesa

ABSTRACT

Distributed Denial of Service (DDoS) attacks remain among the most dangerous and noticeable attacks on the Internet. Differently from previous attacks, many recent DDoS attacks have not been carried out over the Transport Layer, but over the Application Layer. The main difference is that in the latter, an attacker can target a particular application of the server, while leaving the others applications still available, thus generating less traffic and being harder to detected. Such attacks are possible by exploiting application layer protocols used by the target application. This work proposes a novel defense, called SeVen, for Application Layer DDoS attacks (ADDoS) based on the Adaptive Selective Verification (ASV) defense used for Transport Layer DDoS attacks. We used two approches to validate the *SeVen*: 1) Simulation: The entire defense mechanism was formalized in Maude tool and simulated using the statistical model checker (PVeStA). 2) Real scenario experiments: Analysis of efficiency *SeVen*, implemented in C++, in a real experiment on the network. We investigate the resilience for mitigating three attacks using the HTTP protocol: HTTP-POST, Slowloris, and HTTP-GET. The defence is effective, with high levels of availability, for all three types of attacks, despite having different attack profiles, and even for a relatively large number of attackers.

Keywords: Denial of Service, Application Layer, Defense Mechanism

LISTA DE FIGURAS

LISTA DE TABELAS

GLOSSÁRIO

ADDoS – *Application Layer DDoS attacks*

ASV – *Adaptive Selective Verification*

DDoS – *Distributed Denial of Service*

DoS – *Denial of Service*

HTTP – *Hypertext Transfer Protocol*

PVeStA – *A Parallel Statistical Model Checking and Quantitative Analysis Tool*

SeVen – *Selective Verification in Application Layer*

SUMÁRIO

Capítulo 1

INTRODUÇÃO

Desde o surgimento da internet, Ataques de Negação de Serviço Distribuídos (DDoS) têm sido um grande problema para os administradores de redes. Vários desses ataques podem ser facilmente efetuados, acarretando na indisponibilidade de serviços. A principal técnica utilizada por atacantes é denominada de *flooding* (inundação), onde o objetivo é consumir recursos computacionais da vítima, enviando uma grande quantidade de pacotes com a finalidade de sobrecarregá-lo, provocando sua indisponibilidade para usuários legítimos. É importante frisar que esse ataque não se trata de uma invasão ao sistema e sim, a uma indisponibilidade do mesmo.

Nos últimos anos, um grupo chamado "*Anonymous*" tem realizado ataques de *flooding* a várias grandes empresas, como MasterCard, Paypal, Visa e PostFinance [3]. Esses ataques afetaram 9 dos maiores bancos americanos: Bank of America, Citigroup, Wells Fargo, U.S. Bancorp, PNC, Capital One, Fifth Third Bank, BB&T e HSBC. Eles ainda têm sofrido ataques de um grupo ativista, chamado "Izz ad-Din al-Qassam Cyber Fighters" [4], afetando a disponibilidade de seus serviços online. Em 2013, um desentendimento envolvendo o grupo Spamhaus e a empresa Cyberbunker acarretou em um DDoS gerando um tráfego inútil de 300 Gbps [5], um dos maiores ataques da história.

Tradicionalmente, ataques DDoS acontecem na camada de transporte do modelo TCP/IP [6]. No entanto, uma versão mais sofisticada desses ataques, agora na camada de aplicação, tem sido utilizada com o objetivo de afetar uma aplicação específica (i.e., DNS, HTTP, SIP) do servidor. Isso significa que o número de atacantes e tráfego necessários para executar esse ataque é bem menor do que os tradicionais executados na camada de transporte e, portanto mais difíceis de detectar, tornando-o mais eficiente. Esse tipo de abordagem vem crescendo nos últimos anos [3] devido ao grande desenvolvimento de ferramentas automatizadas, as quais possibilitam que até mesmo usuários leigos consigam executar ataques DDoS.

Este trabalho se concentra nos ataques direcionados ao protocolo HTTP (*Hypertext Transfer Protocol*) da camada de aplicação, como GET FLOOD, Slowloris e o POST. A seguir são apresentadas algumas razões que influenciaram nessa escolha:

1) HTTP é o maior alvo para ataques de negação de serviço na camada de aplicação (ADDoS): Em 2013, 37.2% de todos os ataques DDoS foram direcionados ao protocolo HTTP, sendo o segundo ataque mais realizado, ficando apenas atrás do ataque TCP SYN Flood realizado na camada de transporte que somaram 38.7% de todos os ataques DDoS [7]. Além disso, o grupo "*Anonymous*" populariza ADDoS através do desenvolvimento de ferramentas [8].

2) Apesar de ser uma grande ameaça às aplicações, poucas defesas eficientes foram propostas até o momento. O principal motivo que dificulta combater tais ataques é que o tráfego gerado por esses atacantes é bem similar ao de um usuário legítimo [9].

3) Esses ataques exploram diferentes campos do protocolo HTTP, tendo um comportamento distinto para cada ataque. Por isso, eles acabam sendo um bom teste para verificar o quão resistente é uma defesa para mitigar diferentes estratégias de ataques.

Diversos modelos de defesa para ataques DDoS têm sido implementados ao longo dos anos com o objetivo de bloquear parcialmente ou completamente um DDoS. Esses modelos estão divididos em duas partes: Identificação (momento em que o ataque é detectado) e Resposta (momento que o atacante é desconectado da aplicação). No entanto, tanto a Identificação como a Resposta, são ações consideradas complexas, principalmente pelo grande número de máquinas infectadas utilizadas em um ataque distribuído, tornando confuso diferenciar o tráfego legítimo de um ataque. Sendo assim, caso não sejam desenvolvidos corretamente, esses mecanismos tornam-se ineficientes, apresentando um elevado índice de falsos positivos.

Algumas abordagens podem ser utilizadas para evitar esse tipo de erro e garantir ao máximo a eficiência de um serviço. Uma delas são os testes empíricos, que é um método que envolve a construção de um protótipo ou um sistema completo, onde são realizados vários testes. Sua principal desvantagem é que essa técnica pode contribuir na observação de erros, mas não para a prova de corretude, um requisito essencial em aplicações que envolvem segurança. Os métodos formais, por outro lado, são baseados em técnicas matemáticas que oferecem uma estrutura onde sistemas podem ser especificados, desenvolvidos e analisados de maneira sistemática. Esses métodos vêm obtendo resultados satisfatórios em suas análises na área de segurança, por exemplo, falhas de sistemas de segurança conhecidos, como o Kerberos [10], Needham e Schroeder [11] que foram descobertos através de Métodos Formais.

Portanto, é de extrema importância utilizar métodos robustos que proporcionam uma aná-

lise rigorosa sobre mecanismos de segurança, encontrando suas falhas e determinando suas correções. Este trabalho propõe *Selective Verification in Application Layer (SeVen)*, um mecanismo de defesa probabilístico para mitigar ataques DDoS na camada de aplicação (ADDoS), baseado em *Adaptive Selective Verification* (ASV). Foram utilizadas duas abordagens para validar a defesa: 1) Simulação: Todo o mecanismo de defesa foi formalizado usando a ferramenta *Maude* e simulado usando um modelo estatístico (*PVeStA*). 2) Experimentos na rede: Análise da eficiência de *SeVen*, implementado em C++, em um experimento real na rede. Em ambas abordagens, o *SeVen* obteve um elevado índice de disponibilidade e um tempo de resposta (TTS) satisfatório. Por exemplo, a seção 5.4.1 do Capítulo 5 mostra que o *SeVen* consegue mitigar um ataque Slowloris elevando o índice de disponibilidade de 0%, em um cenário sem defesa, para aproximadamente 95%.

1.1 Objetivos

O objetivo geral deste trabalho é propor uma defesa para ataques de negação de serviço na camada de aplicação, utilizando Métodos Formais com a finalidade de especificar formalmente a defesa e implementar um protótipo para realizar experimentos na rede. Para isso, o trabalho apresenta os seguintes objetivos específicos:

Objetivo 1: Formalizar ADDoS

Especificar formalmente três diferentes ataques: GET FLOOD, Slowloris e POST.

Objetivo 2: Propor uma nova defesa para ADDoS

Apresentar uma nova defesa para ataques DDoS, chamada *SeVen*, baseado no ASV [12]. Como o ASV foi projetado para atenuar ataques DDoS na camada de transporte, ele assume uma simples comunicação *stateless* (requisições são totalmente independente uma da outra) entre clientes e servidores. No entanto, essa abordagem não é suficiente para mitigar ADDoS, pois os protocolos utilizados nesses ataques, como HTTP, têm uma noção de estado (*stateful*). Portanto, *SeVen* propõe estender o ASV, incorporando à defesa uma noção de estados necessárias para mitigar esses tipos de ataques na camada de aplicação.

Objetivo 3: Formalizar e validar o *SeVen*:

Formalizar a defesa *SeVen* em *Maude* e validar usando *PVeStA* [13], um verificador automático probabilístico. Além disso, implementar um protótipo da defesa em C++ e realizar experimentos reais na rede.

1.2 Trabalhos Relacionados

Nesta seção são descritos alguns trabalhos relacionados com a estratégia proposta na dissertação.

1.2.1 Yi Xie e Shun-Zheng Yu (2009)

Yi Xie e Shun-Zheng Yu [14] propõem um algoritmo de aprendizagem, utilizando Cadeia de Markov, para classificar um tráfego DDoS como normal ou anômalo, ou seja, o intuito do trabalho é encontrar um método eficaz para identificar se o aumento no tráfego em uma aplicação WEB é causada por atacantes ou por usuários legítimos.

Na prática, é realizado um treinamento no modelo em dois cenários distintos: 1) O modelo recebe, como parâmetro, o tráfego de clientes legítimos da aplicação, variando o número de sessões TCP. 2) O modelo é atualizado com um volume maior de requisições (i.e. tráfego do atacante e do cliente). Se alguma anormalidade no tráfego for encontrada, o sistema de defesa é acionado, criando uma fila de prioridade de atendimento para o tráfego classificado como normal.

Esse tipo de mecanismo é eficiente para combater ataques, como o GET Flood, no qual o tráfego gerado pelos atacantes é diferente dos clientes legítimos. No entanto, essas defesas são ineficazes em mitigar ataques mais espertos, como o Slowloris [15] ou POST [16], pois o tráfego é bem similar ao de um cliente legítimo.

1.2.2 Traffic Analysis Defense – TAD (2013)

Leandro Cavalcanti [17] propõe um modelo de detecção e bloqueio em tempo real de ataques de negação de serviço distribuído, GET Slowloris, contra servidores WEB, utilizando um sistema de detecção baseado em assinaturas de ataques. Essas assinaturas são baseadas nas regularidades do ataque GET Slowloris, pois várias ferramentas disponíveis para ADDoS, como o slowloris [15] e r-u-dead-yet [16], usam um tráfego padrão para efetuar o ataque. Por exemplo, ao executar o slowloris, a ferramenta envia pacotes periódicos com o tempo fixo e com o mesmo número de bytes.

Sendo assim, o TAD utiliza um conjunto de parâmetros para analisar o ataque, dentre eles o endereço IP de origem do ataque, tamanho e tempo relativo de chegada dos pacotes, porta de destino e etc. Com essas informações, para cada pacote, o módulo de análise consegue identificar o padrão do ataque, invocando assim o módulo de bloqueio.

Como mostrado em [17], TAD é capaz de mitigar ataques Slowloris gerado por um pequeno número de atacantes. No entanto, uma vez que o número de atacantes cresce esse mecanismo de defesa não garante um bom desempenho, como previsto em [18].

1.2.3 LSDDoS Defense (2014)

Mark Shtern et al. [19] apresenta uma arquitetura de defesa adaptativa para identificar e mitigar ADDoS. Essa defesa, nomeada de LSDDoS, utiliza o conceito de SDN para modificar, sob demanda, a arquitetura e os recursos computacionais da rede, como por exemplo, um servidor WEB. A arquitetura do LSDDoS possui alguns módulos, entre eles podemos destacar:

1) LSDDoS Detection Sensor – Seu objetivo é identificar possíveis ataques DDoS na camada de aplicação e verificar quais são os endereços IPs que estão realizando o ataque. Para isso, o módulo é separado em duas etapas: 1.1) Criar um modelo classificador e treiná-lo para identificar o ataque. Uma forma de treinamento pode ser realizada quando um servidor WEB não está sofrendo um ataque DoS, ou seja, o fluxo de entrada são as requisições de clientes legítimos. 1.2) Comparar o modelo com as métricas de medição, como a utilização do buffer da aplicação, CPU e disco rígido, tempo de resposta, taxa de transferência e etc. Dinamicamente esses valores são comparados e um limiar discrepância pode sinalizar um ataque. Caso uma anormalidade seja encontrada, o módulo tenta identificar qual usuário está realizando o ataque. Neste momento, todas as requisições são examinadas individualmente. Se houver muitos pedidos de um usuário em particular, esse usuário se torna suspeito e é redirecionado para outro módulo, chamado Shark Tank.

2) Shark Tank – O objetivo deste módulo é atuar como uma área restrita para o atacante, colocando-o sob vigilância. Ao receber o fluxo de dados, o Shark Tank absorve todo o tráfego do ataque DoS e o armazena como tráfego de treinamento, o qual pode ser utilizado para classificar futuros tráfegos anômalos. Outra característica desse módulo é sua capacidade de informar, aos outros módulos, quando o ataque é finalizado.

3) Automation Controller – Este módulo é o controlador da arquitetura de defesa e responsável por proteger as aplicações da rede. É ele quem, de fato, redireciona o tráfego, classificado como malicioso, do LSDDoS Detection Sensor para o Shark Tank. Além disso, ele é também responsável por remediar ou restauras as aplicações, ou seja, caso uma aplicação tenha seus recursos esgotados por um ataque DoS, uma estratégia seria criar uma nova instância daquela aplicação fazendo a cópia de todos seus eventos corretamente.

Apesar de apresentar uma possível estratégia para mitigar ADDoS, o trabalho é puramente

teórico, sem nenhum tipo de experimento. Sendo assim, um dos seus trabalhos futuros é mostrar a eficiência do LSDDoS.

1.2.4 Chuan Xu (2014)

Uma outra forma de consumir os recursos de uma aplicação WEB é enviar requisições custosas (i.e., *high-workload*), como, por exemplo, solicitações de páginas dinâmicas, banco de dados e etc. Esse ataques são denominados de Ataques de Negação de Serviço Assimétricos. Chuan Xu et al. [1] apresentam uma defesa para identificar e mitigar esses ataques analisando a sequência de requisições HTTP de usuários.

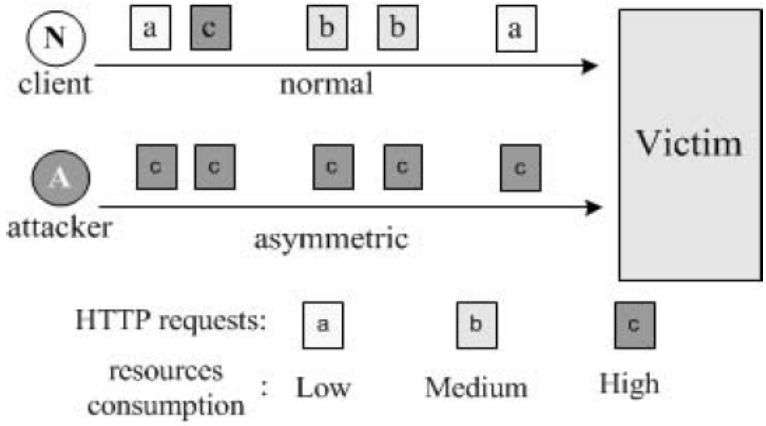


Figura 1.1: Exemplo de um Ataque Assimétrico [1]

Como pode ser visto na Figura 1.1, em um ataque assimétrico, o atacante adota uma sequência (c,c,c,c,c) de clicks *high-workload* com o objetivo de consumir rapidamente os recursos da vítima. Por outro lado, diferente do atacante, a sequência de solicitações do usuário (a,c,b,b,a) é considerada normal para uma aplicação em particular. Ao analisar a sequência de clicks das requisições, é realizado um cálculo para verificar a semelhança entre os clicks, caso a similaridade seja baixa, um alerta é acionado, sendo possível identificar imediatamente os endereços que estão realizando o ataque.

Como avaliado em [1], essa estratégia consegue identificar e mitigar ataques assimétricos. Entretanto, para verificar a sua robustez é necessário realizar mais experimentos, como, por exemplo, um atacante simulando os clicks de um usuário legítimo.

1.3 Estrutura de Dissertação

Este trabalho é composto por mais cinco capítulos, além da introdução. O Capítulo 2 discorre sobre os fundamentos necessários para o entendimento do trabalho. O Capítulo 3 especifica a metodologia proposta para mitigar ADDoS. O Capítulo 4 apresenta a especificação formal e os resultados obtidos na simulações, o Capítulo 5 a arquitetura do protótipo e os experimentos na rede e o Capitulo 6 apresenta as considerações finais. Ao final do trabalho, encontram-se as referências bibliográficas utilizadas para a elaboração do mesmo.

Capítulo 2

FUNDAMENTAÇÃO TEÓRICA

Neste capítulo serão destacados conceitos fundamentais para o entendimento do trabalho proposto: Ataques de Negação de Serviço, *Botnet*, Métodos Formais e alguns ataques direcionados ao protocolo HTTP.

2.1 Ataques de Negação de Serviço

Um Ataque de Negação de Serviço (DoS) consiste em uma tentativa de tornar um serviço ou aplicação indisponível para seus usuários legítimos. Para isso, um atacante utiliza algum tipo de estratégia para ocupar totalmente os recursos de uma aplicação, forçando as próximas requisições serem rejeitadas, como pode ser visto na Figura 2.1. Esse ataque é usualmente realizado de maneira distribuída (Ataque de Negação de Serviço Distribuído – DDoS), no qual um atacante, através um conjunto de máquinas infectadas (*botnet*), realiza um DDoS em larga escala.

Ataques de negação de serviço distribuídos sempre foram uma grande preocupação para os administradores de rede. Tradicionalmente, esses ataques são realizados na camada de transporte, através de um grande envio de requisições a um servidor, tornando-o indisponível para os usuários legítimos. Embora ainda perigosos, tais ataques podem ser mitigados pelas defesas [20] [21] [12]. A principal hipótese dessas defesas é que os ataques na camada de transporte seguem uma comunicação *stateless* syn-ack, gerando uma grande quantidade de requisições. Assim, administradores podem analisar o tráfego da rede e aplicar as providências necessárias.

Nos últimos anos, esses ataques foram substituídos por estratégias mais sofisticadas que exploram os protocolos da camada de aplicação. Um novo cenário, no qual o alvo pode ser uma única aplicação do servidor, por exemplo, um servidor WEB. Ao explorar os protocolos (i.e.,

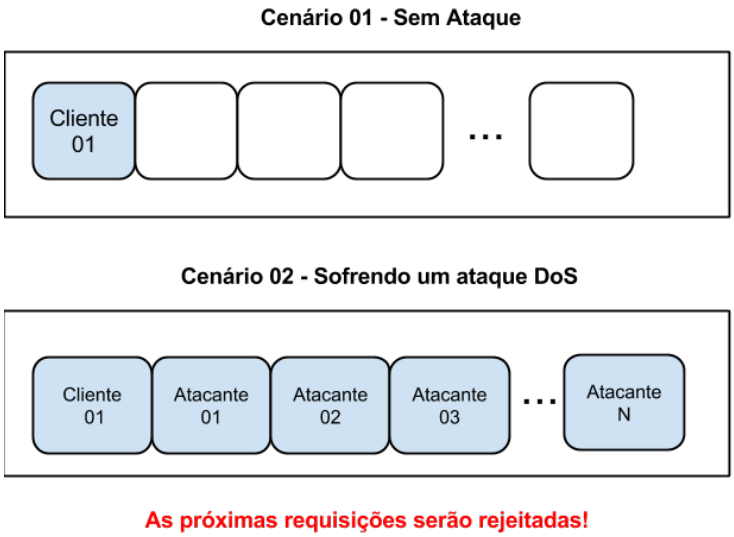


Figura 2.1: Comportamento do buffer de uma aplicação durante um DoS

HTTP, VoIP, DNS) da camada de aplicação, o atacante não precisa gerar uma grande quantidade de requisições, tornando as defesas existentes ineficazes [20]. De acordo com a documentação do Slowloris [15], uma ferramenta para a realização de ataques DoS na camada de aplicação, é possível negar serviço a clientes legítimos de uma aplicação WEB utilizando um número relativamente pequeno de atacantes. Por exemplo, os experimentos da seção 5.4.1 mostram que 250 atacantes, efetuando o ataque Slowloris, são suficientes para negar completamente o serviço de uma aplicação com um *buffer* de 200 posições. Essa eficiência é refletida em seu uso, pois mais de 60% de todos os ataques em 2013 foram realizados sobre a camada de aplicação [22]. A seguir é descrito como funciona uma *botnet* e os ataques utilizados neste trabalho.

