

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Gabriel Miranda Costa

**Implementação de um aplicativo para detecção
de *botnets* IoT em ambientes domésticos**

Uberlândia, Brasil

2018

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Gabriel Miranda Costa

**Implementação de um aplicativo para detecção de
botnets IoT em ambientes domésticos**

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: Dr. Rodrigo Sanches Miani

Universidade Federal de Uberlândia – UFU

Faculdade de Ciência da Computação

Bacharelado em Sistemas de Informação

Uberlândia, Brasil

2018

Gabriel Miranda Costa

Implementação de um aplicativo para detecção de *botnets* IoT em ambientes domésticos

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, 18 de dezembro de 2018:

Dr. Rodrigo Sanches Miani
Orientador

Dr. Paulo Henrique Ribeiro Gabriel

Dra. Maria Adriana Vidigal de Lima

Uberlândia, Brasil
2018

Dedico este trabalho a minha esposa, meus pais e a minha filha Mariana.

Agradecimentos

Agradeço a todos os professores que me ajudaram a obter conhecimento necessário para entender melhor a área da computação. Em especial agradeço ao meu orientador Dr. Rodrigo Sanches Miani e ao meu amigo Marcelo pela ajuda, dicas e excelentes trocas de ideias que tornaram possíveis este trabalho.

“Those who can imagine anything, can create the impossible”

Alan Turing

Resumo

Este trabalho descreve o desenvolvimento de um aplicativo Android voltado para usuários domésticos e suas redes Wi-Fi particulares, buscando identificar dispositivos IoT que apresentem falhas de segurança que podem ser exploradas pelo *botnet* Mirai, entre outros, e conscientizar o usuário dos riscos que ele pode ter em sua residência. Além disso, é feita uma análise geral sobre a questão da segurança destes dispositivos IoT que, devido a enormes ataques DDoS recentes que utilizaram falhas de segurança destes dispositivos, tem se tornado uma preocupação e um desafio para os especialistas em segurança da informação. O aplicativo desenvolvido é capaz de escanear redes Wi-Fi, encontrar dispositivos e verificar riscos de segurança (relacionados ao Mirai) presentes nos mesmos, auxiliando na conscientização sobre a segurança em dispositivos domésticos.

Palavras-chave: Mirai, IoT, DDoS, Internet, segurança da informação.

Lista de ilustrações

Figura 1 – Ataque Distribuído de Negação de Serviço.	16
Figura 2 – Funcionamento genérico de um <i>botnet</i>	19
Figura 3 – Logo da interface de comandos do <i>botnet</i> Mirai disponibilizado por Anna-senpai (2016)	22
Figura 4 – Funcionamento do <i>botnet</i> Mirai. Extraído de (PRADO, 2018)	24
Figura 5 – Diagrama de Classes do Aplicativo MiraiScanner	29
Figura 6 – Tela Sobre	31
Figura 7 – Telas do aplicativo durante o processo de escaneamento da rede Wi-Fi	32
Figura 8 – Telas de ameaças do dispositivo	32
Figura 9 – Conexão via protocolo <i>Telnet</i> utilizando apenas usuário e senha. Extraído de SSH.com (2018)	35
Figura 10 – Listagem de dispositivos durante realização de testes do aplicativo	37
Figura 11 – Dispositivo sem vulnerabilidades encontradas	38
Figura 12 – Verificação de acesso Telnet	39

Lista de abreviaturas e siglas

ACK	Flag de Reconhecimento
API	Application Programming Interface
ARP	Address Resolution Protocol
C&C	Command and Control
DDoS	Distributed Denial of Service
DNS	Domain Name System
DVR	Digital Video Recorder
DoS	Denial of Service
FBI	Federal Bureau of Investigation
Gbps	Gigabits por Segundo
GRE	Generic Routing Encapsulation
HTTP	Hupertext Transfer Protocol
IDE	Integrated Development Environment
IoT	Internet of Things
IP	Internet Protocol
MAC	Media Access Control
RAM	Random Access Memory
SSH	Secure Shell
STOMP	Simple Text Oriented Messaging Protocol
SYN	Flag de Sincronização
Tbps	Terabit por Segundo
TCP	Transmission Control Protocol
UDP	User Datagram Protocol

VSE Valve Source Engine

Wi-Fi Wireless Fidelity

XSS Cross-site Scripting

Sumário

1	INTRODUÇÃO	11
2	REVISÃO BIBLIOGRÁFICA	14
2.1	Segurança da Informação	14
2.1.1	Ataques Distribuídos de Negação de Serviço	15
2.2	Internet das Coisas	17
2.3	<i>Botnets</i>	18
2.4	Trabalhos Correlatos	20
3	MIRAI	22
3.1	Histórico	22
3.2	Componentes	24
3.2.1	Módulo <i>Bot</i>	25
3.2.2	C&C	27
3.2.3	<i>Loader</i>	28
4	APLICATIVO MIRAI SCANNER	29
4.1	Escaneamento da Rede Doméstica	30
4.2	Classes	33
4.3	Ambiente de Testes	36
5	CONCLUSÃO	40
	REFERÊNCIAS	42
	ANEXOS	45
	ANEXO A – TRECHOS DE CÓDIGO	46

1 Introdução

A Internet das coisas ou em inglês, *Internet of Things* (IoT) pode ser definida como a comunicação máquina a máquina via Internet que permite a troca de dados e informações entre diferentes objetos a fim de realizar determinadas tarefas. A base para funcionamento da IoT é composta por sensores e dispositivos, além de um sistema computacional capaz de analisar os dados recebidos e gerenciar ações destes objetos (SAP, 2016). Vários destes dispositivos tem a premissa de serem de baixo custo, a fim de serem vendidos em massa, como é o caso de várias câmeras de segurança. Para terem custo mais baixo, geralmente possuem poucas ou nenhuma medida de segurança embarcada, o que contribui para que estes dispositivos tenham várias vulnerabilidades e sejam o ambiente perfeito para que um usuário mal intencionado consiga fácil acesso e controle, utilizando-os para ataques maliciosos em grande escala, também conhecidos como ataques distribuídos de negação de serviço, em inglês *Distributed Denial of Service* (DDoS) *attacks*.

Em segurança da informação, o termo *bot*, abreviação de *robot*, refere-se a um dispositivo que realiza tarefas automatizadas sem necessariamente o conhecimento do dono do dispositivo. Quando vários destes dispositivos estão conectados via Internet e controlados por um mesmo usuário, dizemos que este conjunto de dispositivos forma uma rede chamada de *botnet* (MICROSOFT, 2012). Estes *botnets* podem ser utilizados por usuários mal intencionados para enviar *spam*, propagar vírus e realizar ataques a computadores. De acordo com Gubbi et al. (2013), em 2011 o número de dispositivos conectados na rede mundial de computadores superou o número de habitantes do planeta e, prevê-se que em 2020, devido ao grande aumento de dispositivos IoT, o número de dispositivos conectados deva chegar a 24 bilhões. Dado o número de dispositivos disponíveis, é extremamente necessário adotar medidas de segurança para evitar que estes dispositivos se tornem um exército de *botnets* controlados por usuários que queiram prejudicar pessoas e empresas.

Em 21 de outubro de 2016 a empresa Dyn, responsável por serviços de infraestrutura na Internet para empresas como Twitter, Amazon, Spotify e Netflix sofreu um grande ataque DDoS vindo de um *botnet* conhecido como Mirai. Este *botnet* é o mesmo que foi utilizado um mês antes em um ataque DDoS ao blog do pesquisador e consultor em segurança da informação, Brian Krebs. Este ataque ao blog de Krebs gerou um tráfego de 620 Gbps e outro ataque, também feito utilizando o Mirai, ao servidor web e de serviços de armazenamento na nuvem francês OVH gerou um tráfego superior a 1 Tbps (KOLIAS et al., 2017). Estes números impressionantes gerados nos ataques mostram como os *botnets* apresentam uma ameaça real e significativa à computação atual.

O que diferencia o Mirai de outros *botnets* conhecidos é o seu foco e competência

em infectar dispositivos IoT como roteadores, câmeras de segurança e DVRs. Estes dispositivos normalmente vêm com uma senha padrão do fabricante para acesso e que os seus donos geralmente não trocam ou o próprio dispositivo não vem equipado de fábrica com capacidade para que esta senha seja alterada para uma senha mais segura. Deste modo, o Mirai utiliza uma tabela de credenciais padrão e, por meio de uma técnica conhecida como força bruta, consegue acesso ao dispositivo IoT, incluindo-o em seu *botnet*. Estes dispositivos, tem capacidade de processamento muito baixa mas, como estão conectados na Internet em grande quantidade, geram este poder impressionante visto nos ataques atribuídos ao Mirai.

O presente trabalho visa desenvolver um aplicativo para *smartphones* com sistema operacional Android capaz de detectar dispositivos domésticos, conectados na rede *Wi-Fi* do usuário, vulneráveis ou infectados pelo Mirai. Uma vez detectados, o usuário será informado das vulnerabilidades e será oferecida uma solução para tentar contornar a vulnerabilidade do dispositivo em risco. Atualmente, a linha de pesquisa de detecção e prevenção de falhas de segurança exploradas em dispositivos IoT por *botnets* tem crescido bastante. Nesta linha, temos trabalhos como o AntibIoTic (De Donno et al., 2017), que também é um *botnet* que busca dispositivos IoT conectados a Internet que possuam falhas de segurança mas que, ao invés de usar o dispositivo maliciosamente, tenta resolver o problema atualizando o *firmware* do dispositivo, se for possível, e instalando uma versão não maliciosa de um *bot* que impedirá a infecção do dispositivo por um *bot* malicioso.

Também nesta linha, a Kaspersky anunciou em seu blog em 20 de abril de 2018 que está em fase beta de testes de um aplicativo voltado a segurança de dispositivos IoT (PEREKALIN, 2018). A diferença desses trabalhos e aplicativos para este trabalho é o desenvolvimento de um aplicativo voltado ao usuário doméstico e onde o mesmo participará ativamente da solução e, conseqüentemente, se tornará mais consciente da seriedade do problema.

Para avaliar a eficiência do aplicativo em escanear a rede doméstica e descobrir vulnerabilidades que possam ser exploradas ou criadas pelo Mirai, foi criado um ambiente doméstico, sem a presença do Mirai, onde foram simuladas falhas de segurança utilizando computadores pessoais para isto. A montagem deste ambiente e equipamentos utilizados será descrita na seção 4.3.

Este trabalho encontra-se estruturado na forma de capítulos da seguinte forma:

- O capítulo 2 desenvolve a base teórica de sustentação do trabalho assim como faz uma correlação entre trabalhos existentes para contextualizar o leitor.
- No capítulo 3 é feita uma análise mais detalhada do *Mirai*, a fim de elucidar o leitor quanto ao funcionamento do *botnet*, quais vulnerabilidades são exploradas para a

infecção, os tipos de ataques implementados por ele e as consequências que esses ataques podem trazer para o dia a dia do usuário comum.

- No capítulo 4 é mostrado o desenvolvimento do aplicativo, as técnicas utilizadas para solucionar os problemas encontrados, a montagem e detalhamento do ambiente de testes e como foram conduzidos os testes finais visando a publicação do aplicativo.
- E no capítulo 5 é feita a conclusão do trabalho com as considerações finais e apresentação de perspectivas de trabalhos futuros que possam complementar ou continuar o trabalho que foi desenvolvido.

2 Revisão Bibliográfica

Este capítulo apresenta um levantamento bibliográfico sobre segurança da informação, botnets, IoT e outros conceitos fundamentais para contextualizar e fundamentar o trabalho.

2.1 Segurança da Informação

No mundo interconectado que vivemos, a segurança da informação é algo tão importante quanto a segurança patrimonial. Garantir a estabilidade, resiliência e segurança da Internet devem ser os principais objetivos dos envolvidos na gerência da Internet pois, os usuários esperam que suas informações sejam mantidas em sigilo, sejam confiáveis e estejam disponíveis quando e onde forem necessárias ([HOEPERS; STEDING-JESSEN, 2014](#)). Tais expectativas são a base proposta por [Spanceski \(2004\)](#) como os princípios da segurança da informação, mostrados a seguir:

- Autenticidade, está relacionada com a garantia que o solicitante de uma informação realmente é quem alega ser.
- Confidencialidade, está relacionada com a garantia que a informação solicitada é realmente disponibilizada somente para quem tem permissão de acesso a ela.
- Disponibilidade, está relacionada com a acessibilidade da informação sempre que necessário, garantindo que o usuário esteja protegido contra a degradação e indisponibilidade do serviço. Um desafio dos sistemas *web* atuais é conseguir manter a disponibilidade evitando os ataques DDoS dos *botnets*.
- Integridade, está relacionada com a garantia que a informação não seja modificada ou apagada sem a autorização do proprietário da mesma.

Ainda segundo [Hoepers e Steding-Jessen \(2014\)](#), a definição de risco, em sistemas de informação, é o potencial de dano que um sistema pode sofrer quando agentes externos exploram vulnerabilidades. Vulnerabilidade é qualquer fraqueza do sistema que pode ser explorada a fim de comprometer a segurança do mesmo. A exploração de vulnerabilidades em sistemas computacionais e em aspectos humanos (envio de mensagens eletrônicas a fim de comprometer as máquinas dos receptores ao se clicar em *links* maliciosos recebidos) são alguns dos fatores que podem ser explorados a fim de comprometer os princípios da segurança da informação ([NAKAMURA; GEUS, 2007](#)). Um exemplo de ataque que compromete a disponibilidade de um sistema computacional, são os ataques distribuídos

de negação de serviço (DDoS), que impedem o usuário de acessar determinado serviço ou informação.

