Um Módulo de Defesa para Ataques DDoS na Camada de Aplicação usando Estratégias Seletivas

Túlio A. Pascoal, João H. G. Correa, Vivek Nigam e Iguatemi E. Fonseca

***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 entre outros existentes na literatura, além de resultados similares a outra estratégia utilizada para o mesmo fim, mas que opera como um *proxy*. Nos experimentos realizados mostrou-se que o módulo proposto apresenta bons resultados em termos de disponibilidade, TTS (*time to service*), consumo de memória e CPU da máquina em que está sendo executado.**

***Palavras-Chave*— Ataques de negação de serviço, Segurança na *Internet*, Redes de computadores.**

***Abstract*— This paper proposes a module that can be used as a defense against applications denial of service attacks. This module works in an Apache (*Web*) server and presents advantages when compared with others proposals in literature, as well as similar performance to another strategy that runs as a *proxy*. The experimental results show that the proposed module presents appropriated values for availability, TTS (*time to service*), and memory and CPU consumption.**

***Keywords*— Denial of service attacks, Internet security, Computer networks.**

I. Introdução

Ataques de Negação de Serviço (DoS – *Denial of Service*), são considerados uns dos mais perigosos e utilizados ataques contra redes e serviços [1]. Seu objetivo é causar indisponibilidade do serviço, website ou aplicação alvo para usuários honestos, consumindo todos os recursos disponibilizados de forma temporária ou indefinida. Seu poder é exponencialmente aumentado com a aplicação de Ataques de Negação de Serviço Distribuídos (DDoS – *Distributed DoS*) [2]. No DDoS, a fonte do ataque não é mais única, e sim dezenas, centenas e até milhares [3]. Ataques DoS na camada de aplicação (ADoS – *Application Layer DoS*) são ainda mais difíceis de serem detectados, pois o alvo desses ataques são as vulnerabilidades de protocolos utilizados na camada de aplicação do modelo OSI [4], como: HTTP, HTTPS, DNS, VoIP, FTP e SMTP. ADoS permitem aos atacantes a possibilidade de focar seu ataque somente em uma aplicação ou serviço, deixando outros serviços disponíveis, e assim, dificultando a detecção do ataque [4]. Outro fator é que o tráfego gerado por ataques do tipo ADoS *LowRate* é similar ao de usuários honestos, dificultando sua detecção e mitigação [5].

Uma defesa para mitigação de ataques ADoS, chamada SeVen (*Selective Verification in Application Layer*) foi desenvolvida com base em uma estratégia seletiva. O SeVen é uma estratégia nova na literatura, datada de 2014, que possui resultados bastante satisfatórios, mantendo servidores *Web*, quando sob ataque, com cerca de 95% de disponibilidade [5]. SeVen funciona como um *proxy*, fazendo a *interface* entre as requisições dos clientes com o servidor, 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) qual(is) o(s) protocolo(s) esta(ão) sendo utilizado(s); (3) configuração (saber e replicar a atual configuração do servidor para a estratégia funcionar de acordo); (4) segurança (integridade, confidencialidade, disponibilidade e autenticidade da ferramenta em si); (5) robustez (garantir funcionamento ininterrupto em qualquer situação, para não prejudicar o servidor que está sendo protegido). Devido a esses fatores, faz-se necessário o uso de uma abordagem menos dependente e que possa ao mesmo tempo ser eficaz e fácil para o uso de administradores de redes. Para atingir esse objetivo pode-se aplicar o conceito de módulo, que funciona como um *mini-software* diretamente acoplado em um servidor [8].

