UNIVERSIDADE FEDERAL DA PARAÍBA

CENTRO DE INFORMÁTICA

PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

SHADE: UMA ESTRATÉGIA SELETIVA PARA MITIGAR ATAQUES DDOS NA CAMADA DE APLICAÇÃO EM REDES DEFINIDAS POR SOFTWARE

JOÃO HENRIQUE GONÇALVES CORRÊA

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

João Pessoa – PB

Julho/2017

UNIVERSIDADE FEDERAL DA PARAÍBA

CENTRO DE INFORMÁTICA

PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

SHADE: UMA ESTRATÉGIA SELETIVA PARA MITIGAR ATAQUES DDOS NA CAMADA DE APLICAÇÃO EM REDES DEFINIDAS POR SOFTWARE

JOÃO HENRIQUE GONÇALVES CORRÊA

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
Orientador: Prof. Dr. Vivek Nigam

João Pessoa – PB

Julho/2017

C824s Corrêa, João Henrique Gonçalves.
Shade: uma estratégia seletiva para mitigar ataques DDoS na camada de aplicação em redes definidas por software / João Henrique Gonçalves Corrêa. - João Pessoa, 2017.
80 f. : il. -

Orientador: Vivek Nigam.
Dissertação (Mestrado) - UFPB/CT

1. Ciência da Computação. 2. Lista branca dinâmica. 3. Negação de Serviço. 4. *Get Flooding*. I. Título.

UFPB/BC CDU: 007(043)

UNIVERSIDADE FEDERAL DA PARAÍBA
CENTRO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

Ata da Sessão Pública de Defesa de Dissertação de Mestrado de João Henrique Gonçalves Correa, candidato ao título de Mestre em Informática na Área de Sistemas de Computação, realizada em 27 de julho de 2017.

Aos vinte e sete dias do mês de julho, do ano de dois mil e dezessete, às oito e meia, no Centro de Informática da Universidade Federal da Paraíba, em Mangabeira, reuniram-se os membros da Banca Examinadora constituída para julgar o Trabalho Final do Sr. João Henrique Gonçalves Correa, vinculado a esta Universidade sob a matrícula nº 2015105192, candidato ao grau de Mestre em Informática, na área de "*Sistemas de Computação*", na linha de pesquisa "*Sinais, sistemas digitais e gráficos*", do Programa de Pós-Graduação em Informática, da Universidade Federal da Paraíba. A comissão examinadora foi composta pelos professores: Vivek Nigam (PPGI-UFPB), Orientador e Presidente da Banca, Iguatemi Eduardo Da Fonseca (PPGI-UFPB), Co-Orientador e Examinador Interno, Gustavo Henrique Matos Bezerra Motta (PPGI-UFPB), Examinador Interno, e Moises Renato Nunes Ribeiro (UFES), Examinador Externo à Instituição. Dando início aos trabalhos, o Presidente da Banca, cumprimentou os presentes, comunicou aos mesmos a finalidade da reunião e passou a palavra ao candidato para que o mesmo fizesse a exposição oral do trabalho de dissertação intitulado "Shade: uma estratégia seletiva para mitigar ataques DDOS na camada de aplicação em redes definidas por software". Concluída a exposição, o candidato foi arguido pela Banca Examinadora que emitiu o seguinte parecer: "***aprovado***". Do ocorrido, eu, Clauirton de Albuquerque Siebra, Coordenador do Programa de Pós-Graduação em Informática, lavrei a presente ata que vai assinada por mim e pelos membros da banca examinadora. João Pessoa, 27 de Julho de 2017.

Prof. Dr. Clauirton de Albuquerque Siebra

Clauirton de Albuquerque Siebra
Coordenador do Programa de Pós-Graduação em Informática
SIAPE 1723491

Prof. Dr. Vivek Nigam
Orientador (PPGI-UFPB)

Prof. Dr. Iguatemi Eduardo Da Fonseca
Co-Orientador (PPGI-UFPB)

Prof. Dr. Gustavo Henrique Matos Bezerra Motta
Examinador interno (PPGI-UFPB)

Prof. Dr. Moises Renato Nunes Ribeiro
Examinador Externo à Instituição (UFES)

AGRADECIMENTOS

Primeiramente a Deus, pela oportunidade de seus caminhos.

À Maria Cristina, minha esposa, por ter me acompanhado nessa caminhada e me apoiado em todas as minhas decisões.

À Ana e Pedro, meus pais, pelos seus conselhos, ensinamentos valiosos, e todo amor incondicional.

Aos meus irmãos, Filipe, José Pedro e Gustavo, que me acompanharam desde o início da minha vida.

Aos meus amigos, Alan, Gil, Michael, Nadja, Guto, Isis, Arthur, Fernanda e Wal, pela amizade sincera, por me indicarem, seja diretamente ou indiretamente, os melhores caminhos nas minhas dúvidas.

Aos meus orientadores Dr. Vivek Nigam e Dr. Iguatemi Eduardo Fonseca, pela oportunidade de ser orientado por vocês e por todos os ensinamentos na área acadêmica.

A todos do LAR-UFPB, pela ajuda e discussões.

A todos do Laboratório NERDS, da UFES, pelo aprendizado primordial para esse estudo.

Ao Prof. MSc. Leandro Cavalcanti de Almeida, pelo incentivo ao mestrado.

A todos os professores do IFPB - Campus Campina Grande, que me apoiaram durante a realização do mestrado e contribuíram nas discussões inerentes a atividade desenvolvida no campus.

"Se fosse fácil achar o caminho das pedras, tantas pedras no caminho não seria ruim."

Humberto Gessinger

RESUMO

O processo de evolução dos computadores chegou a tal ponto que informações, antes distantes, se encontram a um clique do usuário. Com o desenvolvimento da tecnologia, houve também a evolução de mecanismos que atentam contra a segurança da informação, inclusive quanto a disponibilidade de uma determinada informação. Os ataques de negação de serviço (DoS) ou sua forma distribuída (DDoS) estão entre os que mais impactam nos negócios, afetando a disponibilidade do serviço. Ataques de negação de serviço na camada de aplicação exploram características existentes nos protocolos da referida camada (HTTP e SIP). Um grande desafio na mitigação deste tipo de ataque é o fato que as requisições de atacantes tem igual validade a de clientes legítimos. Este trabalho propõe o uso de técnicas de identificação de assinaturas para a manutenção de listas brancas dinâmicas, a fim de mitigar ataques de negação de serviço na camada de aplicação. Essas assinaturas permitem a identificação mais precisa de um usuário dentro de um fluxo de rede por meio da monitoração de parâmetros, como sistema operacional, navegador, IP utilizados. A efetividade da técnica foi demonstrada por meio de experimentos realizados na rede, em que sem o uso da técnica, uma Aplicação *web* só conseguiu atender 17% dos clientes durante um ataque Get-Flooding, enquanto que com a utilização da lista branca dinâmica a disponibilidade subiu para mais de 99%.

Palavras-chave: Lista branca dinâmica, Negação de Serviço, *Get Flooding*

ABSTRACT

The process of evolution of computers has reached the point that information before distant, are just a user click away. With the development of technology, there have also been developments in mechanisms that threaten the security of information, including the availability of certain information. The denial of Service (DoS) or a distributed form (DDoS) attacks are among the most important business impacts nowadays, affecting service availability severely. Application Layer Denial of Service attacks exploit vulnerabilities protocols, such as HTTP and SIP protocols. The main challenge in mitigating such attacks is due to the fact that attacker requests have the same status as legitimate clients. This paper proposes the use of fingerprinting techniques for dynamic whitelist in order to mitigate denial of service attacks at the application layer. Fingerprinting allows for more accurate identification of a user within a network flow by monitoring parameters such as operating system, browser and IP used. The effectiveness of the technique was demonstrated through experiments carried out in a controlled network, in which, without the use of the technique, a web Application was only able to serve 17% of the clients during a Get-Flooding attack, whereas with the use of the dynamic whitelist the availability rose to more than 99%.

Keywords: Dynamic Whitelist, Denial of Service, Get Flooding

LISTA DE FIGURAS

LISTA DE TABELAS

GLOSSÁRIO

ADDoS – *Application Layer DDoS attacks*

DDoS – *Distributed Denial of Service*

DoS – *Denial of Service*

HTTP – *Hypertext Transfer Protocol*

IP – *Internet Protocol*

ISO – *International Organization for Standardization*

NAT – *Network Address Translation*

OSI – *Open Systems Interconnection*

SDN – *Software Defined Network*

SHADE – *Selective High Rate DDoS Defense*

SeVen – *Selective Verification in Application Layer*

TCP – *Transport Control Protocol*

SUMÁRIO

Capítulo 1

INTRODUÇÃO

A cada dia, pode-se presenciar uma verdadeira revolução na computação e como ela altera o nosso dia a dia. Computadores estão conectados, por meio das redes de computadores, quebrando barreiras geográficas, pois, a informação que antes estava indisponível devido a distância do utilizador, agora está ao alcance de um clique. Todavia, pessoas não autorizadas podem ter acesso a dados confidenciais, ou então, usuários maliciosos deixam a informação indisponível a clientes legítimos ou provocam modificação do conteúdo.

Dentre as várias ameaças que existem à Segurança da Informação em Redes de Computadores, os Ataques de Negação de Serviço (DoS – *Denial of Service* – em inglês) estão entre os que mais impactam nos negócios atualmente [5]. Esse tipo de ameaça atenta exatamente contra um dos pilares da Segurança de Redes: a Disponibilidade. A intenção do ataque é simplesmente fazer com que um serviço, como uma página *web*, pare de responder a clientes que solicitam essa página. Ou seja, é uma tentativa de um usuário não legítimo de degradar ou negar recursos a um usuário legítimo [6]. O ataque pode ser realizado de duas formas, apenas um atacante realizando a indisponibilidade, ou então, o ataque realizado de forma distribuída (DDoS - *Distributed Denied of Service*), sendo essa segunda forma a mais utilizada, pois é possível atingir o objetivo mais rápido, gerando tráfegos maiores e possibilitando ao autor do ataque não ter sua identidade comprometida.

Tradicionalmente, os ataques de DoS ou DDoS são realizados nas camadas de Rede e/ou Transporte, tendo como características uma grande quantidade de pacotes. Visan deixar um servidor, como um todo, indisponível, ou seja, todas as aplicações em execução param de responder a clientes legítimos. Por exemplo, o ataque *Syn flood*, que consiste no atacante enviar várias solicitações de abertura de conexão, por meio da opção *Syn* do protocolo TCP. O servidor aceita essas solicitações e aloca recursos (memória e tempo de CPU). Dessa forma, esgotará os recursos do servidor, fazendo com que seja rejeitado toda nova conexão que chegar. Atual-

mente, os ataques migraram para a camada de Aplicação, que visam indisponibilizar apenas um serviço dentro do servidor. Além disso, os ataques de Negação de Servido na camada de Aplicação (ADDoS) tem como característica um tráfego semelhante aos de clientes honestos [7]. Dessa forma, as ferramentas tradicionais de análise do tráfego, não detectam esses tipos de ataques [8].

Dentro dos ataques ADDoS, o tipo *high-rate* utiliza características dos protocolos da camada de aplicação, para realizar uma inundação de requisições. Por exemplo, o ataque *Get-Flooding*, que utiliza o método *Get* para realizar um pedido a uma página da internet. O ataque consiste em realizar uma grande quantidade de requisições, e como se assemelha a pedidos de clientes legítimos, o servidor *web* aloca recursos para atenter a todos esses pedidos. Esse tipo de ataque, de acordo com relatório do ano de 2014, divulgado pela NSFOCUS [9], foi o quarto mais praticado no mundo.

Como se assemelha a requisições legítimas, a simples análise do tráfego não é suficiente para determinar se uma requisição é um cliente ou um atacante. Dessa forma, essa dissertação propõe algumas estratégias para mitigar ataques de negação de serviço na camada de aplicação, as quais são explicadas adiante:

- Identificação de um cliente, por meio de um *fingerprinting*;
- Redirecionamento do fluxo, e uma seleção das requisições que irão até a aplicação *web*;
- Informações precisas da taxa de utilização do serviço *web*.

A caracterização do cliente, diferenciando de um atacante, pode ser realizada na utilização de uma técnica de assinatura *web* (que será chamada de *fingerprinting*), que permite identificar um usuário em um fluxo de dados, na qual monitoram-se parâmetros, como sistema operacional, IP, versão do navegador, *add-ons*[1] utilizados. Assumindo que dois usuários diferentes não usam os mesmos parâmetros[2], pode-se determinar se uma requisição é de um usuário que já foi visto anteriormente. Realizando essa identificação, o usuário pode ter seu tráfego priorizado.

O redirecionamento de fluxo e a seleção se faz necessário, pois, o ataque tem como característica um volume considerável de requisições, e uma forma de seleção irá diminuir a quantidade de requisições. Essa seleção deve ser realizada fora do servidor *web*, para não sobrecarregá-lo. Portanto, o fluxo, que tinha como destino o servidor *web*, será redirecionado a essa outra entidade que realizará a seleção. Mas os fluxos identificados como clientes legítimos não devem sofrer a seleção e o descarte.

[1] Ferramentas extras adicionadas a funcionalidade dos navegadores.

[2] É muito baixa a probabilidade de dois usuários terem os mesmos parâmetros.

Quanto informação da taxa de utilização do servidor *web*, esta deve ser resgatada para que o sistema de defesa tenha informações precisas, pois, o redirecionamento do fluxo e a seleção das requisições só devem entrar em ação quando há uma sobrecarga do servidor *web*, evitando assim um *overhead* desnecessário de processamento na rede.

Portanto, este trabalho propõe uma defesa contra ataques DoS na camada de aplicação usando listas brancas (*whitelists* - em inglês) dinâmicas baseadas em *fingerprinting*. Foi implementado tais listas dinâmicas sobre a defesa SHADE (*Selective High-Rate DDoS Defense*) [10]. Na proposta, foi estendida a função da ferramenta *SeVen* [8], para atuar como defesa e como monitoramento de carga, e uma estratégia para identificar características de clientes legítimos por meio de um *fingerprinting* do cliente, por exemplo, identificando um cliente por meio do padrão de interação com a página *web*. Havendo a identificação, esse cliente é colocado em uma *whitelist* dinâmica, assim, todo o tráfego é encaminhado para o SHADE, que realiza uma seleção das requisições, e os clientes legítimos identificados são encaminhados diretamente para o servidor *web*. A seleção e o descarte se faz necessário, pois como a característica do *Get-Flooding* é enviar uma quantidade de *GET's*, o tráfego inútil será descartado, diminuindo assim o tráfego que chega no servidor *web*.

Para verificar a consistência da defesa, foram realizados experimentos na rede. Neste caso específico, foram realizados em uma rede definida por *software* (SDN - *Software Defined Network*), utilizando controlador RYU [11] e em um *switch* programável de código aberto, com o OpenWRT [4, 12]. A opção de utilizar SDN se verifica pelas vantagens que este paradigma oferece: uma visão mais centralizada da rede, permitindo um maior controle e gerenciabilidade, oferecendo a possibilidade de diferentes elementos da rede possam se comunicar e trocar informações, a fim de obter mais elementos para a tomada de decisão. Dessa forma, a proposta deste trabalho utiliza a rede SDN para agrupar informações da aplicação defendida, e realizar o redirecionamento do fluxo destinado a essa aplicação para o sistema SHADE, que realiza uma seleção das requisições.

Foram realizados experimentos, em que foi verificado que uma aplicação *web* só conseguiu prover disponibilidade para 19% dos clientes, durante um ataque de negação de serviço, sem nenhum mecanismo de defesa. Enquanto que com a utilização do mecanismo proposto por esse trabalho, a disponibilidade dos clientes subiu para mais de 99%.

1.1 Objetivos

Este trabalho tem como objetivo geral a utilização de *whitelist* dinâmicas, baseadas em *fingerprinting* para propor uma defesa contra ataques de negação de serviço na camada de aplicação (ADDoS), especificamente os ataques do tipo *high-rate*, como por exemplo o *Get-Flooding* que utiliza um método do HTTP para enviar uma inundação de solicitações ao servidor. Para atingir tal objetivo, este trabalho tem como objetivos específicos:

- Estudar a utilização de *whitelist* dinâmicas no contexto de Segurança de Redes de Computadores;
- Verificar as formas de identificar um cliente legítimo, no tráfego HTTP, por meio de um *fingerprinting* do usuário, presentes na literatura e propor um uso do *fingerprinting* em tráfego HTTP: caracterizar clientes legítimos a fim de que se possa distinguir de um tráfego atacante, para mitigar ataques ADDoS;
- Propor a defesa, chamada SHADE, para mitigar ataques de negação de serviço na camada de aplicação, utilizando *whitelist* baseadas em *fingerprinting*;
- Validar o SHADE em experimentos na rede, por meio dos cenários construídos utilizando as Redes Definidas por *Software*.

1.2 Trabalhos relacionados

Nesta seção são descritos alguns trabalhos relacionados com a estratégia proposta na dissertação, levando em consideração as semelhanças, diferenças e propostas para o presente trabalho.

A utilização de técnicas de *fingerprinting* nos serviços *web* se iniciou por meio do conceito primário do *fingerprinting*, que é a autenticação do usuário utilizador da aplicação *web* [13] [14]. Ou seja, o serviço *web* utiliza, por meio da impressão digital, padrão da palma da mão, reconhecimento da iris/retina entre outras formas, uma autenticação do usuário que irá utilizar o sistema. O *fingerprinting* também é utilizado por atacantes para verificar vulnerabilidades existentes em serviços *web* [15]. No artigo [16], é proposto um mecanismos para evitar que atacantes realizem essas varreduras e utilizem as vulnerabilidades da aplicação. Em contrapartida, o *fingerprinting* também é utilizado para verificar aplicações maliciosas em serviços *web*. Em Qassrawi e Zhang [17], é utilizado um *honeyclient* para verificar se páginas estão com algum código malicioso, tentando roubar alguma informação de clientes.

Este trabalho propõe um novo uso para *fingerprinting*: utilizar *fingerprinting* para caracterizar clientes legítimos a fim de que se possa distinguir de um tráfego atacante, colocando-o em uma *whitelist* dinâmica para mitigar ataques de negação de serviço na camada aplicação do tipo *high-rate*, *e.g.*, *Get-Flooding*.

A utilização de *whitelist* para mitigar ataques de negação de serviço é pouco difundido na literatura, ainda mais quando se trata em ataques contra HTTP. De um modo em geral, é extremamente utilizado o conceito de *blacklist*, em que se bloqueia o usuário malicioso. O *blacklist*, não é efetivo contra ataques de negação de serviço na camada de aplicação, já que não é possível distinguir tráfego de atacante e de cliente, assim, se adicionar algum tráfego na *blacklist* poderá estar negando serviço a um cliente legítimo. Dessa forma, a utilização de *whitelist* é mais recomendado para esse tipo de ataque, pois é dado prioridade para clientes legítimos genuinamente reconhecido de alguma forma. No estudo de Bianchi *et al.* [18], é difundido entre os provedores de internet uma *whitelist*, com IP's de clientes legítimos, cujo tráfego não pode ser bloqueado em caso de DoS. Ou seja, apenas organizações, em que os IP's são conhecidos, serão adicionados na *whitelist* em caso de ataque de negação de serviço. Os outros usuários legítimos, que não são conhecidos, poderão ter o serviço negado.

Em Chen e Itoh [19], é utilizado uma *whitelist* para dar prioridade a clientes legítimos de sistemas *Voice over IP* (VoIP). Quando um cliente se registra no servidor VoIP, utilizando o seu *login* e a senha, é enviado uma mensagem ao sistema de defesa e esse usuário é inserido em uma *whitelist* e assim, o tráfego tem prioridade de transmissão. Dessa forma, Chen e Itoh [19] propõem uma defesa contra ataques de negação de serviço utilizando *whitelist*. Há duas diferenças com o presente trabalho: a primeira é em relação ao protocolo utilizado, enquanto Chen e Itoh [19] propõem a defesa na aplicação VoIP, o presente trabalho propõe para aplicação *web*; a segunda diferença está relacionada a forma de identificação dos clientes legítimos, que em Chen e Itoh [19] o cliente é reconhecido pela sua autenticação no sistema VoIP, enquanto que neste trabalho além de autenticar por meio do sistema de *login* da aplicação *web*, é proposto também identificar por meio da interação de um cliente com a página *web*. Em ambos os trabalhos, os tráfegos dos clientes identificados e inseridos na *whitelist* terão prioridade na transmissão.

Em relação as redes SDN, verifica-se na literatura vários artigos propondo defesas contra ataques de negação de serviço. No entanto, a maioria dos estudos abordam ataques nas camadas de Rede e/ou Tranporte [20–23], como por exemplo, ataque do tipo *SYN-Flood*, que tem uma taxa de volume muito maior que os ataques ADDoS; outra parte dos trabalhos abordam ataques DDoS utilizando brechas provenientes das características de uma rede SDN. Algumas estratégias tem sido utilizadas na literatura, como por exemplo: monitorar a quantidade de *handshakes*

TCP (SYN, SYN-ACK e ACK), que não são concluídas para cada IP de origem [20]; uso de redes neurais para avaliar se um fluxo é ou não um ataque, com treinamento da rede baseado em parâmetros que podem ser extraídos durante um ataque sobre a camada de Rede/Trasporte (*SYN-flood, UDP-flood, e ICMP-flood*) [21]; identificar possíveis ataques que exploram pontos fracos da arquitetura SDN [22], em particular, a possibilidade de negar o serviço de todas as aplicações da rede SDN enchendo a memória do *switch* com um grande número de regras. Essas abordagens são bem diferentes do SHADE, ou usam estratégias que não são aplicáveis a ataques ADDoS, que tem um comportamento diferente com tráfego que se assemelha ao tráfego cliente [7].

Por fim, mais recentemente, Fayaz *et al.* [24] propôs a combinação de redes SDN e virtualização de funções de redes (NFV – *Network Function Virtualization* - em inglês) para mitigar grandes ataques de amplificação. A solução é identificar grandes volumes de tráfego e usar aplicações adequadas (por meio do NFV) em locais estratégicos (utilizando SDN) para mitigar o ataque. A diferença para o SHADE, é o que se propõe mitigar, pois, o tráfego gerado por ataques ADDoS do tipo *high-rate*, é insignificante em relação aos ataques de amplificação.

1.3 Estrutura da Dissertação

O trabalho está dividido em um total de cinco capítulos, a Introdução, com os objetivos, gerais e específicos, do trabalho, e a discussão do Estado da Arte. No segundo capítulo encontra-se a fundamentação teórica, que trata, de forma mais aprofundada, dos assuntos que esse trabalho aborda.

O Capítulo 3 tem-se a descrição da proposta para mitigar ataques de negação de serviço na camada de aplicação. No Capítulo 4 discorre sobre o cenário desenvolvido, as ferramentas utilizadas para os experimentos e as métricas escolhidas para avaliar a proposta. Há no Capítulo 4 também a discussão acerca dos resultados obtidos durante os experimentos.

E no Capítulo 5, tem-se as considerações finais sobre o trabalho, bem como sugestões para trabalhos futuros. Por fim encontram-se as referências bibliográficas utilizadas no trabalho.

Capítulo 2

FUNDAMENTAÇÃO TEÓRICA

Esse capítulo trata de alguns dos pilares que norteiam o trabalho proposto, são eles: Segurança da Informação, especificamente sobre Ataques de Negação de Serviço, a utilização de *whitelist*, *fingerprinting* como forma de reconhecer clientes legítimos, *Software Defined Network* - SDN, e a estratégia *SeVen* (*Selective Verification in Application Layer*).

2.1 Segurança da Informação

Atualmente, vivencia-se uma era em que a informação é mais valiosa do que o próprio dinheiro, por isso empresas e/ou indivíduos gastam milhões para tentar proteger ou roubar a informação.

Com as empresas cada vez mais dependentes de serviços de rede, manter os serviços seguros e em execução torna-se sinônimo de manter o negócio funcionando.

Segundo a NBR ISO/IEC 27001 na sua introdução, afirma que "Segurança da informação é a proteção da informação de vários tipos de ameaças para garantir a continuidade do negócio, minimizar o risco ao negócio, maximizar o retorno sobre os investimentos e as oportunidades de negócio."

