Módulo de Proteção contra Ataques de Negação de Serviço na Camada de Aplicação: uma Análise de Qualidade de Serviço e Experiência de Usuário*

Túlio A. Pascoal[1], João H. G. Correa[1], Rafael Brayner[1], Vivek Nigam[1], Iguatemi E. Fonseca[1]

[1]Universidade Federal da Paraíba (UFPB)
Caixa Postal 5.115 - 58.051-970 – João Pessoa – PB – Brasil

{tuliopascoal,jhenrique,rafabrayner92}@gmail.com, {vivek,iguatemi}@ci.ufpb.br

Abstract. *This paper proposes a module that can defend application layer denial of service attacks. This module works in an Apache (Web) server and presents advantages when compared with other proposals in literature, as well as similar performance to a strategy that runs as a proxy. Our experimental results show that our module delivers high levels of availability, low TTS (Time To Service), memory and CPU consumption, Quality of Service and Quality of Experience even when the server is under attack.*

Resumo. *Este artigo propõe um módulo para defesa de ataques de negação de serviço na camada de aplicação. O módulo é executado em um servidor Web Apache e apresenta vantagens dentre outros existentes na literatura, com resultados similares à estratégia que opera como um proxy. Nos experimentos realizados verificou-se que o módulo proposto apresenta alta performance em termos de disponibilidade, TTS (Time To Service), consumo de memória, CPU, QoS (Quality of Service) e QoE (Quality of Experience) da aplicação em que está sendo executado.*

1. Introdução

Ataques de Negação de Serviço (DoS – *Denial of Service*) e sua versão distribuída (DDoS – *Distributed DoS*) são considerados uns dos mais perigosos e utilizados ataques contra redes e serviços. Seu objetivo é causar indisponibilidade do serviço, website ou aplicação alvo para usuários legítimos, consumindo todos os recursos disponibilizados de forma temporária ou indefinida. A grande disponibilidade e facilidade de acesso a ferramentas capazes de gerar ataques DDoS, tornam esses ataques ainda mais poderosos e comuns. Ferramentas são facilmente encontradas na Internet, como: LOIC (*Low Orbit Ion Cannon*), R.U.D.Y (*Are You Dead Yet*), *Slowloris* etc (Infosec 2013). Muitas dessas ferramentas possuem interfaces amigáveis e simples, facilitando e alavancando ainda mais o uso das mesmas, pois não requerem conhecimentos avançados de programação e redes de computadores.

Recentemente, ataques DoS na camada de aplicação (ADoS – *Application Layer DoS*), vêm sendo bastante utilizados por atacantes (Kaspersky 2016). Ataques ADoS são mais difíceis de serem detectados, pois esses ataques exploram vulnerabilidades de

*Este trabalho foi financiado e apoiado pela CAPES, CNPq e RNP.

protocolos utilizados na camada de aplicação do modelo OSI (Xie and Yu 2009), como: HTTP, HTTPS, DNS, VoIP, FTP e SMTP. Além de permitirem aos atacantes a possibilidade de focar seu ataque somente em uma aplicação ou serviço, enquanto mantém outros disponíveis, dificultando a detecção do ataque (Xie and Yu 2009). Outro fator é que o tráfego gerado por ataques do tipo ADoS *LowRate* é similar ao de usuários legítimos, dificultando sua detecção e mitigação (Dantas et al. 2014).

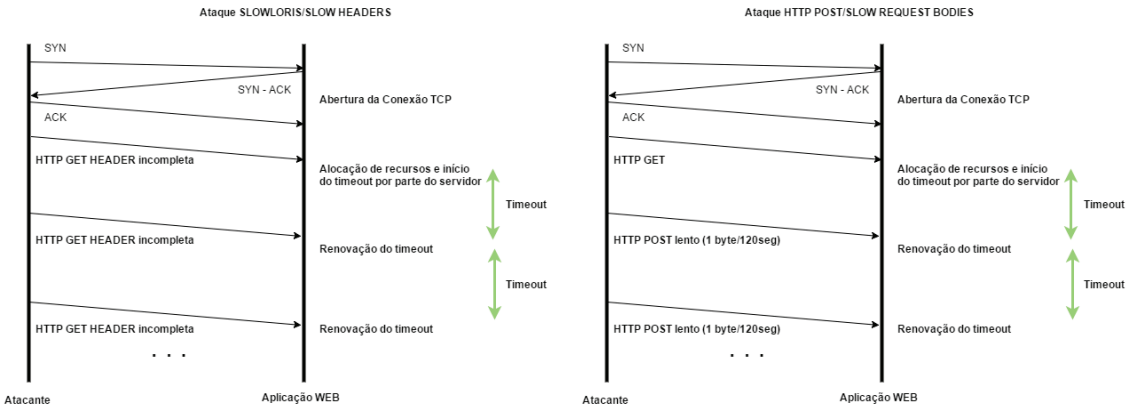
(Dantas et al. 2014) introduziram uma defesa chamada SeVen (*Selective Verification in Application Layer*) para mitigação de ataques ADoS em servidores Web. Eles demonstraram que com o SeVen ativado, o servidor Web é capaz de manter uma taxa de disponibilidade da aplicação para clientes legítimos de 95%. SeVen funciona como um *proxy*, fazendo a *interface* entre as requisições dos clientes com o servidor protegido, aumentando a complexidade da ferramenta, pois deve se preocupar, além da execução da estratégia em si, com outros aspectos como: (1) qual tipo e versão de servidor estão sendo utilizados e suas peculiaridades; (2) quais os protocolos que estão sendo utilizados, *e.g.* HTTP, HTTPS, etc; (3) segurança (integridade, confidencialidade, disponibilidade e autenticidade da ferramenta em si); (4) robustez para garantir funcionamento ininterrupto em qualquer situação, para não prejudicar o servidor que está sendo protegido; (5) limitações e dependência do sistema operacional instalado na máquina no qual a defesa está instalada. Devido a esses fatores, faz-se necessário o uso de uma abordagem menos dependente e que possa ao mesmo tempo ser eficaz e de fácil uso. Para atingir esse objetivo propomos um módulo, chamado *mod_seven* que funciona como um *mini-software* diretamente acoplado no servidor.