O Apache HTTP Server Project, mais conhecido por servidor Apache, é o servidor *Web* mais popular e utilizado atualmente. De acordo com pesquisas da W3Tech e a BuiltWith, servidores Apache são usados por 55,9% e 51% entre todos os sites na *Internet* em Dezembro de 2015 [6] [7]. Por ser *OpenSource*, servidores Apache fornecem a liberdade e extensibilidade de seu funcionamento, a partir da implementação de módulos [8]. Os objetivo e contribuições deste trabalho são, portanto, desenvolver um módulo Apache, chamado *mod_seven*, para mitigar ataques ADoS. Os resultados dos experimentos mostraram que o *mod_seven* obteve resultados equivalentes e alinhados com os obtidos na versão *proxy* do SeVen, em alguns casos com resultados até melhores. Além de replicar os experimentos realizados na versão *proxy*, também foram realizados testes baseados em situações reais de ataques, ou seja, ataque com durações maiores, além de ataques no protocolo HTTPS (não suportado pela versão *proxy*). Também comparamos o módulo proposto com outros encontrados na literatura, relatando suas vantagens sobre os mesmos.

Outros módulos Apache existentes no mercado propõem-se a mitigar ataques ADoS [17] [16] [19]. O *mod_antiloris* tem como estratégia a contagem de conexões simultâneas abertas de um mesmo endereço IP. Quando essa contagem superar o valor configurado (o padrão é 10), o módulo rejeita todas as requisições provenientes daquele IP. O *mod_pacify_loris* [16] possui a mesma estratégia de defesa (mas utiliza 50 conexões por padrão), porém ainda implementa mais duas análises: contagem de GET HEADER enviados em uma mesma conexão e a taxa de requisições GET HEADER enviadas por segundo.

Túlio A. Pascoal, João H. G. Correa, Vivek Nigam e Iguatemi E. Fonseca, Universidade Federal da Paraíba, João Pessoa-PB, Brasil, E-mails: tuliopascoal@gmail.com, {vivek, iguatemi}@ci.ufpb.br.

Já o *mod_reqtimeout* [19], 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 chamada *RequestReadTimeout* em que há dois parâmetros configuráveis para cada um dos campos (cabeçalho e corpo) de uma requisição, que são: *timeout* e *minrate*. O seu diferencial quanto aos outros módulos é que avalia tais parâmetros em conjunto e a cada vez que pacotes de dados de uma requisição chegam ao servidor, o módulo renova suas janelas de *timeout*.

A Seção II apresenta a natureza, característica e funcionamento de ataques ADDoS. Na Seção III é apresentada a adaptação da estratégia SeVen como um módulo, bem como a arquitetura, funcionamento e interligação do módulo com o servidor. A Seção IV descreve os experimentos realizados, discute e compara os resultados obtidos. Finalmente, na Seção V são colocadas as conclusões e trabalhos futuros.

II. Ataques de Negação de Serviço na Camada de Aplicação

Dentre os ataques ADDoS destaca-se: o *Flooding* e o *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 honestos, porém, utilizando-se de vulnerabilidades encontradas nos protocolos para manter requisições em atendimento por tempo indeterminado [9], assim não necessitando de grandes recursos para geração dos ataques [10]. Ferramentas de geração desse tipo de ataque são facilmente encontradas na Internet e simples de usar [14] [12] [11] [13]. Por esses motivos, ataques *LowRate* são mais traiçoeiros e perigosos, dificultando detecção e proteção por parte da defesa. A seguir 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 [15], em um certo intervalo de tempo. 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 um momento, todos os recursos estarão sendo consumidos pelas requisições maliciosas, indisponibilizando a aplicação;
- **HTTP POST**: Este ataque envia requisições HTTP GET HEADER completas, realizando o *Handshake* com o servidor, porém seus dados do campo BODY, enviados pelo método HTTP POST são enviados 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.