Deve-se ter em mente que a Informação não se restringe a dados digitais, mas sim a toda e qualquer forma que for apresentada. Mas todos os tipos de Informação, a não ser a digital, estão fora do escopo desse trabalho, devendo, aos que tem interesse, buscar as Normas técnicas referentes ao tema ou a trabalhos específicos. O que trataremos é da Segurança da Informação relacionada a Redes de Computadores.

Segundo Tanenbaum [25]:

> "Em sua forma mais simples, a segurança se preocupa em garantir que pessoas mal-intencionadas não leiam ou, pior ainda, modifiquem secretamente mensagens enviadas a outros destinatários. Outra preocupação da segurança são as pessoas que tentam ter acesso a serviços remotos que elas não estão autorizadas a usar. Ela também lida com meios para saber se uma mensagem supostamente verdadeira é um trote. A segurança trata de situações em que mensagens legítimas são capturadas e reproduzidas, além de lidar com pessoas que tentam negar o fato de terem enviado determinadas mensagens.".

De um modo geral, o que Tanenbaum [25] explica são os conceitos fundamentais da Segurança em Redes de Computadores: Disponibilidade, Confidencialidade e Integridade (da Origem e dos Dados). A Disponibilidade é a garantia que a Informação estará disponível quando necessária; a Confidencialidade é a garantia que apenas pessoas autorizadas tenham acesso à informação; e a Integridade que subdivide em duas: Origem quando o emissor não pode negar que enviou uma mensagem (também chamada de não-repúdio) e dos Dados que é a garantia que a mensagem não foi alterada.

Tanenbaum [25] afirma que a maioria dos problemas relacionados à Segurança de Redes é causada intencionalmente, e que foi levada adiante por pessoas que tentam obter algum benefício, chamar atenção a algo ou até mesmo prejudicar alguém. Além disso, todas as camadas relacionadas a redes de computadores, podem ser alvo de ataques, levando em consideração as características de cada camada e/ou comportamento ou brechas oriundas dos protocolos existentes em cada camada. Todas as camadas devem ser também objeto de implementação de Segurança.

2.2 Ataques de Negação de Serviço

Ataques de Negação de Serviço (DoS – *Denial of Service* – em inglês) são frequentes nas Redes de Computadores, impactando nos negócios de diversas empresas [5]. Esse tipo de ameaça, atenta exatamente contra um dos pilares da Segurança de Redes: a Disponibilidade. A intenção do ataque é simplesmente fazer com que um serviço, como uma página da internet, pare de responder a clientes legítimos. Shea e Liu [6] definem DoS como tentativas de um usuário não legítimo de degradar ou negar recursos de um usuário legítimo. Conceitualmente o ataque pode ser de duas formas, apenas um atacante provocando a indisponibilidade ou então o ataque sendo realizado de forma distribuída (DDoS – *Distributed Denial of Service* – em inglês). Atualmente, o DDoS é o que está sendo usado, pois é uma forma de atingir o objetivo

mais rápido e sem muito esforço, pois, o atacante utiliza máquinas de terceiros para realizar o ataque.

De forma intuitiva, a negação de serviço ocorre quando a aplicação não consegue servir mais nenhuma requisição de um usuário legítimo. Dessa forma, existem diversas formas de realizar a indisponibilidade: quando há consumo da memória e CPU de um servidor; consumo de toda banda de rede disponível; preenchimento de todos os espaços de atendimento em uma aplicação específica; etc. Os ataques de negação de serviço na camada de aplicação objetivam a ocupação dos espaços de atendimento da aplicação, e é neste tipo de ataque que este presente trabalho se propõe a mitigar.

Neste tipo de ataque, de acordo com a Figura 2.1, uma aplicação começará a negar serviço quando todos os espaços reservados para o atendimento estão preenchidos, assim, uma nova solicitação será descartada, gerando a indisponibilidade. Essa quantidade de atendimentos simultâneos é definido, entre outros fatores, pela limitação de memória que a aplicação tem destinada dentro do sistema operacional.

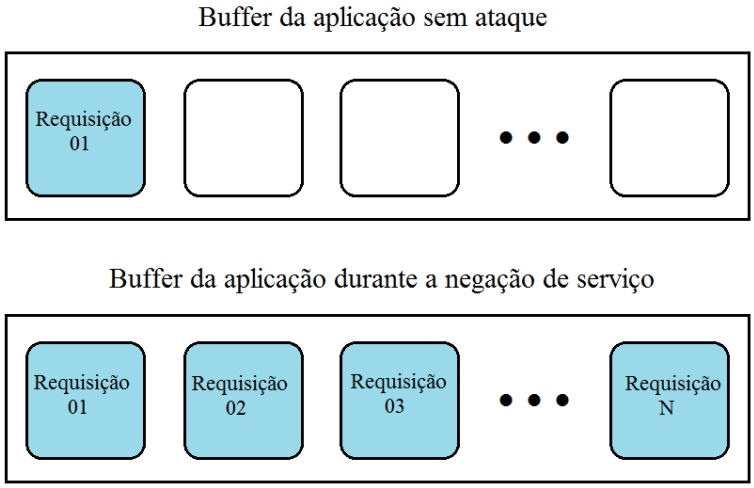
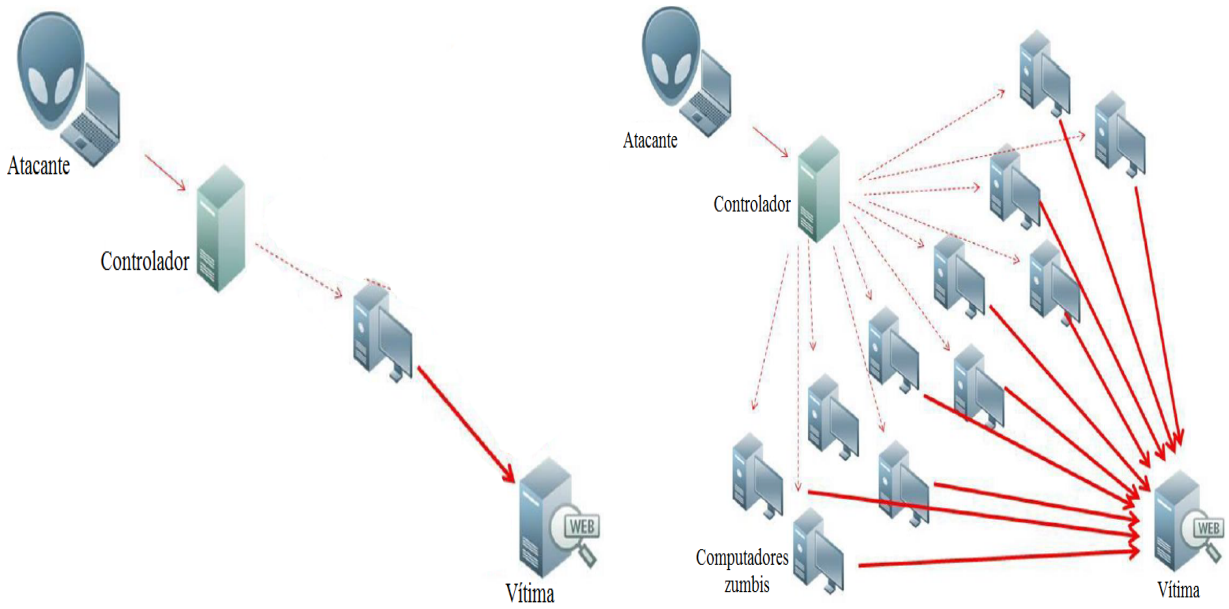


Figura 2.1: Exemplo de preenchimento do *pool* de atendimento da aplicação. Adaptado de Dantas [1].

Assim, usuários maliciosos, sabendo que uma aplicação tem um tamanho limitado para o atendimento, enviam uma quantidade de requisições que superam esse limite, a fim de gerar uma indisponibilidade a usuários. Apesar do ataque ser uma forma de realizar a indisponibilidade, o preenchimento do *buffer* de atendimento pode ser realizada por um grande acesso de usuários legítimos. Assim, na Figura 2.1, as requisições que estão em todos os *slots* de atendimento são de clientes que estão acessando aquele servidor *web*. Isso pode acontecer em grande promoções de sites de venda; em época de inscrição para cursos superiores em universidades públicas; ou então em final de prazo para a entrega da declaração do imposto de renda. A indis-

ponibilidade pode acontecer pela falta do dimensionamento da infraestrutura de servidores por parte da equipe da tecnologia da informação. Neste trabalho a infraestrutura foi bem dimensionada para que não ocorra esse tipo de indisponibilidade, como verificado na primeira parte da Seção 4.3

A Figura 2.2 mostra o esquema de realização de um ataque de negação de serviço, individual, representada na Figura 2.2(a), e distribuída, esquematizada na Figura 2.2(b). Nas duas imagens há a presença do atacante, que é o agente malicioso com intuito de gerar a indisponibilidade. O controlador é o serviço de comunicação, de onde será gerado as instruções do ataque. Na Figura 2.2(a) o controlador se comunica apenas com uma máquina, sendo ela que irá realizar o ataque. Enquanto que na Figura 2.2(b), o controlador se comunica com os computadores zumbis, que são máquinas que foram infectadas e obedecem aos comandos do atacante. A diferença do DoS para o DDoS é justamente a quantidade de máquinas gerando o tráfego malicioso. Por fim, no esquema, temos a presença da vítima do ataque, que pode ser um servidor *web*, por exemplo.



(a) DoS (b) DDoS

Figura 2.2: Esquema de ataques de negação de serviço, de forma individualizada e distribuída. Adaptado [2].

O ataque de DDoS é normalmente realizado por meio de "recrutamento" de máquinas zumbis utilizando *worms* (de acordo com o Centro de Estudos, Respostas e Tratamento de Incidentes de Segurança no Brasil – CERT.BR, *worms* é um processo automatizado de propagação de códigos maliciosos na rede), em seguida esses recrutas enviam, de forma coordenada, um grande números de pacotes para um servidor a fim de que possam sobrecarregar a capacidade

de processamento tornando-o indisponível a clientes legítimos [8]. Em comparação do ataque de DoS com o DDoS, Almeida [26] afirma que o poder da segunda forma é muito superior ao da primeira, por dois motivos: primeiro que sistemas de detecção tem uma maior dificuldade de encontrar a origem do ataque; e o segundo motivo é a quantidade de pacotes que o ataque de DDoS acarreta.

Para se ter uma ideia, Dantas, Nigam e Fonseca [8] relatam sobre vários ataques de inundação, realizada pelo "Anonymous", durante anos anteriores, levando a sites indisponíveis de instituições como Mastercard.com, Paypal, Visa.com e PostFinance, que afetaram nove dos maiores bancos americanos: Bank of America, Citigroup, Wells Fargo, U.S. Bancorp, PNC, Capital One, Fifth Third Bank, BB&T e HSBC [27]. Além do que, com a difusão de informação pela Internet, a possibilidade de disseminar técnicas para ataques de Negação de Serviço é imensa, levando com que os próprios usuários, sem tanta ligação ou motivação com as vítimas, realizem esses ataques.

Normalmente, Ataques de Negação de Serviço são realizados utilizando características dos protocolos da camada de Rede e de Transporte com a intenção de tornar todos os serviços de um servidor indisponível. Para tanto, é necessário uma grande quantidade de pacotes. Atualmente, estão surgindo ataques que visam a camada de Aplicação[1]. Essa mudança acarreta não apenas utilizar brechas de outros protocolos, mas também na intenção do ataque e na característica do ataque. Quanto a intenção, não se quer mais tornar um servidor como um todo indisponível, mas sim apenas uma aplicação específica desse servidor que ficará indisponível. Em relação à característica do ataque, não é necessária mais uma grande quantidade de tráfego para alcançar o objetivo, mas sim um tráfego mais pontual e semelhante ao tráfego de um cliente. Diante disso, as ferramentas tradicionais de análise e de detecção do tráfego não são mais eficientes para esse tipo de ataque, fazendo com que nenhuma ação seja realizada justamente por não haver um aviso de que está acontecendo o ataque propriamente dito.

Dentre dos tipos de Ataques de Negação de Serviço Distribuídos na camada de Aplicação (ADDoS - *Application-layer Distributed Denial of Service* - em inglês) podemos classificar de duas formas: ataques do tipo *low-rate* e do tipo *high-rate*. Esses ataques serão descritos em sequência.

[1]Todo o processo de comunicação entre dois computadores é dividida em camadas, em que cada camada é responsável por uma etapa da comunicação. As camadas, de acordo com modelo OSI/ISO [25] são: física, enlace, rede, transporte, sessão, apresentação e aplicação.

2.2.1 Ataques do tipo *Low-Rate*

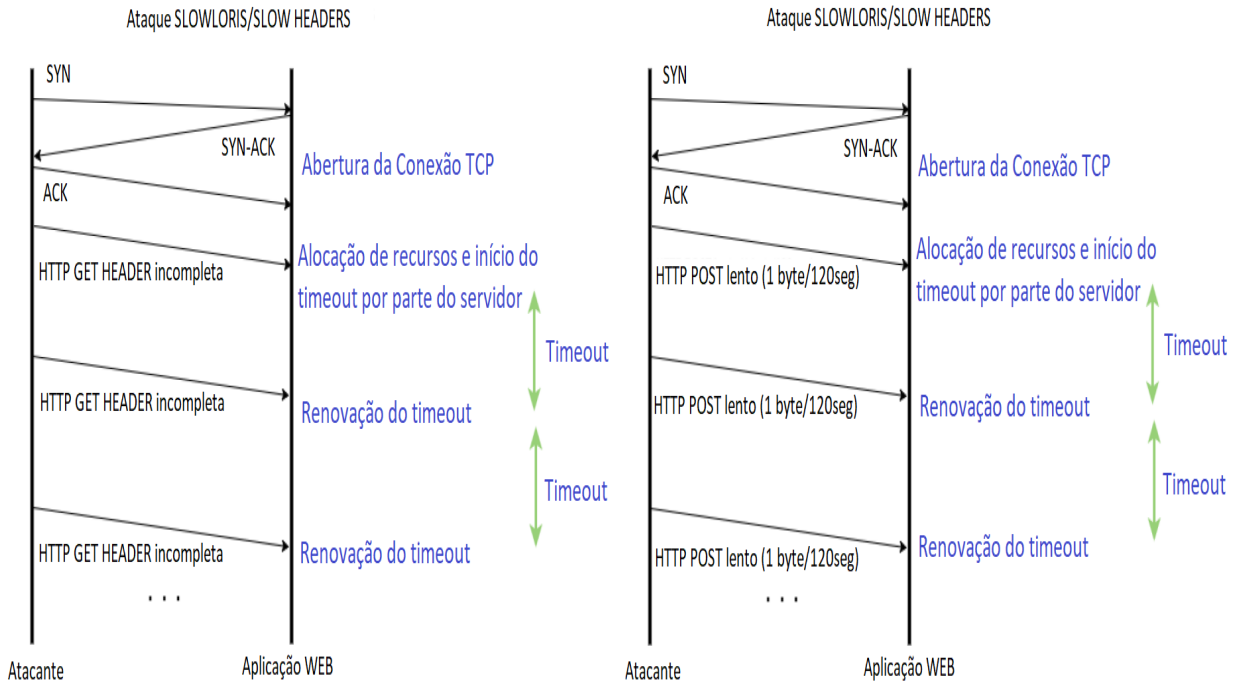
Os ataques do tipo *low-rate* têm a característica de ser, como o próprio nome diz, de baixa taxa de tráfego, ou seja, o atacante tenta simular o tráfego de um cliente normal, dificultando assim a detecção do tráfego malicioso. De acordo com Dantas, Nigam e Fonseca [8], esse tipo de ataque procura explorar vulnerabilidades existentes em protocolos de aplicação, enviando menos tráfego na rede, mas ocupando espaço no atendimento do servidor. Com isso, clientes legítimos não encontram espaço para atendimento, tornando o serviço indisponível.

Entre as variações desse tipo de ataque, existem três formas de ataques que utilizam características dos protocolos. Os ataques *Slowloris* [28], *HTTP POST* [29] e *Slowread* [30], são realizados contra o protocolo HTTP, responsável por páginas da Internet. Os dois primeiros trabalham em cima do *timeout* da aplicação *web*, enquanto que o *Slowread* combina elementos dos protocolos da camada de transporte e de aplicação. Esse *timeout* é o tempo que uma requisição permanece na fila de atendimento da aplicação. Dessa forma, a intenção dos ataques é permanecer no *buffer* de atendimento por tempo indeterminado.

O *Slowloris* [28] consiste no envio de requisições HTTP GET HEADER incompletas (sem os campos CR - *Carriage Return*, ASCII 13, */r*, e o LF - *Line Feed*, ASCII 10, */n*) no final do pacote, sempre em um certo intervalo de tempo, para forçar suas renovações, fazendo com que o servidor nunca saiba quando as requisições foram finalizadas, mantendo-as no pool de atendimento até o valor de *timeout* configurado previamente no servidor, a partir de um momento, todos os recursos estarão sendo consumidos pelas requisições maliciosas, indisponibilizando a aplicação. A Figura 2.3(a) exemplifica a realização do ataque *Slowloris*.

O *HTTP POST* [29] envia requisições HTTP GET HEADER completas, realizando o *TCP HANDSHAKE* com o servidor, porém envia seus dados pelo campo BODY (via método HTTP POST) de maneira muito lenta, fazendo com que o servidor aguarde até o *timeout* configurado ou a quantidade máxima de dados no BODY de uma requisição seja atingida. Assim, essas requisições lentas tomam posse de todo o *pool* de atendimento e indisponibilizam a aplicação. Esse ataque é exemplificado na Figura 2.3(b).

O *Slowread* [30] controla o tamanho da janela TCP, utilizado para informar quantos *bytes* pode receber, sendo um mecanismo para controle de fluxo. Ou seja, o ataque consiste em diminuir gradativamente a quantidade de *bytes* que o atacante pode receber, até chegar a 0 *bytes*. Assim, ocupa os recursos do servidor *web*, demorando para enviar por completo a resposta para o atacante. Dessa forma, o servidor fica indisponível, tentando responder a atacantes, pensando em se tratar de clientes com uma máquina/processador lento. A Figura 2.4 exemplifica esse



(a) Ataque *Slowloris*. (b) Ataque *HTTP POST*.

Figura 2.3: Ataques do tipo *Low-Rate*: *Slowloris* e HTTP POST.

ataque: após o *handshake* entre a máquina atacante e do servidor *web*, o atacante realiza um GET e na janela TCP afirma que pode receber até 1472 *bytes*, assim o servidor *web* aloca recursos para enviar o conteúdo ao usuário. Inicia-se o envio do conteúdo, sendo o tamanho do segmento limitada a 1472 *bytes* na camada de transporte, quando o atacante responde com o ACK que recebeu esse segmento, envia também o novo tamanho da janela TCP com valor de 778 *bytes*, diminuindo a quantidade que o servidor *web* pode enviar. Isso se repete até o atacante chegar ao valor de 0 *bytes*. A conexão permanece aberta e consumindo recursos do servidor *web* até que o tempo limite estabelecido na aplicação se esgote.

Como foi dito anteriormente, justamente por enviar menos tráfego na rede e se assemelhar a tráfego cliente, esses tipos de ataque são extremamente difíceis de serem identificados pelas ferramentas de detecção tradicionais.

Este trabalho não trata sobre ataques do tipo *low-rate*, apesar de utilizar a ideia proposta por Dantas [1, 31], que propõe uma defesa para esses tipos de ataques. A ferramenta *SeVen* proposta por Dantas [1, 31], que será explicada na Seção 2.5, é adaptada e acrescentada a um sistema de defesa, visando mitigar outros tipos de ataque na camada de aplicação, os do tipo *high-rate*, explicado a seguir.

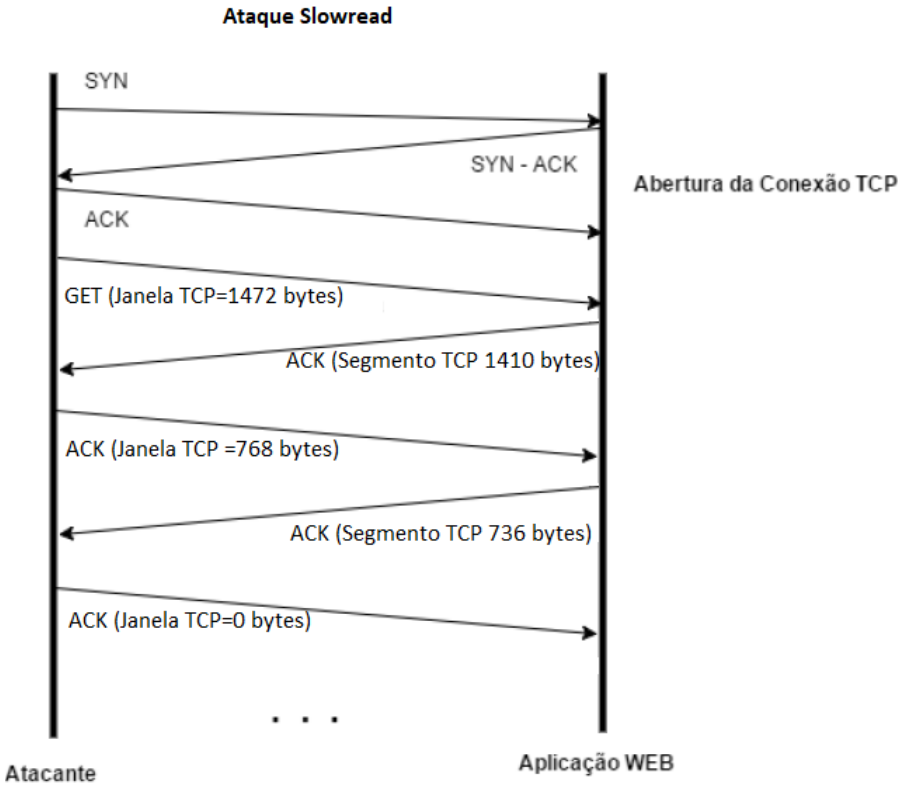


Figura 2.4: Diagrama do ataque *Slowread*.

2.2.2 Ataques do tipo *High-Rate*

Enquanto que ataques do tipo *low-rate* têm a característica de enviar pouco tráfego, sempre próximos ao *timeout*, simulando tráfego de clientes legítimos, os ataques do tipo *high-rate* têm como ação enviar diversas requisições, caracterizando uma inundação. A sua característica é a mesma em ataques de inundação em camadas inferiores (de rede e de transporte), em que uma grande quantidade de tráfego é enviado ao servidor para haver uma sobrecarga no processamento e gerar a indisponibilidade [7].

A diferença entre os ataques de inundação nas camadas de rede/transporte, para os ataques do tipo *high-rate* na camada de aplicação se dá de duas formas: primeiro pela intenção do ataque, nas camadas inferiores a intenção é indisponibilizar todos os serviços, é deixar o servidor como um todo indisponível, enquanto que nos ataques *high-rate* pretende-se indisponibilizar apenas uma aplicação específica do servidor; a segunda diferença entre os ataques de inundação é pela quantidade de tráfego gerado, pois nos ataques das camadas de rede/transporte a quantidade de tráfego que deve ser gerado para alcançar o objetivo descrito acima é extremamente alto, podendo assim haver uma detecção pelas ferramentas de análise de tráfego, enquanto que nos ataques do tipo *high-rate*, a quantidade de tráfego é grande se comparado ao tipo *low-rate*, mas que se assemelha a tráfego de uma grande quantidade de clientes utilizando

o serviço, como, por exemplo, nos dias de promoções do tipo *Black Friday*.

Dentro desse tipo de ataque, o protocolo mais utilizado é o HTTP. De acordo com o relatório da NSFOCUS do ano de 2014 [9], o ataque *GET-Flooding* (que envia uma inundação de pacotes com o método GET do HTTP) ficou em quarto lugar em popularidade de ataque, ficando atrás apenas dos ataques ao protocolo DNS e dos ataques *flooding* ao TCP e UDP. No terceiro trimestre de 2016, de acordo com o relatório da Arkamai [32], entre os ataques de negação de serviço na camada de aplicação, o mais utilizado é o *Get-Flooding*.

