UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

IOT HONEYNET COM EMULAÇÃO DA INTERNET

DOUGLAS BAPTISTA DE GODOY

ORIENTADOR: PROF. DR. HERMES SENGER

São Carlos – SP

Fevereiro/2019

UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

IOT HONEYNET COM EMULAÇÃO DA INTERNET

DOUGLAS BAPTISTA DE GODOY

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação, área de concentração: Sistemas Distribuídos e Redes de Computadores
Orientador: Prof. Dr. Hermes Senger

São Carlos – SP

Fevereiro/2019

Centro de Ciências Exatas e de Tecnologia
Programa de Pós-Graduação em Ciência da Computação

Folha de Aprovação

Assinaturas dos membros da comissão examinadora que avaliou e aprovou a Defesa de Dissertação de Mestrado do candidato Douglas Baptista de Godoy, realizada em 25/02/2019:

Prof. Dr. Ricardo Menotti
UFSCar

Prof. Dr. Cesar Augusto Cavalheiro Marcondes
ITA

Prof. Dr. Hermes Senger
UFSCar

Certifico que a defesa realizou-se com a participação à distância do(s) membro(s) Hermes Senger e, depois das arguições e deliberações realizadas, o(s) participante(s) à distância está(ao) de acordo com o conteúdo do parecer da banca examinadora redigido neste relatório de defesa.

Prof. Dr. Ricardo Menotti

Dedico a todos aqueles que estiveram presentes no desenvolvimento deste trabalho.

AGRADECIMENTOS

Agradeço primeiramente a Deus, por mais esta conquista em minha vida.

Agradeço ao Centro Paula Souza e a todas as pessoas envolvidas neste processo, que assim tornaram possível a realização deste mestrado.

Agradeço ao meu orientador Prof. Dr. Hermes Senger e ao Prof. Dr. Cesar Augusto Cavalheiro Marcondes pelos ensinamentos, ajudas e oportunidades dadas, bem como o inestimável auxílio e apoio dado com muita competência e ética durante a confecção deste trabalho.

Agradeço ao Emerson Rogério Alves Barea pelas importantes contribuições e ajudas com a parte técnica deste trabalho, sobretudo pelo conhecimento compartilhado.

”Inteligência é a capacidade de absorver informação em tempo real. De fazer perguntas que façam sentido. É ter boa memória. É traçar pontes entre assuntos que não parecem estar relacionados e inovar ao fazer essas conexões”

”É ótimo celebrar o sucesso, mas mais importante ainda é assimilar as lições trazidas pelos erros que cometemos”

Bill Gates

RESUMO

Este trabalho argumenta que o crescimento do número de dispositivos de Internet das Coisas (IoT) em nossas vidas (ex. Amazon Echo, câmeras, etc), bem como seu crescente poder computacional, desperta o interesse de hackers a atacá-los. Em sua maioria, esses ataques visam obter lucros, espionagem ou ativismo. Entretanto, apesar de anos de pesquisa e experiência, ainda não produzimos sistemas computacionais com programação segura o suficiente para impedir tais ataques em larga escala. Em geral, as técnicas empregadas são posteriores ao ataque, como a detecção do ataque e a análise do *malware*, onde são empregadas ferramentas capazes de realizar processos que permitem monitorar as interações do *malware* com o ambiente. Esses processos podem ser de dois tipos: (I) análise estática que é o processo de analisar o *malware* sem executá-lo; (II) análise dinâmica que executa o *malware* em ambiente controlado e monitora suas interações. As ferramentas de captura, tais como *honeypots* e *honeynets*, requerem um ambiente controlado e esse é o tema central de nosso trabalho, especialmente voltado para os dispositivos IoTs. Dessa forma, propomos uma arquitetura de *honeynet* que é capaz de identificar os ataques e as interações dos ciberataques com o seu controle, em que partimos da premissa que tais interações são feitas por meios de endereços em listas negras. Além disso, a execução do *malware* deve ser feita por processo similar aos de dispositivos de IoT. Finalmente, a arquitetura precisa ser autossuficiente e estar em um ambiente controlado, de modo a garantir que sua execução não gere um ataque real na internet, mas que a reproduza por meio de emulação. Em suma, foi desenvolvido uma prova de conceito com redes definidas por *software* (SDN) e os resultados mostram que a arquitetura é autossuficiente e seu ambiente controlado e escalável.

Palavras-chave: Análise, Ataque, *Honeynet*, *Honeypot*, IoT, *Malware*

ABSTRACT

This work argues that the growth in numbers of IoT (Internet of Things) in our lives (eg Amazon Echo, cameras, etc.), as well as their increasing computing power, arouses the interest of hackers and consequently, their attacks. Most of these attacks are aimed at making profits, espionage or activism. However, despite years of research and experience, we have not yet produced computer systems with enough programming safety to prevent such large-scale attacks. In general, the techniques employed are post-attack, such as attack detection and *malware* analysis. The tools used in this analysis can execute processes that allow you to monitor the interactions of the *malware* with the environment. These analysis can be of two types: (I) static analysis, which is the process of analyzing *malware* without executing it; (II) dynamic analysis that executes *malware* in a controlled environment and monitors its interactions. Capture tools, such as *honeypots* and *honeynets*, require a controlled environment and this is the central theme of our work, focused on IoTs. Thus, we propose a *honeynet* architecture able to identify the attacks and interactions of the cyber attacks thru its control, in that we start from the premise that such interactions are made through addresses in black lists. In addition, the *malware* must be executed by a process similar to that of the IoT devices. Finally, the architecture needs to be self-sufficient and to be in a controlled environment, to ensure that its execution does not generate a real Internet attack, but replicate it by emulation. A proof of concept with software-defined networks (SDN) was developed and the results show that the architecture is self-sufficient, its environment controlled and scalable.

Keywords: Analysis, Attacks, Honeynet, Honeypot, IoT, Malware

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

Capítulo 1

INTRODUÇÃO

Este capítulo apresenta a introdução ao tema tratado neste trabalho. A Seção 1.1 contextualiza o tema principal; a Seção 1.2 trata das motivações e objetivos; e a Seção 1.3 descreve a estrutura deste documento.

1.1 Contexto

Nos últimos anos, foi perceptível o aumento da abrangência da utilização de dispositivos computacionais, comumente chamados de dispositivos de Internet das Coisas, para o monitoramento e a automatização em áreas até então pouco imaginadas. Essa automatização também veio acompanhada da necessidade de interconexão desses dispositivos através de redes de computadores privadas ou até mesmo pela internet, possibilitando seu gerenciamento e utilização remota. O desenvolvimento de toda essa estrutura, conhecida como Internet das Coisas (Internet of Things - IoT), tem sido bastante impulsionado principalmente pela evolução tecnológica de aparelhos com uso cotidiano como *smartphones* e TVs. Porém, a IoT é mais ampla e pode estar presente em diversas áreas tais como: indústrias, vendas, eletrodomésticos, medicina, entre outras.

Em relação à segurança, esses dispositivos de IoT apresentam vulnerabilidade. Na sua maioria, eles deixam a aplicação *web* ativa com a possibilidade de obtenção de controle (acesso remoto), monitoramento e troca de dados, o que as torna cada vez mais vulneráveis a ataques cibernéticos. Podemos destacar como exemplo as câmeras IP, estas câmeras por apresentarem diversos fabricantes e a fim de criarem produtos competitivos fazem uso de sistemas obsoletos, assim sua aplicação *web* poderá apresentar vulnerabilidade e assim estar acessível pela Internet. De fato, o site Shodan[1] é especializado em indexar *banners* de serviços e apresenta diversas

[1] http://www.shodan.io

câmeras nessa situação. E esse cenário de vulnerabilidade tende ao crescimento devido ao fato de a IoT estar sendo implantada em larga escala em *Smart Cities*. Pois acredita-se que as *Smart Cities* levem a um caminho promissor, pois elas buscam melhorar a vida urbana (DOWLING; SCHUKAT; MELVIN, 2017).

De acordo com o Grupo Gartner, os dados indicam que 6,4 bilhões de coisas já estavam conectadas em todo o mundo em 2016, um crescimento de 30% acima de 2015 e a projeção é que chegarão a 20,8 bilhões em 2020 (SIMPSON; ROESNER; KOHNO, 2017) (DOWLING; SCHUKAT; MELVIN, 2017) (GARTNER, 2017).

Tudo indica que os dispositivos IoT tendem somente a crescer e, com isso, os ataques a eles também. Portanto, o crescimento de equipamentos IoT e o aumento de ataques são problemas relevantes de pesquisa e, neste trabalho, propomos uma prova de conceito de uma *honeynet* para dispositivos de IoT com alta interatividade, onde eles são emulados e deixados expostos na Internet com o propósito de serem atacados e comprometidos. Além disso, criamos um ambiente controlado que emule a internet para analisar o comportamento dinâmico dos *malwares* e como são realizadas as ações e ataques por eles, através de monitoração de conexões e outras interações.

1.2 Motivação e Objetivos

Com o decorrer do tempo e o desenvolvimento tecnológico, percebe-se que a cada dia nos tornamos mais dependentes dos computadores e dos equipamentos tecnológicos. Assim, nos tempos atuais, passamos a ser dependentes da Internet, pois ela nos conecta ao mundo o tempo todo. Entretanto, apesar de anos de pesquisas e experiências, ainda somos incapazes de criar sistemas de computadores com programação segura ou mesmo medir adequadamente a segurança desses equipamentos.

No que diz respeito a dados gerais de segurança, o CERT (Centro de Estudos, Resposta e Tratamento de Incidentes de Segurança no Brasil), mantido pelo NIC.br e responsável por tratar de incidentes no Brasil, mostra a variedade de ameaças através de relatos dos incidentes reportados ao CERT conforme mostra a Figura 1.1, onde destacam-se os tipos de ataques que concentram-se em Scan, DoS, Web e Fraude.

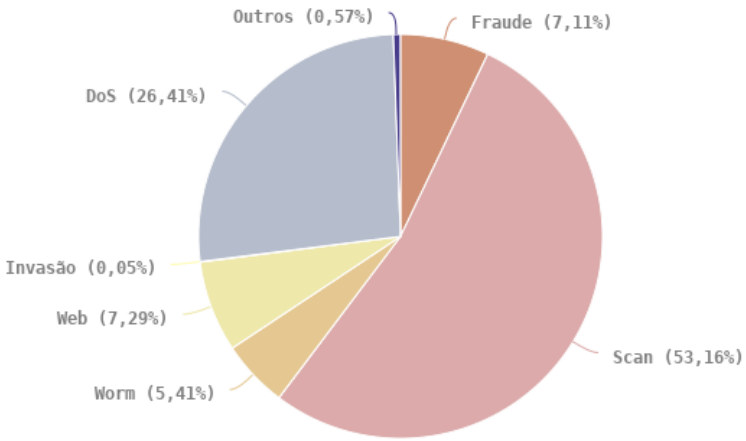


Figura 1.1: Tipos de ataques em 2017 - extraída do (Cert.BR,2017)

Outro dado relevante, em conjunto com os incidentes reportados ao CERT, mostrando quais são os maiores tipos de ataques, também é apresentado na Figura 1.2 de acordo com o CERT quais foram as portas com registros de maior quantidade de Scans.

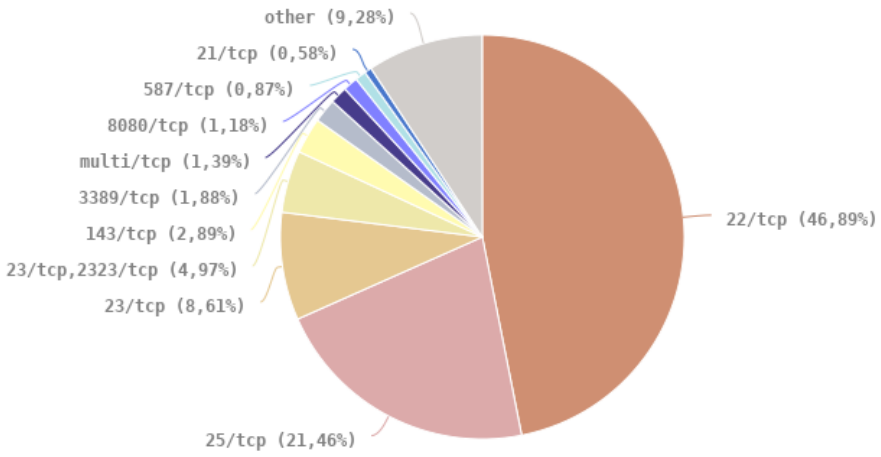


Figura 1.2: Scan por portas - extraída do (Cert.BR,2017)

De acordo com o Symantec, em seu relatório de ameaças, foi demonstrado que no período de 2016 a 2017, os ataques a dispositivos IoT tiveram um aumento de 600%. Para ter uma noção, considerou-se que em 2016 haviam cerca de 6 mil ataques e já no ano seguinte esses ataques aumentaram para 50 mil. Dessa forma, mostrou-se uma clara tendência de aumento significativo dessas ameaças. Ao relatar onde ocorreram esses ataques pelo endereço IP, pode-se fazer uma classificação de quais países esses ataques estão centralizados. O Symantec mostrou

que em 2017 tivemos: China, Estados Unidos e Brasil como os principais países de ataques centralizados. Outra informação importante é que a conscientização de segurança está muito fraca, pois o relatório aponta que os *logins* mais utilizados nesses ataques são os seguintes: *root*, *admin* e *enable*; com senhas frágeis, sendo aquelas utilizadas com maior frequência: system, sh e 123456. Especificamente com relação aos dispositivos de IoT, as principais ameaças detectadas pelos *honeypots* de IoT em 2017 são: Linux.Lightaidra, Trojan.Gen.NPE, Linux.Mirai, entre outras, os quais representam *malwares* de IoT. Isso demonstra que esses *malwares* estão ganhando tração. Finalmente, em relação a IoT, para a classificação das portas e serviços mais atacados temos a seguir a Tabela 1.1, onde de acordo com dados obtidos, é possível relacionar uma escala de frequências, de acordo com a quantidade de ataques sofridos.

Tabela 1.1: Classificação das portas atacadas(SYMANTEC, 2018)

Rank	Port	Percent
1	23/tcp (Telnet)	43.1
2	80/tcp (HTTP)	31.6
3	443/tcp (HTTPS)	7.7
4	2323/tcp (Telnet)	7.2
5	445/tcp (SMB)	5.8
6	22/tcp (SSH)	1.9
7	1900/udp (UPnP)	0.9
8	8080/tcp (HTTP)	0.8
9	2222/tcp (SSH)	0.2
10	21/tcp (FTP)	0.2

Podemos observar que como consequência dessas ameaças relatadas na Tabela 1.1, os serviços mais atacados em IoT são: Telnet, HTTP, HTTPS, SMB, SSH, entre outros. Ainda temos que destacar que os dispositivos mais atacados ou que sofreram mais ameaças nos *honeypots* da Symantec são os seguintes: Router, DVR (Digital Video Recorder), Network e Satellite Dish (SYMANTEC, 2018).