III. O módulo Apache: *mod_seven*

Em [21] é descrito o *Apache HTTP Server* composto por um pequeno *core* e um conjunto de módulos. Dentre todos os módulos, existe um módulo "especial", chamado MPM (Módulo de Processamento Múltiplo), que serve como um otimizador para a comunicação entre o sistema operacional e a APR (Apache Portable Runtime, conjunto de bibliotecas de suporte para o servidor Apache que fornece um conjunto de APIs para comunicação com o sistema operacional). Módulos são a chave para a extensibilidade provida pelo Apache, com eles tem-se a liberdade de customizar e criar novos processos. A execução de módulos é baseada em ganchos, que funcionam como métodos que trabalham diretamente com *core* do Apache para melhor desempenho [8]. Há vários tipos de ganchos já definidos no Apache, mas novos ganchos também podem ser criados por usuários [21]. O *mod_seven* possui três ganchos: (1) o *ap_hook_post_config* para recolher informações do servidor, como tamanho do *pool* de atendimento, criação e alocação de *threads*; (2) o *ap_hook_create_request* para detecção de *dummy requests* [22], um *bug* encontrado no Apache durante a implementação desse módulo, e registro de requisições no filtro utilizado no módulo; (3) o *mod_seven_input_filter* que é onde ocorre praticamente toda a aplicação da estratégia. Quando uma requisição é registrada em um filtro, todo e qualquer dado enviado por aquela 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 uma requisição no *pool* de atendimento do Apache, chamado *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 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, o *worker_score* da mesma é adicionado ao *scoreboard* de acordo com a *thread* e número de processo alocados pelo servidor para atender aquela requisição;
- **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 aten-

der. Neste ponto, uma 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 não, uma requisição é escolhida de acordo com uma função de distribuição uniforme FP_2, para ser substituída do *pool* de atendimento (é daqui de onde vem a característica seletiva da estratégia). Após a escolha, o módulo resgata informação do *worker_score* escolhido, e adiciona a *flag* para deleção, assim, quando qualquer dado referente àquela requisição chegar ao servidor, a mesma será rejeitada automaticamente na Fase de Detecção.

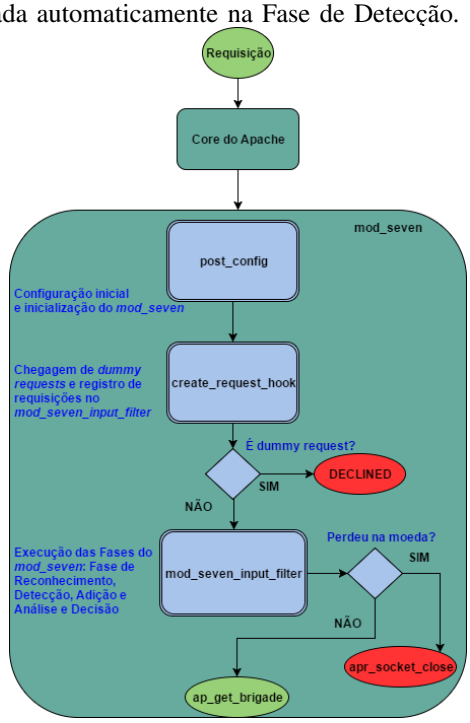


Fig. 1: Organização e ordem de fluxo de execução do *mod_seven*

Vale salientar que enquanto a aplicação não se encontra sobrecarregada, o módulo simplesmente executa as Fases de Reconhecimento e Adição. Na Figura 1 tem-se o fluxo de funcionamento e processos do *mod_seven* realizados pelos seus ganchos e métodos internos e na Figura 2 tem-se uma visão geral da divisão e execução de processos ocorridos nas distintas fases do *mod_seven_input_filter*.

IV. Experimentos e Resultados

A. Cenário e Configuração dos Experimentos

Os testes foram realizados usando três máquinas em dois diferentes *Campus* da Universidade Federal da Paraíba (UFPB). Duas máquinas situadas no *Campus* V gerando os tráfegos de clientes legítimos e atacantes (duas máquinas distintas). E uma máquina no *Campus* I hospedando a página Web padrão do Apache utilizando o MPM PREFORK, que é o MPM padrão para sistemas operacionais Unix [20]. 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 é de simular, com um maior grau de

