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

DDoS por reflexão amplificada usando o Mirai

Gabriel Cunha Bessa Vieira

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

Orientador
Prof. Dr. João José Costa Gondim

Brasília
2023

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

DDoS por reflexão amplificada usando o Mirai

Gabriel Cunha Bessa Vieira

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. Ricardo Pezzuol Jacobi
Departamento de Ciência da Computação

Prof. Dr. Marcelo Grandi Mandelli
Coordenador do Bacharelado em Ciência da Computação

Brasília, 13 de Fev de 2023

Dedicatória

Dedico este trabalho ao meu pai Fernando Bessa Vieira, a minha mãe Dione Pinto da Cunha, e a toda minha família, a minha namorada, a todos meus amigos que estão sempre comigo me apoiando.

Agradecimentos

Agradeço ao meu pai por tanto apoio ao longo da minha jornada. Foram muitos altos e baixos que ele se colocou no meu lugar, ouviu e me aconselhou corretamente. A minha mãe por todo o amor e por todos os conselhos, ao meu irmão por ter falado bastante comigo nesse período bem conturbado que foi esse trabalho de graduação. A minha namorada, Júlia Lustosa, que sempre me animava a fazer o TG, sempre olhando o lado positivo das coisas nas vezes que eu reclamava do trabalho de graduação. O meu grupo que se formou no primeiro semestre e vai sair da UnB comigo para a vida, Galo de Kalsa. Sem eles não sei se teria sido tão feliz na UnB, foram muitos aprendizados ao longo desse tempo. Em específico Thiago Veras, Giovanni Guidini, Vitor Dullens, André Cássio, Mikael Mello, Gabriel Karan, Fernando Grasser, Matheus Vieira e também pelas amizades frutíferas que a UnB me trouxe, em especial Artur Zorron, Isabela Harumi e a todos os outros amigos e colegas que tiveram um impacto relevante na minha trajetória dentro da Universidade. Gostaria de agradecer a todos os professores que foram meus docentes e a oportunidade de ter aprendido muitas lições com os mesmos, em especial João Gondim por ter me introduzido dentro do mundo de segurança e Alcyon Junior por ter me introduzido no mercado de segunraça cibernética e ter me ajudado absurdamente a trilhar meu caminho até segurança ofensiva no mercado profissional no início da minha carreira.

Resumo

Desde 2016 com o surgimento da *botnet* Mirai, foi possível notar o surgimento de novas variantes do Mirai, visto que o seu código foi postado em um fórum público, e hoje se encontra no Github. O Mirai por si só já é conhecido por realizar ataques complexos utilizando diversos protocolos como HTTP (*Hypertext Transfer Protocol*), DNS (*Domain Name System*), UDP (*User Datagram Protocol*), TCP (*Transmission Control Protocol*) entre outras variações de ataque do próprio Mirai. O propósito deste trabalho é implementar uma nova função dentro do Mirai para entender o comportamento de um ataque de negação de serviço por reflexão amplificada aplicada a uma *botnet*. Por meio desse ataque, dentro de um ambiente controlado, o intuito do trabalho é mostrar o poder de negação de serviço que um ataque de negação de reflexão amplificada possui em contraste com os outros ataques existentes no Mirai. O trabalho está dividido em cinco partes. A introdução onde são apresentados a motivação, justificativa e objetivo do trabalho. O segundo capítulo consiste no referencial teórico do Mirai que sustenta todos os pontos necessários para a realização do ataque de reflexão amplificada. O terceiro capítulo consiste nas instruções para a montagem do laboratório para execução do Mirai. O quarto capítulo consiste nos testes e resultados com o ataque de reflexão amplificada. Por último o quinto capítulo consiste na conclusão do trabalho.

Palavras-chave: reflexão amplificada, negação de serviço distribuída, Mirai

Abstract

Since 2016, when Mirai botnet was born, it was possible to note new variants of Mirai was being deployed into the wild, since its source code has been published on a public forum, and later on Github. Mirai itself is already well known, since it does complex attacks utilizing protocols like HTTP (Hypertext Transfer Protocol), DNS (Domain Name System), UDP (User Datagram Protocol), TCP (Transmission Control Protocol) alongside other variants that is implemented on Mirai. The purpose of this undergraduate thesis is to implement a new function inside Mirai to understand how an amplified reflection attack applied to a botnet works and how much more power it has compared to other native Mirai attacks. The thesis is divided onto five parts, which are introduction, theorical refence about Mirai, laboratory setup alongside with code alterations on Mirai source code, results and conclusion.

Keywords: amplified reflection, distributed denial of service, mirai

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

O ataque de negação de serviço por reflexão amplificada se aproveita de uma fragilidade que foi a implementação do principal protocolo da Internet, o *Internet Protocol* (IP), não foi levado em conta o fator segurança durante a transmissão dos pacotes. Inicialmente, não existia nenhum tipo de proteção robusta no protocolo que impeça que este seja forjado por um terceiro (*ip spoofing*). O ataque de negação de serviço por reflexão amplificada utiliza o protocolo UDP *User Datagram Protocol*, um protocolo leve, visto que o seu cabeçalho apenas consiste em: endereço de origem, endereço de destino, porta de origem e porta de destino. Ou seja, além do protocolo nativo da Internet não implementar proteções nativas para evitar o *spoofing*, o próprio protocolo UDP não tem nenhuma verificação se o pacote enviado realmente foi enviado por aquela máquina que diz ter enviado o pacote. Completando o pacote, temos o cabeçalho do protocolo de DNS (*Domain Network System*), responsável por resolver consultas que são feitas a partir de *hosts* utilizando como base do protocolo, o próprio protocolo UDP.

1.1 Motivação

A motivação da realização deste trabalho em específico, foi o de entender como um *malware* realmente funciona, podendo entender como a sua estrutura é montada e toda a lógica por trás da sua implementação. Com a ajuda de trabalhos anteriores [4] que detalharam todas as funções do Mirai, foi possível entender alguns mecanismos que o Mirai utiliza com o intuito de ofuscação como randomização de strings, modificação do header do executável do *bot* para não ser possível utilizar *debuggers*, ofuscação dentro do próprio sistema operacional como um nome de processo aleatório.

A escolha do Mirai em específico foi pelo simples motivo do código fonte completo, ainda que defeituoso, estar disponível livremente no Github. Caso o vírus escolhido para a implementação de um ataque só estivesse com o executável disponível, seria necessário

realizar uma engenharia completa no vírus, o que seria um estudo muito extenso e bem trabalhoso no caso de um trabalho de graduação.

1.2 Justificativa

O intuito de simular uma negação de serviço por reflexão amplificada dentro de um ambiente controlado é para entender como o ataque de negação de serviço pode ser potencializado utilizando artifícios que estão disponíveis na internet utilizando dispositivos conectados a internet que não estão seguros ou que estão com a configuração padrão de fábrica. Com o conhecimento certo, é possível escalar de tal forma que o ataque de negação de serviço tenha uma complexidade muito maior de ser evitado.

É um tipo de ataque que não necessita de muita técnica ou manipulação de baixo nível para ser realizado, mas que necessita de uma *botnet* para gerar tráfego o suficiente para deixar um serviço indisponível. O processo de inicialização, infecção do dispositivo e o seu controle são os pontos principais para a realização do ataque.

1.3 Objetivo

O principal ponto do trabalho é a implementação de um ataque de negação de serviço por reflexão amplificada utilizando módulos existentes do Mirai, verificando o seu poder destrutivo quando se trata de negação de serviço, mostrando o tempo de resposta de um serviço sendo atacado pelo módulo novo do Mirai em contraste com os antigos, e a diferença de tráfego gerado por um ataque de negação de serviço por reflexão amplificada e por outros ataques de negação de serviço que não utilizam a reflexão amplificada, ou seja, os nativos do próprio Mirai.

1.4 Organização do Texto

O trabalho será organizado da seguinte maneira

- Capítulo 2: Apresentação do quadro conceitual, tendo ali todos os .conceitos que são necessários para o entendimento do trabalho
- Capítulo 3: Abordagem de como está sendo montado o gerador de pacotes juntamente com o acoplamento na arquitetura do Mirai.
- Capítulo 4: Apresentação de testes e resultados obtivos.
- Capítulo 5: Conclusão do trabalho e considerações finais.

Capítulo 2

Quadro conceitual

Em sua maioria, os ataques de negação de serviço partem de *botnets*, que de acordo com [5], é um conjunto de máquinas que são infectadas com malware ou máquinas que são comprometidas por um atacante que deixam-nas controláveis por uma máquina mestre, comando e controle (C&C). Esta máquina mantém e controla conexões com bots e dão a possibilidade para o atacante dar comando para todas as máquinas comprometidas a partir do servidor de comando e controle.

A rigor, as máquinas podem ser infectadas de diversas formas [6], sendo elas:

- *network scanning*
  - Escaneamento de diferentes *endpoints* é feito com o intuito de descobrir possíveis dispositivos vulneráveis. Esse ataque em especial é utilizado pelo Mirai com pequenas modificações.
- *drive-by-download*
  - Download não intencional de código malicioso em uma máquina arbitrária. [7]
- *social engineering*
  - Engenharia social usa a influencia e a persuasão para enganar pessoas, convencedo-as utilizando diversas técnicas de manipulação. [8],
- *buffer overflow*
  - Estouro de buffer acontece quando um espaço em memória alocado para uma variável em memória é menor que a quantidade de dados que foi definido para a variável. [9]
- *0-day attack*

– Se trata de um ataque que é totalmente desconhecido, apenas seu autor conhece e portanto, não possui um método conhecido de se defender contra o ataque.

De acordo com [10], o design da Internet foi pensado com o intuito de manter o seu núcleo simples e deixar qualquer tipo de complexidade para os usuários de borda. Como essa complexidade desde o seu design é pensada apenas como prioridade para o usuário final. Nos roteadores por onde os pacotes passam, só é necessário que seja entregue ao destinatário o pacote IP sem conhecimento prévio de complexidade dos serviços das outras camadas de rede.

A falta de complexidade nos roteadores de núcleo resulta na falta de complexidade para implementação de aplicações mais sofistifacadas, como a implementação de um sistema de autenticação. A falta de robustez na camada de rede dá margem para o forjamento de endereços IP (*IP spoofing*), que nada mais é criar pacotes com o protocolo IP, porém com informações falsas, geralmente com o intuito de se passar por alguém que o atacante não é.

O excesso de descentralização da internet tem lados positivos e lados negativos. O lado positivo é que como não há um ponto central de falha, entretanto a parte de segurança fica por completa responsabilidade do usuário final. A falta de segurança, criptografia, autenticação e confiabilidade em endereços IPs. Esses fatores, somados a inúmeros outros que deixam o cenário ideal para o ataque de protocolos em camadas mais baixas de rede, já que não existe uma proteção muito robusta por parte dos usuários de borda.

2.1 Ataques de negação de serviço - *DoS*

Ataque de negação de serviço é um tipo de ataque que esgota recursos de conexão e processamento de um alvo, com a finalidade de impedir um usuário legítimo de acessar serviços.[11]. Este ataque é feito a partir de inúmeras requisições chegando em uma mesma máquina.

Os ataques de negação de serviço são lançados de formas distintas [12]. Um dos modos de fazer a negação de serviço é mandar pacotes que são feitos de tal forma que exploram uma vulnerabilidade dentro do protocolo ou sistema em questão. Uma forma de realizar esse ataque explorando uma vulnerabilidade de protocolo é fragmentando pacotes ICMP, mandando vários datagramas para o alvo, que fazia o sistema parar de funcionar. Outra forma é utilizando um tráfego mais volumoso que tem objetivo de esgotar os recursos que poderiam estar sendo consumidos por tráfego legitimo.

Conforme a tecnologia foi evoluindo, os mecanismos de defesa foram ficando mais complexos e sendo assim, os ataques de negação de serviço evoluiram junto ficando mais complexos. Para aumentar a complexidade desses ataques, eram necessários mais agentes

maliciosos para cumprir com esta finalidade, aumentando o tráfego do ataque por meio da amplificação, dando origem ao ataque de negação de serviço distribuída ($DDoS$).

2.1.1 Ataques de negação de serviço distrubuída - *DDoS*

O ataque de negação de serviço distribuída se trata de uma tentativa de interromper o tráfego normal de um servidor, serviço ou rede, sobrecarregando o alvo ou sua infraestrutura utilizando vários dispositivos que são coordenados para atacar uma dessas vítimas [13]. Esse tipo de ataque não depende de explorar uma vulnerabilidade específica de algum protocolo ou sistema, ele se baseia na quantidade enorme de tráfego que é gerada por múltiplas máquinas que mandam pacotes para a máquina da vítima ao mesmo tempo.

A parte de amplificação desse ataque é feita a partir de uma rede de dispositivos vulneráveis que juntos formam uma *botnet*. Uma *botnet* se refere a um grupo de computadores infectado por malware que estão sob controle de um indivíduo mal-intencionado [14]. O controle da botnet é feita por um comando e controle, que é capaz de se comunicar com todas as máquinas que estão infectadas, dado que a conexão com a máquina centralizada de comando e controle é feita. A maior parte dos computadores infectados são dispositivos de Internet das coisas (IoT), dado que a sua grande maioria tem um custo baixo e a preocupação com a segurança do produto não é uma prioridade nem do fabricante, nem de conhecimento prévio de quem opera o dispositivo.

2.2 O protocolo *domain name system* - DNS

O protocolo DNS é responsável por resolver endereços que são lidos em caracteres alfanuméricos com o seu endereço de IP real [15].O protocolo funciona por meio de um sistema de hierarquia de servidores onde cada um tem a sua função [16].

- Domínios e recursos para consultas
  - Sistema onde acontece a identificação de *hosts*. Tem uma estrutura com o Nome e para qual endereço IP o nome está apontado.
- *Name servers*
  - Servidores que armazenam informações a cerca da estrutura de domínios. São servidores autoritativos
- *Resolvers*
  - Aplicações por parte do cliente onde a informação extraída dos *name servers* a partir de uma consulta do cliente.

Funcionamento de um servidor DNS na prática:

1. Cliente faz requisição para um servidor de DNS (*resolver*)

   (a) Antes de realizar a requisição para o servidor root, é checado no próprio servidor se o endereço já está em cache

2. Requisição pro servidor root, que contém informações acerca dos TLD(*top-level domains*) como '.com', '.org' '.edu'

3. Servidor root responde para o DNS *resolver* o endereço IP de um *top-level domain* que é relevante

4. DNS*Resolver* manda requisição para o servidor TLD requisitando a página

5. Servidor TLD responde com o endereço IP de um servidor autoritativo com o *domain name server* domínio específico, no caso example.com

6. O servidor autoritativo agora sim manda o endereço IP de um subdomínio de outro servidor autoritativo, no caso beta.example.com

7. O servidor não possui nenhum subdomínio cujo nome tenha 'download' antes de beta.example.com. Então é retornado um erro para o usuário.