Conforme o Spamhaus Project (organização internacional sem fins lucrativos que rastreia *spam* e ameaças cibernéticas), foi identificado e publicado que em 2017 havia mais de 9.500 servidores com *botnets* de C&C (Command & Control) em 1.122 redes diferentes com o objetivo de controlar máquinas infectadas por *malwares* para extrair dados pessoais. Tais máquinas infectadas, controladas por criminosos cibernéticos, podem ser utilizadas para enviar *spam*, *ransomware*, ataques DDOS, fraude de *internet bank* ou até mesmo minerar criptomoedas.

A maioria destes servidores de C&C foram hospedados por cibercriminosos com o objetivo de ser controladores de *botnets*. Esses servidores foram adquiridos, em sua maioria, por identidades falsas, sendo que os dados apresentados não levam em conta as *botnet*s controladas a partir de servidores na *Dark Web.*

Ao analisar os dados, destacamos os dois primeiros provedores de serviços de internet com maior número de servidores de C&C de *botnets* são: Ovh.net da França, que em 2016 havia 395 e em 2017 tinha 402; e a Amazon do Estados Unidos com 54 em 2016 e 317 em 2017. Assim, em ambos os casos, houve crescimento. Outro dado que não podemos deixar de lado é a geo-localização destes controladores de *botnets* que se destacam, primeiramente Estados Unidos seguido da Rússia. Também, conforme Tabela 1.2 o número de C&C de *botnets* para IoT dobrou de um ano para o outro, passou de 393 em 2016 para 943 em 2017.

De acordo com o que conclui o artigo exposto pelo Spamhus, não há sinal de que o número de ameaças cibernéticas diminua com o passar do tempo. O grande aumento das ameaças de IoT em 2017 provavelmente continuará em crescimento no ano de 2018, assim conclui-se que proteger dispositivos IoT será um tópico central e deverá estar em destaque em 2018 (SPAMHAUS, 2018).

No mundo digital, existe todo tipo de pessoa: as pessoas mal-intencionadas denominadas de cibercriminosos têm como objetivo danificar e suspender serviços dos usuários. De acordo com (LYU et al., 2017), embora cibercriminosos tenham interesses nos usuários, esse não é o seu principal objetivo. Os cibercriminosos tendem a buscar negar o acesso aos dispositivos através de ataques de Negação de Serviço (DoS), e a buscar vulnerabilidade para controlar o dispositivo. Com essa última, os dispositivos tornam-se fonte de geração de ataques e seu impacto será maior. Portanto, o equipamento será usado como parte de uma *botnet* para atacar um alvo, que leva o dispositivo IoT a ser explorado para executar ações antiéticas e ilegais, conforme o desejo do invasor remoto.

Recentemente, uma infraestrutura específica de DNS gerenciada pelo Dyn[2] foi atacada com o uso de IoT, quando milhares de dispositivos geraram um ataque Distributed Denial of Service (DDoS) massivo. Isso aconteceu devido ao *malware* Mirai comprometer dispositivos IoT em todo o mundo. E esse tipo de ataque está sendo replicado, de modo que ataques de *Zero-day* em IoT estão sendo observados regularmente (BHUNIA; GURUSAMY, 2017) (JR., 2018).

Para contrabalencear o avanço das ameaças, soluções tecnológicas têm sido criadas e a variedade de ferramentas é enorme. Por exemplo, o repositório de revisão sistemática (chamado

[2]https://dyn.com

informalmente de *awesome*) de ferramentas de *honeypots*[3] apresenta dezenas de exemplos de *honeypots* baseados em desktop, IoT, ICS (Sistema de Controle Industrial), entre outros. Isso é uma fonte interessante de dados, pois existem vários projetos de propósito específico, como por exemplo, atendem portas específicas ou serviços específicos de uma determinada arquitetura ou um de determinado sistema operacional, não muito usual, mas relevantes em termos de IoT.

Tabela 1.2: Classificação de *Malware* (SPAMHAUS, 2018)

Rank	C&Cs	*Malware*	Note
1	1015	Downloader.Pony	Dropper / Credential Stealer
2	943	IoT *Malware*	Generic IoT *Malware*
3	933	Loki	Dropper / Credential Steale
4	437	Chthonic	e-banking Trojan
5	389	Smoke Loader	Dropper / Credential Stealer
6	325	JBifrost	Remote Access Tool (RAT)
7	293	Cerber	Ransomware
8	281	Gozi	e-banking Trojan
9	264	Redosdru	Backdoor
10	258	Heodo	e-banking Trojan
11	258	Adwind	Remote Access Tool (RAT)
12	211	Glupteba	Spam bot
13	203	TrickBot	e-banking Trojan
14	175	Dridex	e-banking Trojan
15	168	Neutrino	DDoS bot / Credential Stealer
16	162	ISRStealer	Backdoor
17	148	Worm.Ramnit	e-banking Trojan
18	148	Hancitor	Dropper
19	132	AZORult	e-banking Trojan
20	131	PandaZeuS	e-banking Trojan

Tendo em vista os dados levantados e o avanço da tecnologia na atualidade, percebe-se que a cada dia se torna mais comum a utilização de dispositivos IoT no cotidiano da população. Mas esses dispositivos têm mostrado que são suscetíveis a ataques e cada vez mais são alvos de *malwares* de infecção de larga escala. Atualmente, os ataques buscam cada vez mais os

[3]https://github.com/paralax/awesome-honeypots

dispositivos IoT, demonstrando assim um grande interesse por eles. Portanto, percebe-se que o interesse dos cibercriminosos nos dispositivos de IoT se dá devido ao grande poder computacional que esses dispositivos proporcionam e à suíte de protocolos neles disponível.

Outra premissa utilizada por cibercriminosos é que esses dispositivos, como forma de facilitar o gerenciamento remoto, têm poderosos interpretadores de comandos de fácil acesso, como relatado anteriormente, e além disso possuem credenciais inseguras e *software* com baixa atualização, o que os tornam vulneráveis.

É evidente que o número de ataques tende a crescer juntamente com o número de *malwares*. No que diz respeito aos dispositivos de IoT e à internet o mesmo se aplica em nossas vidas. Dessa forma, as perguntas que gostaríamos de responder são: estamos prontos para toda esta tecnologia? Estamos seguros dos ataques e dos *malwares*? Os *honeypots* e as *honeynets* são eficientes? E a internet, assim como estes dispositivos, são seguros o suficiente? Assim como na vida real, não podemos deixar a porta da nossa casa aberta para que qualquer um tenha acesso a ela, portanto, devemos agir da mesma forma com a Internet e com os dispositivos de IoT.

Com todos os fatos relatados no que diz respeito ao crescimento dos ataques e dos *malwares*, justamente por não existirem *honeypots* e *honeynets* eficientes, este trabalho preenche essa lacuna propondo uma arquitetura dinâmica e flexível de captura e análise de *malwares*. Essa arquitetura deverá ser uma estrutura modularizada, autossuficiente que suporte o núcleo de captura a partir de *honeypots* variados e que permita alta interatividade de modo a proporcionar a realização de estudos avançados em *malwares* para IoTs.

Essa arquitetura será representada pelo nome de **HonIoT** que faz referência à *Honeynet IoT*. Ela tem como intuito ser eficaz e com bom desempenho para estudar a segurança de dispositivos de IoTs, sendo que o cenário de execução acontece dentro de uma abstração de alto nível, buscando uma *honeynet* com emulação da internet, que dará suporte à pilha de protocolos.

Portanto, pretendemos analisar ações produzidas por *malwares* e buscar a flexibilidade para trabalhar com dispositivos IoT, que podem ter diferentes sistemas operacionais e arquiteturas de computadores. Esse aspecto é considerado no nível de arquitetura e a nossa proposta é que a arquitetura seja capaz de aceitar qualquer tipo de *honeypot* e analisar diferentes tipos de *malwares* em diferentes arquiteturas, inspirada por (PA et al., 2015), dentro do ambiente controlado e permitindo a interação com uma internet emulada.

Visando os aspectos descritos no nosso trabalho de pesquisa, buscamos relatar como os ataques são realizados e se eles obtêm sucesso, assim teremos a análise do *malware* para que possamos identificar suas ações.

1.3 Organização do Trabalho

O restante deste trabalho está organizado com a seguinte estrutura de 6 capítulos, sendo os seguintes:

O capítulo 2 apresenta os conceitos teóricos relacionados à proposta desenvolvida.

O capítulo 3 discute e traz críticas com relação aos trabalhos relacionados e relevantes à nossa proposta.

O capítulo 4 descreve toda a arquiterura necessária para desenvolvimento e levantamento de dados reacionados ao trabalho proposto.

O capítulo 5 descreve todo o desenvolvimento prático e de resultados obtidos com o trabalho que foi desenvolvido.

O capítulo 6, por sua vez, apresenta um resumo do que foi concluído com este trabalho, assim como ideias e sugestões para serem desenvolvidas em trabalhos futuros.

Capítulo 2

FUNDAMENTAÇÃO TEÓRICA

Neste capítulo, são apresentados os Fundamentos Teóricos para o desenvolvimento deste trabalho. A Seção 2.1 descreve o conjunto dos dispositivos IoT, juntamente com a arquitetura presente nestes dispositivos de IoT (Seção 2.1.1), também explora a definição de honeynets e honeypots na Seção 2.2. A Seção 2.3 apresenta o conceito de malware e diferentes técnicas de análise.

2.1 Fundamentos de IoT

IoT (Internet of Things) pode ser considerada uma extensão da Internet atual. Isso ocorre ao proporcionar que objetos do dia a dia se conectem à internet, tais como: TVs, Laptops, automóveis, *smartphones*, consoles de jogos, *webcams*, etc. Contudo, essas novas habilidades desses objetos comuns geram um grande número de oportunidades, mas essas possibilidades apresentam riscos e acarretam desafios técnicos e sociais. Portanto, essas interligações podem ser de diferentes tipos, como: fios de cobre, fibra óptica, ondas eletromagnéticas, entre outras. (SANTOS et al., 2016)

2.1.1 Arquitetura IoT

Os dispositivos de IoT têm uma ampla cobertura de fabricantes e integradores. Iniciando-se a partir das arquiteturas de *hardware* para IoT, as mais importantes são: ARM, MIPS e SPARC que detalhamos em seguida.

O processador ARM evoluiu de um projeto RISC (Reduced Instruction Set Computer) utilizado em sistemas embarcados. Atualmente, os sistemas embarcados podem ser encontrados em uma variedade de lugares tais como ambiente automotivo, eletrônico, controle industrial,

médico e automação de escritório. Um sistema embarcado é projetado para uma função dedicada e ele apresenta bom desempenho pela combinação de *hardware* e *software* específicos. Um bom exemplo de sistemas embarcados cada vez mais presentes no dia de hoje são os produtos eletrodomésticos (para consumo). Assim podemos destacar aparelhos de cozinha como refrigeradores, torradeiras entre outros. Já o MIPS é um processador, desenvolvido em Standford por John Hennessy. E finalmente, o SPARC é uma arquitetura da Sun Microsystems. Esse tipo de processador é inspirado na máquina RISC I de Berkeley (STALLINGS, 2010).

2.2 Honeypots e Honeynets

A motivação para a utilização de *honeypots* e *honeynets* cresce a cada dia, pois a quantidade de detecções obtidas pelo IDS (Instrusion Detection Systems) está diminuindo diante de técnica de evasão devido ao aumento do número de protocolos que usam criptografia e do alto índice de taxas de falsos positivos. Dessa forma, uma maneira adequada de continuar à frente na análise de ameaças virtuais é pelo uso de *honeypots* e *honeynets*.

Os *honeypots* são sistemas criados como chamariz para os *malwares*, com a intenção estabelecida de serem atacados e comprometidos. E, desse modo, registrar traços de teclas, detectar vulnerabilidades não apresentadas até o momento e detectar tráfegos suspeitos. Logo, em um *honeypot* não há nenhum sistema em produção e qualquer tentativa em contatá-lo tem o potencial de ser um possível ataque por projeto. Portanto, isso leva a uma redução de falsos positivos.

Estes sistemas são divididos em duas categorias: *Honeypots* de alta interatividade e baixa interatividade. Alta iteratividade são sistemas e serviços reais onde o atacante pode obter acesso total ao sistema e lançar novos ataques, já os *honeypots* de baixa interatividade também são sistemas reais, mas os serviços são simulados e o sistema de arquivos é de somente leitura. Assim, eles não podem ser explorados para obter acesso completo, são mais limitados, com um grau de contaminação menor, em caso de vazamento.

Do outro lado, temos as *Honeynets*, cuja abordagem é um pouco diferente. Elas tratam efetivamente em montar ambientes maiores que se pareçam com a rede institucional. Desse modo, é possível analisar padrões como tráfego dessa rede. No contexto atual, *honeypots* e *honeynets* devem ser utilizados em conjunto, pois o objetivo é identificar as ações realizadas pelos *malwares* tanto nos sistemas, quanto no tráfego de dados, tais como origem e destino (PROVOS, 2004)(MELO et al., 2011). As *honeynets* por sua vez, podem ser classificadas em 2 subtipos:

1. *Honeynet Real*: que são representadas pelo uso de equipamentos reais, por exemplo, no componente de *honeypots* desses ambientes, temos dispositivos reais de IoTs, ou ambientes Desktop ou até mesmo ICS (Sistema de Controle Industrial). Ness abordagem, as *honeynets* apresentam mecanismos de contenção, de alerta e de coleta de informações para estudo (CERT.BR, 2017b).

2. *Honeynets Virtuais*: faz uso de representação virtual dos dispositivos de IoT disponibilizados para infecção. Por exemplo, em um elemento como um Router WiFi ou até mesmo dispositivos inteligentes, faz-se uso de emulação tanto do *hardware* quando do sistema. A título de exemplificação, é possível emular a arquitetura de *hardware* a fim de garantir que esse dispositivo tenha a mesma capacidade real e que possa executar binários pré-compilados nesse ambiente. A vantagem de utilizar dispositivos IoT emulados é a relação aos custos representados com IoTs, desse modo, existe uma redução de custos no geral nos projetos.(CERT.BR, 2017b)

2.3 Malwares

Ao serem atacados, os dispositivos IoT são, em geral, infectados com código malicioso, os chamados *Malwares*. Esses softwares são desenvolvidos com intenção prejudicial, ou seja, feitos por um invasor malicioso. Os *malwares* podem ser de vários tipos ou podem ser um combinado de tipos classificados segundo o seu comportamento malicioso. São eles: Worm, Vírus, Cavalo de Troia, entre outros. Inicialmente, muitos anos atrás, a motivação dos criadores de *malware* era destacar algumas vulnerabilidades de segurança existentes ou apenas demonstrar habilidades técnicas de um grupo de *hackers*. Porém, com o passar do tempo, a motivação para essas criações foram mudando, tornando-se cada vez mais maliciosas, pois muitas vezes trata-se da perspectiva de ganhar dinheiro através delas (EGELE et al., 2008).