2.1.1 Ataques Distribuídos de Negação de Serviço

Em ataques DDoS o atacante utiliza várias máquinas infectadas para realizar um ataque, geralmente visando servidores *Web*, de maior escala (CANALTECH, 2014). Em um ataque DDoS o servidor pode não responder à tentativa de acesso a algum de seus serviços por usuários legítimos pois, um usuário mal intencionado promove acessos simultâneos e em larga escala a determinada página *Web*, aplicação *Web* ou até mesmo a serviços interligados dependentes de um servidor *Web* de maneira que o servidor não comporte o volume de tráfego de dados solicitado durante o ataque, levando a interrupção do funcionamento do mesmo.

Para se entender os ataques DDoS, primeiro deve-se entender os ataques de negação de serviço (DoS). Este tipo de ataque existe desde a década de 1970, sendo que em meados de 1988 passou a ser levado mais a sério quando se estendeu em larga escala pela Internet. Um dos primeiros *worms* da história, foi o *Morris Worm* que podia infectar uma mesma máquina múltiplas vezes, criando um novo processo na máquina a cada reinfecção, reduzindo drasticamente sua velocidade até o ponto em que se tornava impossível do usuário obter resposta da máquina infectada, caracterizando um ataque de negação de serviço (PAGE, 1988).

Já os DDoS, utilizam-se de vários dispositivos interconectados, geralmente via Internet, para realizar ataques DoS a um determinado serviço *web*, na tentativa de esgotar os recursos do servidor. Na figura 1 abaixo é mostrada uma estrutura comum de um ataque DDoS. Neste tipo de ataque, um usuário malicioso tem o controle de servidores mestres que enviam comandos para os dispositivos infectados, chamados de *bots*, que realizam ataques em determinada vítima. O conjunto de *bots* controlados por determinado usuário é chamado de *botnet*.

Os ataques de esgotamento de recurso visam sobrecarregar o servidor, prejudicando assim o acesso a seus serviços. Dentro desta modalidade de ataque existem os ataques por inundação que consistem em enviar o máximo possível de pacotes de tráfego a fim do servidor não conseguir responder todas as solicitações, gerando lentidão ou interrupção por completo do servidor (MICROSOFT, 2009). Outro ataque de esgotamento de banda é o ataque por amplificação, que consiste em forjar endereços reais de IP da vítima para enviar requisições, em grandes quantidades, para computadores que responderão. Nas respostas, o atacante utiliza técnicas de reflexão e endereços de *broadcast* para ampliar as cargas dos pacotes, causando dano ainda maior à vítima que terá sua rede sobrecarregada e não conseguirá atender a demanda devido a alta carga criada pelo atacante (SAB; FERREIRA; ROZENDO, 2013).

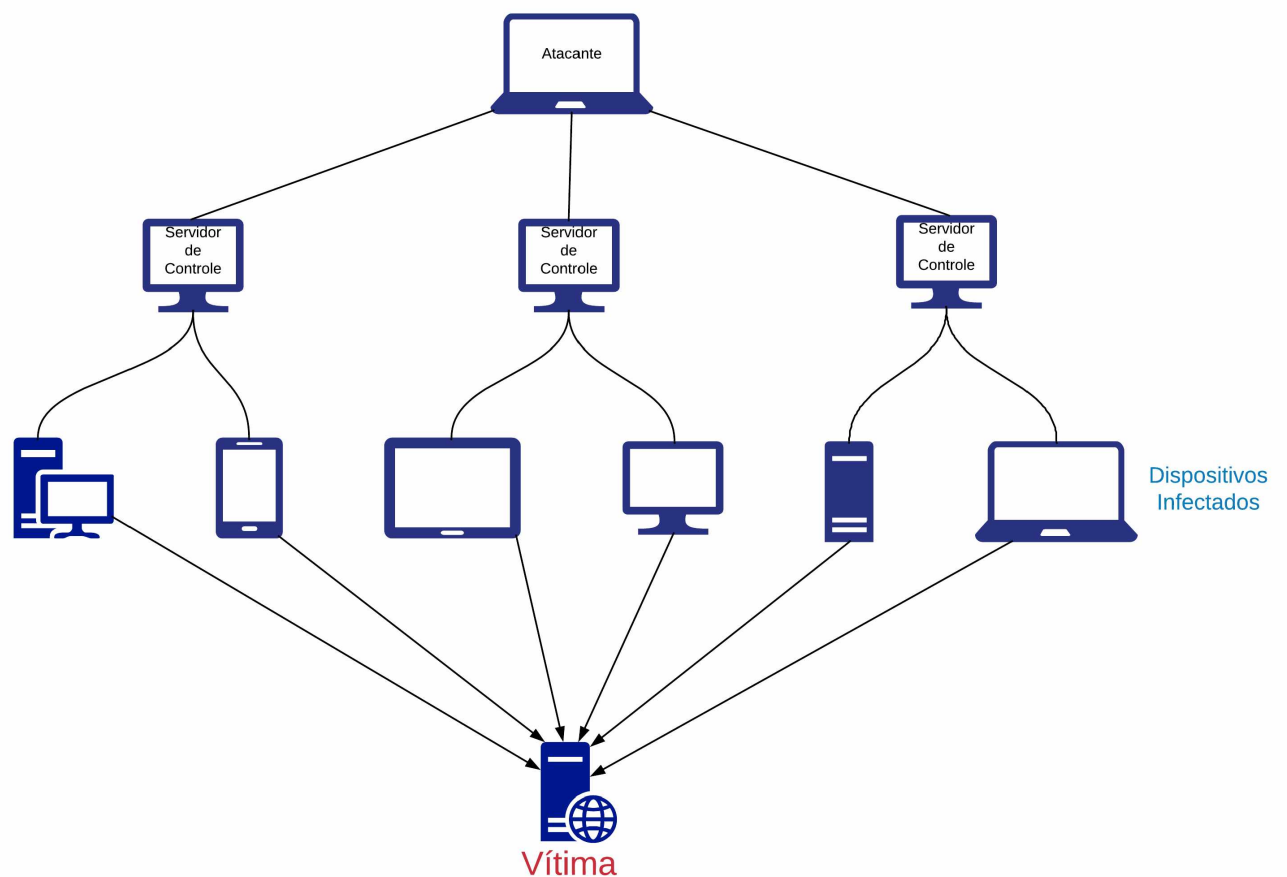


Figura 1 Ataque Distribuído de Negação de Serviço.

Ainda dentro deste tipo de ataque, pode-se destacar os que utilizam a má formação de pacotes de tráfego, ou seja, utilizando dados de cabeçalho obrigatórios confusos e preenchendo dados opcionais a fim de sobrecarregar o sistema. Também existem os ataques que exploram os protocolos utilizados pela vítima, que utiliza o padrão de arquitetura do protocolo contra a vítima, por exemplo, utilizando o ataque *SYN Flood*. Este ataque explora o processo de estabelecimento de conexão em três vias, onde o atacante envia vários pacotes contendo *SYN flags* ao servidor que responde com *SYN-ACK* e salva as informações de estado para estas tentativas de conexões. Como o servidor não recebe o pacote *ACK* do atacante, ele acaba alocando muitos recursos inúteis para conexões que nunca irão acontecer (SAB; FERREIRA; ROZENDO, 2013).

Cada vez mais os atacantes têm obtido êxito em infectar um número maior de dispositivos conectados à Internet, criando *botnets* maiores a fim de maximizar os ataques DDoS, gerando tráfegos de rede impressionantes e nunca antes vistos (KOLIAS et al., 2017).

2.2 Internet das Coisas

A introdução de sensores e dispositivos conectados à Internet com a finalidade de fornecer dados e realizar ações, a chamada Internet das Coisas (IoT), mudou o panorama da Internet, criando um novo leque de oportunidades e desafios (RIEGEL, 2017). Com estes dispositivos conectados cada vez mais comuns no nosso dia a dia, podemos coletar dados em tempo real e utilizá-los para melhorar a qualidade de vida da população. Entre exemplos de IOTs estão sensores e dispositivos para monitoramento agrícola, da saúde humana e para criação de cidades inteligentes. Estima-se que a indústria de IOTs esteja na casa de US\$470 bilhões e com aproximadamente 20.8 bilhões de dispositivos conectados até 2020 (MEULEN, 2015).

Com o aumento rápido do número destes dispositivos conectados à Internet, é necessário que as empresas fabricantes tomem cuidado com os protocolos de segurança destes dispositivos. O usuário comum, que cada vez mais compra estes aparelhos, precisa ficar atento às questões de segurança, verificando se o fabricante do dispositivo é confiável e se aquele dispositivo é realmente necessário. Todo dispositivo conectado à Internet, é uma possível fonte de informação sobre o usuário e seus hábitos, que pode levar a ataques como os de *phishing* (CIPHER, 2018).

Porém, a preocupação dos fabricantes em manter baixo custo para popularizar estes dispositivos rapidamente, faz com que o *hardware* e *software* sejam reduzidos ao máximo, sem dar muita importância para protocolos de segurança. Utilizando estes *hardwares* mínimos e sem a preocupação dos *softwares* em obedecer os protocolos de segurança, nem sempre estes dispositivos atendem às premissas básicas de segurança, como o uso de senhas de administrador fortes, fazendo que eles sejam fáceis de se explorar.

A Hewlett Packard (HP) (2014) realizou um estudo onde 70% dos dispositivos IoT apresentam alguma vulnerabilidade de segurança e, portanto, estão sujeitos a ataques. Para chegar a estes resultados, foram testados 10 dos dispositivos IoT mais comuns no mercado. Para isso foram utilizadas técnicas padrão de descoberta de vulnerabilidades, associadas com testes manuais e automatizados, baseados nas diretrizes do projeto OWASP (2014).

Os problemas de segurança mais comuns encontrados nos dispositivos e em seus serviços na nuvem, são listados a seguir.

- Problemas de privacidade: 80% dos dispositivos armazenavam vários dados pessoais dos consumidores como nome, e-mail, endereço, data de nascimento, número de cartão de crédito e informações sobre a saúde do usuário. Ainda no estudo, 90% dos dispositivos armazenavam pelo menos uma dessas informações;
- Protocolos de autorização insuficientes: 80% dos dispositivos não pediam uma senha

segura ao usuário, deixando que o mesmo utilizasse senha curtas e padrão, como por exemplo "1234";

- Ausência de criptografia: 70% dos dispositivos não possuíam nenhum tipo de criptografia dos dados trocados entre a Internet e a rede local. A criptografia destes dados é fundamental uma vez que estes dispositivos enviam dados sensíveis dos usuários por estes canais;
- Interface *web* insegura: 60% dos dispositivos apresentam problemas de segurança como XSS persistente, controle de sessão ruim, credenciais padrão fracas e credenciais sendo transmitidas sem criptografia;
- Proteção de *software* inadequada: 60% dos dispositivos não usam criptografia para realizar o *download* de atualizações de *software*. Desta maneira os *downloads* poderiam ser interceptados, extraídos e montados em outros sistemas para serem visualizados ou modificados.

2.3 Botnets

Botnets são redes de computadores controladas por um usuário, chamado de *botmaster*, com o objetivo de usar estes computadores para alguma tarefa, positiva ou negativa, como por exemplo manter *websites* funcionando, realizar ataques DDoS, mandar *spam*, roubar informações ou espionar (SYMANTEC, 2016). A infecção maliciosa destes computadores normalmente se dá através de *worms*, que conecta estes dispositivos a um servidor de comando e controle (C&C). Quando os *botnets* começaram a aparecer no mundo, o comportamento infeccioso do *malware* causava dano visível quase que imediato ao dispositivo infectado. Já nos *botnets* modernos, os dispositivos infectados podem passar despercebidos por um longo período de tempo antes de realizar alguma ação danosa. O motivo desta mudança se dá ao fato que o *botmaster* deseja manter um número expressivo de dispositivos infectados, os *bots*, disponíveis em seu *botnet*. Ao manter os donos dos dispositivos infectados sem notarem a infecção, o *botmaster* consegue que o *bot* não seja sanitizado, fazendo que estes persistam no *botnet* e possam ser utilizados de maneira discreta para cometer seus crimes (RIEGEL, 2017).