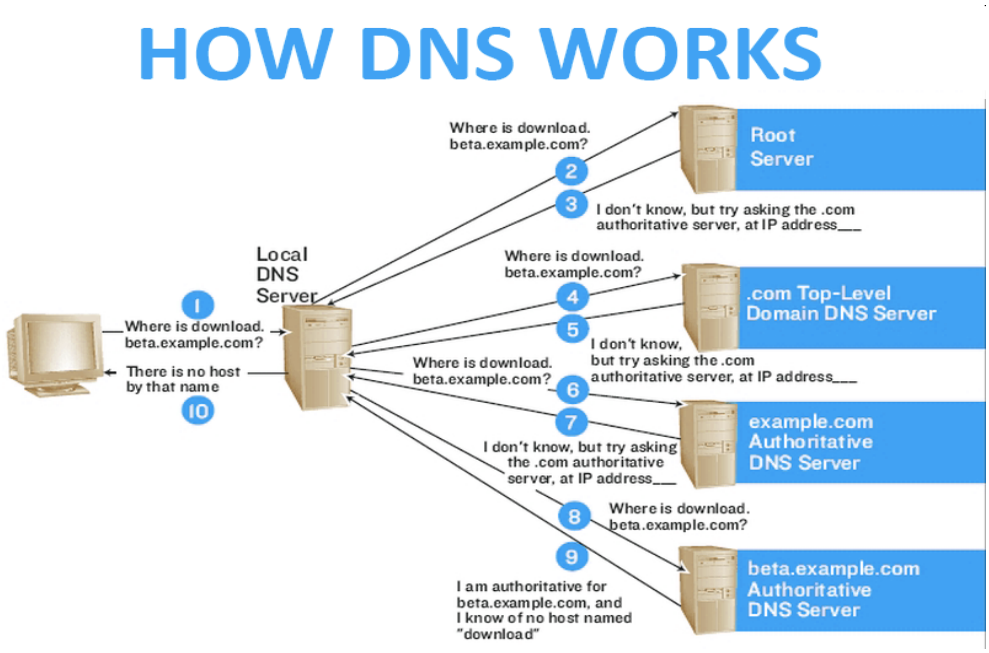

Figura 2.1: Funções de um servidor DNS

2.3 Ataques de negação de serviço por Reflexão Amplificada

Ataque de negação de serviço por reflexão amplificada explora uma funcionalidade dos resolvedores de DNS abertos, com o intuito de sobrecarregar o alvo com uma quantidade amplificada de tráfego, que é gerada pelo servidor DNS[17].

Conforme a Internet foi evoluindo, diferentes soluções ganharam tração, se fazendo mais necessário o aumento do número de servidores de DNS. Como o DNS é baseado em cima do UDP (*User Datagram Protocol)*, ele não tem um estado de conexão, ou seja, não é necessário validar se há uma conexão como ocorre no TCP (*Transmission Control Protocol*). Pelo fato de não exigir a checagem de conexão, é possível que agentes maliciosos realizem ataques sem revelar a sua verdadeira identidade.

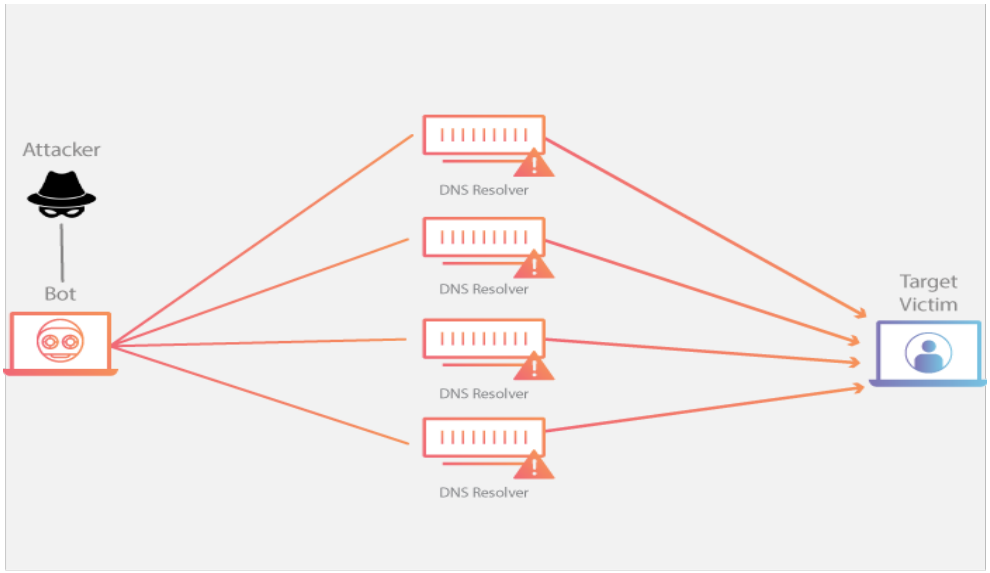

Figura 2.2: Amplificação por reflexão

O ataque tem duas etapas definidas [15]. Antes dos refletores realizarem as requisições, é necessário que mas máquinas de onde sairão os datagramas mudem o seu endereço de origem para o endereço de IP da vítima em questão. Depois de feita a mudança do endereço de origem, as requisições DNS são enviadas para servidores DNS com o parâmetro de consulta ANY. O padrão de consulta ANY permite que a resposta do refletor seja a maior possível, visto que todos os registros disponíveis dentro de um domínio são encapsulados no datagrama. Quando o servidor DNS recebe as requisições e processa a consulta ANY, são amplificados pacotes com todos os registros são redirecionados para a vítima, como não há nenhum tipo de checagem de segurança no servidor DNS, ele apenas repassa o datagrama. Como a vítima fica com inúmeros pacotes de DNS legítimos dentro de sua rede, ocorre a saturação da rede e, por consequência, a rede começa a ter

mais lentidão e eventualmente há uma indisponibilidade pela falta de recurso para gerir a quantidade de tráfego produzida pela amplificação.

A amplificação por reflexão é um ataque simples porém muito efetivo, visto que os atacantes não utilizam de muito recurso para gerar grandes impactos. A maior dificuldade do atacante é encontrar servidores DNS desprotegidos que aceitem a requisição com a consulta ANY. Além disso, o ataque é camuflado por inúmeros endereços IPs distintos, visto que cada DNS tem o seu próprio endereço e o ataque não parte diretamente do atacante e sim dos refletores.

2.4 Mitigação de ataques de negação de serviço

Caso não tenha nenhuma proteção contra as inúmeras requisições sendo feitas na máquina alvo, não tem nada melhor do que desconectá-la da internet. Os ataques de negação de serviço tendem a consumir muitos recursos e esgotá-los muito rápido [18]. Um dos grandes problemas de se defender contra ataques de negação de serviço se dá pelo fato dos atacantes usarem endereços IP forjados, o que dificulta a sua identificação e o seu bloqueio [19].

Para se defender do excesso de requisições, é necessário entender o que é tráfego legítimo e o que é tráfego malicioso. Com isso, as políticas para descartar pacotes que não são legítimos precisam estar bem escritas para funcionarem e não descartarem tráfego legítimo [20]. Para diferenciar esse tráfego e descartá-lo, são criadas regras específicas para coletar informações a respeito da quantidade de pacotes trocados entre dois IPs distintos a estrutura dos pacotes e , essa forma de identificação de tráfego não legítimo se chama diferenciação semântica de tráfego. Outras formas de mitigar o ataque de negação de serviço [21], é possível utilizar diversas técnicas, como a diferenciação do tipo de serviço (*ToS header*), já que é um campo pequeno de 1 *byte*, que os pacotes recebidos estão sendo utilizados, e a partir disso realizar uma filtragem dos pacotes. Outra forma é o estabelecimento de filas para diferente tipos de pacotes para cada tipo de serviço que está sendo requerido. Outra forma de mitigar, de certa forma, o ataque de negação de serviço é utilizando o *rate limit* em roteadores ou em próprios serviços, descartando o excesso de pacotes que estão sendo mandados. Por último, uma das formas mais eficientes é delegar essa tarefa a serviços especializados em lidar com esses tipos de ataques diariamente, como faz a Cloudflare ou a Akamai. Estes serviços geralmente utilizam uma rede própria que possuem nós no elo mais externo da rede, que são responsáveis por receber os ataques e por trás um balanceador de carga distribui o ataque por diversas máquinas.

2.5 Ataque de negação de serviço terceirizada - *DDoS for Hire*

Ataques de negação de serviço ficaram cada vez mais comuns, com isso, foi vista uma possibilidade de mercado em cima desse nicho de ataque, oferecendo-o como serviço. Pessoas que oferecem esse ataque como uma forma de serviço são chamadas de *booters* [5]. Os *booters*, são grupos especializados na oferta desses serviços, possuindo uma infraestrutura, que geralmente se trata de uma botnet, com inúmeras máquinas comprometidas, e a depender a infraestrutura, cada máquina pode ter uma função diferente.

A motivação desses ataques tem lados diferentes, de um lado a motivação é financeira, que varia conforme o tamanho do ataque, onde o hacker que está vendendo o serviço, do outro lado pode-se ter vários motivos, como os citados na seção 1.1 , o problema está na facilidade de se contratar um serviço de negação de serviço sem ter conhecimento técnico prévio, apenas executando ferramentas.

O próprio Mirai causou diversos incidentes [6] dentro do mundo corporativo. O que pode ter facilitado bastante, pode ter sido justamente o fornecimento do Mirai como serviço de *DDoS for Hire*, dentre eles KrebsonSecurity, OVH ISP, DYN, Liberia, Lappeenranta, WikiLeaks e bancos russos.

2.6 O Mirai

Por definição, *malware* é qualquer tipo de código que é adicionado, modificado ou removido de um sistema com o intuito de causar danos ou mudar a função do sistema [22]. *Malware* é a combinação de duas palavras '*malicious*' e '*software*', usado para se referir a softwares maliciosos. Como existem diversos tipos de *malwares* espalhas pela Internet, é importante entender a classificação dos mais conhecidos [23]:

- *Ransomware*: É um tipo de malware que usa criptografia de ponta com o intuito de tirar completamente o controle da máquina da vítima cifrando todos os arquivos e dados até que seja pago um resgate para que seja entregue uma chave para descriptografar os arquivos.
- *Fileless Malware*: Inicialmente é um tipo de malware que não instala nenhum tipo de arquivo, mas faz uma modificação nos arquivos nativos do sistema operacional.
- *Trojan*: Uma família de *malware* que se disfarça de código ou software legítimo. Geralmente está escondido atrás de aplicações pirateadas, emails com *phising*.
- *Virus*: Código que é inserido dentro de uma aplicação que depende da interação do usuário para que este seja executado.

- *Rootkits*: *Malwares* que após a infecção, dá acesso total ao computador da vítima, geralmente com privilégios de administrador.
- *Worms*: Esse tipo de *malware* explora vulnerabilidades para se instalar dentro de redes. Após sua instalação, estes são capazes de se replicarem dentro da rede e se espalharem por outras redes.

O Mirai foi codificado com a intenção de exfiltrar dados de uma máquina, abrir portas dentro de uma rede arbritrário que anteriormente não estavam abertas, tudo isso levando em conta que o dispositivo vulnerável seria controlado pelo Mirai após a invasão. Tendo base esse contexto de classificação de *malwares*, o Mirai está em uma família específica de *malware* que são os *worms*.

Malwares possuem inúmeras formas de se propagar, o Mirai em específico se propagava pela rede. Para se propagar, o Mirai fazia um processo de escanemanto, onde enviava *probes* TCP SYN para endereços de IP pseudoaleatórios, onde procurava por portas abertas com o serviço telnet configurado 23 e 2323 [10]. Quando um dispositivo com esse serviço aberto era encontrado, havia uma lista pronta com usuários e senhas para realizar um ataque de força bruta. Após o login, um programa separado denominado de *loader* infectava os dispositivos enviando versões já compiladas e executáveis do Mirai de acordo com a arquitetura da máquina. Para realizar sua ofuscação, o Mirai deletava o próprio binário depois de executado e ofuscava o nome do processo com uma string aleatória. Se tratando de persistência, o Mirai escaneava por outros processos que usavam a porta 22 ou 23 e excluia os processos que estavam rodando antes dele se estabelecer na telnet e comunicar com outros servidores da infraestrutura da botnet pela porta 22.

Se tratando do Mirai, foi necessário entender como o código funcionava e sua arquitetura como um todo para introduzir um novo módulo de ataque. Seguindo as análises estáticas e dinâmicas previamente realizadas [4], foi possível entender como o Mirai se comportava de maneira mais objetivo, visto que já era sabido qual era o comportamento esperado do Mirai dentro de um laboratório controlado.

2.7 Arquitetura do Mirai

Levando em conta o código fonte do Mirai, ele é dividido em quatro principais pastas, onde cada pasta desempenha uma função específica dentro do Mirai.

- **/mirai**: Possui os subdiretórios bot, cnc e tools, um script shell build.sh e o prompt.txt. São os respectivos diretórios e arquivos necessários para o completo funcionamento do Mirai, com as devidas alterações feitas no código que serão explicadas nas próximas subseções.

- /bot: A função do código nesse subdiretório é de varrar a rede a procura de novos dispositivos vulneráveis a partir de IPs pseudoaleatórios que são potenciais alvos para serem invadidos e transformados em *bots*. Dentro do Mirai, sua função é de servir como o primeiro bot de dentro da rede. Neste subdiretório, o código está escrito em C. O código é responsável por resolver o DNS, armazenar os ataques que estão implementados no Mirai, reimplementação de função já existentes em outras bibliotecas para deixar o código enxuto, geração de valores aleatórios, matar outros processos que estejam utilizando a mesma porta que o Mirai irá utilizar, mecanismo de varredura do bot e a implemetação de protocolos para ataques.

- /cnc: Se comportar como o servidor principal que comanda todos os *bots* da rede do Mirai. Sua função é direcioná-los para um alvo em específico com o intuito de fazer o ataque de negação de serviço distribuído. Neste subdiretório, o código está escrito na linguagem Golang. O código é responsável por armazenar informações a respeito das variáveis que são utilizadas nos arquivos, definir variáveis importantes para o banco de dados, comunicação do cliente com a API e o código da interação entre cliente e servidor.