Para fins de terminologia, a Cartilha da entidade brasileira responsável por segurança na Internet, CERT.BR (Centro de Estudos, Respostas e Tratamento de Incidentes de Segurança no Brasil), define os *malwares de IoT* (Códigos Maliciosos) como programas criados e projetados unicamente para causar danos e executar atividades maliciosas em dispositivo IoT (SCHMIDT; GUARDIA, 2014).

2.3.1 Tipos de Malwares

Conforme comentado acima, existem vários tipos de *malware*. Todos eles diferem com relação aos diferentes objetivos e determinações. Os tipos mais frequentes de comportamentos de *malware* são:

1. *Worm:* este pode ser definido como "um programa que pode ser executado independentemente e pode propagar uma versão totalmente funcional de si mesmo para outras máquinas" (EGELE et al., 2008). Portanto, a propagação de *Worms* se dá através de cópias de si mesmo, pela rede, afetando assim outros computadores (SCHMIDT; GUARDIA, 2014).

2. Vírus: São programas de computadores ou apenas parte de um programa, que infectam outros sistemas através de cópias de si mesmos. Muitas vezes os Vírus costumam infectar seções de um arquivo hospedeiro, propagando-se com a distribuição deste arquivo (SCHMIDT; GUARDIA, 2014) (FILHO et al., 2011). De acordo com o CERT.br existem quatro variações de Vírus:

   (a) Vírus propagado por e-mail;

   (b) Vírus de script;

   (c) Vírus de Macro;

   (d) Vírus de telefone celular.

3. *Trojan Horse* (Cavalo de Tróia): Esses são tipos comuns de *malware*, onde muitas vezes, o modo de infecção envolve o despertar da curiosidade do usuário. Caso este venha a executar, acaba comprometendo o sistema. Portanto, este é um programa que possui códigos maliciosos dentro de uma programação aparentemente inofensiva (SCHMIDT; GUARDIA, 2014) (FILHO et al., 2011). Muitas vezes, esses *softwares* fingem ser úteis para executarem ações maliciosas, em segundo plano. Então, quando instalado o Cavalo de Troia, sua parte mal-intencionada pode baixar outro *malware* adicional, modificar configurações do sistema ou infectar outros arquivos no sistema. (EGELE et al., 2008)

4. *Backdoor*: Este *malware* permite que o sistema comprometido abra uma porta de conexão escondida e, desse modo, libere o livre acesso pelo invasor. Essa "porta dos fundos" pode tanto permitir ao atacante a manutenção de uma máquina comprometida, como também, o acesso por eventos, abrindo portas que permitem a conexão remota (SCHMIDT; GUARDIA, 2014).

5. *Spyware*: O *Spyware* é o software que captura e coleta informações confidenciais do sistema e transfere essas informações para o invasor ou mesmo envia os dados para um terceiro. Existem três tipos de Spyware: Keylogger, Screenlogger e Adware, dependendo do dado sendo capturado (EGELE et al., 2008) (SCHMIDT; GUARDIA, 2014).

6. *Bot* e *Botnet*: Este *malware* pode ser considerado similares ao *Worm*, mas tem capacidade para trabalhar em time, e ser controlado por um comando central remotamente (SCHMIDT; GUARDIA, 2014).

7. *Downloader:* Este *malware* é um programa que se conecta à rede para instalar e para obter um conjunto de outros programas maliciosos ou de ferramentas que levem ao domínio do hospedeiro. Normalmente os *Downloaders* são enviados em anexos de mensagens através de e-mails e, a partir de sua execução, propagam conteúdos maliciosos de uma fonte externa (FILHO et al., 2011).

8. *Dropper*: Este *malware* possui características semelhantes ao Downloader, porém a sua principal diferença é que o Dropper é considerado, além de que faz o download, também é um instalador, pois ele executa passos de instalação do código malicioso, e isso está compilado dentro dele (FILHO et al., 2011).

9. *Rootkit*: Este *malware* tem como característica principal, a capacidade de ocultar informações sobre ataques bem sucedidos à máquinas (ocultando a presença, por exemplo). Seu objetivo é permanecer atuando no sistema comprometido sem ser detectado, e dessa forma atacar o sistema modificando certos comandos de modo que seja imperceptível, ou seja, de modo camuflado, a maneira de executar ações maliciosas (EGELE et al., 2008).

2.3.2 Malwares de IoT

2.3.2.1 Botnet Mirai

Dando sequência, vamos aprofundar em *malwares*, dando destaque aos recentes *malwares* de IoT com capacidade de formar *botnets*. A *botnet* Mirai, por exemplo, é um alerta para a indústria de que é preciso dedicar mais recursos à proteção dos dispositivos da Internet. Pois, sem a proteção necessária, ela estará correndo ao risco de expor sua infraestrutura de Internet com ataques de negação de serviço.

Atualmente, a expansão e a grande popularidade da IoT e a grande quantidade de dispositivos tornaram-na uma poderosa plataforma de ataques dos cibercriminosos. Pode-se notar que os incidentes envolvendo dispositivos IoT são cada vez mais frequentes. Como os dispositivos

estão constantemente conectados à internet e têm *software* obsoleto, são alvos "fáceis" para aos cibercriminosos.

Recentemente, houve um caso reconhecido mundialmente do *Botnet* Mirai. Identificado pela primeira vez em agosto de 2016 pelo MalwareMustDie (grupo de pesquisa de segurança), suas variantes e imitações serviram como veículos para uma gama de ataques DDoS. Em setembro de 2016, certos *sites* foram atingidos com tráfego de 620 Gbps. Chegando em certos momentos do ataque DDoS usando o *Malware* Mirai a picos de 1,1 Tbps. Para conter essa invasão, o consultor de informática Brian Krebs (um dos profissionais que descobriu o Mirai) teve que segmentar a rede em nuvem da OVH[1] para lidar com o volume gigantesco de tráfego. Após a análise e lançamento em público do código-fonte do Mirai, os cibercriminosos fizeram atualizações e conseguiram infectar mais de 400.000 dispositivos conectados ao mesmo tempo, oferecendo-se para ataques de aluguel. Outros ataques famosos do Mirai continuaram a acontecer em outubro de 2016 contra provedores de serviço grandes e famosos que afetaram centenas de *sites* por várias horas, como por exemplo: Twitter[2], Netflix[3], Reddit[4] e GitHub[5].

Tecnicamente, o Mirai se espalha, a princípio, infectando dispositivos muito comuns do dia a dia como *webcams*, DVRs e roteadores que executem alguma versão BusyBox (uma versão embarcada de Linux). Atualmente as mutações de Mirai são geradas diariamente e o fato de que elas continuam se proliferando e causando danos reais mostra a dificuldade em combater esse tipo de ameaça.

A *botnet* Mirai é composta por quatro componentes principais: (1) O *software bot* é o *malware* que infecta o dispositivo, seu objetivo é espalhar a infecção para dispositivos vulneráveis e para atacar um servidor de destino assim que receber o comando do controlador do *bot* (botmaster). O (2) servidor de C&C fornece ao *botmaster* uma interface de gerenciamento centralizada que verifica a situação atual da *botnet* e permite o mesmo, comandar o ataque DDoS. A comunicação entre os *bots* e outras partes da infraestrutura são conduzidas através da rede Tor. O *malware* tem um carregador de *payload* (3) que distribui executáveis que atuam em diferentes plataformas (incluindo ARM, MIPS e x86) e assim que infectados comunicando-se diretamente para atacar novas vítimas. O servidor de C&C mantém um sistema de relatório de um banco de dados (4) detalhado sobre todos os dispositivos na *botnet*.

O Mirai busca endereços IP públicos aleatoriamente e tenta conexão através das portas 23

[1] https://www.ovh.pt/
[2] https://twitter.com
[3] https://www.netflix.com
[4] https://www.reddit.com
[5] https://github.com

ou 2323. Alguns grupos de endereços IP são excluídos da busca: Serviços Postal dos EUA, Departamento de Defesa americano, entre outros, provavelmente com a intenção de evitar chamar a atenção do governo.

E finalmente, um aspecto interessante é que em comparação com outros *malwares* similares, percebe-se que o Mirai, diferentemente, não tenta evitar a detecção. Pois quase todas as fases da infecção deixam rastros que podem ser reconhecidos através de uma análise simples da tráfego de rede.

O fluxo de ação da Botnet Mirai pode ser observada na Figura 2.1 onde o operador da rede controla os *bots* à distância por meio de comandos e atualizações de binários para criar um ataque ao servidor alvo.

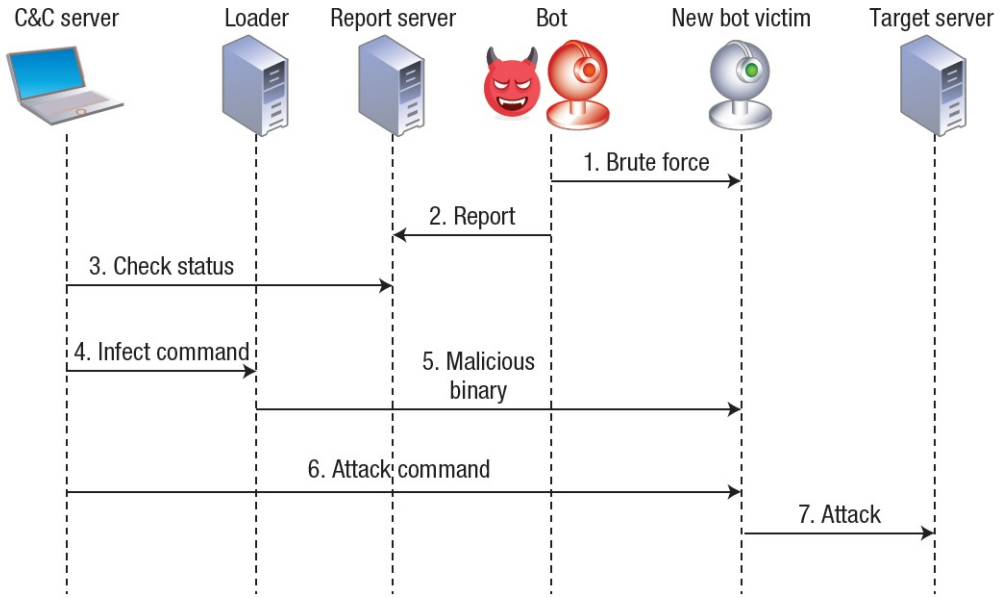


Figura 2.1: Botnet Mirai Operação e Comunicação - extraída de (KOLIAS et al., 2017)

Os Ataques DDoS do Mirai ocasionaram diversos impactos, destacando os riscos advindos dos dispositivos IoT. A facilidade de infecção e a estabilidade da população de *bots* são fatores atraentes aos atacantes. Portanto, existem motivos para os dispositivos IoT serem os principais alvos na criação de novas *botnets*, são eles:

- Operação constante e discreta: ao contrário dos *laptops* e *desktops* que possuem frequentes ciclos *on-off*, a maioria dos dispositivos IoT operam 24/7 e em diversos casos não são sequer reconhecidos pelos leigos como dispositivos de computação.

- Proteção contra fraudes: na tentativa de se inserir rapidamente no mercado de IoT, muitos fornecedores negligenciam a segurança necessária em favor da facilidade de uso e acessibilidade.

- Pouca manutenção: na maioria dos dispositivos IoT, os usuários e a rede administradora se "esquecem" de atualizá-los ou de realizar manutenções adequadas, em função de dificuldade de acesso a esses equipamentos e ao grande número deles. As únicas vezes que recebem manutenção é quando param de operar ou não estão trabalhando corretamente.

- Tráfego de ataque considerável: dispositivos IoT são poderosos e bem situados para produzir DDoS. Eles têm a capacidade de produzir ataque com tráfego comparáveis aos sistemas de *desktop* modernos e são em maior número.

- Não interagem ou têm baixa interação com a interface de usuários: os dispositivos IoT tendem a exigir alguma interação mínima com o usuário. Dessa forma, as infecções são muito mais propensas a passar despercebidas. Até mesmo quando são notadas, não há nenhum caminho fácil ao usuário para consertar o problema.

Atualmente, os ataques de variantes do Mirai estão cada vez mais sofisticados e estão surgindo em uma proporção alarmante. Sendo o *malware* multiplataforma, pois é executado em diversas plataformas e normalmente sendo ele leve o suficiente para executar em uma pequena quantidade de RAM, isso torna a possibilidade de atingir uma grande quantidade de dispositivos um fato real (KOLIAS et al., 2017).

2.3.3 Métodos de Infecção

Para entender o modo operante dos *malwares* de IoT, precisamos descrever os métodos utilizados por *softwares* maliciosos para infectar e evoluir a ameaça dentro de um sistema. Esses métodos também são conhecidos como vetores de infecção.

Esses métodos são caracterizados pela comunicação em rede e podem ser descritos nos seguintes termos:

- Exploração de serviços vulneráveis através da rede: Os serviços de rede em execução fornecem aos clientes recursos e serviços que são partilhados, isso os torna vulneráveis, permitindo muitas vezes que o invasor execute códigos maliciosos na máquina em que o serviço esteja sendo fornecido. Isso torna a exploração de serviços de rede o método preferido para a infecção por *Worms* (EGELE et al., 2008) (FILHO et al., 2011).

- *Drive-by Downloads:* Os *Downloads Drive-by* segmentam o navegador da *web*, portanto ao explorar uma vulnerabilidade no aplicativo de navegador da *web*, um *download* por unidade é capaz de buscar o código malicioso da *web* e executá-lo na máquina a ser infectada. Normalmente, isso ocorre sem nenhum tipo de interação com o usuário (EGELE et al., 2008) (FILHO et al., 2011).

Destacamos que além dos métodos citados anteriormente, os *malwares* fazem uso de C&C, a fim de obter *playloads* necessários para a proliferação de um ciberataques. Estes C&C tendem a ser ferramentas de controle, servir como repositório de dados ou até mesmo repositório de variantes de *malware*. Uma das formas de obter estes endereços de C&C é pela técnica de análise de binário. Desse modo, é possível obter as requisições encaminhadas a destinos incomuns. Nesse contexto, há diversos *sites* na internet que disponibilizam lista negras de C&C também conhecidos como inteligência de fontes abertas (*Open Source Intelligence*, OSINT). Essas listas são empregadas por diversos auditores de *PenTest*, busca obter informações de fontes abertas tais como: Shodan, Google Hacking Database (GHDB). Nessas fontes, também é possível obter o histórico das ameaças, visto que elas constantemente evoluem. Porém, o importante é sempre fazer a utilização de uma base atual, pois casos antigos podem ter sido consertados. Entre os sites de listas negras, destaca-se o Verizon DB [6], entre outros (DUFFY, 2016).