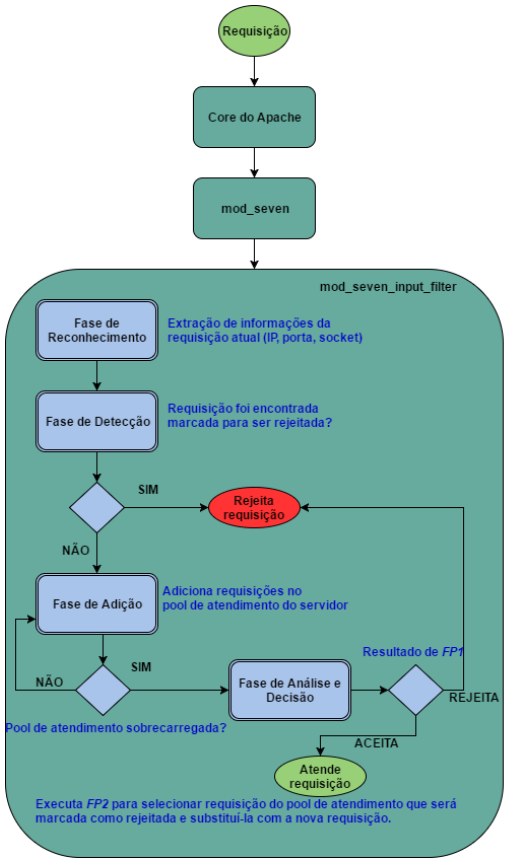


Fig. 2: Organização e ordem de fluxo das fases do *mod_seven_input_filter*

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 [18]) e para o tráfego atacante usamos duas ferramentas, *Slowloris* [12] e *Slowhttptest* [13]. À quesito de comparação de resultados, a configuração e cenários dos experimentos realizados nesse trabalho foram replicados de acordo com os realizados por [5]. As configurações dos testes levam ao cenário ideal para que o ataque obtenha sucesso e indisponibilize a página Web alvo. O servidor foi configurado da seguinte maneira:

- **Timeout**: Tempo, em segundos, que o servidor irá aguardar para continuar a receber dados de requisições em uma mesma conexão. Configurado com 40 segundos;
- **MaxRequestWorkers**: É o número de requisições simultâneas que serão atendidas pelo servidor. Ou seja, o limite de atendimento da aplicação *Web*. Configurado com 200;

Os experimentos realizados no *mod_seven*, nos outros módulos e com o Apache puro (sem nenhum módulo) tiveram as seguintes especificações:

- **Quantidade de Atacantes**: 250 atacantes para cada tipo de ataque: *Slowloris* e *HTTP POST*;
- **Tráfego atacante**: Conexões enviando requisições a cada 35 segundos. Aproximadamente 7,14 requisições por segundo;
- **Quantidade de clientes honestos**: 100 clientes gerados pelo *Siege*. Aproximadamente 10 requisições por segundo;

- **Tráfego de clientes honestos**: Requisições de cada cliente são enviadas em um intervalo de 0 a 3 segundos a fim de melhor simular um tráfego Web legítimo;
- **Protocolos**: HTTP (porta 80) e HTTPS (porta 443);
- **Duração**: três repetições para os testes de 5 minutos e uma para testes de 2 horas;
- **Tipo de Testes**: sem *mod_seven* e sem ataque; sem *mod_seven* e com ataque; com *mod_seven* e sem ataque; com { *mod_seven*, *mod_antiloris*, *mod_pacify_loris*, *mod_reqtimeout* } e com ataque.

Como métricas de desempenho, utilizou-se os seguintes parâmetros: i) Disponibilidade: Porcentagem dos clientes atendidos com sucesso; ii) TTS: Tempo médio de resposta para cada requisição; iii) Consumo de memória: Porcentagem do consumo médio de memória durante o teste; iv) Consumo de CPU: Porcentagem do consumo médio da CPU durante o teste.