2.1.1 Botnet

Uma *botnet* é formada por um conjunto de máquinas, usualmente denominada de *bots* ou zumbis, infectadas por algum tipo de *malware*. Onde os *bots* são controlados remotamente por um atacante, denominado de *botmaster*. A Figura 2.2 ilustra um ataque DDoS tradicional utilizando uma *botnet*. Os principais elementos dessa arquitetura de ataque são:

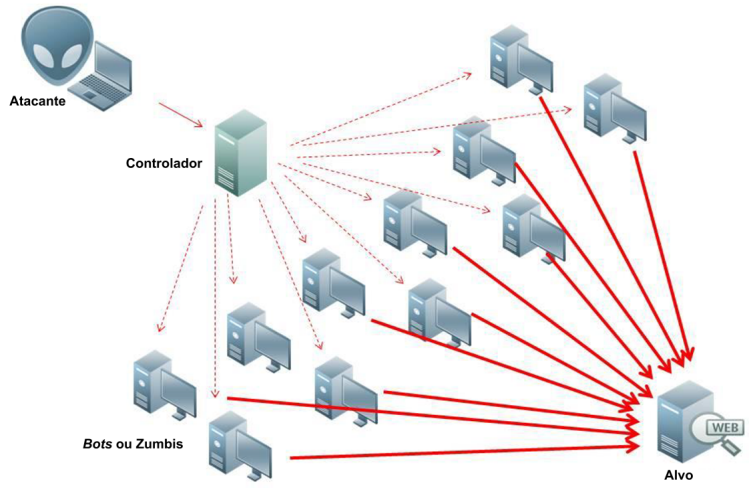


Figura 2.2: Esquema de uma botnet – modificado de [2]

1) **Atacante:** Usuário que conduz o ataque, denominado de *botmaster*.

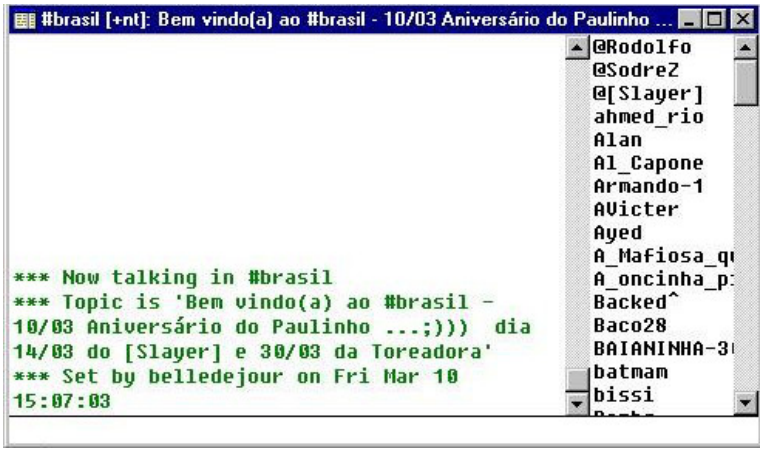
2) **Controlador (*Handler*):** É um serviço de comunicação utilizado na internet, onde o mais tradicional em ataques DDoS é o *IRC* [23]: Um protocolo da camada de aplicação que permite a troca de mensagens em forma de texto. Atacantes utilizam desses serviços para enviar comandos aos *bots*, ou seja, conduzir ataques em massa.

3) **Máquinas Infectadas (*Bots*):** Conjunto de máquinas infectadas utilizadas para enviar grandes volumes de mensagens para um ou mais destinos.

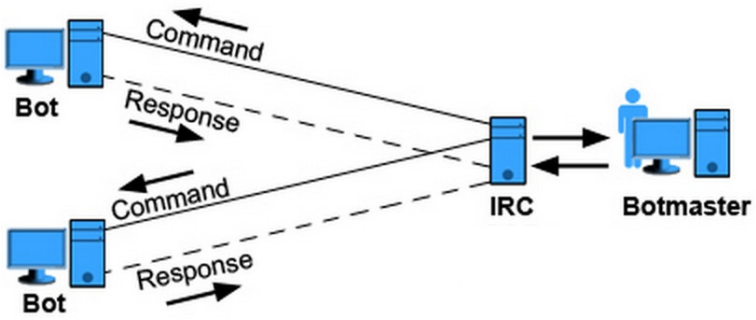
4) **Alvo (*Target*):** Vítima do ataque. Ex: Servidores WEB

A primeira geração de *botnets* utilizaram um servidor *Internet Relay Chat – IRC*, um protocolo da camada de aplicação que possui uma arquitetura cliente-servidor, como controlador. Ao estabelecer a conexão com um servidor IRC, o cliente tem acesso a diversos canais de comunicação. Geralmente, quando o *botmaster* possui uma *botnet* em um servidor IRC, o atacante cria um canal para adicionar todos os *bots*. Em seguida, ele deve se autenticar como *botmaster* no canal, tornando-se capaz de enviar comandos de ataques para todos os *bots* conectados, como ilustrado na Figura 2.3(b). Um exemplo de uma interface do mIRC [25], um cliente do protocolo IRC, é apresentado na Figura 2.3(a).

Pela facilidade de uso, simples controle e gerenciamento, o IRC é o controlador mais tradicional. No entanto, ele possui suas fragilidades, pois seu serviço é totalmente centralizado. Isto é, caso o servidor seja desconectado, toda a *botnet* também será e o ataque não terá sucesso. Por esse motivo, atacantes têm utilizado, como alternativa, o protocolo HTTP para realizar a troca mensagens com os *bots*. Neste cenário, caso uma conexão seja desconectada, apenas um *bot*



(a) Exemplo de um Canal IRC.



(b) Troca de mensagens em uma *Botnet* IRC. [24]

Figura 2.3: Protocolo IRC

não executará o comando, tornando o ataque mais confiável.

2.1.2 Ataques de Inundação (HTTP GET FLOOD)

Um Ataque de Inundação [20] se caracteriza por enviar um grande volume de requisições com o objetivo de inundar e tornar indisponível uma aplicação para seus usuários legítimos. Esses ataques são similares aos que ocorrem na camada de transporte, como pode ser visto na Figura 2.4, no entanto em vez de afetar o servidor completamente, o alvo é uma aplicação, como um servidor WEB.

No TCP SYN FLOOD, o atacante inicia o processo de estabelecimento de muitas conexões TCP enviando para vítima milhares de pacotes SYN. O servidor, seguindo o funcionamento normal do protocolo, aloca recursos para receber as novas conexões e devolve vários pacotes SYN-ACK e fica aguardando os pacotes ACK para finalizar a fase do *three-way handshake* de cada nova conexão. O problema é que este último pacote nunca chegará, e os recursos (processador, memória, etc.) permanecem alocados por um período de tempo, fazendo com

que a vítima entre em um processo de colapso. [17]. Como o alvo do TCP SYN FLOOD é um servidor com todas as suas aplicações, a quantidade de tráfego necessária para realizar esse ataque de negação de serviço é bem maior do que o HTTP GET FLOOD, pois os ataques na camada de aplicação têm como alvo uma aplicação específica de um servidor.

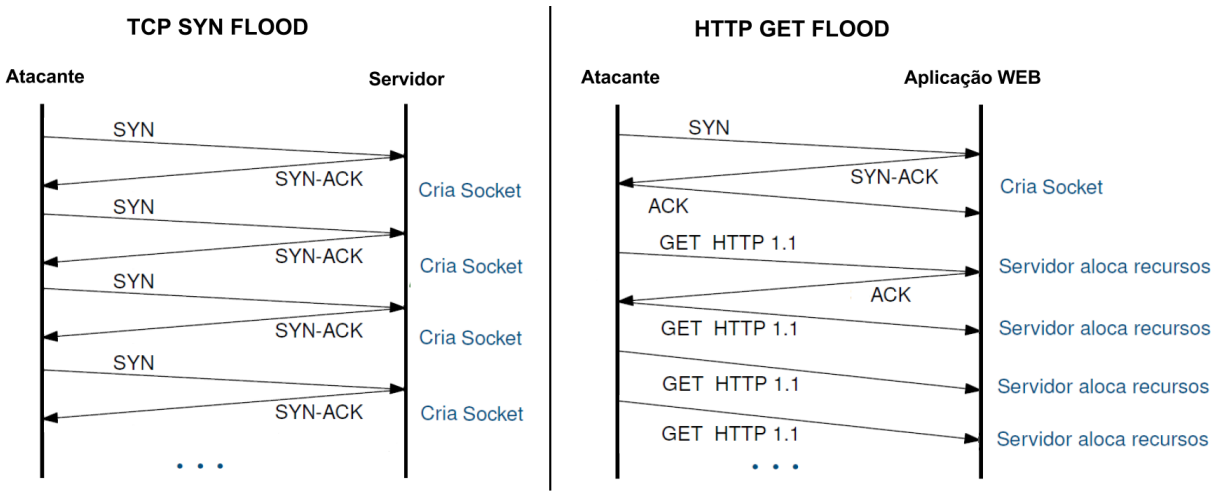


Figura 2.4: Comparação entre os ataques SYN FLOOD e GET FLOOD

No ataque HTTP GET FLOOD, ilustrado no lado direito da Figura 2.4, a aplicação recebe um grande número de requisições (GET HTTP 1.1), após realizar *three-way handshake* completo. Inicialmente, o atacante verifica se a aplicação está disponível, enviando uma solicitação GET HTTP. Se o atacante receber uma resposta (ACK) da aplicação, em seguida, ele envia periodicamente novas mensagens GET HTTP, sem esperar novas mensagens ACK, como especificado no autômato da Figura 2.5.

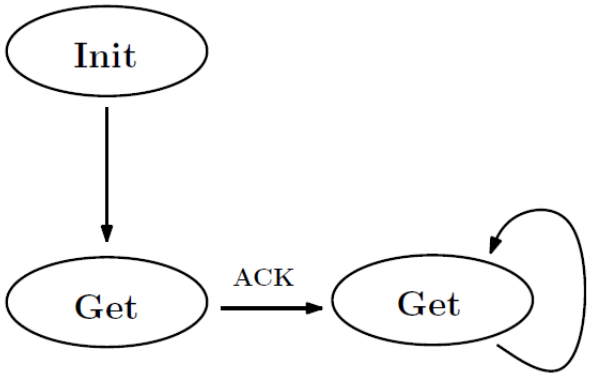


Figura 2.5: Autômato do ataque GET FLOOD

2.1.3 Ataques de Low-Rate

Enquanto os ataques de inundação são semelhantes aos executados na camada de transporte, os ataques de *Low-Rate* procuram explorar algumas peculiaridades nos protocolos utilizados na camada de aplicação. Seu objetivo é utilizar um número menor de atacantes e, consequentemente gerar menos tráfego na rede durante um ataque de negação de serviço. Neste trabalho é apresentado dois diferentes ataques *Low-Rate*: Slowloris e POST. Em particular, esses ataques exploram o *timeout* [1] de um servidor WEB. Valor que pode ser descoberto através de um *sniffer*, como, por exemplo o *Wireshark* [26].

2.1.3.1 Slowloris

Slowloris [15] é uma ferramenta desenvolvida pelo grupo *Anonymous* com a finalidade de executar ataques DoS na camada de aplicação. Ao utilizar a ferramenta Slowloris, múltiplas conexões TCP são solicitadas à um servidor WEB. Em cada conexão, um atacante envia uma requisição GET incompleta, ou seja, sem o \r\n no final da requisição. Em seguida, próximo ao *timeout*, ele envia novos cabeçalhos incompletos para manter a conexão ativa, pois a aplicação WEB não pode responder uma requisição até recebê-la completa. A medida que o ataque progride, o número de conexões aumenta e, eventualmente, consome todos os recursos da aplicação, tornando o servidor indisponível para responder as próximas requisições.

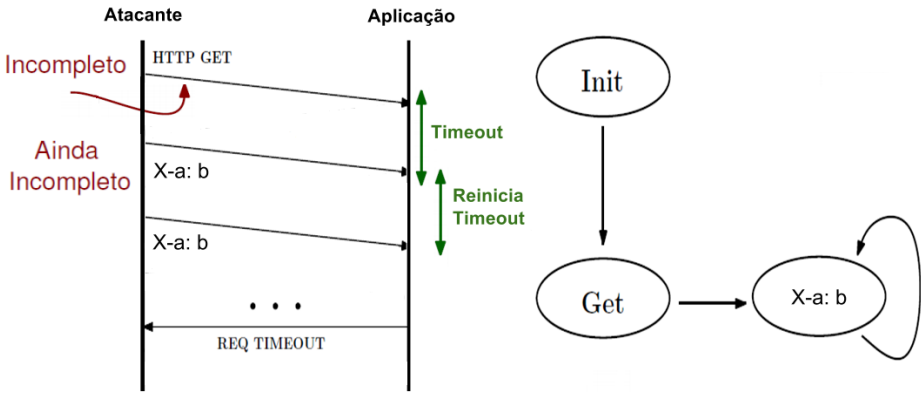


Figura 2.6: Ataque - Slowloris

O ataque Slowloris é ilustrado na Figura 2.6, onde um atacante envia uma mensagem GET incompleta, com o intuito que sua requisição seja alocada na memória da aplicação. Em seguida, envia uma mensagem incompleta qualquer, como, por exemplo "X-a: b", próximo ao *timeout*, com o objetivo de reiniciar o seu tempo de interação, permitindo o atacante permanecer

[1] *Timeout* é o tempo, em segundos, que uma requisição permanece no *buffer* de uma aplicação.

alocado no *buffer* da aplicação. Como exibido em [17] [20], um único atacante, seguindo essa estratégia, é capaz de realizar um ADDoS.

O autômato descrito na Figura 2.6, especifica os estados de um atacante realizando um ataque DoS Slowloris. Primeiro ele envia uma mensagem GET para a aplicação, após isso, periodicamente, o atacante envia mensagens "X-a: b".

2.1.3.2 HTTP POST

O método POST é um mecanismo geralmente utilizado em formulários de cadastro e registros de usuários em servidores WEB. Quando isso ocorre, os dados são enviados para aplicação em uma ou mais mensagens. O ataque ocorre justamente na manipulação dessas mensagens.

A sequência de mensagens de um ataque HTTP POST é ilustrada na Figura 2.7. Primeiro um atacante envia uma requisição POST com x *bytes* para a aplicação, onde x é relativamente grande. A aplicação, por sua vez, aguarda um *timeout* para receber todos os bytes da requisição para enviar uma resposta. No entanto, em vez do atacante enviar vários *bytes* na mesma mensagem, como clientes legítimos fazem, ele envia poucos *bytes* por mensagem, ocupando, por um tempo maior, os recursos da aplicação. Na Figura 2.7, o atacante envia um *byte* por mensagem (1/x) até que atinja o *timeout* ou complete os *bytes* de sua requisição. A máquina de estados finitos na figura 2.7 também descreve o ataque POST, onde o atacante envia uma mensagem POST e, em seguida, envia periodicamente os *bytes* do formulário.

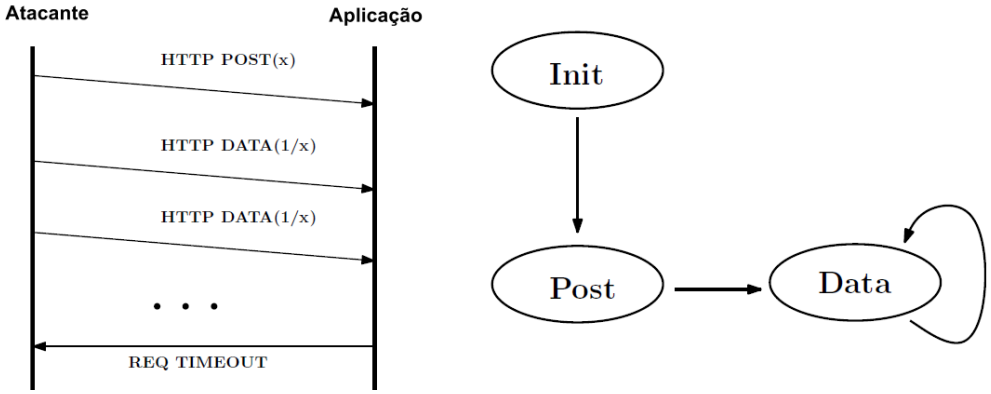


Figura 2.7: Ataque - HTTP POST

2.2 Métodos Formais

Nesta seção serão abordadas as técnicas utilizadas na especificação formal das simulações: Lógica de Reescrita, Linguagem *Maude* e *PVeStA*.

2.2.1 Lógica de Reescrita

Lógica de reescrita (Rewriting Logic, rwl) [27] é uma lógica de transição em que aspectos estáticos e dinâmicos de um sistema podem ser especificados. É também um framework lógico que pode representar diferentes lógicas, linguagens, formalismos operacionais e modelos computacionais [28] [29] [30]. Os aspectos dinâmicos são especificados na lógica de reescrita usando regras de reescritas condicionais e rotuladas e os aspectos estáticos são especificados usando uma lógica equacional. Este trabalho foca nos aspectos dinâmicos, aplicando a lógica de reescrita com a finalidade de formalizar mecanismos de defesa. Esse tipo de formalização oferece algumas vantagens:

- Nível de especificação bastante detalhado
- Prototipação rápida
- Depuração executando diretamente as especificações do mecanismo utilizado

As reescritas são compostas por triplas $(\sum, E, R)$, onde $(\sum, E)$ são estados com sintaxe do tipo especificado por $\sum$, e R é um conjunto de reescritas (condicionais ou não) da forma:

$$t \rightarrow t' \; if \; cond$$

onde t e t' são construídos a partir de $\sum$ e $cond$ é regra condicional, ou seja, ocorrerá o mapeamento do estado t para o t' caso aconteça o $match$ em t e $cond$ seja verdadeiro.

Lógica de Reescrita tem sido aplicada com sucesso nas análises formais em mecanismos de segurança, tais como os protocolos de autenticação de Kerberos [10] e Needham and Schroeder [11] e na análise baseada na assinatura de certificado digital em [31].

Para o melhor entendimento da lógica de reescrita, a seguir é formalizado um problema, retirado de [32], chamado *Blocks World*, onde o objetivo é implementar um algoritmo onde vários blocos de madeira são empilhados, através de um robô, uns sobre os outros ou sobre uma mesa. Uma simples amostra do *Blocks World* é ilustrada na Figura 2.8, onde na parte superior encontra-se o robô e sobre a mesa quatro blocos, denominados por A, D, C e B, respectivamente.

Após familiarizar-se com o problema, são descritos os predicados dos blocos, do robô e suas reescritas.

Predicados pertencentes aos blocos: Assumimos que os blocos possuem três predicados, dois com aridade um e outro com aridade dois:

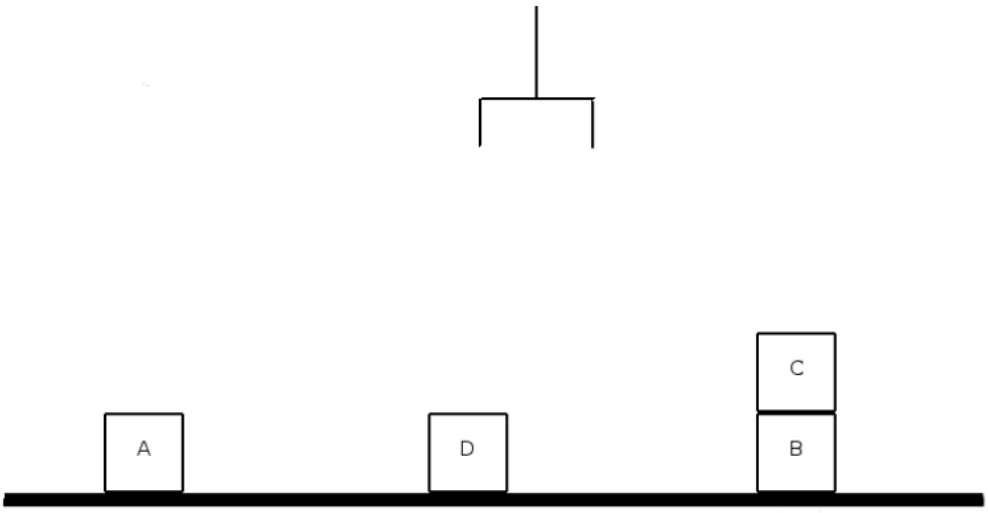


Figura 2.8: Exemplo do problema *Blocks World*

OnTable(A): O bloco A está sobre a mesa

Clear(A): O bloco A está livre

On(C,B): O bloco C está sobre o bloco B

Predicados pertencentes ao robô: Assumimos que o robô possui dois predicados, um unário e outro sem argumento:

Hold (X): O robô está segurando um bloco X

Empty(): O robô está livre

Reescritas do Blocks World: A seguir é especificado quatro estados e suas transições utilizando os predicados descritos:

PickUp: Robô pega o bloco que está sobre a mesa

Putdown: Robô solta o bloco sobre a mesa

UnStack: Robô desempilha um bloco que está sobre outro

Stack: Robô empilha um bloco sobre outro

Assim, temos:

PickUp: empty clear(X) ontable(X) $\rightarrow$ hold(X) .