O algoritmo do ataque HTTP *Get-Flooding* está demonstrado na Figura 2.5, e se realiza da seguinte forma: o atacante, inicialmente, realiza uma conexão TCP com o Servidor *web*, a partir disso envia uma solicitação HTTP GET para verificar se o servidor está disponível. Se o atacante receber uma resposta da aplicação então o atacante envia periodicamente várias solicitações HTTP GET sem esperar a resposta do servidor. Com isso há uma inundação de requisições e que o servidor tenta responder todas. Assim há um consumo de memória e CPU do servidor, levando-o a não conseguir servir a todas requisições, gerando a negação de serviço.

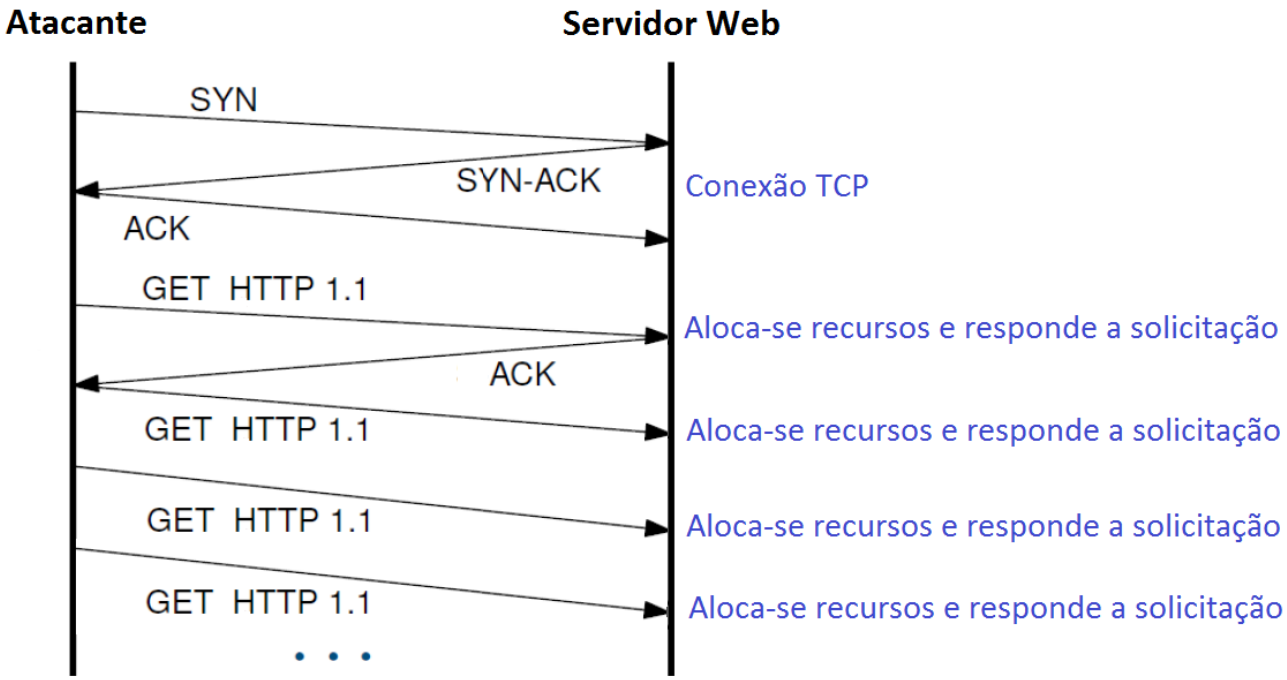


Figura 2.5: Diagrama do ataque *Get Flooding***.**

Dessa forma, quando um serviço está sob ataque, um sistema de defesa que realiza sua mitigação baseada na análise de tráfego não consegue distinguir se aquele tráfego é oriundo de um ataque ou se é apenas uma maior utilização do serviço por parte de clientes legítimos. Assim, para se ter uma maior precisão, o ideal é que o sistema de defesa troque informações com as aplicações que está sob a sua defesa, para, com mais informações, poder tomar alguma

ação efetiva contra o ataque. Esses dados podem ser a quantidade de clientes em atendimento, a frequência que cada cliente requisita alguma informação, a origem desses clientes, etc. Qualquer informação, e o cruzamento dessas informações com o próprio tráfego auxiliam para ter uma melhor tomada de decisão do sistema de defesa em relação ao ataque.

2.2.3 *Blacklist* e *Whitelist* na Segurança da Informação

A utilização de *blacklist* e *whitelist*, listas que negam ou aceitam requisições, são largamente difundidos nas redes de computadores quando se trata em segurança da informação. A *blacklist*, em uma tradução livre "lista negra", tem como objetivo negar o acesso a informação dos usuários, ou endereços IPs, que estejam inseridos nesta lista. A lista é alimentada por meio de informações fidedignas, ou seja, o administrador de rede tem certeza que um determinado endereço IP é de um atacante, por exemplo. Assim toda vez que esse endereço IP realizar uma requisição, como estará inserido na *blacklist*, essa requisição não será atendida.

Em contrapartida, as *whitelist*, na tradução livre "lista branca", podem ser utilizadas de duas formas: a primeira forma é de bloquear todos os usuários, só atendendo quem estiver inserido na *whitelist* (mais restritiva, pois, o administrador da rede só libera o acesso a quem tem certeza que seja cliente); a segunda forma é aceitar todo tipo de tráfego, mas dando prioridade na transmissão/processamento/resposta aos usuários inseridos na *whitelist*.

O *blacklist*, o qual nega acesso, não é efetivo contra ataques de negação de serviço na camada de aplicação, já que não é possível distinguir tráfego de atacante e de cliente [7], assim, se adicionar algum tráfego na *blacklist* poderá estar negando serviço a um cliente legítimo. Por exemplo, um cliente legítimo, que por algum motivo, alterou seu padrão de utilização em um servidor *web*, e esse padrão se assemelhou a uma regra para inserir na *blacklist*. Assim, um cliente será negado serviço, pois, foi identificado como um atacante. Ou então, uma organização que utiliza apenas um IP público para acessar a *Internet*, e dentro dessa organização há um computador infectado realizando ataque. O tráfego desse único IP será colocado na *blacklist*, e será negado serviço a toda a organização por trás desse IP.

Dessa forma, a utilização de *whitelist* é mais recomentado para esse tipo de ataque, pois prioridade é dada para clientes legítimos genuinamente reconhecido, por meio, por exemplo, da autenticação realizada na aplicação *web*. A utilização de *whitelist* para mitigar ataques de negação de serviço é pouco difundido na literatura, ainda mais quando se trata em ataques contra HTTP.

2.3 *Fingerprinting*

Sobre *fingerprinting*, entende-se que é uma técnica de utilizar alguma parte do corpo humano para realizar uma autenticação do usuário, pois há partes do corpo que têm características únicas, servindo assim para identificar unicamente um usuário [33], substituindo a utilização de senhas complexas e difíceis de serem memorizadas por uma senha que sempre estará com o usuário. Pode-se traduzir o termo *fingerprinting* por "impressão digital", trazendo assim o sentido da técnica descrita acima, mas não se resumindo apenas a utilização da impressão da digital.

De um modo geral, a utilização da técnica de *fingerprinting* se iniciou nos serviços *web* realizando a autenticação do usuário, por meio da captura das digitais, da disposição da palma da mão, do formato da iris e da retina. Assim, o utilizador da aplicação *web* realiza sua autenticação por meio de alguma informação do corpo [13, 14]. Essa forma de autenticação é bastante utilizada em sistemas que requerem uma autenticação mais específica, tanto que o sistema eleitoral brasileiro está migrando sua forma de autenticação para a biometria.

A vantagem de se utilizar essa forma é que o usuário não precisa memorizar uma senha com requisitos de segurança (com caracteres alfanuméricos, maiúsculas, minúsculas e com caracteres especiais), mas apenas com informações do seu próprio corpo, dificultando muito assim que algum usuário malicioso tente se passar pelo usuário legítimo. Em contrapartida, caso a identificação codificada da informação do corpo do usuário seja roubada por algum atacante, não há como gerar uma nova digital, iris ou retina, como seria o caso de senhas [34].

Além dessa utilização de autenticar o usuário, o *fingerprinting* também foi utilizado por atacantes para verificar vulnerabilidades das aplicações *web* [15]. Os atacantes realizam uma inspeção minuciosa, uma "impressão digital", do serviço para encontrar brechas que possibilitem realizar um ataque. Da mesma forma, o *fingerprinting* é utilizado para verificar a existência de alguma aplicação maliciosa, tentando roubar informações de algum cliente. Assim, empresas de segurança da informação realizam uma verificação mais precisa do serviço *web*, buscando códigos maliciosos que visam obter, sem consentimento, informações de clientes legítimos [17].

Por fim, a utilização do *fingerprinting* que este trabalho propõe: buscar informações específicas que clientes legítimos a fim de distinguir entre tráfego que podem ser maliciosos. A proposta consiste em verificar, minuciosamente, o tráfego, a forma que se relaciona com o serviço *web*, características do *hardware* e *software* utilizado pelo usuário. Com essas informações é possível estabelecer um perfil de cliente legítimo. Dessa forma, diante de um tráfego de ataque que se assemelha a um tráfego cliente, pode-se distinguir um cliente legítimo por meio do

fingerprinting.

2.4 *Software Defined Network* - SDN

As Redes Definidas por *Software* (*Software Defined Network* - em inglês) são redes de computadores que inserem um novo paradigma no controle e utilização das redes. Nas redes de computadores tradicionais o plano de dados e o plano de controle residem no mesmo equipamento, ou seja, toda a lógica utilizada seja na comutação ou no encaminhamento dos pacotes estava a mercê das implementações dos fabricantes. Caso alguém quisesse experimentar um novo protocolo, como o *firmware* dos equipamentos não suportava, não era possível realizar o experimento.

Além disso, todo o controle de tarefas, como roteamento, engenharia de tráfego, controle de acesso, tem que ser realizado em cada equipamento participante da rede para que essas funções sejam realizadas. Isso gera uma propensão a erros na implementação e na manutenção dessas funções, pois, além de serem muitos equipamentos, cada fabricante tem sua forma de implementar cada função [35]. Adicionado a essas desvantagens, todos os equipamentos das redes tradicionais devem executar o mesmo algoritimo das funções exercidas, roteamento por exemplo, e garantir que tudo está funcionando corretamente.

Os propositores das Redes Definidas por *Software* quebraram a lógica de inserir, no mesmo equipamento, o plano de dados e o plano de controle. A proposição SDN é a separação do plano de dados e de controle [36]. A Figura 2.6 mostra a diferença entre a localização do plano de dados e de controle nas redes tradicionais (localizadas apenas nos próprios equipamentos) e nas Redes Definidas por *Software* (distribuída entre equipamentos e uma outra entidade). De acordo com Kreutz *et al.* [3] o plano de dados é a comutação/encaminhamento dos pacotes em si, a movimentação dos dados, enquanto que a decisão de como e onde os dados trafegarão é tarefa do plano de controle. O plano de dados continua a cargo dos equipamentos *switches* e o plano de controle é transferido para um novo elemento na rede: o controlador. O controlador é o elemento central da rede SDN, em que todas as decisões estão implementadas. Portanto, percebe-se na Figura 2.6, no que se destina as Redes Definidas por *Software*, a existência do Controlador junto ao plano de controle e no *switch* apenas o plano de dados. Ainda de acordo com a Figura 2.6, verifica-se que as funções que a rede irá realizar e as aplicações executadas nesta rede, estão resididas no controlador.

No *switch* da rede SDN, existe uma estrutura de dados chamada Tabela de Fluxo. Nessa Tabela de Fluxo estão guardadas informações sobre as regras da rede inseridas pelo controlador,

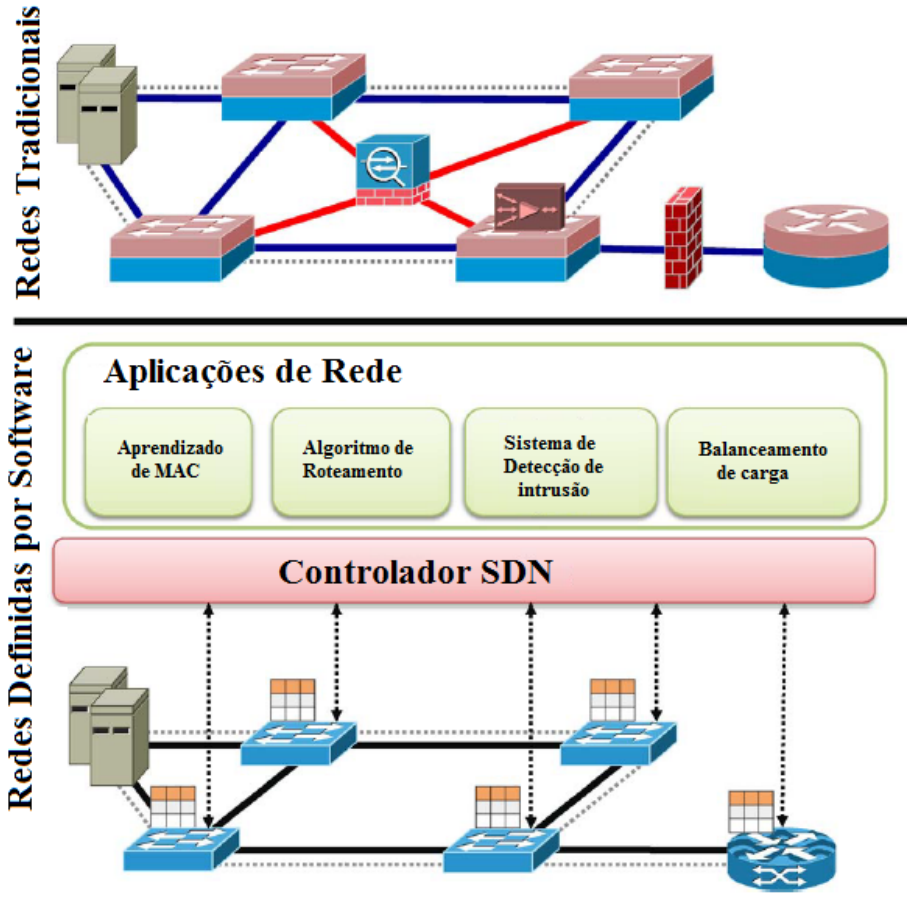


Figura 2.6: Diferença entre as redes tradicionais e SDN. Adaptado de Kreutz *et al.* [3].

as ações que serão realizadas e contadores referente a cada fluxo. Quando um pacote chega em um *switch* na rede SDN, informações acerca desse pacote são retiradas do cabeçalho, então é realizado uma busca na tabela de fluxos do *switch* a fim de encontrar alguma entrada correspondente. Caso encontre uma regra que satisfaça as informações do pacote, então é realizada a ação associada a regra. A ação pode ser descartar ou encaminhar o pacote para uma porta de saída (ou várias portas de saída). Se não encontrar correspondência na tabela de fluxo, acontece então o *packet-in*, em que o pacote é copiado e enviado ao controlador [4]. O controlador recebe o *packet-in* e verifica na sua aplicação o que deve ser feito com aquele tipo de fluxo (o controlador pode tomar a decisão baseado na lógica que a sua aplicação foi escrita, inclusive utilizando informações de outros elementos da rede). Decidindo o que deve ser feito, o controlador responde ao *switch*, inserindo uma nova regra na tabela de fluxo, que para esse tipo de fluxo deve realizar uma determinada ação (encaminhar o pacote, alterar parte do cabeçalho, descartar o pacote ou encaminhar para o controlador realizar a inspeção). Assim, o próximo pacote que chegar desse fluxo encontrará uma regra já instalada no *switch*.

É importante perceber que, a vantagem da rede ser programável está diretamente ligada

ao tempo de duração de uma regra na tabela de fluxo. Se a regra inserida tem uma duração muito longa (ou infinita) a rede se comporta de forma previsível, sempre realizando a mesma ação. Por outro lado, se o tempo de duração da regra é muito curta, todo o *overhead* adicionado com o *packet-in* torna a experiência de utilização da rede muito ruim, aumentando a latência nas aplicações. Dessa forma, o administrador da rede definida por *software* deve encontrar um tempo mediano, em que se utilize das vantagens da reconfiguração da rede, mas que não acrescente a latência. De acordo com [37], o tempo de duração da regra SDN na tabela de fluxo deve ser de 10 segundos.

Com a separação do plano de controle e do plano de dados, a rede se torna programável (o nome Redes Definidas por *Software* deriva justamente dessa vantagem) assim o administrador da rede ganha um maior controle sobre o funcionamento da mesma [3]. Além disso, obtém uma maior escalabilidade e possibilidade de inovação, pois cada parte da rede pode crescer e incluir novos funcionamentos sem afetar as demais partes. O custo da rede também é reavaliado, pois os equipamentos de rede não necessitam dos software que os fabricantes colocam nos equipamentos. Placas de baixo custo que entendem o protocolo de comunicação com o controlador podem ser inseridos na rede. Outra vantagem é a possibilidade de evolução, pois com a rede programável, as possibilidades de sua utilização serão limitadas pela criatividade do programador. Sendo assim, SDN possibilita uma complexidade de gerência e operações, que as redes tradicionais não oferecem [38].

Por outro ladro, tarefas básicas realizadas por equipamentos tradicionais, por exemplo um simples encaminhamento, deverá ser definido e programado no controlador. O *switch* SDN não implementa nenhum protocolo de encaminhamento ou de roteamento para popular sua tabela de rotas. É necessário que o controlador insira suas regras e ações para o *switch* realizar a função proposta.

Portanto, com o controlador está inserido o plano de controle, e no *switch* o plano de dados, há uma necessidade de comunicação entre esses elementos. Para isso, foi criado, na Universidade de Stanford, o protocolo Openflow [39]. O Openflow é um protocolo para estabelecer uma sessão de controle definindo a estrutura da mensagem para trocar modificações nos fluxos (*flowmods*), coletar estatísticas, e definir a estrutura fundamental de portas e tabelas de um *switch*. Atualmente, o protocolo Openflow está especificado na sua versão 1.4.0 [39]. A troca de informação entre o controlador e o *switch* é feita utilizando criptografia [3].

Contudo, nas redes definidas por *software*, a centralização das decisões podem acarretar alguns problemas, principalmente ligados à segurança da arquitetura. Com o paradigma SDN, é inserido também a preocupação em relação ao controlador, porque, se esse elemento não

está ativo, toda a rede pode ser penalizada. Além de requerer alguma forma de redundância, é necessário também proteger toda a lógica da rede contra inserções de código não autorizadas. Além disso, a comunicação entre controlador e *switch* insere um *overhead*, já que o *switch* não sabe o que fazer, levando a uma demora a mais nos primeiros pacotes dos clientes.

2.4.1 Plano de Dados - *Switch* programável de código aberto

Na literatura, quando se trata na realização de experimentos em redes SDN, normalmente utiliza-se o Emulador Mininet [40]. O Mininet é uma maneira rápida e fácil para prototipação de novos sistemas na rede SDN, oferecendo um ambiente emulado contendo vários nós independentes [41].

Apesar de ser bastante utilizado na literatura, o Mininet é ainda uma emulação da rede em apenas um sistema operacional, compartilhando uma única máquina física. Essa característica pode acarretar em divergência de resultados quando comparado aos experimentos realizados em uma rede SDN real.

Em contrapartida, *switches* SDN são de alto custo, dificultando a realização de experimentos em uma rede real SDN. Além disso, os fabricantes estão projetando os equipamentos SDN com uma arquitetura fechada, em que o projetista da rede não pode experimentar modificações no plano de dados, por exemplo. Isso, acarreta em uma limitação do potencial inovador que as redes definidas por *software* sugere. Diante disso, existe proposta de *switch* programável de código aberto [4, 12] que utiliza equipamentos tradicionais de baixo custo, alterando o *firmware* para executar como um equipamento SDN. Para isso é necessário, além do *hardware* de baixo custo, um Sistema Operacional (SO) compatível com o *hardware* (pois esse Sistema Operacional irá substituir o SO que vem de fábrica no equipamento), e por fim um sistema de encaminhamento via *software*.

De um modo geral, um *switch* SDN pode ser definido de acordo com a Figura 2.7, formado por *hardware* e *software*, tendo como componentes: Comunicação, Sistema Operacional e Plano de Dados. O componente Comunicação é responsável entre as trocas de informações entre o controlador da rede e o *switch*, implementando o protocolo Openflow [39] ou outro equivalente. A estrutura Tabela de Fluxo, quando implementada em *software*, é considerada como parte integrante do componente Comunicação.

De acordo com Mafioletti [4], o componente Sistema Operacional tem como função a tradução de alto nível das interfaces (por exemplo as mensagens Openflow) para a linguagem de baixo nível, compreendida pelo *hardware*. A Tabela de Fluxo pode ser implementada direta-

mente no *kernel*, possibilitando assim uma melhor performance, comparada a implementação em *software* fora do *kernel*, em contrapartida, tem uma memória mais limitada, garantindo poucas regras inseridas.

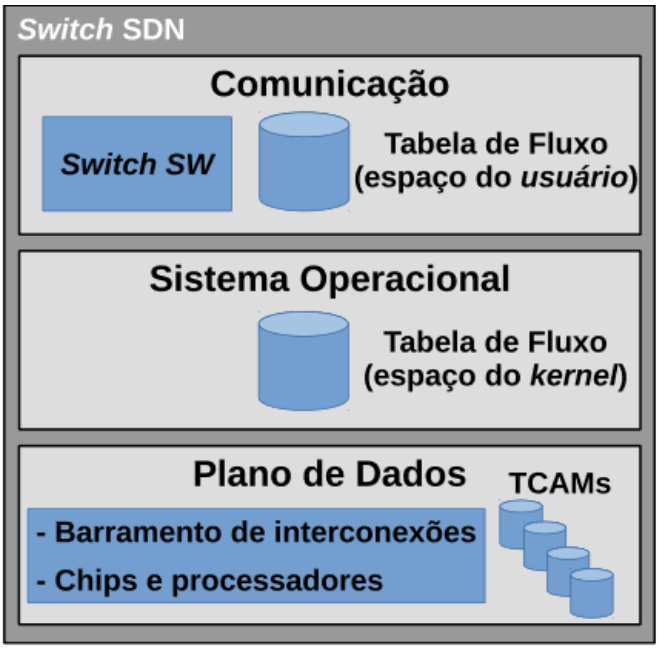


Figura 2.7: Arquitetura de um *switch* SDN. Adaptado de Mafioletti [4].

Por fim, o componente Plano de Dados é responsável por implementar a conectividade entre as portas do *switch*. Quando a Tabela de Fluxo é implementada em *hardware*, ela compõe uma TCAM (*Ternary Content-Addressable Memory*. Apesar de ter um alto custo para implementar e um consumo considerável de energia, a TCAM consegue realizar uma busca em todas as entradas da tabela com um único ciclo de *clock* do processador [42].

Este trabalho optou pela utilização do equipamento da Mikrotik RouterBoard, que são equipamentos extremamente utilizados em pequenas e médias empresa de Internet, e tem um custo aceitável. Quanto ao Sistema Operacional, foi escolhido utilizar o OpenWRT [43], pois é um SO *open source*, sendo flexível, e tem compatibilidade e suporte com o *hardware* da Mikrotik. Por fim, optou-se por utilizar o *Open vSwitch* (OvS) [44], um *switch* virtual com integração ao *kernel* do Linux e que segue a implementação do Openflow.

Dessa forma, foi retirado o RouterOS do equipamento Mikrotik RouterBoard, e inserido o OpenWRT como o Sistema Operacional e com o *Open vSwitch* realizando o encaminhamento via *software*. A arquitetura do *switch* programável de código aberto pode ser verificado na Figura 2.8, com sua estrutura em cada área modificada (*Hardware*, Sistema Operacional e Encaminhador em *Software*). Em [4, 12], essa mesma configuração foi testada e obtida resultados satisfatórios comparado a outros equipamentos SDN. Nos testes realizados, por exemplo, a pla-

taforma RouterOS tem um consumo maior de CPU, mas oferecendo apenas 0,28% maior vazão em tráfego TCP comparado a proposta do *switch* programável de código aberto. Ou seja, a arquitetura RouterOS tem uma vazão quase igual, mas com consumo de CPU maior, e ainda a desvantagem de trabalhar apenas a versão 1.0 do Openflow [39], enquanto que a proposta entende até a versão 1.4 do Openflow.