B. Resultados e Discussão

Pela Tabela I percebe-se que o *mod_seven* não influencia na disponibilidade da aplicação em situações normais (sem ataque), percebe-se somente um aumento insignificante do consumo de CPU (de 7,8% para 8,7%) e de 0,02 segundos no TTS, devido aos processamentos adicionais realizados pelo módulo.

TABELA I: 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%	0,03 s	0,9%	8,7%
Slowloris	0,0%	∞	17,5%	0,0%	98,7%	0,07 s	18,2%	2,6%
HTTP POST	0,0%	∞	16,6%	0,0%	96,6%	0,03 s	13,5%	1,8%

Analisando os cenários com ataques, percebe-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 valores de TTS baixos.

Pela Figura 4 nota-se que, o consumo de memória quando utilizando o módulo não é elevado e no caso do ataque *HTTP POST* o consumo foi reduzido, devido a natureza do ataque que não envia tantas requisições como o *Slowloris*. Outro fator interessante é o consumo de CPU ser nulo (teve pico de 0,3%) quando sob ataque e sem módulo. Isso é mais uma prova da eficiência do ataque, pois, com a aplicação indisponibilizada, a mesma não processa mais nenhuma requisição, fazendo com que não haja mais consumo de CPU. Porém, há consumo de memória, uma vez que requisições continuam chegando e sendo "estocadas", esperando atendimento (ver Figura 3). Já no cenário com *mod_seven* (ver Figura 4) vê-se o consumo de recursos intercalados, mostrando que o servidor encontra-se ativo e em funcionamento.

Nos testes de 2 horas (HTTP e HTTPS), confirmou-se o bom desempenho do *mod_seven*. Verifica-se uma pequena queda na disponibilidade para o ataque *HTTP POST* (ver Tabela II (a)), uma vez que a taxa de requisições por segundo aumentou, 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.pl* mostrou-se incapaz de realizar o ataque, por isso utilizamos o *Slowhttptest*. Houve uma redução da disponibilidade e um pequeno aumento no TSS, 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 clientes nos testes de 2 horas).

TABELA II: Resultados dos experimentos de 2 horas do *mod_seven*

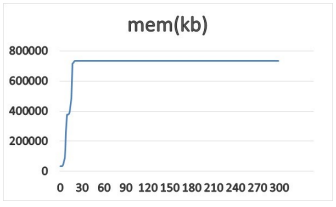
	Disponibilidade	TTS
Slowloris	97,5%	0,14s
HTTP POST	91,2%	0,05s

(a) Resultados no protocolo HTTP

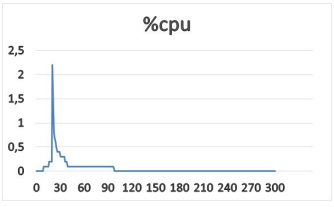
	Disponibilidade	TTS
Slowloris	91,6%	0,09s
HTTP POST	92,4%	0,07s

(b) Resultados no protocolo HTTPS

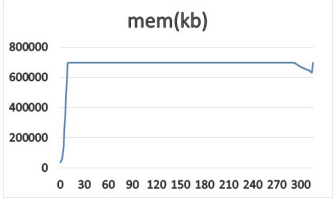
Em relação ao *mod_antiloris* e *mod_pacify_loris*, o *mod_seven* se mostra uma defesa mais sensata, pois não discrimina as requisições recebidas, todas possuem as mesmas chances de serem atendidas. Esses módulos utilizam uma estratégia discriminatória, trazendo algumas desvantagens. Além disso, esses módulos podem prejudicar usuários honestos que estejam no mesmo IP público de um atacante, dado que a defesa bloqueia requisições provenientes pelo IP público (o que ocorre em redes de grandes organizações, onde um único IP público pode representar mais de uma máquina na sua rede interna). Pela tática de taxa de cabeçalhos por segundo implementada pelo *mod_pacify_loris*, ele pode erroneamente rejeitar requisições de clientes honestos com conexões lentas.