O Apache HTTP Server Project, mais conhecido como Apache, é o servidor Web mais popular e utilizado atualmente. Servidores Apache são usados por 55,9% entre todos os sites na rede em Dezembro de 2015 (W3tech 2016). Por ser código livre, servidores Apache fornecem a liberdade e extensibilidade de seu funcionamento, a partir da implementação de módulos (MódulosApache 2016).

Para a realização de testes que envolvem a qualidade de experiência e de serviço em websites, desenvolvemos uma ferramenta chamada webbots devido ao fato da não existência de ferramentas para a automação e execução desses tipos de testes, tanto no mercado quanto na literatura. Os webbots simulam vários usuários reais com diferentes comportamentos acessando um website, medindo parâmetros essenciais que contribuem para análise da medida de satisfação do usuário e do serviço durante a navegação.

Os objetivos e contribuições deste trabalho são, portanto: i) **mod_seven**, um módulo Apache para mitigação de ataques ADoS; ii) **webbots**, uma ferramenta para análise qualitativa da experiência de usuário e serviço; iii) **Diversos tipos de experimentos**, como *Loading Test*; *QoS*; *QoE*; replicação e comparação dos testes realizados por (Dantas et al. 2014); testes de longa duração (2 horas); e testes sob o protocolo HTTPS (não suportado pelo SeVen *proxy*); iv) **Facilitação e disseminação da utilização da estratégia**, dado que o módulo foi criado para o servidor Web mais utilizado da atualidade.

A Seção 2 apresenta a natureza, característica e funcionamento de ataques ADoS. Posteriormente, os trabalhos relacionados são apresentados na Seção 3. Na Seção 4 é apresentada a adaptação da estratégia SeVen como um módulo, bem como sua arquitetura, funcionamento e interligação com o servidor. Seção 5 demonstra a criação e carac-



(a) Simulação do ataque *Slowloris* (b) Simulação do ataque *HTTP POST*

Figura 1: Funcionamento dos ataques *Slowloris* e *HTTP POST*

terística dos *webbots* criados para análise qualitativa do módulo. A Seção 6 descreve os experimentos e discute-os. Por fim, Seção 7 expõe as conclusões e trabalhos futuros.

2. Ataques de Negação de Serviço na Camada de Aplicação

Ataques DoS consistem na interrupção temporária ou indefinida de um serviço alvo (Gu and Liu 2007), quando esses ataques exploram protocolos da camada de aplicação, podemos classificá-los como ADDoS (Xie and Yu 2009). Dentre os ataques ADDoS podemos destacar dois tipos: *Flooding* e *LowRate*. No primeiro, tem-se a geração de um enorme fluxo de tráfego, para consumir todos os recursos da aplicação, até que ela fique incapaz de atender novos clientes. O segundo, consiste na geração de tráfego similar ao de clientes legítimos, porém, utilizando-se de vulnerabilidades encontradas nos protocolos (HTTP e HTTPS, por exemplo) para manter requisições em atendimento por tempo indeterminado (Durcekova et al. 2012), assim não necessitando de grandes recursos para geração dos ataques (Dantas 2015). Existem duas grandes dificuldades para a detecção de ataques *LowRate*: sua capacidade de indisponibilizar apenas serviço(s) alvo(s), sem afetar outros disponíveis em um servidor e por seu tráfego ser bastante similar ao de clientes legítimos. Ainda mais, ferramentas que geram esses tipos de ataques são bastante simples e facilmente encontradas na Internet. Por isso, o enfoque nesses tipos de ataque neste trabalho. Abaixo tem-se a descrição e funcionamento dos dois ataques do tipo *LowRate* mais utilizados:

- **Slowloris**: 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 requisições foram finalizadas, mantendo-as no *pool* de atendimento até o valor de *timeout* pré-configurado no servidor. A partir de certo momento, todos os recursos estarão sendo consumidos pelas requisições maliciosas, indisponibilizando a aplicação. Figura 1(a) ilustra o ataque *Slowloris*;
- **HTTP POST**: Este ataque 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. Na Figura 1(b) temos o funcionamento desse tipo de ataque;

Em ambos os ataques a quantidade de tráfego gerada pelo atacante é pequena, passando assim desapercebida por defesas de monitoramento de tráfego.

3. Trabalhos Relacionados

Ataques distribuídos na camada de aplicação (ADDoS) vêm se tornando cada vez mais utilizados para indisponibilizar aplicações Web hoje em dia e seu uso não para de crescer. Além disso, (Kaspersky 2016) prevê que seu uso tende a ser alavancado. Atualmente, segundo um relatório de segurança da empresa Incapsula (Incapsula 2016), foram mitigados mais ataques na camada de aplicação do que na camada de rede. Eles também relatam que ataques ADoS estão se tornando cada vez mais pesados (470 GB de tráfego gerado em um único ataque no segundo quarto de 2016) e inteligentes.

Em (Dantas et al. 2014) uma nova estratégia para mitigação de ataques ADDoS foi implementada e criada. Nomeada de SeVen, a ferramenta é utilizada como um *proxy* e é capaz de defender servidores contra ataques DDoS. O SeVen utiliza simples funções de probabilidade e de distribuição uniforme, que quando o servidor encontra-se sobrecarregado, determinam as chances de novas requsições serem atendidas ou não. Nos experimentos realizados em (Dantas et al. 2014) e (Dantas 2015), a aplicação Web sendo protegida pela estratégia obteve disponbilidade em torno de 95% quando atacada contra dois tipos diferentes de ataques ADDoS, *Slowloris* e HTTP POST.