2.3.4 Análise de Malwares

Finalmente, para contextualizar a atuação das *honeynets* e o trabalho de análises feito nesses ambientes, apresentamos as abordagens de análise de *malwares*. Em termos gerais, a análise de *malwares* é uma área que cresce cada vez mais, devido às ameaças, e portanto, a análise de *malware* é caracterizada em alguns passos tais como: analisar o artefato de modo a descobrir se o mesmo é uma nova ameaça e compreender os riscos e as intenções do *malware*, esses são alguns dos aspectos considerados.

Quando falamos de *malware*, devemos pensar em dois lados do problema. Temos quem cria o *malware* e quem analisa o *malware* com a intenção de proteger. Essa é uma uma luta travada por ambos os lados. No que diz respeito à criação do *malware*, procura-se criar *malwares* com ações para disfarçar as intenções maliciosas. Técnicas para evitar que o *malware* seja analisado e até mesmo técnicas de automodificação ou geração de códigos e até mesmo a detecção do próprio ambiente de análise. Essas precauções são para que não seja possível descobrir o seu comportamento. Do outro lado, busca-se entender o funcionamento, sua atuação no sistema operacional, quais são as técnicas de ofuscação utilizadas, quais os fluxos de dados, quais

[6] http://verizonenterprise.com/DBIR/

operações de rede e se possui *download* de outros arquivos. Portanto, com tudo isso, os analistas devem possuir ferramentas e técnicas adequadas para analisar os *malwares*.

Uma das técnicas mais diretas e que não necessita de um ambiente é a análise estática. Ela trata em extrair características sem executar o código malicioso com um conjunto de ferramentas e técnicas, tais como geração do *hashes*, verificação de padrões binários e checagem do seu código através de técnicas de análise de *strings*, *disassembling* e engenharia reversa.

Por sua vez, a análise dinâmica do *malware* lida de forma diferente da análise estática. A ideia é colocar o *malware* em execução, por meio de ambientes controlados, e assim realizar o monitoramento aprofundado das chamadas de funções do sistema operacional, dos comportamente por meio de observação de entradas e saída de dados na interface de rede, ações internas realizadas no processo, modificações em registros e até mesmo em arquivos (FILHO et al., 2011).

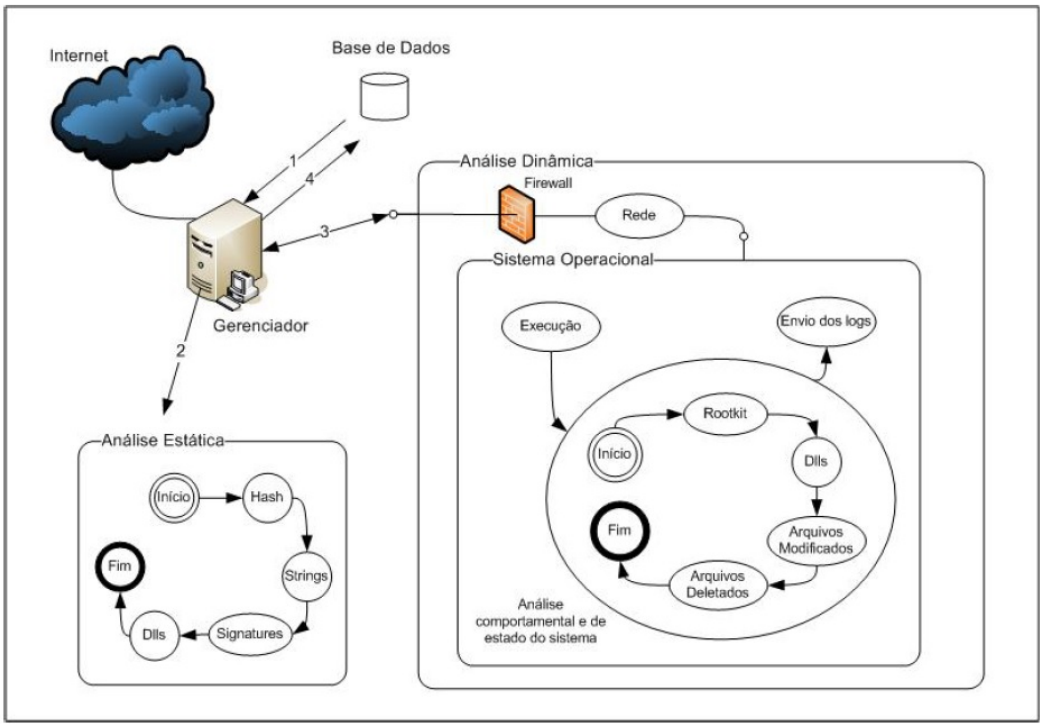


Figura 2.2: Análise Estática e Dinâmica - extraída de (FILHO et al., 2011)

Alguns ambientes de código aberto como o Cuckoo Sandbox se caracterizam como um ambiente híbrido que permite fazer investigações estáticas e dinâmicas. A Figura 2.2 apresenta simplificadamente essas operações com o gerenciador controlando o que tem que ser feito. Em nossa proposta, estaremos focando em um ambiente que permita a análise dinâmica, com maior nível de detalhe e com alta interação, através da criação da rede emulada.

2.4 Redes Definidas por Software

Tradicionalmente, *Switches* e Roteadores apresentam as seguintes características internas: Plano de Dados e Plano de Controle, que foram desenvolvidos para serem fortemente acoplados a uma forma fechada e proprietária. A Figura 2.3(a) representa a visão geral desse modelo.

De forma geral, a utilização de Redes Definida por *Software*, também conhecido como SDN, permite a centralização programável do plano de controle e a abstração do plano de dados, em que o plano de controle e o plano de dados são separados para que os operadores e/ou serviços de rede possam controlar e gerenciar diretamente seus próprios recursos, nesse contexto, podemos destacar que o SDN é essencial para operadores de redes e provedores de serviços que possam controlar e gerenciar com mais facilidade e eficiência suas redes (SHIN; NAM; KIM, 2012).

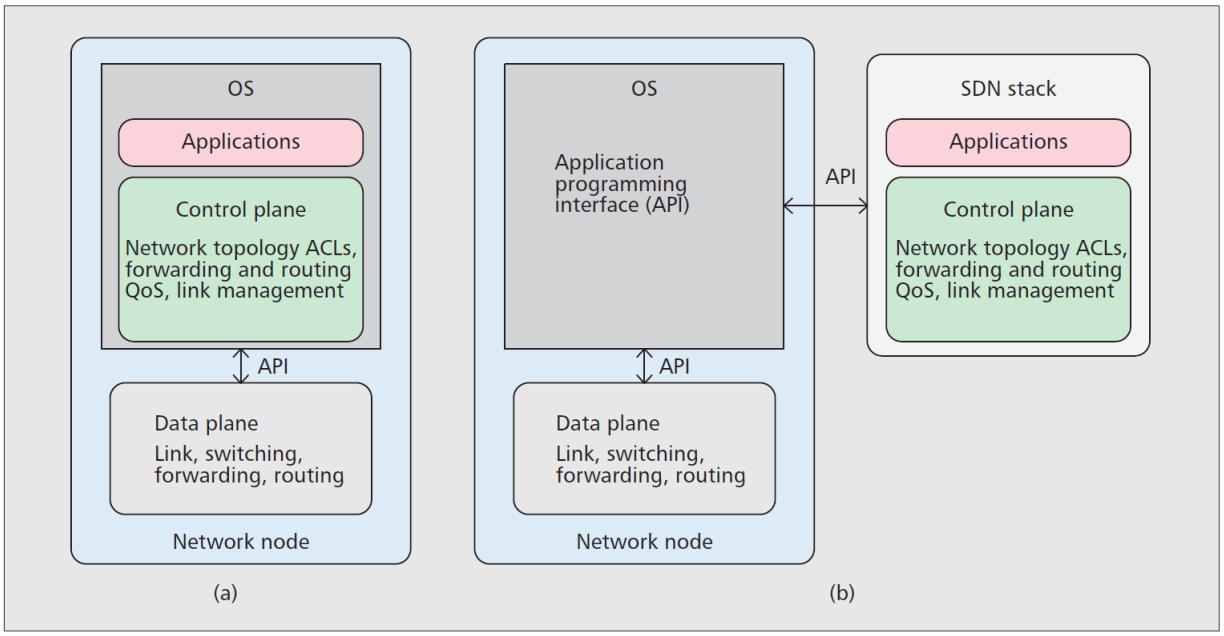


Figura 2.3: Visualização de rede tradicional comparada com a exibição de rede SDN: a) abordagem tradicional (cada nó da rede possui seu próprio controle e plano de gerenciamento); b) abordagem SDN (o plano de controle é extraído do nó da rede). - extraída de (SEZER et al., 2013)

Redes Definidas por *Software* são uma nova abordagem de rede que buscam simplificar as operações de rede, reduzir custos e acelerar a entrega de serviços, sendo que seus atributos incluem: (1) separação do plano de controle e dados; (2) centralizado, equipamentos de rede programável; e (3) Suporte de redes virtuais múltiplas e isoladas. A Figura 2.3(b) representa a visão geral e a distribuição do plano de dados e controle, assim o SDN proporciona a abertura que dá aos seus clientes a possibilidade de escolher a sua rede, algumas das principais propriedades do SDN são a arquitetura aberta, as interfaces, os APIs, etc (SHIN; NAM; KIM, 2012).

Redes definidas por *Software* tiveram início com a criação do protocolo OpenFlow (MCKEOWN et al., 2008), pois até o momento não era possível interagir com o fluxo de dados, a final os *Switches* e roteadores trabalham de modo a garantir seu funcionamento através de regras definidas pelos fabricantes, tais regras bem definidas, que assim garantem seu funcionamento, nesse sentido, podemos destacar a Internet. Com o advento de Redes Definidas por *Software* também conhecido como SDN, juntamente com o protocolo OpenFlow, nos trouxeram a possibilidade de gerenciar o fluxo e tomar decisões para onde este fluxo será encaminhado.

Assim, o plano de dados continua o mesmo, já o plano de controle passa a ser gerido por meio de uma interface e independente do fornecedor do elemento, pelo OpenFlow, API de acesso à tabela pertencente ao controlador que assim define o comportamento do encaminhamento de baixo nível de cada elemento de encaminhamento (*switch*, roteador, ponto de acesso ou estação base). Podemos destacar o OpenFlow que define uma regra para cada fluxo, se um pacote corresponder a uma regra, as ações correspondentes serão executadas (por exemplo, descartar, encaminhar, modificar ou enfileirar) (LANTZ; HELLER; MCKEOWN, 2010).

A principal consequência da SDN é que a funcionalidade da rede é definida após sua implantação, sob o controle do proprietário e do operador da rede. Novos recursos podem ser adicionados em software, sem modificar os switches, permitindo que o comportamento evolua em velocidades de software, em vez de padrões (LANTZ; HELLER; MCKEOWN, 2010).

2.5 Considerações Finais

Internet of things nos trouxe uma nova visão computacional que fundamentou nossa pesquisa. Dessa forma, podemos verificar que houve um aumento dos riscos e que existe a necessidade de se proteger as informações dos usuários. Portanto, fizemos o levantamento dos conceitos fundamentais que aprofundaremos neste trabalho como *honeypot* e *honeynet* e seu funcionamento interno. Esses ambientes permitem estudar os *malwares* que são criados pelos cibercriminosos. Também detalhamos um *malware* de IoT e o impacto que esse novo tipo de ameaça tem nos dias atuais.

Capítulo 3

TRABALHOS RELACIONADOS

Este capítulo apresenta e detalha trabalhos similares ao que estamos propondo. A Seção 3.1 apresenta critérios levantados para a escolha destes trabalhos e a Seção 3.2 descreve cada uma dessas ferramentas.

3.1 Definição e escolha dos trabalhos relacionados

Nesta Seção, buscamos apresentar os critérios utilizados para escolha dos trabalhos antecendentes (Seção 3.2). Portanto, iremos relatar os critérios utilizados.

Nossa pesquisa foca em *honeynets* com foco em IoT, portanto, um ponto de partida foi o trabalho IoTPOT (PA et al., 2015). Ele foi desenvolvido em 2015 e apresenta alguma similaridade ao nosso trabalho desenvolvido com relação à execução multiplataforma de dispositivos de IoT. Dando sequência, do ponto de vista nacional, foi feita uma busca de propostas que combinam *honeypot* e *honeynet* no Brasil e encontramos um projeto de larga escala desenvolvido pelo CERT.br chamado de "Distributed Honeypots Project"(CERT.BR, 2017a), cuja interface de captura está baseada no Honeyd (PROVOS, 2004) que inspira o nosso *design* para *banners* de IoT. Finalmente, a nossa abordagem tem a intenção de possibilitar que para a *honeynet* seja utilizada qualquer *honeypot* em termos de *backend* então, propostas similares que buscam essa flexibilidade foram estudadas. Entre elas, temos as baseadas em *containers* como o Qebex (SONG; HAY; ZHUGE, 2018) e o Dockerpot (POPERESHNYAK et al., 2018). Como os *containers* são ambiente de execução padronizados para diferentes arquiteturas, eles se tornam ideais para execução de *honeypots* desenvolvidos para um nicho específico. Finalmente, para finalizar o escopo de trabalhos relacionados, dado que nosso desenvolvimento foi feito em Python em função do uso de um controlador de redes definidas por software, posicionamos a busca de trabalhos relacionados para entender o funcionamento dos *honeypots*. Desses destacamos nesse texto, os

seguintes: HoneyWRT , MTPot e Honeyport.

3.2 Trabalhos Relacionados

3.2.1 Honeyport

Honeyport[1][2] apresenta um *honeypot* simples e uma grande portabilidade, mas ao mesmo apresenta sua implementação em Python e também em Bash. Suas funções e implementação são de fácil entendimento, os script Bash e Python têm praticamente as mesmas funções, mas o script em Python é possível executar no Mac OS, Linux e Windows, garantido ainda mais a portabilidade em sistemas.

Uma das vantagens é que o mesmo analisa as requisições enviadas a ele. Essas requisições são analisadas se pertencem à *whitelist*, requisições que não pertençam à whitelist serão bloqueadas pelo IPFW (Mac), pelo Iptables (Linux) e pelo Firewall (Windows). Também é possível utilizar listas negras disponíveis na internet. No contexto de desvantagens, podemos dizer que o *Honeyport* trabalha com apenas uma porta, assim disponibilizar recursos para monitorar somente uma porta não é muito vantajoso. Outro fato que também classificamos como desvantagem é por esta ferramenta não ser interativa, ao disponibilizar a porta 23 de Telnet, não é possível interagir, pois o *Honeyport* deixa a porta disponível, mas ao conectar essa porta é fechada, evitando a possibilidade de coletar informações com tentativas de credenciais e comandos.