O funcionamento geral de um *botnet* envolve três componentes básicos: o *botmaster*, o C&C e os *bots*. O *botmaster* é quem desenvolve e controla o *botnet*, definindo como será feita a infecção, a forma de distribuir o *worm* para infectar mais dispositivos e qual ação o *botnet* realizará. Como o próprio nome diz, o servidor de comando e controle tem como objetivo enviar comandos e controlar o *botnet*, é através desta interface que o *botmaster* interage com os seus *bots*. E por fim, os *bots* são os dispositivos que foram infectados e agora estão sob o controle do *botmaster*, podendo ser utilizados para lançar ataques DoS,

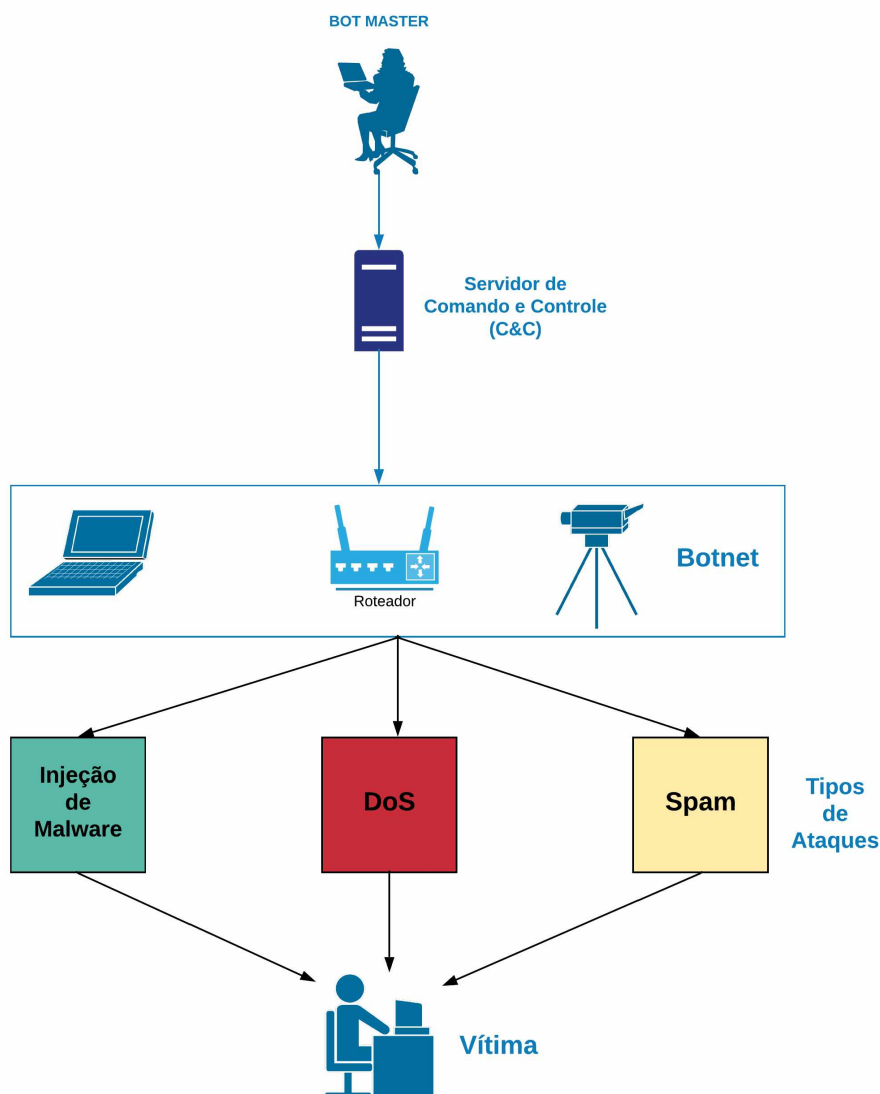


Figura 2 – Funcionamento genérico de um *botnet*.

spam e infectar outros dispositivos, entre outros (VERTON, 2014). A Figura 2 mostra este funcionamento.

Dentre os *botnets* modernos, destaca-se o Mirai. O Mirai foi o centro de vários estudos acadêmicos e investigativos desde que foi identificado em 2016. O motivo deste destaque se deve ao tráfego gerado durante seus ataques, considerados os maiores registrados em ataques DDoS até então, e a liberação de seu código fonte na Internet. Com o interesse em se estudar um *botnet* capaz de gerar tamanho volume de dados e ainda tendo-se disponível seu código fonte, o Mirai ganhou notoriedade na comunidade de Segurança da Informação mundial. Uma análise mais profunda do Mirai é feita no Capítulo 3.

2.4 Trabalhos Correlatos

Esta seção descreve trabalhos que estão sendo realizados em torno da detecção, prevenção e eliminação do *malware* Mirai.

O trabalho proposto por [De Donno et al. \(2017\)](#) visa construir um *white worm* chamado AntibIoTic que tem como principal objetivo oferecer uma proteção paliativa para impedir que dispositivos IoT sirvam de *bots* para *botnets* em ataques DDoS. Para isto, esta solução busca competir com os *malwares* pelas mesmas vulnerabilidades encontradas nos dispositivos IoT, e infectá-los com uma versão protetiva pois, quando o AntibIoTic obtem o controle dos dispositivos, ele instala um bot que tenta protegê-lo contra *malwares*. Para tal feito, o *white worm* escaneia a Internet e infecta dispositivos vulneráveis. Depois coleta dados dos dispositivos, sanitiza-os e notifica os donos, tentando acabar com as vulnerabilidades. A ferramenta também tem resistência ao *reboot* do dispositivo, com uma estrutura parecida com a do Mirai onde um servidor remoto verifica se o dispositivo continua ativo e enviando resposta ou se foi reiniciado.

De acordo com os autores, o ponto forte do AntibIoTic é o fato dele agir continuamente na Internet, sem a necessidade que os donos dos dispositivos tenham que acessar algum *website* ou rodar algum programa para descobrir que possui problemas de segurança em sua rede. Porém, a solução apresenta um ponto crítico que é tentar resolver os problemas de vulnerabilidades atualizando a *firmware* do dispositivo automaticamente. Ao fazer isto, caso o dono desligue o aparelho no meio da operação, danos irreparáveis podem acontecer. Outro ponto problemático é o fato de ter que se manter uma estrutura relativamente grande para o funcionamento seguro do *white worm* pois, caso o controle do servidor seja comprometido, um usuário mal intencionado já tem a base de um *botnet* pronta para ser modificada para realizar ataques.

Outro trabalho interessante que visa a proteção de dispositivos domésticos IoT é o desenvolvido pela Kaspersky ([PEREKALIN, 2018](#)). Apesar de ser desenvolvido por uma grande fabricante de antivírus, o aplicativo ainda está em fase de testes e só está disponível para os usuários do aplicativo de segurança completo da Kaspersky. Sendo assim, ainda não foram divulgadas informações sobre se o aplicativo será gratuito ou se será fornecido apenas para os clientes da empresa nem sobre suas principais funcionalidades.

Em geral já existem vários trabalhos que identificam e analisam o Mirai, como os trabalhos de [Sinanovic e Mrdovic \(2017\)](#), [Kolias et al. \(2017\)](#) e [Riegel \(2017\)](#). Eles têm em comum a análise do código do Mirai e da busca de um método de detectá-lo, seja através da análise do tráfego de rede gerado ou buscando pela vulnerabilidades dos dispositivos e, também fazem relações entre o Mirai e outros *malwares* semelhantes. Porém, além do aplicativo sendo desenvolvido pela Kaspersky, não foram encontrados trabalhos buscando realizar a identificação, prevenção e sanitização do Mirai utilizando apenas um aplicativo

para dispositivos móveis. O apelo para o desenvolvimento de tal ferramenta é grande visto que smartphones estão presentes em grande parte das redes domésticas e o uso de aplicativos para dispositivos móveis já é uma tendência mundial ([STATISTA, 2016](#)).

3 Mirai

Este capítulo apresenta uma visão geral do funcionamento do *botnet* Mirai. São discutidos os módulos da implementação do Mirai e os tipos de ataques que estão implementados no código fonte que foi disponibilizado no GitHub pelo usuário [Anna-senpai \(2016\)](#), posteriormente identificado como Paras Jha ([KREBS, 2018](#)). A seção 3.1 trata de como foi identificado o Mirai, seus primeiros ataques e a liberação de seu código fonte no GitHub. Logo depois, na seção 3.2, é feita uma análise técnica sobre o funcionamento do Mirai e seus componentes.



Figura 3 – Logo da interface de comandos do *botnet* Mirai disponibilizado por [Anna-senpai \(2016\)](#)

3.1 Histórico

Um dos primeiros ataques atribuídos ao Mirai foi realizado no segundo semestre de 2016, tendo como alvo o *blog* do jornalista Brian Krebs, que trata de assuntos relacionados a segurança da informação. Naquele ataque, o que mais chamou atenção foi o volume de dados gerado durante o ataque, na casa de 665 Gbps, considerado um dos maiores ataque DDoS já registrados até então. Ao invés de utilizar poucas máquinas infectadas e amplificar os números dos ataques com técnicas tradicionais, o Mirai utilizava uma enorme quantidade de máquinas infectadas, os *bots*.

Estes *bots* eram compostos em sua maioria de dispositivos da chamada Internet das Coisas (IoT), como roteadores, câmeras de segurança e gravadores de vídeo, cujo objetivo é bem específico e seu poder computacional é bem limitado (ZEIFMAN; BEKERMAN; HERZBERG, 2016). Essa limitação e a vontade dos fabricantes de produzir estes equipamentos cada vez mais baratos para serem mais competitivos neste enorme mercado, fez com que estes dispositivos possuísem pouca ou nenhuma medida de segurança im-

plementada de fábrica. Até mesmo as senhas padrão, já embutidas nos dispositivos, não podem ser alteradas devido a falta de *software* para permitir esta mudança (SINANOVIC; MRDOVIC, 2017).

Essa falta de segurança associada com a vasta informação de credenciais de acesso *root* dos aparelhos, manuais e guias de uso disponíveis na Internet (RIEGEL, 2017), fez com que o Mirai obtivesse acesso a um número tão grande de dispositivos que o fato do poder computacional destes equipamentos ser baixo não atrapalhava o poder final do *botnet*.

Após o ataque ao *blog* do jornalista Krebs, o Mirai voltou a fazer vítimas em um novo ataque de proporções ainda maiores. No dia 21 de outubro de 2016, o alvo do Mirai foi a empresa de serviços de infraestrutura de redes, como o DNS, Dyn.

O ataque, que foi baseado em requisições para traduções de nomes de domínio através dos protocolos TCP e UDP, gerou um tráfego de rede superior a 1 Tbps, sendo quase o dobro do tráfego gerado durante o ataque ao *blog* do Krebs. A Dyn tem como clientes grandes empresas como Twitter, Amazon, Spotify e Netflix, o que gerou instabilidades percebidas mundialmente.

Apesar do dano causado pelo Mirai nos servidores da Dyn, o objetivo inicial de sua criação era obter vantagens nos servidores do jogo *Minecraft* (GRAFF, 2016). Segundo Graff (2016), os desenvolvedores do Mirai queriam derrubar serviços concorrentes de hospedagem para servidores do jogo, em um mercado que movimentava até US\$100 mil por mês, porém quando viram o potencial que o Mirai possuía, resolveram ampliar os ramos de abordagens para seu uso, o alugando para terceiros realizarem ataques DDoS, como o que foi feito na Dyn.

Após estes ataques e a liberação do código fonte no fórum Hackforums (KREBS, 2016), ficou em evidência o enorme problema de segurança dos dispositivos IoT, uma vez que a técnica utilizada para a infecção é a chamada força bruta, ou seja, o aplicativo malicioso tem uma tabela com usuários e senhas mais utilizados implementada em seu código fonte e tenta o acesso aos dispositivos por tentativa e erro. Quando alguma das credenciais tentadas obtém sucesso na tentativa de acesso, o *malware* infecta o dispositivo e o torna parte de seu *botnet*. Sendo assim, medidas simples que já são implementadas por algumas empresas em outros dispositivos, como a geração de usuários e senhas randômicos para cada aparelho, evitariam novos ataques como esse. Como consequência destes ataques, uma das medidas tomadas foi o *recall* de vários dispositivos da *XiongMai Thechnologies*, uma das empresas cujos dispositivos foram amplamente infectados (SYMANTEC, 2016).

Atualmente já se sabe que o Mirai teve três responsáveis: Paras Jha, Dalton Norman e Josiah White, todos entre 21 e 22 anos de idade. Eles foram presos no final de 2017, se declararam culpados de suas acusações e foram sentenciados a pagar uma multa

de US\$8.6 milhões além de prestar 2.500 horas de serviços comunitários. Apesar de se declararem culpados, a sentença não os enviou para a prisão devido a uma colaboração técnica com o FBI para a resolução e prevenção de ataques massivos como os que foram realizados pelo Mirai (BING, 2018).

3.2 Componentes

O Mirai possui três componentes: *bot*, servidor de comando e controle (C&C) e o *loader*. Eles foram escritos utilizando as linguagens de programação C e Go e, cada um desses componentes tem funções específicas durante a infecção e utilização do Mirai por seu *botmaster*. O funcionamento do Mirai pode ser visto na Figura 4 e os seus componentes serão detalhados nas subseções a seguir.