Outros módulos Apache existentes no mercado propõe-se a mitigar esses tipos de ataques (Monshouwer 2013) (Morimoto 2013) (Reqtimeout 2014). O *mod_antiloris* tem como estratégia a contagem de conexões simultâneas abertas por um mesmo endereço IP. Quando essa contagem superar o limiar configurado na defesa (o seu padrão é 10), o módulo rejeita todas as requisições provenientes daquele IP. O *mod_pacify_loris* (Morimoto 2013) possui a mesma estratégia de defesa (mas utiliza 50 conexões por padrão), porém ainda implementa mais duas análises: contagem de GET HEADER enviados por uma mesma requisição e a taxa de GET HEADER enviados por uma requisição por segundo, a fim de rejeitar requisições lentas, o que pode ocasionar falso positivos, prejudicando clientes legítimos que tenham conexões lentas. A abordagem desses módulos prejudicam clientes legítimos situados em redes internas, uma vez que para acesso externo todos eles utilizam um único IP público, enquanto a defesa proposta não faz distinção entre seus clientes.

Já o *mod_reqtimeout* (Reqtimeout 2014), o mais atual e utilizado dentre eles, baseia-se em uma análise mais avançada das taxas de envio dos cabeçalhos e corpo das requisições. O módulo possui uma diretiva configurável chamada *RequestReadTimeout* que possui dois parâmetros configuráveis para cada um dos campos (cabeçalho e corpo) de uma requisição, que são: *timeout* e *minrate*. O seu grande diferencial quanto aos outros módulos é que ele avalia esses parâmetros em conjunto e a cada vez que pacotes de dados de uma requisição chegam ao servidor, o módulo renova suas janelas de *timeout*. Porém, com uma simples modificação na forma de execução do ataque, é possível burlar

as estratégias do *mod_reqtimeout*, o que não ocorre no *mod_seven*

O JMeter (JMeter 2016) é uma ferramenta gratuita que permite a realização de testes de carga em websites, permitindo a avaliação da qualidade de serviço (QoS - Quality of Service) por meio de valores técnicos de rede. Porém usuários percebem o desempenho de uma forma subjetiva que é conhecida como qualidade de experiência (QoE - Quality of Experience) (Shaikh et al. 2010). Existe uma estreita relação entre a qualidade de serviço e a qualidade de experiência dos usuários (Khirman and Henriksen 2002). Devido a esta relação, o longo tempo para se obter resposta tem se refletido como um dos principais fatores para a frustração de um usuário (Schatz et al. 2013). Além desse parâmetro de avaliação, os reloads realizados na página refletem condições insatisfatórias para o usuário e pode ser considerada a única maneira imparcial de avaliação (Khirman and Henriksen 2002). Como o JMeter apenas realiza testes voltados para análise de QoS, pois não simulam usuários reais, os webbots criados neste trabalho tentam se aproximar ao máximo do comportamento de vários usuários reais diferentes bem como monitora e avalia parâmetros de qualidade para interpretar a satisfação dos usuários.

4. O Módulo Proposto

4.1. Estratégia de Defesa SeVen

O SeVen é baseado em estratégia seletiva, que seleciona quais de novas requisições, dado o momento em que o servidor encontra-se sobrecarregado, serão atendidas ou rejeitadas, a partir de funções de probabilidade. Em resumo, quando os recursos para o atendimento de novas conexões encontram-se esgotados e uma nova requisição R chega, o SeVen funciona da seguinte maneira:

1. SeVen decide a partir de uma função de probabilidade FP_1 se a nova requisição R será aceita ou não;
2. Se SeVen decide não processar R, uma mensagem erro é enviada ao usuário e a conexão com cliente é fechada (*close_connection*);
3. Se SeVen aceita processar R, ele decide a partir de uma função de probabilidade FP_2 qual conexão que atualmente está sendo processada pelo servidor deve ser substituída por R no *pool* de atendimento da aplicação.

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:

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

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:

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

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 não

entram no contexto desse artigo). Caso FP_1 decida que a requisição id_5 não será atendida, uma mensagem de erro é enviada e a conexão fechada. Assim:

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

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:

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

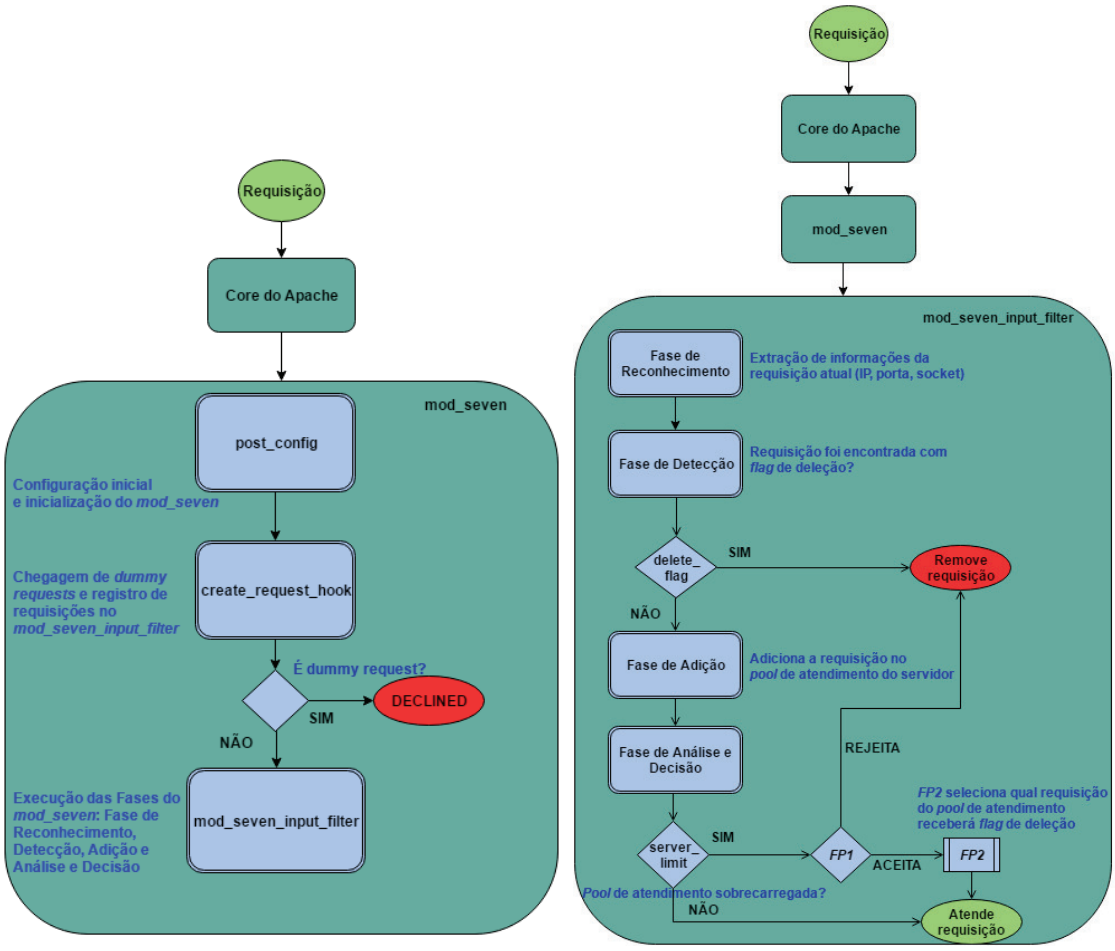
SeVen funciona pois quando um servidor encontra-se sobrecarregado, ele muito provavelmente estará sofrendo um ataque DoS. Consequentemente, o seu *pool* de atendimento estará repleto de conexões atacantes, ou com mais atacantes do que clientes. Portanto, quando o SeVen decide processar uma nova requisição, a chance de remover uma conexão atacante do *pool* é muito maior do que de um cliente legítimo (Dantas et al. 2014).