3.2.2 MTPot

MTPot[3] é um *honeypot* desenvolvido pelo Cymmetria[4] de código aberto, sua implementação é através do Python e seu desenvolvimento foi pensado no ciberataque do Mirai descrito em capítulos anteriores. Uma de suas vantagens é que ele além de ficar ouvindo a porta 23 de Telnet, também implementa a coleta de credenciais em tentativas de ataques, tais como: usuário e senha. Uma desvantagem dessa plataforma é que podemos ouvir somente a porta 23, assim deixando este *honeypot* exclusivo para Telnet, o que não o torna eficiente por deixar outras portas de lado. A Figura 3.1 demonstra a visão geral de funcionalidade do MTPot.

[1] https://github.com/securitygeneration/Honeyport
[2] https://www.securitygeneration.com
[3] https://github.com/Cymmetria/MTPot
[4] https://cymmetria.com

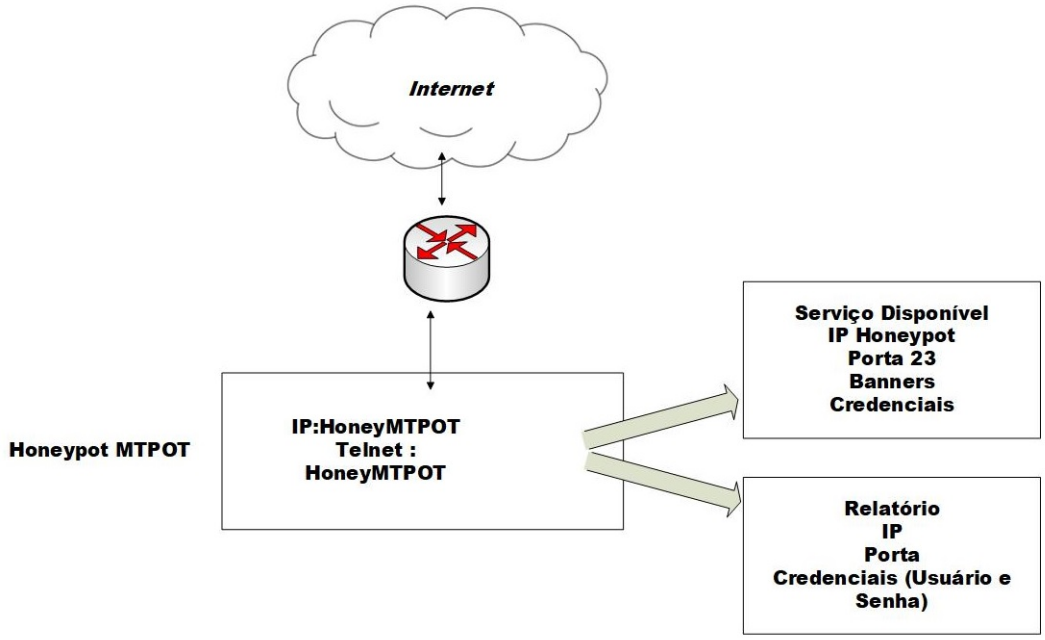


Figura 3.1: Visão geral do MTPot de Telnet

A Cymmetria além de apresentar um *honeypot* abordando o Mirai, também implementou um *honeypot* baseado no Apache2[5], chamado de StrutsHoneypot[6] este *honeypot* detecta e / ou bloqueia a exploração do struts CVE 2017-5638, a exploração desta falha permite a execução de comandos remotos nos servidores que, por sua vez, possibilitam controle total sobre o servidor com tal vulnerabilidade. Por outro lado, essa ferramenta apresenta um recurso bastante utilizado por profissionais da área de TI. Os Contêineres para esta implementação da Cymmmetria fazem uso da ferramenta Docker, o *StrutsHoneypot* cria instâncias em contêiner com a porta 80.

3.2.3 HoneyWRT

HoneyWRT[7] é um *honeypot* desenvolvido em Python de baixa interação, este *honeypot* busca emular serviços e portas e não os dispositivos IoT. A grande vantagem desta ferramenta é possuir uma modularidade, uma gama de serviços e portas e ainda este *honeypot* pode ser extensível ao ponto de vista de portas, assim o torna um *honeypot* completo. Ele oferece serviços tanto para TCP quanto para UDP.

Porém, como desvantagem o *HoneyWRT* apresenta diversos módulos e APIs para seu funcionamento, no que se refere a ser extensível, pode se dizer que para cada novo serviço, deve-se inserir os *banners* correspondentes.

[5]https://www.apache.org
[6]https://github.com/Cymmetria/StrutsHoneypot
[7]https://github.com/CanadianJeff/honeywrt

3.2.4 Qebex

Qebex é um *honeypot* baseado em QEMU[8] de alta interação, pois nos últimos anos, diversas ferramentas de *honeypots* de baixa interação estão ficando cada vez mais poderosas, mas no contexto de alta interação, isso tem sido um pouco lento como podemos citar o antecessor ao Qebex o Sebex[9]. Com o lançamento de Qebex o Sebex, teve sua última versão que apresenta alguns problemas tais como: falta de invisibilidade, tela azul do cliente, sistema e rede. Uma desvantagem do Qebex é que ele oferece suporte somente ao Windows, assim ficando limitado a uma única plataforma (SONG; HAY; ZHUGE, 2018).

3.2.5 Dockerpot

Dockerpot simula a operação de um *host* criando contêiner para cada nova conexão, como o próprio nome já diz, essa prova utiliza a ferramenta de virtualização Docker[10], que assim fornece um nível suficiente de isolamento do sistema. Um fato que pode ser considerado como vantagem ou desvantagem. Isso por se tratar de Docker, o Docker não usa contêineres desprivilegiados. No momento, isso representa acessar o nível superusuário dentro do contêiner que é equivalente ao acesso no sistema *host* (POPERESHNYAK et al., 2018).

3.2.6 Honeyd

Honeyd fundamenta os termos iniciais sobre *honeypot*, onde se define que um *honeypot* pode executar qualquer sistema operacional e um número qualquer de serviços. Esses *honeypots* podem ser *honeypots* físicos que representam máquinas reais com IPs reais e *honeypots* virtuais que se referem a máquinas virtuais ou emuladas, mas para o autor malicioso deve ficar evidente que esses sistemas são reais com tráfego de rede entre outros, a fim de quando um ciberataque utilizar ferramentas para um ataque, obtenha respostas verdadeiras. A Figura 3.2 apresenta o cenário computacional a que se refere ao Honeyd.

O Honeyd utiliza o conceito de *honeypots* Virtuais conforme citado no artigo *"A Virtual Honeypot Framework"*. Esse sistema tem suporte ao protocolo IP (STEVENS; WRIGHT, 1994), assim o Honeyd virtualiza comportamentos de rede do sistema operacional configurado e cria redes virtuais com topologias de roteamento, para que assim não haja dúvidas com relação aos ataques (PROVOS, 2004).

[8] https://www.qemu.org/
[9] https://projects.honeynet.org/sebek/
[10] https://www.docker.com/

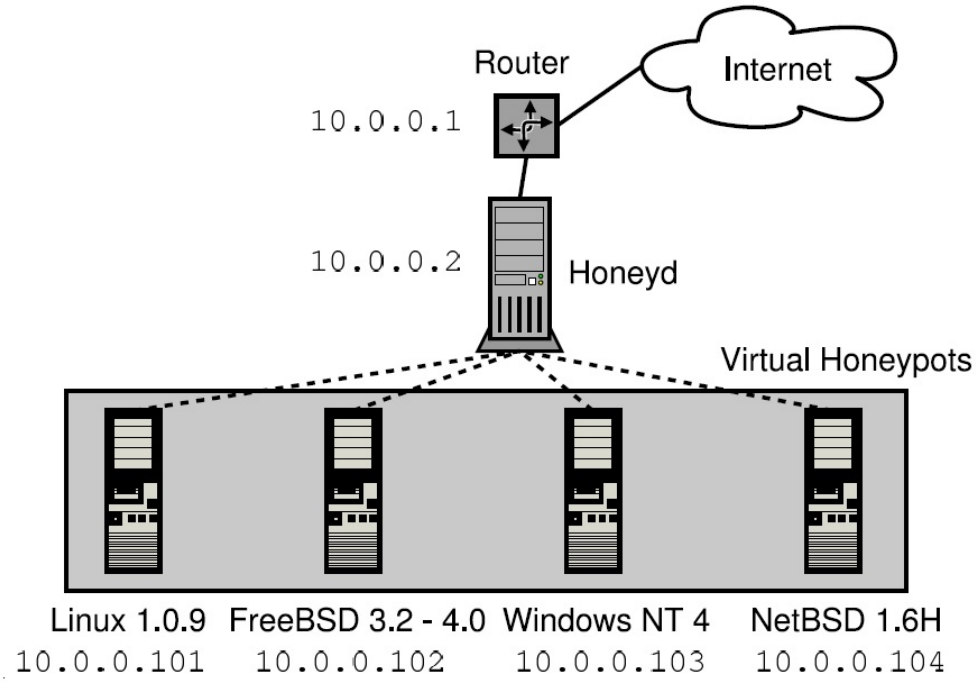


Figura 3.2: Cenário Computacional do Honeyd - extraída de (PROVOS, 2004)

3.2.7 IoTPOT

O IoTPOT é o trabalho que mais se aproxima desta pesquisa, pois ele apresenta as seguintes características: 1- Investigação dos ataques que tiveram início através de dados obtidos de uma rede do Japão que monitora mais de 270 mil endereços IP, assim a partir destes dados, foi possível verificar quais portas eram mais atacadas e identificar que tipo de sistema operacional estava sendo utilizado para realizar os ciberataques. Portanto, com as ferramentas, concluiu-se que 91% era Linux e também que havia dispositivos de IoT; 2- IoT *Honeypot* foi a solução encontrada a partir da investigação de que há necessidade da criação de um *honeypot* para dispositivos IoT, pois os *honeypots* existentes não desempenhavam esta formação, afinal dispositivos IoT trabalham com diferentes arquiteturas tais como MIPS e ARM.

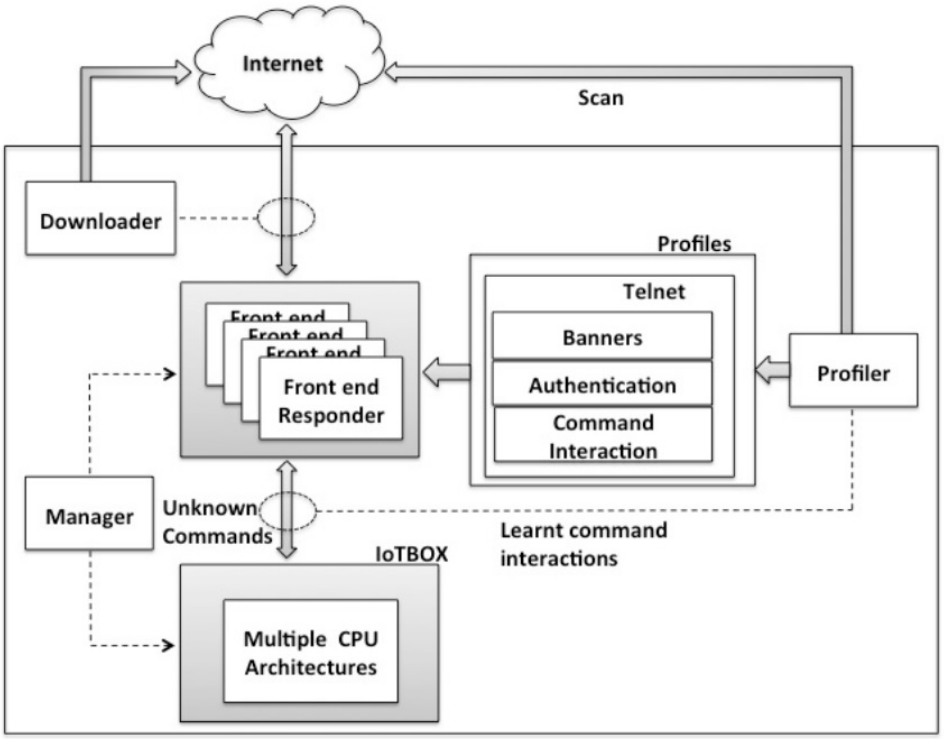


Figura 3.3: Visão Geral do IoTPOT - extraída de (PA et al., 2015)

A Figura 3.3 apresenta a visão geral do IoTPOT, sendo que ele conta com passos essenciais para o relatório de ciberataques. O primeiro é a intrusão onde o cibercriminoso encontra sistemas de IoT emulados reais para eles e assim o IoTPOT coleta as interações dos ciberataques; o segundo passo trabalha com a infecção, pois após o cibercriminoso obter acesso a sua intenção é controlar e produzir novos ataques para isso realiza *downloads* de *malwares*; já o terceiro passo se refere ao monitoramento que além do *honeypot* (IoTPOT), conta com um poderoso Sandbox (IoTBOX) que lida com o *malware* de diferentes arquiteturas de CPU para a análise dinâmica do *malware* e assim verifica as interações realizadas por ele, tais como: *downloads* e ataques. Enfim, essa ferramenta é responsável pela análise do *malware* e suas ações (PA et al., 2015). Conforme desmonstrada na Figura 3.4.

De acordo com (PA et al., 2015), "Em mais de 15 sistemas de Sandbox pesquisados, nenhum suporta arquitetura de CPU diferente, como MIPS e ARM".

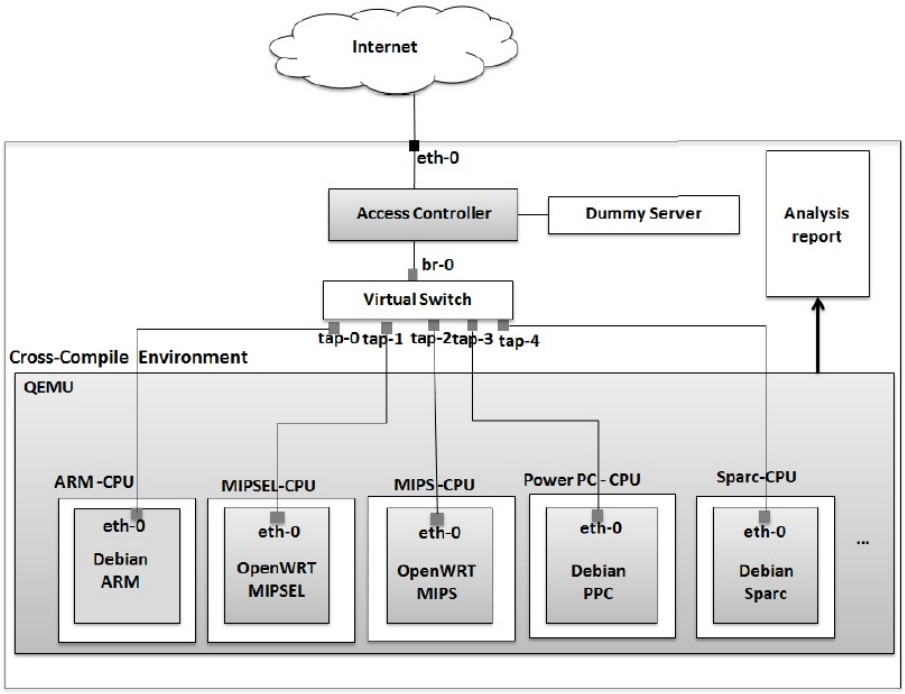


Figura 3.4: Visão Geral do IoTBOX - extraída de (PA et al., 2015)

Embora o IoTPOT esteja trabalhando com dispositivos de IoT, ele ainda apresenta desvantagem, pois o IoTPOT é um *honeypot* de baixa interação e não emula um sistema real para seu *Front End*, portanto, existe a possibilidade do *malware* não interagir com ele.