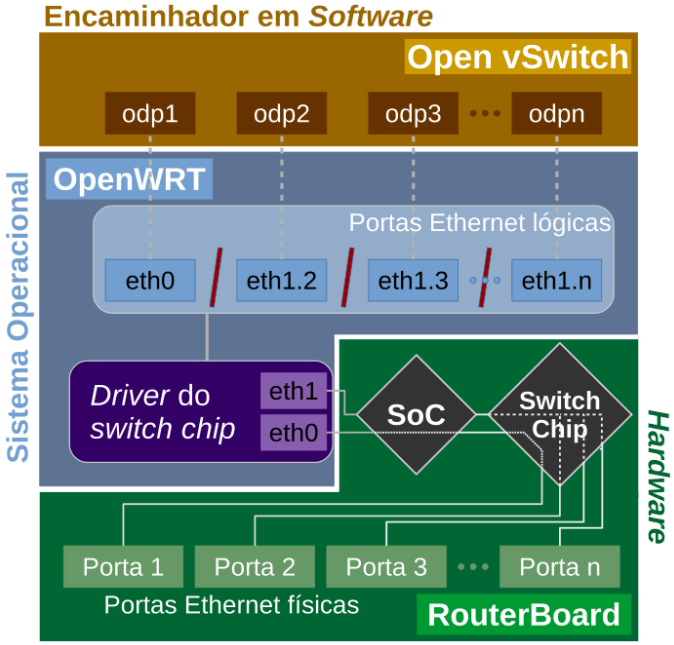


Figura 2.8: Arquitetura do *switch* programável de código aberto. Adaptado de Mafioletti [4].

2.4.2 Plano de Controle

Em uma rede SDN, o plano de controle é gerido pelos Controladores, que nada mais são do que um Sistema Operacional de Rede (NOS, sua sigla em inglês). Os controladores são elementos críticos na rede SDN, pois é a peça chave para o gerenciamento e configuração da rede. Assim, o administrador da rede não se preocupa de como realizar a interação entre o plano de controle e o plano de dados, pois, os controladores oferecem uma API, que realiza uma abstração dessa implementação de baixo nível. Utilizando a API, programar uma rede SDN se assemelha a desenvolver uma aplicação em um linguagem de programação qualquer.

Na literatura existe diversos controladores desenvolvidos por grupos diferentes e com filosofias e intensões diferentes. Na Tabela 2.1 tem-se um quadro com o resumo das características dos principais controladores. O Nox [45] foi um dos primeiros controladores desenvolvidos e disponibilizado para a comunidade. Escrito em C++, o Nox foi pensado e tem como característica de ser voltado para o alto desempenho. Atualmente, possui suporte à versão 1.0 do

OpenFlow, entretanto existe uma versão modificada pelo CPqD (Centro de Pesquisa e Desenvolvimento em Telecomunicações) que suporta parcialmente a versão 1.3 [46].

Tabela 2.1: Tabela com o resumo das características dos principais Controladores. Adaptado de Kreutz *et al.* [3].

Controlador	Suporte Openflow	Licença	Linguagem API
Nox	1.0	GPLv3	C++
Pox	1.0	GPLv3	Python
OpenDayLight	1.0 e 1.3	EPL v1.0	Java
Beacon	1.0	GPLv2	Java
Floodlight	1.1 e 1.3	Apache	Java
Maestro	1.0	LGPLv2.1	Java
Ryu	1.0, 1.1, 1.2, 1.3, 1.4 e 1.5	Apache 2.0	Python

Uma derivação do controlador Nox surgiu, com objetivo de oferecer uma interface mais amigável e simples para controladores SDN. O Pox [47] é uma versão do controlador Nox escrita em Python. Muito utilizado para prototipação e experimentos.

Os controladores OpenDayLight, Beacon, Floodlight e Maestro são controladores escritos na linguagem Java [3]. Com objetivos bastante semelhantes, de acelerar o processo de popularização do uso de SDN. Todos esses controladores suportam o Openflow 1.0, com exceção do OpenDayLight e do Floodlight que suporta também o Openflow 1.3.

O RYU [11] é um controlador *open source*, escrito em Python. A vantagem desse controlador é a possibilidade de utilizar outros protocolos, além do Openflow, como por exemplo o NetConf e OF-Config. O desenvolvimento constante por parte da comunidade permite que o controlador esteja atualizado com a versões mais recentes do Openflow. Atualmente, possui suporte total até a versão 1.5 do Openflow.

Este trabalho escolheu utilizar o controlador Ryu por sua API facilitada e pela constante atualização, sendo possível utilizar as versões mais novas do protocolo Openflow.

2.5 *Selective Verification in Application Layer - SeVen*

A estratégia *SeVen* [1] é um mecanismo de defesa contra ataques de negação de serviço na camada de aplicação (ADDoS), principalmente no que se refere a ataques do tipo *low-rate*. O

SeVen é baseado em ASV (*Adaptive Selective Verification*) [48], com a diferença de que o ASV é proposto para ataques na camada de transporte.

O algoritmo do *SeVen* é realizado durante um determinado tempo, chamado de rodada. Durante a rodada, o *SeVen* recebe as requisições e as mantêm em dois *buffers* (*P,R*), sendo que o *buffer P* representa as requisições parcialmente processadas e *R* são as requisições que podem ser processadas após o tempo da rodada. Tanto *P* e *R* tem um tamanho máximo, chamado *k*, que é a quantidade que os *buffers* podem armazenar.

Durante uma rodada, o *SeVen* não realiza a verificação seletiva caso o número de requisições for menor ou igual a *k*. Se for maior e chegar uma nova requisição, o *SeVen* tem duas opções para realizar: a primeira é descartar a requisição; a segunda é aceitar a nova requisição. Sendo a segunda escolhida, alguma requisição armazenada deve ser substituída pelo nova. A decisão entre as duas opções é escolhida por meio de uma distribuição de probabilidade.

Dessa forma, quando o servidor está sofrendo ataque, a probabilidade de descartar um atacante é muito maior do que um cliente legítimo, pois, os atacantes enviam várias requisições a fim de tornar o servidor indisponível. Nos experimentos realizados por Dantas [1], *SeVen* conseguiu propor uma disponibilidade para 95% dos clientes honestos, durante ataques do tipo *low-rate*. Para os ataques do tipo *high-rate*, não foram realizados testes para verificar a eficácia do *SeVen* com ataques *flooding*. O presente trabalho realizou testes com ataques *high-rate* e *SeVen*, e estão descritos na Seção 4.3.1, mas os resultados não foram bons comparando com os ataques *low-rate*, oferecendo disponibilidade apenas a 13,8% dos clientes legítimos.

Para melhor entender a defesa, supõe-se uma aplicação *web* que possa atender, no máximo, até 4 requisições simultâneas. Iremos chamar seu *pool* de atendimento de *P*. Requisições são representadas pela tupla $\langle id, estado \rangle$, o valor de *id* identifica a requisição e *estado* representa o estado da requisição: *PROC* (requisição está sendo atendida) ou *ESP* (requisição está aguardando atendimento). Agora, supõe-se que a aplicação esteja atendendo 3 requisições simultâneas, assim temos o *pool* com a seguinte configuração:

$$\mathscr{P}_0 = [\langle id_1, \mathsf{PROC} \rangle, \langle id_2, \mathsf{PROC} \rangle, \langle id_3, \mathsf{PROC} \rangle]$$

Agora, uma nova requisição id_4, $\langle id_4, \mathsf{ESP} \rangle$ chega a aplicação. Como o *pool* de atendimento ainda possui espaço para atender novas requisições, id_4 é simplesmente adicionada e atendida:

$$\mathscr{P}_1 = [\langle id_1, \mathsf{PROC} \rangle, \langle id_2, \mathsf{PROC} \rangle, \langle id_3, \mathsf{PROC} \rangle, \langle id_4, \mathsf{PROC} \rangle]$$

Logo após, uma nova requisição id_5, $\langle id_5, \mathsf{ESP} \rangle$ é recebida pela aplicação, o *SeVen* detecta que o *pool* está na sua capacidade máxima e deve decidir se id_5 será atendida ou não, isso é feito pelo resultado obtido na função de probabilidade FP_1 (seus detalhes estão descritos em [1, 8]). Caso FP_1 decida que a requisição id_5 não será atendida, uma mensagem de erro é enviada e a conexão fechada. Assim:

$$\mathscr{P}_2 = [\langle id_1, \mathsf{PROC} \rangle, \langle id_2, \mathsf{PROC} \rangle, \langle id_3, \mathsf{PROC} \rangle, \langle id_4, \mathsf{PROC} \rangle]$$

Porém, caso FP_1 decida que a nova requisição id_5 deva ser processada, uma segunda função, de distribuição uniforme (FP_2) decidirá qual requisição dentre as que estão sendo processadas atualmente deve ser eliminada do *pool* de atendimento. Supondo que a FP_2 decidiu que a requisição id_3 deva ser eliminada, temos:

$$\mathscr{P}_3 = [\langle id_1, \mathsf{PROC} \rangle, \langle id_2, \mathsf{PROC} \rangle, \langle id_5, \mathsf{PROC} \rangle, \langle id_4, \mathsf{PROC} \rangle]$$

SeVen funciona pois quando um servidor encontra-se sobrecarregado, ele muito provavelmente estará sofrendo um ataque DoS. Consequentemente, o seu *pool* de atendimento estará repleto de conexões atacantes, ou com mais atacantes do que clientes. Portanto, quando o *SeVen* decide processar uma nova requisição, a chance de remover uma conexão atacante do *pool* é muito maior do que de um cliente legítimo [8].

A metodologia utilizada no *SeVen* para HTTP também foi utilizada para mitigar ataques de negação de serviço contra aplicações VoIP (*Voice over IP* - em inglês) [49, 50], sendo adaptado a estratégia para esse tipo de aplicação.

2.6 Considerações do Capítulo

Este capítulo apresentou temas acerca da proposta deste trabalho. Na primeira seção do capítulo foi apresentado conceitos básicos que envolve a Segurança da Informação, principalmente sobre a Disponibilidade, conceito que este trabalho visa garantir. Ainda na primeira seção, foi discutido a utilização de *whitelist* e *blacklist* dentro da esfera de segurança da informação. Posteriormente, foi discutido os Ataques de Negação de Serviço (DoS - *Denial of Service* - em inglês), que visa impossibilitar o acesso de um usuário a uma informação. Especificamente, foi tratado sobre os ataques na camada de aplicação do modelo OSI, inclusive diferenciando ataques do tipo *low-rate* e o tipo *high-rate*, sendo o segundo o foco deste trabalho.

Na terceira seção abordou-se sobre as utilizações das técnicas de *fingerprinting*, desde o seu conceito inicial, até ao conceito utilizado por este trabalho. Continuando, a próxima seção discutiu sobre as Redes Definidas por *Software* (SDN - *Software Defined Network* - em inglês). Um novo paradigma no gerenciamento e utilização das redes de computadores, permitindo uma maior usabilidade e possibilidade da utilização da rede. Posteriormente, a discussão se deteve aos *Switch* programável de código aberto, um equipamento de baixo custo proposto para as redes SDN. Forma eficaz e barata para realizar experimentos reais na rede SDN, contrapondo o emulador de redes SDN, o Mininet. Abordou-se também acerca do plano de controle da redes SDN, os diversos controladores existentes, e a escolha do controlador para ser utilizado neste trabalho.

Por fim, encontra-se um breve relato da proposta *SeVen* (*Selective Verification in Application Layer*) que é uma defesa contra ataques de negação de serviço na camada de aplicação, do tipo *low-rate*.

Todos esses conceitos são importantes para o trabalho, visto que será proposto a utilização de *whitelist* dinâmicas baseadas em *fingerprinting* para mitigar ataques de negação de serviço na camada de aplicação, especificamente do tipo *high-rate*. O próximo capítulo é apresentado a proposta da defesa e a especificação da utilização dos conceitos discutidos.

Capítulo 3

WHITELIST DINÂMICA BASEADAS EM FINGERPRINTING: TÉCNICAS PARA MITIGAR DOS

Este presente capítulo se destina a apresentação e discussão da utilização de *whitelist* dinâmicas, populadas por meio de *fingerprinting* de clientes legítimos, e uma seleção para descarte das requisições, para mitigar ataques de negação de serviço.

3.1 Whitelist dinâmica

Como as requisições de clientes e atacantes são semelhante, ambas utilizando corretamente o protocolo HTTP, é muito difícil distinguir se uma requisição é de uma atacante ou é legítima. Por exemplo, em um ataque Get-Flooding, um atacante recruta uma botnet que envia requisições *GET* a um alvo que por sua vez também recebe requisições *GET* de clientes legítimos (que podem estar na mesma rede da botnet recrutada).

Nesses cenários, a utilização de *blacklist* não é recomendada, pois, não é simples afirmar que um determinado tráfego é malicioso. Por outro lado, pode-se ter uma melhor afirmação acerca de que um tráfego é um cliente, de acordo com algumas situações específicas exclusivas de um cliente. Por exemplo, espera-se que um usuário é legítimo caso ele tenha realizado um *login* na aplicação ou realizado uma operação esperada na página. Assim, a utilização de uma *whitelist* é mais recomendada, pois sabe-se quem colocar nessa lista.

O maior questionamento que se faz é como identificar um cliente para inseri-lo nessa lista, pois, como foi dito anteriormente, uma análise do tráfego não é suficiente para distinguir legítimos de maliciosos, dado que um usuário realizando um ataque pode estar na mesma rede pri-

vada que um cliente honesto, e ambos utilizando o mesmo endereço IP público. Dessa forma, é necessário resgatar algumas informações de usuários, realizando assim um *fingerprinting* desse cliente, e com essas informações, comparando-as com padrões de clientes legítimos, adicionar este cliente na *whitelist*. A primeira, proveniente de cabeçalhos do HTTP e do HTTPS [51], que, por meio do campo *user agent*, é possível resgatar informações acerca do navegador, versão do navegador, sistema operacional e configurações de linguagem utilizado pelo cliente. A Figura 3.1 exemplifica cabeçalhos utilizados nos protocolos HTTP e HTTPS, destacando que o usuário está utilizando o navegador Google Chrome, no sistema operacional Windows, resgatado pelo campo *User-Agent*.

```
Hypertext Transfer Protocol
▾ GET / HTTP/1.1\r\n
  › [Expert Info (Chat/Sequence): GET / HTTP/1.1\r\n]
    Request Method: GET
    Request URI: /
    Request Version: HTTP/1.1
  Host: www.brasil.gov.br\r\n
  Connection: keep-alive\r\n
  Cache-Control: max-age=0\r\n
  Upgrade-Insecure-Requests: 1\r\n
  User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/54.0.2840.99 Safari/537.36\r\n
  Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8\r\n
  Accept-Encoding: gzip, deflate, sdch\r\n
  Accept-Language: pt-BR,pt;q=0.8,en-US;q=0.6,en;q=0.4\r\n
```

Figura 3.1: Exemplo de cabeçalho do HTTP.

Outro campo que é possível resgatar de acordo com o cabeçalho do protocolo é o método utilizado. Os protocolos HTTP e HTTPS tem diversos métodos, que são sinalizações para o servidor, da ação realizada pelo cliente. Por exemplo, na Figura 3.1 o cliente está realizando um *GET*, que é um pedido de documento ao servidor *web*. Outro método importante do HTTP é o *POST*, que é semelhante ao método *GET*, mas com a possibilidade de se passar parâmetros para o servidor no corpo da requisição. O método *POST* é muito utilizado em páginas com preenchimento de formulários.

De acordo com Takasu *et al.* [52], é possível saber se o navegador do cliente está habilitado para utilizar *Cookies*. Os *Cookie* são utilizados para facilitar e tornar mais rápida o acesso do cliente em páginas que já foram visitadas. Dessa forma, *scripts* de ataques não utilizam os *Cookies*, pois, além de não ter a necessidade de utiliza-los para resgatar informações da página, os *Cookies* ocupam espaço em disco. Ainda de acordo com Takasu *et al.* [52], é possível resgatar a quantidade de espaço destinada para os *Cookies* e a quantidade de espaço destinada para a sessão do navegador com a aplicação *web*.

Outra informação, acerca dos navegadores, que é possível resgatar e utilizar para distinguir clientes legítimos e atacantes é sobre os *plugins* instalados no navegador. *Plugins* são pequenos programas, que são integrados aos navegadores, e realizam alguma função específica para auxi-

liar a experiência do usuário. *Scripts* de ataques não utilizam *plugins*, assim, pode-se diferenciar clientes e atacantes.

Além de informações acerca do *software* de clientes, é possível também resgatar informações do *hardware* utilizado [52]. Resolução da tela, quantidade de cor utilizada, orientação do terminal, se a interface é *touchscreen*, tipo de fontes utilizada no Sistema Operacional, a taxa de pixels do equipamento, quantidade de núcleos de processadores, possibilidade de identificar o tipo de GPU, o estado da bateria, quantidade de espaço livre no equipamento e se há algum tipo de microfone e câmera instalada no computador.

Sendo assim, se verifica que é possível distinguir um cliente legítimo de um atacante, não apenas com o tráfego, pois os dois se assemelham. Um cliente legítimo pode ser identificado a partir de diversas informações que podem ser recuperadas, tanto de *software* como de *hardware* e assim inserido na *whitelist*. Estando nessa lista, suas requisições serão identificadas como oriundas de um cliente legítimo.

Por fim, pode-se utilizar um *fingerprinting* da utilização e da interação do cliente com a página *web*, ou seja, a forma que o cliente legítimo interage com a página pode ser medida, quantificada e qualificada, para que essa forma de utilização se caracterize no padrão de um cliente legítimo. Esse padrão de utilização seria caracterizado de acordo com a página de cada servidor *web*.

Portanto, este trabalho propõe duas formas de realizar a identificação de um usuário legítimo:

- **Critério Login**: A primeira forma, aproveita-se uma aplicação *web* que requisita o *login* de um usuário. Dessa forma, por meio da utilização do método *POST* (utilizado para preencher formulários) e do cabeçalho *user agent* do HTTP, é possível identificar um usuário legítimo;
- **Critério sem Login**: A segunda forma é destinada para páginas *web* que não tem sistema de *login*, em que se verificaria a forma de utilização e de interação da página de um cliente legítimo. Assim, se esse padrão de uso for identificado, um usuário será identificado também.

As duas propostas são discutidas nas seções seguintes.

Após a inserção do cliente na *whitelist*, essa entrada permanece por um período de tempo, conferindo assim o aspecto dinâmico da lista. Neste trabalho, o cliente fica na *whitelist* enquanto houver requisição desse usuário sendo trafegada, caso o cliente passe 60 segundos sem enviar

nenhuma nova requisição, ele é retirado da *whitelist*. Assim, caso volte a requisitar após 60 segundos, será realizado novamente o *fingerprinting* desse usuário e inserido na *whitelist*.

3.2 Reconhecendo cliente legítimos

Como verificado anteriormente, reconhecer um cliente legítimo não é trivial quando se trata de ataques de negação de serviço na camada de aplicação. Mas é possível por meio da junção de informações de clientes legítimos.

Este presente trabalho propõe identificar um cliente de duas formas: identificação utilizando um cruzamento de informações obtidas do cabeçalho HTTP, que é o critério de *login*; e a identificação do cliente por meio da sua utilização da página acessada, que é o critério sem login. Ambas propostas foram implementadas e realizadas experimentos para verificar se a identificação e a priorização do tráfego conseguem mitigar ataques de negação de serviço. Os resultados das propostas estão relatadas na Seção 4.3.4 e 4.3.5, respectivamente.

3.2.1 Identificação do usuário utilizando cabeçalho do HTTP

Diante das diversas formas de se realizar a identificação de um cliente legítimo, a mais simples e intuitiva é por meio de informações específicas do usuário que são informadas por meio do cabeçalho do protocolo HTTP.

Essa forma de identificar o cliente utiliza uma aplicação *web*, em que o usuário deverá utilizar um *login* e senha para utilizar o sistema. Dessa forma, se o sistema reconhece que um determinado usuário é legítimo, pois entrou com um *login* e senha reconhecidos pela aplicação, então o sistema de defesa pode inferir também que esse usuário é um cliente legítimo, e dar prioridade no seu tráfego quando houver um ataque de negação de serviço.

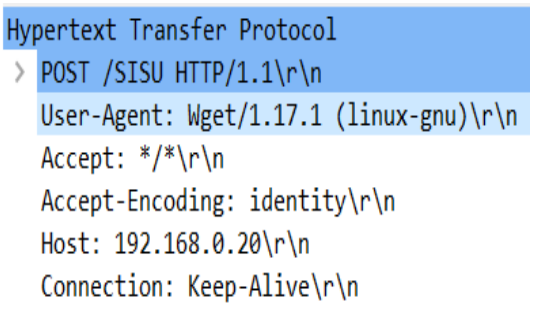
Para isso, o sistema de defesa resgata algumas informações para identificar o cliente legítimo. A primeira informação utilizada é o método *POST* do protocolo HTTP [51]. O HTTP tem dois métodos para solicitar uma página, o *GET* e *POST*. Em ambos, é possível enviar dados como parâmetros para a aplicação *web*. Para preenchimento de formulário (o que inclui a página de *login* e senha), normalmente os servidores *web* utilizam o método *POST*, pois, este método é mais seguro que o *GET*. Enquanto que o *POST* envia os parâmetros de forma mais segura, no campo opcionais do cabeçalho, o método *GET* envia os dados junto a URL (*Uniform Resource Locator* - em inglês) acessada, podendo ser visualizada mais facilmente.

Assim, necessariamente, quando um usuário estiver preenchendo o formulário de *login* e

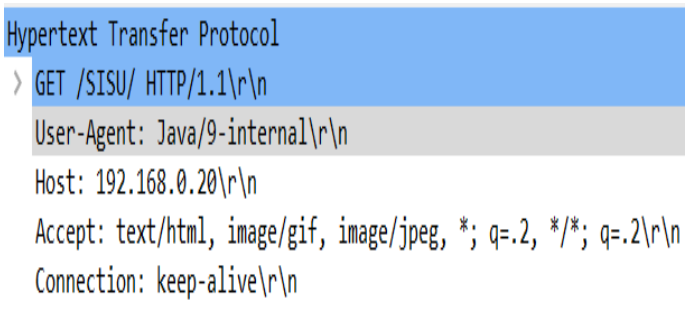
senha, o navegador utilizará o método *POST*. Dessa forma, o sistema de defesa pode resgatar a informação e identificar que essa requisição pode ter sido enviada por um provável cliente legítimo, pois, além de utilizar o método destinado a autenticação, a aplicação *web* respondeu com sucesso. Dessa forma, o sistema de defesa pode inferir que se trata de um usuário da aplicação.

A simples utilização do método *POST* não é garantia que aquele usuário seja um utilizador legítimo e não atacante, pois, o cliente malicioso pode alterar seu ataque para utilizar o *POST*. Assim, o sistema proposto utiliza ainda de outra informação oriunda do cabeçalho.

Essa informação está contida no campo chamado *user agent*. Este campo contém informações sobre o navegador que o usuário está utilizando, a versão desse navegador, o sistema operacional que está executando no cliente e o idioma. Se o usuário não utilizou um navegador para solicitar a página, neste campo encontra-se o nome da aplicação utilizada. Por exemplo, na Figura 3.2(a) a solicitação da página foi realizada pelo programa *Wget*, que é uma aplicação do terminal do linux para realizar uma requisição ao servidor *web*. Se a aplicação não for conhecida, no campo *user agent* aparecerá o nome da linguagem de programação utilizada para desenvolver essa aplicação, como exemplificado na Figura 3.2(b), que utilizou um programa escrito na linguagem Java.



(a) *User agent* apresentado aplicação *Wget*.



(b) *User agent* apresentando a linguagem Java, utilizada no desenvolvimento da aplicação.

Figura 3.2: Exemplo de preenchimento do campo *user agent*.