4.2. O Módulo Apache Proposto: *mod_seven*

De acordo com (Kew 2007), o Apache HTTP Server é composto por um pequeno *core* e um conjunto de módulos, que acrescentam funcionalidades ao Apache. O processo de atendimento de requisições em servidores Apache funciona da seguinte forma: assim que uma nova requisição chega ao servidor, o *core* do Apache recebe-a e realiza uma checagem de quais módulos ativados no servidor tem interesse de processar a requisição e a repassa. Dessa forma, os módulos executam suas funções extras em conjunto com o atendimento normal do servidor *web*.

A primeira ação do módulo *mod_seven* é auto declarar-se ao *core* do Apache, para deixá-lo informado sobre sua existência, ativação e registro de seus ganchos. Isso é realizado pela diretiva *AP_MODULE_DECLARE_DATA mod_seven*. Ganchos funcionam como métodos que informam ao *core* do Apache quais tipos de requisições e em qual etapa (processamento da requisição ou geração de conteúdo, por exemplo) o módulo deseja operar. O *mod_seven* possui três ganchos: (1) o *ap_hook_post_config*, utilizado para recolher informações do servidor, como tamanho do *pool* de atendimento, criação e alocação de *threads* da defesa; (2) o *ap_hook_create_request* para detecção de *dummy requests* (Hayden 2008), um *bug* do Apache encontrado durante a implementação e registro das requisições no filtro do módulo; (3) o *mod_seven_input_filter* que é o filtro que executa praticamente toda a estratégia. Quando uma requisição é registrada em um filtro, todo e qualquer dado enviado pela requisição será analisado e processado pelo filtro. Para simplificação e melhor entendimento, o funcionamento do *mod_seven_input_filter* do módulo foi dividido em 4 fases distintas, explicadas abaixo:

- **Fase de Reconhecimento**: Nessa fase o filtro extrai informações da requisição (endereço IP, porta e *socket* da conexão), e aloca em uma variável-estrutura interna da APR do tipo *worker_score* para representar a requisição no *pool* de atendimento do Apache, chamado de *scoreboard*;
- **Fase de Detecção**: Aqui acontece uma adaptação necessária à estratégia para adequar-se ao comportamento de Módulos Apache. Como o Apache não permite a extração e manuseamento direto de requisições que se encontram no *scoreboard*, essa fase realiza uma varredura no *scoreboard* atual da aplicação verificando



(a) Organização e ordem de fluxo de execução do *mod_seven*

(b) Organização e ordem de fluxo das fases do *mod_seven_input_filter*

Figura 2: Divisão de processos e execução do *mod_seven*

se a requisição atual (que está sendo processada pelo filtro) está com *flag* que foi selecionada para ser removida pela estratégia, caso positivo, a conexão daquela requisição será fechada imediatamente; caso negativo, o fluxo do módulo continua para a próxima fase, a Fase de Adição;

- **Fase de Adição**: É uma fase simples e direta, após colher as informações da requisição e verificar que a mesma não encontra-se selecionada para deleção, o módulo conclui que ela está apta a ser atendida e processada, e a requisição é adicionada ao *pool* de atendimento do Apache.
- **Fase de Análise e Decisão**: É a fase mais completa e que aplica toda a lógica da estratégia. Uma contagem de requisições no *pool* de atendimento é realizada, similar ao da Fase de Detecção, para concluir se o servidor encontra-se sobrecarregado ou não, utilizando uma variável para comparar com o valor do parâmetro *server_limit* (que representa o máximo de conexões simultâneas que o servidor pode atender). Se o servidor estiver sobrecarregado, a função de probabilidade FP_1 decide a aceitação ou não da requisição. Caso a requisição seja rejeitada pela estratégia, sua conexão é automaticamente fechada usando *APR_DECLINED*, *ap_conn_close_close*

e *apr_socket_close*. Caso contrário, a requisição atual será atendida, para isso a função de distribuição uniforme FP_2 (origem da característica seletiva da estratégia) seleciona uma requisição para ser substituída do *pool* de atendimento. Após a escolha, o módulo resgata informação do *worker_score* escolhido, e adiciona a *flag* de deleção, assim, quando qualquer dado referente àquela requisição chegar ao servidor, a mesma será rejeitada automaticamente na Fase de Detecção.

Enquanto a aplicação não se encontra sobrecarregada, o módulo simplesmente executa as Fases de Reconhecimento e Adição. Na Figura 2(a) tem-se o fluxo de funcionamento e processos do *mod_seven* realizados pelos seus ganchos e métodos internos e na Figura 2(b) tem-se uma visão geral da divisão e execução de processos ocorridos nas distintas fases do *mod_seven_input_filter*.