- /tools: Código auxiliar que é responsável por diferentes funções que complementam o Mirai, seja na configuração de arquivos, ofuscação de código fonte, dificultar a engenharia reversa do código fonte, implementação de arquivos a partir do protocolo HTTP.

- build.sh: Compilar os códigos fonte armazenados nos subdiretórios.

- prompt.txt: Apenas responsável por enviar uma mensagem de boas-vindas para o bot com uma mensagem em russo.

- **/loader**: Possui dois subdiretórios e dois scripts. Essa parte do bot é responsável por realizar a infecção de uma máquina vulnerável, carregando um binário do *bot* na máquina vulnerável.

  - /bins: Possui os códigos executáveis em diferentes arquiteturas que serão carregados na máquina vulnerável que for invadida pelo Mirai.

  - /src: Contém o código fonte do servidor Loader, que é responsável pela infecção de máquinas vulneráveis ao Mirai. Seu código está escrito apenas em C. Como no servidor de Comando e Controle, neste caso também é possível notar a implementação de funções que auxiliam a complementar esse módulo

e a deixar o este módulo mais enxuto. Além disso, é possível notar programas que auxiliam os bots na leitura do protocolo alvo do *bot*, que é o telnet. Recuperação de binários executáveis que estão localizados no subdiretório /bin e que serão injetados na máquina vulnerável com o auxílio de funções que fazem a conexão com a máquina vulnerável e com o servidor base do Mirai.

  - build.sh/debug.build.sh: Responsável por compilar os códigos fontes do subdiretório /src, onde é possível habilitar um modo mais verboso, de modo que quem esteja modificando o bot tem um melhor entendimento do que está acontecendo durante a compilação.

- **/dlr**: Os códigos desse diretório está diretamente associado ao código do /loader. Possui um código que gera códigos executáveis para arquiteturas diferentes que realizam a mesma função.

  - main.c: Este programa possui funções responsáveis pela recuperação de binários. Sua implementação permite detectar a presença do *wget* no cliente. Em caso negativo, será carregado um programa similar ao wget que se encontra nesse arquivo, o *bot* será carregado na máquina vulnerável e a máquina irá virar um *bot*. Basicamente faz a mesma função do programa *wget.c*, porém funciona como uma etapa a mais caso a máquina infectada não recupere o binário do próprio servidor.

  - build.sh: Responsável por compilar o código fonte em diversas arquiteturas diferentes, e o executável é armazenado posteriormente no subdiretório /release

2.8 Síntese

Neste capítulo abordamos de forma concisa os principais conceitos que dizem a respeito de negação de serviço, negação de serviço distribuída, possíveis formas de mitigação de ataques de negação de serviço, serviços terceirizados que o próprio Mirai oferecia, conceito do próprio Mirai e sua taxonomia de acordo com a forma de infecção e por fim sua arquitetura, detalhando a função de cada arquivo dentro do contexto geral do Mirai. No próximo capítulo, serão mencionadas todas as etapas de configuração do laboratório, como a função de cada máquina na arquitetura do Mirai, a criação da rede isolada sem conexão com a Internet, alterações necessárias no Mirai para a execução do ataque de negação de serviço por reflexão amplificada, configuração do servidor DNS da rede virtual e instalação de ferramentas necessárias dentro de cada máquina que seriam necessárias na compilação do código fonte do bot e do servidor de comando e controle.

Capítulo 3

Implementação do ataque de negação de serviço distribuída por reflexão amplificada

O objetivo deste trabalho é a implementação do ataque de reflexão amplificada abusando o DNS no Mirai. Assim, foram consideradas duas estratégias: a primeira foi a inclusão do código da geração de pacotes DNS; a outra foi a adaptação de ataques já existentes no Mirai que utilizam datagramas DNS.

Inicialmente, o intuito do código que foi implementado, era realizar a montagem de um datagrama DNS, com o endereço de origem sendo o endereço da vítima, e o endereço de destino o endereço do servidor DNS que estivesse vulnerável a um ataque de amplificação por reflexão, que nesse trabalho, foi também configurado sem nenhuma diretiva de segurança que mitigasse o ataque. A razão dessa implementação, foi para facilitar a simulação em um ambiente vulnerável. Essa implementação do gerador de pacotes DNS, como já explicitado, tinha por objetivo a montagem de um pacote DNS utilizando os protocolos [1], UDP [2] e DNS [3]. Essa codificação foi feita na linguagem C, visto que o módulo de ataque do Mirai também estava escrito em C. A partir do momento que essa implementação estivesse funcionando, a ideia era integrar o código dentro do Mirai como um novo módulo de ataque.

Entretanto essa estratégia não teve sucesso, pelo fato do código, ainda que na mesma linguagem, não estava compatível com o que o Mirai estava chamando nos outros arquivos na compilação. Como o Mirai possuia suas próprias funções e as bibliotecas já pré-selecionadas de acordo com a sua arquitetura, a decisão foi de não inserir um código totalmente novo com chamadas de bibliotecas que não estavam sendo executadas em nenhum outro arquivo, se não o arquivo que possuia o módulo de ataque novo. É importante mencionar que esta decisão foi tomada após entender que um dos módulos do

Mirai, implementava parcialmente o ataque a um servidor DNS, só era necessário algumas adaptações para que o módulo de ataque em questão realizasse uma reflexão amplificada. O módulo em questão é o *attack_udp.c*. A partir do momento que isso foi observado, a implementação inicial do gerador de pacotes de DNS foi descartada, seguindo para a implementação do código de reflexão amplificada utilizando um módulo já existente no Mirai.

A segunda estratégia teve êxito, sendo sua implementação descrita em mais detalhes nas próximas seções.

3.1 Requisitos e restrições do Mirai

Por se tratar de um *worm* que poderia se espalhar pela rede rapidamente caso esta não estivesse protegida, ou caso o *worm* não estivesse corretamente isolado dentro de um ambiente controlado, foi necessário ser cauteloso com a execução do Mirai. Por este motivo, foi necessário a utilização de máquinas virtuais, que são máquinas responsáveis por simular um ambiente controlado que está contido dentro de um ambiente virtualizado. Para qualquer *malware* conseguir sair do ambiente virtualizado e entrar no ambiente real do hospedeiro, é necessário explorar uma falha da máquina virtual, e o *malware* em questão que está sendo análisado não é capaz de explorar tal vulnerabilidade, caso ela existisse.

Como o Mirai realizava infecção de dispositivos IoT (*Internet of Things)*, fazia-se necessário do seu código executável ser pequeno, com o intuito dos dispositivos conseguirem executá-lo, no caso dispositivos que se comunicariam com um C&C e realizariam ataques a partir do comando que foi pré-estabelecido, uma vez que as máquinas infectadas podem estabelecer uma conexão direta com o atacante. Portanto a estratégia de utilizar inserir um código totalmente novo dentro de um *malware* que já necessitava ser simples, não é uma tarefa tão simples, uma vez que o próprio Mirai implementava algumas próprias funções nativas do C, com o intuito de não chamar uma biblioteca completa apenas para utilizar poucas funções dentro de uma grande variedade de funções que uma biblioteca possui. O resultado disso, foi a utilização de um código já existente no Mirai, entretanto que foi adaptado à realidade do trabalho com o intuito de testar uma negação de serviço por reflexão amplificada.

3.2 Decisões de projeto

Para a simulação do ambiente do Mirai, foi necessário utilizar um ambiente controlado, visto que como se tratava de um vírus que atuava na camada de redes, é possível que se perca o controle do vírus, uma vez que ele escaneia inúmeros IPs ao mesmo tempo

buscando novas vítimas. Ainda que o código fonte fosse alterado, não foi descartada a opção de fazer a simulação da forma mais correta que seria dentro de um ambiente controlado, portanto este trabalho foi feito inteiramente utilizando o VirtualBox.

A implementação do ataque de negação de serviço por reflexão amplificada no Mirai é feita a partir da alteração do código fonte do arquivo onde se encontra os ataques que utilizam como base o protocolo UDP. Sabendo que esse é o protocolo que o DNS utiliza para consultas e respostas, foi decidido que este seria o módulo do Mirai que seria modificado.

Inicialmente o gerador de pacotes DNS seria implementado como um módulo separado no Mirai, contando com uma entrada nova no código do *bot*, e do servidor de comando e controle. Entretanto, a implementação que anteriormente já estava funcionando corretamente, não pôde ser integrado facilmente no Mirai, visto que utilizava bibliotecas que não estavam sendo importadas pelo Mirai. Portanto, a implementação de um gerador de pacotes DNS no Mirai foi descartada. Analisando o código do Mirai, em especial os módulos de ataque, é possível observar que o Mirai já implementava um módulo de ataque de *DNS Water Torture*, onde os cabeçalhos dos protocolos já estavam sendo definidos como no gerador de pacotes anteriormente implementado, porém desta vez utilizando a arquitetura do Mirai. Com o esqueleto inicial do ataque que é basicamente a formação do pacote, foi necessário apenas alterar os parâmetros como endereço de destino e de origem no cabeçalho IP, a porta que seria utilizada como origem e o tipo de consulta no servidor DNS, já que o ataque de *DNS Water Torture* fazia uma consulta do tipo **A**, e para o ataque de reflexão amplificada o tipo de consulta teria que ser do tipo **ANY**. Com essas alterações pontuais no código fonte do Mirai na parte que se refere ao ataques utilizando protocolos UDP, foi possível montar o ataque de reflexão amplificada a partir do ataque anterior.

É importante salientar que esse tipo de ataque só é possível dentro de um servidor DNS recursivo que esteja aberto aceitando consultas do tipo **ANY**, portanto, o servidor implementado dentro do ambiente de virtualização não tinha restrição em relação ao tipo de consultas, pois o intuito do trabalho é demonstrar como o ataque de negação de serviço por reflexão amplificada pode escalar exponencialmente a medida que se aumenta a quantidade de *hosts* realizando consultas no servidor DNS se passando por uma mesma vítima.

3.3 Implementação no Mirai

O ataque parte de um host malicioso que realiza o *spoofing* do endereço de origem que é anexado ao pacote, indicando o endereço IP da vítima. Uma requisição para o servidor

DNS possui inúmeros campos que precisam ser definidos, dentre eles, o campo mais importante para esse ataque é o tipo de código da requisição. Cada tipo de requisição exige um código de requisição diferente, a depender da funcionalidade que está sendo requisitada pelo cliente ao servidor DNS. No caso do ataque de amplificação por reflexão, como o intuito é deixar o pacote com o maior número de bytes possível, o código da requisição utilizado é o ANY, que extrai todas as tabelas de zona que estão em um servidor DNS, aumentando assim a quantidade de informação dentro de um pacote, e por consequência aumentando o número de *bytes* do pacote.

O Mirai possui inúmeros módulos de ataque com diferentes finalidades, um deles é o módulo de DNS, dentro das variações de ataques utilizando o protocolo UDP. Entretanto, este módulo não está implementado com reflexão amplificada, e sim com um servidor DNS padrão que será o alvo da rede para um possível ataque de negação de serviço. Sabendo que o Mirai não possui um módulo de reflexão amplificada para DNS, o módulo já existente de DNS foi alterado para simular um ataque de reflexão amplificada.

3.3.1 Reflexão amplificada no Mirai

O intuito da implementação de um gerador de pacote DNS é para que o Mirai tenha o seu próprio módulo de ataque no que se diz respeito a reflexão amplificada. O Mirai não possui um módulo para o ataque de reflexão amplificada. Entretanto, dentro do arquivo que se encontra no caminho **mirai/bot/attack_udp.c** existem diferentes tipos de ataques utilizando a suíte de protocolo UDP. Um dos ataques implementados é o *DNS Water Torture* [4], que em resumo, é um ataque que visa atacar o servidor DNS dentro de um domínio especificado dentro do ataque. A partir do momento que o DNS primário deste domínio é encontrado, ele fica marcado e consequentemente, é atacado por um tempo que é determinado quando ainda se está lançando o ataque.

Para que o ataque seja corretamente implementado, é necessário seguir recomendações de implementações que estão descritas em *request for comments* dos dois protocolos (IP e UDP) e do DNS, com o intuito de saber como é feita uma consulta ao servidor.

A implementação do gerador de pacote DNS é dividida em três etapas, visto que o esqueleto do pacote DNS [3] é composto por três cabeçalhos principais: IP, UDP e DNS. Todos estes cabeçalhos possuem tamanhos variados de acordo com o conteúdo que está definido no cabeçalho.

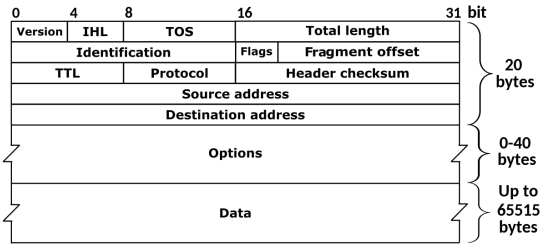

Figura 3.1: Cabeçalho IP [1] (Fonte: [24]).

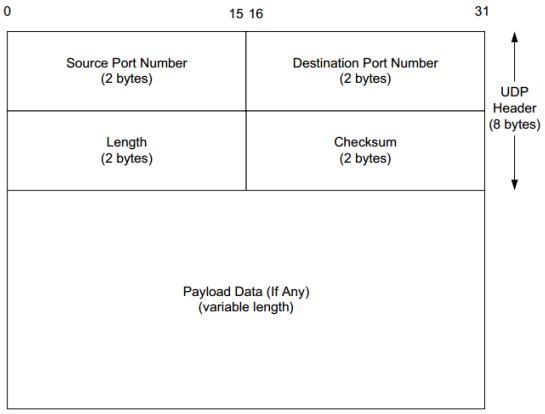

Figura 3.2: Cabeçalho UDP [2] (Fonte: [24]).

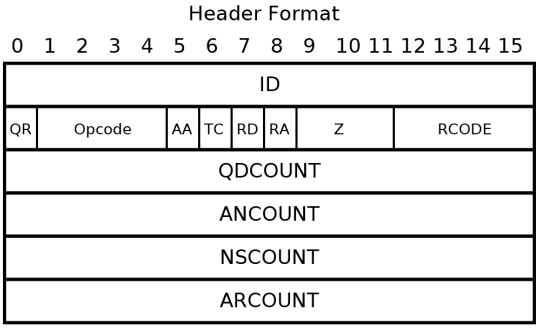

Figura 3.3: Cabeçalho DNS[3] (Fonte: [24]).

1. **Cabeçalho IP**

   - *Version*
     - Referente a versão que o protocolo IP será utilizado. Possui o tamanho de quatro bits.