Assim, a identificação de um cliente legítimo é realizado em duas etapas, como mostrado na Figura 3.3: a primeira o sistema de defesa verifica se as requisições que estão chegando estão utilizando o método *POST* e se realizou o *login* corretamente. Se o método não for o correspondente, é então finalizada o *fingerprinting* dessa requisição e não é identificada como um cliente legítimo. Se está utilizando o método *POST* e houve uma resposta de sucesso da autenticação na aplicação, então a requisição passa para a segunda fase de verificação, em que é analisado o *user agent*. Essa análise consiste em verificar se esse campo do cabeçalho tem um valor válido, ou seja, que tenha valores que são utilizados atualmente [53].

Caso não tenha um *user agent* válido, a requisição do usuário não é adicionada na *whitelist* e não terá preferência nas suas requisições. Mas, se tiver com valor válido, o usuário será adicionado na *whitelist*. Assim, as novas requisições desses usuários serão encaminhados diretamente ao servidor *web*, e serão atendidas.

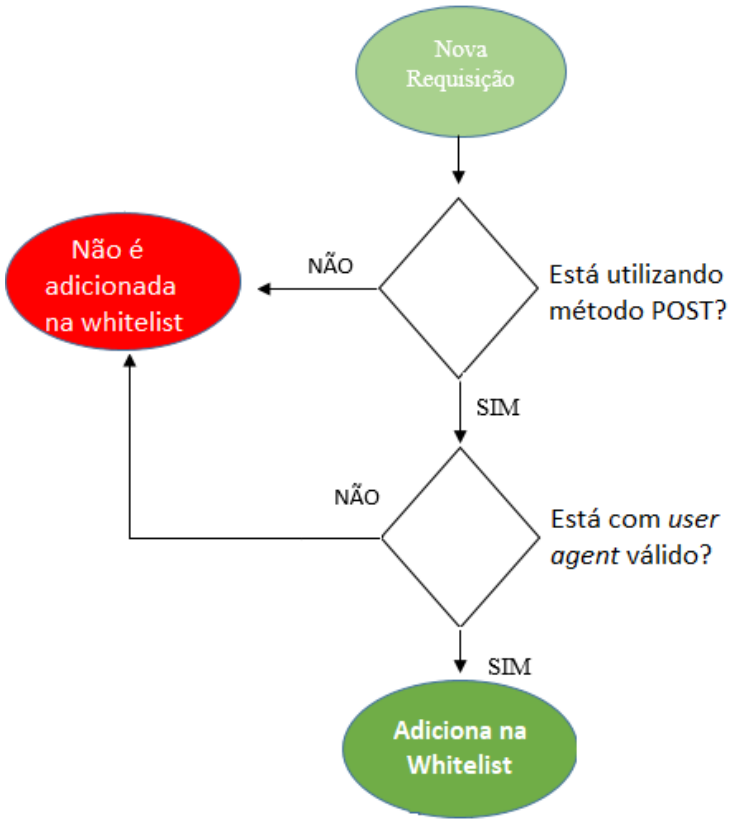


Figura 3.3: Fluxograma do sistema de identificação do cliente, por meio do cabeçalho HTTP.

Uma desvantagem dessa forma de identificar um cliente legítimo, é se o atacante tentar utilizar técnicas para descobrir as senhas de acesso ao sistema *web*, utilizando ataques de força bruta, por exemplo. O sistema de defesa proposto não se preocupa com a segurança a nível de informação da aplicação, se detêm a mitigar ataques de negação de serviço. Mas, como sugestão, o desenvolvedor da aplicação *web* pode inserir o sistema de *captchas*, assim, todo usuário que for se autenticar no sistema, terá que completar a solicitação preenchendo o *captcha*. Além disso, a utilização de *captcha* também é uma forma de mitigar ataques de inundação, que utiliza o método *POST* para realizar o ataque [54].

Os resultados obtidos com essa forma de identificação dos clientes legítimos estão descritos na Seção 4.3.5.

3.2.2 Identificação do usuário por meio dos objetos da página

A forma de identificar um usuário legítimo descrita na Seção anterior pode ser utilizada em páginas *web* que tenham um sistema de autenticação com *login*. Mas não se torna eficiente em páginas que não necessitam de *login*. Assim, se faz necessário uma outra técnica de *fingerprinting* para identificar um usuário legítimo.

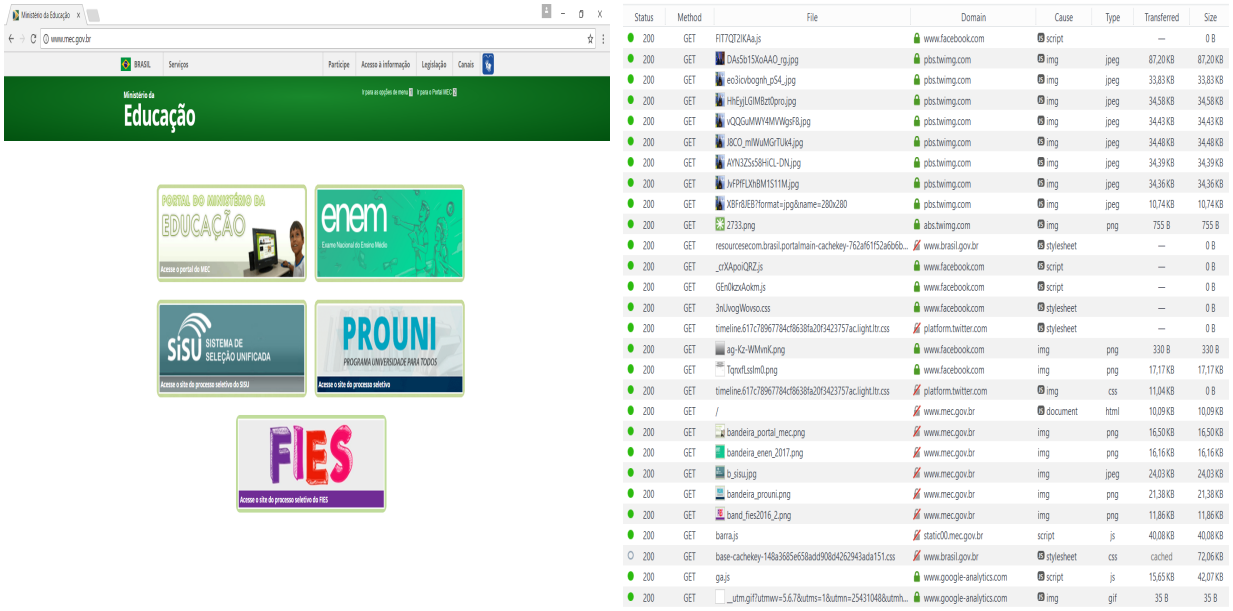
Essa outra forma de realizar o *fingerprinting*, é por meio do padrão de utilização de uma página *web*. Um cliente tem um padrão de utilização em uma determinada página, assim, percebendo esse padrão, todos os usuários que repitam ou se aproximem neste padrão serão identificados como usuários legítimos, podendo assim ter prioridade no seu tráfego.

Uma página *web* é composta por vários objetos dentro dela, que informam ao navegador como aquela determinada página deverá ser demonstrada ao usuário. Assim, a página é composta pelo HTML (*HyperText Markup Language* - em inglês), que é responsável por demarcar a posição de cada elemento na página; o CSS (*Cascading Style Sheets* - em inglês) tem como função adicionar estilo (cores, fontes, espaçamento etc) a um documento *web*. O Javascript é uma linguagem de programação inserida nas páginas *web*, oferecendo maiores possibilidades a um desenvolvedor. Tanto o CSS quanto o Javascript podem estar inserido no mesmo arquivo HTML ou em outro arquivo separado. Além disso, qualquer imagem ou outro elemento que compõem a página *web*, essa imagem/elemento são considerados como um objeto.

Dessa forma, um usuário para visitar, visualizar e utilizar um *site*, ele deverá obter todos os objetos dessa página para visualiza-la e interagir como foi desenvolvida. Por exemplo, na Figura 3.4(a) tem-se a página do Ministério de Educação do governo brasileiro. Para que o navegador apresente essa página da forma que está visualizada, foi necessário que realizasse o *download* de 28 objetos. Esses objetos podem ser visualizados na Figura 3.4(b).

Assim, um cliente legítimo pode ser identificado se esse usuário realizou o *download* dos objetos. Neste presente trabalho, um usuário foi considerado legítimo se for obtido 80% ou mais dos objetos da página. Foi escolhido 80% dos objetos por considerar que a quantidade de 4/5 é uma quantidade necessária para exibir e utilizar a página com um mínimo de informações nela contida. Dessa forma, o sistema de defesa monitora quantos objetos o usuário obteve, a cada novo objeto requisitado é incrementado um contador referente a esse cliente na estrutura de dados da defesa. Passando de 80%, o cliente é inserido na *whitelist* e terá prioridade no tráfego a partir daquele momento.

A Figura 3.5 tem um exemplo do fluxograma de execução da estratégia para identificar um cliente legítimo. Por exemplo, um *site* tem 10 objetos que compõe a página. A primeira



(a) Página *web* do MEC. (b) Objetos presentes na página do MEC.

Figura 3.4: A página e os objetos que compõe o *site* do MEC.

requisição chegará no sistema de identificação, o sistema verifica se é uma requisição de um objeto válido. Se não é, a requisição não realiza mais nenhum passo[1], e a requisição é retirada do sistema de identificação. Caso a requisição seja de um objeto válido, então é verificado se o usuário está inserido na estrutura de dados que guarda as informações de quantos objetos um determinado usuário solicitou. Se é a primeira requisição desse usuário, então insere-se as informações dele nesta estrutura de dados e a sua requisição é retirada do sistema de identificação, sem ser adicionada na *whitelist*.

Caso não seja a primeira requisição do usuário, ou seja, ele já está na estrutura de dados, então o sistema de defesa verifica se o cliente solicitou 80% dos objetos presentes na página. Se não chegou neste limite, então é incrementado a quantidade de objetos requisitados na estrutura de dados e a requisição é retirada do sistema de identificação. Caso seja 80% ou mais dos objetos solicitados por esse usuário, então o sistema identifica-o como um cliente legítimo, inserindo-o na *whitelist*, e assim todas as outras requisições terão prioridade neste servidor *web*.

Essa estratégia se mostra efetiva, pois um usuário realiza a solicitação dos objetos já no primeiro acesso a página. Assim, sendo identificado, toda a interação seguinte com a página terá a prioridade, não sendo mais necessário passar pelo processo de *fingerprinting* e nem de seleção, melhorando a comunicação entre servidor *web* e cliente.

A estratégia se mostra eficiente também em relação aos atacantes, pois, o atacante tem

[1]Ressalta-se que se não é um objeto válido, o servidor *web* retornará ao usuário um erro, informando que não existe esse objeto.

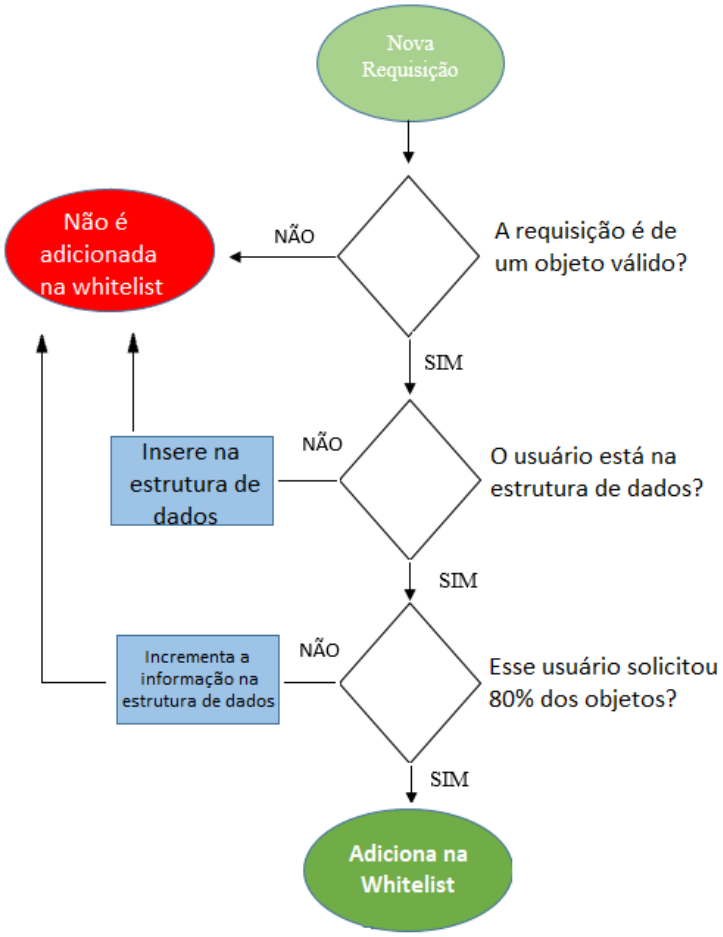


Figura 3.5: Fluxograma do sistema de identificação do cliente, utilizando os objetos da página.

como intenção o envio de várias solicitações, com um menor custo. O atacante poderia verificar quais objetos existem na página, para realizar a solicitações desses objetos. Mas para isso, o atacante deve fazer uma varredura no HMTL da página, verificar quais objetos existem, e realizar o *download*. Realizar essas etapas insere ao atacante mais processamento e largura de banda. Assim, se por um lado seria interessante ao atacante solicitar todos os objetos da página, por outro lado haveria uma maior sobrecarga no gerador do ataque. Se o atacante tem como premissa o menor custo, assim ele consegue uma maior efetividade realizando o *download* apenas do arquivo principal da página, e tem uma maior largura de banda para essas solicitações. De fato, as ferramentas utilizadas para gerar os ataques não realizam a busca e a solicitação dos objetos contidas no HTML da página.

Os resultados dos experimentos utilizando essa forma de identificar clientes legítimos estão descritos na Seção 4.3.5.

3.3 SeVen

A defesa *SeVen* foi introduzida por Dantas *et al.* [8], que visa propor uma defesa seletiva para mitigar ataques de negação de serviço na camada de aplicação. *SeVen* funciona como um proxy e monitora o uso do servidor *web* sob sua proteção. Quando o servidor *web* não está sobrecarregado, ou seja, o número de pedido que está sendo processado é menor do que a sua capacidade máxima, *SeVen* permite que qualquer requisição (mesmo de atacantes) possa ser processado. No entanto, quando o servidor *web* está sobrecarregado, isto é, já não consegue processar um novo pedido ao mesmo tempo, e chega um novo, *SeVen* decide (usando alguma função de probabilidade) se a aplicação *web* deve processar a nova requisição. Há dois resultados:

1. *SeVen* decide que não deve processar este novo pedido, assim simplesmente retorna ao cliente uma mensagem que o servidor não está disponível;
2. *SeVen* decide que ele deve processar este novo pedido. Em seguida, ele também deve decidir qual o pedido que está sendo processado no momento deve ser descartado. Esta decisão é tomada com base em outra distribuição de probabilidade.

Intuitivamente, as estratégias seletivas funcionam porque sempre que uma aplicação está sobrecarregada, é muito provável que ela está sofrendo um ataque. Em vez de bloquear todos os novos pedidos (de ambos, os atacantes e clientes legítimos) como feito sem a defesa, *SeVen* abre a possibilidade de nova solicitação de clientes legítimos serem processados, melhorando assim a disponibilidade do aplicativo. Dantas *et al.* [8] realizou testes com ataques de negação de serviço na camada de aplicação do tipo *low-rate* que também exploram vulnerabilidades do protocolo HTTP, mas que tem uma taxa de requisições extremamente baixa. Esses ataques são conhecidos como *Slowloris* e Ataque *POST*.

Embora *SeVen* funcione bem para mitigar ataques *low-rate*, ele sofre ao tentar mitigar ataques do tipo *high-rate*. Isso ocorre porque *SeVen* tem limitada memória e poder de processamento, e não pode lidar com a esmagadora maioria das requisições que chegam quando um aplicativo está sofrendo ataques de DDoS com altas taxas, como no ataque *Get-Flooding*. Quando o ataque é do tipo *low-rate*, como um ataque *Slowloris*, *SeVen* pode garantir serviço a 95% dos clientes legítimos [1]. No entando, quando um ataque é do tipo *High-Rate*, como *Get-Flooding*, *SeVen* só pode garantir serviço a uma média de 13,8% dos clientes legítimos. Os resultados dos experimentos da estratégia *SeVen* com ataques *high-rate* são apresentados na Seção 4.3.1.

Neste trabalho especificamente, *SeVen* foi utilizado: primeiro pela sua característica e possibilidade de mitigar ataques do tipo *low-rate*, incrementando a defesa proposta por este trabalho; segundo, pela proximidade que *SeVen* tem da aplicação defendida, pois estando junto a aplicação, *SeVen* consegue identificar qual o percentual de utilização da aplicação. Dessa forma, *SeVen* foi alterado para verificar e monitorar a carga atual da aplicação a qual está defendendo, e alimenta, com essa informação, todo o sistema de defesa proposto. Essa alteração é explicada na seção a seguir.

3.4 SHADE - Selective High Rate DDoS Defense

O sistema proposto nesta dissertação, chamada SHADE (**S**elective **H**igh r**A**te **D**DoS d**E**fense), utiliza o *SeVen* como monitoramento de carga, ou seja, além da estratégia seletiva que essa defesa proporciona, o *SeVen* pode indicar quando a aplicação sob sua proteção está sobrecarregado, recebendo muitas requisições. Além disso, é na aplicação *SeVen* que é realizado o *fingerprinting* dos clientes, ou seja, onde é verificado se foi realizado o *login* corretamente, ou se requisitou 80% dos objetos da página, dependendo da estratégia de identificação de clientes legítimos. Dessa forma, foi acrescentada mais uma função a ferramenta *SeVen*.

A proposta inicial do SHADE [10], verificada por meio de emulação, não se verificou eficiente quando foi realizado experimentos em ambiente real, obtendo uma disponibilidade para clientes abaixo do aceitável, devido principalmente ao fato da emulação sobrecarregar a máquina onde estava sendo executada. Por isso, o presente trabalho apresenta proposta diferente do trabalho inicial.

Para utilizar o *SeVen* como monitoramento de carga, além da sua estratégia seletiva, foi elaborado níveis de utilização, descrito abaixo:

- **Livre**: indica que o servidor *web* está em seu trabalho normal, ou seja, processa uma nova requisição quando ela chega. Na prática, a taxa considerada é até 75% da ocupação do servidor *web*. Essa taxa foi encontrada de acordo com experimentos realizados para verificar um valor adequado que indicasse o nível proposto;
- **Sobrecarregado**: indica que o servidor *web* está sendo muito utilizado e se continuar a receber mais pedidos irá atingir a sua capacidade máxima. A taxa de ocupação é maior que 75%.

Portanto, o *SeVen* fica constantemente monitorando a capacidade do servidor e indicando qual o nível de utilização. Estando sobrecarregado, o sistema SHADE entra em ação. Ou seja,

todo o tráfego destinado para o servidor *web* passaria antes pelo SHADE, que irá realizar uma seleção, descartando algumas requisições. Os que forem aceitos pelo SHADE, seguem o fluxo normal, sendo endereçado ao servidor *web* e recebendo atendimento.

Se faz necessária a seleção e o descarte, pois o *Get-Flooding* tem característica de enviar uma grande quantidade de requisições. Essas requisições, se chegam todas até o servidor *web*, geram indisponibilidade. Assim, de alguma forma, esse tráfego deve ser minimizado. Portanto, há chance de clientes legítimos serem descartados (o que deve ser evitado), mas a utilização de *whitelist* dinâmicas, proposta nesse trabalho, visa evitar que clientes sejam descartados. Ou seja, os clientes legítimos identificados com o *fingerprinting*, não são encaminhados para a seleção. Portanto, de um modo geral, o descarte permite que o servidor tenha níveis de disponibilidade.

Além disso, escolhe-se uma seleção dos pacotes e não o descarte total das requisições que não estão na *whitelist* porque algum cliente legítimo pode não ter sido identificado e inserido na *whitelist*. Assim, o cliente não identificado ainda tem chance de ser atendido, mesmo passando pelo método de seleção de SHADE. Por isso que foi escolhido utilizar a metodologia de *whitelist* e não de *blacklist*.

Os nossos experimentos demonstraram que o descarte de quatro requisições a cada cinco recebida é uma boa relação. Esse valor foi definido de acordo com experimentos variando a taxa de atacante e a taxa de descarte, e os melhores resultados de disponibilidade de clientes foi o descarte de quatro a cada cinco requisições. As informações sobre esses experimentos e a justificativa da quantidade do descarte estão relatadas na Seção 4.3.3. A seleção é feita sequencialmente, ou seja, a primeira requisição será aceita, enquanto que da segunda até a quinta será descartada por SHADE, e assim sucessivamente.

O redirecionamento do fluxo, que primeiramente iria ao servidor *web*, para o SHADE é realizado com uma regra de NAT (*Network Address Translate*) que pode ser inserida no roteador da rede. Tanto na rede SDN (caso dos experimentos deste trabalho), como em redes tradicionais, essa modificação do fluxo é possível, pois, basta inserir uma regra de redirecionamento no roteador da rede tradicional, no sistema operacional deste equipamento específico. A diferença estará na ativação, pois no caso da rede SDN, por meio do controlador, essa alteração é realizada quando o sistema reconhece Sobrecarregado. Na rede tradicional, a ativação deverá acontecer por meio de um *script* na linguagem SHELL, por exemplo.

A ação de redirecionar o fluxo só se inicia quando o *SeVen* detecta que o servidor *web* está no nível Sobrecarregado. Assim, *SeVen* envia a informação sobre o nível para o controlador da rede SDN, que por sua vez insere uma nova regra no *switch* para redirecionar o fluxo, que tinha como destino o servidor *web*, para o SHADE. Apenas os fluxos que não foram identifi-

cados e inseridos na *whitelist* serão redirecionados, os clientes que foram identificados com o *fingerprinting* serão direcionados diretamente ao servidor *web*.

Na Figura 3.6 tem-se um exemplo de regra que o controlador SDN insere no *switch*. Na direita temos um computador cliente, com endereço IP 10.0.0.50, que irá realizar uma requisição *web* para um servidor cujo endereço IP é 192.168.1.200. Com a regra SDN instalada no *switch*, quando o pacote chega no *switch*, o equipamento realiza o *Destination NAT* (*Network Address Translation* - em inglês), que é a alteração do endereço IP de destino. Ou seja, no pacote original, o cliente enviou para o IP 192.168.1.200, mas quando chegou no *switch* houve a alteração para 192.168.1.100. Da mesma forma o pacote de retorno, o servidor 192.168.1.100 responde ao cliente, quando chega no *switch* o IP de origem é alterado para 192.168.1.200, o endereço original que o cliente enviou. Essa alteração de volta é necessária para não comprometer a confiança do lado do cliente.

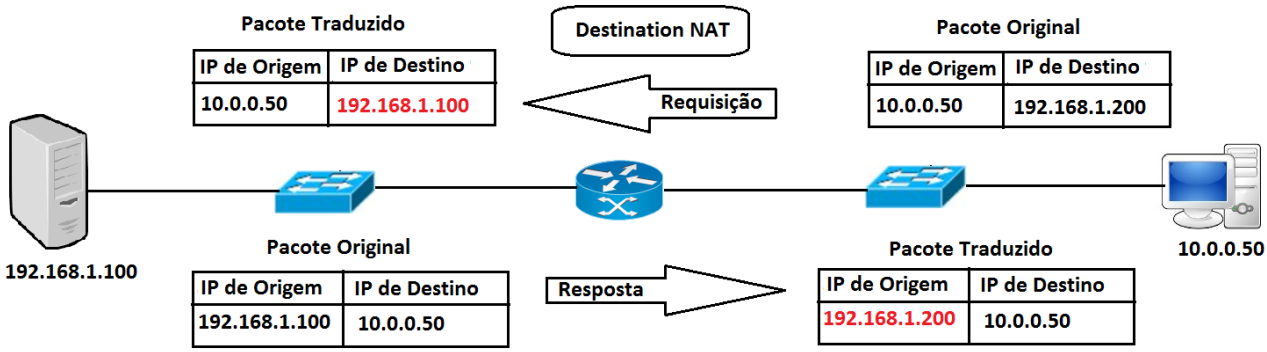


Figura 3.6: Exemplo da regra de NAT de alteração de fluxo.