5. Webbots - Medindo Qualidade de Experiência e de Serviço

Qualidade de Serviço (do inglês Quality of Service - QoS) na web está diretamente relacionada ao desempenho perceptível pelos usuários. Essa qualidade pode ser medida quantitativamente por meio da análise de diferentes aspectos do serviço como taxa de erro, taxa de bits, atraso de transmissão, disponibilidade, taxa de transferência, dentre outros. QoS são tipicamente parâmetros de rede. No entanto, usuários percebem o desempenho de forma subjetiva, eles não estão interessados em valores técnicos da rede, e sim em obter resposta dentro de um tempo razoável. Essa percepção subjetiva é conhecida como Qualidade de Experiência (do inglês Quality of Experience - QoE) (Shaikh et al. 2010).

O JMeter (JMeter 2016) é uma das mais utilizadas ferramentas para verificação de desempenho de websites, por apresentar uma vasta gama de configurações, além de ser uma ferramenta já conceituada e gratuita. JMeter é um projeto do Apache Software Foundation que permite realizar testes simulando vários usuários realizando tarefas em website. Ele permite especificar diversos tipos de parâmetros, dentre eles, o número de usuários a serem criados, o tempo total em que esses usuários devem ser criados e o número de vezes que cada usuário vai realizar as tarefas. Além disso, ele também permite definir o fluxo das atividades, como por exemplo, executar o login apenas uma vez. No entanto, o JMeter não é capaz de realizar testes mais elaborados, principalmente em relação a qualidade de experiência e serviço da aplicação, quando tratando-se na simulação comportamental de usuários reais.

Devido ao fato de que a qualidade de serviço está intimamente ligada a qualidade de experiência dos usuários (Khirman and Henriksen 2002), desenvolveu-se os webbots para simular um grande número de usuários reais, realizando tarefas em um website, e medir a satisfação dos mesmos durante o processo de navegação. Os parâmetros de qualidade escolhidos para mensurar a satisfação foram o valor máximo de tempo que um robô espera por uma resposta do servidor e o número máximo de *"reloads"* realizados durante a navegação. O primeiro está relacionado com um dos maiores problemas que tem sido enfrentado pelos usuários na web e que causa mais frustração, que é o longo tempo para se obter resposta (Schatz et al. 2013) (Selvidge 2003). O segundo está relacionado com as condições insatisfatórias no website que podem fazer com que o usuário pressione o botão de recarregar. Apesar dessa não ser a única maneira de avaliar a insatisfação do usuário em relação qualidade da comunicação, pode ser considerada a única maneira de se obter uma forma de medida independente e imparcial (Khirman and Henriksen 2002).

Os webbots oferecem alguns diferenciais dos quais podem-se listar três principais características que são exclusivas: Primeiro, os webbots 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, ao contrário do JMeter que não oferece essa funcionalidade. Segundo, quando os webbots estão realizando uma sequência de tarefas e se deparam com um erro causado pela indisponibilidade do , 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. 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.

Enfim, os webbots são uma ferramenta inédita capaz de realizar testes e análises qualitativas de um website a partir da simulação de usuários "robôs"que simulam usuários reais interagindo com a aplicação e colhendo métricas importantes para o mensarumento da qualidade de navegação.

6. Experimentos e Resultados

6.1. Cenário e Configuração dos Experimentos

Os testes foram realizados usando três máquinas em dois diferentes *Campus* da Universidade Federal da Paraíba (UFPB). Duas máquinas distintas situadas no *Campus* V gerando o tráfego de clientes legítimos e atacantes, e uma máquina no *Campus* I hospedando a página Web padrão do Apache. As máquinas para geração de tráfego possuem processador Intel i5-3470 de 3.20Ghz e 4GB de RAM e o servidor um Intel Xeon E5-2620 de 2.00GHz e 8GB de RAM. O objetivo dessa configuração de cenário é de simular, com um maior grau de realidade, um ataque DoS, em que o tráfego situa-se em redes distintas e separadas fisicamente. Para geração do tráfego cliente utilizamos a ferramenta Siege (ferramenta de benchmark para medir desempenho de aplicações Web (Siege 2015)) e para o tráfego atacante duas ferramentas: *Slowloris* (Slowloris 2013) e *Slowhttptest* (Slowhttptest 2013).

O servidor foi configurado com um *timeout* (tempo, em segundos, que o servidor aguarda para receber dados de requisições em uma mesma conexão) de 40 segundos e *MaxRequesWorkers* (número máximo de requisições simultâneas atendidas pelo servidor) de 200, essa configuração representa um servidor de pequeno/médio porte. À quesito de comparação de resultados, a configuração e cenários dos experimentos realizados nesse trabalho foram replicados de acordo com os realizados por (Dantas et al. 2014).

- **Tráfego atacante**: 250 atacantes para cada tipo de ataque, enviando requisições a cada 35 segundos. Aproximadamente 7,14 requisições por segundo;
- **Tráfego de clientes legítimos**: 100 clientes simultâneos enviando requisições em um intervalo de 0 a 3 segundos, a fim de melhor simular um tráfego Web legítimo. Aproximadamente 10 requisições por segundo.
- **Duração**: três repetições para os testes de 5 minutos e uma para os testes de 2 horas;

Tabela 1: Comparação dos resultados entre *mod_seven* e SeVen *proxy*

	mod_seven		**SeVen *proxy***	
	Disponibilidade	TTS	Disponibilidade	TTS
Sem Ataque	100%	0,03s	100%	0,03s
Slowloris	98,7%	0,07s	97,3%	0,05s
HTTP POST	95,1%	0,02s	94,5%	0,06s

Nós utilizamos os seguintes parâmetros como métricas de desempenho: i) **Disponibilidade**: Porcentagem dos clientes atendidos com sucesso; ii) **TTS**: Tempo médio de resposta para cada requisição; iii) **Consumo de memória e CPU**: Porcentagem do consumo médio de memória e CPU durante o teste;