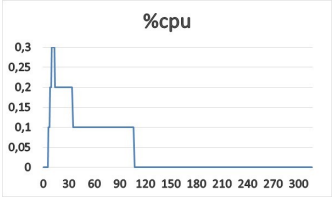
(a) Consumo de memória no ataque *Slowloris* e sem *mod_seven*



(b) Consumo de CPU no ataque *Slowloris* e sem *mod_seven*



(c) Consumo de memória no ataque *HTTP POST* e sem *mod_seven*

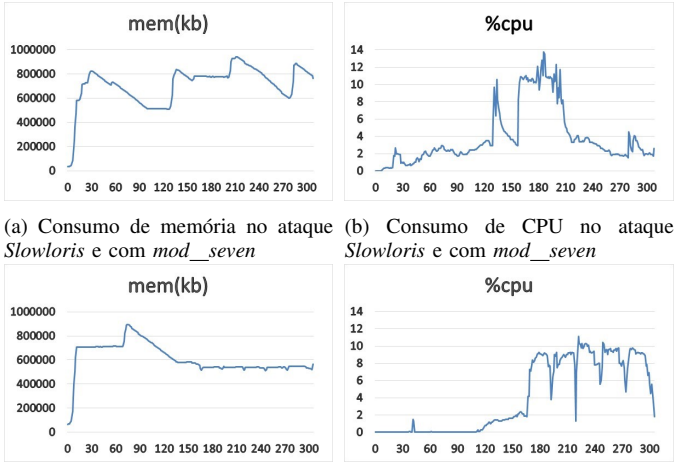


(d) Consumo de CPU no ataque *HTTP POST* e sem *mod_seven*

Fig. 3: Consumo de memória e CPU com ataque e sem *mod_seven*

Já o *mod_reqtimeout*, apesar de possuir uma estratégia mais

inteligente, baseada em *timeouts* e *minrate* renováveis, possui vulnerabilidade pois os atacantes podem utilizar estratégias de detecção desses 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.



(a) Consumo de memória no ataque *Slowloris* e com *mod_seven*

(b) Consumo de CPU no ataque *Slowloris* e com *mod_seven*

(c) Consumo de memória no ataque *HTTP POST* e com *mod_seven*

(d) Consumo de CPU no ataque *HTTP POST* e com *mod_seven*

Fig. 4: Consumo de memória e CPU com ataque e com *mod_seven*

Comparando com o SeVen *proxy*, os resultados do módulo mostra uma pequena melhora da disponibilidade da aplicação (aumento médio de 1%) nos experimentos com os mesmos cenários usados em [5], além de possibilitar a aplicação da estratégia no protocolo HTTPS e em qualquer outro, desde que suportado pelo Apache. Outra vantagem é que a estratégia pode ser melhor difundida dado a grande utilização de servidores Apache pela comunidade além da sua facilidade de utilização.

V. Conclusão e Trabalhos Futuros

Esse artigo apresentou uma solução para uns dos maiores problemas na Internet na atualidade, ataques DDoS. A defesa proposta é construída como um Módulo Apache, que é o servidor *Web* mais utilizado nos dias atuais. O *mod_seven* obteve resultados consistentes em vários cenários de testes e também quando comparado com o SeVen e com outros módulos Apache existentes na literatura. Além disso, o *mod_seven* trouxe melhorias e vantagens em relação ao SeVen *proxy*, conseguindo: mitigar ataques DoS no protocolo HTTPS; melhora de performance; modesto consumo de memória e CPU; utilização e implantação mais fácil e maior robustez da defesa. Como trabalho futuro, objetiva-se testar o *mod_seven* na mitigação de ataques DoS em outros protocolos suportados pelo Apache; testá-lo contra um ataque do tipo *LowRate* mais recente, chamado *SlowRead* [23], bem como contra ataques do tipo *Flooding*.

Agradecimentos

A CAPES, CNPq e RNP pelo apoio no desenvolvimento deste trabalho.