Putdown: hold(X) → empty clear(X) ontable(X) .

UnStack: empty clear(X) on(X,Y) → hold(X) clear(Y) .

Stack: hold(X) clear(Y) → empty clear(X) on(X,Y)

Na primeira regra (PickUp), é verificado se o robô e o bloco estão livres e se está sobre a mesa, após os três predicados serem verdadeiros o robô pega o bloco X. A segunda reescrita é o inverso da primeira, o robô vai soltar um bloco X sobre a mesa. Na terceira (UnStack), a finalidade é desempilhar um bloco que está sobre outro, caso o robô esteja livre, o bloco X esteja sobre Y e não exista outro bloco sobre X, o robô pega X deixando o livre. A quarta e última reescrita, é o inverso da anterior, o intuito é empilhar um bloco X sobre outro Y, caso não exista outro bloco sobre Y, i.e. clear(Y).

2.2.2 Maude

Maude é uma ferramenta computacional baseada em lógica de reescrita que pode ser utilizada para especificar uma grande variedade de sistemas. Um sistema é formalmente descrito como um tipo algébrico por meio de uma equação [33].

Ao utilizar *Maude*, é possível definir novos tipos (*sort(s)*); subtipos (*subsorts*) que são subgrupos pertencentes ao seu respectivo tipo. Operadores (*op*) que atuam sobre esses tipos, inserindo seus argumentos e o valor de seu retorno. Esses operadores podem ter atributos como: associativo [*assoc*], comutativo [*comm*], identidade [*id*] e construtores [*ctor*]. Equações (*eq*) identificam termos que são construídos com esses operadores. E as variáveis (*var(s)*), onde cada variável pertence a um tipo (*sort*). [34] Por fim, a parte considerada a mais poderosa de *Maude*, a reescrita (*rl*, consiste de um conjunto de estados e transições, que são mapeamentos de um estado para outro). Tais especificações são introduzidas através de módulos, com as sintaxes *fmod* (módulo funcional), *mod* (módulo do sistema) e *omod* (módulo orientado à objetos).

Maude é atualmente desenvolvido no SRI International e na Universidade de Illinois, e viabiliza uma grande capacidade de checagem, podendo ser utilizada em diversos focos. Neste trabalho, *Maude* é utilizado para especificar defesas e ataques de negação de serviço, ambos apresentados no Capítulo 4.

2.2.3 PVeStA

PVeStA (*A Parallel Statistical Model Checking and Quantitative Analysis Tool*) é uma extensão e paralelização da ferramenta *VeSta*, com uma arquitetura cliente-servidor. Sua finalidade

é realizar a verificação de modelos estatísticos e análises quantitativas de sistemas probabilísticos. *PVeStA* implementa algoritmos nas quais suas amostras aleatórias são computadas em paralelo com a tarefa de realizar simulações Monte Carlo através de diversos elementos computacionais [13].

PVeStA pode ser utilizado para a verificação de modelos estatísticos e análises quantitativas das teorias de reescritas probabilísticas expressas em *Maude*. Esse é o motivo principal para sua utilização neste trabalho, pois com o uso dessa ferramenta é possível fazer com que as reescritas tenham um comportamento estocástico. Como foi explicado em 2.2.1, as reescritas possuem um comportamento determinístico, ou seja, caso ocorra um *match* do lado esquerdo da reescrita, haverá um mapeamento entre os estados das reescritas. Por outro lado, com o uso do *PVeStA*, as reescritas têm a seguinte forma:

$$t \rightarrow t' \; if \; cond \; with \; probability \; P$$

Onde do lado direito aparece um novo termo P, que indica que a reescrita será realizada com probabilidade P. Além disso, é necessário acontecer o *match* com o estado t e *cond* ser verdadeiro. Essa nova propriedade modifica o comportamento padrão das reescritas determinísticas, tornado-as estocásticas. Logo, as simulações apresentadas neste trabalhos são probabilísticas.

Capítulo 3

SELECTIVE VERIFICATION IN APPLICATION LAYER - SEVEN

No capítulo anterior, foi apresentado fundamentos necessários para compreender o conceito de ADDoS, bem como uma breve explicação sobre o funcionamento da Lógica de Reescrita, *Maude* e *PVeStA*. Neste capítulo, é apresentado a metodologia do *SeVen*, seu algoritmo e possíveis instâncias de uso.

A estratégia, *Selective Verification in Application Layer* (*SeVen*), proposta por este trabalho é baseado no ASV [12]. Enquanto o ASV foi implementado para mitigar ataques de negação de serviço na camada de transporte assumindo uma comunicação *stateless* entre clientes e servidores, o *SeVen* foi projetado para mitigar ADDoS adotando uma noção de estado entre as mensagens utilizadas no protocolo. Essa é uma diferença fundamental, pois vários ataques realizados na camada de aplicação, como o HTTP POST, não são *stateless*, como apresentado no Capítulo 2.

Como no ASV, o *SeVen* não responde imediatamente as requisições recebidas pela aplicação, ou seja, é esperado um tempo *ts*, no qual denominamos de rodada. Durante uma rodada, o *SeVen* acumula mensagens em um buffer interno e responde de acordo com os critérios explicados a seguir. O *SeVen* é composto por um número natural (*PMod*) e dois buffers (*P,R*), representados por uma lista.

$$\{P, R, PMod\}$$

O buffer *P* representa as requisições parcialmente processadas e *R* as requisições recebidas que podem ser processadas após uma rodada, caso não sejam rejeitadas. O número natural *PMod* é um contador, como no ASV, usado para modificar a distribuição de probabilidade

quando o buffer P está cheio. Cada elemento dos buffers P e R é uma tupla da forma $\{id, N, T, \ldots\}$, onde id é um identificador único para cada requisição, representado da seguinte forma:

$$\{ip : porta\}$$

É assumido que não existem duas requisições distintas com o mesmo identificador. Os atributos N e T são números naturais. O número T indica o total de *bytes* a serem processados por uma requisição, enquanto N tem sua definição vinculada ao *buffer*: Em P, o número N indica a quantidade de *bytes* processados, em R representa o número de *bytes* recebidos que podem ser processados após uma rodada. As reticências significam que outros parâmetros podem ser adicionados a tupla, como, por exemplo, o tempo. A inserção ou não de novos parâmetros vai depender da estratégia utilizada para mitigar ataques DoS, sendo que algumas possíves estratégias do *SeVen* são apresentadas na seção 3.2.

O buffer R é um espelho do buffer P, ou seja, para cada requisição em R existe uma tupla correspondente em P.

$$\{id, M, T\} \in R \Rightarrow \{id, N, T\} \in P$$

Por exemplo, considere um caso genérico com os seguintes *buffers*:

P_1 = [$\{id_1, 30, 100\}$, $\{id_2, 5, 10\}$, $\{id_3, 15, 100\}$]

R_1 = [$\{id_1, 2, 100\}$, $\{id_2, 0, 10\}$, $\{id_3, 85, 100\}$]

O exemplo mostra que a aplicação recebeu e processou respectivamente 30, 5, 15 bytes dos identificados *id1, id2 e id3*. E apenas recebeu, mas não processou os bytes 2, 0, 85 dos identificados *id1, id2 e id3*. respectivamente. Esses bytes podem ser processados ao final de uma rodada.

Outro parâmetro do *SeVen* é o tamanho máximo (k) de requisições que os buffers P e R podem armazenar. Esse parâmetro é fundamental para o comportamento da defesa. Durante uma rodada, o *SeVen* não rejeita pacotes enquanto o número de requisições for menor ou igual a k. No entanto, se o buffer R já possui k requisições e chega um novo pacote, então, o *SeVen* possui duas opções:

- Descartar o pacote, isto significa que as requisições armazenadas no buffer não sofrem alteração.

- Aceitar o pacote, isto significa que uma das requisições armazenadas no buffer deve ser substituída pelo novo pacote.

A decisão de ficar com o novo pacote e, se for caso, escolher qual requisição armazenada no *buffer* será removida é determinada probabilisticamente, ou seja, usando uma distribuição de probabilidade ou simplesmente adicionando pesos probabilísticos as requisições. No final de uma rodada, o *SeVen* responde todas as requisições que permanecem no *buffer*.

Esse tipo de defesa funciona porque sempre que uma aplicação está sofrendo um DoS, o atacante tenta ocupar todos os recursos da aplicação e, portanto mensagens de atacantes têm uma maior probabilidade de serem descartadas, proporcionando uma maior disponibilidade para os usuários legítimos. Além disso, os ataques do tipo *Low-Rate* tentam ocupar o *buffer* de uma aplicação o máximo de tempo possível, como pode ser visto na Figura 3.1, onde as linhas vermelhas tracejadas representam os atacantes alocados no *buffer* em relação ao tempo e as linhas tracejadas pretas representam os clientes inseridos no *buffer* durante um período mais curto. Logo, quando o *buffer* do *SeVen* estiver cheio e a estratégia de descarte for acionada, a probabilidade de escolher um atacante é bem maior do que um cliente legítimo.

Figura 3.1: Espaço temporal de clientes e atacantes no buffer de uma aplicação

3.1 Algoritmo do SeVen

A definição das probabilidades do *SeVen* é uma etapa essencial em seu algoritmo. Essas probabilidades são configuráveis e podem aparecer em duas etapas: 1) Quando o *buffer* está lotado e uma nova requisição chega ao *SeVen*, é preciso decidir se mantém ou descarta essa nova requisição. Nessa etapa, é possível configurar o *PMod* para modificar a distribuição de probabilidade. 2) Caso o *SeVen* decida ficar com o novo pacote, é preciso escolher qual das requisições armazenadas no *buffer* será removida. Nesse momento, é preciso definir qual o real escopo da defesa. Por exemplo, se o tempo alocado no *buffer* de uma aplicação é um atributo importante, o *SeVen* pode ser configurado com uma distribuição de probabilidade que priorize ou não essas requisições no momento da escolha de qual requisição será removida.

Com o intuito de facilitar o entendimento do algoritmo do *SeVen*, a explicação a seguir usa o *PMod* como um simples contador, ou seja, sempre que uma nova requisição chega e o *buffer* está cheio, ele é incrementado em uma posição. E a distribuição de descarte utilizada é a uniforme discreta. Além disso, é importante salientar que o algoritmo apresentado foi baseado na formalização apresentada no Capítulo 4.

No início de cada rodada, os elementos em R são da seguinte forma: $\{id_i, 0, T\}$, indicando que o valor associado a requisição é 0, pois no início de cada rodada nenhuma solicitação foi recebida. Além disso, *PMod* também é zero. A seguir é descrito o algoritmo do *SeVen* durante uma rodada de *ts* segundos, onde $\setminus$ é o operador que remove um elemento do buffer e @ é o operador que concatena duas listas:

1. Enquanto o buffer P não está cheio, isto é, comprimento de $P < k$, a aplicação permanece acumulando mensagens da seguinte forma {id, N, T}:

   (a) Se existe uma solicitação em P com o mesmo identificador *id* e com o mesmo número de bytes (T), isto significa que a requisição recebida é uma continuação de uma solicitação anterior, logo, o valor de N é atualizado da seguinte forma: R := (R $\setminus$ {id, M, T}) @ [{id, N + M, T}], ou seja, o novos M bytes são concatenados aos N bytes anteriores.

   (b) Caso contrário, é considerada uma nova requisição, logo: R :=R@ [{id, 0, T}] e P :=P@[{id, 0, T}].É importante observar que a requisição ainda foi não processada, caracterizada pelo valor 0 adicionado ao buffer P.

2. Se o buffer P está cheio, isto é, o comprimento de $P \geq k$ e chega uma nova mensagem r = {id, N, T}:

(a) Incrementa *PMod: PMod* := *PMod* + 1

(b) Como acontece em 1.(a), se {id, M, T} $\in R$, o valor de N é atualizado, isto é, $R :=$ (R $\setminus$ {id, M, T}) @ [{id, N + M, T}].

(c) Caso contrário, é considerada uma nova requisição. Isto significa que a aplicação não é capaz de alocar a nova mensagem. Assim, a defesa deve decidir entre aceitar ou descartar o pacote. Esta opção é descrita a seguir:

i. Para decidir entre aceitar ou descartar um pacote, o *SeVen* gera um número aleatório *rand*. Se $rand \leq Prob$, onde *Prob* é descrito abaixo, então, o *SeVen* aceita o pacote, caso contrário o pacote r é descartado.

$$Prob = \frac{k}{k+PMod}$$

Sempre que o buffer P está lotado o valor de *PMod* é incrementado, com isso, a probabilidade de descarte aumenta. É importante ressaltar que o *SeVen* não está limitado a esta probabilidade, ou seja, outras estratégias em 2(c)i podem ser utilizadas.

ii. Se o *SeVen* decide descartar o pacote, uma mensagem é enviada para o *id* informando que não foi possível responder a solicitação.

iii. Se o *SeVen* decide aceitar o pacote, então, de acordo com uma distribuição uniforme de probabilidade (U) é escolhido um usuário que deve ser removido do buffer. Vale salientar que o objetivo do *SeVen* ao escolher a distribuição uniforme (U) é ser o mais genérico possível. Isto é, não ser configurado para um ataque em particular. No entanto, outras configurações podem ser utilizadas, como pode ser visto na seção seguinte.

3. Uma vez que a rodada termina, ou seja, se passaram *ts* segundos, a aplicação processa as requisições em R, executando os seguintes passos:

(a) $PMod = 0$

(b) Em cada tupla {id, N, T} $\in R$, com $N > 0$, é atualizado o valor de N em P e reiniciado o valor de N no buffer R:

i. P:= (P $\setminus$ {id, M, T}) @ [{id, N + M, T}]

ii. R:= (R $\setminus$ {id, M, T}) @ [{id, 0, T}]

(c) Cada tupla {id, M, T} $\in P$, com $M \geq T$, é removido dos buffers P e R:

i. P:= (P $\setminus$ {id, M, T})

ii. R:= (R $\setminus$ {id, M, T})

Esta etapa representa que a requisição foi concluída e para cada tupla é enviado uma mensagem de confirmação (ACK).

A seguir é descrito um exemplo ilustrativo do funcionamento do *SeVen*:

Considere os buffers P_1, R_1 e os parâmetros *PMod* e k:

P_1 = [$\{id_1, 30, 100\}$, $\{id_2, 5, 10\}$, $\{id_3, 15, 100\}$]

R_1 = [$\{id_1, 2, 100\}$, $\{id_2, 0, 10\}$, $\{id_3, 85, 100\}$]

$PMod = 0$

$k = 4$ (Tamanhos dos buffers P e R)

Se a aplicação recebe uma requisição $\{id_4, 2, 10\}$, de acordo com o algoritmo, apenas adicionamos a nova requisição ao *buffer*:

P_2 = [$\{id_1, 30, 100\}$, $\{id_2, 5, 10\}$, $\{id_3, 15, 100\}$, $\{id_4, 0, 10\}$]

R_2 = [$\{id_1, 2, 100\}$, $\{id_2, 0, 10\}$, $\{id_3, 85, 100\}$, $\{id_4, 2, 10\}$]

Considerando que chegou uma nova requisição id_5, 4, 10. Uma vez que o *buffer* está cheio, é gerado um número aleatório e comparado com *Prob* (Passo 2(c)i), para decidir se aceita ou descarta o pacote. Supondo que o *SeVen* decidiu aceitar a requisição, neste caso, uma das requisições armazenada no *buffer* será descartada, com isso, é gerado um número aleatório baseado na distribuição uniforme (Passo 2(c)iii). Digamos que a requisição sorteada foi o id_2, logo os buffers P e R sofrem alterações:

P_3 = [$\{id_1, 30, 100\}$, $\{id_3, 15, 100\}$, $\{id_4, 0, 10\}$, $\{id_5, 0, 10\}$]

R_3 = [$\{id_1, 2, 100\}$, $\{id_3, 85, 100\}$, $\{id_4, 2, 10\}$, $\{id_5, 4, 10\}$]

Neste momento, uma mensagem é enviada para id_2 informando que sua requisição foi descartada. Após isso, suponha que a rodada termine, ou seja, se passaram *ts* segundos. A aplicação processa as requisições em R (Passo 3b), obtendo:

P_4 = [$\{id_1, 32, 100\}$, $\{id_3, 100, 100\}$, $\{id_4, 2, 10\}$, $\{id_5, 4, 10\}$]

R_4 = [$\{id_1, 0, 100\}$, $\{id_3, 0, 100\}$, $\{id_4, 0, 10\}$, $\{id_5, 0, 10\}$]

Finalmente, visto que a requisição id_3 foi concluída (100/100), a aplicação o remove dos buffers e envia uma mensagem de confirmação (ACK) para id_3, resultando em:

$P_5 = [\ \{id_1, 32, 100\}\ ,\ \{id_4, 2, 10\}\ ,\ \{id_5, 4, 10\}\ ]$

$R_5 = [\ \{id_1, 0, 100\}\ ,\ \{id_4, 0, 10\}\ ,\ \{id_5, 0, 10\}\ ]$

3.2 Instâncias do SeVen

Como explicado anteriormente, o *SeVen* dispõe de um algoritmo probabilístico simples para mitigar ADDoS. Um ponto importante do exemplo ilustrado na seção 3.1, é a escolha da distribuição uniforme discreta (U) para decidir aleatoriamente qual requisição deve ser removida quando seu *buffer* estiver cheio. Ao escolher a distribuição uniforme (U), o *SeVen* toma todo o cuidado necessário para não favorecer qualquer requisição em especial. Isso significa que todas as requisições têm a mesma probabilidade de serem removidas. Neste caso, o *SeVen* se encontra na sua forma "mais genérica", pois não é configurado para mitigar um ataque em uma aplicação em particular.

Embora o *SeVen* tenha sido configurado com a distribuição uniforme, ele não está limitado a essa configuração. Diante disso, além do *SeVen (U)* apresentado na seção anterior, esta seção discute outras possíveis instâncias do *SeVen* em relação ao *PMod* e a distribuição de descarte. A árvore dessas instâncias é ilustrada na Figura 3.2.

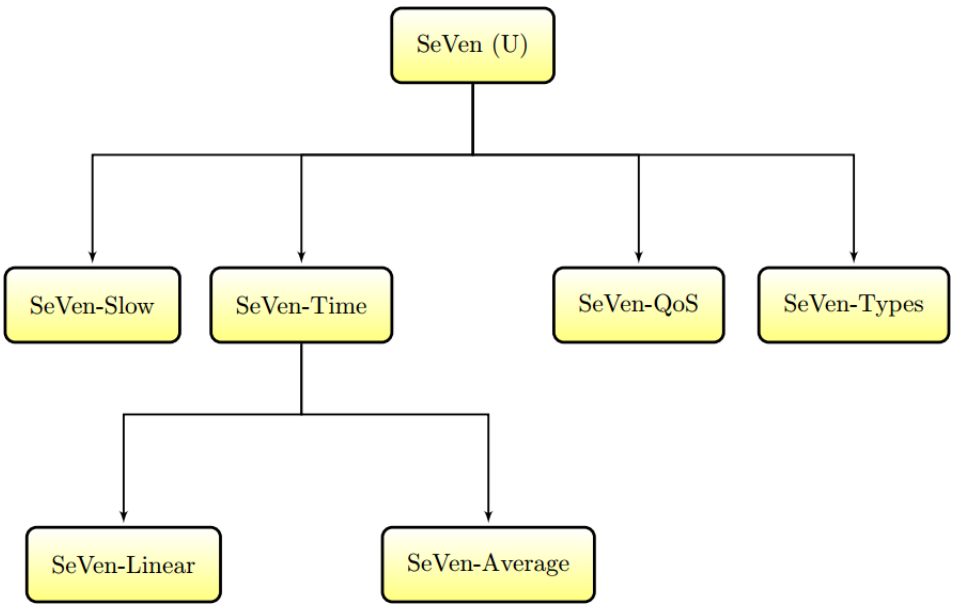


Figura 3.2: Árvore das possíveis instâncias do SeVen.