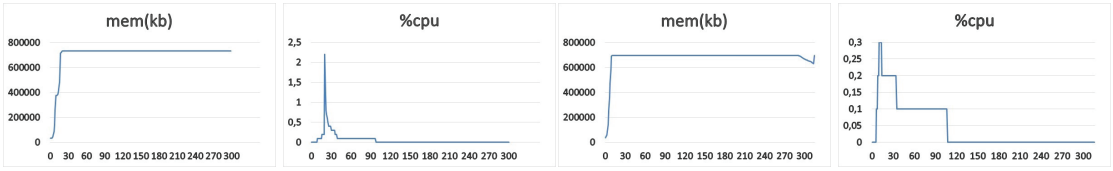
Um segundo tipo de experimentos foi realizado utilizando os webbots discutidos na seção 5 para uma melhor análise do módulo em quesito de *QoS* e *QoE* da aplicação Web. Para alcançar esse objetivo, uma aplicação Web que simula a página do SISU (Sistema de Seleção Unificada (veja `http://sisu.mec.gov.br/`) para inscrições de alunos em cursos superiores das universidades brasileiras, que ocorrem semestralmente e recebem um enorme tráfego em um curto período de tempo; e que já enfrentaram casos de ataques DoS (RNP 2016). O *site* consiste de cinco páginas com formulários que devem ser preenchidas pelos alunos, desde a primeira página de *Login*, Escolha de Cursos (duas opções), Modalidade até a quinta e última página, que finaliza a inscrição. Em algumas páginas acontecem consultas ao banco de dados da aplicação, para listar universidades e cursos, por exemplo. Para hospedar a página foi utilizado o Apache Tomcat, versão 8.0.14 devido a necessidade de executar procedimentos Java, não suportado pelo Apache Server. Também foi necessário a instalação de um servidor Apache, funcionando como um *proxy* (com o *mod_seven* instalado), recepcionando requisições e repassando-as ao Tomcat, dado que o módulo não funcionaria diretamente no Tomcat.

Cada webbot preenche cada um dos cinco formulários necessários para finalizar a inscrição em uma velocidade randômica e compatível com a humana, ou seja, alguns robôs podem ser mais rápidos que outros. Quando encontram algum tipo de erro ou a resposta recebida ultrapassou **20 segundos**, o webbot faz um *"reloads"* na página para reiniciar a inscrição a partir do último formulário preenchido com sucesso. Robôs realizam no máximo até **três** *"reload"* em uma mesma página ou **10** no total (entre as cinco páginas) antes de desistirem de realizar a inscrição. Se um robô não desistiu da inscrição pelos fatores acima explicados, eles concluíram a inscrição com sucesso.

Tabela 2: Resumo dos resultados nos experimentos de 2 horas do *mod_seven*

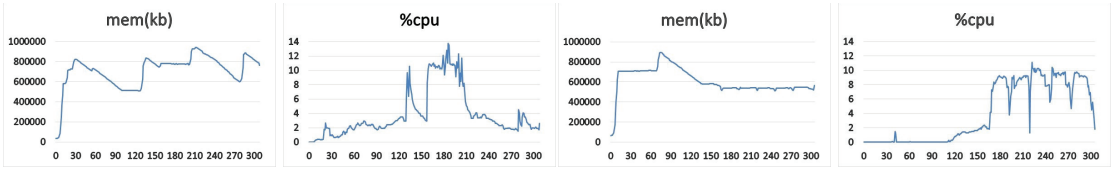
	Protocolo HTTP		**Protocolo HTTPS**	
	Disponibilidade	TTS	Disponibilidade	TTS
Slowloris	97,5%	0,14s	91,6%	0,09s
HTTP POST	91,2%	0,05s	92,4%	0,07s

Por SeVen se tratar de uma defesa agnóstica, os testes com os robôs medem o desempenho da defesa nos casos em que (mesmo com menores chances de ocorrerem) clientes legítimos tenham suas conexões fechadas durante a realização da inscrição. Assim, simulando um cenário real de aplicações Web, principalmente aquelas que precisam ar-



(a) Memória e CPU no ataque *Slowloris* (b) Memória e CPU no ataque *HTTP POST*

Figura 3: Gráfico do consumo de memória e CPU com ataque e sem *mod_seven*



(a) Memória e CPU no ataque *Slowloris* (b) Memória no ataque *HTTP POST*

Figura 4: Gráfico do consumo de memória e CPU com ataque e com *mod_seven*

mazenar sessões dos usuários e com vários formulários para preenchimento, por exemplo: *Booking* de passagens aéreas, operações em *Internet Banking*, *E-Commerce* etc. Nesses experimentos, as configurações dos servidores e o tráfego atacante foram as mesmas dos testes anteriores. O tráfego dos webbots foi de 1.000 robôs por experimento, variando sua taxa de criação em 10, 50 e 100 por minuto por teste realizado.

6.2. Resultados e Discussão

Comparando com o SeVen *proxy*, os resultados (Tabela 1) do módulo mostram um pequeno aumento da disponibilidade da aplicação (acréscimo médio de 1%) quando comparado com os resultados nos mesmos experimentos realizados em (Dantas et al. 2014). Além de possibilitar a aplicação da estratégia no protocolo HTTPS (não suportado pelo SeVen *proxy*) e em qualquer outro, desde que suportado pelo Apache. Outra vantagem é que a estratégia pode ser melhor difundida dado a preferência de uso dos servidores Apache pela comunidade, além de fácil instalação e utilização.