  - Valor final: 4

- IHL (*Internet Header Length*)
  - Se refere ao tamanho do cabeçalho. É incrementado por 4 bytes. Possui um valor mínimo de 5 e máximo de 15, visto que o tamanho máximo de um pacote são 20 bytes e o máximo são 60.
  - Valor final: 5

- ToS (*Type of Service*)
  - Se refere ao tipo de tratamento que o pacote receberá. Possui oito bits de tamanho. Nessa implementação, não é necessário nenhum tipo de tratamento diferente, então o pacote será tratado como um pacote padrão.
  - Valor final: 0

- *Total Length*
  - Se refere ao tamanho do pacote como um todo, juntando o cabeçalho e os dados que serão carregados no pacote. Possui o tamanho de dezesseis bits.
  - Valor final: Soma dos tamanhos dos cabeçalhos no que diz respeito aos pacotes IP, UDP e DNS.

- *Identification*
  - Valor identificador que é atribuído por quem manda o pacote. Esse cabeçalho é necessário caso ocorra fragmentação do pacote. Possui tamanho de dezesseis bits.
  - Valor final: Genérico

- *Flags*
  - Valor relacionado a fragmentação do pacote. Possui dois bits reservados. Como esse valor é opcional, ele não foi definido dentro do pacote.
  - Valor final: N/A.

- *Fragment Offset*
  - Valor relacionado a fragmentação do pacote. Possui treze bits reservados. Como esse valor é opcional, ele não é definido dentro do pacote.
  - Valor final: N/A.

- TTL (*Time to Live*):
  - Indica quanto tempo o pacote permanecerá ativo. Caso esse tempo limite seja atingido, o datagrama é descartado. Possui oito bits reservados
  - Valor final: 64

- *Protocol*
  - Protocolo que será utilizado dentro do pacote. Possui oito bits reservados. Será utilizado o protocolo UDP, tendo em vista que o serviço de DNS utiliza esse protocolo como base na porta 53.
  - Valor final: IPPROTO_UDP
- *Header Checksum*
  - Valor opcional, cuja função é verificar se não houve nenhuma modificação do datagrama desde a sua origem. Possui dezesseis bits reservados.
  - Valor final: Genérico.
- *Source Address*
  - Valor que indica o endereço de origem, de onde o datagrama está saindo. Esse valor será um dos valores adulterados, visto que o intuito é realizar o *spoofing* (referência sobre spoofing aqui) de um servidor DNS. Possui trinta e dois bits reservados
  - Valor: Endereço do servidor DNS, no caso 192.168.220.10
- *Destination Address*
  - Valor que indica o endereço de destino do datagrama. Esse valor será um dos valores adulterados, visto que o intuito é realizar o spoofing (referência sobre spoofing aqui) do endereço da vítima. Possui trinta e dois bits reservados.
  - Valor: Endereço IP da vítima, no caso 192.168.220.9

2. **Cabeçalho UDP**

- *Source Port*
  - Valor que indica a porta de onde o pacote está saindo na origem. Possui dezessesis bits reservados
  - Valor final: 80
- *Destination Port*
  - Valor que indica a porta onde o pacote será enviado. Dado que o serviço usado é o DNS, então será utilizada a porta padrão do DNS como destino
  - Valor final: 53
- *Length*
  - Tamanho do pacote como um todo.

  - Valor final: Tamanho do datagrama IP e o que foi definido dentro do datagrama UDP

- *Header checksum*
  - Valor opcional, cuja função é verificar se não houve nenhuma modificação do datagrama desde a sua origem.
  - Valor final: Genérico

3. **Cabeçalho DNS**

- ID
  - Valor que representa o identificador da requisição DNS. Pode ser um valor arbitrário. Possui tamanho de 16 bits.
  - Valor final: Genérico
- QR
  - Campo que identifica que a requisição é uma consulta(0) ou uma resposta(1). Possui tamanho de 1 bit.
  - Valor final: 0
- OPCode
  - Representa que tipo de consulta será feita na mensagem. Possui tamanho de quatro bits.
  - Valor final: 0
- AA (*Authorirative Answer*)
  - Valor que aparece apenas em respostas de servidores DNS, indicando se o servidor é uma autoridade para o domínio em questão. Possui tamanho de um bit.
  - Valor final: 1
- TC (*Truncation*)
  - Especifica se a mensagem foi fragmentada. Possui tamanho de um bit
  - Valor utilizado: 0
- RD (*Recursion Desired*)
  - Especifica se será necessária a utilização de recursão. Possui tamanho de um bit
  - Valor utilizado: 0
- RA (Recursion Available)

  - Especifica se a utilização de suporte a consultas recursivas estão disponíveis no servidor. Possui tamanho de 1 bit.
  - Valor final: 0
- Z:
  - Reservado para o futuro. Possui tamanho de quatro bits.
  - Valor final: 0
- RCODE (*Response Code*)
  - Valor que vem acoplado a resposta do DNS, após uma consulta. Possui tamanho de quatro bits.
- QDCOUNT
  - Valor que representa o número de campos na seção question do DNS. Possui tamanho de dezesseis bits.
  - Valor final: 1
- ANCOUNT
  - Valor que representa o número de registros na parte da resposta. Possui tamanho de dezesseis bits.
  - Valor final: 0
- NSCOUNT
  - Valor que representa o número de name servers dentro dos registros na seção de registros autoritativos. Possui valor de dezesseis bits.
  - Valor final: 0
- ARCOUNT
  - Valor que representa o número de registros na seção de registros adicionais. Possui tamanho de dezesseis bits.
  - Valor final: 0
- QNAME
  - Nome de domínio é representado como uma sequência de bytes, terminando a string com o byte 0x00.
  - Valor final: tg.local
- QTYPE
  - Especifica o tipo de consulta que será feita
  - Valor final: ANY (255)
- QCLASS

- Especifica a classe da consulta
- Valor final: IN

3.4 Configurando o ambiente

Nessa seção será descrita como o ambiente controlado foi montado para que o *malware* fosse corretamente simulado. Esse ambiente em específico não tinha conexão externa a internet, as únicas vezes que essa conexão foi feita foi com o intuito de instalar pacotes externos que foram utilizados para a configuração do laboratório.

As máquinas foram montadas com o intuito de simular todos os módulos do Mirai operando concomitantemente. Dado a impossibilidade de inserir dispositivos IoT no modelo adotado por este trabalho, apenas a virtualização foi utilizada. Dessa forma, foi possível fazer a análise do código do Mirai e sua análise em tempo de execução.

A máquina principal utilizada para montar o laboratório tem as seguintes configurações:

- Memória RAM - 16Gb
- Processador - Intel i7 5ª geração
- Armazenamento - SSD de 512Gb
- Sistema operacional - Windows 10
- Software de virtualização - Virtual Box 6.1

3.4.1 Configuração das máquinas virtuais

Configuração da rede interna

A configuração da rede é uma das partes bem complicadas do Mirai, foi usado como base o trabalho [4] para a implementação do ambiente, com algumas alterações pela diferença da versão do sistema operacional. Sua topologia se encontra na figura 3.4.

Sabendo que a simulação que será executada é de um *malware* que se multiplica pela rede, uma vez perdida o controle sobre a multiplicação do *malware*, poderia ter que arcar com prejuízos, possíveis consequências judiciar. É extremamente importante que essa rede seja completamente isolada da Internet, para que não cause nenhum dano, perca de dados, ou qualquer tipo de acesso não autorizado a qualquer outra máquina que não seja as do laboratório.

Dentro do VirtualBox [25], é possível definir IPs fixos dentro de um intervalo fixo dentro da rede. Abrindo o gerenciador de redes do hospedeiro no VirtualBox, é possível

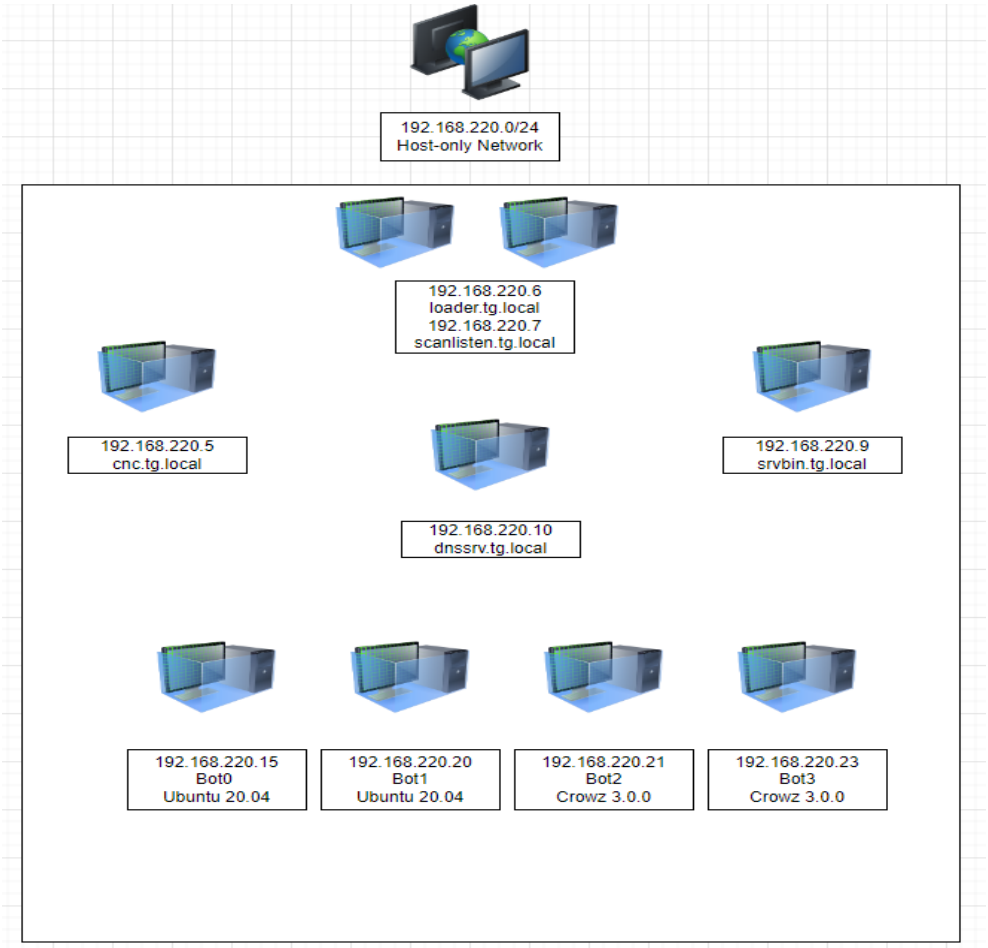

Figura 3.4: Topologia da rede do laboratório

criar uma rede, nomeá-la, atribuir o endereço IP, habilitar servidor DHCP e definir a máscara de rede que será utilizada. As configurações de redes que foram utilizadas foram as seguintes:

- Endereço IP do *gateway*: 192.168.220.1
- Máscara de Rede IP: 255.255.255.0
- Faixa mínima de endereçamento: 192.168.220.3
- Faixa superior de endereçamento: 192.168.220.254

Para essa rede funcionar, todas as máquinas que foram criadas foram inseridas dentro dessa rede interna controlada que não tem acesso a internet.

Servidor de Comando e Controle

É o principal responsável por comandar toda a *botnet* que está a sua disposição, realizar serviços de *ddos for hire* e mantém o registro das máquinas que já foram previamente

infectadas. Dado suas três funções principais, a configuração do servidor de comando e controle ficou a seguinte:

- Sistema Operacional *Ubuntu Server 20.04 LTS*
- 4096 Mb de memória RAM
- 20Gb de armazenamento virtual, podendo ser expansível

Para configurar o servidor de comando e controle, foi necessário acesso à Internet. Sempre que algum servidor precisava de ter acesso à internet para baixar pacotes, era certificado que a máquina em si não estava com o Mirai configurado e executando. Esse acesso a internet possível quando trocava a máquina da rede interna isolada para rede NAT que vem configurada por padrão para ter acesso a Internet dentro do VirtualBox.

Todos os comandos foram executados como *root* do sistema, sem privilégios de administrador, não é possível executar o Mirai, nem fazer sua *build*. Seguem as etapas de configuração do servidor de comando e controle:

1. De acordo com [26], o servidor de comando e controle precisa de ter o gcc, eletric-fence, git e o golang para executar normalmente. Entretanto o pacote nativo que é instalado pelo Ubuntu 20.04 não é o apropriado para a execução do Mirai pelas bibliotecas que são usadas no código fonte do servidor de comando e controle, portanto a instalação do golang será feita de forma separada. Para instalar os três primeiros programas que serão utilizados na configuração do Mirai:
   ```
   # sudo apt-get install gcc electric-fence git -y
   ```

2. Seguindo o fluxo do servidor de comando e controle, é necessária a instalação de um banco de dados. O banco de dados que é usado no Mirai é o *MySQL*, que é instalado a partir do seguinte comando:
   ```
   # sudo apt-get install mysql-server mysql-client -y
   ```
   No decorrer da instalação do banco de dados, será necesário definir um usuário de administrador juntamente com a senha. O mais correto é deixar ambos os dados idênticos aos que estão em código para o Mirai fazer a conexão de forma correta no banco de dados.

3. Recuperação do código fonte do Mirai que está disponível em [27]:
   ```
   # git clone https://github.com/jgamblin/Mirai-Source-Code
   ```

4. Feita a recuperação do código fonte do Mirai, foi necessária a instalação da linguagem Go, utilizada na versão 1.17.5

```
# wget https://storage.googleapis.com/golang/go1.17.5.linux-amd64.tar.gz
```

Este arquivo comprimido possui a versão compactada no que diz respeito a versão 1.17.5 da linguagem Go. O próximo passo seria criar um diretório para descompactar o arquivo baixado.

```
# mkdir golang
```

Após a criação do diretório, é feita a descompactação do arquivo.

```
# tar -zxvf go.1.17.5.linux-amd64.tar.gz -C /golang
```

5. Feita a descompactação do tradutor da linguagem Go, foi necessário definir os valores das variáveis de ambiente que são exigidos pela própria linguagem. Como o Mirai está salvo no diretório de root, é importante que essa alteração no arquivo **.bashrc** seja feito dentro do diretório de **/root**. Para definir as variáveis do Go em inicialização, os comandos dentro do **.bashrc** são:

   (a) `export GOROOT=/root/golang/go`

   (b) `export PATH=$PATH:$GOROOT/bin`

   É possível verificar se tudo foi instalado corretamente verificando se o comando `go version` mostra a versão que foi instalada.