3.2.1 Tempo alocado na Aplicação (*SeVen-Time*)

A configuração desta subseção inclui o parâmetro "tempo"como peso na decisão de qual requisição será escolhida e, consequetemente removida do *buffer* da aplicação. Nesse contexto, dois possíveis cenários são apresentados a seguir.

3.2.1.1 *SeVen-Linear*

A estratégia *SeVen-Linear* retrata o cenário no qual o *SeVen* prioriza os clientes mais rápidos em relação ao tempo. Isso significa que no momento do *SeVen-Linear* escolher qual requisição deve ser removida, ele analisa individualmente todas as requisições HTTP armazenadas e calcula suas probabilidades inserindo o tempo de chegada do pacote como parâmetro, gerando um cenário em que clientes que estão ocupando o *buffer* por mais tempo, têm uma probabilidade maior de serem escolhidos. O gráfico ilustrando o comportamento do *SeVen-Linear* é apresentado na Figura 3.3.

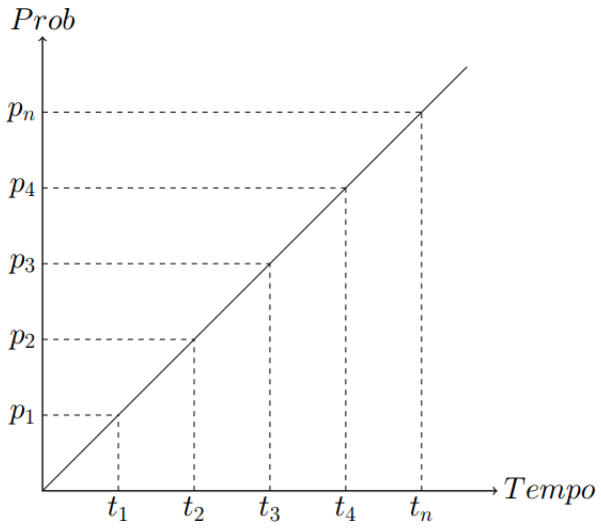


Figura 3.3: Comportamento do SeVen utilizando o SeVen linear.

Embora não tenha sido realizado experimentos com o cenário linear, sua implementação leva a crer que é possível mitigar ataques cujo objetivo seja ocupar o máximo de tempo possível o *buffer* de uma aplicação, como, por exemplo, os ataques de *Low Rate* apresentados no Capítulo 2. Além do HTTP, outros protocolos que usam o tempo como um parâmetro relevante, como o protocolo SIP[1], podem utilizar esse tipo de estratégia para evitar que atacantes ocupem todas as linhas ou recursos do serviço.

[1]SIP é um protocolo da camada de aplicação que pode criar, modificar e finalizar sessões multimídia (conferências), tais como chamadas telefônicas via Internet. [35]

Em contrapartida, é preciso ter uma cautela sobre a real eficiência dessa estratégia. Por exemplo, se a aplicação WEB for de uma companhia aérea, os usuários legítimos têm de preencher alguns formulários para realizar a compra de uma passagem aérea. Nesse cenário, o uso do *SeVen-Linear* não parece ser apropriada, pois os usuários realizam solicitações lentas e o *SeVen* provavelmente removerá esses clientes durante suas requisições, gerando um alto índice de falsos positivos.

A seguir, um exemplo de uma possível configuração do *SeVen-Linear*.

Como o tempo é um parâmetro necessário no *SeVen-Linear*, as tuplas dos elementos armazenados no *buffer* do *SeVen* sofrem uma alteração e o atributo tempo (*TIME*) é inserido:

$$\{ID, N, T, TIME\}$$

Onde o atributo *TIME* armazena uma abstração do tempo, em segundos, em que a requisição chegou ao *SeVen*. Por exemplo, considere os seguintes *buffers* P_1 e R_1 e os parâmetros k, *PMod* e gt:

P_1 = [$\{id_1, 60, 100, 1\}$, $\{id_2, 50, 100, 4\}$, $\{id_3, 40, 100, 6\}$]

R_1 = [$\{id_1, 40, 100, 1\}$, $\{id_2, 50, 100, 4\}$, $\{id_3, 60, 100, 6\}$]

k = 3 (Tamanhos dos buffers P e R)

PMod = 0

gt = 6 (Tempo atual)

No momento $gt = 7$, o *SeVen* recebe a requisição $\{id_4, 70, 100, 7\}$, no entanto, como o *buffer* se encontra cheio, o *SeVen* precisa decidir se mantém ou descarta a nova mensagem. Nesse instante, *PMod* = 1 e a nova requisição tem 75% ($\frac{k}{k+PMod}$) de chance de entrar no *buffer*. Vamos supor que o *SeVen* opte por manter a mensagem. Logo, baseado na estratégia do *SeVen-Linear*, é realizado os cálculos probabilísticos e a requisição do identificador id_1 é escolhida, o que faz sentido, pois é a solicitação que está a mais tempo armazenada no *buffer*.

Logo os *buffers* P e R sofrem alterações:

P_2 = [$\{id_4, 0, 100, 7\}$, $\{id_2, 50, 100, 4\}$, $\{id_3, 40, 100, 6\}$]

R_2 = [$\{id_4, 70, 100, 7\}$, $\{id_2, 50, 100, 4\}$, $\{id_3, 60, 100, 6\}$]

k = 3 (Tamanhos dos buffers P e R)

$PMod = 1$

$gt = 7$ (Tempo atual)

Esse é o cenário, em relação ao tempo, considerado mais simples, pois o único critério levado em consideração no cálculo probabilístico é o tempo em que um cliente está armazenado no *buffer*. Além disso, note que o *PMod* foi mantido como um contador simples, pois para as instâncias *SeVen-Time* não viu-se a necessidade de alterá-lo. O próximo cenário leva em consideração o tempo médio em que uma requisição leva para ser respondida.

3.2.1.2 Tempo Médio das Requisições (*SeVen-Average*)

Além de priorizar os clientes rápidos, este método considera o tempo médio como um dos critérios nos cálculos probabilísticos. Nesse sentido, o comportamento do *SeVen* é ilustrado na Figura 3.4, onde é assumido que a aplicação conhece o tempo médio (t_m) de resposta das requisições. Assim, clientes com tempo menor ou igual a t_m têm uma menor probabilidade de serem escolhidos, atenuando o problema de rejeitar clientes legítimos com requisições lentas citado anteriormente.

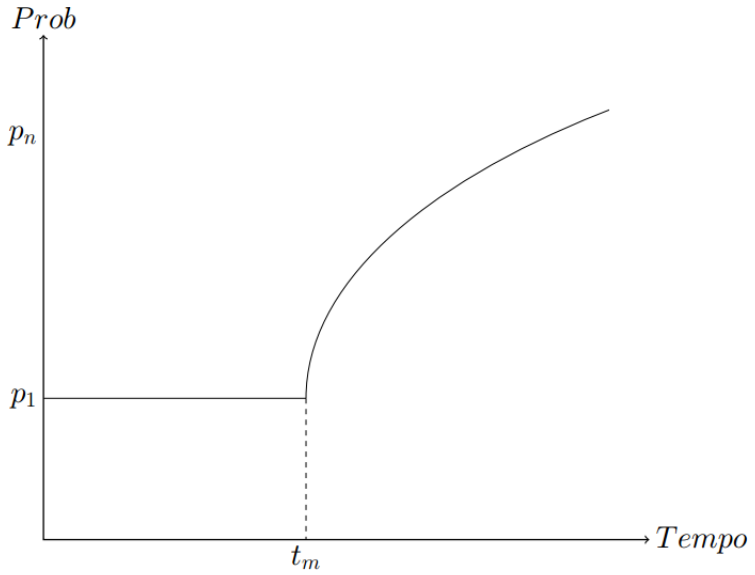


Figura 3.4: Comportamento do SeVen utilizando como critério o tempo médio das requisições.

Para o melhor entendimento, considere os *buffers* P_1 e R_1, com o atributo *TIME*, e os parâmetros k, *PMod*, t_m e gt:

$P_1 = [\ \{id_1, 10, 250, 6\}\ ,\ \{id_2, 210, 300, 7\}\ ,\ \{id_3, 60, 100, 10\}\ ]$

R_1 = [$\{id_1, 1, 250, 6\}$, $\{id_2, 90, 300, 7\}$, $\{id_3, 30, 100, 10\}$]

k = 3 (Tamanhos dos buffers P e R)

$PMod$ = 0

t_m = 5 (Tempo médio)

gt = 12 (Tempo atual)

No instante gt = *13*, uma nova solicitação $\{id_4, 10, 25, 13\}$ chega e com 75% ($PMod$ = *1*) de chance de manter o novo pacote, o *SeVen* decide ficar com id_4. Nesse momento, o *SeVen* verifica o atributo de tempo (*TIME*) em todas as solicitações e efetua seus cálculos probabilístico baseados na estratégia da Figura 3.4. Supondo que o *SeVen* escolha remover o id_1, o que é coerente, pois é a única requisição que excedeu o tempo médio e, consequentemente a requisição com maior probabilidade de ser escolhida, os *buffers* P e R sofrem as alterações:

P_2 = [$\{id_4, 0, 25, 13\}$, $\{id_2, 210, 300, 7\}$, $\{id_3, 60, 100, 10\}$]

R_2 = [$\{id_4, 10, 25, 13\}$, $\{id_2, 90, 300, 7\}$, $\{id_3, 30, 100, 10\}$]

k = 3 (Tamanhos dos buffers P e R)

$PMod$ = 1

Tm = 5 (Tempo médio)

gt = 13 (Tempo atual)

Além dessa estratégia parecer mais adequada do que a anterior ao protocolo HTTP, ela também parece ser apropriada ao protocolo SIP, pois, em virtude do conhecimento prévio do tempo médio de uma ligação, a defesa proporciona uma aplicação com uma qualidade de serviço melhor.

3.2.2 Qualidade de Serviço (*SeVen-QoS*)

A Qualidade de Serviço ou QoS (*Quality of Service*) é um conjunto de tecnologias para a gerenciamento de tráfego de rede em uma forma rentável para melhorar as experiências de usuário em ambientes corporativos, bem como para escritórios pequenos e domésticos. As tecnologias QoS permitem medir a largura de banda, detectar mudanças nas condições de rede (como congestionamento ou disponibilidade de largura de banda) e priorizar ou suprimir o

tráfego [36]. Esta subseção apresenta uma estratégia do *SeVen* que prioriza a disponibilidade de serviço em relação ao QoS.

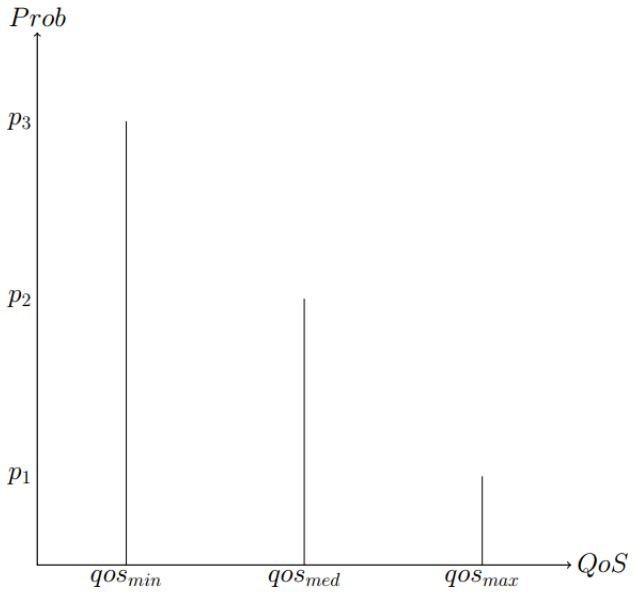


Figura 3.5: Comportamento do SeVen utilizando QoS.

Um termo bastante usado ao falar em qualidade de serviço é o SLA (*Service Level Agreement*). O SLA é um acordo entre um cliente e um provedor de serviço ou aplicação [37]. SLAs distintos implicam em diferentes modalidades de QoS que permitem diferenciar a maneira em que as requisições são tratadas, de forma a garantir um melhor atendimento aos usuários.

Nesse sentido, o esboço do *SeVen-QoS*, ilustrado na Figura 3.5, apresenta três tipos de SLAs: QoS_{min}, QoS_{med} e QoS_{max}. Essas modalidades de QoS podem ter diferentes configurações, onde cada uma delas vai depender da aplicação que está oferecendo o serviço. Com o intuito de exemplificar um caso do *SeVen-QoS*, os SLAs foram definidos das seguintes formas:

- **QoS_{min} –** É a modalidade mais acessível, pois os usuários podem acessar gratuitamente os recursos da aplicação. No entanto, quando o *buffer* está cheio, os usuários QoS_{min} têm uma menor probabilidade de serem aceitos e caso já estejam alocados no *buffer* e um novo pacote chega ao *SeVen-QoS*, eles têm a maior probabilidade de serem removidos.

- **QoS_{med} –** A partir da modalidade QoS_{med}, clientes devem pagar para desfrutar de uma aplicação com maior desempenho e disponibilidade. Os clientes associados a essa modalidade têm uma menor probabilidade de serem removidos quando o *SeVen-QoS* estiver sobrecarregado. O que pode ser visto na Figura 3.5, onde $p_2 < p_3$. Além disso, tem uma probabilidade média de serem aceitos quando o *buffer* estiver cheio. Dependendo da aplicação, pode ser considerada a modalidade mais rentável.

- **$\mathbf{QoS_{max}}$ –** É a modalidade mais custosa (financeiramente) para os usuários, no entanto, ao mesmo tempo, é a que oferece a maior disponibilidade e desempenho. *SeVen-QoS* prioriza os usuários QoS_{max} quando seu buffer está cheio e uma nova requisição é recebida. Isso pode ser visualizado na Figura 3.5, onde os clientes associados QoS_{max} têm a menor probabilidade de serem escolhidos ($p_1 < p_2 < p_3$).

Para uma melhor compressão do *SeVen-QoS*, é apresentado um exemplo a seguir, onde foi inserido o atributo de *QoS* informando qual o SLA de cada usuário:

$$\{ID, N, T, QoS\}$$

Além disso, considere os *buffers P* e *R* e os parâmetros *k* e *PMod*:

P_1 = [$\{id_1, 60, 100, QoS_{med}\}$, $\{id_2, 50, 100, QoS_{min}\}$, $\{id_3, 40, 100, QoS_{max}\}$]

R_1 = [$\{id_1, 30, 100, QoS_{med}\}$, $\{id_2, 40, 100, QoS_{min}\}$, $\{id_3, 60, 100, QoS_{max}\}$]

k = 3 (Tamanhos dos buffers P e R)

$PMod_{min} = 0$

$PMod_{med} = 0$

$PMod_{max} = 0$

Nesse exemplo, vamos considerar três instâncias e comportamentos distintos para o *PMod*: Caso chegue um novo pacote com QoS_{max}, o $PMod_{max}$ será um contador simples, como nos exemplos anteriores. Caso o novo pacote seja vinculado ao QoS_{med}, $PMod_{med} = PMod_{med} + 3$. Por fim, caso o novo pacote seja associado ao QoS_{min}, $PMod_{min} = PMod_{min} + 5$. O intuito é priorizar os clientes com maior QoS quando enviarem uma requisição HTTP e o *buffer* da aplicação estiver cheio. Além disso, esse exemplo também ilustra a estratégia usada para determinar se um pacote será mantido ou não no *buffer*.

Suponha que chegue uma nova mensagem $\{id_4, 57, 100, QoS_{med}\}$ e o *buffer* está cheio, *SeVen-QoS* precisa decidir se fica ou descarta id_4. Como id_4 tem o QoS_{med}, $PMod_{med} = 3$, logo a requisição tem 50% de ser aceito pela aplicação. Vamos supor que o *SeVen-QoS* decida ficar com a mensagem. Para isso, é necessário remover alguma requisição que esteja no *buffer*. Neste momento, é realizado um cálculo probabilístico seguindo o esboço apresentado na Figura 3.5. Admitindo que id_2 seja escolhido, visto que id_2 está associado a modalidade QoS_{min} e, consequentemente, é a requisição que tem a maior probabilidade de ser removida, os *buffers* sofrem as alterações:

P_2 = [$\{id_1, 60, 100, QoS_{med}\}$, $\{id_4, 0, 100, QoS_{med}\}$, $\{id_3, 40, 100, QoS_{max}\}$]

R_2 = [$\{id_1, 30, 100, QoS_{med}\}$, $\{id_4, 57, 100, QoS_{med}\}$, $\{id_3, 60, 100, QoS_{max}\}$]

k = 3 (Tamanhos dos buffers P e R)

$PMod_{min} = 0$

$PMod_{med} = 3$

$PMod_{max} = 0$

Neste momento, suponha que chegue uma nova mensagem $\{id_5, 73, 100, QoS_{min}\}$ e o *buffer* continua cheio. Como id_5 tem o QoS_{min}, $PMod_{min}$ deve ser incrementado em 5 posições, logo $PMod_{min} = 5$. Consequentemente, a requisição tem 27% de ser aceito pela aplicação. Digamos que o *SeVen-QoS* decida descartar id_5, logo os *buffers P* e R não sofrem alterações. Ao final de cada rodada, as instâncias do *PMod* são zeradas.

A estratégia de QoS apresentada privilegia os usuários que possuem as melhores modalidades de QoS, ou seja, sempre que uma aplicação estiver sobrecarregada, o *SeVen-QoS* dará preferência as modalidades QoS_{max}, QoS_{med}, QoS_{min}, nessa ordem.

3.2.3 Requisições Lentas (*SeVen-Slow*)

Assim como o *SeVen* é capaz de inserir o tempo em sua estratégia de defesa, ele também pode ser modificado para monitorar a quantidade de *bytes* das requisições, ou seja, é possível investigar a taxa de transmissão de *bytes* que chegam durante um intervalo de tempo e combinar essas informações para mitigar ataques de negação de serviço.

Foi encontrado na literatura um módulo apache, chamado *reqtimeout* [38], que usa os mesmo critérios para mitigar ADDoS. No entanto, esta subseção mostra que o *reqtimeout* pode ser visto como uma instância do *SeVen*, denominada de *SeVen-Slow*.

A abstração do *SeVen-Slow* ilustrada na Figura 3.6, funciona da seguinte forma:

Ao receber uma requisição HTTP (*req*) e o *buffer* estiver disponível, o *SeVen-Slow* adiciona *req* no *buffer* e inicia uma *thread* para verificar, em t_{esp} segundos, quantos *bytes* (x) do cabeçalho HTTP chegaram na aplicação. Após t_{esp}, caso o x seja menor do que um valor predeterminado (y) a requisição é removida imediatamente do *buffer*. Caso $x >= y$, *req* terá mais t_{esp} segundos para enviar o restante de seu cabeçalho. Em suma, *SeVen-Slow* tem dois ciclos de espera ($2 \times t_{esp}$) para receber o cabeçalho HTTP completo. No primeiro ciclo, ele checa a condição $x >= y$, se sim, *req* permanece no *buffer* e o *SeVen-Slow* espera por mais t_{esp} segundos para receber

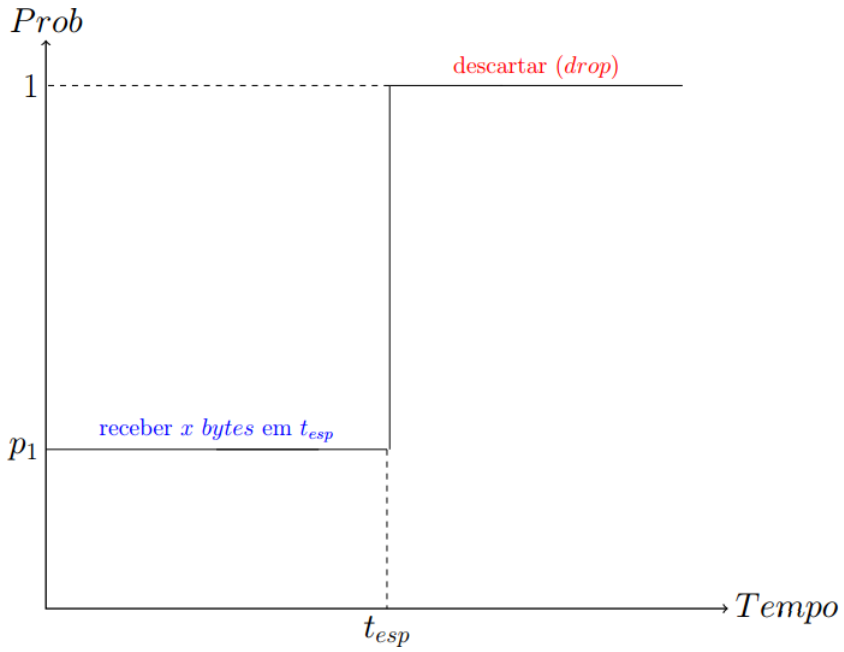


Figura 3.6: Configuração do SeVen para mitigar requisições lentas.

cabeçalho HTTP completo, se não a requisição é removida do *buffer*. Se a requisição HTTP recebida for completa, o *SeVen-Slow* responde a solicitação ao final de uma rodada.