Tabela 3: Disponibilidade, TTS, consumo de memória e CPU dos testes realizados

	Sem mod_seven				**Com mod_seven**			
	Disponibilidade	TTS	Memória	CPU	Disponibilidade	TTS	Memória	CPU
Sem Ataque	100%	0,01s	0,9%	7,8%	100%	0,03s	0,9%	8,7%
Slowloris	0,0%	∞	17,5%	0,0%	98,7%	0,07s	18,2%	2,6%
HTTP POST	0,0%	∞	16,6%	0,0%	96,6%	0,03s	13,5%	1,8%

Pela Tabela 3 percebe-se que o *mod_seven* não influencia na disponibilidade da aplicação em situações normais (sem ataque), identifica-se um pequeno aumento do consumo de CPU (de 7,8% para 8,7%) e de 0,02 segundos no TTS, devido aos processamentos adicionais realizados pelo módulo. Analisando os cenários com ataque, verifica-se a eficiência do *mod_seven*, que manteve a aplicação com uma disponibilidade de 98,7% e 96,6% nos ataques *Slowloris* e *HTTP POST*, respectivamente, e com baixo TTS. Pela Figura 4 nota-se que, o consumo de memória quando utilizando o módulo não é elevado em ambos os ataques. Outro fator interessante é o consumo de CPU ser nulo (com pico de 0,3%) quando sob ataque e sem módulo, isso é uma prova da eficiência do ataque, pois, uma vez que a aplicação está indisponível, a mesma não processa mais nenhuma requisição, fazendo com que não haja mais consumo de CPU (ver (Figura 3). Já no

cenário com *mod_seven* (Figura 4) temos o consumo de recursos intercalados, mostrando que o servidor encontra-se ativo (atendendo requisições).

Tabela 4: Resumo dos resultados dos experimentos com os robôs

Taxa de Criação: 10 robôs/minuto				
Ataque	Apache + Tomcat		Apache + Tomcat + *mod_seven*	
	Sucesso	Falharam	Sucesso	Falha
Sem Ataque	1000	0	1000	0
Slowloris	0	1000 (2/998)	947	53 (23/30)
HTTP POST	0	1000 (122/878)	999	1(0/1)

Taxa de Criação: 50 robôs/minuto				
Ataque	Apache + Tomcat		Apache + Tomcat + *mod_seven*	
	Sucesso	Falharam	Sucesso	Falha
Sem Ataque	1000	0	1000	0
Slowloris	0	1000 (0/1000)	929	71 (12/59)
HTTP POST	0	1000 (58/942)	943	57 (4/53)

Taxa de Criação: 100 robôs/minuto				
Ataque	Apache + Tomcat		Apache + Tomcat + *mod_seven*	
	Sucesso	Falharam	Sucesso	Falha
Sem Ataque	1000	0	1000	0
Slowloris	0	1000 (0/1000)	913	87 (18/69)
HTTP POST	0	1000 (1/999)	937	63 (1/62)

Nos testes de 2 horas (HTTP e HTTPS), confirmou-se o desempenho do *mod_seven*. Verifica-se uma pequena queda na disponibilidade para o ataque *HTTP POST* (Tabela 2), uma vez que a taxa de requisições por segundo aumentou (devido ao uso da ferramenta *Slowhttptest*), transformando o ataque em um *mini-flooding*, dando indícios que o *mod_seven* possa obter bons resultados contra ataques do tipo *Flooding*. No HTTPS, verificou-se que o *Slowloris* mostrou-se incapaz de realizar o ataque, por isso foi utilizado o *Slowhttptest*. Houve uma redução da disponibilidade e um pequeno aumento no TTS, sobretudo porque o protocolo HTTPS aplica mais métodos de segurança e criptografia (contudo, uma maior e mais específica análise deve ser aplicada em relação ao *overhead* causado pelo HTTPS). Outra importante conclusão é da robustez e bom gerenciamento de *threads* e processos do módulo, uma vez que suportou uma carga elevada (recebeu cerca de 79.945 pacotes atacantes e 422.511 requisições de clientes nos testes de 2 horas).

Em relação ao *mod_antiloris* e *mod_pacify_loris*, o *mod_seven* se mostra uma defesa mais inteligente, pois não discrimina as requisições recebidas, todas possuem as mesmas chances de serem atendidas. Esses módulos utilizam uma estratégia discriminatória, dado que bloqueiam requisições baseado no IP público da conexão. Assim, prejudicando

usuários que estejam numa mesma rede interna (Abordagem bastante utilizada em redes de grandes organizações, no qual um único IP pode representar mais de uma máquina da sua rede interna). Pela tática da taxa de cabeçalhos por segundo do *mod_pacify_loris*, ele pode erroneamente rejeitar requisições de clientes legítimos, porém com conexões lentas.

Já o *mod_reqtimeout*, apesar de possuir uma estratégia mais elaborada, baseada em *timeouts* e *minrate* renováveis, possui vulnerabilidades. Atacantes podem utilizar estratégias para detectar esses valores a partir de experimentos prévios ao ataque. Por exemplo, mandando requisições em diferentes intervalos de tempo, e verificando o comportamento e as respostas do servidor, encontra-se um valor aproximado do tempo que suas conexões mantêm-se em atendimento até que comecem a ser rejeitadas. Esse valor de tempo é provavelmente o *timeout* configurado na defesa. Com essa informação, realiza-se o ataque com o *timeout* de suas conexões com um valor próximo ao detectado, assim, renovando suas conexões antes de serem removidas, mantendo-as indefinidamente em atendimento, burlando a defesa do *mod_reqtimeout*.

Tabela 4 expõe os resultados obtidos nos testes com os robôs no SISU, no qual podemos ver que quando a aplicação não estava protegida pelo *mod_seven*, a maioria dos robôs não conseguiram realizar a inscrição com sucesso, na sua grande maioria, devido ao longo tempo de espera para receeber a resposta do servidor. Para os robôs que falharam foi utilizado a seguinte notação: $T(N_R,T_R)$, T é o total de robôs que falharam, N_R, a quantidade de robôs que falharam pelo número máximo de *"reload"* e T_R pelo tempo de espera de resposta do servidor. Com o *mod_seven*, todavia, a grande maioria dos robôs conseguiram realizar suas inscrições em todos os casos realizados. Mesmo no cenário de maior tráfego (100 robôs/minuto) a taxa de sucesso foi superior a 91%.