6. Na próxima etapa, é necessário instalar compiladores cruzados pois era necessário que pelo menos uma máquina compilasse o código dos bots para todas as arquiteturas propostas pelo Mirai. Para a recuperar o código dos binários, foi necessário executar os seguintes comandos:

   ```
   # wget -recursive -no-parent https://www.uclibc.org/downloads/binaries/0.9.30.1/
   ```

   Com esse comando foi possível baixar todos os *cross-compilers* do Mirai. Dentro dessa URL tem arquivos que não são usados, estes são descartados, apenas os arquivos com o prefixo *cross-** serão utilizados. Para armazenar o código executável de cada compilador, foi necessário criar uma pasta em /etc com um nome genérico, no caso deste trabalho o diretório novo criado foi o `/etc/xcompile`. Cada arquivo relacionado a um compilador estava compactado, então todos foram descompactados seguindo a sua respectiva nomenclatura.

   Após a descompactação dos compiladores dentro de `/etc/xcompile/`, foi necessário editar o arquivo `.bashrc` novamente, pois era necessário que o sistema conhecesse o caminho dos respectivos binários quando estes fossem chamados dentro do Mirai. Foram definidos as seguintes variáveis de ambiente dentro do `.bashrc`

   (a) `export PATH=$PATH:/etc/xcompile/cross-compiler-armv41/bin`

(b) `export PATH=$PATH:/etc/xcompile/cross-compiler-armv51/bin`

(c) `export PATH=$PATH:/etc/xcompile/cross-compiler-armv61/bin`

(d) `export PATH=$PATH:/etc/xcompile/cross-compiler-i586/bin`

(e) `export PATH=$PATH:/etc/xcompile/cross-compiler-i686/bin`

(f) `export PATH=$PATH:/etc/xcompile/cross-compiler-m68k/bin`

(g) `export PATH=$PATH:/etc/xcompile/cross-compiler-mips/bin`

(h) `export PATH=$PATH:/etc/xcompile/cross-compiler-mipsel/bin`

(i) `export PATH=$PATH:/etc/xcompile/cross-compiler-powerpc/bin`

(j) `export PATH=$PATH:/etc/xcompile/cross-compiler-powerpc-440fp/bin`

(k) `export PATH=$PATH:/etc/xcompile/cross-compiler-sh4/bin`

(l) `export PATH=$PATH:/etc/xcompile/cross-compiler-sparc/bin`

(m) `export PATH=$PATH:/etc/xcompile/cross-compiler-x86_64/bin`

7. Próximo passo da configuração é recuperar as bibliotecas [28][29] que a linguagem Go utiliza dentro do código do Mirai que são essenciais para a interação com o banco de dados criado pelo Mirai. Os comandos executados como *root* foram:

   (a) `go get github.com/go-sql-driver/mysql`

   (b) `go get github.com/mattn/go-shellwords`

8. Após a instalação tanto do banco de dados *MySQL* e dos pacotes auxiliares da linguagem Go que o servidor de comando e controle precisa para fazer a interação, foi necessário definir um usuário e uma senha de administrador. É importantíssimo frisar que é estritamente necessário que essa combinação seja lembrada, pois será ela que o servidor de comando e controle utilizará para acessar e gerir a *botnet*. Para acessar o banco de dados, o seguinte comando foi utilizado:

   ```
   # mysql -u root -p
   ```

   A partir desse comando, o *MySQL* irá pedir pela senha que foi definida anteriormente, basta entrar e a partir daí, é necessário criar manualmente o registro no banco de dados que será utilizado pelo servidor de comando e controle para controlar o Mirai [26].

   ```
   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`)
);
```

Sendo criado o banco de dados, é necessário que seja criado um usuário administrador do servidor de comando e controle, ou seja, inserindo um valor dentro da tabela *users*.

(a) `INSERT INTO users VALUES (NULL, 'root', 'root', 0, 0, 0, 0, -1, 1, 30, ");`

Vale lembrar que o par de credencial `root:root` foi usado somente nesse trabalho com o intuito de deixar tudo padronizado e evitar a necessidade de gerar mais de

uma senha diferente pro laboratório. Para consolidar essas ações é necessário reiniciar o serviço do *MySQL* ou reiniciando a máquina.

```
# service mysql restart
```

Feita a instalação completa do serviço de banco de dados, é necessário compilar o código do Mirai que estava dentro do subdiretório `/mirai` utilizando o *script* `build.sh`. Para verificar o acesso ao servidor de comando e controle, é necessário que a porta 23 e a porta 101 estejam abertas. Dado que elas estão corretamente configuradas, o acesso utilizando o cliente *Telnet* foi realizado a partir do arquivo executável `cnc` que foi gerado após a execução do `build.sh`.

Figura 3.5: Comando e controle esperando por conexão

Figura 3.6: Conexão estabelecida no C&C por meio do *Telnet*

Servidor *Loader* e Servidor *ScanListen*

Em primeiro plano, temos o Servidor *Loader* que é responsável por carregar o *bot* nas máquinas automaticamente utilizando uma das interfaces de rede, a outra interface é responsável por realizar a infecção quando o servidor *ScanListen* adicionar outras informações ao arquivo que é utilizado como base de infecção. Em segundo plano tem o servidor *ScanListen* que é responsável por aceitar conexões de *bots* recebidas na porta 48101 e dispobilizar os dados necessários para a invasão ao servidor *Loader*.Seguem as configurações da máquina que hospeda ambos servidores:

- Sistema Operacional *Ubuntu Server 20.04 LTS*
- 4096 Mb de memória RAM
- 20Gb de armazenamento virtual, podendo ser expansível

Ao contrário do Servidor de Comando e Controle, o Servidor *Loader* é implementado na linguagem C. Entretanto o Servidor de *ScanListen* está codificado na linguagem Go. Para não precisar compilar o código do Servidor de Comando e Controle novamente para gerar o código `scanListen`, o código que é gerado dentro do Servidor de Comando e Controle é transferido para o Servidor *ScanListen*. Além dos executáveis compilados, os *cross-compilers* serão transferidos também.

1. Assim como descrito na configuração do Servidor de Comando e Controle, houve a instalação dos pacotes para a recuperação do código fonte do Mirai, sendo eles o git, gcc e o eletric-fence.
2. Para a instalação do pacote da linguagem Go, foram seguidos exatamente os passos (4) e (5) que estão descritos configuração do Servidor de Comando e Controle. Após a instalação da linguagem Go, foi feita a compilação do subdiretório `loader/`. Em seguida, a transferência do arquivo '**scanListen**' que estava na máquina de Comando e Controle foi transferida por meio da utilização do protocolo SSH[30] utilizando a diretiva scp(*secure copy)*. Depois do copiar este arquivo para o servidor correto, ele foi movido para o subdiretório `loader/`.
3. Os *cross-compilers* foram copiados a partir do Servidor de Comando e Controle, juntamente com as diretivas corretas que devem estar no `.basrc` utilizando a diretiva scp. Após a cópia o *script* que se encontra no subdiretório `dlr/build.sh` foi executado. Todos os arquivos com formato **dlr.*** foram movidos para o subdiretório do *Loader* `bins/`.
4. O código do loader que se encontra no subdiretório `loader/` foi compilado.

Servidor de Binários

O Servidor de Binários tem a função de hospedar os arquivos executáveis dos *bots*, podendo ser disponibilizados para download pelos clientes que contratassem o Mirai para prestar um serviço de *DDoS for hire*. Segue a configuração do Servidor de Binários:

- Sistema Operacional *Ubuntu Server 20.04 LTS*
- 2048 Mb de memória RAM
- 20Gb de armazenamento virtual, podendo ser expansível

1. Instalação do servidor *Web Apache* utilizando o seguinte comando
   ```
   # apt-get install apache2
   ```

2. Depois de realizar a instalação do c*apache2*, foi necessário atualizar seu arquivo base `000-default.conf`, se localizando precisamente no caminho `/etc/apache2/sites-available/`. A razão pela qual foi necessária alterar o arquivo foi para definir qual diretório o servidor iria considerar quando fosse verificar qual era o diretório raiz quando fosse acessado [31]. A alteração foi trocar `/var/www/html` para `/var/www`, com o intuito de acessar diretamente o diretório da aplicação que hospedaria os binários do Mirai.

3. Com o servidor *Web* configurado, o próximo passo foi passar os *bots* que foram previamente compilados dentro do Servidor de Comando e Controle para o subdiretório `/bins` dentro de `/var/www/`. Essa pasta foi criada pois o próprio *bot* quando está procurando algum arquivo para baixar, é feito nessa pasta. A transferência foi realizada pelo scp.

4. Após ter configurado o servidor e copiado os bots para o Servidor de Binários, foi necessário reiniciar o serviço do apache por meio do seguinte comando:
   ```
   # systemctl restart apache2
   ```
   Para verificar que o servidor estava funcionando, é necessário executar o comando:
   ```
   # systemctl status apache2
   ```

Servidor DNS

Todos os dispositivos que são infectados pelo Mirai realizam consultas DNS em tempo de execução. A realização da consulta tem como base requisições que são feitas por parte do *bot* diretamente para o Servidor DNS. Como qualquer servidor DNS tem inúmeros registros de diferentes tipos [32] de um endereço, é mais fácil fazer essa consulta a este

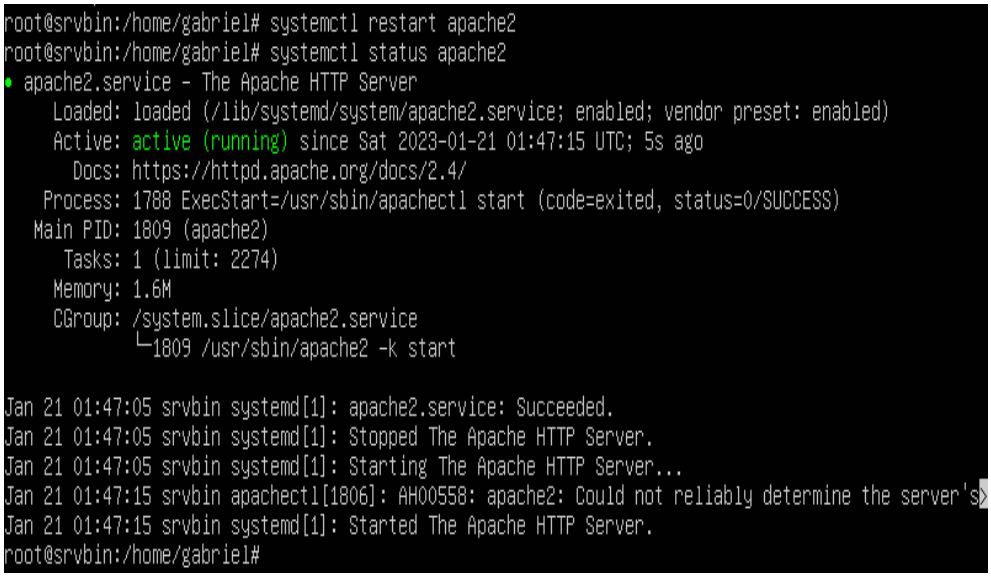

Figura 3.7: Servidor *Web* Apache2

servidor e verificar se o domínio possui registros dentro do servidor. Para configurar o DNS dentro do laboratório, foi necessário editar o arquivo `/etc/hosts` [33] para abranger todos os domínios que serão utilizados e que estão configurados dentro do laboratório. A função do deste arquivo é agir como um servidor local de DNS [34]. Além de editar esse arquivo, em cada máquina foi necessário editar o arquivo `/etc/resolv.conf` para apontar o endereço do DNS como resolvedor principal.

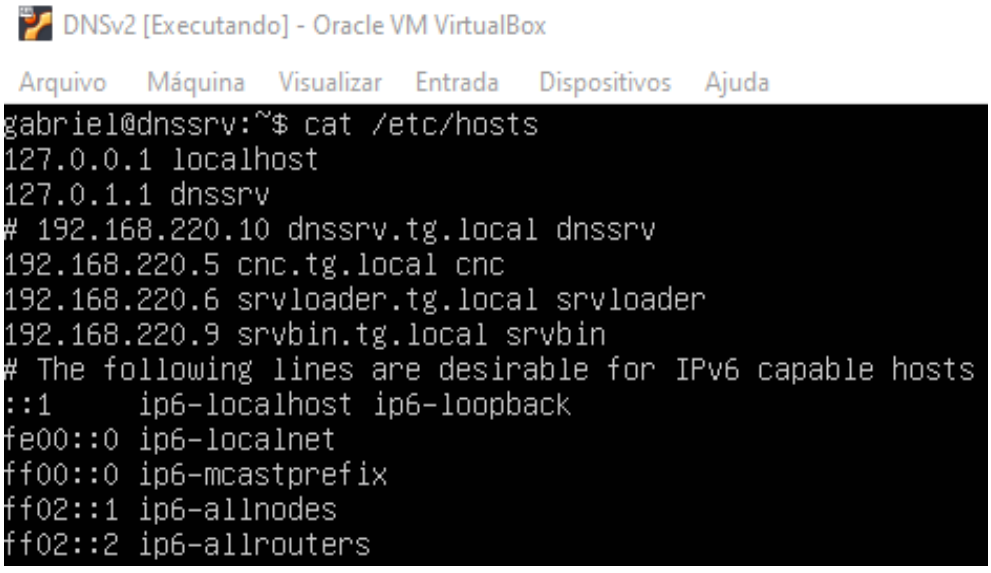

Figura 3.8: Configuração do servidor DNS no arquivo `/etc/hosts`

A configuração do servidor ficou a seguinte:

- Sistema Operacional *Ubuntu Server 20.04 LTS*

Figura 3.9: *Script* para alterar arquivo `/etc/resolv.conf` em todos servidores

- 2048 Mb de memória RAM
- 20Gb de armazenamento virtual, podendo ser expansível

O Servidor DNS ficou responsável por criar um nome de domínio para ser referenciado de acordo com a configuração. Tal domínio ficou definido como `tg.local` assim era possível cada máquina ter um FQDN (*Fully Qualified Domain Name*) próprio e apontar para o servidor DNS. Os domínios criados foram:

- dnssrv.tg.local
- cnc.tg.local
- srvloader.tg.local
- srvbin.tg.local

1. Para a configuração inicial do servidor DNS, foi utilizado o BIND9 para criar uma zona primária de DNS [35]. Envolve a modificação e criação de diversos arquivos novos dentro de `/etc/bind/`.o principal arquivo foi o `/etc/bind/db.tg.local`. Esse arquivo é responsável por definir os registros que cada domínio tem. No caso desse trabalho, só iremos trabalhar com registros do tipo **A**. Neste arquivo temos as definições de diferentes parãmetros que possuem diferentes funções dentro da transferência de zona [36]:

   (a) TTL: *Time to live* diz respeito a quanto tempo o registro DNS vai deixar em *cache* aqueles dados que foram recuperados antes de necessitar de outra consulta.