Referências

[1] Gu, Qijun and Liu, Peng, "Denial of service attacks.", *Handbook of Computer Networks: Distributed Networks, Network Planning, Control, Management, and New Trends and Applications*, 3: pp. 454–468, 2007.

[2] Chang, Rocky KC, "Defending against flooding-based distributed denial-of-service attacks: a tutorial", *Communications Magazine, IEEE*, 40(10), pp. 42–51, 2002

[3] Specht, Stephen M and Lee, Ruby B, "Distributed Denial of Service: Taxonomies of Attacks, Tools, and Countermeasures.", *ISCA, PDCS*, pp. 42–51, 2004

[4] Xie, Yi and Yu, Shun-Zheng, "Monitoring the application-layer DDoS attacks for popular websites", *Networking, IEEE/AcM Transactions on*, 17(1), pp. 15–25, 2009

[5] Dantas, Yuri Gil and Nigam, Vivek and Fonseca, Iguatemi E, "A Selective Defense for Application Layer DDoS Attacks", *Intelligence and Security Informatics Conference (JISIC), 2014 IEEE Joint*, 17(1), pp. 75–82, 2014

[6] W3tech, 2015. "*Usage of web servers for website*". `http://w3techs.com/technologies/overview/web_server/all`. Acessado em: 18 de Dezembro de 2015.

[7] BuiltWith, 2015. "*Web server usage statistics - Statistics for websites using web server technologies*". `http://trends.builtwith.com/web-server`. Acessado em: 18 de Dezembro de 2015.

[8] Apache, 2015. "*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 2015.

[9] Durcekova, Veronika and Schwartz, Ladislav and Shahmehri, Nahid, "Sophisticated denial of service attacks aimed at application layer", *ELEKTRO, 2012, IEEE*, 17(1), pp. 55–60, 2012

[10] Dantas, Yuri Gil, "Estratégias para Tratamento de Ataques de Negação de Serviço co na Camada de Aplicação em Redes IP", *Master Thesis in Portuguese*, 2015

[11] RUDY, 2013. "*R.U.D.Y - Are You Dead Yet*". `https://code.google.com/p/r-u-dead-yet/`. Acessado em: 07 de Fevereiro de 2015.

[12] Slowloris, 2013. "*Slowloris tool*". `http://ha.ckers.org/slowloris/`. Acessado em: 02 de Fevereiro de 2015.

[13] Slowhttptest, 2013. "*Slowhttptest tool*". `https://code.google.com/p/slowhttptest/`. Acessado em: 02 de Fevereiro de 2015.

[14] LOIC, 2013. "*A network stress testing application*". `https://github.com/NewEraCracker/LOIC/downloads`. Acessado em: 28 de Janeiro de 2015.

[15] OWASP, 2009. "*CRLF Injection*". `https://www.owasp.org/index.php/CRLF_Injection`. Acessado em: 27 de Março de 2015.

[16] 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.

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

[18] Siege, 2015. "*Linux man page: siege - An HTTP/HTTPS stress tester*". `http://linux.die.net/man/1/siege`. Acessado em: 3 de Novembro de 2015.

[19] Apache Reqtimeout, 2014. "*mod_reqtimeout*". `https://httpd.apache.org/docs/2.4/mod/mod_reqtimeout.html`. Acessado em: 11 de Agosto de 2015.

[20] Apache MPMs, 2014. "*Multi-Processing Modules (MPMs)*". `https://httpd.apache.org/docs/2.2/en/mpm.html`. Acessado em: 18 de Agosto de 2015.

[21] Kew, Nick, "The Apache Portable Runtime", in *The Apache modules book: application development with Apache*. Prentice-Hall, 2007.

[22] 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.

[23] Park, J and Iwai, K and Tanaka, H and Kurokawa, T, "Analysis of Slow Read DoS Attack and Countermeasures on Web servers", *International Journal of Cyber-Security and Digital Forensics (IJCSDF)*, 4(2), pp. 339–353, 2015