3.5 Exemplo de Execução do SHADE com *whitelists* dinâmicas

O sistema age de acordo com a Figura 3.7. Quando recebe uma nova requisição é verificado se o usuário está inserido na *whitelist*. Se estiver, a requisição será aceita e processada. Se não está inserido é realizado então o *fingerprinting*, recolhendo informações da requisição e verificando se é ou não um cliente. Se a identificação for positiva, então esse cliente será adicionado na *whitelist* e seu tráfego terá privilégio durante um determinado tempo (o tempo de duração do usuário na *whitelist* durante os experimentos desse trabalho foi de 300 segundos). Caso não consiga determinar que é um cliente legítimo, então é verificado se o sistema está no nível sobrecarregado, se não estiver, a requisição é aceita. Se o sistema está sobrecarregado, a requisição é direcionada para o SHADE, que por meio da sua estratégia de seleção, decidirá se irá descartar ou aceitar a requisição.

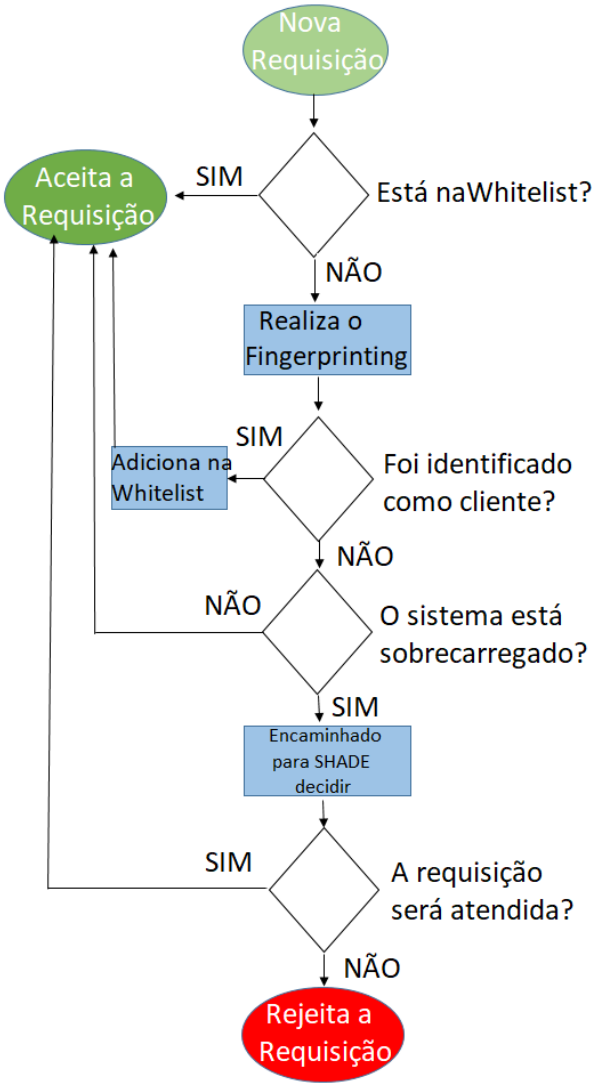


Figura 3.7: Fluxograma de execução do SHADE.

Exemplificando: Considere que um servidor *web* suporte 10 requisições simultaneamente. Portanto, entende-se que o servidor está no estado Livre se o servidor *web* estiver com até sete requisições e Sobrecarregado caso tenha oito ou mais requisições.

Assuma que no início o servidor está livre, sem requisições, e que chegam sete requisições. Quando essas requisições chegam, é realizado o *fingerprinting*, verificando se é um cliente legítimo. Se for um cliente, ele é inserido na *whitelist*, caso não seja identificado, não é realizado nenhuma ação e as requisições são aceitas.

Chega mais uma requisição de um outro cliente legítimo, somando oito no total. Assim, *SeVen* recebe a requisição, faz o *fingerprinting* e adiciona na *whitelist*. Além disso, *SeVen* altera para o nível Sobrecarregado, e assim todo o sistema SHADE entra em ação, alterando o fluxo que iria para o servidor *web* para SHADE, que irá realizar a seleção. Então, chegam mais sete requisições, sendo as duas primeiras de clientes na *whitelist*. Essas duas requisições de clientes

já identificados são redirecionadas diretamente para o servidor *web*, e as cinco restantes são direcionadas para SHADE. Chegando no SHADE, a seleção é realizada: a primeira requisição é aceita e direcionada para o servidor; as quatro seguintes serão descartadas.

3.6 Considerações do Capítulo

Neste capítulo foi discutido a proposta da defesa, considerando as diversas formas de identificação do cliente legítimo e colocando-o em uma *whitelist* dinâmica. Foram apresentadas duas formas de realizar a identificação: a primeira, leva em consideração o método HTTP utilizado na requisição e se o usuário está utilizando o valor do campo *user agent* válido; a segunda forma de identificar, se baseia no padrão de utilização do site por um cliente legítimo, por meio dos objetos que compõe a página, ou seja, quando um usuário solicita 80% ou mais dos objetos da página, este usuário é considerado um cliente legítimo.

A seção seguinte discutiu sobre a utilização do *SeVen*, modificando sua proposta inicial de defesa contra ataques do tipo *low-rate*. O *SeVen* é modificado para funcionar como monitoramento de carga, indicando a taxa de utilização do servidor *web* para todo o sistema de defesa.

Posteriormente foi discutida a proposta SHADE (*Selective High Rate DDoS Defense*), utilizando o monitoramento realizado por *SeVen*. Quando o *SeVen* avisa que está sobrecarregado, é realizado então um redirecionamento do fluxo das requisições, realizado pelo controlador da rede SDN. O tráfego que se destinaria para o servidor *web* é direcionado para o SHADE, que realiza uma seleção das requisições.

Na última seção do capítulo foi apresentado o fluxograma de execução da proposta, bem como um exemplo didático do funcionamento. O próximo capítulo se destina a abordar o cenário construído para validar a proposta, as ferramentas e as métricas utilizadas nos experimentos. A segunda parte do capítulo discute os resultados obtidos nos experimentos.

Capítulo 4

CENÁRIO E RESULTADOS

O presente capítulo se detém em descrever o cenário desenvolvido, as ferramentas utilizadas, as métricas avaliadas e a discussão dos resultados obtidos com a proposta de defesa para mitigar ataques de ADDoS, do tipo *high-rate*.

4.1 Cenário, ferramentas e métricas

4.1.1 Cenário

Para verificar a proposta, foi construído um cenário de acordo com a Figura 4.1, consistindo do Atacante, Servidor-*web*/*SeVen*, Cliente, SHADE e Equipamentos. Todo o cenário foi construído em ambiente real, utilizando máquinas virtuais (VM - *Virtual Machine*) em dois servidores de virtualização, com o *Hypervisor* XenServer 6.2 e VMWare ESXi 6.0. Cada VM utilizou uma interface física distinta, evitando assim o encaminhamento interno dos servidores de virtualização, e forçando o tráfego passar pelos equipamentos de rede.

Cada item do cenário contém as seguintes características:

- Atacante: Para criar o tráfego atacante, foram utilizados entre uma e quatro máquinas virtuais com Sistema Operacional Ubuntu Server 16.04 LTS, com 1 vCPU, 1 GB de memória e 20 GB de HD em cada VM. A ferramenta utilizada foi o *Goldeneye* [55], que gera ataque *Get-Flooding*. A quantidade de atacantes foi variada, verificando o impacto de diferentes níveis de tráfego. Nos experimentos, a quantidade de tráfego atacante chegou a ser até trinta vezes superior que o tráfego do cliente;
- Servidor-Web/SeVen: A VM que foi instalado a aplicação *web* tem Sistema Operacional Debian 8, 1 vCPU, 3 GB de memória e 50 GB de HD. Para hospedar toda a aplicação

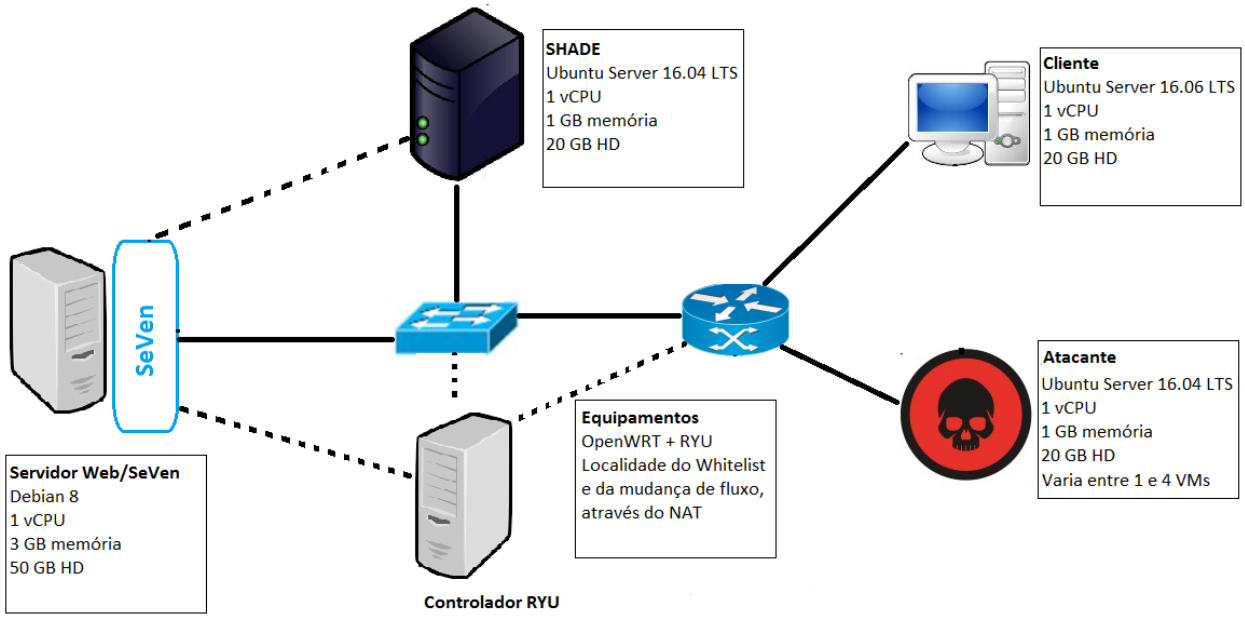


Figura 4.1: Diagrama do cenário, com informações de cada elemento utilizado, para realização dos experimentos.

web, foi utilizado o Apache Tomcat versão 8, suportando 200 clientes simultaneamente, correspondendo um servidor *web* de médio porte. Além disso, na mesma máquina, foi instalado e configurado o *SeVen*, realizando sua estratégia contra ataques do tipo *low-rate* e como monitoramento de carga;

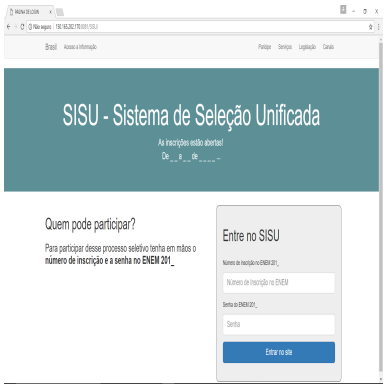
- Cliente: Os experimentos utilizaram 1000 clientes interagindo com a aplicação *web*, sendo gerado 10 a cada minuto. Os clientes foram executados em uma máquina virtual com Sistema Operacional Ubuntu Server 16.04 LTS, com 1 vCPU, 1 GB de memória e 20 GB de HD;

- SHADE: Escrito em C++, SHADE foi configurado em uma VM Ubuntu Server 16.04 LTS, com 1 vCPU, 1 GB de memória e 20 GB de HD. O SHADE funcionando de acordo como descrito na Seção 3.4;

- Equipamentos: Nos experimentos, foi utilizado uma rede SDN (*Software Defined Network*), por meio de um *switch* programável de código aberto, com OpenWRT [43] e *Open vSwitch* [44] instalado [4, 12], como descrito na Seção 2.4.1. Foi utilizado também o controlador RYU [11] na versão 4.1.

No Servidor *web*, foi desenvolvido uma aplicação *web*, similar ao Sistema de Seleção Unificado (SISU - `htt://sisu.mec.gov.br`) utilizado para inscrições de alunos em cursos superiores, conforme a Figura 4.2. O SISU foi escolhido, por ser um sistema com alta demanda na época de inscrição aos cursos da universidades brasileiras e também muito visado por ataques

de negação de serviço durante esses períodos, segundo a Rede Nacional de Pesquisa - RNP [56]. A aplicação *web* contém as páginas: login, escolha de curso (primeira e segunda opção), modalidade e finalização da inscrição. Cada cliente realiza seu login, representado na Figura 4.2(a), depois é redirecionado até a página das opções de curso, conforme a Figura 4.2(b). Quando escolhe a página para selecionar a primeira opção de curso, o cliente é redirecionado para a página de escolha da Universidade, do Curso e da Modalidade, como mostrado na Figura 4.2(c). Em cada consulta desse processo é realizado utilizando buscas e inserção no banco de dados da aplicação. Após a escolha e a confirmação da primeira opção, o usuário retorna para a página de seleção das opções, mostrado na Figura 4.2(b). Continua então a inscrição da segunda opção, realizando os passos anteriores. Findada a escolha da segunda opção, é realizada a confirmação da inscrição no processo seletivo como um todo, assim o usuário finaliza a sua inscrição e pode sair do sistema.



(a) Página inicial do SISU.

(b) Página de opções de curso.

(c) Página de escolha de universidade, modalidade e de confirmação da inscrição.

Figura 4.2: As páginas que compõem a aplicação SISU.

Para gerar e simular o tráfego de um cliente legítimo, foi elaborado um *script* que criam robôs, que preenchem cada uma das páginas da aplicação SISU. Uma das características desses robôs é o preenchimento das páginas do SISU em velocidade aleatória, simulando diferentes clientes reais que preenchem páginas em velocidades distintas. Em cada página os robôs podem dar até três *reloads* na mesma página ou dez em todo processo de inscrição, se passar desses valores é caracterizado uma desistência. Além disso, os robôs esperam até vinte segundos para a resposta do servidor.

Os robôs criados oferecem alguns diferenciais dos quais podem-se listar três principais características: (i) Os robôs tentam se aproximar o máximo possível de usuários reais, interagindo com as páginas, de maneira que as requisições são realizadas em intervalos aleatório de tempo a fim de simular diferentes tipos de usuários, desde o mais lento ao mais rápido; (ii) Quando os

robôs estão realizando uma sequência de tarefas e se deparam com um erro causado pela indisponibilidade do servidor *web*, eles tentam realizar a mesma tarefa novamente por um número 'x' de vezes. Esse número é considerado como o máximo aceitável de tentativas por um usuário (como dito anteriormente, neste trabalho foi utilizado três *reloads*); (iii) Por fim, eles geram um relatório estatístico baseado nas suas ações de sucesso ou de falha (que estão relacionados aos parâmetros de qualidade de serviço da perspectiva do usuário). Neste relatório cada robô informa os tempos de resposta para cada ação, o número de tentativas para cada uma delas, o tempo total para realizar as tarefas, bem como o status de sucesso ou falha na realização das tarefas.

As métricas escolhidas, para verificar a consistência da defesa, foram:

- **Disponibilidade**: Essa métrica é a porcentagem da quantidade dos robôs que tiveram êxito em todo o processo de inscrição do SISU, ou seja, não desistiram pelo número de *reload* definido, ou pelo tempo que o servidor levou para responder;
- **Tempo de Serviço (TTS - *Time to Service*)**: Essa métrica é o tempo, em segundos, que demora para o cliente enviar uma solicitação, o servidor *web* receber e responder, e o cliente receber a resposta;
- **Quantidade de tráfego descartado**: É a quantidade de tráfego, de cliente e de atacante, que SHADE descartou durante a seleção;
- **Quantidade de Falsos Negativos**: É a quantidade de clientes legítimos que não foram identificados por meio do *fingerprinting* e não tiveram seu tráfego priorizado;
- **Quantidade de Falsos Positivos**: A quantidade de atacantes que foram identificados erroneamente por meio do *fingerprinting* e inseridos na *whitelist*.

4.2 Experimentos

Foram realizados experimentos em rede real em seis cenários diferentes, para verificar a proposta:

- **Sem ataque e sem defesa**: A intenção desse cenário é verificar se a rede está se comportando normalmente, sem anomalias, para poder inserir os elementos dos experimentos, sem variáveis externas que possam influenciar nos resultados;

- **Sem defesa e com ataque**: Nesse cenário, não há nenhum mecanismo de defesa, ou seja, as requisições dos atacantes e dos clientes vão diretamente ao servidor *web*. A intenção de realizar experimento nesse cenário é verificar a consistência do ataque, constatando a negação de serviço por parte do servidor *web*;

- **Apenas SeVen e com ataque**: Neste terceiro cenário foi adicionado uma camada de defesa, o *SeVen*. Comprovada a efetividade contra ataques *low-rate*, a proposta desse cenário é verificar se a defesa *SeVen* é capaz de mitigar ataques do tipo *high-rate*;

- **SHADE sem whitelist e com ataque**: Neste quarto cenário foram realizados experimentos na rede, sem a identificação do cliente descrito na Seção 3.2 e sem dar prioridade no tráfego desses clientes. Verificando assim, se o simples descarte das requisições é o bastante para mitigar ataques de inundação;

- **SHADE e com ataque**: Finalmente o quinto cenário inclui toda a proposta de defesa do SHADE com *whitelist* como exemplificado na Seção 3.5. Foram realizados experimentos distintos, diferenciando a forma de identificação do cliente: por meio da autenticação do usuário na aplicação *web* e do *User-Agent*, descrito na Seção 3.2.1; e a outra forma verificando a porcentagem dos objetos da página que foram requisitados, de acordo com a Seção 3.2.2;

- **SHADE e sem ataque**: Por fim, no ultimo cenário, a intensão é verificar se a inserção do mecanismo de defesa influencia na disponibilidade e no tempo de resposta quando não há um ataque.

Cada experimento teve duração total de 25 minutos. Em cada experimento foi coletado a porcentagem dos robôs que tiveram sucesso, o tempo de serviço que os robôs levaram na comunicação com o servidor, a quantidade do tráfego que foram descartado por SHADE, as quantidades de clientes legítimos que não foram identificados, e a quantidade de atacantes que foram identificados como clientes e inseridos na *whitelist*.

4.3 Resultados

Antes de serem realizados os experimentos envolvendo as defesas, foram realizados testes em que estavam executando as máquinas que continham os clientes, o servidor e todos os equipamentos de rede. Tanto as máquinas que estavam os atacantes quanto que estavam o mecanismo de defesa não estavam em operação durante esses testes. Ou seja, os clientes realizavam suas requisições a aplicação *web* sem interferência ou inserção de outro elemento.

O objetivo é verificar se o cenário construído obtinha resultados esperados, em condições de normalidade, principalmente quanto à quantidade de clientes gerados. Pois, uma negação de serviço pode ser gerado por uma grande quantidade de clientes requisitando a um servidor com pouca capacidade de atendimento. Além disso, a intenção é verificar se a inserção de elementos de uma rede definida por *software*, como o controlador e o *switch* programável de código aberto, influenciava em algum aspecto a disponibilidade e o tempo de serviço. Foram realizadas dez repetições deste experimento.

Na Tabela 4.1 tem-se a porcentagem dos clientes que foram atendidos pela aplicação *web* e o tempo de serviço, em segundos. Verificou-se que não há influência na disponibilidade, pois, dos 1000 robôs gerado como cliente, todos foram atendidos, constatou-se também que nenhum robô precisou realizar *reload* nas páginas, tendo assim uma disponibilidade de 100%. Da mesma forma, o tempo de serviço verificado foi o esperado para uma aplicação que está sendo solicitada dentro de uma mesma rede. O valor de 0,01 segundo é um tempo aceitável, inclusive quando se inclui os elementos de uma rede SDN.

Tabela 4.1: Resultado da disponibilidade e o tempo de serviço, no cenário sem ataque e sem a defesa.

	Disponibilidade (%)	Tempo de serviço (s)
Sem Defesa e sem ataque	100	0,01

Dessa forma, foi verificado que não ha interferência do cenário construído na comunicação entre os clientes e o servidor *web*, pois, obteve resultados esperados. A partir dessa configuração foi inserido os elementos de ataque e de defesa, com a certeza de que a rede está funcionando normalmente, sem influenciar os resultados dos outros experimentos.

4.3.1 Verificando a efetividade da defesa *SeVen* em ataques do tipo *High-Rate*

A defesa *SeVen* foi proposta para ataques de negação de serviço na camada de aplicação, do tipo *low-rate*, ou seja, que tem pouca taxa de envios de requisições, por exemplo o ataque *Slowloris* [28]. De acordo com Dantas [1], sem a defesa *SeVen*, a disponibilidade era de 0% e o tempo de serviço infinito, enquanto que com a estratégia, foi verificado uma disponibilidade de 95% dos clientes em um servidor *web* sofrendo ataque *Slowloris*, e um tempo de serviço de 0,06 segundo. Esses valores estão apresentados na linha correspondente da Tabela 4.2.

A partir desses resultados, foram realizados experimentos para verificar a eficiência da defesa *SeVen* em ataques do tipo *high-rate*, por exemplo, o *Get-Flooding*. Ou seja, no cenário construído para o presente trabalho, foi inserido apenas a defesa *SeVen*, foi realizado o ataque com quatro atacantes e foram coletados os dados acerca da disponibilidade e de tempo de serviço.

A Tabela 4.2, na última linha, mostra o resultado, em média, da disponibilidade e do tempo de serviço, obtidos nesses experimentos. A defesa *SeVen* só conseguiu prover disponibilidade para 13,8% dos clientes legítimos, durante um ataque de negação de serviço, do tipo *high-rate*. Esse resultado se mostrou até pior que os constatados sem nenhuma defesa, verificado na Tabela 4.3, que foi de 17,7%. Isso pode ser explicado pela introdução de um processo a mais na conversação entre o cliente e servidor, assim piora a disponibilidade. Além disso, *SeVen* tem memória limitada e poder de processamento também limitado, e não consegue gerenciar a maioria das requisições destinadas ao servidor *web*, pois, como foi dito anteriormente, os ataques *high-rate* tem como característica enviar várias requisições.

Dentre os 13,8% dos clientes atendidos, o tempo de serviço, em média, foi de 7,1 segundos, de acordo com a Tabela 4.2. Ou seja, o processo do cliente realizar uma requisição, o servidor *web* receber, responder e a resposta chegar ao cliente, demorou, em média, sete segundos. Um tempo relativamente alto para um cliente esperar por uma solicitação.

Tabela 4.2: Resultados dos experimentos utilizando *SeVen* com ataques do tipo *Low-Rate* e *High-Rate*.

	Disponibilidade (%)	Tempo de serviço (s)
SeVen em ataques Low-Rate	95	0,06
SeVen em ataques High-Rate	13,8	7,1

Dessa forma, não é recomendado utilizar a estratégia *SeVen* em ataques *high-rate*, pois, não consegue prover disponibilidade razoável para os clientes legítimos. Por outro lado, quando se trata de ataques *low-rate*, *SeVen* é extremamente eficiente, conseguindo que quase a totalidade dos clientes sejam atendidos.

4.3.2 Encontrando a taxa de descarte do SHADE, identificando cliente por meio do protocolo HTTP

Para verificar a quantidade de tráfego descartado por SHADE, foi realizado diversos experimentos, variando a quantidade de atacantes, desde uma máquina virtual até quatro VMs (*Virtual Machines*), e variando também a quantidade de tráfego descartado: descartar um a cada cinco requisições, dois a cada cinco requisições, um a cada duas requisições, três a cada cinco requisições e quatro a cada cinco requisições. Foram verificadas a disponibilidade dos clientes em cada valor, e o que obter melhor resultado será utilizado nos experimentos. A Tabela 4.3 demonstra as médias dos resultados obtidos, sendo as linhas a variação do tráfego, e as colunas a variação na taxa de descarte. Observa-se que o tráfego com um, dois ou três atacantes não é o suficiente para, tecnicamente, indisponibilizar um servidor *web*. É com quatro atacantes que o ataque se torna efetivo no cenário sem a defesa.