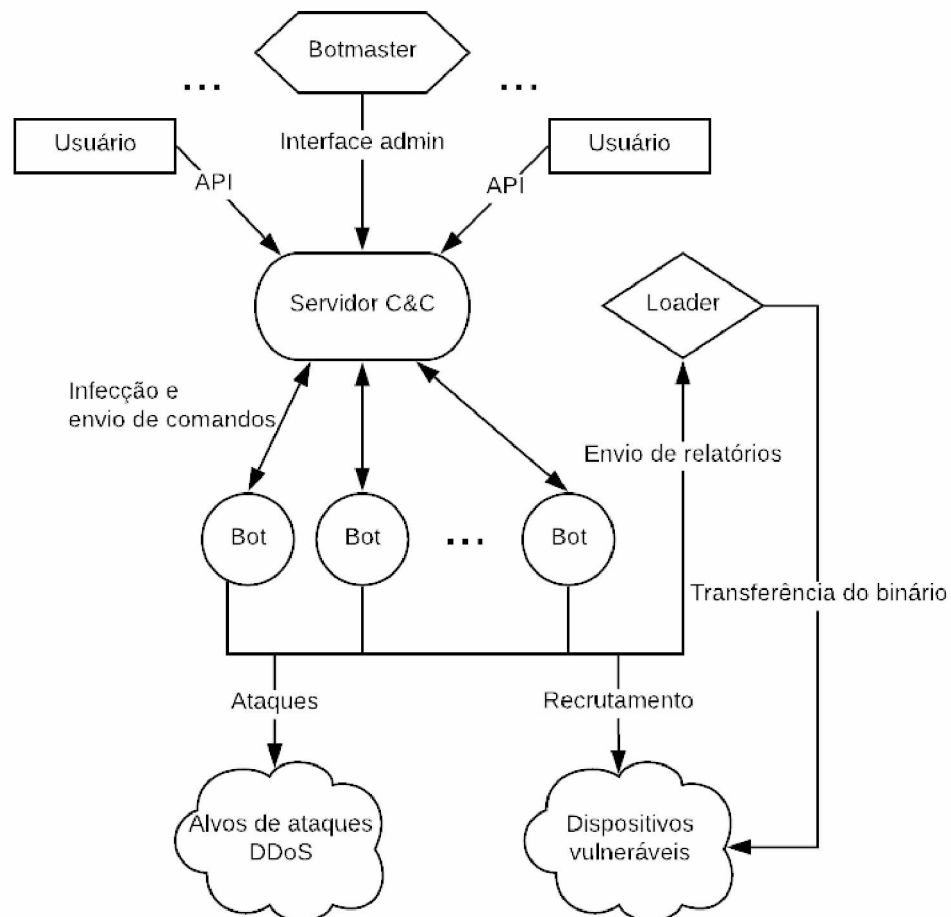


Figura 4 – Funcionamento do *botnet* Mirai. Extraído de (PRADO, 2018)

3.2.1 Módulo *Bot*

No contexto do Mirai, chamamos de Módulo *Bot* o componente que fica instalado diretamente no *bot*. Ele tem funções implementadas para encerrar processos de *botnets* concorrentes, escanear outros dispositivos e realizar ataques. Além disso, o Módulo *Bot* implementa as funções de apagar o executável baixado durante a infecção, ficando somente na memória RAM do dispositivo. Também possui a função de prevenção contra a reinicialização do aparelho infectado.

Uma vez que o Módulo *Bot* apaga seu executável e fica apenas na memória RAM, é essencial que este dispositivo não seja reinicializado, pois assim, o Módulo *Bot* seria apagado da memória e uma nova infecção teria que ser feita, perdendo tempo e recursos no processo. Para fazer esta prevenção o Módulo *Bot* implementa uma função que desabilita um componente de *hardware* chamado de *watchdog timer*. Esse componente, que vem instalado de fábrica em cada dispositivo, tem como função reiniciar o aparelho se alguma erro ocorrer durante o seu uso, para tentar corrigir o erro e voltar a utilização normal do mesmo.

Com o Módulo *Bot* já sendo executado no dispositivo, ele procura por outros *malwares* conhecidos que também infectam dispositivos IoTs. Um exemplo é o Kami, que também é conhecido pelo nome "Anime". É interessante este comportamento pois o Kami é um *malware* que tem características semelhantes ao Mirai. Ele também utiliza uma lista de pelo menos sessenta e duas credenciais salvas em seu código (SYMANTEC, 2016) e tenta a infecção pelo método de força bruta, via protocolo Telnet. Este comportamento tão similar mostra que essa falha de segurança é antiga e já explorada além de também mostrar como o mercado de ataques DDoS é competitivo a ponto de diferentes *malwares* brigarem ativamente pelo mesmo espaço (RIEGEL, 2017), reservando uma parte importante de suas implementações somente para tentar eliminar o concorrente de seus dispositivos infectados. Além de procurar e eliminar o Kami, o Mirai também procura e finaliza todos os serviços que estejam rodando nas portas 22 (SSH), 23 (Telnet) e 80 (HTTP), além disso ele previne que estas aplicações se reiniciem e voltem a rodar nestas portas. Este procedimento é feito pois outros *malwares* exploram falhas em processos que rodam nestas portas para infectar os dispositivos IoTs.

A função de escaneamento é utilizada pelo Módulo *Bot* para encontrar possíveis novos dispositivos a serem infectados pelo Mirai. Para isto, são gerados IPs aleatórios, regulados com uma *blacklist* de IPs que não devem ser escaneados, onde o Mirai realizará *pings* nas portas 23 e 2323 (Telnet). Para otimizar o processo, este escaneamento é feito usando uma abordagem distribuída, utilizando várias *threads* para os *pings* e deixando a parte de carregar o binário no dispositivo vulnerável encontrado para o módulo *loader*, que é mais poderoso, tornando o desempenho do escaneamento do Mirai muito superior a maioria de *malwares* IoTs conhecidos (RIEGEL, 2017). Quando o escaner encontra os

dispositivos vulneráveis ele cria uma lista com o IP, a porta e credenciais utilizadas para acessar cada dispositivo. A troca de mensagens entre o Módulo *Bot* e o Mirai é feita através da porta 48101 pela ferramenta **ScanListen**, que fica aguardando conexões TCP a fim de transmitir a listagem de dispositivos encontrados pelo Módulo *Bot*.

Como o Mirai tem como objetivo realizar ataques DDoS, seu Módulo *Bot* disponibiliza vários tipos de ataques DoS prontos para serem usados pelo *botmaster*. Para utilizá-los, o servidor C&C envia para o Módulo *Bot* qual tipo, duração e o alvo do ataque a ser realizado. Apesar de ter diversos tipos de ataques DoS implementados, não é isso que trouxe notoriedade ao Mirai. O que o tornou tão eficaz em seus ataques foi a sua grande capacidade de infectar e manter um grande número de *bots* ativos. A seguir os tipos de ataques disponíveis pelo Mirai.

- SYN Flood: é explorada uma característica do protocolo TCP, a *handshake* de três vias, que é o estabelecimento de comunicação entre o usuário e o receptor obedecendo três passos, o envio de uma *flag* chamada de SYN pelo usuário; o reconhecimento desta *flag* SYN pelo receptor, a alocação de recursos para a conexão e resposta ao usuário com um SYN-ACK; o reconhecimento do SYN-ACK pelo usuário e resposta com um pacote ACK, estabelecendo a conexão. Durante este ataque, o atacante envia vários pacotes com a *flag* SYN, fazendo o alvo alocar os recursos para a conexão mas não responde com o ACK, fazendo que o alvo fique com recursos alocados que nunca serão utilizados, até que se expire o tempo de espera pelo ACK (KUROSE; ROSS, 2010).
- ACK Flood: este ataque explora o mesmo protocolo explicado acima. Nele, vários pacotes ACK são enviados ao alvo, sem que nenhum dos passos anteriores tenham sido feitos. Deste modo, o alvo ficará analisando estes pacotes, procurando por sua tentativa de conexão que nunca será encontrada. Durante essa análise, são perdidos tempo e recursos, podendo provocar a indisponibilidade do serviço para usuários legítimos.
- TCP STOMP Flood: é explorado aqui o protocolo de mensagens orientadas a texto simples (STOMP), onde duas aplicações feitas em diferentes linguagens de programação podem se comunicar. Aqui é estabelecida uma conexão TCP entre o atacante e o alvo utilizando STOMP. Uma vez realizada a conexão, são enviados pacotes TCP STOMP para o alvo que, se estiver configurado para analisar este tipo de solicitação, gastará recursos analisando dados falsos.
- UDP Flood: neste ataque inúmeros pacotes UDP são enviados ao alvo do ataque com o objetivo de torná-lo indisponível para requisições de usuários legítimos. Neste ataque são sobrecarregadas portas aleatórias no alvo, uma vez que requisições são

feitas nestas portas e o alvo tem que responder a estas requisições informando que o destino do pacote não estava disponível.

- Plain UDP Flood: este ataque é uma variação mais rápida e otimizada do UDP Flood. A variação consiste em utilizar uma porta fixa, que o *bot* se conecta antes de executar o ataque de *loop*.
- GRE IP Flood e GRE Ethernet Flood: são exploradas características do protocolo de comunicação GRE. Este protocolo é utilizado para estabelecer conexão ponto a ponto direta, ou seja, em pares, representando um túnel entre emissor e receptor onde não há interceptações no meio do caminho. Devido a esta característica, somente o receptor poderá desencapsular o pacote GRE enviado, analisando a informação enviada e então tomar ações em seguida. Neste ataque o Mirai especificamente envia pacotes de 512 bytes ao alvo para serem desencapsulados, gastando recursos consideráveis do alvo nesta tarefa e gerando instabilidade e até indisponibilidade do serviço.
- DNS Flood: este ataque, que é uma variação do UDP Flood, é feito a servidores DNS, que são responsáveis pela tradução de um endereço de IP para um valor mais amigável ao usuário, como por exemplo "www.ufu.br". O intuito é realizar várias requisições ao servidor DNS de modo que ele fique sobrecarregado e não consiga atender a requisições de usuário legítimos.
- HTTP Flood: neste ataque existem duas possibilidades exploradas, o ataque *GET* e o ataque *POST*. No ataque *GET* são enviadas ao alvo requisições de recursos enquanto no ataque *POST* são enviados ao alvo dados para que sejam manipulados e persistidos. A intenção desta abordagem é esgotar os recursos do alvo com a enorme quantidade de requisições HTTP enviadas, tornando o serviço indisponível para solicitações de usuários legítimos.
- VSE Flood: é um ataque de inundação *Valve Source Engine* (VSE) que consiste na amplificação UDP, buscando consumir recursos de servidores de jogos através do envio de requisições *TSource Engine Query*.

3.2.2 C&C

O servidor de comando e controle (C&C) do Mirai foi todo implementado na linguagem de programação Go, que foi desenvolvida e disponibilizada pelo Google. Este servidor C&C tem como finalidade comandar e controlar o Mirai. É nele que o *botmaster* gerencia os dispositivos infectados que formam a *botnet* do Mirai, enviando comandos e gerenciando ataques. A comunicação entre *bot* e servidor C&C se dá através da porta 23, utilizada para comunicação direta entre eles, e da porta 101, utilizada para comunicação

com uma API alternativa do Mirai para realização de ataques. No servidor C&C existe um banco de dados *MySql* onde são armazenadas as informações dos ataques realizados e os dados dos usuários do Mirai. Dentro destes dados dos usuários estão armazenados as credenciais para acesso, se o usuário é administrador ou não, última vez que pagou pelos serviços, número máximo de *bots* por ataque, duração de cada ataque, tempo de espera entre ataques (RIEGEL, 2017).

O Mirai possui uma interface de administrador do sistema, utilizada somente por administradores, onde é possível gerenciar ataques, visualizar a quantidade de *bots* ativos na *botnet* e criar novos usuários para uso do Mirai e definir as condições de uso destes usuários. Além desta estrutura há também uma API, onde é possível que o usuário envie ordens de ataques a *botnet*, informando o endereço de IP do alvo e a duração do ataque. Com isso é possível perceber que o Mirai foi implementado de maneira profissional, para que seus criadores conseguissem alugá-lo para que terceiros realizassem ataques sem que estes pudessem realizar modificações diretas nas configurações do Mirai. É importante ressaltar que o servidor C&C não realiza nenhum ataque, sendo que estes ficam a cargo dos *bots*. O papel do servidor C&C é apenas de gerenciamento, enviando e controlando ordens de ataques aos *bots*.

3.2.3 *Loader*

A finalidade deste módulo, codificado em linguagem C, é disseminar o Mirai pela Internet infectando novos dispositivos vulneráveis encontrados pelos *bots*. O Loader, que em tradução livre seria "carregador", carrega o arquivo binário do *malware* no dispositivo que está sendo infectado de acordo com sua arquitetura computacional. Para isso, o Loader realiza uma conexão com o dispositivo através do protocolo Telnet e, com as credenciais informadas anteriormente pelo *bot*, analisa o dispositivo para identificar qual sua arquitetura. A seguir ele utiliza comandos *bash* do sistema baseado em Linux, *Busy-Box*, que foi criado especialmente para sistemas operacionais embarcados limitados, para carregar o arquivo no dispositivo que está sendo infectado.