3.3 Conclusão da Pesquisa

Após a revisão bibliográfica, chegamos à conclusão de que o Honeyd atende uma gama de sistemas operacionais e diferentes tipos de serviços, mas seu foco principal é em *Desktop* e Servidores e não dispositivos de IoT. O Honeyd trabalha com portas de nosso interesse, é um *honeypot* de baixa interação. Temos outros *honeypots* conhecidos como Telnet senha, mas o seu foco principal é a coleta de senhas de Telnet e interações de comando que não são suportados (PA et al., 2015).

O IoTPOT analisa tentativas de ataques via Telnet e assim monitora as interações desses ataques. Sendo que logo após estas interações, os autores maliciosos, já com o usuário e senha do terminal, realizam o processo de infecção. Assim, o IoTPOT entra em ação novamente para monitorar as interações desse *malware* com a ajuda do IoTBOX que é um Sandbox. Isso significa que há um analisador dinâmico de *malware*.

Um fato importante a destacar é que a maioria dos *honeypots* existentes não oferecem su-

porte a dispositivos IoT e um outro fator interessante é que os ataques também são realizados nesses dispositivos, pois eles têm um alto poder computacional e estão disponíveis 24/7.

Podemos concluir que com o desenvolvimento da tecnologia, há uma mudança de paradigma da utilização de Desktop para dispositivos de IoTs, portanto percebe-se que cada vez mais no cenário de segurança são constantes os *honeypots*, pois até o momento eram destinados a *Desktops* tais como: o Honeyd e em outros trabalhos citados anteriormente, porém o IoTPOT trabalha com uma frente de ataques voltados para IoT.

No que diz a respeito à Sandbox o mesmo se aplica, pois há inúmeras ferramentas para *Desktop* e que na sua maioria tem o foco em ambientes Windows, mas IoTBOX desempenha a ação de analisar de forma dinâmica *malwares* para dispositivos de IoT.

Tabela 3.1: Classificação dos Trabalhos Relacionados

Trabalhos Relacionados	Conexão Internet	Suporta IoT	Interatividade
Honeyport	Não	Não	Baixa
MTPot	Não	Não	Baixa
HoneyWRT	Não	Não	Baixa
Qebex	Não	Não	Alta
Dockerpot	Não	Não	Baixa
Honeyd	Não	Não	Baixa
IoTPOT	Não	Sim	Baixa

Portanto, de acordo com a Tabela 3.1 podemos observar um comparativo direto dos prós e contras (vantagens e desvantagens) de cada trabalho relacionado acima. Assim como identificar que este trabalho foi desenvolvido com o objetivo de suprir algumas necessidades até então não analisadas ou visualizadas anteriormente.

Capítulo 4

ARQUITETURA DA HONEYNET PROPOSTA

Neste capítulo é apresentada a arquitetura da honeynet proposta, bem como há o detalhamento de seus módulos formadores.

4.1 Arquitetura

Para atingir os objetivos propostos neste trabalho, foi definida uma arquitetura modularizada de uma *honeynet* para dispositivos de IoT. A arquitetura suporta a análise de todo ciclo de infecção e propagação de *malwares*, com estrutura de *honeypot* de alta interatividade e autossuficiente. Nesta arquitetura há uma exposição total e monitorada de um sistema de *honeypot* para IoT aos ataques provindos da Internet, mas que ao mesmo tempo encontra-se em um ambiente controlado que cria a representação de parte da Internet quando este *honeypot* passa a ser um ponto de origem de disseminação de ataques.

Para facilitar a compreensão de sua estrutura e o funcionamento de cada componente, a arquitetura foi dividida em quatro módulos principais, conforme pode ser visto na Figura 4.1. As funcionalidades dos elementos pertencentes a cada módulo, bem como a relação entre eles, estão descritas abaixo:

Dispositivos IoT: representam os dispositivos de IoT que serão utilizados como *honeypot*. Este módulo deve suportar a utilização flexível de uma ampla variedade de *honeypots* de baixa ou alta interatividade. Além disso, pode ser usado com sistemas IoT virtualizados em *hardware* específico ou através da emulação de plataformas e sistemas construídos especificamente para capturar dados de acessos maliciosos. Ademais, também pode suportar a utilização de dispositivos IoT reais. Essa característica é inovadora no projeto e traz a vantagem de integrar esforços de *honeypot* de alvo específico que foram construídas, bem como o uso de máquinas

em laboratório. Por exemplo, no departamento de Engenharia Elétrica da USP (MIRANDA, 2016), existem controladores de tensão elétrica da Siemens que são usado em experimentos. Tais equipamentos com alguma interface podem fazer parte da rede experimental e executarem, em ambiente controlado, ataques reais e permitir a monitoração.

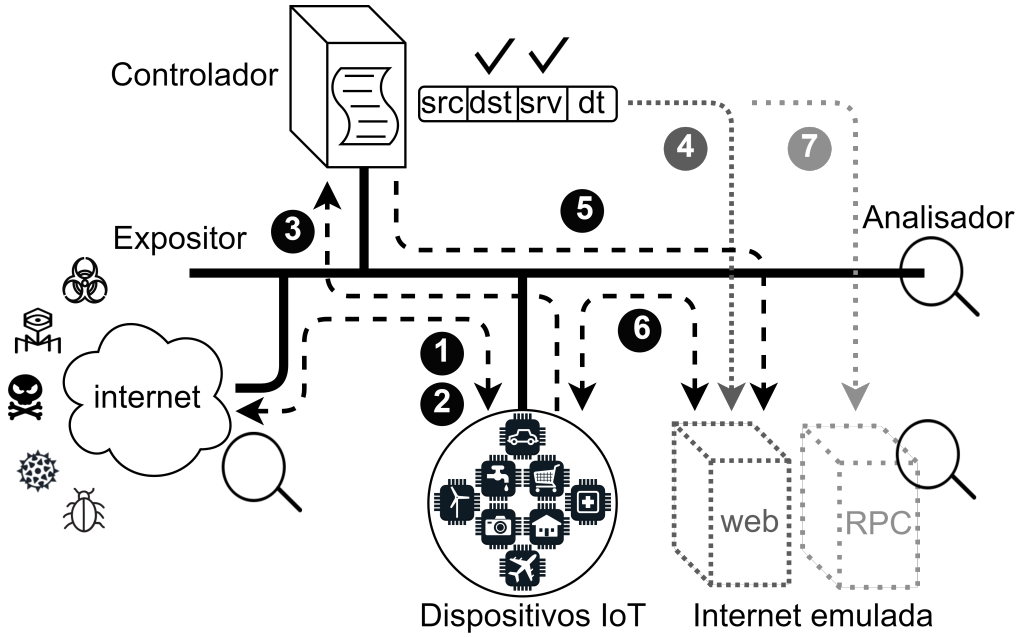


Figura 4.1: Arquitetura proposta de uma *honeynet* de alta interatividade e autossuficiente para dispositivos IoT.

Além disso, um objetivo secundário desse módulo é garantir que o *malware* não seja capaz de identificar o ambiente como sendo um *honeypot*, a fim de possibilitar que a infecção ocorra completamente. Coisas como controladores de dispositivos feitos para VirtualBox precisam ter os nomes modificados. Alguns *malwares* utilizam-se de artifícios variados para identificar se o ambiente é um *honeypot* e, caso a identificação seja positiva, seu funcionamento fica limitado ou até mesmo nulo, atrapalhando a análise de seu comportamento. Portanto, quando for necessário, colocar máquinas reais, sem virtualização para testes, é possível nesse ambiente flexível. Sendo assim, é preciso garantir a execução em diferentes tipos de arquiteturas de computadores (ARM, etc), caso o binário tenha sido feito para tal.

Expositor: este módulo, de interface direta com a rede externa, consiste em toda infraestrutura de rede necessária para exposição completa do *honeypot* na Internet. Sua função é possibilitar que *malwares* da Internet acessem o *honeypot* para infectá-lo. Na Figura 4.1, pode-se observar que o Fluxo 1 permite a comunicação abertamente entre o *honeypot* e possívelmente os *malwares* externos para infecção, bem como a comunicação aberta para com os servidores de C&C a fim de que a infecção ocorra completamente (Fluxo 2). Essas comunicações, no entanto,

serão mediadas pelo controlador, que será descrito em seguida.

As implementações de expositores devem levar em conta uma base de dados de *banners* que possam ser usados para engajar os *malwares* a se conectarem na *honeynet*. Maneiras de fazer esse trabalho são variadas, por exemplo, uma ampla coleta de dados do *site* Shodan, e posterior exposição desses *banners* capturados, pode ser passo para uma ampla atração de artefatos.

Controlador Este módulo é considerado o mais importante da *honeynet*, pois, é ele que garante a inteligência da rede atuando como um gerenciador de tráfego pró-ativo. Esta é umas principais contribuições desse trabalho. Inicialmente, a função desse módulo é controlar o fluxo de dados entre os dispositivos IoT e a Internet, permitindo que todo tráfego originado na Internet com destino ao *honeypot* seja previamente autorizado e encaminhando para o dispositivo IoT, conforme mostra o Fluxo 1. Dessa forma, em específico, temos o interesse que potenciais *malwares* acessem o *honeypot* para iniciar a infecção. Os acessos podem ser feitos através de conexões telnet ou ssh, ou servidor *web* nos IoT.

Esse primeiro contato é garantido sem interferência do controlador. Portanto, o fluxo de comunicação permanece livre e deve permitir também que o *honeypot* envie e receba dados diretamente de servidores de C&C acionados pelo *malware* que estão infectando o dispositivo. Todo esse processo é necessário para que a infecção prossiga completamente, portanto, Fluxo 2 complementar ao primeiro contato.

Toda essa comunicação é mapeada baseada em regras de permissão de tráfego e definição de rotas para encaminhamento dos dados entre os dispositivos envolvidos na comunicação. Conforme apresentaremos posteriormente na seção contendo detalhes da implementação, faremos uso de redes definidas por *software* para esse componente. Embora, ele possa ser implementado de diferentes formas, por exemplo, usando regras de ACL do Linux, ou mesmo programas compilados de filtros usando eBPF (BERTRONE et al., 2018).

A partir da infecção completa do *honeypot* pelo *malware*, o Controlador passará a interceptar todo tráfego originado pelo *honeypot*, conforme mostra o Fluxo 3. A ideia é que sejam definidas regras reativas, que serão gerenciadas por ele. Isso se deve ao fato de que *malware* pode tentar infectar outras máquinas a partir de máquinas já infectadas. Desse modo, a partir desse momento, qualquer tráfego poderá estar relacionado aos procedimentos de propagação e dissiminação do próprio *malware*. E esse tráfego de rede é fundamental material para estudo e análise desses mecanismos e entendimento completo do funcionamento do *malware*. Portanto, é preciso controlar esses tráfegos e redirecionar para uma sandbox inofensiva interna, ao invés de deixar o *malware* livre e permitir que ele tente se propagar e a *honeynet* passe a ser um ponto de ataque às outras redes.

Com o tráfego interceptado pelo Controlador, ele deve ser capaz de identificar os parâmetros de endereçamento da comunicação constantes no fluxo de dados. Informações tais como: endereços de destino, protocolos e portas de comunicação utilizadas. Dessa forma, seria possível com tais informações, solicitar a criação sob demanda de uma estrutura de Internet emulada. Nesse cenário, cada máquina emulada terá o mesmo endereço IP e serviços solicitados pelo *honeypot*, conforme mostra o Fluxo 4.

A partir da criação rápida da máquina que era o destino da comunicação do *honeypot* na Internet emulada, o controlador pode então encaminhar os dados da solicitação de comunicação diretamente a essa máquina na Internet emulada. E portanto, o fluxo 5 dessa arquitetura, onde serão criadas regras nas tabelas de encaminhamento de pacotes, para garantir o desvio de todo o fluxo de dados que venha ocorrer entre o *honeypot* e a máquina solicitada para esta nova estrutura. Isso é demonstrado no Fluxo 6, e com a possibilidade de inspeção dos dados trafegando entre esses 2 elementos, e também, evitando que o *honeypot* seja um ponto de ataque às outras redes existentes e externas. A cada nova solicitação de comunicação iniciada pelo *honeypot* a uma máquina ainda não conhecida, esse processo, sob demanda, se repete por completo, conforme mostra o fluxo final 7, que pode ser replicado múltiplas vezes.

Esse comportamento do controlador de levantar dispositivos emulados sob demanda e no ambiente controlado, visa garantir que o *malware* tenha sempre um ambiente completo para seu funcionamento, permitindo, portanto, que o próprio *honeypot* possa ser replicado como uma outra máquina emulada e possibilitando, no ambiente controlado, a propagação do *malware*. Por conseguinte, permite a análise completa de todo seu ciclo de infecção e propagação, mas em um ambiente totalmente controlado e que não ofereça riscos às outras redes.

Analisador: este módulo é responsável por capturar dados que servirão como base de informações para análise de todo processo de infecção, comprometimento e propagação do *malware*. As ferramentas pertencentes a esse módulo estão distribuídas em três áreas específicas da arquitetura:

- Link de Internet: aqui o analisador é responsável por monitorar todas as tentativas de acesso ao *honeypot*, incluindo tempo, volumetria, origem dos ataques e quais mecanismos são utilizados.
- Dispositivo IoT: o monitoramento interno no dispositivo IoT provê informação que pode ser utilizada na análise do comportamento do *malware* dentro do dispositivo IoT, através de monitoramento de memória, processos utilizados, chamadas de sistemas e outras que se fizerem necessárias.

- Rede interna de comunicação entre *honeypot* e Internet emulada: aqui o analisador é responsável por monitorar todo tráfego de rede existente entre o *honeypot* e todos elementos da Internet emulada, como realizado no *link* de Internet. Nesse caso, as mesmas informações são fornecidas. Também seria possível fazer uma análise dos componentes emulados um a um, para entender o que foi solicitado e quais as reações esperadas para a montagem de um banco de elementos emulados típicos.