Salienta-se ainda que foram realizados experimentos aumentando a quantidade de máquinas virtuais realizando ataque, mas esse aumento não modificaram os resultados da disponibilidade em todos os cenários. Ou seja, o simples aumento na quantidade de atacante não influencia nos experimentos, sendo quatro atacantes a quantidade ideal, sem prejuízo na qualidade do ataque e sem os próprios atacantes interferir entre si.

Tabela 4.3: Resultados da disponibilidade, dada em porcentagem, de 1000 robôs, com variação do tráfego atacante e do descarte.

	Sem Defesa	SHADE sem Whitelist [Corrêa et al. 2016]	Whitelist e Descarte 1/5	Whitelist e Descarte 2/5	Whitelist e Descarte 1/2	Whitelist e Descarte 3/5	Whitelist e Descarte 4/5
Um Atacante	61,5	44,9	97,3	97,8	99,4	99,9	99,9
Dois Atacantes	54,3	20,6	87,7	93,7	98,5	98,8	99,5
Três Atacantes	42	12,9	47,7	70,5	96,3	97,1	99,9
Quatro Atacantes	17,7	4,2	17,4	44,2	70,9	84	99,8

Dessa forma, para a padronização dos experimentos, foram utilizados sempre quatro atacantes em todos os cenários. Além disso, a taxa de descarte do SHADE foi escolhida de acordo com os melhores resultados. Considerando apenas os cenários com quatro atacantes, quando se realiza o descarte de uma requisição a cada cinco que chega, de acordo com a Tabela 4.3, observou-se uma disponibilidade de apenas 17,4%, número esse um pouco pior do que registrado sem defesa. Isso se deve a quantidade do tráfego que ainda chega ao servidor *web* e por

todo o processamento inserido com as defesas. No descarte de dois a cada cinco requisições, verifica-se uma disponibilidade de 44,2%, uma melhora, mas ainda sem efeitos práticos aos clientes, que continua, em muitos casos, a não conseguir acessar o sistema SISU.

Com o descarte da metade do tráfego, ou seja, uma a cada duas requisições, a disponibilidade subiu para 70,9%, obtendo um valor aceitável em uma situação de ataque. Nos experimentos com taxa de descarte de três requisições a cada cinco, a disponibilidade chegou ao valor de 84%. E por fim, com o descarte quatro a cada cinco requisições, a disponibilidade alcançou o valor de 99,8%. Foram realizados experimentos com valores ainda mais agressivos, o que se mostraram sem efeito nos valores da disponibilidade, e portanto retirados dos resultados. Assim, com o melhor resultado obtido com o descarte de 80% do tráfego, todos as sequências de experimentos foram realizados com as seguintes configurações: quatro atacantes e SHADE descartando quatro a cada cinco requisições. Salienta-se que uma das propostas de trabalho futuro é utilizar essa taxa de descarte de forma dinâmica, ou seja, que varie de acordo com a taxa de utilização do servidor *web* e/ou da quantidade de tráfego que está chegando no servidor.

Analisando a Tabela 4.3, nos valores obtidos nos três cenários, sem defesa, com SHADE sem *whitelist* e com toda a defesa SHADE com *whitelist* descartando quatro a cada cinco requisições, tendo todos os cenários com quatro atacantes, observa-se, no cenário sem a defesa, uma disponibilidade variando entre 17,1% e 29%, tendo como média 17,7%, ou seja, dos 1000 robôs que tentaram realizar a inscrição no site do SISU, apenas 177 conseguiram completar a inscrição durante um ataque de negação de serviço. Os 823 restantes desistiram, exclusivamente, por atingir o tempo máximo de resposta, ou seja, desistiram pois a resposta do servidor *web* ultrapassou os vinte segundos.

Nos experimentos do SHADE sem *whitelist* conforme proposto em Corrêa *et al.* [10], a disponibilidade variou entre 2,8% e 6,5%, tendo como média 4,2%, mostrando que a simples seleção e diminuição do tráfego não é suficiente para mitigar o ataque, pelo contrário, a disponibilidade piorou em relação a disponibilidade do cenário sem defesa. Os resultados foram diferentes dos encontrados na proposta de Corrêa *et al.* [10], pois os experimento foram realizados utilizando o emulador de redes SDN, o Mininet [40]. Dessa forma, como no presente trabalho os experimentos foram realizados na rede, os resultados representam uma maior fidelidade, pois não há a variável da emulação do ambiente.

Em contrapartida, nos experimentos com toda a proposta SHADE, utilizando *whitelist* em clientes legítimos e descartando 80% do tráfego que não está nessa lista, a disponibilidade variou entre 99,4% e 100%, tendo a média de 99,8%, ou seja, dos 1000 robôs que tentaram realizar a inscrição, 998 conseguiram completa-la, e apenas 2 não conseguiram pelo número

máximo de *reload* nas páginas. O desvio padrão para a disponibilidade foi baixo, ficando em 0,24 no universo de 1000 robôs, inserindo uma maior confiabilidade da média obtida.

4.3.3 Encontrando a taxa de descarte do SHADE, identificando cliente por meio do padrão de utilização

Para a técnica de *fingerprinting* do cliente utilizando a quantidade de objetos requisitados da página, foram repetidos os experimentos variando a quantidade de requisições rejeitadas. A intenção de verificar os resultados novamente é justamente por mudar a forma de inserir o cliente na *whitelist*, assim, com os novos experimentos, verificar se há alguma alteração nos resultados.

Dessa forma, a quantidade de atacantes permanece em 4 máquinas virtuais, pois conseguiram indisponibilizar o servidor *web*, de acordo com a Tabela 4.3. Foi variado então a taxa de descarte das requisições que não estão na *whitelist*. As taxas de descarte testada foram: 20%, 40%, 50%, 60% e 80% do tráfego que chega em SHADE. Foram realizados dez repetições dos experimentos. A Tabela 4.4 apresenta os valores, em média, obtidos nesses experimentos.

Tabela 4.4: Resultados da disponibilidade, dada em porcentagem, de 1000 robôs, com variação de descarte, utilizando a identificação por meio dos objetos da página.

	Descarte 20%	Descarte 40%	Descarte 50%	Descarte 60%	Descarte 80%
Quatro Atacantes	13,9	41,7	70,3	80	99,5

Observa-se, na Tabela 4.4, que quando há a seleção de 20% do tráfego, o sistema de defesa só consegue prover disponibilidade para 13,9% dos clientes, mesmo os clientes legítimos sendo identificados, pois, 80% do tráfego malicioso ainda está chegando ao servidor *web* e influenciando nessa disponibilidade. Quando o descarte sobe para 40% do tráfego malicioso, foi verificado que 41,7% dos robôs clientes conseguiram acessar e se inscrever no SISU.

Para o descarte de 50% do tráfego, a disponibilidade foi de 70,3%, sendo uma disponibilidade aceitável para um servidor *web* que está sob ataque de negação de serviço. Se a quantidade de descarte aumenta para 60% do tráfego, observa-se também um aumento na disponibilidade, oferecendo para 80% dos clientes legítimos.

Por fim, a disponibilidade no cenário de quatro atacantes e o descarte de 4/5 do tráfego foi de 99,5%. Ou seja, dos 1000 robôs gerados, apenas 5 não conseguiu realizar a inscrição no sis-

tema do SISU, enquanto que 995 conseguiram completar a inscrição e em tempo hábil. Assim como nos experimentos envolvendo a identificação por meio do cabeçalho HTTP, foram também realizados experimentos com maior taxa de descarte do que 80%, mas não foi encontrado uma melhora na disponibilidade dos clientes. Portanto, foram excluídos dos resultados. Dessa forma, para o resto dos experimentos envolvendo essa forma de identificação, foi considerado o descarte de 80% do tráfego não identificado.

Comparando as Tabela 4.3 e 4.4, nos experimentos com quatro atacantes, percebe-se uma pequena queda na disponibilidade em cada taxa de descarte. Quando o SHADE utiliza a técnica de *fingerprinting* por meio do cabeçalho HTTP, a disponibilidade é maior do que quando SHADE insere na *whitelist* por meio da quantidade de objetos da página que foram solicitados. Essa diferença pode ser explicada pelo tempo de processamento para identificar um cliente legítimo e inseri-lo na *whitelist*, pois na segunda forma de identificação, o sistema espera que o cliente solicite 80% dos objetos da página, enquanto que na primeira forma de identificação, na primeira requisição o cliente pode ser inserido na *whitelist*.

4.3.4 SHADE em operação: Identificando clientes por meio do cabeçalho HTTP

De acordo com o cenário proposto, sendo quatro máquinas realizando o ataque, o SHADE realizando a seleção e o descarte de quatro a cada cinco requisições, e a inserção de clientes na *whitelist* por meio das informações oriundas do protocolo HTTP, foram realizados experimentos para verificar a consistência da defesa. Concluídos os experimentos e a captura dos dados, uma análise da eficiência da defesa e a discussão acerca dos resultados são realizadas a seguir.

A Figura 4.3 mostra o gráfico da quantidade de requisições que chegam no servidor *web* por segundo, no eixo y, durante os 300 primeiros segundos dos experimentos, decorrido no eixo x, dos testes realizados com o SHADE sem a utilização da *whitelist*. As linhas em vermelho representam as requisições dos clientes, enquanto que as linhas em verde representam as requisições geradas pelos atacantes. Observa-se que o tráfego em vermelho contém alguns picos, que são os minutos em que são gerados novos robôs. Entre esses picos, percebe-se que quase não há requisições de clientes. Isso ocorre, porque os clientes não conseguem continuar a inscrição no site, pois o servidor *web* está sobrecarregado. A outra linha do gráfico, o tráfego em verde, apresenta diversas variações no decorrer do tempo, tendo um pico nos primeiros sessenta segundos dos experimentos. É justamente o atacante, que inicia com grande rajada, e depois apenas administra a indisponibilidade do servidor. Dessa forma, o tráfego atacante sobrepõe e influencia o tráfego de clientes legítimos, gerando assim a negação de serviço. Nos experi-

mentos, foi verificado que a quantidade do tráfego atacante é trinta vezes maior que tráfego de clientes.

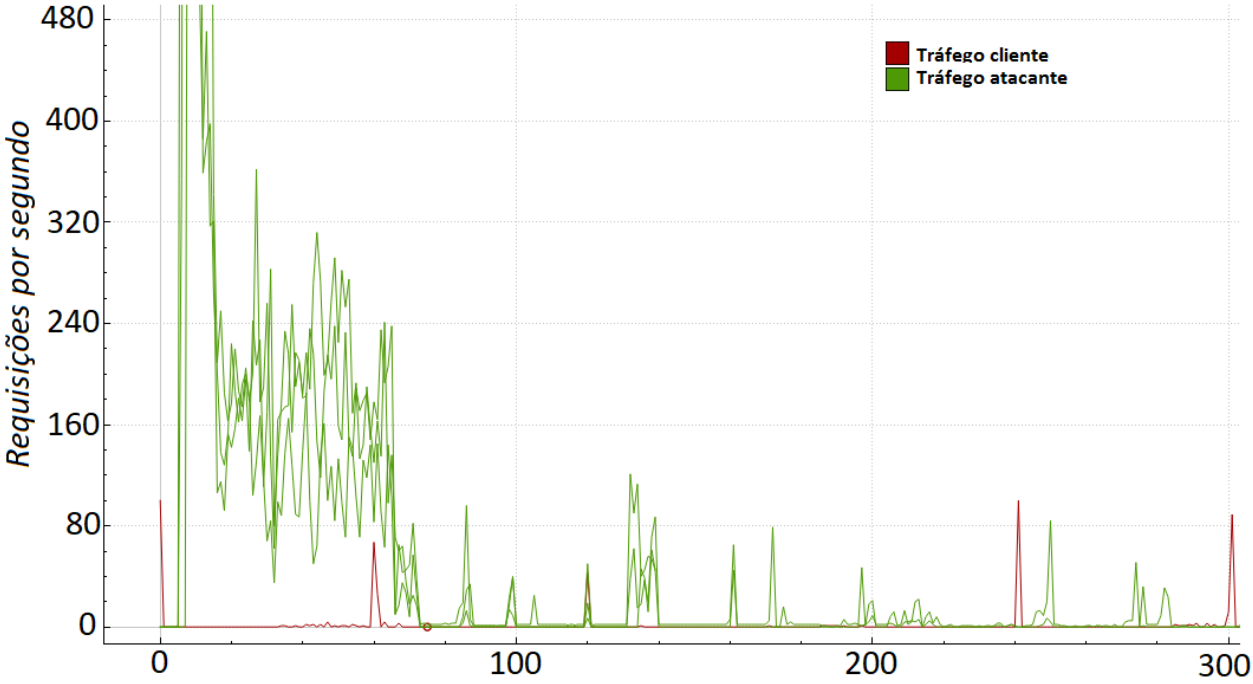


Figura 4.3: Gráfico da quantidade do tráfego cliente e atacante que chegam no servidor *web*, nos testes sem whitelist, durante os primeiros 300 segundos.

Na Figura 4.4, assim como na anterior, tem-se o gráfico da quantidade de requisições que chegam no servidor *web* por segundo, nos primeiros 300 segundos, mas nos experimentos realizados com SHADE e *whitelist*, sendo mais uma vez o tráfego cliente em vermelho, e do atacante em verde. Percebe-se que a cada sessenta segundos ocorre um pico no tráfego do cliente, que é justamente a geração de dez robôs a cada minuto. É interessante perceber que nos primeiros segundos, após o primeiro pico, quase não há requisições dos clientes, enquanto que há requisições dos atacantes. Este período inicial, em que cliente não é atendido, é justamente o período em que os clientes estão sendo identificados e adicionados na *whitelist*, para poder ter prioridade no tráfego. Assim, passado o primeiro momento, percebe-se que o tráfego do cliente se torna permanente. Portanto, apenas o tráfego atacante é submetido ao descarte realizado por SHADE.

Na Figura 4.5 tem-se o tempo de serviço, em média, dos quatro cenários. O tempo é medido em segundos, que está demonstrado no eixo das ordenadas, e no eixo das abscissas estão as barras de cada cenário dos experimentos: a barra da esquerda é o cenário sem a defesa; a barra central é o cenário apenas *SeVen*; e por fim, as duas barras da direita são os cenários com SHADE, sendo a primeira sem *whitelist* e a última do lado direito utilizando a estratégia de inserir clientes na *whitelist*.

Figura 4.4: Gráfico da quantidade do tráfego cliente e atacante que chegam no servidor *web*, nos testes com whitelist, durante os primeiros 300 segundos.

O TTS no cenário sem a defesa variou entre 13,7 segundos e 16,2 segundos, tendo como média 14,8 segundos e o desvio padrão de 1,29. Esse valor é obtido dos robôs que conseguiram completar a inscrição. Portanto, apesar de 17,7% dos robôs terem conseguido completar a inscrição, o tempo de respostas que obtiveram foi quase próximo ao limiar de desistência, que é de vinte segundos. Esse valor reflete a situação do servidor *web* diante de um ataque de inundação, pois existe uma grande fila de atendimento e as requisições precisam esperar para serem atendidas.

No cenário com a defesa *SeVen*, observa-se uma queda de 50% no tempo de resposta em relação do cenário sem a defesa. O TTS variou entre 5,4 e 8,1 segundos, tendo como média 7,1 segundos, com o desvio padrão de 0,9. Isso demonstra que a estratégia *SeVen* consegue diminuir o tempo de serviço, mas não consegue ser efetivo, pois, apesar de não ter um consenso de quanto um usuário espera para a página carregar, entende-se que sete segundos é um tempo considerável de espera. O TTS no cenário da estratégia SHADE sem *whitelist* mostrou-se pior em relação a todos os cenários. Variou entre 16,9 segundos e 18,4 segundos, tendo a média de 17,3 segundos, quase no limiar dos vinte segundos, tendo o desvio padrão de 4,2. Comprova, mais uma vez, que o simples descarte não é efetivo para ataques de negação de serviço.

Com o cenário do SHADE utilizando *whitelist*, tem-se um tempo de resposta variando entre 0,19 e 0,27 segundo, sendo a média 0,2 segundo e o desvio padrão de 0,1. Portanto, o TTS demonstrado com SHADE durante um ataque de negação de serviço se assemelha o tempo de serviço obtido em redes que não há ataque, mostrando assim a efetividade da proposta, principalmente dando prioridade para clientes legítimos e utilizando uma seleção, diminuindo a quantidade de requisições que chegam ao servidor *web*.

Na Tabela 4.5 observa-se a quantidade de tráfego que foi descartado e aceito pela seleção feita por SHADE e pela estratégia do *SeVen*. O tráfego gerado pelo ataque, em média, é de 56

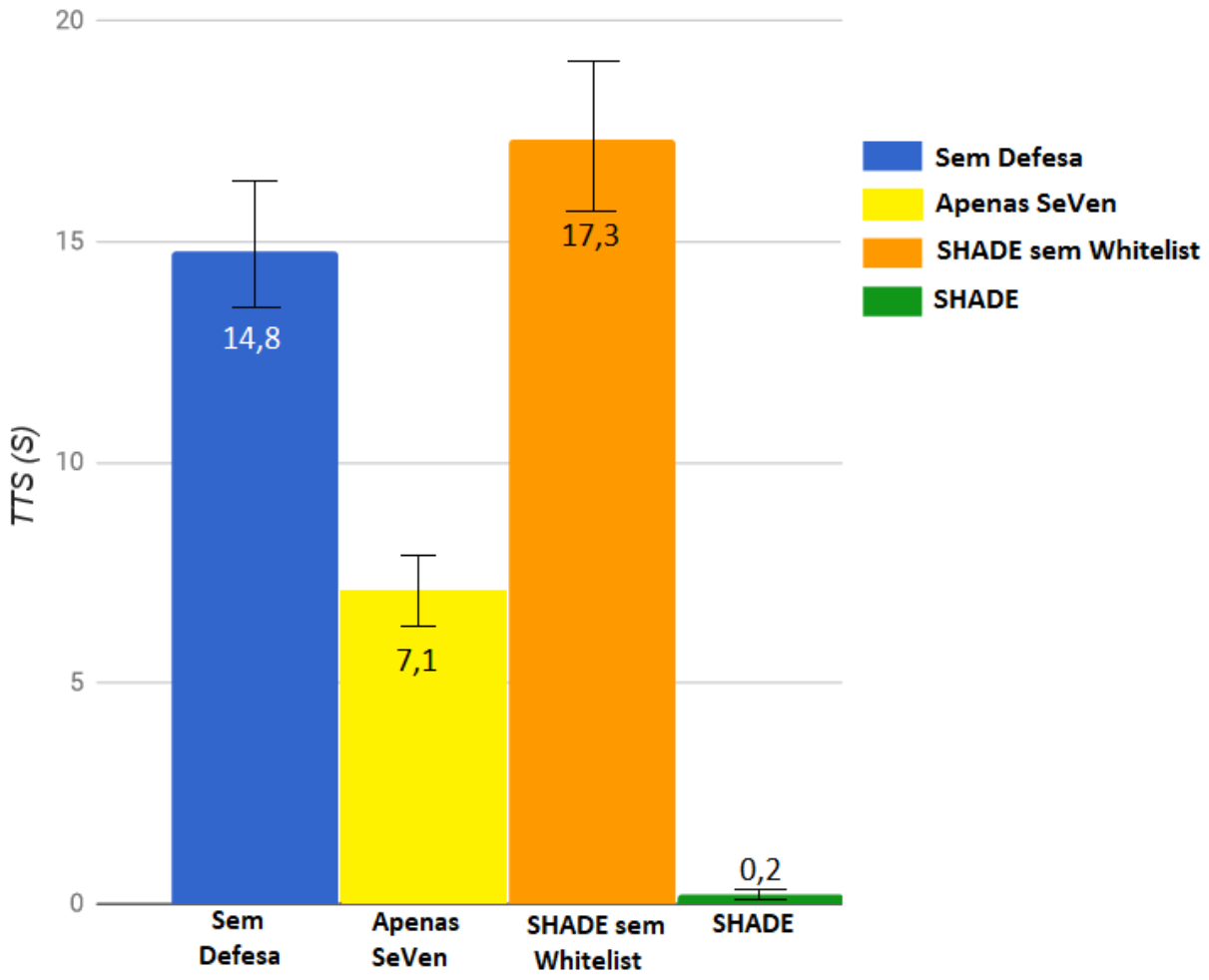


Figura 4.5: Tempo de serviço nos cenários sem defesa, apenas SeVen, SHADE sem whitelist e SHADE com whitelist.

mil requisições, sendo descartado 45 mil pela seleção do SHADE. Dos 20% do tráfego atacante que chegaram até o *SeVen*, 19,1% foi repassado ao servidor *web* e 0,9% foi descartado pela estratégia do *SeVen*. Com isso, por mais que um cliente não seja identificado pelo *fingerprinting*, ainda há chance de ser atendido. Enquanto que, nas requisições dos clientes, nenhum foi descartado pela seleção do SHADE, justamente por estar na *whitelist* e não ser direcionado ao processo de descarte. Quando as requisições dos clientes chegam ao *SeVen*, devem concorrer ainda com as requisições de atacantes que foram repassadas por SHADE. A quantidade das requisições dos clientes é apenas 10% da quantidade das requisições dos atacantes, o que acarreta a perda de 0,2% das requisições clientes.

É importante observar que durante os experimentos não houve registro de falso positivo, ou seja, que um atacante seja identificado erroneamente como um cliente legítimo. Quanto aos falsos negativos, quando um cliente não foi classificado como cliente legítimo e foi redirecionado ao SHADE, também não houve registro. Ou seja, todos os clientes que solicitaram informação ao servidor *web* foram identificados e inseridos na *whitelist*. Essa informação é verificada na

Tabela 4.5: Quantidade de tráfego descartado por SHADE e aceito por SeVen.

	Tráfego	
	Descartado por SHADE (em média)	Aceito por SeVen (em média)
Atacante	80%	19,1%
Cliente	0%	99,8%
Número de requisições – Atacante	45.000 de 56.000	10.000 de 11.000
Número de requisições – Cliente	0 de 1.700	1.600 de 1.700

linha "Cliente" e na coluna "Descartado por SHADE"na Tabela 4.5, em que 0% do tráfego foi descartado. Os 0,2% dos clientes que não conseguiram completar a inscrição foram descartados durante a estratégia de defesa realizada por *SeVen*, durante as funções de probabilidade.

Outra informação que pode-se obter da Tabela 4.5, é a quantidade de tráfego inútil que o sistema SHADE consegue descartar. Ou seja, de todas as requisições de atacante, ocupando largura de banda e processamento da rede, SHADE mantém apenas 11 mil requisições, diminuindo drasticamente o tráfego.

4.3.5 SHADE em operação: Identificando cliente por meio da quantidade de objetos requisitados

Foram realizados experimentos com o mesmo cenário, mas alterando a forma de identificação dos clientes. Nesses experimentos os clientes foram identificados e inseridos na *whitelist* quando solicitarem 80% dos objetos que compõem a página. Essa forma de reconhecer um usuário legítimo é ideal para páginas *web* que não tenha nenhum sistema de autenticação, como o método anterior preconizava.

Dessa forma, para os experimentos, foi utilizada a página principal do sistema SISU desenvolvido. Na Figura 4.6 tem-se a página do SISU, enquanto que os objetos contidos nessa página estão representados na Figura 4.7, totalizando cinco objetos. Dessa forma, quando um cliente realiza o pedido de quatro objetos da página ele é inserido na *whitelist* e seu tráfego terá prioridade.

Assim, como os robôs clientes terão que requisitar os objetos da página para preencher a inscrição, o sistema de defesa identifica que realizou quatro ou mais objetos e inseri-o na

Figura 4.6: Página *web* do SISU.

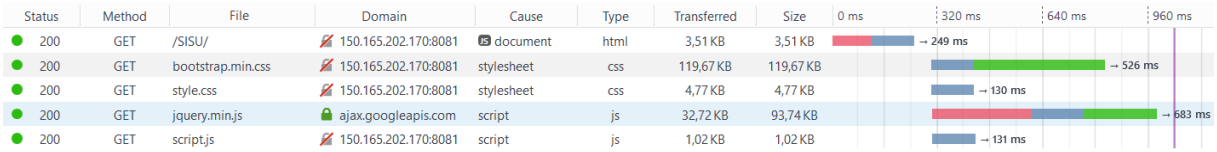


Figura 4.7: Objetos presentes na página do SISU.

whitelist. Dessa forma, na medida que os robôs continuam realizando o preenchimento do sistema SISU, pode-se colher informações acerca da qualidade de serviço e de experiência que os robôs clientes oferecem: quantidade de clientes que conseguiram terminar a inscrição, o tempo de serviço, a quantidade de clientes que desistiram por esperar mais que vinte segundos e a quantidade de clientes que desistiram por ter realizados vários *reloads*.