4 Aplicativo MiraiScanner

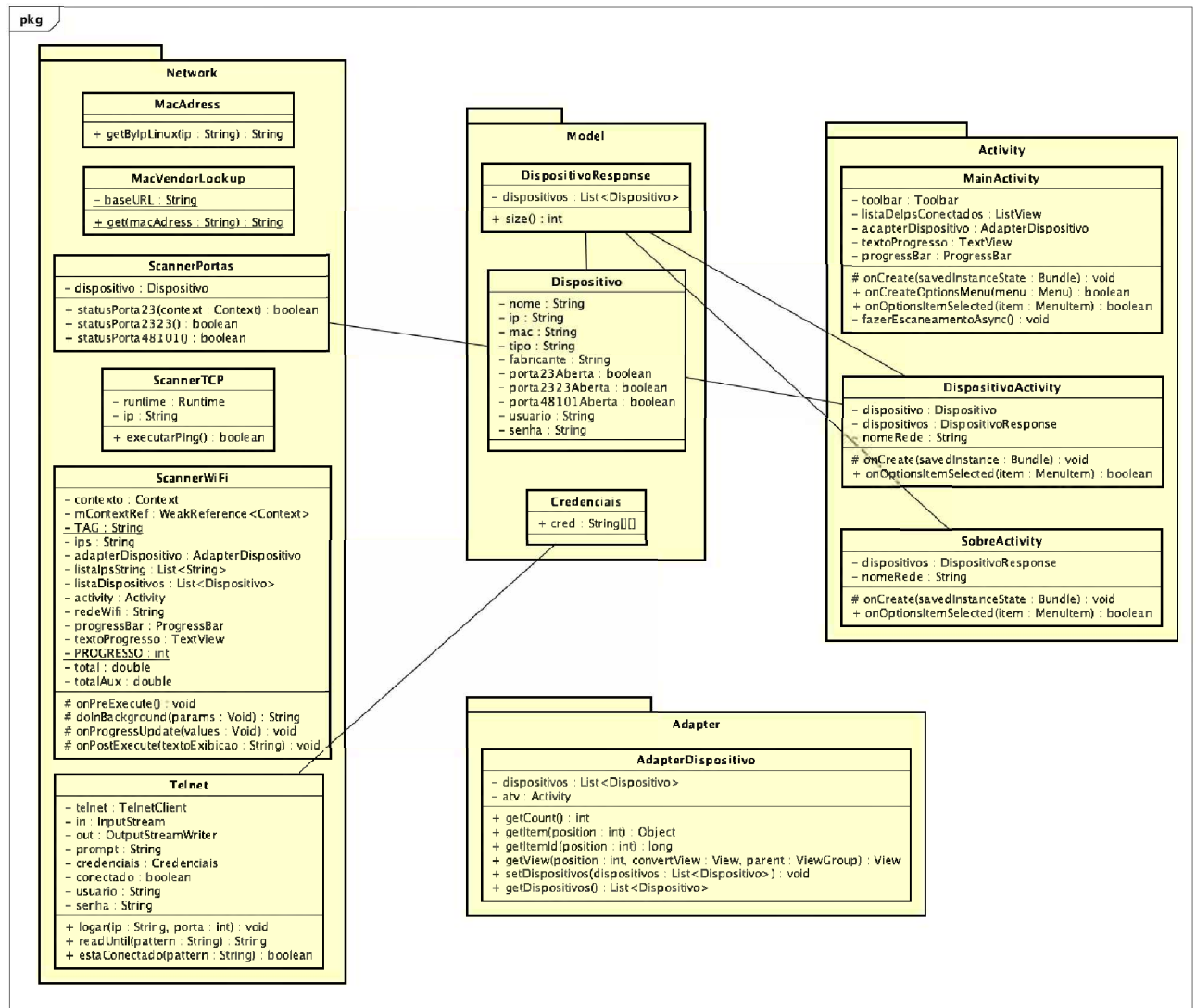


Figura 5 – Diagrama de Classes do Aplicativo MiraiScanner

Este capítulo descreve, de forma geral, o aplicativo e suas funcionalidades, explicando detalhadamente o seu funcionamento, como foi realizado o desenvolvimento das funcionalidades, as técnicas utilizadas para solucionar os problemas encontrados, as motivações para uso de *APIs* externas e os protocolos de redes que foram usados durante a codificação. Na figura 5, está um diagrama de classes que mostra a organização arquitetural do aplicativo.

Na seção 4.1 é apresentada uma visão geral sobre o funcionamento do aplicativo, do ponto de vista do que é perceptível ao usuário durante a utilização do mesmo, explicando detalhes dos itens da tela e os valores que são exibidos durante a execução.

Nesta seção também é mostrado de forma geral como é feito o escaneamento da rede e as particularidades das redes domésticas que foram consideradas durante a codificação.

Na seção 4.2 é apresentada uma visão mais técnica das classes utilizadas na codificação, mostrando suas heranças e interfaces herdadas assim como *APIs* externas utilizadas. Nesta seção também são explicados os protocolos de redes utilizados, a motivação de utilização dos mesmos e trechos de código fonte do aplicativo.

4.1 Escaneamento da Rede Doméstica

O aplicativo foi desenvolvido somente para dispositivos móveis que utilizam o sistema operacional do *Google*, o *Android*. Foi utilizada a IDE do *Android Studio* para codificação e compilação. Esta escolha se deve aos seguintes fatores:

- Utilização da linguagem Java pelo *Android Studio*, já conhecida e explorada amplamente durante a graduação.
- Acesso fácil a *smartphones* com sistema operacional *Android* para realização dos testes do aplicativo.
- Maior facilidade técnica e financeira de se publicar o aplicativo na loja de aplicativos própria do *Google* para o *Android*.

Também se optou por utilizar apenas o escaneamento de redes *Wi-Fi* uma vez que esta é a modalidade mais adotada em redes domésticas.

Na tela inicial do aplicativo, na parte superior é exibido o nome do aplicativo, Mirai Scanner, um botão de reinício do escaneamento da rede e um botão que leva às opções: Sobre e Sair. Na opção "Sobre" é aberta uma tela que mostra a versão do aplicativo, a data da última atualização e um breve resumo sobre a motivação da aplicação, conforme pode ser visto na Figura 6. Já a opção sair simplesmente encerra o aplicativo.

Os primeiros problemas a serem resolvidos pelo aplicativo são verificar se o dispositivo está conectado a uma rede *Wi-Fi* e identificá-la, para depois descobrir outros dispositivos conectados nesta mesma rede doméstica em que ele se encontra.

Para resolver estes problemas o aplicativo primeiro verifica a conexão do dispositivo, somente permitindo o escaneamento caso o mesmo esteja conectado a uma rede *Wi-Fi*. Uma vez verificada a conexão, a aplicação busca os dados da rede em que está conectada, como a identificação e *IP* base, que é o endereço *IP* do roteador. Cada endereço *IP* possui 4 octetos e pode abrigar até 255 dispositivos, modificando apenas seu último octeto. Como este número de dispositivos dificilmente será ultrapassado em uma rede doméstica, optou-se por fazer uma busca somente nestes 255 endereços *IP* que estão na

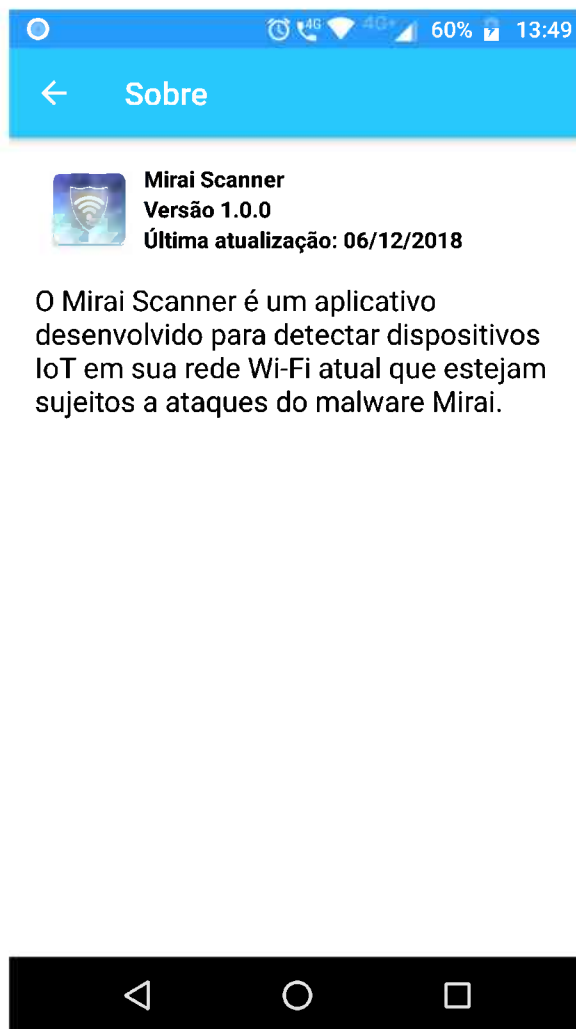


Figura 6 – Tela Sobre

mesma rede do dispositivo rodando o aplicativo. Ao se considerar uma rede doméstica, não há necessidade de realizar uma busca mais ampla pois raramente uma residência precisará utilizar mais de uma sub-rede para abrigar todos os seus dispositivos. Como o objetivo do aplicativo é unicamente escanear a rede em busca de vulnerabilidade de segurança, o escaneamento de rede é feito automaticamente ao se abrir o aplicativo, desde que o mesmo esteja conectado a uma rede *Wi-Fi*, não sendo necessário que o usuário clique em nenhum botão. Apesar do escaneamento automático ao abrir o aplicativo, o usuário também pode reiniciar o escaneamento a qualquer momento, através de um botão localizado no canto superior direito da tela não precisando nem mesmo esperar o fim do escaneamento atual.

A medida que o escaneamento é feito, uma barra exibe a progressão do processo em porcentagem. Feito o escaneamento, o aplicativo mostrará o nome da rede *Wi-Fi* que o dispositivo está conectado, o número de dispositivos encontrados no escaneamento e os dados básicos de cada dispositivo encontrado. Estes dados são o endereço *IP*, fabricante e número de *MAC*, como pode ser visto na Figura 7.

Caso nada tenha sido encontrado, a mensagem exibida informa que o dispositivo não possui nenhuma vulnerabilidade relacionada ao *Mirai*, conforme pode ser visto na Figura 8. É importante ressaltar que o aplicativo não diferencia dispositivos IoTs dentre os dispositivos encontrados. Isto se deve ao fato que o *Mirai* ataca qualquer dispositivo que apresente as vulnerabilidades exploradas. Normalmente dispositivos como computadores pessoais apresentam protocolos de segurança robustos e por isso não são infectados.

O código fonte da implementação base dessas funcionalidades está localizado na classe `MainActivity.java`, mostrada no Trecho de Código A.1.

4.2 Classes

A classe *ScannerWiFi*, mostrada no Trecho de Código A.2, implementa o escaneamento de rede e estende a classe, própria do *Android*, *AsyncTask*. O objetivo de se estender esta classe é utilizar o escaneamento de maneira assíncrona, melhorando a experiência do usuário, ou seja, as tarefas de escaneamento serão rodadas no processador do celular sem que a tela do aplicativo fique travada durante este processo.

O escaneamento da rede é implementado sobrescrevendo o método *doInBackground*, onde a aplicação busca os dados da rede em que o dispositivo está conectado, identificando o seu endereço *IP*. Este endereço *IP* é fragmentado e seu prefixo, que consiste nos três primeiros octetos do endereço *IP*, é salvo para identificar a rede. Com esta identificação, é criada uma *thread pool*, que é responsável por não deixar que um número excessivo de *threads* sejam criadas durante o escaneamento, o que poderia prejudicar o desempenho da aplicação.

A implementação de *threads* na aplicação foi necessária para que o tempo de resposta do escaneamento não fosse alto demais, tornando o seu uso inviável pelo usuário doméstico comum. Com as *threads*, o desempenho do escaneamento melhorou seu tempo de resposta mais de vinte vezes, diminuindo de mais de cinco minutos para menos de quinze segundos. Com a *thread pool* configurada, para que não haja *threads* demais sendo executadas ao mesmo tempo pelo processador, a aplicação começa a testar, utilizando a classe *ScannerTCP*, os duzentos e cinquenta e cinco endereços *IPs* que podem ou não conter dispositivos ativos.

Na classe *ScannerTCP*, mostrada no Trecho de Código A.3, é utilizado o protocolo *Transmission Control Protocol* (TCP) complementado pelo protocolo *IP*, para testar se um determinado endereço *IP* pertence a um dispositivo ativo da rede. Estes protocolos foram utilizados por serem robustos e versáteis, além de garantirem a entrega de determinado pacote enviado pela aplicação. Como o escaner necessita ter certeza que o pacote foi recebido pelo outro dispositivo a fim de confirmar sua presença na rede, o TCP se torna uma opção melhor que protocolos mais simples da camada de transporte, como é o caso

do *User Datagram Protocol* (UDP), que não garante tal entrega (KUROSE; ROSS, 2010).

O escaneamento consiste em enviar um comando de *ping*, que testa a conectividade entre dispositivos, para o endereço *IP* que está sendo verificado e esperar uma resposta ou o estouro do tempo, o *timeout*. Caso o endereço *IP* contenha um dispositivo ativo, a aplicação tenta identificar o endereço do *hardware* da placa de rede do dispositivo, conhecido como endereço *MAC*, através da *interface NetworkInterface*, do *Android*, e da classe *MacAddress*.

A implementação da classe *MacAddress* foi necessária para contemplar dispositivos que rodam uma versão do sistema operacional *Linux*, e que não são tão bem mapeados pela interface própria do *Android*. A implementação desta classe foi inspirada no aplicativo *evercam* Evercam (2018) e nela é feita a leitura da tabela ARP (*Address Resolution Protocol*) via *broadcast*. Este endereço é então analisado e formatado de acordo com o padrão usual de visualização de endereços *MACs*. A seguir é feita uma requisição *HTTP* do tipo *GET* para uma *API rest* no endereço <https://api.macvendors.com/>, passando o número de *MAC* como parâmetro e recebendo o nome do fabricante como resposta. Esta *API* foi escolhida por ser gratuita e disponibilizada *online* na Internet pelo seu desenvolvedor. Esse procedimento de verificação do fabricante é importante pois, para o usuário doméstico, números de endereço *IP* e de *MAC* podem não fazer sentido, enquanto o nome do fabricante pode facilitar a identificação do dispositivo sendo mostrado no aplicativo.

Identificar possíveis vulnerabilidades nos dispositivos escaneados e uma possível infecção pelo *Mirai* é o objetivo principal da classe *ScannerPortas*, mostrada no Trecho de Código A.4. Nesta classe é utilizado o conceito de *sockets TCP/IP*, orientado à conexão (NUNES, 2018), para testar se as portas 23, 2323 e 48101 estão abertas nos dispositivos. *Socket* é um mecanismo de comunicação entre aplicações que permite a troca de mensagens entre eles. O endereço do *socket* é uma combinação do número do endereço *IP* e do número da porta, sendo assim, o processo de comunicação entre o aplicativo e o dispositivo ocorre da seguinte forma: o aplicativo solicita uma conexão no endereço *IP* do dispositivo na porta 23 ou 48101, via *socket*. A escolha destas portas será explicada a seguir.