   (b) @: Denota a origem atual do domínio, que neste caso que é `dnssrv.tg`.

   (c) SOA: Faz parte da definição do FQDN (*Fully Qualified Domain Name*)

   (d) IN: Registro DNS que é a abreviaviação de "Internet"

   (e) Serial: Um número que sempre precisa ser incrementado toda vez que ocorre uma mudança

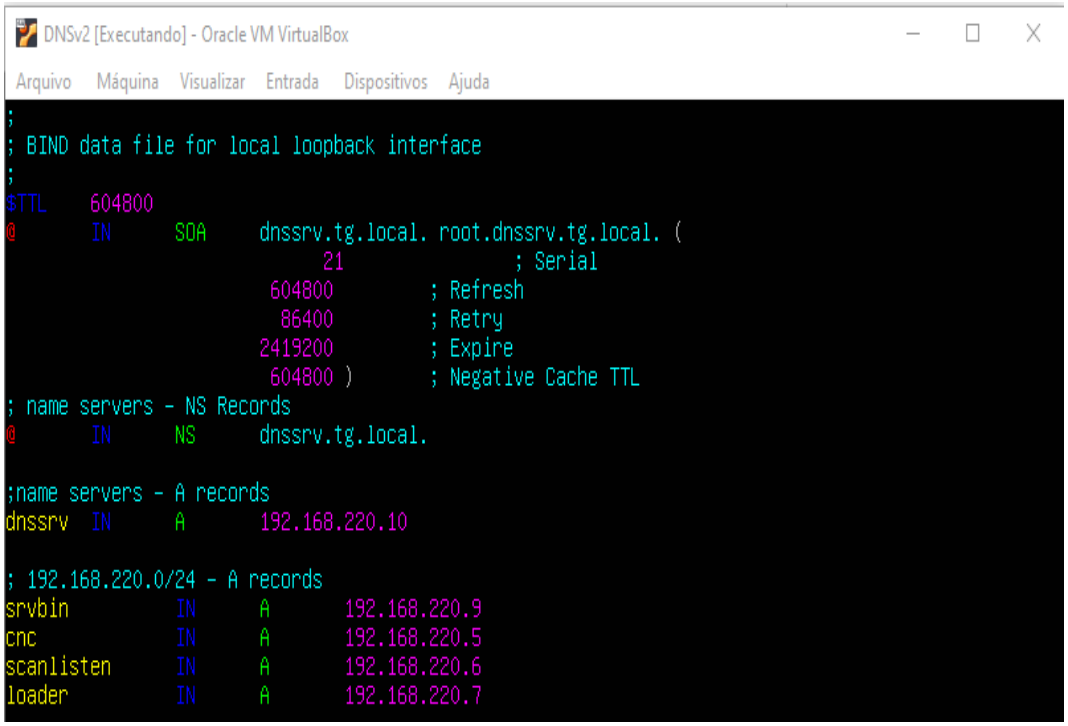

Figura 3.10: Configuração do arquivo principal para transferência de zona.

(f) *refresh*: Se refere a quanto tempo o servidor de DNS primário deve esperar para fazer consultas nos servidores de DNS secundários.

(g) *retry*: Se o servidor de DNS primário demorar esse tempo para responder, é o tempo que o servidor de DNS secundário deve esperar para fazer uma consulta ao DNS primário novamente.

(h) *expire*: Em caso de falha do DNS primário de resposta a um DNS secundário, o DNS secundário deve parar de oferecer consultas para o servidor DNS completamente.

(i) *negative caching TTL*: Define por quanto tempo que uma resposta negativa a uma consulta deve permancer armazenada em *cache*.

2. Depois da configuração inicial 3.8 dos arquivos envolvendo a ferramenta, foi necessário alterar o arquivo `/etc/hosts` dentro do Servidor de DNS com o intuito de enumerar todas as máquinas possíveis que seriam usadas, além dos outros servidores de DNS que seriam usados na amplificação por reflexão.

3. Por fim, após fixar o `/etc/hosts` do Servidor DNS, é possível ligar os servidores e executar o *script* em 3.9. A função desse *script* é basicamente a de referenciar um novo servidor DNS *nameserver* para o domínio em questão. O código apenas

abre uma conexão SSH em cada máquina que está no vetor, e muda o arquivo `/etc/resolv.conf`, apontando o novo nameserver como 192.168.220.10

Configuração dos *bots*

O laboratório é composto por quatro *bots*, sendo dois deles máquinas ubuntu server 20.04 e os outros dois um sistema operacional *Crowz*, para simular melhor dispositivos IoT que naturalmente possuem uma capacidade de processamento menor e um poder de ataque menor do que uma máquina com um sistema operacional mais robusto.

- Sistema Operacional *Ubuntu Server 20.04 LTS*
- 1024 Mb de memória RAM
- 15Gb de armazenamento virtual, podendo ser expansível

- Sistema Operacional *Crows 3.0.0*
- 512 Mb de memória RAM
- 10Gb de armazenamento virtual, podendo ser expansível

1. Alteração da senha do administrador para equivaler a um par que está dentro do `mirai/bot/scanner.c`, utilizando o comando `sudo passwd root`.
2. Instalação do serviço *Telnet*.
   (a) `# apt-get install telnetd xinetd`
   (b) `# ufw allow 23`
   (c) Criação de um arquivo dentro de `/etc/xinetd.d` com as seguintes diretivas para a configuração do servidor *Telnet*

```
# 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
}
```

Feitas essas configurações, os *bots* estavam prontos para serem testados sem maiores problemas.

3.5 Alterações no código original

O código que está disponível no repositório do github [27], não compila diretamente por ter algumas configurações alteradas pelo próprio criador do *malware*, visto que, uma vez que as configurações fossem alteradas, é possível executar o Mirai em outro ambiente. Seguem todos os arquivos que foram modificados para a execução do Mirai.

3.5.1 Alterações do código do Servidor Comando e Controle

1. `cnc/main.go`: As alterações feitas neste arquivo foram pontuais, com o intuito de modificar o endereço onde se encontrava o banco de dados, que era no próprio servidor de Comando e Controle, alterar o par usuário e senha que do *MySQL* que foram definidos na instalação e o nome da própria tabela no banco de dados com o usuário.

3.5.2 Alterações no código do Servidor *Loader* & *ScanListen*

Durante o estudo da arquitetura do Mirai, após a configuração da máquina em que o *loader*, foi verificado que a função principal do *loader* dentro do ecossistema era de realizar a detecção e infecção automática de outros dispositivos vulneráveis que estão dentro do intervalo de rede onde o *bot* se encontra. Pelo fato de ter sido verificado que a execução do Mirai não dependia do Loader dentro do ambiente controlado, foi descartado o seu uso dentro desse estudo, pois o mesmo não iria intervir diretamente nos resultados do experimento, visto que ele não tem função na hora do ataque, apenas na fase de infecção. A função do *scanlisten* também não afetava diretamente, visto que ela escaneava intervalos de redes por máquinas que possivelmente estariam vulneráveis, passando-as para o *loader* tentar carregar o *bot* nas máquinas.

3.5.3 Alterações no código executado pelos *bots*

Código para a execução do ataque de negação de serviço por reflexão amplificada adaptado ao Mirai

Para a execução do ataque de negação de serviço por reflexão amplificada, foi utilizado um módulo já existente do Mirai, o `attack_udp_dns` foi alterado com o intuito de trocar o ataque que já existia *DNS Water Torture* pelo ataque de reflexão amplificada.

1. attack_udp_dns.c

   (a) Linha 298: Endereço de origem foi trocado pelo endereço da vítima que sofreria com a reflexão do DNS. No caso deste trabalho, foi o servidor de binários do próprio Mirai "192.168.220.9".

   (b) Linha 332: O tipo de consulta DNS no código original é uma consulta ao registro do tipo 'A', que buscava o endereço IPv4 de um certo domínio. Esse valor foi mudado de 1 para 255, para que a consulta utilizada fosse a 'ANY'.

O fluxo principal da função é encapsular um pacote com os cabeçalhos IP, UDP e DNS. Quando são acoplados, o cabeçalho de cada pacote é definido para que este seja tratado corretamente pelo servidor DNS e redirecionado corretamente para o endereço IP da vítima. Depois da definição dos cabeçalhos, o *bot* procura o servidor DNS da rede possuindo o domínio como parâmetro. Para achar o servidor de DNS, o código do *bot* abre os arquivos das máquinas infectadas e procura pelo primeiro *nameserver* que se encontram em `/etc/resolv.conf`. Ajustado o cabeçalho e o domínio, o bot entra em um loop infinito, que é quebrado pela quantidade de segundos que o cliente executa o *bot*, para criar uma estrutura de 'n' pacotes que serão enviados a diferentes *hosts* caso sejam específicados na hora do ataque, como só é especificado um servidor DNS, o *bot* faz somente um pacote e coloca o mesmo na fila. Depois de criar essa estrutura para os pacotes, o *bot* mascara o domínio de origem do pacote com uma string alfanumérica utilizando a função `rand_alphastr`. Feito isso, os *checksums* dos pacotes são definidos, apesar de não serem obrigatórios nesse tipo de ataque por se tratar de um ataque UDP. Feito esses passos, o *bot* envia o pacote com o endereço de origem adulterado e volta pro início do loop onde é definida nova estrutura para o mesmo tipo de pacote que será enviado novamente.

Além do ataque de negação de serviço por reflexão amplificada, que foi adaptado a arquitetura do Mirai, temos também ajustes para que os *bots* funcionem corretamente.

Còdigo para os *bots* funcionarem corretamente

1. `bot/util.c`: Troca pelo endereço de DNS público do Google para o servidor DNS que era utilizado dentro do laboratório

   (a) `addr.sin_addr.s_addr = INET_ADDR(192,168,220,9);`

2. `bot/table.c`: Preenchimento de constantes dentro de uma tabela que é consultada pelo *bot*. Os valores são ofuscados por meio de um algoritmo que se encontra em `mirai/tools/enc.c`. Os valores que mudam é o domínio do servidor de Comando e Controle e o domínio do servidor de *ScanListen*.

   (a) Caracteres cifrados para cnc.tg.local: ("\x41\x4C\x41\x0C\x56\x45\x0C\x4E \x4D\x41\x43\x4E\x22", 13);

   (b) Caracteres cifrados para scanlisten.tg.local: ("\x51\x41\x43\x4C\x4E\x4B\x51\x56 \x47\x4C\x0C\x56\x45\x0C\x4E\x4D\x41\x43\x4E\x22", 20);

3. `bot/resolv.c`: Linha 84 o DNS da Google (8,8,8,8) é substituído pelo DNS que será utilizado por todas as máquians do laboratório (192,168,220,10).

4. `bot/scanner.c`: Neste arquivo, entre as linhas 126 e 185 foram comentadas todas as senhas que estavam sendo utilizdas na hora do *bruteforce*, apenas três pares de usuário:senha foram utilizados. Entre eles, o par `root:1234`, o qual era a senha que dava acesso privilegiado a todas as máquinas do laboratório. Além disso, a função `get_random_ip` foi modificada com o intuito de deixar IPs gerados pseudoaleatoriamente dentro do escopo da rede.

   (a) o1 = 192. o2 = 168; o3 = 220; o4 = (tmp » 24) & 0xff;

   (b) while (o3 = 255 || o4 < 8 || o4 == 255);

   (c) return INET_ADDR(o1,o2,o3,o4);

5. `bot/rand.c`: No bloco else, na linha 78 do código dentro da função `rand_alphastr`, a randomização do domínio não estava sendo corretamente feita, pois caso o *byte* do domínio não fosse divisível por 4 dentro de 0 a 32, não eram corretamente exibidas pois não são caracteres alfanuméricos.

   (a) uint8_t aux = rand_next() & 0xff;

   (b) aux = aux » 3;

   (c) *str++ = alphaset[aux];

   (d) len−−;

3.6 Síntese

Neste capítulo foi possível entender como a reflexão amplificada será implementada dentro do Mirai, visto que a geração do pacote DNS já está parcialmente implementada no

próprio Mirai, sendo necessário ter alterações pontuais no *bot* para a execução da reflexão amplificada. Depois da explicação da implementação e de como funciona a reflexão amplificada dentro do Mirai, são evidenciados todas as configurações necessárias para a execução do Mirai, especificações de máquinas, configuração de máquina, instalação de pacotes, possíveis alterações no código fonte do Mirai, configuração da rede para as máquinas se comunicarem entre si e não possuírem ligação com o meio externo para evitar possíveis problemas. No próximo capítulo serão evidenciados os testes feitos que foram a execução do ataque de negação de serviço por reflexão amplificada utilizando diferentes cenários e os resultados obtidos, se foi possível esgotar os recursos do servidor alvo.

Capítulo 4

Testes e Resultados

Nesta seção são apresentados os testes que foram realizados dentro do contexto da amplificação por reflexão e a análise da quantidade de pacotes que foram utilizados. O principal objetivo do teste nessa seção, é de verificar a funcionalidade do ataque de amplificação por reflexão e o seu respectivo desempenho. Essas duas verificações são feitas a partir da análise do tempo de resposta do servidor que está sendo atacado, no caso, o servidor de binários, cujo domínio é: *srvbin.tg.local*.

4.1 Execução do ataque

Aqui estão os testes de demonstração da implementação de funcionalidade que foi proposta, a geração de probes DNS.

Com a análise estática do código fonte do Mirai, foi possível avaliar como a implementação do ataque de reflexão amplificada seria feita. A princípio, o Mirai iria receber um módulo novo que forjaria os pacotes DNS e a partir dessa nova funcionalidade, o ataque seria lançado. Entretanto, durante a análise do Mirai, foi possível verificar que um dos arquivos do Mirai já implementava a lógica do gerador de pacotes dentro de um de seus ataques, portanto, o gerador de DNS que havia sido implementado anteriormente foi descartado. A seguir, o passo a passo de execução do vírus será explicitado:

1. Inicialização das máquinas virtuais: servidor de comando e controle (cnc), servidor de binários (srvbin), servidor DNS (dnssrv) e dos quatro *bots* que serão utilizados neste experimento.

2. Antes de realizar qualquer ação, é necessário que todas as máquinas que estão ligadas, estejam com o acesso privilegiado.