7. Conclusão e Trabalhos Futuros

Este artigo apresenta uma solução para uns dos maiores problemas atuais na Internet, ataques DoS. Mas que também facilita o uso e gerenciamento (a partir da abordagem de módulos) de uma defesa eficaz, chamada SeVen. Além de proporcionar seu uso em diversos tipos de protocolos (SeVen *proxy* (Dantas 2015) só funciona no HTTP), desde que suportados pelo Apache (servidor Web mais utilizado atualmente). O *mod_seven* obteve melhores resultados em diversos cenários de experimentos e quando comparado com o SeVen *proxy* e outros módulos de defesa. Concomitantemente, com baixo consumo de CPU e memória e garantindo a *QoS* da aplicação, verificado a partir de experimentos com robôs simulando inscrições no *site* do SISU, um cenário mais complexo, com páginas contendo vários formulários, consulta em banco de dados e que necessitam de conexões persistentes dos usuários. Como trabalho futuro, objetiva-se testar o *mod_seven* na mitigação de ataques DoS em outros protocolos suportados pelo Apache; testá-lo contra ataques *Slowread*, ataque do tipo *Lowrate* mais recente (Park et al. 2015), bem como ataques *Flooding*. Ainda, estudar o comportamento do módulo e dos ataques em outros tipos de MPMs do Apache.

Referências

[Dantas 2015] Dantas, Y. G. (2015). Estratégias para tratamento de ataques de negação de serviço na camada de aplicação em redes ip. Master Thesis in Portuguese.

[Dantas et al. 2014] Dantas, Y. G., Nigam, V., and Fonseca, I. E. (2014). A selective defense for application layer ddos attacks. In *Intelligence and Security Informatics Conference (JISIC), 2014 IEEE Joint*, pages 75–82. IEEE.

[Durcekova et al. 2012] Durcekova, V., Schwartz, L., and Shahmehri, N. (2012). Sophisticated denial of service attacks aimed at application layer. In *ELEKTRO, 2012*, pages 55–60. IEEE.

[Gu and Liu 2007] Gu, Q. and Liu, P. (2007). Denial of service attacks. *Handbook of Computer Networks: Distributed Networks, Network Planning, Control, Management, and New Trends and Applications*, 3:454–468.

[Hayden 2008] Hayden, M. (2008). *Apache 2.2: internal dummy connection*. https://major.io/2008/09/23/apache-22-internal-dummy-connection/. Acessado em: 25 de Julho de 2015.

[Incapsula 2016] Incapsula (2016). *DDoS Threat Landscape Report 2015-2016*. https://lp.incapsula.com/DDoSThreatLandscapeReport2015-2016_LP.html. Acessado em: 27 de Setembro de 2016.

[Infosec 2013] Infosec (2013). *Layer 7 DDoS Attacks*. http://resources.infosecinstitute.com/layer-seven-ddos-attacks/. Acessado em: 25 de Outubro de 2015.

[JMeter 2016] JMeter (2016). *The Apache Software Foundation - Apache JMeter*. http://jmeter.apache.org/. Acessado em: 01 de Dezembro de 2016.

[Kaspersky 2016] Kaspersky (2016). *Kaspersky DDoS Intelligence Report for Q1 2016*. https://securelist.com/analysis/quarterly-malware-reports/74550/kaspersky-ddos-intelligence-report-for-q1-2016/. Acessado em: 22 de Agosto de 2016.

[Kew 2007] Kew, N. (2007). *The Apache modules book: application development with Apache*. Prentice Hall Professional.

[Khirman and Henriksen 2002] Khirman, S. and Henriksen, P. (2002). Relationship between quality-of-service and quality-of-experience for public internet service. In *In Proc. of the 3rd Workshop on Passive and Active Measurement*.

[Monshouwer 2013] Monshouwer, K. (2013). *mod_antiloris*. https://sourceforge.net/projects/mod-antiloris/. Acessado em: 10 de Agosto de 2015.

[Morimoto 2013] Morimoto, S. (2013). *mod_pacify_loris*. http://mod-pacify-slowloris.googlecode.com/svn/trunk/mod_pacify_loris.c. Acessado em: 10 de Agosto de 2015.

[MódulosApache 2016] MódulosApache (2016). *Developing modules for the Apache HTTP Server 2.4*. http://httpd.apache.org/docs/2.4/developer/modguide.html. Acessado em: 07 de Outubro de 2016.

[Park et al. 2015] Park, J., Iwai, K., Tanaka, H., and Kurokawa, T. (2015). Analysis of slow read dos attack and countermeasures on web servers. *International Journal of Cyber-Security and Digital Forensics (IJCSDF)*, 4(2):339–353.

[Reqtimeout 2014] Reqtimeout (2014). *mod_reqtimeout*. https://httpd.apache.org/docs/2.4/mod/mod_reqtimeout.html. Acessado em: 11 de Agosto de 2015.

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

[Schatz et al. 2013] Schatz, R., Hoßfeld, T., Janowski, L., and Egger, S. (2013). From packets to people: quality of experience as a new measurement challenge. In *Data traffic monitoring and analysis*, pages 219–263. Springer.

[Selvidge 2003] Selvidge, P. (2003). Examining tolerance for online delays. *Usability News*, 5(1):1–5.

[Shaikh et al. 2010] Shaikh, J., Fiedler, M., and Collange, D. (2010). Quality of experience from user and network perspectives. *annals of telecommunications-annales des telecommunications*, 65(1-2):47–57.

[Siege 2015] Siege (2015). *Linux man page: siege - An HTTP/HTTPS stress tester*. http://linux.die.net/man/1/siege. Acessado em: 18 de Dezembro de 2015.

[Slowhttptest 2013] Slowhttptest (2013). *Slowhttptest tool*. https://code.google.com/p/slowhttptest/. Acessado em: 02 de Fevereiro de 2015.

[Slowloris 2013] Slowloris (2013). *Slowloris tool*. http://ha.ckers.org/slowloris/. Acessado em: 02 de Fevereiro de 2015.

[W3tech 2016] W3tech (2016). *Usage of web servers for website*. http://w3techs.com/technologies/overview/web_server/all. Acessado em: 18 de Agosto de 2016.

[Xie and Yu 2009] Xie, Y. and Yu, S.-Z. (2009). Monitoring the application-layer ddos attacks for popular websites. *Networking, IEEE/AcM Transactions on*, 17(1):15–25.