A porta 23 está normalmente associada ao protocolo *Telnet* e, em alguns dispositivos *IoT*s, a porta 2323 também costuma ser utilizada (RIEGEL, 2017).

Este protocolo permite ao usuário uma conexão ao interpretador de comandos do dispositivo, sem que medidas de segurança sejam implementadas (SSH.COM, 2018). Sendo assim, caso esta porta esteja aberta, um usuário mal intencionado pode facilmente utilizá-la para baixar e instalar programas maliciosos no dispositivo, caso tenha conhecimento de suas credenciais de acesso.

No caso específico do *Mirai*, é exatamente este processo que é utilizado para infectar

o dispositivo, onde o *Mirai* usa um método de força bruta para tentar descobrir o usuário e senha do dispositivo e ganhar acesso a ele através do protocolo Telnet na porta 23 ou 2323.

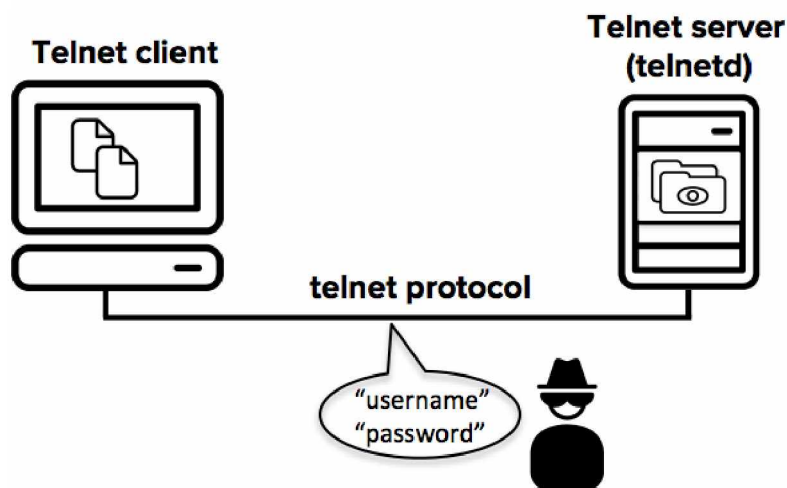


Figura 9 – Conexão via protocolo *Telnet* utilizando apenas usuário e senha. Extraído de [SSH.com](https://www.ssh.com) (2018)

A porta 48101 é utilizada para enviar comandos ao *bot* (RIEGEL, 2017). Ou seja, caso esta porta esteja aberta, é um grande indício que o dispositivo está infectado pelo *Mirai* e já pertence a seu *botnet*, estando pronto para ser utilizado em um possível ataque.

Ao utilizar somente a verificação das portas, a fim de verificar que existe um *bot* do *Mirai* presente na rede doméstica, o aplicativo apresenta uma limitação. Devido a inviabilidade de monitorar o tráfego de dados da rede em tempo real, que consumiria grande parte do processamento do dispositivo do usuário, caso o administrador do *Mirai* resolva modificar as portas utilizadas, o aplicativo poderá encontrar dificuldades em localizar os dispositivos vulneráveis na rede doméstica. Outro possível problema que pode ocorrer é o fato de existirem algumas poucas impressoras que utilizam a mesma porta, 48101, para troca de mensagens entre o computador e a impressora, não sendo possível diferenciar quando é uma infecção e quando é somente uma impressora utilizando a porta 48101. Como o aplicativo tem como objetivo informar o usuário sobre possíveis falhas de segurança em seus dispositivos, no caso de um falso alerta é melhor ter um excesso do que uma falta de informação e precaução, não causando assim problemas maiores do que uma verificação extra de segurança ao usuário. No futuro, pretende-se modificar o modo de verificação do aplicativo para se obter melhores resultados e contornar tais problemas.

Com todo o escaneamento concluído, a aplicação mostra ao usuário todos os dispositivos encontrados assim como seus endereços *IPs*, *MACs* e a situação das portas 23, 2323 e 48101 de cada um deles. Caso seja encontrada alguma das portas abertas, o aplicativo mostra uma mensagem de aviso ao usuário alertando sobre a existência de vulnerabilida-

des. Caso nenhum problema seja detectado, a aplicação exibe uma mensagem informando ao usuário que nenhum problema foi encontrado.

4.3 Ambiente de Testes

Para os testes de funcionamento do aplicativo, foi utilizada uma rede doméstica *Wi-Fi*, denominada BATMAN, com quatro dispositivos conectados a ela, conforme Figura 10, com a seguinte estrutura:

- Um roteador *Wi-Fi* Arris TG1692, cujo endereço *IP* é 192.168.0.1
- Celular Moto X4 rodando o aplicativo, cujo endereço *IP* é 192.168.0.2
- Um *notebook* HP com Windows 7 e servidor Telnet ativo na porta 23, cujo endereço *IP* é 192.168.0.9
- Um Macbook Pro para verificar os *logs* durante o escaneamento, cujo endereço *IP* é 192.168.0.3

Para testar todas as funcionalidades do aplicativo, foram testados dispositivos sem vulnerabilidades, conforme Figura 11, relacionadas ao Mirai e configurado um dispositivo para simular a vulnerabilidade do serviço Telnet na porta 23.

Para os testes de conexão ao servidor Telnet, foi modificado o arquivo de credenciais de acesso ao serviço Telnet para incluir uma credencial a fim de ser compatível com o servidor Telnet ativo configurado no *notebook* HP. Depois deste teste, o arquivo de credencias foi modificado novamente para conter apenas as credenciais utilizadas no código fonte do Mirai que foi disponibilizado na Internet. O resultado pode ser visto na Figura 12.

Para o futuro, seria interessante montar um ambiente fechado contendo o Mirai e dispositivos IoT vulneráveis, ou com *bots* do Mirai já instalados neles, a fim de realizar testes do aplicativo para apurar sua confiabilidade.

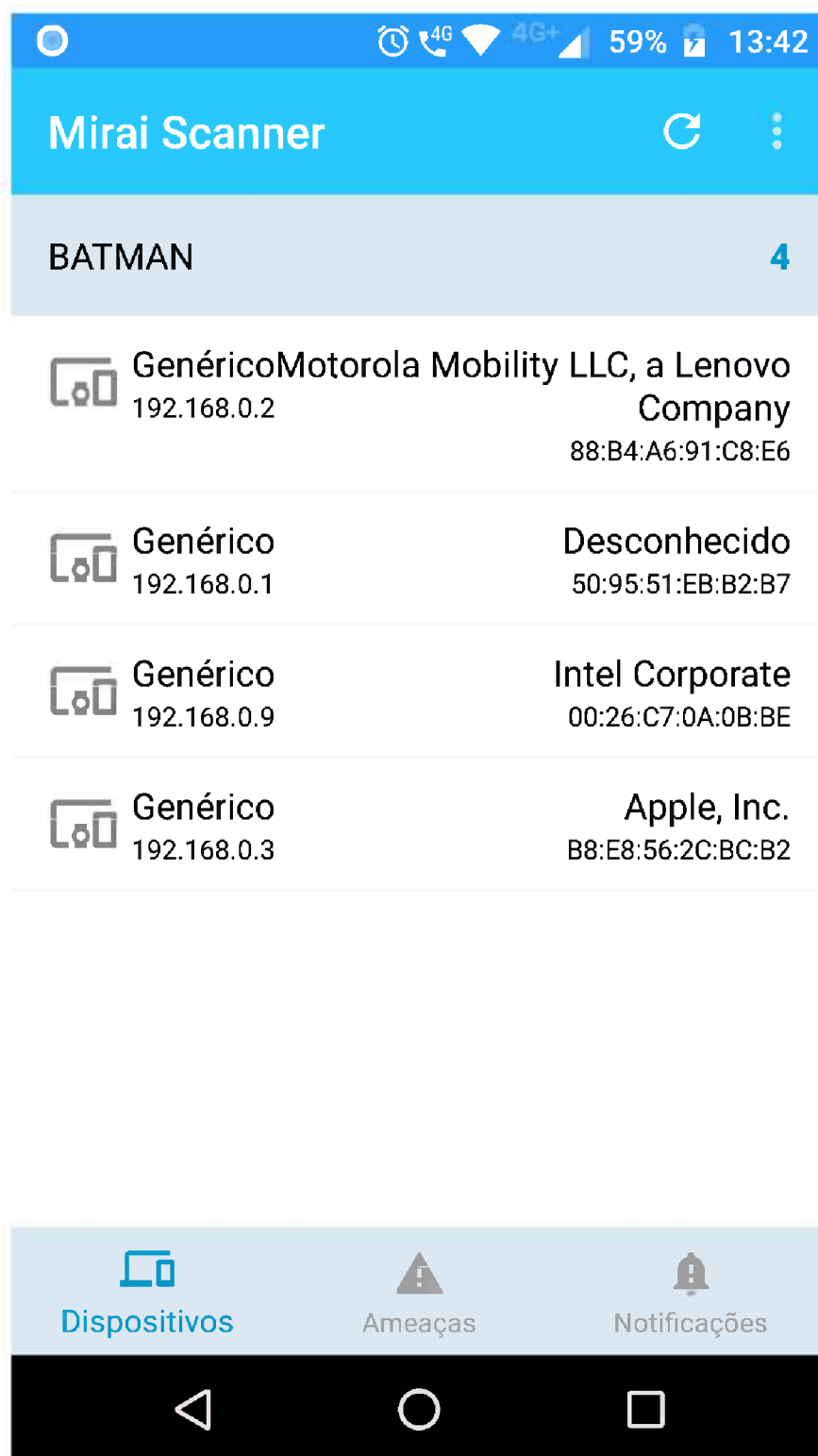


Figura 10 – Listagem de dispositivos durante realização de testes do aplicativo



Figura 11 – Dispositivo sem vulnerabilidades encontradas



Figura 12 – Verificação de acesso Telnet

5 Conclusão

O aumento de dispositivos IoT nos ambientes domésticos tem levado a uma nova evolução das ameaças à Segurança da Informação. Ataques DDoS tem alcançando volumes de tráfego antes inimagináveis devido a quantidade de *bots* ativos nos *botnets*. O fato do Mirai se preocupar em eliminar processos de *malwares* concorrentes mostra que estes dispositivos IoTs já são alvos visados pelos *hackers* por serem fáceis de ser infectados e por serem abundantes na rede mundial de computadores. Além disso, empresas especializadas em segurança da informação, como a Kaspersky, já estão desenvolvendo aplicativos com a finalidade de encontrar dispositivos vulneráveis e informar aos seus donos, o que mostra como o mercado já sente a necessidade da melhora da segurança nestes dispositivos.

É fácil notar que estes dispositivos que surgiram com o intuito de serem populares e baratos, na sua maioria não vem de fábrica com medidas básicas de segurança que já são implementadas em computadores pessoais, por exemplo. Ao não tomar cuidado com as medidas de segurança que saem embarcadas em seus produtos, os fabricantes acabaram deixando uma brecha de segurança que pode prejudicar todos que dependem da Internet no seu dia a dia e alimentam ainda mais o mercado negro de ataques DDoS voltados a derrubar servidores de serviços, páginas web de concorrentes, entre outros. Sem que haja uma conscientização mundial que medidas de segurança devem ser elementos básicos em todos os dispositivos que venham a ser conectados na Internet, continuaremos a ver ataques ainda mais poderosos e novas brechas surgindo constantemente.

Aplicativos móveis, como o Mirai Scanner podem ajudar os usuários domésticos e com pouco ou nenhum conhecimento técnico de computação, na conscientização sobre a importância da segurança de seus dispositivos uma vez que, eles ajudam a mostrar vulnerabilidades que os donos dos dispositivos não sabiam que existiam. Talvez com esta conscientização, os usuários passem a se preocupar em comprar apenas produtos que forneçam um grau de segurança adequado, o que forçaria os grandes fabricantes a implementar medidas fortes de segurança em seus produtos, dificultando a vida dos *hackers*.

Tentar instalar aplicações do bem que consertem as vulnerabilidades encontradas e elimine os *malwares* dos dispositivos, pode ser uma solução viável no curto prazo. Porém, é importante se preocupar com o fato que caso o controle destas aplicações positivas caiam em poder de pessoas má intencionadas, o que poderia ser uma solução pode se tornar um problema ainda maior. No longo prazo implantar legislações exigindo um mínimo de protocolos de segurança implantados em dispositivos que se conectarão à Internet parece ser a solução que melhor reduziria o problema.

Para trabalhos futuros, novas verificações de vulnerabilidades podem ser imple-

mentadas ampliando ainda mais a conscientização dos usuários sobre os riscos existentes em seus dispositivos. Entre algumas das possibilidades, pode-se tentar encontrar uma forma de identificar *malwares* conhecidos e realizar prevenções específicas contra eles. Também planeja-se lançar na versão 2.0 do Mirai Scanner uma implementação que consiga identificar ativamente um dispositivo que esteja infectado com o Mirai. Para isso, pretende-se fazer uma análise da troca de mensagens entre o *bot* e o Mirai e tentar fazer o *bot* pensar que o aplicativo é o Mirai, enviando assim uma resposta rastreável que confirmaria sua identidade. Ainda em trabalhos futuros, pode-se explorar meios de trocar senhas padrão de dispositivos por senhas robustas através do próprio aplicativo, com a devida autorização do usuário.