Como descrito na Seção 4.3.3, a seleção de descarte realizado por SHADE também foi de 80% do tráfego que não estava inserido na *whitelist*. Portanto, 20% do tráfego não identificado seria encaminhado para o servidor *web*. Assim, essa forma de identificar um cliente legítimo, junto com todo o sistema de defesa, conseguiu prover uma disponibilidade que variou entre 98,3% e 100%, tendo como média 99,5% dos clientes atendidos, de acordo com a Tabela 4.4. O desvio padrão dos resultados da disponibilidade foi de 0,52.

A Figura 4.8 representa o gráfico de barras, comparando o tempo de serviço nos cinco cenários: sem defesa, apenas SeVen, SHADE sem utilizar *whitelist*, SHADE com a identificação de clientes utilizando o cabeçalho do protocolo HTTP, e por fim, SHADE identificando os clientes por meio da quantidade de requisições dos objetos da página.

O interessante de comparar na Figura 4.8 são os resultados envolvendo a estratégia SHADE identificando clientes legítimos usando o cabeçalho HTTP e SHADE identificando os usuários pela quantidade de objetos requisitados. No segundo caso, o tempo de serviço variou entre 0,6 e 1,3 segundo, tendo como média 0,9 e um desvio padrão de 0,18, enquanto que no primeiro

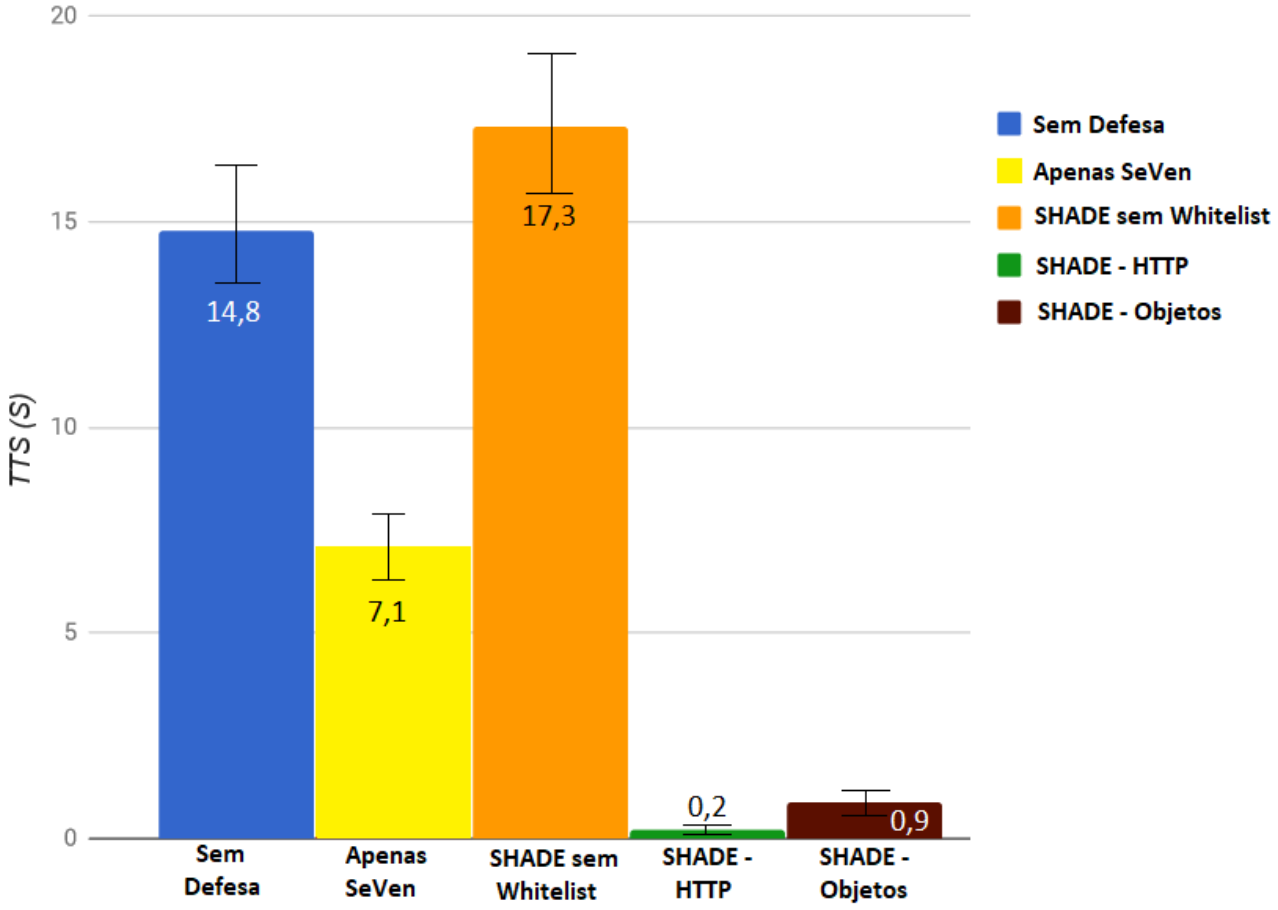


Figura 4.8: Tempo de serviço nos cenários sem defesa, apenas SeVen, SHADE sem whitelist, SHADE identificando por meio do HTTP e SHADE identificando por meio dos objetos da página.

caso, a média foi de 0,2 segundo no tempo de serviço.

Ou seja, a estratégia de selecionar os clientes legítimos, usando a quantidade de objetos da página para identifica-lo, tem um tempo de serviço maior. Isso se deve ao fato que o sistema de defesa só considera como cliente e insere na *whitelist* após a quarta requisição de objeto (no caso do cenário construído para este trabalho), enquanto que, o *fingerprinting* do cliente por meio do cabeçalho HTTP, é realizado já na primeira requisição que o usuário realiza. Assim, diminui o tempo de processamento inserido pela estratégia.

4.3.6 SHADE em cenário sem ataques

Por fim, foram realizados experimentos utilizando a proposta de defesa, mas no cenário sem ataque, ou seja, clientes legítimos realizavam requisições ao servidor *web* sem que houvesse usuários maliciosos realizando ataque. A intenção desse cenário é verificar se o sistema de defesa proposto influenciava na relação entre cliente e aplicação *web*.

A Tabela 4.6 apresenta o resultado, em média, dos experimentos nesse cenário. A disponi-

bilidade foi de 100%, ou seja, dos clientes legítimos que realizaram a requisição todos foram atendidos. O desvio padrão foi de 0, ou seja, em todas as repetições a disponibilidade foi de 100%. Na última coluna tem-se a média do tempo de serviço neste cenário, tendo como valor de 0,1 segundo. Mais uma vez, em todas as repetições, o tempo de serviço se manteve em 0,1 segundo.

Tabela 4.6: Resultados dos experimentos utilizando SHADE, sem ataque.

	Disponibilidade (%)	Tempo de serviço (s)
SHADE	100	0,1

Comparando os resultados desse cenário com o primeiro (sem ataque e sem defesa), apresentado na Tabela 4.1, verifica-se que a disponibilidade se manteve, não influenciando. Mas o tempo de serviço foi diferente. No cenário sem ataque e sem defesa, o tempo verificado foi de 0,01 segundo, enquanto que com a inserção do sistema de defesa, o tempo de serviço subiu para 0,1 segundo. Essa diferença não influencia na experiência do usuário final, mas que há sim um processamento a mais, aumentando assim o tempo de serviço.

Vale salientar ainda que, o redirecionamento de fluxo, a seleção e o descarte das requisições só acontecem quando *SeVen* verificar que o servidor *web* está sobrecarregado. Neste cenário específico, por causa da quantidade de clientes gerados e pela ausência de ataque, *SeVen* não ativou o restante da defesa. Assim, só a identificação do usuário, por meio do *fingerprinting*, e a inserção na *whitelist* estavam ativados. Caso haja necessidade (inicio de um ataque), esses usuários já estão na *whitelist* e terão prioridade no seu tráfego.

Outro ponto necessário observar é que todo o sistema de defesa não avisa ao usuário (seja ele cliente legítimo ou atacante) que ele foi adicionado na *whitelist* ou se foi descartado pela seleção de SHADE. Assim, não tem como saber se foi ou não inserido na *whitelist* e se vai ter o tráfego priorizado. Da mesma forma quando uma requisição é descartada na seleção, SHADE avisa ao usuário que o servidor está indisponível, ou seja, envia um código 503 do HTTP, que significa indisponibilidade [51]. Assim, o sistema de defesa tende a ser o mais transparente possível aos usuários, impedindo inclusive que usuários maliciosos tentem formas de burlar o sistema de defesa.

Portanto, a inserção de todo o sistema de defesa, do *fingerprinting* de cliente legítimo e adiciona-lo na *whitelist*, não influencia na rede em que o servidor *web* e os clientes estão inseridos, mas acrescenta um benefício, que é a proteção desse servidor *web* contra ataques de

negação de serviço na camada de aplicação, do tipo *high-rate*.

4.4 Breve comparação das técnicas de assinaturas

As duas técnicas de identificação de um cliente legítimo dentro de um fluxo se mostraram efetivas, havendo diferenças entre elas nos resultados. O *fingerprinting* do cliente, por meio da autenticação no sistema *web* obteve uma disponibilidade ligeiramente maior e um tempo de serviço menor comparado a técnica de identificação por meio da quantidade de objetos da página, 99,8% e 0,2 segundo contra 99,5% e 0,9 segundo, respectivamente.

Isso se deve a forma e o processamento envolvido nas técnicas propostas, pois, a identificação do cliente pela autenticação na aplicação *web* é mais rápida, sendo necessário apenas a requisição que realiza o *login* na aplicação. Enquanto que na segunda técnica, é necessário esperar que o usuário realize as solicitações de 80% dos objetos da página.

Capítulo 5

CONCLUSÃO E TRABALHOS FUTUROS

De acordo com o que foi exposto, a partir da fundamentação teórica sobre segurança da informação, mais especificamente sobre ataques de negação de serviço na camada de aplicação (ADDoS), passando pela discussão acerca da utilização de *fingerprinting*, analisando as Redes Definidas por *Software*, incluindo sobre a utilização de *switches* programáveis de código aberto nas redes SDN, este trabalho propôs uma nova estratégia para mitigar ADDoS do tipo *high-rate*.

A proposta visa criar uma *whitelist* dinâmica, alimentada por *fingerprinting* de clientes legítimos, dando preferência no tráfego dos usuários inseridos nessa lista, durante ataques de negação de serviço na camada de aplicação, mais especificamente ataques *Get-Flooding*. Quando um cliente é adicionado na *whitelist*, suas requisições tem prioridade em relação as outras requisições. A defesa consiste em utilizar a tecnologia *SeVen* [8], proposto para mitigar ADDoS do tipo *low-rate*, adicionando a função de sensor do nível de utilização do servidor *web*. Quando *SeVen* verifica que o servidor está Sobrecarregado, é realizado um redirecionamento do fluxo das requisições que não estão inseridos na *whitelist*. Esse fluxo é direcionado para o SHADE (*Selective High-Rate DDoS Defense*), que realiza uma seleção das requisições que serão processadas ou que serão descartadas.

O trabalho propõe duas formas de identificar clientes legítimos:

- por meio dos campos do cabeçalho HTTP, verificando se a requisição está utilizando o método *POST* e se há um valor no campo *user agent* que seja válido. Essa forma aproveita que uma aplicação *web* requisite um *login* e senha a um usuário;
- Enquanto que a segunda proposta se baseia no padrão de utilização de um cliente legítimo para identifica-lo, podendo ser utilizado em servidores *web* que não tem sistema de autenticação. A proposta é que se um usuário requisitar mais que 80% dos objetos que compõe a página, ele é considerado como um cliente legítimo.

Para verificar a consistência da proposta, foi elaborado um cenário e realizado experimentos reais em uma rede SDN, utilizando um *switch* programável de código aberto [4, 12]. Apesar dos experimentos serem realizados em uma rede SDN, a proposta pode ser utilizada nas redes tradicionais, inserindo a mudança de fluxo em uma regra de NAT no roteador da rede. Foram realizados experimentos com seis configurações diferentes: sem defesa e sem ataque, sem defesa e com ataque, apenas com *SeVen* e com ataque, utilizando SHADE sem *whitelist* e com ataque, a quarta utilizando SHADE e a proposta da *whitelist* e ataque, e por fim, utilizando SHADE sem ataque. Em todas configurações mediu-se a disponibilidade aos clientes, o tempo de serviço e a quantidade de tráfego descartados pelo SHADE (no cenário em que o SHADE está em ação).

No cenário sem defesa e sem ataque, a intenção é verificar se a rede estava consistente, sem problemas, antes de iniciar os testes com ataque. Sem defesa e com ataque, foi para verificar se os ataques conseguem indisponibilizar o servidor *web*. No cenário apenas o *SeVen*, verificou-se como se comporta esta defesa em ataques do tipo *high-rate*. Enquanto que na configuração de SHADE sem *whitelist*, a intenção é verificar se a simples eliminação do tráfego é necessário para mitigar o ataque.

No cenário de toda a defesa, incluindo *whitelist*, a finalidade é verificar a proposta como um todo deste trabalho, incluindo variando as formas realizar o *fingerprinting* de clientes legítimos. Por fim, o cenário com todo o sistema de defesa mas sem ataque, verificando se o sistema influencia na disponibilidade dos clientes quando não há ataque.

Observou-se um aumento na disponibilidade dos clientes legítimos, comparando os resultados sem a defesa e com SHADE. A disponibilidade saiu de 17,7% e chegou a 99,8% com a identificação por meio do cabeçalho HTTP, e uma disponibilidade de 99,5% na identificação por meio do padrão de utilização do cliente. Foi observado também uma redução no Tempo de Serviço do servidor *web*, que respondia clientes em 14,8 segundos e passou a responder em 0,2 segundo na primeira forma de identificar, enquanto que com o *fingerprinting* por meio da utilização do cliente, o Tempo de Serviço foi de 0,9 segundo. Além disso, SHADE conseguiu eliminar cerca de 80% do tráfego inútil da rede, diminuindo consideravelmente o ataque *Get-Flooding*, ou seja, o tráfego de atacante que iria consumir largura de banda e processamento, foi descartado da rede.

Como trabalhos futuros, propõe-se a utilização de mais elementos de *fingerprinting* para identificar clientes legítimos, como descrito em Takasu *et al.* [52]. Sugere-se a utilização do mecanismo de defesa SHADE contra outros tipos de ataques de negação de serviço na camada de aplicação, incorporando ataques do tipo *high-rate* e *low-rate* juntos, para verificar a efe-

tividade da utilização de *whitelist* e o descarte em outros tipos de ataque, como em ataques combinados. Propõe-se ainda adaptar a quantidade de requisição descartada por SHADE, diminuindo ou aumentando de acordo com a taxa de utilização da aplicação e/ou de acordo com a quantidade de tráfego que está chegando, tornando assim a defesa mais adaptativa em relação a proporção do ataque. Por fim, propõe-se integrar os mecanismos para identificação de clientes legítimos em estratégias conhecidas e efetivas contra outros tipos de ataques, a fim de se criar uma *whitelist* de clientes legítimos.

REFERÊNCIAS

[1] Yuri Gil Dantas. Estratégias para tratamento de ataques de negação de serviço na camada de aplicação em redes ip. Master Thesis, 2015.

[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 23 de Março de 2016.

[3] D. Kreutz, F.M.V. Ramos, P. Esteves Verissimo, C. Esteve Rothenberg, S. Azodolmolky, and S. Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76, Jan 2015.

[4] Diego R. Mafioletti. Crops – uma proposta de comutador programável de código aberto para prototipação de redes. Master Thesis, 2015.

[5] Verisign. https://www.verisign.com/en$_US$/$security-services/ddos-protection/what-is-a-ddos-attack/index.xhtml$. 2015.

[6] R. Shea and Jiangchuan Liu. Understanding the impact of denial of service attacks on virtual machines. In *Quality of Service (IWQoS), 2012 IEEE 20th International Workshop on*, pages 1–9, June 2012.

[7] 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.

[8] 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.

[9] NS Focus. http://www.prnewswire.com/news-releases/2014-mid-year-ddos-threat-report-documents-high-volume-high-rate-attacks-on-the-rise-276438821.html. 2014.

[10] João Corrêa, Vivek Nigam, Moises R. N. Ribeiro, Diego Mafioletti, and Iguatemi E. Fonseca. SHADE: uma estratégia seletiva para mitigar ataques DDoS na camada de aplicação em redes definidas por software. In *XXXIV SBrT*, pages 964–968, Santarém, Brazil, August 2016.

[11] ryu. http://osrg.github.io/ryu/. 2016.

[12] Alextian Liberato, Diego Mafioletti, Eros Spalla, Rodolfo Villaca, and Magnos Martinello. Avaliação de desempenho de plataformas para validação de redes definidas por software. In *WPerformance - XIII Workshop em Desempenho de Sistemas Computacionais e de Comunicação*, Brasília, Brasil, 2014.

[13] T. Zhang, F. Gao, W. Shen, and H. Zhang. Architecture analysis and implementation of a web-based fingerprint management system. In *2009 First International Conference on Information Science and Engineering*, pages 2866–2869, Dec 2009.

[14] U. Dubey, A. Trisal, J. Bose, M. Brabhui, and N. Ahamed. A hybrid authentication system for websites on mobile browsers. In *2014 ICACCI*, pages 266–270, Sept 2014.

[15] Dafydd Stuttard and Marcus Pinto. *The Web Application Hacker's Handbook: Discovering and Exploiting Security Flaws*. Wiley Publishing, Inc, 2008.

[16] K. x. Yang, L. Hu, N. Zhang, Y. m. Huo, and K. Zhao. Improving the defence against web server fingerprinting by eliminating compliance variation. In *2010 Fifth International Conference on Frontier of Computer Science and Technology*, pages 227–232, Aug 2010.

[17] M. T. Qassrawi and H. Zhang. Using honeyclients to detect malicious websites. In *2010 2nd International Conference on E-business and Information System Security*, pages 1–6, May 2010.

[18] G. Bianchi, H. Rajabi, A. Caponi, and G. Picierro. Conditional disclosure of encrypted whitelists for ddos attack mitigation. In *2013 IEEE Globecom Workshops (GC Wkshps)*, pages 200–206, Dec 2013.

[19] E. Y. Chen and M. Itoh. A whitelist approach to protect sip servers from flooding attacks. In *2010 IEEE International Workshop CQR 2010*, pages 1–6, June 2010.

[20] Seungwon Shin, Vinod Yegneswaran, Phillip Porras, and Guofei Gu. Avant-guard: Scalable and vigilant switch flow management in software-defined networks. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 413–424, New York, NY, USA, 2013. ACM.

[21] R. Braga, E. Mota, and A. Passito. Lightweight DDoS flooding attack detection using nox/openflow. In *Local Computer Networks (LCN), 2010 IEEE 35th Conference on*, pages 408–415, Oct 2010.

[22] S. Hommes, R. State, and T. Engel. Implications and detection of dos attacks in openflow-based networks. In *Global Communications Conference (GLOBECOM), 2014 IEEE*, pages 537–543, Dec 2014.

[23] Haopei Wang, Lei Xu, and Guofei Gu. Floodguard: A dos attack prevention extension in software-defined networks. In *Dependable Systems and Networks (DSN), 2015 45th Annual IEEE/IFIP International Conference on*, pages 239–250, June 2015.

[24] Seyed Kaveh Fayaz, Yoshiaki Tobioka, Vyas Sekar, and Michael Bailey. Bohatei: Flexible and elastic ddos defense. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.*, pages 817–832, 2015.

[25] Tanenbaum A. S. *Redes de Computadores*. Elsevier, 4a edição edition, 2003. ISBN 85-352-1185-3.

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

[27] *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.

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

[29] Slowhttptest. *Slowhttptest tool, 2013*. https://code.google.com/p/slowhttptest/ – Acesso em 02 de Fevereiro de 2015.

[30] Slowread. *Slowread, 2013*. https://blog.qualys.com/securitylabs/2012/01/05/slow-read – Acesso em 02 de Fevereiro de 2015.

[31] Seven https://github.com/ygdantas/SeVen.git. 2013.

[32] Akamai. https://www.akamai.com/us/en/our-thinking/state-of-the-internet-report/global-state-of-the-internet-security-ddos-attack-reports.jsp. 2016.

[33] P. J. Ochieng, Kani, H. Harsa, and Firmansyah. Fingerprint authentication system using back-propagation with downsampling technique. In *2016 2nd International Conference on Science and Technology-Computer (ICST)*, pages 182–187, Oct 2016.

[34] J. Bringer, C. Morel, and C. Rathgeb. Security analysis of bloom filter-based iris biometric template protection. In *2015 International Conference on Biometrics (ICB)*, pages 527–534, May 2015.

[35] Albert G. Greenberg, Gísli Hjálmtýsson, David A. Maltz, Andy Myers, Jennifer Rexford, Geoffrey G. Xie, Hong Yan, Jibin Zhan, and Hui Zhang. A clean slate 4d approach to network control and management. *Computer Communication Review*, 35(5):41–54, 2005.

[36] Hamid Farhady, HyunYong Lee, and Akihiro Nakao. Software-defined networking: A survey. *Computer Networks*, 81:79 – 95, 2015.

[37] Adam Zarek. Openflow timeouts demystified. Master Thesis, 2012.

[38] A. Lara, A. Kolasani, and B. Ramamurthy. Network innovation using openflow: A survey. *Communications Surveys Tutorials, IEEE*, 16(1):493–512, First 2014.

[39] OpenFlow. https://www.opennetworking.org/Openflow. 2016.

[40] Mininet. http://mininet.org/. 2016.

[41] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: Rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets-IX, pages 19:1–19:6, New York, NY, USA, 2010. ACM.

[42] S. Yun. An efficient tcam-based implementation of multipattern matching using covered state encoding. *IEEE Transactions on Computers*, 61(2):213–221, Feb 2012.

[43] OpenWRT. https://openwrt.org/. 2016.

[44] Open vSwitch. http://openvswitch.org/. 2016.

[45] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. Nox: Towards an operating system for networks. *SIGCOMM Comput. Commun. Rev.*, 38(3):105–110, July 2008.

[46] Eder Leão Fernandes. *NOX 1.3 Oflib, 2013*. https://github.com/CPqD/nox13oflib – Acesso em 16 de Março de 2016.

[47] Pox. *POX - Noxrepo, 2013*. http://www.noxrepo.org/pox/ – Acesso em 16 de Março de 2016.

[48] 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.

[49] Yuri Gil Dantas, Marcilio O. O. Lemos, Iguatemi Fonseca, and Vivek Nigam. Formal specification and verification of a selective defense for TDoS attacks. In *11th WRLA*, 2016.

[50] Marcilio O. O. Lemos, Yuri Gil Dantas, Iguatemi Fonseca, Vivek Nigam, and Gustavo Sampaio. A selective defense for mitigating coordinated call attacks. In *34th Brazilian Symposium on Computer Networks and Distributed Systems (SBRC)*, 2016.

[51] Roy T. Fielding and Julian F. Reschke. Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. RFC 7230, June 2014.

[52] K. Takasu, T. Saito, T. Yamada, and T. Ishikawa. A survey of hardware features in modern browsers: 2015 edition. In *2015 9th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, pages 520–524, July 2015.

[53] User Agent String.Com. http://www.useragentstring.com/pages/useragentstring.php. 2016.

[54] V. Ragavi and G. Geetha. Mitigating dos using sensing keys. In *2012 International Conference on Computing Sciences*, pages 312–315, Sept 2012.

[55] Jan Seidl. https://wroot.org/projects/goldeneye/. 2014.

[56] RNP. *RNP participa da operação de monitoramento do Sisu*. https://www.rnp.br/noticias/rnp-participa-operacao-monitoramento-sisu. – Acesso em 01 de Junho de 2016.