Caso o *buffer* esteja cheio e uma nova solicitação chegue na aplicação, o *SeVen-Slow* volta a funcionar como o *SeVen* genérico. Isto é, todas as requisições têm a mesma probabilidade de serem removidas. No entanto, outras estratégias podem ser escolhidas, como, por exemplo, o *SeVen-Linear*. A preferência por uma estratégia em particular vai depender do propósito de cada aplicação.

Um exemplo do *SeVen-Slow* é apresentado a seguir com os *buffers* P e R e os parâmetro k, gt, t_{esp} e y. Além disso, foi preciso adicionar o atributo *TIME* nas tuplas das requisições:

$$\{ID, N, T, TIME\}$$

P_1 = [$\{id_1, 30, 100, 7\}$, $\{id_2, 10, 100, 7\}$, $\{id_3, 40, 100, 7\}$]

R_1 = [$\{id_1, 20, 100, 7\}$, $\{id_2, 1, 100, 7\}$, $\{id_3, 20, 100, 7\}$]

k = 3 (Tamanhos dos buffers P e R)

gt = 7 (Tempo atual)

t_{esp} = 5 (Tempo do ciclo)

y = 50 (Número mínimo de *bytes* para o 1º ciclo)

Para facilitar o exemplo, vamos supor que as três requisições chegaram no instante gt = 7, sendo assim, pela estratégia descrita acima, todas as requisições serão avaliadas no instante gt = 12. Digamos que a rodada seja de 1 segundo, então após cinco rodadas, os *buffers P* e *R* se encontram da seguinte maneira:

P_5 = [$\{id_1, 95, 100, 7\}$, $\{id_2, 15, 100, 7\}$, $\{id_3, 90, 100, 7\}$]

R_5 = [$\{id_1, 5, 100, 7\}$, $\{id_2, 1, 100, 7\}$, $\{id_3, 10, 100, 7\}$]

gt = 12 (Tempo atual)

Nesse momento, é preciso verificar quantos *bytes* chegaram dentro do intervalo t_{esp}. Analisando individualmente cada requisição, id_1 e id_3 enviaram mais do que *y bytes* no 1º ciclo, portanto estão aptos a continuar no *buffer*. Já o id_2 enviou apenas 15 *bytes* durante os primeiros t_{esp}, como $15 < y$, id_2 é removido imediatamente:

P_5 = [$\{id_1, 95, 100, 12\}$, $\{id_3, 90, 100, 12\}$]

R_5 = [$\{id_1, 5, 100, 12\}$, $\{id_3, 10, 100, 12\}$]

gt = 12 (Tempo atual)

Por sempre enviar um *byte* a cada rodada, possivelmente o usuário id_2 estava efetuando o ataque POST. Além disso, note que o 4º parâmetro de cada tupla foi atualizado com o tempo atual, ou seja, o *TIME* é usado para verificar em quanto tempo será checado o 2º ciclo de cada requisição. Nessa situação, quando gt for igual a 17.

3.2.4 Tipos de Requisições (*SeVen-Types*)

O Protocolo HTTP possui um conjunto de métodos, onde clientes podem utilizá-lo para realizar a troca de mensagens com uma aplicação WEB [39]. Desse conjunto, os mais populares são os métodos GET e POST. Enquanto o GET é geralmente usado para solicitar um objeto ou página WEB, o POST é usado para enviar arquivo de dados ou preencher formulários HTML. De acordo com [40], é recomendável utilizar o método POST para requisições longas, pois seu *buffer* aceita requisições até 8 Mb, enquanto o GET suporta no máximo 7607 caracteres (7.25×10^{-3} Mb).

Tanto a discrepância de *bytes* enviados entre os dois métodos, quanto o tempo que um cliente pode levar para preencher um formulário, podem implicar que o processamento da requisição GET seja mais rápido do que o POST. Por exemplo, o tempo para processar uma solicitação

de uma imagem comum usando o método GET é bem menor do que o preenchimento de um cadastro de uma loja virtual *online*, pois além de ser um processo tedioso, a aplicação deve fazer operações no banco de dados, o que pode causar um *delay* na resposta.

Baseado nessas informações, esta subseção apresenta o *SeVen-Types*, uma combinação entre o *SeVen-Linear* e o *SeVen-Average*.

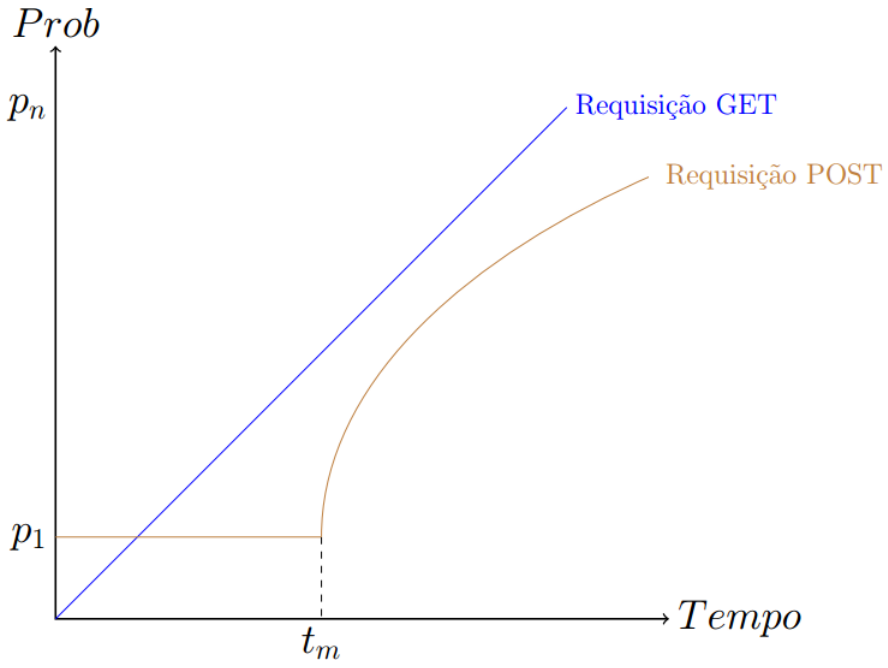


Figura 3.7: Configuração do SeVen-Types para mitigar ADDoS.

Partindo da premissa que as requisições GET são processadas rapidamente, a estratégia *SeVen-Linear* parece ser apropriado para monitorar essas requisições quando o *buffer* está cheio, pois quanto maior o tempo alocado, maior a probabilidade de ser removido, como pode ser visto na linha azul da Figura 3.7. Por outro lado, uma aplicação precisa de um tempo maior para processas requisições POST. Por esse motivo, supondo que o administrador de uma aplicação em particular saiba o tempo médio (t_m) de resposta de uma requisição POST, a estratégia *SeVen-Average* pode ser utilizada, pois propicia aos clientes, com tempo menor ou igual ao t_m, uma probabilidade menor de serem removidos. Isto é, oferece uma maior oportunidade de sucesso para os clientes legítimos realizarem suas solicitações em um tempo habitual (t_m).

Em particular, o *SeVen-Types* precisa analisar individualmente as requisições e verificar qual o método utilizado (GET ou POST) por cada um. Por exemplo, se o *buffer* estiver cheio, cálculos probabilísticos são efetuados para cada método seguindo o comportamento das curvas da Figura 3.7. A linha azul, usado para o método GET, representa o comportamento da *SeVen-Linear* e a linha marrom, utilizado para o método POST, é o *SeVen-Average*, ambos

exemplificados nas subseções 3.2.1.1 e 3.2.1.2, respectivamente.

Além disso, é possível modificar o *PMod* para priorizar um método em particular. Por exemplo, uma aplicação de uma companhia aérea possui diversos formulários que devem ser preenchidos na hora da compra de uma passagem aérea. Nesse contexto, é sensato priorizar as requisições POST. Logo, se o *buffer* do *SeVen-Types* estiver lotado, é possível configurar o *PMod* com uma probabilidade maior de aceitação para as requisições POST e menor para as requisições GET.

Capítulo 4

ESPECIFICAÇÃO FORMAL DO SEVEN

Para a formalização em Maude, foi adotado um modelo matemático para computação concorrente em sistemas distribuídos denominado de modelo de atores (*Actor Model*) [41]. Seguindo a filosofia da Programação Orientada a Objetos (POO), em que tudo pode ser considerado um objeto, o modelo de atores considera que tudo é um ator. Um ator é um agente concorrente que encapsula um estado e pode ter um identificador único. Esses atores podem se comunicar uns com os outros através de mensagem. Ao receber uma mensagem, um ator pode mudar o seu estado e responder a mensagem para outros atores [41]. Esse modelo é adequado e simples para especificar a troca de mensagens em um ambiente concorrente, como, por exemplo, uma rede de computadores.

Nesta capítulo são descritos os *sorts*, operadores e reescritas dos principais atores (Cliente, Atacante e Servidor/*SeVen*) da especificação em Maude e os resultados das simulações.

4.1 Cliente

Primeiramente, definimos os *sorts* do cliente, ou seja, os tipos utilizados em sua formalização:

- **Actor:** Representa o próprio ator, neste caso, o cliente.
  sort Actor .
- **Address:** Endereço único atribuído ao cliente.
  sort Address .
- **Status:** Especifica o estado de um cliente.
  sort Status .

- **AttributeSet:** Lista de atributos usados no construtor do cliente.
  sort AttributeSet .

- **Scheduler:** Todas as mensagens geradas por atores são inseridas em um *scheduler*, ordenado pelo tempo de cada requisição.
  sorts Scheduler ScheduleMsg ScheduleList .

Em seguida, seus operadores:

- **Estados do Cliente:** Operador utilizado para modificar o estado de um cliente. Dessa forma, caso o cliente esteja esperando uma resposta do servidor, seu estado é *waiting*. Caso o cliente tenha recebido uma resposta com sucesso do servidor, seu estado é *connected*. Esses operadores são usados para computador a disponibilidade dos usuários legítimos.
  op waiting : -> Status .
  op connected : -> Status .

- **Construtor do Cliente:** Construtor do Cliente. Ao instanciar um cliente, os parâmetros *Address* e *AttributeSet* são solicitados.
  op <name:_|_> : Address AttributeSet -> Actor .

- **TTS:** É o tempo entre um pedido de um cliente e uma resposta do servidor. Sendo assim, são utilizados dois operadores, *atime* e *stime*. O primeiro armazena o momento (*time*) da solicitação do cliente e o *stime* armazena o tempo ao receber uma resposta com sucesso do servidor. Nesse momento, é realizada a subtração entre eles (*stime - atime*). Esse operadores retornam um *Attribute*, o qual é um *subsort* do *AttributeSet*.
  op atime:_ : Float -> Attribute .
  op stime:_ : Float -> Attribute .

A seguir, um exemplo da instância de um cliente. O qual tem o identificador *cli*, enviando mensagem para o servidor *serApp*, com o estado inicial *waiting*. No momento do envio de uma mensagem, o *atime* recebe o tempo global *gt* e o *stime* é iniciado com *Infinity*, esse operador apenas armazena um valor (*time*) ao receber uma mensagem com sucesso do *serApp*.

```
<name: cli | server: serApp, atime: gt , stime: Infinity , status: waiting , AS >
```

Uma abstração das regras de reescrita do cliente é apresentada a seguir:

```
rl [Client-Send-GET] :
    < name: c(i) | server: serApp, status: waiting, tts: atime >
    (gt, c(i) <- poll)
    =>
    < name: c(i) | server: serApp, status: waiting, tts: atime >
    [gt + delay, serApp <- GET(c(i))] .
```

A reescrita *Client-Send-GET* representa o envio de uma mensagem GET para o servidor. Assim, ao receber uma mensagem *poll* (mensagem recebida ao criar um cliente), o ator c(i) envia uma mensagem GET para o servidor *serApp* no tempo gt + delay. Essa mensagem é adicionada no *scheduler* e ordenada pelo tempo gt + delay. Já a reescrita *Client-Receive-GET* especifica o comportamento de um cliente ao receber uma resposta com sucesso do servidor *serApp*. Nesse momento, o cliente recebe mensagem *ACK*, muda seu estado para *connected* e calcula o *tts* entre a hora atual e o *atime* (*gt - atime*).

```
rl [Client-Receive-GET] :
    < name: c(i) | server: serApp, status: waiting, tts: atime >
    (gt, c(i) <- ACK)
    =>
    < name: c(i) | server: serApp, status: connected, tts: (gt - atime) > .
```

Outra possível mensagem que um cliente pode receber do servidor é mensagem de *FAILED*, isso ocorre quando o servidor está com todos os seus recursos ocupados. Neste modelo, isso acontece quando o servidor está sofrendo um ataques de negação de serviço. Ao receber essa mensagem, o cliente não modifica seu estado, permanecendo como *waiting*. E o *tts* recebe *Infinity*, pois esse cliente, em particular, não faz uma nova solicitação ao servidor.

```
rl [Client-Receive-FAILED] :
    < name: c(i) | server: serApp, status: waiting, tts: atime >
    (gt, c(i) <- FAILED)
    =>
    < name: c(i) | server: serApp, status: waiting, tts: Infinity > .
```

4.2 Atacante

O atacante, ator responsável por efetuar o ataque de negação de serviço ao servidor, utiliza os mesmos *sorts* definidos na subseção do cliente. Sendo assim, seus operadores são:

- **Estados do Atacante:** Os operadores utilizados para modificar o estado de um atacante varia de acordo com o ataque escolhido. 1) GET Flood – *none* e *get*. 2) Slowloris – *none* e *getInc*. 3) POST – *none* e *post*. A aplicabilidade de cada operador é explicado com detalhes nas reescritas do atacante.
  ops none get : -> Status .
  ops none getInc : -> Status .
  ops none post : -> Status .

- **Construtor do Atacante:** É o operador do construtor de um atacante. Ao instanciar um atacante, os parâmetros *Address* e *AttributeSet* são solicitados.
  op <name:_|_> : Address AttributeSet -> Actor .

As reescritas dos atacantes foram modeladas de acordo com o ataque. Nesta especificação, foram modelados três ataques na camada de aplicação: GET FLOOD, Slowloris e POST.

GET FLOOD: As reescritas do ataque GET FLOOD seguem as transições de estado do autômato ilustrado na Figura 2.5. Ao instanciar um atacante *a(i)*, seu estado é iniciado com *none* e o ator recebe a mensagem *poll*. Em seguida, o atacante envia uma mensagem GET, no tempo *gt* + *delay*, e realiza a transição do estado *none* para o estado *get*. No momento em que a mensagem chega ao servidor, ele pode responder com um *ACK*, caso tenha recurso disponível para processar a nova requisição, ou com um *FAILED*, caso contrário. Se o atacante receber a mensagem ACK, ele segue realizando o ataque *flooding*. Caso receba a mensagem *FAILED*, o ataque é reiniciado, ou seja, ocorre uma transição do estado *get* para o estado *none*.

```
rl [AT-Send-GET-none] :
    < name: a(i) | server: serApp , status: none >
    (gt, a(i) <- poll)
    =>
    < name: a(i) | server: serApp , status: get >
    [gt + delay, serApp <- GET(a(i))] .

rl [AT-Receive-ACK-get] :
    < name: a(i) | server: serApp , status: get >
    (gt, a(i) <- ACK)
    =>
    < name: a(i) | server: serApp , status: get >
    [gt + delay, serApp <- GET(a(i))] .

rl [AT-Receive-FAILED-get] :
```

```
    < name: a(i) | server: serApp , status: get >
    (gt, a(i) <- FAILED)
    =>
    < name: a(i) | server: serApp , status: none >
    [gt + delay, serApp <- poll(a(i))] .
```

Slowloris: O ataque Slowloris é ilustrado na Figura 2.6, bem como seu autômato. Ao gerar um novo atacante *a(i)*, ele recebe a mensagem *poll* e seu estado é iniciado com *none*. Posteriormente, realiza a transição do estado *none* para o estado *getInc*, envia uma mensagem GET incompleta, no tempo *gt* + *delay*, e manda uma mensagem para si mesmo, no tempo *gt* + *Tc'*, com o intuito de enviar uma mensagem próxima ao *timeout* da aplicação WEB. No momento em que a mensagem chega ao servidor, a mensagem é inserida em seu *buffer*, caso tenha recurso disponível para aceitar a nova requisição. Caso contrário, a mensagem pode ser descartada e o servidor envia um *FAILED*. O atacante continua o ataque, enviando mensagens próximas do *timeout*, até que receba uma mensagem *FAILED*. Caso isso ocorra, o ataque é reiniciado.

```
rl [AT-Send-Slow-none] :
    < name: a(i) | server: serApp , status: none >
    (gt, a(i) <- poll)
    =>
    < name: a(i) | server: serApp , status: getInc >
    [gt + delay, serApp <- GET_INC(a(i))]
    [gt + Tc' , a(i) <- keepGet] .

rl [AT-Cont-Slow-getInc] :
    < name: a(i) | server: serApp , status: getInc >
    (gt, a(i) <- keepGet)
    =>
    < name: a(i) | server: serApp , status: getInc >
    [gt, serApp <- GET_INC(a(i))]
    [gt + Tc' , a(i) <- keepGet)] .

rl [AT-Receive-FAILED-getInc] :
    < name: a(i) | server: serApp , status: getInc >
    (gt, a(i) <- FAILED)
    =>
```

```
    < name: a(i) | server: serApp , status: none >
    [gt + delay, serApp <- poll(a(i))] .
```

POST: O ataque POST é ilustrado na Figura 2.7, bem como seu autômato. Ao instanciar um atacante *a(i)*, seu estado é iniciado com *none* e o ator recebe a mensagem *poll*. Em seguida, o atacante envia uma mensagem POST (sem os bytes do formulário), no tempo *gt* + *delay*, realiza o mapeamento do estado *none* para o estado *post* e manda uma mensagem para si mesmo, no tempo *gt* + *Tc'*, com o intuito de enviar uma mensagem próxima ao *timeout* da aplicação WEB. Como no ataque Slowloris, o servidor não pode processar a requisição, pois a requisição está incompleta. Se tiver espaço no buffer, a aplicação armazena a nova requisição, caso contrário, uma mensagem *FAILED* pode ser enviada. Enquanto não recebe um *FAILED*, o atacante mantém um *loop* para preencher o formulário, enviando byte a byte, sempre próximo ao *timeout* da aplicação.

```
rl [AT-Send-POST-none] :
    < name: a(i) | server: serApp , status: none >
    (gt, a(i) <- poll)
    =>
    < name: a(i) | server: serApp , status: post >
    [gt + delay, serApp <- POST_INC(a(i), len_req)]
    [gt + Tc' , a(i) <- data] .

rl [AT-Cont-POST-post] :
    < name: a(i) | server: serApp , status: post >
    (gt, a(i) <- data)
    =>
    < name: a(i) | server: serApp , status: post >
    [gt, serApp <- bytes(a(i), one_byte)]
    [gt + Tc' , a(i) <- data] .

rl [AT-Receive-FAILED-post] :
    < name: a(i) | server: serApp , status: post >
    (gt, a(i) <- FAILED)
    =>
    < name: a(i) | server: serApp , status: none >
    [gt + delay, serApp <- poll(a(i))] .
```

4.3 SeVen

Visando uma maior simplicidade e seguindo a implementação do ASV, o *SeVen* foi especificado como uma defesa e um servidor, ou seja, as reescritas do *SeVen* exercem a estratégia da defesa e também processam requisições (função de uma aplicação WEB). No entanto, essa abstração não deve afetar as análises realizadas nas simulações.

Além dos *sorts* apresentados na subseção do Cliente, o *SeVen* define os seguintes tipos:

- **Lista:** Para armazenar as mensagens HTTP, o *SeVen* implementa um *buffer* com os *sorts*: *EleBuf* – Elementos que podem ser armazenados no *buffer* (i.e., solicitações GET/POST). *Buffer* – Contém os elementos armazenados no *buffer*. *NBuffer* – Contém o *Buffer* e o comprimento (*length*) do *Buffer*. Consequentemente, *EleBuf* e *Buffer* são *subsorts* de *NBuffer*.
  sorts EleBuf Buffer NBuffer .
  subsorts EleBuf Buffer < NBuffer .