3. Feita o acesso privilegiado em todas as máquinas, é estritamente necessário que todas as máquinas, com a exceção da máquina DNS, tenham o arquivo `/etc/resolv.conf`

corretamente configurado com o nameserver apontando para o IP do servidor DNS, como mostra a figura a seguir:

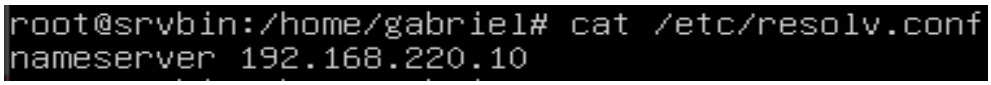

Figura 4.1: Arquivo `/etc/resolv.conf` configurado

4. Assim que o servidor de comando e controle é inicializado, é necessário executar o binário que se encontra em `/mirai/release/cnc` com o seguinte comando:
   `# ./cnc`
   Esse comando é responsável por abrir a porta 23 do servidor de comando e controle e deixá-lo apto a receber conexões de possíveis clientes nesta porta. Após o comando de abertura do servidor de comando e controle, é necessário conectar no servidor utilizando o protocolo *Telnet*, com o seguinte comando:
   `# telnet 192.168.220.5`
   Neste caso, é possível utilizar o próprio servidor de comando e controle para conectar a si mesmo, como mostram as figuras 3.5 e 3.6.

5. Após colocar a senha correta, que é o par usuário:senha definidos na criação da tabela no banco de dados *MySQL*, para a conexão do servidor de comando e controle, é possível verificar quais tipos de ataques são possíveis dentro do contexto do Mirai, como mostra a figura a seguir:

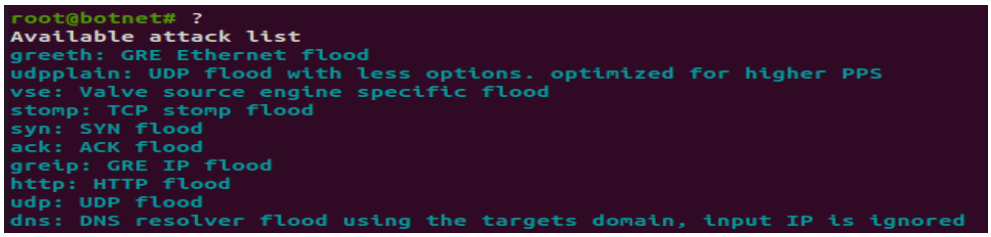

Figura 4.2: Lista de ataques disponíveis no Mirai

6. Após a conexão com o servidor de comando e controle, é necessário que todos os *bots* estejam com o executável correto de acordo com a sua arquitetura. Como todas as máquinas virtuais foram utilizados sistemas operacionais x86, o executável em todas as máquinas é o `mirai.x86`. Feito isso, se não houver ajustes no código que verifica o nome do arquivo que será aberto, é necessário mudar o nome do executável em cada bot para `dvrHelper`. Não há motivo aparente para este nome, mas é o nome de arquivo que é verificado para se abrir em tempo de execução.

7. Após executar o `dvrHelper` nos *bots* em cada máquina virtual que está simulando uma vítima, é possível verificar se o servidor de comando e controle recebeu as conexões, como mostra na imagem a seguir:

Figura 4.3: Quantidade de *bots* conectados no servidor de comando e controle

8. É possível escolher qual ataque os *bots* irão executar a partir do console virtual que está no servidor de comando e controle, neste caso utilizaremos a diretiva DNS. O formato do ataque segue o seguinte padrão para ataques DNS:

   ```
   # dns dns_ip tempo_ataque sport dport domain
   ```

   A diretiva 'dns' é o próprio tipo do ataque, 'dns_ip' é o IP do servidor DNS, que no caso deste ataque é ignorado, pois o próprio ataque tem um módulo que o servidor é atribuído a uma variável a partir do que está configurado no arquivo `/etc/resolv.conf`. O 'tempo_ataque' é o tempo de ataque em segundos. As diretivas 'sport' e 'dport' são respectivamente, a porta do serviço da vítima que será atacada, e a porta de destino, que a vítima supostamente está enviando a requisição, neste caso as portas são 80, pois o servidor *web* apache está executando nesta porta, e a porta destino é a porta 53, pois é a porta padrão que o serviço do DNS é executado. Por fim, a diretiva 'domain' é o domínio que será tomado como base no ataque, também é o domínio que será encontrado o servidor DNS primário. É importante lembrar que essas diretivas após o tempo de ataque são opcionais, mas no código a ser executado pelo *bot*, caso não sejam definidas, é atribuido um valor aleatório a todos esses valores. As outras diretivas que são possíveis definir como: ToS (*type of service*), ID do pacote, ttl (*time-to-live*), e ID do domínio são completamente opcionais, pois na ausência da sua definição, valores aleatórios são atribuídos, que não afetam o ataque como um todo, desde que seja atribuído algum tipo de valor nessas variáveis. A chamada do ataque segue na figura a seguir:

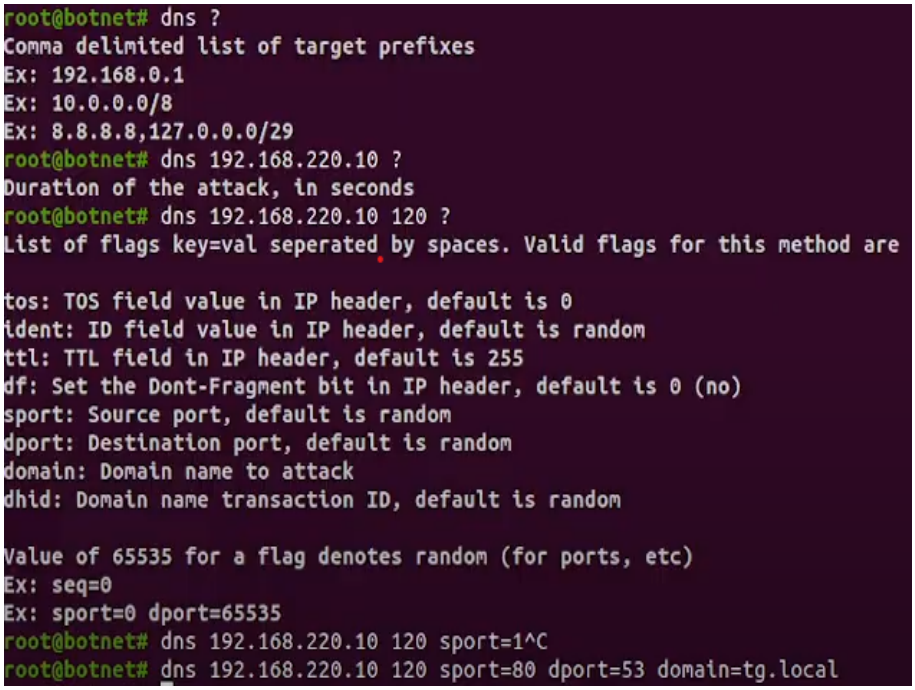

Figura 4.4: Definição do ataque de reflexão amplificada

9. Após o início do ataque, é possível notar o sucesso do ataque, utilizando a ferramenta de análise de pacotes *tcpdump*. Para a análise do ataque dentro do servidor de Binários e do servidor DNS, foram utilizados os seguintes comandos:

   Servidor de binários:

   ```
   # tcpdump not arp and host 192.168.220.9 and port 80 -vvv
   ```

   Servidor DNS:

   ```
   # tcpdump not arp and host 192.168.220.10 and port 80 -vvv
   ```

   Com esses dois comandos foi possível notar que os pacotes estavam se comportando da maneira correta, visto que a consulta no servidor DNS realmente era a consulta ANY, o único servidor que estava recebendo os pacotes de resposta do DNS era o servidor de binários.

10. Durante o ataque, foi verificado também que houve negação de serviço no servidor de binários, visto que o mesmo demorava por volta de 25 segundos para responder a uma requisição GET utilizando o curl, como mostra a imagem a seguir:

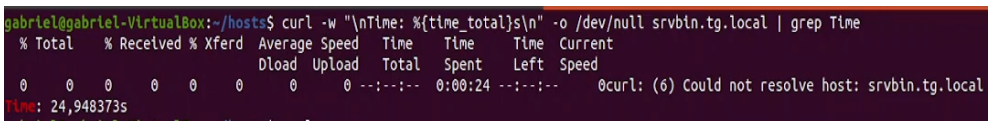

Figura 4.5: Negação de serviço no servidor de binários

4.2 Teste de efetividade de DDoS: misto de funcionalidade/desempenho

Tendo em vista que o protocolo base utilizado na comunicação com um servidor DNS é o protocolo UDP, foram feitas análises comparando o ataque de negação de serviço por reflexão amplificada com outros ataques que são implementados pelo próprio Mirai utilizando a mesma suíte base de protocolo. Os ataques que serão comparados a seguir, são os ataques *UDP generic*, e *UDP plain* e o próprio ataque de reflexão amplificada. O ataque *UDP generic* consiste em enviar pacotes UDP forjados com os cabeçalhos padrões que um pacote UDP possui. Os cabeçalhos que podem ser alterados são o endereço de destino, endereço de origem, porta de destino, porta de origem, tipo de serviço, identificador, tempo de vida, tamanho, randomização de conteúdo dentro do pacote, fragmentação, e origem. O ataque *UDP Plain* consiste em enviar pacotes UDP forjados sem a presença das outras diretivas opcionais dentro do pacote que são enviadas utilizando o outro ataque. As únicas diretivas presentes dentro desse ataque são tamanho, aleatoriedade do conteúdo do pacote, porta de destino. Isso é feito com o intuito de otimizar o ataque e consequentemente dar mais vazão ao número de pacotes por segundo, visto que quanto menor o pacote, mais rápido ele é montado em tempo de execução.

A seguir serão descritos os parâmetros utilizados nos ataques:

1. UDP *Generic*:

   ```
   # udp 192.168.220.9 60 sport=12345 dport=80
   ```

   Dentro deste ataque, há a possibilidade de se trabalhar com mais diretivas, entretanto, nesse caso não há essa necessidade de trabalhar com diretivas opcionais, visto que elas são definidas aleatoriamente pelo próprio *bot*, a não ser a da porta de origem, porém essa diretiva é utilizada apenas com o intuito de verificar se o ataque está saindo do lugar certo de fato. O que está sendo testado é a potência dos ataques. As diretivas utilizadas, com exceção da porta de origem, possuem impacto direto no ataque, pois é necessário estabelecer o lugar certo para que o ataque ocorra, neste caso na porta 80.

   (a) É estritamente importante salientar que a utilização da porta 80(TCP) para um ataque UDP não funciona, pois este não é o tipo de serviço correto na porta. Neste caso o ataque acabou dando certo pela sua agressividade e pelo esgotamento de recurso. O ataque em ambiente de teste só funcionou porque não houve um firewall impedindo as requisições fossem barradas. Para mitigar esse ataque em questão no ambiente de teste, apenas uma filtragem por IP

ou por porta seria o suficiente para conter o ataque de negação de serviço por reflexão amplificada.

2. UDP *Plain*:

   ```
   # udp 192.168.220.9 60 dport=80
   ```

   Este ataque por ser mais simples e menos robusto de se fazer, naturalmente se torna mais leve, o que por consequência aumenta a vazão de pacotes, deixando-o mais potente.

3. Ataque de amplificação por reflexão:

   ```
   # dns 192.168.220.10 60 sport=80 dport=53 domain=tg.local
   ```

   A implementação deste ataque foi o objetivo principal do trabalho, a título de comparação e eficácia, será comparado com outros ataques que utilizam a mesma base de sua suíte de protocolos.

A seguir, temos as comparações em tabelas a respeito da ocorrência da negação de serviço distribuída por degradação do tempo de resposta a requisição GET. Tais requisições foram feitas utilizando a ferramenta *cURL (Client URL)*, previamente instalada dentro das máquinas. O comando executado em todos os casos foi utilizado para medir quantos segundos uma requisição demorava para chegar até o servidor em diferentes cenários. Os cenários que foram comparados foram: Sem ataque acontecendo (N/A), ataque *UDP Generic*, ataque *UDP Plain* e o ataque de negação de serviço por reflexão amplificada.

O comando utilizado para mensurar o tempo da requisição foi o seguinte:

```
# curl -w "\nTime:  %{time_total}s \n-o /dev/null srvbin.tg.local
```

Requisição	N/A	UDP Generic	UDP Plain	Reflexão Amplificada
1ª	0.033s	0.061s	0.082s	0.972s
2ª	0.016s	0.084s	0.090s	0.771s
3ª	0.023s	0.062s	0.071s	1.066s
4ª	0.026s	0.088s	0.052s	1.272s
5ª	0.045s	0.087s	0.050s	1.043s

Tabela 4.1: Tabela de execução dos ataques com um *bot*

Requisição	N/A	UDP Generic	UDP Plain	Reflexão Amplificada
1ª	0.033s	0.068s	0.106s	1.193s
2ª	0.016s	0.069s	0.108s	6.338s
3ª	0.023s	0.072s	0.998s	1.320s
4ª	0.026s	0.063s	0.111s	1.327s
5ª	0.045s	0.112s	0.094s	1.742s

Tabela 4.2: Tabela de execução dos ataques com dois *bot*

Requisição	N/A	UDP Generic	UDP Plain	Reflexão Amplificada
1ª	0.033s	0.125s	0.100s	´9.959s
2ª	0.016s	0.138s	0.177s	9.988s
3ª	0.023s	0.066s	0.070s	5.875s
4ª	0.026s	0.129s	0.096s	9.869s
5ª	0.045s	0.091s	0.105s	1.124s

Tabela 4.3: Tabela de execução dos ataques com três *bot*

Requisição	N/A	UDP Generic	UDP Plain	Reflexão Amplificada
1ª	0.033s	0.198s	0.124s	5.955s
2ª	0.016s	0.228s	0.133s	9.909s
3ª	0.023s	0.157s	0.115s	30.00s
4ª	0.026s	0.101s	0.196s	-
5ª	0.045s	0.225s	0.159s	-

Tabela 4.4: Tabela de execução dos ataques com quatro *bots*

O motivo principal de apenas utilizar o tempo de resposta do servidor como parâmetro principal deste trabalho no que tange a efetividade do ataque de negação de serviço por reflexão amplificada foi com o intuito de mostrar a real efetividade dentro do laboratório do ataque de negação de serviço por reflexão amplificada em relação aos outros ataque disponíveis no *bot*. Com o tempo de resposta dos servidores em relação a cada um dos ataques, é possível notar a grande diferença que o ataque de reflexão amplificada tem em relação aos outros ataques que são feitos pelo Mirai seguindo a mesma proporção de máquinas conectadas a um servidor de comando e controle. Outra observação interessante, é importante mencionar que o ataque é direcionado no servidor que está hospedando um serviço web, não necessariamente no servidor web, visto que os parâmetros utilizados não condizem com a um servidor web real, no caso do trabalho um ataque na porta 80 utilizando protocolo UDP.