Por fim, o código fonte completo do aplicativo Mirai Scanner encontra-se disponível gratuitamente no GitHub, no endereço <<https://github.com/mirandagab/miraiScanner>>, a fim que toda a comunidade possa contribuir para tornar a Internet um local mais seguro para se navegar e facilitar o dia a dia das pessoas.

Referências

- ANNA-SENPAL. *Mirai_How_To_Build.pdf*. 2016. Disponível em: <<https://github.com/jgamblin/Mirai-Source-Code/blob/master/ForumPost.txt>>. Acesso em: 03/12/2017. Citado 2 vezes nas páginas 7 e 22.
- BING, C. *Mirai botnet hacker ordered to pay \$8.6 million in damages*. 2018. Disponível em: <<https://www.reuters.com/article/us-usa-cyber-mirai/mirai-botnet-hacker-ordered-to-pay-8-6-million-in-damages-idUSKCN1N02VY>>. Acesso em: 25/11/2018. Citado na página 24.
- CANALTECH. *O que é DoS e DDoS?* 2014. Disponível em: <<https://canaltech.com.br/produtos/O-que-e-DoS-e-DDoS/>>. Acesso em: 25/11/2018. Citado na página 15.
- CIPHER. *Gerenciando desafios de segurança e privacidade IoT*. 2018. Disponível em: <<https://cipher.com.br/2018/02/02/gerenciando-desafios-de-seguranca-e-privacidade-da-iot/>>. Acesso em: 07/12/2018. Citado na página 17.
- De Donno, M. et al. AntibIoTic: Protecting IoT devices against DDoS attacks. *Advances in Intelligent Systems and Computing*, v. 717, p. 59–72, 2017. ISSN 21945357. Citado 2 vezes nas páginas 12 e 20.
- EVERCAM. *Evercam Construction Time-lapse Cameras*. 2018. Citado na página 34.
- GRAFF, G. M. *HOW A DORM ROOM MINECRAFT SCAM BROUGHT DOWN THE INTERNET*. 2016. Acesso em: 23/11/2018. Citado na página 23.
- GUBBI, J. et al. Internet of Things (IoT): A vision, architectural elements, and future directions. *Future Generation Computer Systems*, v. 29, n. 7, p. 1645–1660, 2013. ISSN 0167739X. Citado na página 11.
- Hewlett Packard (HP). *HP Study Reveals 70 Percent of Internet of Things Devices Vulnerable to Attack*. 2014. Disponível em: <<http://www8.hp.com/us/en/hp-news/press-release.html?id=1744676>>. Acesso em: 22/06/2018. Citado na página 17.
- HOEPERS, C.; STEDING-JESSEN, K. Fundamentos de Segurança da Informação. p. 47, 2014. Disponível em: <<https://www.cert.br/docs/palestras/certbr-egi2014.pdf>>. Citado na página 14.
- KOLIAS, C. et al. DDoS in the IoT: Mirai and other botnets. *Computer*, v. 50, n. 7, p. 80–84, 2017. ISSN 00189162. Citado 3 vezes nas páginas 11, 16 e 20.
- KREBS, B. *Source Code for IoT Botnet 'Mirai' Released*. 2016. Disponível em: <<https://krebsonsecurity.com/2016/10/source-code-for-iot-botnet-mirai-released/>>. Acesso em: 25/11/2018. Citado na página 23.
- KREBS, B. *Mirai Botnet Authors Avoid Jail Time*. 2018. Disponível em: <<https://krebsonsecurity.com/2018/09/mirai-botnet-authors-avoid-jail-time/>>. Acesso em: 25/11/2018. Citado na página 22.

KUROSE, J. F.; ROSS, K. W. Redes de computadores e a internet: uma abordagem top-down. In: *Redes de computadores e a internet: uma abordagem top-down*. [S.l.: s.n.], 2010. ISBN 8588639971. Citado 2 vezes nas páginas 26 e 34.

MEULEN, R. van der. *Gartner Says 6.4 Billion Connected "Things" Will Be in Use in 2016, Up 30 Percent From 2015*. 2015. Disponível em: <<https://www.gartner.com/newsroom/id/3165317>>. Acesso em: 25/11/2018. Citado na página 17.

MICROSOFT. *Planejando a proteção contra ataques de inundação de negação de serviço*. 2009. Disponível em: <<https://technet.microsoft.com/pt-br/Library/dd897007.aspx?f=255&MSPPErr=-2147217>>. Acesso em: 07/12/2018. Citado na página 15.

MICROSOFT. *O que é botnet?* 2012. Disponível em: <<https://www.microsoft.com/pt-br/security/resources/botnet-what-is.aspx>>. Citado na página 11.

NAKAMURA, E. T.; GEUS, P. L. de. *Segurança de Redes em Ambientes Cooperativos*. [S.l.: s.n.], 2007. 488 p. Citado na página 14.

NUNES, L. R. *Sockets em Java*. 2018. Acesso em: 2018-04-11. Citado na página 34.

OWASP. *OWASP*. 2014. Acesso em: 07/12/2018. Citado na página 17.

PAGE, B. *A report on the internet worm*. 1988. Disponível em: <<https://www.ee.ryerson.ca/~elf/hack/iworm.ht>>. Acesso em: 2018-06-20. Citado na página 15.

PEREKALIN, A. *Novo app da Kaspersky ajuda a proteger a rede de sua casa*. 2018. Disponível em: <<https://www.kaspersky.com.br/blog/kaspersky-iot-scanner/10241/>>. Acesso em: 10/05/2018. Citado 2 vezes nas páginas 12 e 20.

PRADO, M. A. Análise experimental da botnet IoT Mirai. n. Dezembro, 2018. Citado 2 vezes nas páginas 7 e 24.

RIEGEL, M. C. Tracking Mirai an indepth analysis of an IoT botnet. n. August, 2017. Citado 8 vezes nas páginas 17, 18, 20, 23, 25, 28, 34 e 35.

SAB, G. A. A.; FERREIRA, R. C.; ROZENDO, R. G. *Negação de Serviço, Negação de Serviço Distribuída e Botnets*. 2013. Disponível em: <https://www.gta.ufrj.br/grad/13{_}1/dos/ataques.h>. Acesso em: 07/12/2018. Citado 2 vezes nas páginas 15 e 16.

SAP. *O que é IoT - a internet das Coisas?* 2016. Disponível em: <<https://news.sap.com/brazil/2016/05/o-que-e-iot-a-internet-das-coisas/>>. Acesso em: 07/12/2018. Citado na página 11.

SINANOVIC, H.; MRDOVIC, S. Analysis of Mirai malicious software. *2017 25th International Conference on Software, Telecommunications and Computer Networks, SoftCOM 2017*, 2017. Citado 2 vezes nas páginas 20 e 23.

SPANCESKI, F. R. Estrutura Nacional de Segurança da Informação (ENSI) - Política de Segurança da Informação Detalhada da Entidade. p. 102, 2004. Citado na página 14.

SSH.COM. *Telnet - And SSH As A Secure Alternative*. 2018. Disponível em: <<https://www.ssh.com/ssh/telnet>>. Acesso em: 11/04/2018. Citado 3 vezes nas páginas 7, 34 e 35.

STATISTA. *Number of smartphones user worldwide from 2014 to 2020*. 2016. Disponível em: <<https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/>>. Acesso em: 07/12/2018. Citado na página 21.

SYMANTEC. *Mirai: what you need to know about the botnet behind recent major DDoS attacks*. 2016. Disponível em: <<https://www.symantec.com/connect/blogs/mirai-what-you-need-know-about-botnet-behind-recent-major-ddos-attacks>>. Acesso em: 25/11/2018. Citado 3 vezes nas páginas 18, 23 e 25.

VERTON, D. *The war on botnets evolves*. 2014. Disponível em: <<https://www.fedscoop.com/war-botnets-evolves/>>. Acesso em: 25/11/2018. Citado na página 19.

ZEIFMAN, I.; BEKERMAN, D.; HERZBERG, B. Breaking Down Mirai: An IoT DDoS Botnet Analysis. *Incapsula.Com*, p. 1–11, 2016. Disponível em: <<https://www.incapsula.com/blog/malware-analysis-mirai-ddos-botnet.html>>. Citado na página 22.

Anexos

ANEXO A – Trechos de Código

Trecho de Código A.1 – Classe MainActivity.java

```
1 public class MainActivity extends AppCompatActivity {
2
3     private android.support.v7.widget.Toolbar toolbar;
4     private ListView listaDeIpsConectados;
5     private AdapterDispositivo adapterDispositivo;
6     private TextView textoProgresso;
7     private ProgressBar progressBar;
8
9     @Override
10    protected void onCreate(Bundle savedInstanceState) {
11        super.onCreate(savedInstanceState);
12        setContentView(R.layout.activity_main);
13
14        //criação da progressBar
15        progressBar = (ProgressBar) findViewById(R.id.barraProgresso);
16        textoProgresso = (TextView) findViewById(R.id.textoProgresso);
17
18        //criação da toolbar
19        toolbar = (Toolbar) findViewById(R.id.toolbar);
20        setSupportActionBar(toolbar);
21
22        //criação da listview
23        listaDeIpsConectados = (ListView) findViewById(R.id.
24            listaDeIpsConectadosID);
25
26        //seta a ação de clique no item
27        listaDeIpsConectados.setOnItemClickListener(new AdapterView.
28            OnItemClickListener() {
29                @Override
30                public void onItemClick(AdapterView<?> parent, View view, int
31                    position,
32                        long id) {
33                    Intent intent = new Intent(MainActivity.this,
34                        DispositivoActivity.class);
35
36                    TextView nomeRede = (TextView) findViewById(R.id.rede_wifi);
```

```

33         DispositivoResponse dispositivoResponse = new
           DispositivoResponse(adapterDispositivo.getDispositivos());
34         Dispositivo disp = (Dispositivo) parent.getItemAtPosition(
           position);
35
36         intent.putExtra("dispositivos", dispositivoResponse);
37         intent.putExtra("nome_rede", nomeRede.getText());
38         intent.putExtra("dispositivo", disp);
39         startActivity(intent);
40     }
41 });
42 adapterDispositivo = new AdapterDispositivo(new ArrayList<Dispositivo
           >(), MainActivity.this);
43 Intent it = this.getIntent();
44
45 if(it.getSerializableExtra("dispositivos") != null) {
46     DispositivoResponse dispositivoResponse = (DispositivoResponse) it.
           getSerializableExtra("dispositivos");
47     adapterDispositivo.setDispositivos(dispositivoResponse.
           getDispositivos());
48
49     TextView nome_rede = (TextView) this.findViewById(R.id.rede_wifi);
50     if(it.getStringExtra("nome_rede") != null)
51         nome_rede.setText(it.getStringExtra("nome_rede"));
52     else
53         nome_rede.setText("Rede desconhecida");
54
55     TextView qtdDispositivos = (TextView) this.findViewById(R.id.
           qtd_dispositivos);
56     qtdDispositivos.setText(dispositivoResponse.size() + "");
57
58     progressBar.setVisibility(ProgressBar.GONE);
59     textoProgresso.setText("Rede escaneada");
60     textoProgresso.setVisibility(TextView.GONE);
61 }
62
63 listaDeIpsConectados.setAdapter(adapterDispositivo);
64
65 if(adapterDispositivo.isEmpty())
66     fazerEscaneamentoAsync();
67 }
68

```

```

69     @Override
70     public boolean onCreateOptionsMenu(Menu menu) {
71         MenuInflater inflater = getMenuInflater();
72
73         inflater.inflate(R.menu.menu_main, menu);
74         return true;
75     }
76
77     @Override
78     public boolean onOptionsItemSelected(MenuItem item) {
79         switch(item.getItemId()){
80             case R.id.item_sair:
81                 finish();
82                 System.exit(1);
83                 return true;
84             case R.id.item_configuracoes:
85                 return true;
86             case R.id.item_atualizar:
87                 fazerEscaneamentoAsync();
88                 return true;
89             case R.id.item_sobre:
90                 Intent intent = new Intent(MainActivity.this, SobreActivity.
91                     class);
92                 TextView nomeRede = (TextView) this.findViewById(R.id.rede_wifi)
93                     ;
94
95                 DispositivoResponse dispositivoResponse = new
96                     DispositivoResponse(this.adapterDispositivo.getDispositivos()
97                     );
98                 intent.putExtra("dispositivos", dispositivoResponse);
99                 intent.putExtra("nome_rede", nomeRede.getText());
100                 startActivity(intent);
101                 return true;
102             default:
103                 return super.onOptionsItemSelected(item);
104         }
105     }
106
107     private void fazerEscaneamentoAsync() {
108         ConnectivityManager cm = (ConnectivityManager) this.getSystemService(
109             Context.CONNECTIVITY_SERVICE);
110         NetworkInfo tipoRede = cm.getActiveNetworkInfo();

```

```

106         if(tipoRede.getType() == ConnectivityManager.TYPE_WIFI) {
107             ScannerWiFi scannerWiFi = new ScannerWiFi(MainActivity.this,
                adapterDispositivo,
108                 this, progressBar, textoProgresso);
109             scannerWiFi.execute();
110         }else{
111             Toast.makeText(this, "Sem conexão WiFi", Toast.LENGTH_LONG).show();
112         }
113     }
114 }