Os principais operadores do *SeVen* são:

- **Operadores da Lista:** O *SeVen* implementa alguns operadores para manipular o seu *buffer*, como, por exemplo: Adicionar (*add*) um elemento, remover (*remove*), atualizar (*update*), checar o tamanho (*size*) do buffer, verificar se um elemento está no buffer (*_in_*), substituir (*swap*) um elemento por outro. E o *reply*, que ao final de cada rodada, verifica quais requisições podem ser respondidas pelo *SeVen*.
  op add : NBuffer EleBuf -> NBuffer .
  op size : NBuffer -> Nat .
  op remove : Buffer Address -> NBuffer .
  op _in_ : Address Buffer -> Bool .
  op update : NBuffer EleBuf EleBuf -> NBuffer .
  op swap : Buffer EleBuf Nat -> Buffer .
  op reply : Address Buffer Float -> ScheduleList .

- **Atributos da Lista:** Como o *SeVen* possui dois *buffers*, foram implementados dois operadores, um para cada *buffer*. Esses atributos fazem parte do construtor de um ator *SeVen*.
  op p-set:_ : NBuffer -> Attribute .
  op r-set:_ : NBuffer -> Attribute .

- **pmod:** É o contador utilizado para modificar a probabilidade de aceitação de uma nova requisição quando o *buffer* está cheio.
  op pmod:_ : Nat -> Attribute .

- **Contador de Mensagens:** Este operador armazena a quantidade de mensagens que chegam no *SeVen*.
  op cnt:_ : Nat -> Attribute . .

- **Construtor do SeVen:** O construtor do *SeVen* tem a mesma assinatura apresentada nos clientes e atacantes, ou seja, ao instanciar um ator *SeVen*, os parâmetros *Address* e *AttributeSet* são solicitados.
  op <name:_|_> : Address AttributeSet -> Actor .

- **Probabilidades do SeVen:** No momento em que o *buffer* está cheio e uma nova requisição chega, o *SeVen* precisa decidir se mantém ou descarta essa requisição. Nesse caso, o *SeVen* invoca os operadores *accept-prob*, passando como parâmetro o valor do *pmod*, e o *sampleBerWithP* (sBWP), o qual retorna um valor booleano informando se a mensagem será aceita (*true*) ou não (*false*). Caso o valor seja verdadeiro, *SeVen* chama o operador *sampleUniWithInt* (sUWI) para escolher, aleatoriamente, qual requisição será removida. Assim, a nova requisição pode ser adicionada.
  op accept-prob : Float -> Float .
  op sBWP : Float -> Bool .
  op sUWI : Nat -> Nat .

Uma abstração das regras de reescrita do *SeVen* é apresentada a seguir:

A reescrita *[SeVen]* ilustra o algoritmo do *SeVen* recebendo um requisição POST. Ao receber essa nova solicitação, o *SeVen* checa se a mensagem não está no *buffer P* e verifica algumas condições dos *buffers P e R*, como, por exemplo, se ambos estão cheios, se apenas o *Buffer P* está cheio e etc. Se o *SeVen* não tiver recursos para processar a nova mensagem, uma moeda é lançada para decidir se a mensagem deve ser descartado ou não, isto é, decide-se aleatoriamente de acordo com uma distribuição uniforme de probabilidade (U). Se a mensagem não for descartada, aleatoriamente uma requisição será removida para que a nova mensagem seja alocada. Neste momento, uma mensagem de *TIMEOUT* é criada para checar em *Tc* segundos se o usuário deve ser removido do *buffer* (mais detalhes na reescrita *SeVen-Timeout*). Por outro lado, caso a nova mensagem já esteja alocado no *buffer P*, entende-se como a continuação de uma requisição POST, logo os bytes do POST são atualizados (*update*) e uma mensagem *TIMEOUT* é criada.

```
rl [SeVen]:

<name: serApp | pmod: p , p-set: [lenP | P] , r-set: [lenR | R], cnt: i >
(Ser <- POST(Actor, lenPost)
=>
if (lenP >= lenBuffer) and not(Actor in P) ) then
   if (lenR >= lenBuffer) then
      if (sBWP(accept-prob(p))) then
```

```
   <name: serApp | pmod: (p + 1), p-set: [lenP | swap(P, < Actor 0 lenPost >, sUWI(lenP) )]
        , r-set: [lenR | swap(R, < Actor 0 lenPost >,sUWI(lenR))], cnt: s(i)>
     [ gt + Tc , (serApp <- TIMEOUT(Actor) )]
      else
          <name: serApp | pmod: (p + 1), p-set: [lenP | P], r-set: [lenR | R], cnt: s(i) >
           [gt + delay, Actor <- FAILED]
   else
       if (sBWP(accept-prob(p))) then
          <name: serApp | pmod: (p + 1.0) , p-set: [lenP | swap(P, < Actor 0 lenPost >
          , sUWI(lenP) )] , r-set: add( [lenR | R], < Actor 0 lenPost > ), cnt: s(i) >
          [ gt + Tc , (serApp <- TIMEOUT(Actor) )]
       else
          <name: serApp | pmod: (p + 1) , p-set: [lenP | P] , r-set: [lenR | R], cnt: s(i) >
          [gt + delay, Actor <- FAILED]
if (Actor in P) then
      <name: serApp | req-cnt: (p + 1) , p-set: update( [lenP | P], < Actor getBytes(Actor,P)
       getBytesPOST(Actor,P) >, < Actor (getBytes(Actor,P) + lenPost) getBytesPOST(Actor,P)) > )
          , r-set: [lenR | R], cnt: s(i) >
      [ gt + Tc , (serApp <- TIMEOUT(Actor) )]
else
    <name: serApp | pmod: p , p-set: add( [lenP | P], < Actor 0 lenPost > )
        , r-set: add( [lenR | R], < Actor 0 lenPost > ), cnt: s(i) >
   [ gt + Tc , (serApp <- TIMEOUT(Actor) )] .
```

```
rl [SeVen-Round]:
    <name: serApp | pmod: p , p-set: [lenP | P] , r-set: [lenR | R], cnt: i >
    (serApp <- ROUND) }
    =>
    <name: serApp | pmod: 0 , p-set: [lenP | P] , r-set: [lenP | P ], cnt: i >
    [gt, reply(serApp, P, gt) ]
    [gt + Ts , (serApp <- ROUND)] .
```

Como explicado neste Capítulo, o *SeVen* não processa imediatamente as requisição que recebe, ou seja, é esperado um tempo (*Ts*), no qual denominamos de rodada. Ao final de cada rodada, o *SeVen* invoca o operador *reply* para verificar quais mensagens podem ser respondidas, isto é, quais mensagens estão completas. Esse processo é especificado na reescrita *[SeVen-Round]*.

Por fim, a reescrita *[SeVen-Timeout]* simula uma função de uma aplicação WEB, o qual é denominado de *timeout*. No momento em que uma solicitação HTTP é armazenada em uma aplicação WEB,

ela pode permanecer por um tempo x, em segundos. Caso nenhuma outra requisição daquele mesmo identificador chegue durante os x segundos, a requisição é removida do *buffer* da aplicação. Na reescrita, a aplicação recebe uma mensagem de *TIMEOUT* e verifica se o tempo expirou para cada cliente em particular, se sim, o cliente é removido.

```
rl [SeVen-Timeout]:
    <name: serApp | pmod: p , p-set: [lenP | P] , r-set: [lenR | R], cnt: i >
    (serApp <- TIMEOUT (Actor))
    =>
    if (gt >= (getLastTime(Actor , P ) ) + Tc) then
      <name: serApp | pmod: p , p-set: [lenP + (-1) | remove(P, Actor) ]
         , r-set: [lenR + (-1) | remove(R, Actor) ], cnt: i >
     [ gt + delay , (Actor <- poll) ] .

\small
```

4.4 Simulações

Como explicado anteriormente, a formalização do *SeVen* em *Maude* adotou a abordagem presente em [42] [43] utilizando o modelo de atores (*Actor Model*), onde servidor, clientes e atacantes são atores enviando e recebendo mensagens. Essas mensagens são criadas através de uma equação determinística e são ordenadas, em relação ao tempo, em um *scheduler* que mantém uma fila de mensagens transmitidas e processadas pelos atores. Além disso, o *PVeStA* foi usado com a finalidade de modificar o comportamento padrão das reescritas determiníticas, tornando-as estocásticas.

Tanto nas simulações, quanto nos experimentos na rede, foi utilizada a estratégia genérica *SeVen (U)* ou simplesmente *SeVen* para mitigar ADDoS, uma vez que não surgiu a necessidade de configurar a defesa para um ataque em particular. O artigo completo das simulações, publicado em 2014 na conferência *Intelligence and Security Informatics Conference (JISIC)*, pode ser encontrado em [44].

A formalização possui os seguintes parâmetros:

- **Timeout da Aplicação - Tc:** Parâmetro de *timeout* da aplicação. Isto significa que, se a aplicação não receber qualquer requisição de um agente em *Tc* segundos, então, a conexão do agente é encerrada, e ele é removido dos buffers *R* e *P*. Neste trabalho, foi mantido o mesmo valor de *Tc* utilizado no ASV [12], portanto *Tc* = 0.4 segundos.
- **Seven Round Time - Ts:** Tempo que o *SeVen* acumula requisições, conforme descrita no capítulo III. Neste trabalho, *Ts* = 0.4 segundos.

- **Tamanho do Buffer - k:** Comprimento dos buffers *P* e *R*.
- **Número de clientes e atacantes:** O número de clientes é fixo, igual a 200. E o número de atacantes varia de 8 a 280 atacantes.
- **Atraso na rede (Delay):** Em todas as simulações, o *delay* no envio de mensagens é de 0.1 segundos.
- **Intervalo de confiança**: O intervalo de confiança no *PVeStA* é de 0.01, ou seja, de 99%.
- **Tempo total da simulação:** Tempo total da simulação no *PVeStA*. Em todas as simulações, total = 40 segundos.

Além do *SeVen*, também foi formalizado uma versão simplificada do TAD [17], uma pesquisa que envolveu apenas ataques slowloris, apresentado em 1.2.2. O principal objetivo é obter algum tipo de comparação com a literatura, pois não há resultados estatísticos disponíveis, apenas resultados de experimentos reais na rede utilizando um pequeno número de atacantes, como em [17].

Nas simulações são descritas três métricas, onde são definidas a seguir. O operador $\bigcirc$ indica o avanço do tempo global, ou seja, um *tick* do relógio para processar uma nova mensagem no *scheduler*.

- *Disponibilidade* – Mede a taxa de disponibilidade do servidor, ou seja, a taxa de clientes com requisições bem sucedidas. Na fórmula dessa métrica, o *contSucesso* é um contador, iniciado com 0, incrementado sempre que um cliente recebe uma confirmação(ACK) do servidor:

$$\begin{aligned} disponibilidade(total) = \quad & \text{if } tempoGlobal > total \text{ then} \\ & \frac{contSucesso}{totalCliente} \\ & \text{else } \bigcirc\, disponibilidade(total) \end{aligned}$$

- *Média TTS* – Calcula o tempo médio, em segundos, que leva para um cliente receber uma confirmação (*ACK*) do servidor. Onde o $somaTTS$ é a soma de TTS dos clientes com requisições bem sucedidas:

$$\begin{aligned} medTTS(total) = \quad & \text{if } tempoGlobal > total \text{ then} \\ & \frac{somaTTS}{contSucesso} \\ & \text{else } \bigcirc\, medTTS(total) \end{aligned}$$

4.4.1 Resultado das Simulações

Nesta subseção são descritos quatro cenários da simulação, três usando o *SeVen* e uma do TAD:

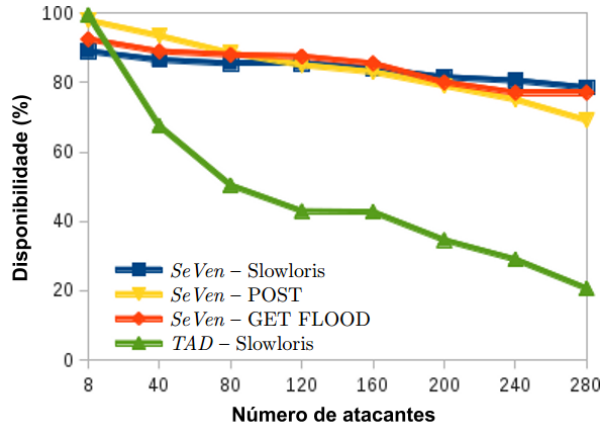
- **SeVen - GET FLOOD:** Simula o HTTP GET FLOOD quando aplicação usa a defesa *SeVen*.

- **SeVen - Slowloris:** Simula o slowloris quando aplicação usa a defesa *SeVen*.
- **SeVen - POST:** Simula o HTTP POST quando aplicação usa a defesa *SeVen*.
- **TAD - Slowloris:** Simula o slowloris quando aplicação usa a defesa TAD.

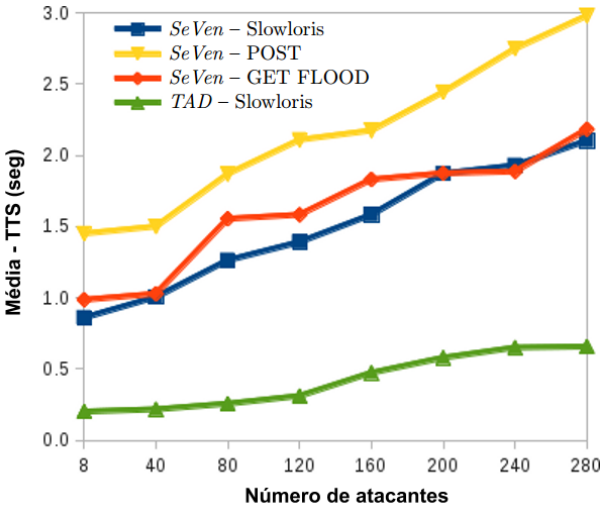
A Figura 4.1 contém os resultados das simulações em relação a disponibilidade, tempo médio de resposta (TTS) e a variação do tamanho do *buffer*. A Figura 4.1(a) exibe a disponibilidade do servidor quando a taxa de atacante aumenta. Para todos os três ataques, o desempenho do *SeVen* é semelhante, quando há 280 atacantes e 200 clientes a aplicação mantém um elevado nível de disponibilidade, ou seja, superior a 70%. Por outro lado, o TAD tem um nível de disponibilidade similar ou superior quando o número de atacantes é pequeno, mas, em seguida, cai rapidamente com o aumento de atacantes. Esse resultado já era esperado, pois, de acordo com [18], defesas baseadas em análise de tráfego não têm bom desempenho quando há um grande número de atacantes.

A Figura 4.1(b) exibe a média entre uma requisição de um cliente e uma resposta do servidor (TTS), variando o número de atacantes. Em todos os cenários, o *SeVen* tem TTS maior do que o TAD. Isso já era esperado, pois o *SeVen* apenas responde solicitações após *ts* segundos, enquanto o TAD responde imediatamente. Além disso, em todos os cenários com o *SeVen*, a medida que cresce o número de atacantes aumenta a média do TTS, o que parece razoável. O mesmo acontece com o TAD. Observe, no entanto, quando se utiliza o TAD o número de clientes que recebem confirmação (ACK) reduz drasticamente com o aumento de atacantes, como ilustrado na Figura 4.1(a) e o TTS é apenas calculado para os clientes com requisições bem sucedidas. Portanto, apesar do TAD ter TTS, em média, menor, o número de clientes de clientes processados é bem inferior.

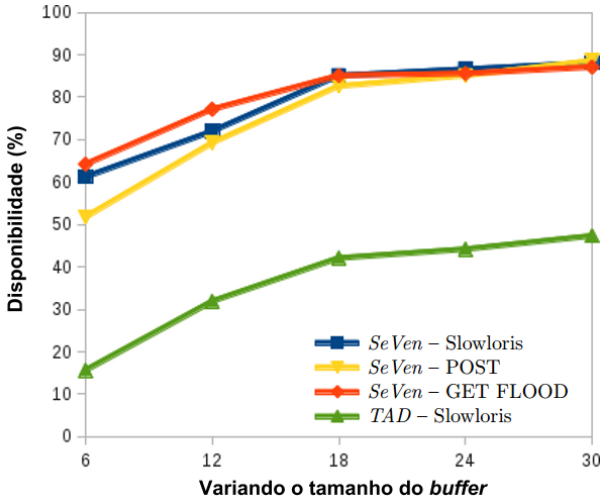
Por fim, a Figura 4.1(c) ilustra a relação de clientes bem sucedidos (clientes que receberam a confirmação do servidor) com o aumento do tamanho do buffer. Em todas as simulações, o número de atacantes é 280. Nos cenários usando o *SeVen*, a disponibilidade da aplicação cresce rapidamente de 50% para 80% quando o tamanho do buffer aumenta de 6 para 18 posições. Por outro lado, o TAD também aumenta sua disponibilidade, mas ainda permanece inferior, em torno de 40%. Esse gráfico pode sugerir uma possível configuração do tamanho do *buffer* de uma aplicação WEB.



(a) Disponibilidade do Servidor com $k = 12$.



(b) TTS, com $k = 12$.



(c) Disponibilidade do servidor variando o tamanho do buffer, com 280 atacantes.

Figura 4.1: Resultado das simulações dos ataques ao SeVen e TAD.

Capítulo 5

PROTÓTIPO DO SEVEN

Esta seção apresenta as características da arquitetura do protótipo do *SeVen*, com destaque para as visões de Camadas e de Processos. Além disso, uma discussão sobre duas formas distintas de integrar o *SeVen* em uma rede e os resultados dos experimentos.

5.1 Arquitetura – Visão das Camadas

A subseção apresenta a Figura 5.1, a qual ilustra uma abstração do esquema arquitetural do protótipo segmentado nas camadas: SeVen–Entrada, SeVen–Defesa, SeVen-Saída. Além disso, as camadas do Cliente e do Servidor são apresentadas para o melhor entendimento da arquitetura.

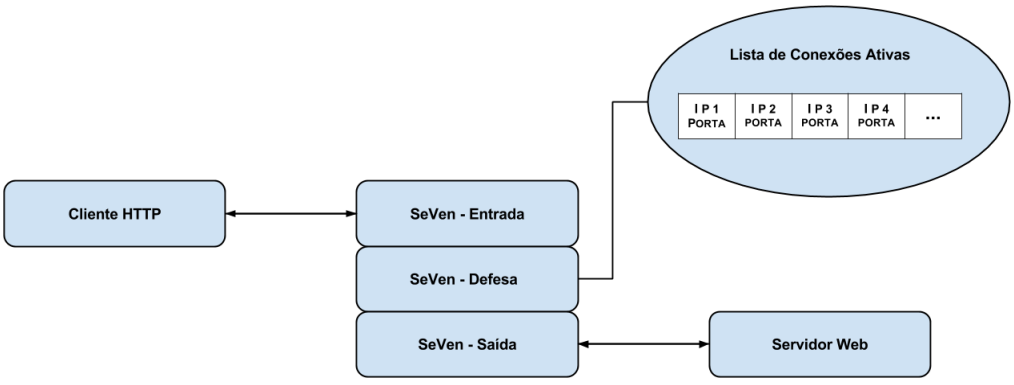


Figura 5.1: Estrutura de Camadas do SeVen

1. **Cliente HTTP:** Esta camada é responsável por iniciar a interação com as páginas WEB, i.e. enviar solicitações HTTP aos servidores WEB.

2. **SeVen – Entrada:** Esta camada trata todas as conexões destinadas ao Servidor WEB, ou seja, o *SeVen* funciona como um filtro para selecionar conexões e requisições HTTP que podem ou não ser encaminhadas ao Servidor WEB.

3. **SeVen – Defesa:** É nesta camada que está implementada toda a configuração da defesa proposta. Caso a lista de conexões ativas esteja disponível para receber a nova requisição, essa será encaminhada para a Camada **SeVen – Saída**, caso contrário, i.e. a lista esteja cheia, o *SeVen* possui duas opções:

   3.1. Descartar o pacote: Isto significa que as requisições armazenadas na lista não sofrem alterações.

   3.2. Aceitar o pacote: Isto significa que uma das requisições armazenadas na lista deve ser substituída pela nova requisição.

   O *SeVen* decide se mantém ou descarta uma requisição baseado na distribuição uniforme de probabilidade (U).

4. **SeVen – Saída:** Esta camada recebe as requisições provenientes da Camada **SeVen - Defesa** e abre novas conexões com o Servidor WEB. Nesse momento, o *SeVen* se torna um cliente do Servidor.