Foram feitas simulações utilizando desde um bot, até o número máximo de *bots* possíveis que são quatro. Em todas as simulações ficou nítido que o ataque de negação de serviço por reflexão amplificada é o mais efetivo quando se trata de negação de serviço. Entretanto, quando o número de *bots* é pequeno, não há negação de serviço, apenas um atraso na requisição. Observando a tabela 4.3 na coluna referente a reflexão amplificada, é possível notar variações nas requisições, sendo que em uma parte do tempo o servidor *web* não estava respondendo mais, enquanto na outra parte ele ainda respondia com um intervalo menor, porém ainda alto se for considerar o tempo de resposta. Na última tabela ficou evidente que a partir de quatro *bots* o servidor *web* parou de funcionar completamente, pois na terceira requisição ele demorou trinta segundos para responder, dando *timeout*. Em nenhuma ocasião foi possível obter uma negação de serviço com os outros ataques mencionados, sendo eles o *UDP generic* ou o *UDP Plain*.

Outro parâmetro que pode ser analisado a partir das requisições que foram feitas foi a quantidade de pacotes por segundo que estava sendo enviada por cada *bot*, visto que conforme a complexidade do ataque aumenta, aumentando o número de *bots*, a quantidade de pacotes enviada é maior.

Seguem as tabelas com a quantidade de pacotes enviados e recebidos pelos servidores, vazão das requisições feitas pelos *bots* durante o ataque, considerando o tempo que o ataque ficou sendo executado por 60 segundos, é possível inferir a vazão dividindo o tempo pela quantidade de pacotes enviados pelos bots. É importante salientar que quando os ataques da tabela se tratam do UDP Plain ou do UDP Generic, o que será utilizado como base, serão os pacotes enviados ao servidor de binários, pacotes enviados pelos *bots* e os pacotes descartados pelo kernel do servidor de binários. Quando o ataque analisado da tabela se referir ao ataque de reflexão amplificada, os números em questão se referem ao número de pacotes recebidos pelo servidor de binários, pacotes recebidos pelo servidor DNS e os pacotes descartados pelo kernel do servidor DNS.

Nº de *bots*	srvbin_pkts	*bot*_pkts/dns_pkts	drop_pkts	Vazão
UDP Generic	5529	3161	9961	92.15
UDP Plain	5789	3123	12239	96,48
R. Amplificada	3095	1854	80344	51,58

Tabela 4.5: Vazão dos pacotes com 1 *bot*

Nº de *bots*	srvbin_pkts	*bot*_pkts/dns_pkts	drop_pkts	Vazão
UDP Generic	3641	22062	49829	60,68
UDP Plain	3496	24280	50751	58,26
R. Amplificada	1783	1603	120437	29,71

Tabela 4.6: Vazão dos pacotes com 2 *bots*

Nº de *bots*	srvbin_pkts	*bot*_pkts/dns_pkts	drop_pkts	Vazão
UDP Generic	2438	26768	54801	40.63
UDP Plain	2561	24280	20736	42.68
R. Amplificada	1449	1241	162231	24,15

Tabela 4.7: Vazão dos pacotes com 3 *bots*

Nº de *bots*	srvbin_pkts	*bot*_pkts/dns_pkts	drop_pkts	Vazão
UDP Generic	2284	26838	59515	38.06
UDP Plain	2578	22201	66122	42.96
R. Amplificada	1562	1236	275438	26.03

Tabela 4.8: Vazão dos pacotes com 4 *bots*

As tabelas acima mostram a quantidade de pacotes em dois instantes distintos, descritos a seguir

1. UDP Generic ou UDP Plain
   (a) **srvbin_pkts**: Se refere ao número de pacotes recebidos pelo servidor de binários.
   (b) **bot_pkts**: Se refere ao número de pacotes enviados pelos *bots* (bot_pkts) podendo ser com **um** até **quatro** *bots* a depender da tabela.
   (c) **drop_pkts**: Se refere a quantidade de pacotes que foi refejeitada pelo kernel do servidor de binários.
   (d) **Vazão**: Se refera a quantidade de pacotes por segundo que estava sendo recebida pelo servidor de binários.
2. Reflexão amplificada
   (a) **srvbin_pkts**: Se refere ao número de pacotes recebidos pelo servidor de binários.
   (b) **dns_pkts**: Se refere a quantidade de pacotes enviados pelo servidor de DNS.

(c) **drop_pkts**: Se refere a quantidade de pacotes que foi refejeitada pelo kernel do servidor de DNS.

(d) **Vazão**: Se refera a quantidade de pacotes por segundo que estava sendo recebida pelo servidor de DNS.

Com estes parâmetros, é possível notar que, apesar da quantidade de pacotes serem baixas, o que é de se esperar em um ambiente controlado com poucos recursos, a quantidade de pacotes que foi descartado pelo kernel dos servidores em questão foi aumentando a medida que a quantidade de *bots* foi subindo, o que comprova o aumento do poder de ataque a medida que o número de *bots* aumenta e a incapacidade do servidor DNS que está sendo utilizado nessa simulação de avaliar todos os pacotes que foram mandados.

4.3 Síntese

No capítulo quatro foram analisados diferentes cenários onde o servidor de binários era constantemente atacado utilizando protocolos da suíte *UDP*. Foram feitas comparações para entender a efetividade de cada ataque contra o servidor de binários. Há uma comparação do tempo de resposta do servidor de binários em cada cenário de ataque, e uma comparação da quantidade de pacotes que foi descartada pelo kernel dos servidores, que aumentou conforme o nível de complexidade do ataque aumentou. Foi obtida negação de serviço quando foram utilizados de três a quatro bots no ataque de reflexão amplificada, nos outros ataques não foi possível esgotar os recursos do servidor.

Capítulo 5

Conclusão

Esse trabalho de graduação teve por objetivo demonstrar como um ataque que possui um potencial de negação de serviço alto pode ser escalado utilizando a manipulação dos protocolos mais comuns da Internet, visto que nesses protocolos não existem muitas checagens de integridade quando se trata de protocolos a nível de rede. Em especial o UDP, que por não exigir checagem de integridade da origem, é passível de ser adulterado.

No início do trabalho foi discutido as principais motivações que levam a um ataque de negação de serviço, sendo citadas algumas delas. Foram explicitados os motivos da realização deste trabalho.

Em seguida foram apresentados diversos conceitos no que tange a negação de serviço, negação de serviço distribuída, os principais protocolos que são utilizados como a base do ataque, como o *Internet Protocol*, *User Datagram Protocol* e o protocolo DNS. Além disso foram discutidas brevemente possíveis mitigações a ataques de negação de serviço e como são feitas. Após esse arcabouço teórico, foi levantado definições de *malware* e onde o Mirai possivelmente se encaixaria dentro de uma taxonomia de acordo com o seu comportamento e rotina de infecção. Foi feita uma breve apresentação da arquitetura do Mirai e como são divididos os subdiretórios do *bot* com as suas respectivas funções.

Após o referencial teórico do Mirai, é mostrado como o ambiente do Mirai é configurado para a infecção ocorrer normalmente, além dos ajustes dentro do código fonte das diferentes estruturas do *bot* e nos arquivos com configurações gerais, que são necessárias para a execução dos ataques vindos das máquinas infectadas.

Dado o referencial teórico, é feita uma análise dos resultados obtidos com o ataque de negação de serviço por reflexão amplificada. Os resultados foram satisfatórios, dentro do que foi seguido, a proposta do trabalho foi concluída com sucesso.

Um dos principais objetivos da escolha deste tema foi para a familiarização de como um *malware* é feito desde a sua motivação, sua codificação, os artifícios que são usados com o intuito de dificultar ao máximo a sua engenharia reversa, artifícios que são utilizados

para ofuscação do *malware* dentro do sistema operacional. Todos esses conhecimentos podem ser aplicados dentro da área de segurança para entender como outras variações de *malware* funcionam.

Referências

[1] Force, Internet Engineering Task: *Internet protocol*. `https://www.ietf.org/rfc/rfc791.txt`, acesso em 2023-10-02. ix, 13, 17

[2] Protocol, User Datagram: *Domain names - implementation and specification*. `https://www.ietf.org/rfc/rfc768.txt`, acesso em 2023-10-02. ix, 13, 17

[3] Force, Internet Engineering Task: *Domain names - implementation and specification*. `https://www.ietf.org/rfc/rfc1035.txt`, acesso em 2023-10-02. ix, 13, 16, 17

[4] Camargo, Camila Imbuzeiro: *Mirai: um estudo sobre botnets de dispositivos iot*. 2018. 1, 10, 16, 22

[5] Douglas, David, José Jair Santanna, Ricardo de Oliveira Schmidt, Lisandro Zambenedetti Granville e Aiko Pras: *Booters: can anything justify distributed denial-of-service (ddos) attacks for hire?* Journal of information, communication and ethics in society, 2017. 3, 9

[6] Scott Sr, James e Winter Summit: *Rise of the machines: The dyn attack was just a practice run december 2016*. Institute for Critical Infrastructure Technology, Washington, DC, USA, 2016. 3, 9

[7] Trendmicro: *Drive-by download*. `https://www.trendmicro.com/vinfo/us/security/definition/drive-by-download`, acesso em 2022-10-01. 3

[8] Kevin D. Mitnick, William L. Simon, Steve Wozniak: *The art of deception*. John Wiley Sons, 2002. 3

[9] Erickson, Jon: *Hacking: The Art of Exploitation*. No Starch Press, 2008. 3

[10] Antonakakis, Manos, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J Alex Halderman, Luca Invernizzi, Michalis Kallitsis *et al.*: *Understanding the mirai botnet*. Em *26th USENIX security symposium (USENIX Security 17)*, páginas 1093–1110, 2017. 4, 10

[11] Yusof, Ahmad Riza'ain, Nur Izura Udzir e Ali Selamat: *Systematic literature review and taxonomy for ddos attack detection and prediction*. International Journal of Digital Enterprise Technology, 1(3):292–315, 2019. 4

[12] Peng, Tao, Christopher Leckie e Kotagiri Ramamohanarao: *Survey of network-based defense mechanisms countering the dos and ddos problems*. ACM Computing Surveys (CSUR), 39(1):3–es, 2007. 4

[13] Nagpal, Bharti, Pratima Sharma, Naresh Chauhan e Angel Panesar: *Ddos tools: Classification, analysis and comparison.* Em *2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom)*, páginas 342–346. IEEE, 2015. 5

[14] Cloudflare: *Botnetdefinition.* `https://www.cloudflare.com/pt-br/learning/ddos/what-is-a-ddos-botnet/`, acesso em 2022-10-01. 5

[15] Aditya Sathish, Jefferson Viana Fonseca Abreu: *Amplified reflection ddos over dns.* Em *In Proceedings of ECE 5584: Network Security. Virginia Tech, Arlington, VA, USA*, páginas 276–280. IEEE, 2022. 5, 7

[16] Karakostas, Bill: *A dns architecture for the internet of things: A case study in transport logistics.* Procedia Computer Science, 19:594–601, 2013. 5

[17] Cloudflare: *Dnsamplification.* `https://www.cloudflare.com/pt-br/learning/ddos/dns-amplification-ddos-attack/`, acesso em 2022-10-01. 7

[18] Zargar, Saman Taghavi, James Joshi e David Tipper: *A survey of defense mechanisms against distributed denial of service (ddos) flooding attacks.* IEEE communications surveys & tutorials, 15(4):2046–2069, 2013. 8

[19] Yaar, Abraham, Adrian Perrig e Dawn Song: *Pi: A path identification mechanism to defend against ddos attacks.* Em *2003 Symposium on Security and Privacy, 2003.*, páginas 93–107. IEEE, 2003. 8

[20] Mirkovic, Jelena e Peter Reiher: *D-ward: a source-end defense against flooding denial-of-service attacks.* IEEE transactions on Dependable and Secure Computing, 2(3):216–232, 2005. 8

[21] Mahajan, Deepika e Monika Sachdeva: *Ddos attack prevention and mitigation techniques-a review.* International Journal of Computer Applications, 67(19), 2013. 8

[22] Saeed, Imtithal A, Ali Selamat e Ali MA Abuagoub: *A survey on malware and malware detection systems.* International Journal of Computer Applications, 67(16), 2013. 9

[23] Crowdstrike: *Types of malware.* `https://www.crowdstrike.com/cybersecurity-101/malware/types-of-malware/`, acesso em 2022-15-01. 9

[24] Pinteric, Marko: *LaTeX on Windows.* `http://www.pinteric.com/miktex.html`, acesso em 2015-06-11. 17

[25] VirtualBox: *Virtualbox downloads.* `https://www.virtualbox.org/wiki/Downloads`, acesso em 2022-20-01. 22

[26] Jihadj4Potus: *Mirai botnet tutorial.* `https://github.com/Screamfox/-Mirai-Iot-BotNet/blob/master/TUTORIAL.txt`, acesso em 2022-20-01. 24, 26

[27] Anna-senpai: *Mirai source code.* https://github.com/jgamblin/Mirai-Source-Code, acesso em 2022-20-01. 24, 35

[28] Golang: *go-driver-sql.* https://github.com/go-sql-driver/mysql, acesso em 2022-20-01. 26

[29] Golang: *go-shellwords.* https://github.com/mattn/go-shellwords, acesso em 2022-20-01. 26

[30] OpenSSH: *Openssh client.* https://linux.die.net/man/1/ssh, acesso em 2022-20-01. 29

[31] Gervais, Curtis: *Setup and configure apache virtual hosts.* https://www.codementor.io/@curtisgervais/setup-and-configure-apache-virtual-hosts-79kt2nuy6, acesso em 2022-20-01. 30

[32] Cloudflare: *O que são registros de dns?* https://www.cloudflare.com/pt-br/learning/dns/dns-records/, acesso em 2022-20-01. 30

[33] page, Linux/Unix programming: *hosts(5) - linux manual page.* https://man7.org/linux/man-pages/man5/hosts.5.html, acesso em 2022-21-01. 31

[34] TechRepublic: *Get to know the linux hosts file.* https://www.techrepublic.com/article/get-know-linux-hosts-file/, acesso em 2022-21-01. 31

[35] Harris, Joe: *How to configure bind9 as a primary dns server on ubuntu 20.04.* https://serverspace.io/support/help/bind9-as-a-primary-dns-server-on-ubuntu/, acesso em 2022-21-01. 32

[36] Salter, Jim: *Understanding dns—anatomy of a bind zone file.* https://arstechnica.com/gadgets/2020/08/understanding-dns-anatomy-of-a-bind-zone-file/, acesso em 2022-21-01. 32