```

Trecho de Código A.2 – Classe ScannerWiFi.java

```

1 public class ScannerWiFi extends AsyncTask<Void, Void, String>{
2
3     private Context contexto;
4
5     private WeakReference<Context> mContextRef;
6
7     private static final String TAG = "ScannerWiFi";
8
9     private String ips = "";
10
11     private AdapterDispositivo adapterDispositivo;
12
13     private List<String> listaIpsString;
14
15     private List<Dispositivo> listaDispositivos;
16
17     private Activity activity;
18
19     private String redeWifi;
20
21     private ProgressBar progressBar;
22
23     private TextView textoProgresso;
24
25     private static int PROGRESSO = 39;
26
27     private double total, totalAux = 0;
28
29     public ScannerWiFi(Context contexto, AdapterDispositivo adapterDispositivo,
        Activity activity,

```

```

30         ProgressBar progressBar, TextView textoProgreso){
31     this.contexto = contexto;
32     this.mContextRef = new WeakReference<Context>(contexto);
33     this.adapterDispositivo = adapterDispositivo;
34     this.activity = activity;
35     this.textoProgreso = textoProgreso;
36     this.progressBar = progressBar;
37 }
38
39 @Override
40 protected void onPreExecute() {
41     total = 0;
42     progressBar.setMax(10000);
43     progressBar.setProgress(0);
44     progressBar.setVisibility(ProgressBar.VISIBLE);
45     textoProgreso.setGravity(Gravity.CENTER_HORIZONTAL);
46     textoProgreso.setVisibility(TextView.VISIBLE);
47     textoProgreso.setText("0%");
48     super.onPreExecute();
49 }
50
51 @Override
52 protected String doInBackground(Void... params) {
53
54     try {
55         listaIpsString = new ArrayList<>();
56         listaDispositivos = new ArrayList<>();
57
58         Context context = mContextRef.get();
59
60         if (context != null) {
61
62             ConnectivityManager cm = (ConnectivityManager)context.
63                 getSystemService(Context.CONNECTIVITY_SERVICE);
64             NetworkInfo activeNetwork = cm.getActiveNetworkInfo();
65             WifiManager wm = (WifiManager)context.getApplicationContext().
66                 getSystemService(Context.WIFI_SERVICE);
67
68             WifiInfo connectionInfo = wm.getConnectionInfo();
69             int ipAddress = connectionInfo.getIpAddress();
70             this.redeWifi = connectionInfo.getSSID().substring(1,
71                 connectionInfo.getSSID().length() - 1);

```

```

69         String ipString = Formatter.formatIpAddress(ipAddress);
70
71         Log.d(TAG, "activeNetwork: " + String.valueOf(activeNetwork));
72         Log.d(TAG, "ipString: " + String.valueOf(ipString));
73         ips += "activeNetwork: " + String.valueOf(activeNetwork) + "
              ipString: "
74               + String.valueOf(ipString);
75
76         String prefix = ipString.substring(0, ipString.lastIndexOf(".")
              + 1);
77         Log.d(TAG, "prefix: " + prefix);
78
79         ExecutorService threadPool = Executors.newFixedThreadPool(85);
80
81         for (int i = 0; i < 255; i++) {
82             final String prefixo = prefix;
83             final int j = i;
84             threadPool.execute(new Runnable() {
85                 @Override
86                 public void run() {
87                     String verificandoIP = prefixo + String.valueOf(j);
88                     InetAddress enderecoInet = null;
89                     try {
90                         enderecoInet = InetAddress.getByName(verificandoIP
91                             );
92
93                     if(enderecoInet != null) {
94                         boolean reachable = false;
95
96                         ScannerTCP scannerTCP = new ScannerTCP(
97                             verificandoIP);
98                         reachable = scannerTCP.executarPing();
99
100                         String hostName = enderecoInet.
101                             getCanonicalHostName();
102                         NetworkInterface niMac = NetworkInterface.
103                             getByInetAddress(enderecoInet);
104
105                     if (reachable) {
106                         final String ip = verificandoIP;
107                         String endereco = "";
108                         if(niMac != null) {

```

```

105         byte[] enderecoMAC = niMac.
            getHardwareAddress();
106         String mac;
107         if (enderecoMAC != null) {
108             for (int i = 0; i < enderecoMAC.
                length; i++)
109                 endereco += (String.format("%02X
                    :", enderecoMAC[i]));
110             endereco = endereco.substring(0,
                endereco.length() - 1);
111         }
112     }
113     else
114         endereco = MacAddress.getByIpLinux(
            verificandoIP);
115
116     final String mac = endereco;
117     final String hname = hostName;
118     listaIpsString.add(String.valueOf(ip));
119     listaDispositivos.add(new Dispositivo(ip,
        mac));
120     Log.i(TAG, "Host: " + String.valueOf(hname)
        +
121         "(" + String.valueOf(ip) + ") está
            acessível!");
122     ips += "Host: " + String.valueOf(hname) +
123         "(" + String.valueOf(ip) + ") está
            acessível!";
124     }
125 }
126 } catch (Exception e) {
127     Log.e(TAG, "Algo deu errado.", e);
128 }
129 publishProgress();
130 }
131 });
132 } // Fim do FOR que percorre os 255 IPs
133
134 threadPool.shutdown();
135 while (!threadPool.isTerminated()) {
136     //Espera o término da pool antes de continuar
137 }

```

```

138
139         //chamar a API de verificar Fabricante por MAC
140         //e também verificar as portas 23, 2323 e 48101
141         for(Dispositivo dispositivo : listaDispositivos){
142             String fabricante = MacVendorLookup.get(dispositivo.getMac())
143             ;
144             dispositivo.setFabricante(fabricante);
145             ScannerPortas scannerPortas = new ScannerPortas(dispositivo);
146             scannerPortas.statusPorta23(context);
147             scannerPortas.statusPorta2323();
148             scannerPortas.statusPorta48101();
149         }
150
151         System.out.println("Escaneamento da rede WiFi concluído!");
152     }
153     catch (Throwable t) {
154         Log.e(TAG, "Algo deu errado.", t);
155     }
156
157     return ips;
158 }
159
160 @Override
161 protected void onProgressUpdate(Void... values) {
162     super.onProgressUpdate(values);
163     total += (PROGRESSO / 100d);
164     totalAux = Math.ceil(total);
165     progressBar.incrementProgressBy(PROGRESSO);
166     String texto = String.valueOf(totalAux);
167     texto += " %";
168     textoProgresso.setText(texto);
169 }
170
171 //Retorna resultados para a thread de exibição ao usuário
172 //e seta os valores dos IPs encontrados em um ListView, com
173 //um IP por linha. Também fecha o ProgressDialog
174 @Override
175 protected void onPostExecute(String textoExibicao) {
176     super.onPostExecute(textoExibicao);
177
178     if(textoExibicao != "" && textoExibicao != null) {
179         adapterDispositivo.setDispositivos(listaDispositivos);

```

```

179         TextView nome_rede = (TextView) activity.findViewById(R.id.rede_wifi
180             );
181         if(!this.redeWifi.equals("unknown ssid"))
182             nome_rede.setText(this.redeWifi);
183         else
184             nome_rede.setText("Rede desconhecida");
185
186         TextView qtdDispositivos = (TextView) activity.findViewById(R.id.
187             qtd_dispositivos);
188         qtdDispositivos.setText(listaDispositivos.size() + "");
189
190         Log.i(TAG, "Setando resultados na tela via Async.");
191     }
192     else {
193         Log.e(TAG, "Erro ao setar o texto na tela via Async.");
194     }
195
196     progressBar.setVisibility(ProgressBar.GONE);
197     textoProgresso.setText("Rede escaneada");
198     textoProgresso.setVisibility(TextView.GONE);
199 }

```

Trecho de Código A.3 – Classe ScannerTCP.java

```

1     public class ScannerTCP {
2         Runtime runtime;
3
4         private String ip;
5
6         public ScannerTCP(String ip){
7             this.ip = ip;
8             this.runtime = Runtime.getRuntime();
9         }
10
11         //método para executar o ping TCP via runtime
12         //quando a variável valorDeSaida == 0, significa que o ping foi bem
13         //sucedido (true)
14         //caso contrário, significa que o ping não foi bem sucedido (false)
15         public boolean executarPing(){
16             try{
17                 Process processo = runtime.exec("/system/bin/ping -c 1 " + ip);
18                 int valorDeSaida = processo.waitFor();

```



```

16         Telnet telnet = new Telnet();
17         telnet.logar(dispositivo.getIp(), 23);
18         dispositivo.setUsuario(telnet.getUsuario());
19         dispositivo.setSenha(telnet.getSenha());
20         return true;
21     } catch (ConnectException c) {
22         System.out.println("[porta23] Não foi possível conectar. Erro: " + c
23             );
24         dispositivo.setPorta23Aberta(false);
25         System.out.println("[ConnectException] Dispositivo [" + dispositivo.
26             getIp() +
27             "] está com a porta 23 fechada.");
28         return false;
29     } catch (SocketTimeoutException t) {
30         System.out.println("[porta23] Não foi possível conectar. Erro: " + t
31             );
32         dispositivo.setPorta23Aberta(false);
33         System.out.println("[SocketTimeoutException] Dispositivo [" +
34             dispositivo.getIp() +
35             "] está com a porta 23 fechada.");
36         return false;
37     } catch (IOException i) {
38         System.out.println("[porta23] Não foi possível conectar. Erro: " + i
39             );
40         dispositivo.setPorta23Aberta(false);
41         System.out.println("[IOException] Dispositivo [" + dispositivo.getIp
42             () +
43             "] está com a porta 23 fechada.");
44         return false;
45     } catch (Exception e) {
46         e.printStackTrace();
47         System.out.println("[porta23] Erro: " + e);
48         dispositivo.setPorta23Aberta(false);
49         System.out.println("[Exception]Dispositivo [" + dispositivo.getIp()
50             +
51             "] está com a porta 23 fechada.");
52         return false;
53     }
54 }
55
56 public boolean statusPorta2323() {
57     try {

```

```

51         Socket socket = new Socket();
52         socket.connect(new InetSocketAddress(this.dispositivo.getIp(), 2323)
53             , 500);
54         socket.close();
55         dispositivo.setPorta2323Aberta(true);
56         System.out.println("[ConnectException]Dispositivo [" + dispositivo.
57             getIp() +
58             "] está com a porta 2323 aberta.");
59         Telnet telnet = new Telnet();
60         telnet.logar(dispositivo.getIp(), 2323);
61         dispositivo.setUsuario(telnet.getUsuario());
62         dispositivo.setSenha(telnet.getSenha());
63         return true;
64     } catch (ConnectException c) {
65         System.out.println("[porta2323] Não foi possível conectar. Erro: " +
66             c);
67         dispositivo.setPorta2323Aberta(false);
68         System.out.println("[ConnectException] Dispositivo [" + dispositivo.
69             getIp() +
70             "] está com a porta 2323 fechada.");
71         return false;
72     } catch (SocketTimeoutException t) {
73         System.out.println("[porta2323] Não foi possível conectar. Erro: " +
74             t);
75         dispositivo.setPorta2323Aberta(false);
76         System.out.println("[SocketTimeoutException] Dispositivo [" +
77             dispositivo.getIp() +
78             "] está com a porta 2323 fechada.");
79         return false;
80     } catch (IOException i) {
81         System.out.println("[porta2323] Não foi possível conectar. Erro: " +
82             i);
83         dispositivo.setPorta2323Aberta(false);
84         System.out.println("[IOException] Dispositivo [" + dispositivo.getIp()
85             () +
86             "] está com a porta 2323 fechada.");
87         return false;
88     } catch (Exception e) {
89         e.printStackTrace();
90         System.out.println("[porta2323] Erro: " + e);
91         dispositivo.setPorta2323Aberta(false);
92         System.out.println("[Exception]Dispositivo [" + dispositivo.getIp()

```

```

        +
85         "] está com a porta 2323 fechada.");
86     return false;
87 }
88 }
89
90 public boolean statusPorta48101() {
91     try {
92         Socket socket = new Socket();
93         socket.connect(new InetSocketAddress(this.dispositivo.getIp(),
94             48101), 500);
95         socket.close();
96         dispositivo.setPorta48101Aberta(true);
97         System.out.println("Dispositivo [" + dispositivo.getIp() +
98             "] está com a porta 48101 aberta.");
99         return true;
100     } catch (ConnectException c){
101         System.out.println("[porta48101] Não foi possível conectar. Erro: "
102             + c);
103         dispositivo.setPorta48101Aberta(false);
104         System.out.println("[ConnectException]Dispositivo [" + dispositivo.
105             getIp() +
106             "] está com a porta 48101 fechada.");
107         return false;
108     } catch (SocketTimeoutException t) {
109         System.out.println("[porta48101] Não foi possível conectar. Erro: "
110             + t);
111         dispositivo.setPorta48101Aberta(false);
112         System.out.println("[SocketTimeoutException] Dispositivo [" +
113             dispositivo.getIp() +
114             "] está com a porta 48101 fechada.");
115         return false;
116     } catch (IOException i) {
117         System.out.println("[porta48101] Não foi possível conectar. Erro: "
118             + i);
119         dispositivo.setPorta48101Aberta(false);
120         System.out.println("[IOException] Dispositivo [" + dispositivo.getIp()
121             +
122             "] está com a porta 48101 fechada.");
123         return false;
124     } catch (Exception e) {
125         e.printStackTrace();
126     }
127 }

```

```
119         System.out.println("[porta48101] Erro: " + e);
120         dispositivo.setPorta48101Aberta(false);
121         System.out.println("[Exception]Dispositivo [" + dispositivo.getIp()
            +
122             "] está com a porta 48101 fechada.");
123         return false;
124     }
125 }
126
127 public Dispositivo getDispositivo() { return dispositivo; }
128
129 public void setDispositivo(Dispositivo dispositivo) { this.dispositivo =
    dispositivo; }
130 }
```