5. **Servidor Web:** Esta camada é responsável por processar as solicitações HTTP após a filtragem realizada pela defesa.

5.2 Arquitetura – Visão dos Processos

Esta subseção define o prótotipo em termos de processos ou threads que o controlam. A Figura 5.2 apresenta o diagrama de interação UML que representa os processos na arquitetura do SeVen.

De acordo com a Figura 5.2, o cliente inicia o processo realizando uma requisição HTTP através de alguma aplicação que permita o envio dessas requisições, i.e. *cURL, Siege, browser e etc*. No momento em que o pedido de conexão chega ao *SeVen*, é verificado se há espaço na memória para abrir o socket. Se houver, a conexão é aceita pelo *SeVen* e encaminhada para o módulo seguinte, caso contrário, a conexão é rejeitada pelo *SeVen*.

Posteriormente à abertura do socket, é verificado se há espaço disponível na lista das conexões ativas do *SeVen*, caso exista, o *SeVen* funciona como um *proxy* básico, ou seja, apenas encaminha a mensagem para o Servidor WEB. Caso contrário, é iniciada a estratégia do *SeVen* para verificar se a requisição HTTP deve ser aceita ou descartada pela defesa. Portanto, para checar essa condição é gerado um número aleatório de acordo com a distribuição uniforme de probabilidade (U), caso o *SeVen* decida descartar o pacote, uma mensagem de erro é enviada para cliente informando que não foi possível responder a solicitação, i.e, servidor indisponível (503). Caso opte por ficar com a requisição, baseado na mesma distribuição de probabilidade (U), é escolhido um usuário que deve ser removido da lista. Por fim, ao final da rodada, a requisição é encaminhada para o Servidor WEB, que por sua vez, processa a informação e envia sua resposta, a qual é encaminhada pelo *SeVen* ao cliente.

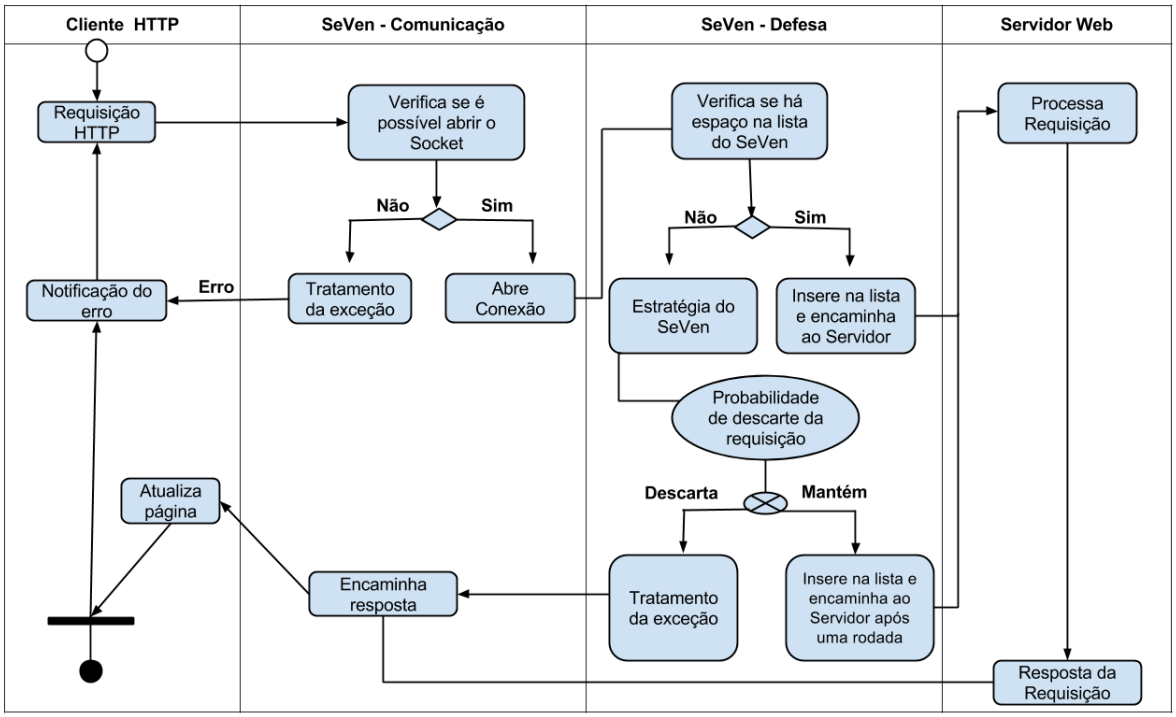


Figura 5.2: Estrutura dos Processos do SeVen

5.3 Integração do SeVen

O *SeVen* pode ser integrado em uma rede de duas formas distintas, conforme ilustrado na Figura 5.3.

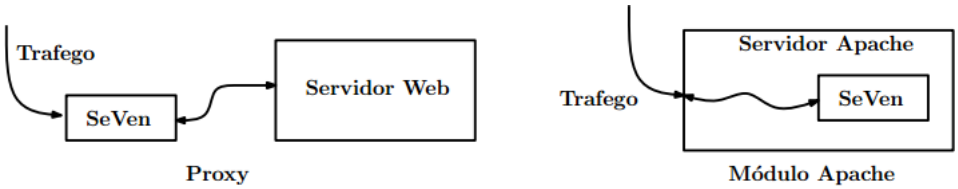


Figura 5.3: Propostas de Integração do SeVen

A primeira proposta, chamada de *Proxy*, a esquerda da Figura 5.3, usa o *SeVen* como uma aplicação *stand–alone*, em que *SeVen* funciona como um *Proxy* encaminhando pacotes para o Servidor Web e enviando respostas para a rede. Para isso, clientes acessam o *SeVen* diretamente enviando pacotes ao *SeVen*. A segunda proposta, chamada de Módulo Apache, a direita da Figura 5.3, implementa a estratégia do *SeVen* como um módulo Apache. Neste caso, clientes acessam diretamente o servidor Apache os quais serão tratados usando a estratégia do *SeVen* conforme descrito acima.

A vantagem da proposta de usar *SeVen* como um *Proxy* é a possibilidade de usar o mecanismo de defesa com qualquer servidor e não somente o servidor Apache. Uma desvantagem é que o *SeVen* não poderá ter acesso direto aos processos do servidor, perdendo em eficiência. Outra desvantagem pode ser o *overhead* devido o estabelecimento de novas conexões *SeVen*–Servidor. Contudo, os experimentos realizados neste trabalho mostram que o *overhead* causado foi irrisório.

A segunda proposta de usar o *SeVen* como um módulo Apache tem a desvantagem de ser específico para o servidor Apache. Contudo, existem algumas vantagens de implementar um módulo do Apache. Uma vantagem é a facilidade de implementação da defesa em redes que usam servidores Apache. Basta instalar o módulo *SeVen*, o que pode ser feito em poucos passos, incluindo a conguração do *SeVen*. Outra vantagem é no quesito de eficiência, quando comparado com a proposta *Proxy* descrita acima. Além de ter acesso direto aos processos executando no Apache e, portanto, possibilitando uma execução mais rápida, não se tem o *overhead* devido ao estabelecimento de novas conexões.

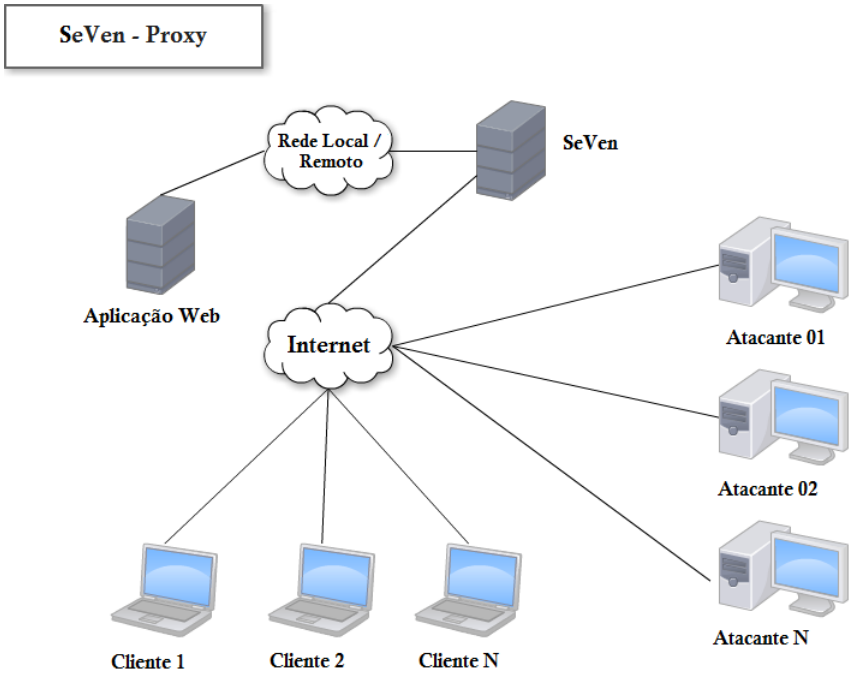


Figura 5.4: SeVen – Proxy

Neste trabalho, o protótipo do *SeVen* em C++ foi implementado como um *Proxy*, ilustrado com mais detalhes na Figura 5.4. A segunda proposta, Módulo apache, será um dos trabalhos futuros.

5.4 Experimentos na rede

Para consolidar a robustez do *SeVen* contra ataques de negação de serviço na camada de aplicação, foram realizados experimentos na rede com a ferramenta *SeVen* implementada em C++. Nesse cenário, o *SeVen* trabalha como um *proxy*, recebendo e encaminhando as mensagens para o servidor WEB, onde sua estratégia é apenas acionada quando o *buffer* de requisições está cheio.

A configuração das máquinas usadas nos experimentos são descritas a seguir:

- **Servidor:** Foi utilizado o servidor *WEB Apache* 2.4 em uma máquina com o sistema operacional *Ubuntu*, com processador Intel Xeon E24XX 2.50GHz e 12GB de memória.

- **Clientes:** Uma máquina com sistema operacional *Debian*, processador Intel Xeon 2.50GHz com 4GB de memória.
- **Atacantes:** Duas máquinas com sistema operacional *Ubuntu*, ambas com 4GB de memória, uma com processador Intel i3 2.40GHz e outra com Core2 Duo 2.16GHz

Os experimentos foram configurados de acordo com os seguintes parâmetros, que podem ser configurados no servidor *Apache* e/ou nas ferramentas disponíveis para a realização de ataques:

- **Sockets:** Número de sockets disponíveis no servidor WEB, ou seja, o número de requisições que podem ser respondidas simultaneamente. Em todos os experimentos, o servidor *apache* foi configurado para suportar até 200 conexões, o mesmo vale para o buffer do *SeVen*, o que responde a um servidor WEB de pequeno porte.
- **Timeout da Aplicação:** Ao instalar o apache 2.4, o valor padrão do *timeout* é de 300 segundos, isto significa que, se a aplicação não receber qualquer requisição de um usuário em 300 segundos, então, a conexão do usuário é encerrada, e ele é removido dos *buffer* da aplicação. Esse valor é considerado alto, principalmente no contexto de ataques de negação de serviço. Portanto, com a finalidade de tornar a aplicação mais robusta contra ataques DDoS, seguiu-se a sugestão apresentada em [45] e o *timeout* foi configurado para 40 segundos.
- **Clientes:** Para simular o tráfego dos clientes foi utilizado o *Siege* [46]. Uma ferramenta que permite realizar testes de carga em aplicações Web. Os clientes são gerados com uma taxa de 10 requisições/segundo, ou seja, 50 requisições a cada 5 segundos, com o objetivo de ocupar 1/4 do *buffer* da aplicação.
- **Atacantes:** Para realizar os ataques Slowloris e POST foram utilizadas as ferramentas Slowloris [15] e Switchblade [47], respectivamente. Os atacantes são gerados com um taxa de 7.14 requisições/segundo, ou seja, 250 requisições a cada 35 segundos. Como mencionado anteriormente, são utilizadas duas máquinas, cada uma gerando 125 requisições a cada 35 segundos.
- **SeVen Round Time - ts:** Tempo que o *SeVen* acumula requisições. Nos experimentos, *ts* = 100 milissegundos.
- **Tempo total dos experimentos:** Tempo total dos experimentos foi de 240 segundos.

Para determinar a eficiência do mecanismo de defesa proposto, são realizados ataques com e sem a utilização do *SeVen*. São usadas duas métricas, *Disponibilidade* e *Média TTS*, ambas descritas na seção 4.4.

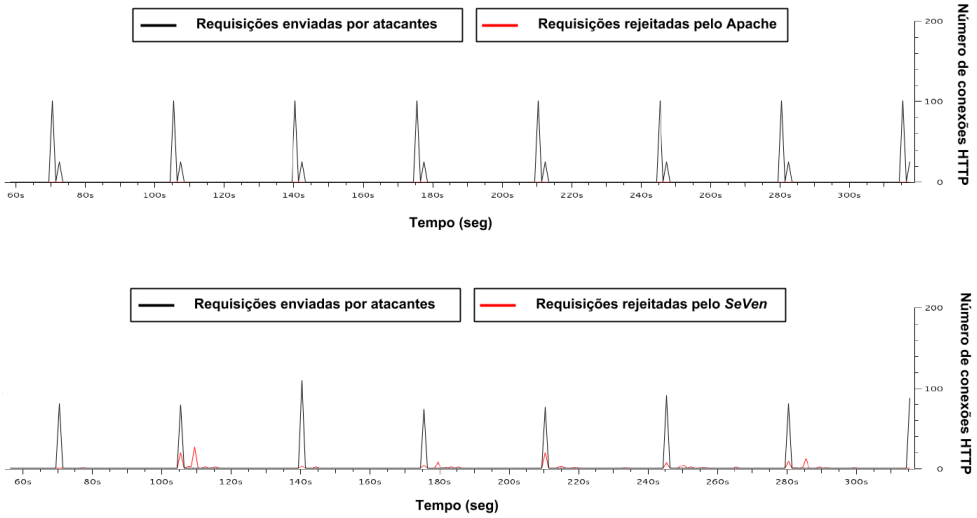
5.4.1 Resultados dos Experimentos

Esta seção apresenta um real cenário de ataque de negação de serviço distribuído na camada de aplicação com os ataques Slowloris e POST. Uma abstração do esquema do ataque é ilustrado na Figura 5.5, onde a aplicação WEB e a defesa *SeVen* foram implantadas na cidade de Vitória e os ataques foram realizados de João Pessoa. Além disso, o tráfego do cliente também foi gerado de João Pessoa. Um fato importante foi que nenhum enlace da rede Ipê [48] entre João Pessoa e Vitória detectou o ataque, o que já era previsto, pois os ataques Slowloris e POST exploram o protocolo HTTP sem a necessidade de gerar um tráfego volumoso.

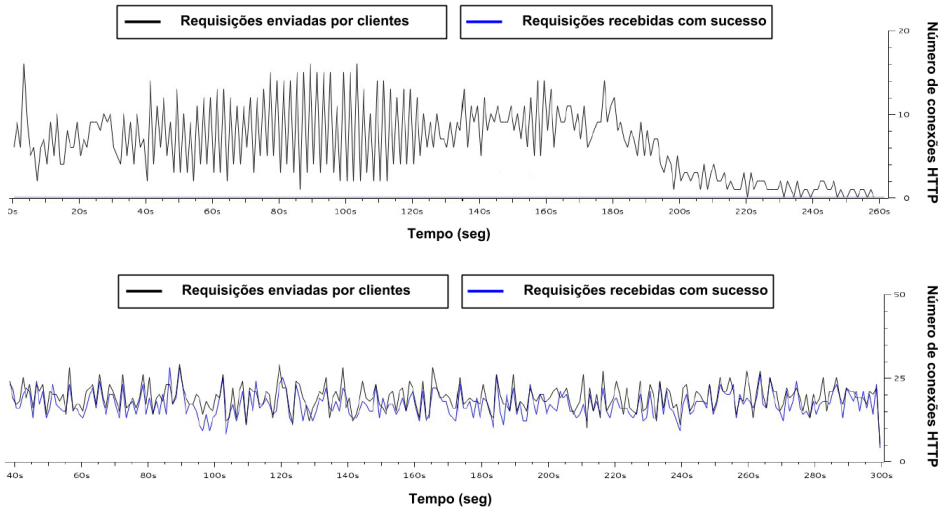
Figura 5.5: Abstração do esquema de DDoS realizado nos experimentos

Para compreender melhor o comportamento dos ataques, as Figuras 5.6 e 5.7 ilustram os tráfegos dos clientes e atacantes capturados pela ferramenta *Wireshark* para os ataques Slowloris e POST, respectivamente. Cada ataque possui quatro gráficos: Os gráficos apresentados em a) correspondem ao tráfego dos atacantes e os em b) apresentam os tráfegos dos clientes. Tanto em a), como em b), o gráfico de cima ilustra o cenário sem defesa e a figura de abaixo corresponde o cenário usando o *SeVen*. As linhas pretas correspondem ao número total de requisições enviadas ao servidor WEB, as linhas vermelhas correspondem as solicitações dos atacantes rejeitadas pelo servidor WEB e as linhas azuis são as solicitações de clientes realizadas com sucesso.

Como foi visto no Capítulo 2, os ataques Slowloris e POST têm o comportamento similar no sentido de usar uma taxa de tráfego similar ao do cliente. Neste experimento, a taxa do ataque foi de 7.14 requisições/segundo e do cliente foi de 10 requisições/segundo. Além disso, a ideia é sempre enviar requisições próximo ao *timeout* da aplicação, como pode ser observado nas Figuras 5.6(a) e 5.7(a).



(a) Tráfego dos Atacantes – Ataque Slowloris



(b) Tráfego dos Clientes – Ataque Slowloris

Figura 5.6: Tráfego dos atacantes/clientes para o Ataque Slowloris

Considere os gráficos do ataque Slowloris mostrados na Figura 5.6. É possível constatar que quando o *SeVen* não está sendo executado, os pacotes dos atacantes não são rejeitados, ou seja, os atacantes conseguem permanecer alocados no *buffer* da aplicação. Por outro lado, quando o *SeVen* está sendo executado, existe uma taxa de requisição de pacotes rejeitados dos atacantes, permitindo que os clientes também tenham acesso à aplicação.

Em relação aos gráficos dos clientes, é possível observar que quando o *SeVen* não está sendo utilizado, o gráfico é irregular. Isso ocorre porque muitos clientes não são capazes de realizar o *three way handshake*, ou seja, SYN-ACK, com o servidor WEB. Portanto, esses clientes não são capazes de enviar requisições GET. Além disso, o número de solicitações com sucesso (linha azul) recebidas é igual a zero, ou seja, sem o *SeVen*, tanto o *Slowloris*, quanto o POST conseguem negar completamente o serviço para

Tabela 5.1: Resumo dos Experimentos

	Sem o *SeVen*		Com o *SeVen*	
	Disponibilidade	TTS	Disponibilidade	TTS
Sem ataque	100.0%	0.01s	100.0%	0.03s
POST	0.0%	∞	97.25%	0.05s
Slowloris	0.0%	∞	94.47%	0.06s

os clientes legítimos. Em contrate, quando o *SeVen* está sendo executado, os clientes consegue realizar o *three way handshake* com o servidor WEB e, portanto, o gráfico é mais regular. Além disso, as solicitações recebidas com sucesso são de aproximadamente 95%.

A primeira linha da Tabela 5.1 apresenta o possível *overhead* causado pelo *SeVen* implementado como *proxy*. Neste cenário, nenhum ataque está sendo executado, apenas clientes estão enviando solicitações HTTP à aplicação WEB. É possível observar que há uma diferença de TTS, entre os cenários Sem *SeVen* e com *SeVen*, de apenas 0.02 segundos. Um valor considerado irrisório para os clientes WEB. Além disso, em ambos os cenários, a disponibilidade é de 100%, o que já era esperado, pois a aplicação WEB não está sofrendo um ADDoS. Sendo assim, é possível concluir que o *overhead* acarretado pelo *SeVen-Proxy* é insignificante.

As linhas dois e três da Tabela 5.1 resumem os resultados da taxa de disponibilidade do servidor WEB e do TTS em relação aos cenários descritos anteriormente. É possível notar que quando o *SeVen* não está sendo utilizado a disponibilidade é de 0% e, consequentemente, o TTS é infinito. É importante ressaltar que a ferramenta *Siege* foi configurada para receber respostas em até 5 segundos, ou seja, caso uma resposta do servidor chegue ao cliente depois de 5 segundos, a mesma não é computada. Por outro lado, nos cenários em que o *SeVen* é usado, praticamente todos os clientes têm acesso à aplicação. No POST a disponibilidade foi de 97.25% e no Slowloris de 94.47%. Além disso, o tempo de espera para receber uma página solicitada (TTS) foi mínimo.