Com a utilização de todas informações identificadas pelo analisador, é possível realizar um estudo aprofundado de todo ciclo de vida de infecção de um *malware*.

4.1.1 Considerações Finais sobre a Arquitetura

No Capítulo 2, percorremos algumas das principais motivações para a montagem de *malwares* de IoT, e também analisamos os fundamentos das superficies de ataque. Com base nesses requisitos, elaboramos uma proposta de arquitetura que permite operação constante, flexibilidade de trabalhar com IoTs diferentes e que permite um estudo aprofundado que faz a mistura de elementos de controle da rede da *honeynet* com elementos de sistemas como o *honeypot* e outros elementos emulados. A arquitetura é inovadora no sentido de que ela dá ao especialista de segurança a possibilidade de um estudo aprofundado de todo o ciclo de vida do *malware*. Desde a comunicação com o mundo externo, através da permissão de tráfego do C&C, até a interação com o mundo emulado e permite estudos sobre a propagação do *malware*, entre outras. No próximo capítulo, discutiremos os detalhes de implementação e resultados da prova de conceito dessa arquitetura, de modo a validá-la.

Capítulo 5

HONIOT (HONEYNET IOT)

*Neste capítulo, são apresentados e detalhados o ambiente computacional e as tecnologias utilizadas para o desenvolvimento do protótipo **HonIoT** (Seção 5.1), na Seção 5.2 são apresentados o detalhamento do código do Controlador e a simulação de infecção, bem como os testes realizados para análise e validação da arquitetura apresentada na Seção 4.1, e finalmente há a discussão e análise dos resultados (Seção 5.3).*

5.1 Ambiente e Tecnologias

A fim de validar a arquitetura apresentada no Capítulo 4, foi desenvolvido o protótipo *Honeynet IoT* (HonIoT), cuja finalidade é simular todo processo de infecção sofrido por um *honeypot* de alta interatividade a partir de *malwares* originados da Internet. E, consequentemente, permitir o estudo da tentativa de propagação pela rede do mesmo. Para fins deste trabalho, esse ambiente não foi colocado em produção. Somente experimentos foram realizados para validar os aspectos mais importantes da arquitetura. Em resumo, o HonIoT é um conjunto de *software*, desenvolvido na Linguagem Python, que une um conjunto diversas de aplicações de rede, emulação leve, já existentes e um controlador de redes definidas por *software* desenvolvido especialmente para gerenciar o fluxo de dados na comunicação entre os elementos da arquitetura.

A estrutura física para a prototipação e validação experimental do HonIoT foi composta por dois servidores Dell PowerEdge R210-II, cada um contendo um processador Intel Xeon E3-1270 v3 de 3.50GHz e 8 núcleos, 16GB de memória RAM DDR3 e múltiplas interfaces de rede de 1Gbps cada. Além disso, outro computador de *hardware* de uso geral, com processador AMD Athlon(tm) II X4 640 de 3GHz e 4 núcleos, 2 GB de memória RAM e duas interfaces de rede de 1Gbps cada. Em todas as máquinas, foi instalado o sistema operacional Linux e nenhum serviço adicional, exceto os necessários para o experimento, de modo a ter um ambiente livre

de interferências para medições.

Conforme apresentado na Figura 5.1, que apresenta o ambiente experimental (*testbed*) o *servidor 1* hospeda todo ambiente de controle e emulação responsáveis por reproduzir a porção da Internet atacada. Tal ambiente é composto por controlador SDN, comutador SDN e Internet emulada baseada em contâiners LXD). Por outro lado, *servidor 2* (*honeypot*) possui *softwares* de servidor e cliente *web* responsáveis por receber as requisições a partir de *malwares* existentes na internet e posteriormente, comunicar com as partes emuladas no *servidor 1*. Desse modo, também originam-se solicitações *web* a outros endereços de rede da internet, que serão redirecionados diretamente para o servidor 1, representando as tentativas de propagação do *malware*.

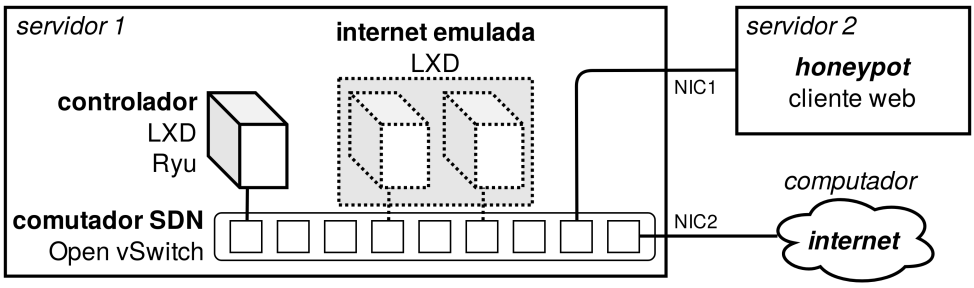


Figura 5.1: Topologia para o protótipo *HonIoT*.

No *servidor 1*, para facilitar a implantação e o gerenciamento de todos os componentes, todo o *software* foi colocando em termos de *containers*. Por exemplo, tanto o controlador, quanto as máquinas pertencentes à internet emulada são instâncias *LinuX Container Daemon* (LXD), versão 3.0.3. O comutador da rede definida por *software* (*Software Defined Network* - SDN) é o Open vSwitch versão 2.9.0; a ferramenta para análise de pacotes na rede, tanto na conexão com a Internet quanto no comutador SDN, é o Wireshark versão 2.6.5; e o *software* controlador da SDN é o Ryu versão 4.24.

5.2 Controlador e Simulação de Infecção

Com base no fluxo de mensagens definido na Arquitetura do Capítulo 4, o funcionamento do protótipo HonIoT é iniciado quando uma requisição *web* originada na Internet é destinada ao *honeypot*, equivalente ao Fluxo 1 da Figura 4.1. O comutador no protótipo está conectado às portas do servidor 1, e, portanto, ele controla o fluxo de mensagens entre Servidor 1, Servidor 2 e Internet.

Quando essa requisição chega ao comutador SDN, este a encaminha diretamente ao controlador para que seja definida qual ação deve ser tomada em relação ao tráfego recebido. Todo

tráfego originado na Internet com destino ao *honeypot* deve ser, a princípio, permitido para garantir que qualquer *malware* possa acessar livremente o *honeypot* para iniciar a infecção.

Assim que recebe o pacote de dados, o controlador inspeciona os campos correspondentes ao protocolo e flags utilizadas do cabeçalho de camada 4 do modelo OSI/ISO, e endereços IP de origem e destino do pacote para identificar se o fluxo de dados tem origem na Internet (*src_port*) e está direcionado ao *honeypot* (*HONEYPOT_IP*). Caso a comunicação siga essa condição, o controlador então monta a regra que permite a comunicação entre a máquina da Internet e o *honeypot* (*match*1) e a instala no comutador SDN (*add_flow*). Figura 5.2. Nesse processo, a condição $bits == 2$ verifica se este pacote corresponde a uma iniciação de comunicação TCP, onde o valor 2 indica que o segmento TCP possui a flag *SYN* em uso.

```
###
# comunicação internet <-> honeypot
if dst_ip == self.HONEYPOT_IP and \
((protocol == 'TCP' and bits == 2) or protocol != 'TCP'):
    ...
    match1 = msg.datapath.ofproto_parser.OFPMatch(eth_type=2048, \
        ip_proto=protocolN, udp_src=src_port, udp_dst=dst_port, \
        ipv4_src=src_ip, ipv4_dst=dst_ip)
    ...
    action1 = [msg.datapath.ofproto_parser.OFPActionOutput(out_port1)]
    ...
    self.add_flow(msg.datapath, 1, match1, action1)
```

Figura 5.2: Exemplo de código do controlador que permite que máquinas da Internet acessem livremente o *honeypot*.

Seguindo esse processo, quando o *honeypot* inicia a comunicação com um servidor C&C, o tráfego deve ser permitido pelo controlador, conforme descrito no Capítulo 2, pois isso é fundamental para que o *malware* continue sua atividade maliciosa, conforme mostrado na Figura 5.3. Durante esse processo, o controlador verifica se o IP de destino da comunicação *dst_ip* existe na *black_list* recuperada em Emerging Threats (2017). Essa filtragem por endereços da *black_list* é que permite que o controlador libere comunicações com a Internet com os *softwares* de controle de *malwares*, mas que também impedirá o acesso externo de um *malware* capturado.

```
###
# comunicação honeypot <-> C&C
def is_attack(self, dst_ip):
    if self.command_ip_list.in_list(dst_ip):
        return False
    else:
        return True
```

Figura 5.3: Exemplo de código do controlador que permite a comunicação do *honeypot* com servidores de C&C.

Quando o *honeypot* origina e, em sequência, envia um novo datagrama IP à qualquer outra máquina, seja da rede local ou da Internet, o comutador SDN receberá esse pacote. E tomará a decisão de encaminhar diretamente ao controlador.

Isso é necessário para que o controlador defina o procedimento a ser realizado nesse caso. Por exemplo, ao receber um datagrama IP encaminhado pelo comutador SDN, o controlador lê os campos correspondentes ao tipo do protocolo e porta de destino da camada 4 do modelo OSI/ISO (por exemplo, porta telnet), bem como o endereço IP de destino contido no cabeçalho da camada 3 do mesmo modelo de protocolos.

Com essas informações, o controlador acessa o gerenciador de *containers* do *servidor 1* através de uma conexão SSH exclusiva e utilizando o *software* Python Paramiko, versão 2.4.1, solicita a criação de um novo *container* LXD com aquele serviço correspondente a partir de modelos pré-estabelecidos (*DEFAULT_CONTAINER_TEMPLATE*). E então, o gerenciador de *containers* completa a configuração de endereço IP correspondente aos parâmetros obtidos no datagrama IP recebido pelo controlador e o embute na "máquina" emulada.

Outro parâmetro importante para acelerar o processo é definir e configurar um endereço de camada 2 (endereço MAC) específico no *container* criado pela variável (*new_container_mac*). Ao final desse processo, o controlador instala uma regra no comutador SDN permitindo a comunicação direta e exclusiva entre o *container* criado e o *honeypot*, bem como informar ao *honeypot* o endereço MAC atribuído ao *container*, possibilitando assim o encaminhamento de tráfego diretamente entre as máquinas, conforme mostra o trecho de código da Figura 5.4. Esse processo é repetido todas vezes em que o *honeypot* dispara datagramas com novas combinações de endereço IP de destino, protocolo e porta de comunicação. Desse modo, uma quantidade virtualmente infinita de elementos emulados pode ser criado sob demanda para testar cobertura e *stress* dos *malwares*.

```
###
# honeypot atacando outras máquinas
    def create_container(self, msg, src_ethernet, ip_version, src_ip, \
                         dst_ip, container_name):
        template = self.DEFAULT_CONTAINER_TEMPLATE
        ...
        ssh = SSH()
        cmd = 'lxc launch %s %s' % (template, container_name)
        ssh.exec_cmd(cmd)
        ...
        cmd = 'lxc exec %s -- cat /sys/class/net/eth0/address' \
            % (container_name)
        new_container_mac = ssh.exec_cmd(cmd).strip()
```

Figura 5.4: Exemplo de código do controlador que cria *containers* na Internet emulada.

Outro ponto importante de todo esse processo é o mapeamento e comutação a partir do endereço MAC de destino para a porta específica do comutador SDN realizada pelo controlador quando o *honeypot* troca de dados com a máquina da Internet emulada. Na Figura 5.5, é possível observar que na *action*1 é feita a troca desse endereço com o valor do parâmetro *new_container_mac*. Dessa forma, a comunicação bilateral transcorre normalmente.

```
action1 = [msg.datapath.ofproto_parser.\
          OFPActionSetField(eth_dst=new_container_mac), \
          msg.datapath.ofproto_parser.OFPActionOutput(out_port1)]
action2 = [msg.datapath.ofproto_parser.OFPActionOutput(out_port2)]
```

Figura 5.5: Exemplo de código do controlador que realiza o encaminhamento a Internet emulada.

Todos esses procedimentos programados no protótipo foram necessários para garantir que toda estrutura do protótipo HonIoT atenda perfeitamente os requisitos da arquitetura apresentada no Capítulo 4.1 e o fluxo de informações conforme descrito.

5.3 Testes de Validação e Discussão dos Resultados

Em seguida à descrição da implementação, foram realizados exaustivos testes no ambiente experiemental dos servidores descritos acima. Os testes foram realizados para validação do protótipo e foram divididos em 4 categorias de validação, chamadas aqui de “etapas”, dependendo da parte do protótipo sendo exercida. O experimento completo contou com 10 execuções de cada etapa, possibilitando assim, perceber e tratar possíveis desvios nos resultados causados por instabilidade do protótipo ou interferência de fatores externos não previstos.

5.3.1 Experimentos de Invasão ao honeypot

Este experimento analisa o correto funcionamento do processo em que máquinas distintas da Internet acessam serviços específicos do *honeypot*. Esse acesso deve ocorrer sem restrição alguma por parte do comutador SDN. Do ponto de vista de resultados, na etapa 1, todas solicitações de acesso ao *honeypot* originadas na Internet foram prontamente encaminhadas pelo comutador SDN e recebidas no *honeypot*, conforme demonstraram os rastros de execução e esta validação foi considerada satisfatória. Os experimentos foram executados a partir da máquina 3, um computador comum atuando em portas de conexão telnet.

5.3.2 Experimentos de Comunicação Honeypot e C&C

Em seguida, um segundo experimento foi o de acompanhar o correto funcionamento da comunicação do *honeypot* com servidores de C&C a qualquer momento. Esse acesso deve ser permitido quando o endereço IP de destino do datagrama emitido pelo honeypot consta na *black list* de servidores C&C. Ao invés de testar com um *malware* imprevisível, foram feitos testes de conexão diretamente de um *container*, e todo o tráfego originado no *honeynet* com destino aos endereços IP dos servidores constantes na *black list* de C&C foram encaminhadas para a saída de internet corretamente. Os resultados foram executados a partir do Servidor 1 para a internet e os rastros de execução do controlador demonstram que esse teste de validação foi considerado satisfatório.

5.3.3 Experimentos de Criação da Internet emulada

Este terceiro experimento tem o propósito de verificar se o controlador solicita corretamente a criação de um ou mais *containers* ao *Servidor 1* após receber datagramas IP com dados "não constantes" em regras já instaladas no comutador SDN e se os *containers* são criados e inicializados corretamente para encaminhamento do tráfego partindo do *honeypot* no tempo necessário.