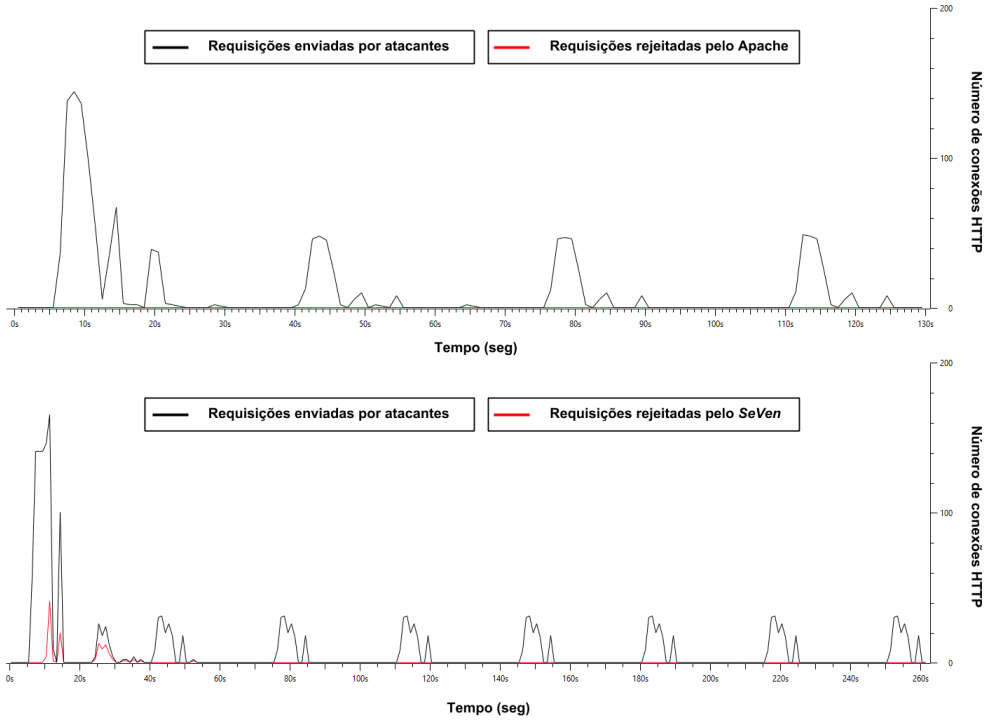
Além da disponibilidade e TTS, também foi verificado o impacto do *SeVen* em relação a memória e processamento, como pode ser visto na Tabela 5.2. Para isso, foi criado um *shell script* para checar, a cada segundo, a porção de memória e CPU utilizada pelo processo (PID) do *SeVen*. Dessa forma, foram investigados dois cenários com a mesma configuração dos experimentos anteriores: 1) Apenas clientes legítimos enviando solicitações HTTP sem sobrecarregar a aplicação. Neste cenário, o *SeVen* usou, em média, 0.5% de memória de um total de 12GB, o que significa 61.44MB. Além disso, ocupou 0.9% de um processador Intel Xeon E24XX com 6 núcleos. 2) Cenário apresentado na Tabela 5.1 em que clientes e atacantes (Slowloris) estão competindo pelos recursos da aplicação. Como o processamento é maior quando o *buffer* está cheio, o *SeVen* usou o dobro de memória em relação ao cenário anterior, ou seja, 122.88MB. Além disso, ocorreu um aumento de 0.9% para 1% no uso da CPU.

Tabela 5.2: Impacto da Memória e Processamento

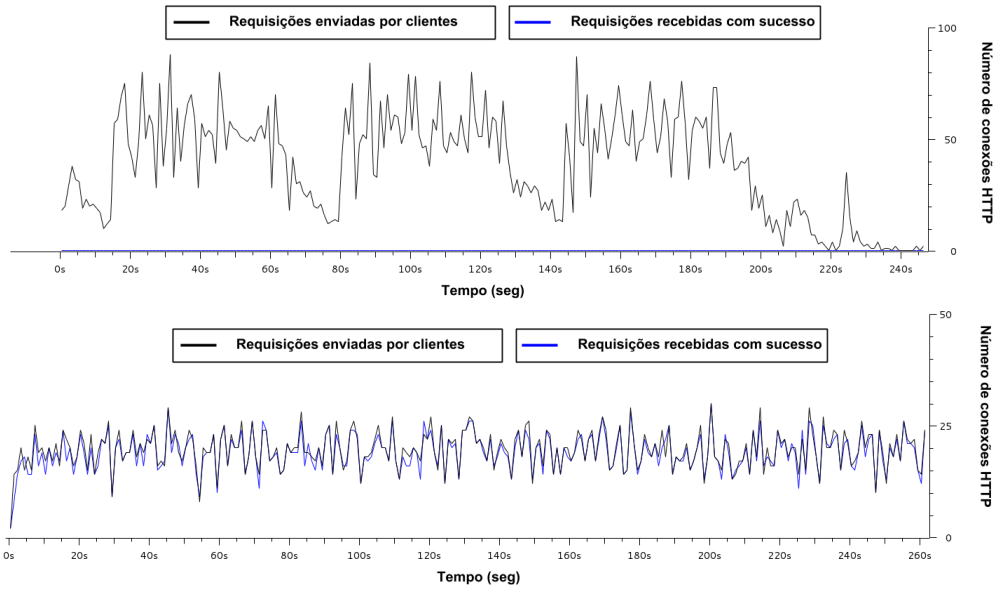
	Sem Ataque		Com Ataque	
	Memória	CPU	Memória	CPU
Com SeVen	0.5%	0.9%	1%	1%

Apesar da otimização não ter sido o foco principal deste trabalho, o *SeVen* usou entre 0.5% e 1% de memória disponível em um servidor com um total de 12GB e aproximadamente 1% de um processador Intel Xeon E24XX com 6 núcleos. Tais valores parecem indicar um bom gerenciamento de *threads* e *sockets*. No entanto, é apenas uma análise preliminar do impacto do *SeVen* em relação a memória e processamento, logo a otimização da defesa é um dos tópicos para trabalhos futuros.

Por fim, a variação do tempo da rodada também foi avaliada. Essa variação influencia diretamente no tempo de resposta de uma requisição, o que é evidente, pois se uma requisição chega no início de uma rodada, o *SeVen* retém a requisição e, somente no final da rodada encaminha para a aplicação WEB. Foram analisados valores, em segundos, no intervalo $[0.1, 0.5]$. Com o parâmetro de 0.1 segundos (100 milissegundos) obteve-se os melhores resultados, tanto em relação a disponibilidade, quanto ao TTS. Ao aumentar gradativamente o tempo de rodada até 0.5s, o tempo de resposta cresceu em média e a disponibilidade não foi afetada de uma maneira substancial.



(a) Tráfego dos Atacantes – Ataque POST



(b) Tráfego dos Clientes – Ataque POST

Figura 5.7: Tráfego dos atacantes/clientes para o Ataque POST

Capítulo 6

CONSIDERAÇÕES FINAIS

Este trabalho apresentou uma nova estratégia para tratamentos de ataques DDoS na camada de aplicação (ADDoS), chamada de *SeVen*, baseada na defesa ASV [12] para ataques DDoS na camada de transporte. Foram utilizadas duas abordagens para validar a defesa: 1) Simulação: Todo o mecanismo de defesa foi formalizado usando a ferramenta *Maude* e simulado usando um modelo estatístico (*PVeStA*). 2) Experimentos na rede: Análise da eficiência de *SeVen*, implementado em C++, em um experimento real na rede. Em seguida, foi demonstrado que o *SeVen* pode ser utilizado para atenuar uma série de ataques, incluindo o Slowloris e o HTTP POST. Por exemplo, a seção de resultados do Capítulo 5 mostrou que o *SeVen* consegue mitigar um ataque Slowloris elevando o índice de disponibilidade de 0%, em um cenário sem defesa, para aproximadamente 95%.

Existem outras defesas para ADDoS, em sua maioria utilizando o padrão do ataque como parâmetro, ou seja, tempo de envio de mensagens, localização do IP e etc. Existem modelos baseados em técnicas de aprendizado de máquina, tais como Neuro-Fuzzy [18], modelos usando Cadeias de Markov [20] ou hard-wired [17]. Nenhum deles, no entanto, foi formalmente verificado, apenas validado utilizando experimentos reais na rede com um número pequeno de atacantes. Em [18], os autores mencionam que defesas baseadas em análise de tráfego funcionam apenas quando há um pequeno número de atacantes. Isto pode ser visto nas simulações deste trabalho, quando foi utilizada uma defesa baseada em [17].

A formalização de ataques e defesas DDoS tem sido o foco de outros trabalhos. Por exemplo, Meaddows propôs um modelo baseado em custos [49], enquanto outros usam lógica temporal [18]. Este trabalho segue a abordagem utilizada em [50] [42] [43], onde formalizou-se todo o sistema em *Maude* e usa um modelo estatístico para fazer as simulações. No entanto, sua formalização baseou-se numa comunicação steteless entre cliente e servidor, típico do DDoS na camada de transporte. O *SeVen* foi formalizado com uma noção de estado, pois seu comportamento irá depender do número de bytes processados, fato importante para mitigar ataques HTTP POST. Além disso, para consolidar a robustez do *SeVen* contra ataques de negação de serviço na camada de aplicação, foram realizados vários experimentos na rede com a ferramenta *SeVen* implementada em C++. Nos experimentos, o *SeVen*

trabalhou como um *proxy* recebendo e encaminhando mensagens para o apache. Em ambas abordagens, o *SeVen* obteve um elevado índice de disponibilidade, sendo surpreendente, pois não foi necessário realizar nenhuma configuração específica para um ataque particular.

Para trabalhos futuros, está sendo desenvolvido um módulo do *SeVen* para o apache. Com o módulo, será mais simples introduzir o *SeVen* em redes que usam servidores Apache. Além disso, ao utilizar o módulo apache, espera-se uma eficiência maior em comparação ao *SeVen-Proxy*. Outro tópico importante é investigar e testar a real eficiência das instâncias do *SeVen* apresentadas na seção 3.2.

Uma possível extensão do *SeVen* é utiliza-lo como um sensor nas redes SDN (Software Defined Networking) [51]. Pois além de proteger as aplicações contra os ataques de negação de serviço, o *SeVen* pode trabalhar em conjunto com o controlador, informando-o a taxa de ocupação dos *buffers* das aplicações. E o controlador, por sua vez, pode iniciar alguma estratégia, como, por exemplo, o algoritmo de Round-Robin [52], com o objetivo de evitar a sobrecarga em uma aplicação ou em sua rede interna.

Por fim, também está sendo investigado o uso do *SeVen* para outros ADDoS. Por exemplo, está sendo pesquisado um ataque DDoS que tem como alvo a infra-estrutura do VoIP, esse ataque explora o protocolo SIP. Logo, para esse tipo de serviço, é necessário o uso de distribuições de probabilidades mais específicas para determinar se uma solicitação é descartada ou não. Uma possível estratégia poderia ser o *SeVen-Average* apresentado em 3.2.1.2, no entanto, é preciso investigar melhor sua utilização nos protocolos SIP. Também está sendo verificado o uso do *SeVen* para os ataques de amplificação, como o NTP [53]. Além disso, é preciso otimizar o código do protótipo do *SeVen* e realizar experimentos na rede com os ataques GET FLOOD.

REFERÊNCIAS

[1] Gaogang Xie Chuan Xu, Guofeng Zhao and Shui Yu. Detection on application layer ddos using random walk model. *IEEE Communication and Information Systems Security Symposium*, 2014.

[2] *Malware-infected home routers used to launch DDoS attacks*. http://www.helpsec.net/malware-infected-home-routers-used-to-launch-ddos-attacks – Acesso em 21 de Abril de 2015.

[3] *Operation Payback cripples MasterCard site in revenge for WikiLeaks ban, 2010*. http://www.theguardian.com/media/2010/dec/08/operation-payback-mastercard-website-wikileaks – Acesso em 01 de Dezembro de 2014.

[4] *DDoS: Lessons from Phase 2 Attacks, 2013*. http://www.bankinfosecurity.com/ddos-attacks-lessons-from-phase-2-a-5420/op-1 – Acesso em 03 de Dezembro de 2014.

[5] L. Dave. *Global Internet slow after "biggest attack in history", 2013*. http://www.bbc.com/news/technology-21954636 – Acesso em 27 de Novembro de 2014.

[6] *T. Socolofsky; C. Kale. A - TCP/IP Tutorial – RFC 1180*. https://tools.ietf.org/html/rfc1180 – Acesso em 21 de Novembro de 2014.

[7] NSFOCUS. *Mid-Year DDoS Threat Report, 2013*. http://www.nsfocus.com/SecurityReport/2013– Acesso em 15 de Novembro de 2014.

[8] Anonymous. *A network stress testing and denial-of-service attack application, 2009*. http://sourceforge.net/projects/loic/ – Acesso em 23 de Novembro de 2014.

[9] Ronen Kenig. *Why low & slow DDoS application attacks are difficult to mitigate, 2013*. http://blog.radware.com/security/2013/06/why-low-slow-ddosattacks-are-difficult-to-mitigate/ – Acesso em 25 de Novembro de 2014.

[10] S. Jha E. Clarke and W. Marrero. Using state space exploration and a naturaldeduction style message derivation engine to verify security protocols. *IFIP Working Conference on Programming Concepts and Methods (PROCOMET)*, 1998.

[11] John Mitchell Nancy Durgin, Patrick Lincoln and Andre Scedrov. Multiset rewriting and the complexity of bounded security protocols. *Journal of Computer Security*, 2004.

[12] Sanjeev Khanna, Santosh S. Venkatesh, Omid Fatemieh, Fariba Khan, and Carl A. Gunter. Adaptive selective verification: An efficient adaptive countermeasure to thwart dos attacks. *IEEE/ACM Trans. Netw.*, 20(3):715–728, 2012.

[13] Musab AlTurki and José Meseguer. Pvesta: A parallel statistical model checking and quantitative analysis tool. In *CALCO*, pages 386–392, 2011.

[14] Yi Xie and Shun-Zheng Yu. Monitoring the application-layer ddos attacks for popular websites. *IEEE/ACM Trans. Netw.*, 17(1):15–25, 2009.

[15] Anonymous. *Slowloris Tool, 2013*. http://ha.ckers.org/slowloris/ – Acesso em 25 de Novembro de 2014.

[16] *r-u-dead yet, 2013*. https://code.google.com/p/r-u-dead-yet/ – Acesso em 18 de Novembro de 2014.

[17] Leandro C. de Almeida. Ferramenta computacional para identificação e bloqueio de ataques de negação de serviço em aplicações web. Master Thesis, 2013.

[18] Ajay Mahimkar and Vitaly Shmatikov. Game-based analysis of denial-of-service prevention protocols. In *CSFW*, pages 287–301, 2005.

[19] Marin Litoiu Chris Bachalo Mark Shtern, Roni Sandel and Vasileios Theodorou. Towards mitigation of low and slow application ddos attacks. *IEEE International Conference on Cloud Engineering*, 2014.

[20] Saman Taghavi Zargar, James Joshi, and David Tipper. A survey of defense mechanisms against distributed denial of service (DDoS) flooding attacks. *IEEE Communications Surveys and Tutorials*, 15(4):2046–2069, 2013.

[21] Sanjeev Khanna, Santosh S. Venkatesh, Omid Fatemieh, Fariba Khan, and Carl A. Gunter. Adaptive selectiveverification. In *INFOCOM*, pages 529–537, 2008.

[22] Radware ERT-Team. *Security Report 2014, details are commentted in*. http://blog.radware.com/events/2014/02/my-perspective-e-crime-congress/ – Acesso em 22 de Novembro de 2014.

[23] *RFC – IRC – Session Initiation Protocol*. https://www.ietf.org/rfc/rfc3261.txt – Acesso em 11 de Junho de 2015.

[24] Pierluigi Paganini. Http-botnets: The dark side of an standard protocol! http://securityaffairs.co/wordpress/13747/cyber-crime/http-botnets-the-dark-side-of-an-standard-protocol.html. 2013.

[25] *mIRC tool*. http://www.mirc.com/ – Acesso em 21 de Junho de 2015.

[26] *Wireshark*. https://www.wireshark.org/ – Acesso em 18 de Janeiro de 2015.

[27] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *2nd Workshop on Concurrency and Compositionality*, 1992.

[28] J.; TALCOTT C. DENKER, G.; MESEGUER. Protocol specification and analysis in maude. *Workshop on Formal Methods and Security Protocols*, 1998.

[29] J.; TALCOTT C. DENKER, G.; MESEGUER. Formal specication and analysisof active networks and communication protocols: The maude experience. *Darpa Information Survivability Conference and Exposition*, 2000.

[30] Nancy A. Durgin, Patrick Lincoln, John C. Mitchell, and Andre Scedrov. Multiset rewriting and the complexity of bounded security protocols. *Journal of Computer Security*, 12(2):247–311, 2004.

[31] A.S.R. Chadha and M. Kanovich. Inductive methods and contract-signing protocols. *8a ACM Conference on Communications and Security*, 2001.

[32] M. et al. CLAVEL. Maude manual (version 2.1), 2004.

[33] *The Maude System*. http://maude.cs.illinois.edu/ – Acesso em 18 de Junho de 2015.

[34] Adrian Riesco and Alberto Verdejo. Parameterized skeletons in maude. 2007.

[35] *RFC – SIP – Session Initiation Protocol*. https://www.ietf.org/rfc/rfc3261.txt – Acesso em 01 de Junho de 2015.

[36] *Visão geral da QoS (Qualidade de Serviço*. https://technet.microsoft.com/pt-br/library/hh831679.aspx – Acesso em 13 de Junho de 2015.

[37] Edgard Jamhour. Qualidade de serviços em redes ip. 2010.

[38] *Apache Module mod_reqtimeout*. https://httpd.apache.org/docs/trunk/mod/mod_reqtimeout.html – Acesso em 16 de Junho de 2015.

[39] *Hypertext Transfer Protocol – HTTP/1.1 – Method Definitions*. http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html – Acesso em 18 de Junho de 2015.

[40] *The Definitive Guide to GET vs POST*. http://blog.teamtreehouse.com/the-definitive-guide-to-get-vs-post – Acesso em 18 de Junho de 2015.

[41] Jonas Eckhardt. Security analysis in cloud computing using rewriting logic. *Master Thesis*, 2014.

[42] Jonas Eckhardt, Tobias Mühlbauer, Musab AlTurki, José Meseguer, and Martin Wirsing. Stable availability under denial of service attacks through formal patterns. In *FASE*, pages 78–93, 2012.

[43] Jonas Eckhardt, Tobias Mühlbauer, José Meseguer, and Martin Wirsing. Statistical model checking for composite actor systems. In *WADT*, pages 143–160, 2012.

[44] Yuri Gil Dantas, Vivek Nigam, and Iguatemi Fonseca. A selective defense for application layer ddos attacks. In *ISI-EISIC*, 2014. http://www.nigam.info/docs/ddos.pdf.

[45] *Optimize Apache for WordPress*. https://thethemefoundry.com/blog/optimize-apache-wordpress/ – Acesso em 29 de Junho de 2015.

[46] *Siege tool*. https://www.joedog.org/siege-home/ – Acesso em 21 de Junho de 2015.

[47] *OWASP Switchblade*. https://www.owasp.org/index.php/OWASP_HTTP_Post_Tool – Acesso em 21 de Junho de 2015.

[48] *Panorama geral dos enlaces da rede Ipê*. http://memoria.rnp.br/ceo/trafego/panorama.php – Acesso em 18 de Junho de 2015.

[49] Catherine Meadows. A formal framework and evaluation method for network denial of service. In *CSFW*, pages 4–13, 1999.

[50] Musab AlTurki, José Meseguer, and Carl A. Gunter. Probabilistic modeling and analysis of dos protection for the asv protocol. *Electr. Notes Theor. Comput. Sci.*, 234:3–18, 2009.

[51] *7 Essentials Of Software-Defined Networking*. http://www.networkcomputing.com/networking/7-essentials-of-software-defined-networking/d/d-id/898899 – Acesso em 29 de Junho de 2015.

[52] *round robin – OS*. http://whatis.techtarget.com/definition/round-robin – Acesso em 29 de Junho de 2015.

[53] Matthew Prince. Technical details behind a 400Gbps NTP amplification DDoS attack, http://blog.cloudflare.com/technical-details-behind-a-400gbps-ntp-amplification-ddos-attack. 2013.