Portanto, para esta etapa, o interesse era em dois cenários possíveis. O primeiro tentando verificar o comportamento do ambiente, na forma dessa implementação, quando solicitados chamadas de grupos de *containers* de forma sequencial, onde um *container* posterior somente é solicitado quando o anterior já está totalmente funcional. A ideia é estudar o comportamento de um *malware* que interage e analisa o ambiente em que está antes de tomar a próxima decisão.

O segundo cenário analisa o comportamento do ambiente quando grupos de *containers* são solicitados em rajadas pelo *honeypot*. Por exemplo, quando na ocorrência de *scan* de rede. O *scan* de rede pode ser intenso e pode requerer a criação de dezenas ou centenas de *containers* para cada pacote de *scan* gerado. Esse cenário permite verificar e analisar a escalabilidade e estabilidade do protótipo. Eventualmente, no futuro, esse tipo de ocorrência poderia ter um comportamente diferente do controlador, reutilizando um mesmo *container* com múltiplos IPs para dar vazão ao procedimento de *scan*, fazendo uma escolha aleatória, se responde ou não. Pois, nem todos os IPs têm máquinas associadas a eles na Internet.

Os resultados demonstrados na Figura 5.6 apresentam a utilização de memória (em MB) agrupada por conjunto de *containers* criados sequencialmente, ou seja, no cenário 1. Os dados mostram que a utilização de memória está diretamente relacionada ao número de *containers*

criados, porém, seu crescimento não é linearmente perfeito, partindo de aproximados 70 MB para criação de 1 *container*, passando por aproximados 463 MB de memória consumida para um grupo de 50 *containers*, chegando a até 925 MB de memória consumida para grupos de 100 *containers*, sempre com 100% de sucesso na criação dos *containers*. Os resultados indicam uma sobrecarga de memória ao serem criados mais *containers*. Não foi investigado a fundo o porquê desse comportamento, mas pode ser em função de replicação de dados do sistema operacional. Outra possibilidade é que o tempo de percorrimento da lista negra pode ser alto por pacote.

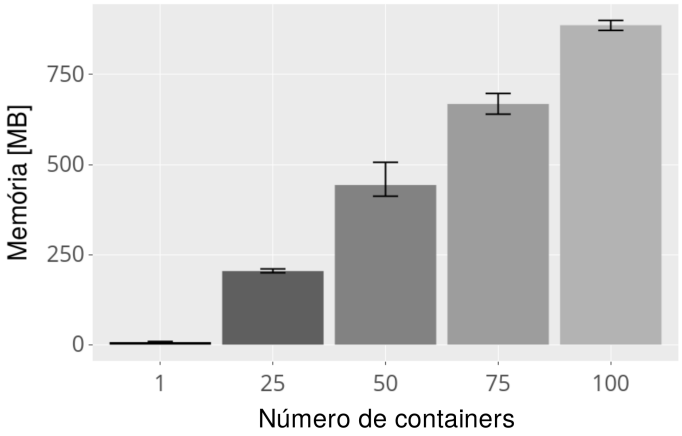


Figura 5.6: Memória (em MB) consumida para criação dos *containers* correspondentes da *Internet emulada*.

Apesar desse comportamento de sobrecarga adicional por *container* adicional, o protótipo comprova que a utilização de *containers* LXD na Internet emulada é promissora quando considerado o consumo total de *hardware* versus o número de máquinas emuladas. Ou seja, 100 *containers* em uma máquina servidora pode ser considerado baixo uso de recursos.

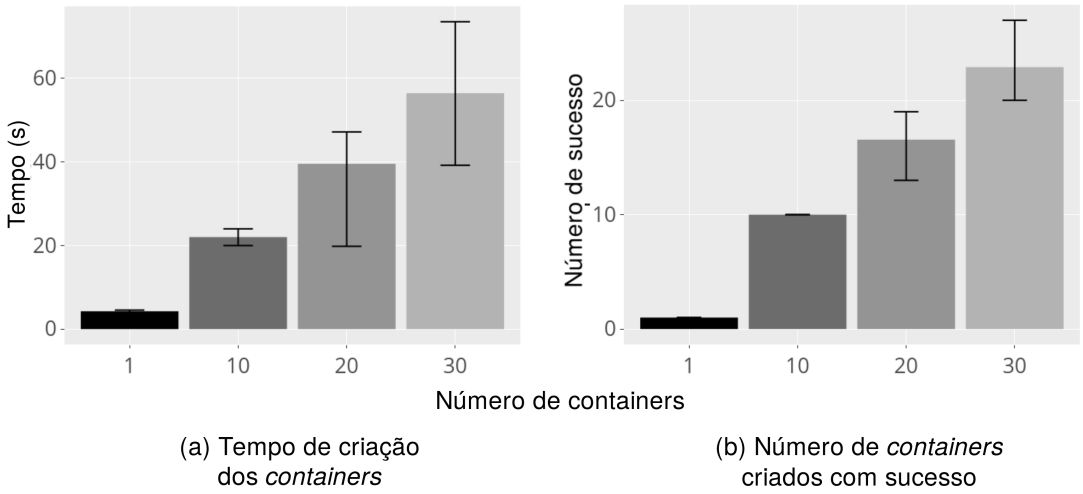
Outros testes nesse cenário também identificaram um comportamento padrão quanto ao tempo de criação dos *containers*, que ficou em 4.1 segundos por *container*. Esse dado tem o revés de ser elevado para tratar o encaminhamento de um pacote e pode gerar uma desconfiança em um *malware* mais sofisticado. No futuro, talvez o teste com *containers* pré-carregados pode acelerar esse procedimento.

Com relação ao cenário 2, onde é analisado o comportamento do protótipo frente a situações de rajadas de pacotes, foram verificados o tempo de criação para os grupos de *containers* e quantidade de sucesso nesse procedimento. Como não existe tempo mínimo entre as chegadas de pacotes, o controlador recebe por meio de eventos em *threads* múltiplas requisições simultâneas e repassa para o gerenciador de *containers*.

Os dados da Figura 5.7 (a) apresentam que, apesar das solicitações terem sido geradas

em rajadas a partir do *honeypot*, onde verificou-se um tempo médio de apenas 0,6 segundos para o encaminhamento de todas solicitações, houve um aumento considerável do tempo para criação dos *containers* da Internet emulada no *servidor 1*, chegando a aproximadamente 53 segundos em média para 30 *containers solicitados*. Pois, os recursos de criação de *containers* são compartilhados e a contenção é grande.

Além dos tempos de instanciação serem muito elevados, outro dado importante é a grande variação do tempo entre os testes, o que demonstrou instabilidade do ambiente do protótipo nesse caso.



(a) Tempo de criação dos *containers*

(b) Número de *containers* criados com sucesso

Figura 5.7: Tempo (segundos) consumidos para criação de grupos de *containers* solicitados em rajadas (a) e quantidade de *containers* criados com sucesso por grupos de *containers* solicitados em rajadas (b).

Para analisar essa questão de instabilidade e a incapacidade de dar vazão rápida para um número muito grande de instanciação de *containers*, uma observação importante é a redução expressiva da taxa de sucesso na criação dos *containers* com o aumento da quantidade nas rajadas. Conforme pode ser visto na Figura 5.7(b), a partir de 20 solicitações de criação de *containers*, a taxa de insucesso aumenta para aproximadamente 20% de falhas para cada grupo de 30 *containers* solicitados.

Análises preliminares desses resultados apontam que as tecnologias utilizadas no protótipo possuem deficiência na forma como o controlador conecta no *servidor 1* para solicitar a criação dos *containers*, utilizando conexões SSH através do Python Paramiko. Os rastros da execução apontaram erros no estabelecimento de conexões entre essas duas máquinas quando as solicitações chegam em rajadas. Uma possível alternativa é a utilização de sockets Unix ou outro tipo de comunicação de rede menos complexa que uma conexão SSH. E também como foi explicado acima, uma outra opção é a pré-instanciação de centenas de *containers* no início da criação do

ambiente experimental, cabendo ao controlador somente a tarefa de configurá-los.

5.3.4 Comunicação com Internet emulada

O último experimento de validação verifica se está ocorrendo corretamente o encaminhamento do tráfego do *honeypot* com os *containers* da Internet emulada, ao invés de direcioná-lo para saída pelo *link* de internet real. Nessa etapa, basicamente após a criação dos *containers* de Internet emulada, é feita a criação das regras de encaminhamento no comutador SDN, atualizando na tabela de fluxos, todo tráfego partindo do *honeynet* foi encaminhando corretamente, sem atrasos relevantes ou quebra de conexão TCP para as partes correspondentes. Os rastros de execução do controlador e dos *containers* envolvidos demonstram que este teste de validação foi considerado satisfatório.

Capítulo 6

CONCLUSÃO E TRABALHOS FUTUROS

Este capítulo demonstra os resutados obtidos através do desenvolvimeto deste trabalho, a Seção 6.1 apresenta o término deste trabalho concluindo-o, enquanto a Seção 6.2 apresenta caminhos possíveis para trabalhos futuros.

6.1 CONCLUSÃO

Neste trabalho, apresentamos uma prova de conceito de uma inovadora arquitetura de análise dinâmica de alta interatividade de *malware* baseada no modelo *Honeynet* e Sandbox específico para IoT. As contribuições com o cenário acadêmico são relevantes e para pesquisas futuras, pois os dados gerados neste trabalho poderão ser utilizados em outros trabalhos, com o crescimento da Internet das Coisas (IoT) e o crescente cenário de segurança ajudará no fortalecimento de ferramentas que possam estudar as ações realizadas por autores Maliciosos e pelos *Malwares*.

Entre as facilidades encontradas em nosso trabalho podemos descrever que existe uma abundância de trabalhos relacionados nas áreas de *Honeypot*, *Honeynet* e Sandbox, entretanto, o foco destes correlatos está na atuação de *malwares* para *desktops* e *laptops*. Portanto, foi necessário buscar na literatura por propostas mais específicas para IoTs, que têm ainda quantidades limitadas no âmbito acadêmico. E algumas implementações estão disponíveis, mas sem arcabouço teórico e experimental apropriado.

Com base nos requisitos de um ambiente de alta interatividade para *malwares* de IoT, concebemos a arquitetura do projeto e realizamos a implementação de partes relevantes da arquitetura para validá-la. Nossos resultados mostram que a implementação da arquitetura foi satisfatória montando um ambiente controlado, de modo que não seja possível espalhar os ataques em am-

biente de alta interatividade. E para isso, fazemos uso dos módulos que permitem criar em tempo factível, a representação da Internet Emulada para a interação com os *malwares*.

Além disso, uma das inovações importantes da arquitetura é prever de maneira modularizada que seja possível adicionar qualquer implementação de *honeypot* disponível e específica para IoT sem modificar a arquitetura, portanto, tornando-a autossuficiente e possibilitando a realização de estudos avançados de *malwares* para IoT.

6.2 Trabalhos Futuros

Como trabalhos futuros, pretendemos investigar maneiras mais sofisticadas de distinguir a comunicação entre o *malware* e o seu C&C. A solução adotada é relativamente simples e usa uma lista exaustiva de endereços IP para verificar se o pacote é para o C&C. Essa lista exaustiva pode atrasar ainda mais o tempo de processamento dos pacotes. Lembrando que na implementação fazemos uso do banco de dados de Command Controller, obtidos através de OSINT Open Source Intelligence (Inteligência de Fontes Abertas). Ao incorporar IA, uma possibilidade é fazer uma análise do binário do *malware* e fazer a marcação das chamadas de sistema (*syscalls*), com base em aprendizado de máquina, que são consideradas maliciosas de outras que são consideradas comunicação com o C&C. Assim, quando estas *syscalls* estiverem na eminência de serem executadas, o próprio *honeypot* pode enviar uma mensagem antes para o controlador, de que o próximo pacote é para o C&C. Finalmente, uma última consideração é a criação de *pools* de *containers* para diminuir a sobrecarga de criação sobdemanda e reduzir os atrasos de resposta entre *malware* e *container* com serviço solicitado pelo mesmo.

REFERÊNCIAS

BERTRONE, M. et al. Accelerating linux security with ebpf iptables. In: *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos*. New York, NY, USA: ACM, 2018. (SIGCOMM '18), p. 108–110. ISBN 978-1-4503-5915-3. Disponível em: <http://doi.acm.org/10.1145/3234200.3234228>.

BHUNIA, S. S.; GURUSAMY, M. Dynamic attack detection and mitigation in iot using sdn. In: *2017 27th International Telecommunication Networks and Applications Conference (ITNAC)*. [S.l.: s.n.], 2017. p. 1–6.

CERT.BR. *Distributed Honeypots Project*. 2017. `https://honeytarg.cert.br/honeypots/`. Acessado em: 21/04/2017.

CERT.BR. *Honeypots e Honeynets: Definições e Aplicações*. 2017. `https://www.cert.br/docs/whitepapers/honeypots-honeynets/`. Acessado em: 20/04/2017.

DOWLING, S.; SCHUKAT, M.; MELVIN, H. Data-centric framework for adaptive smart city honeynets. In: *2017 Smart City Symposium Prague (SCSP)*. [S.l.: s.n.], 2017. p. 1–7.

DUFFY, C. *Aprendendo Pentest com Python 1a Edição*. [S.l.]: Novatec, 2016.

EGELE, M. et al. A survey on automated dynamic malware-analysis techniques and tools. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 44, n. 2, p. 6:1–6:42, mar. 2008. ISSN 0360-0300. Disponível em: <http://doi.acm.org/10.1145/2089125.2089126>.

Emerging Threats. *Emerging Threats Botnet Command and Control drop rules*. 2017. `http://rules.emergingthreats.net/open/suricata/rules/botcc.rules`. Acessado em: 18/12/2017.

FILHO, D. S. F. et al. Técnicas para análise dinâmica de malware. *SBSeg*, v. 11, p. 104–144, 2011.

GARTNER. *6.4 Billion Connected "Things"Will Be in Use in 2016*. 2017. `https://www.gartner.com/newsroom/id/3165317`. Acessado em: 13/04/2017.

JR., A. M. e David Alexander e Elverton Fazzion e Osvaldo Fonseca e Italo Cunha e Cristine Hoepers e Klaus Steding-Jessen e Marcelo H. P. C. Chaves e Dorgival Guedes e W. M. Monitoramento e caracterização de botnets bashlite em dispositivos iot. *Simpósio Brasileiro de Redes de Computadores (SBRC)*, v. 36, 2018. Disponível em: <http://portaldeconteudo.sbc.org.br/index.php/sbrc/article/view/2479>.

KOLIAS, C. et al. Ddos in the iot: Mirai and other botnets. *Computer*, v. 50, n. 7, p. 80–84, 2017. ISSN 0018-9162.

LANTZ, B.; HELLER, B.; MCKEOWN, N. A network in a laptop: Rapid prototyping for software-defined networks. In: *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. New York, NY, USA: ACM, 2010. (Hotnets-IX), p. 19:1–19:6. ISBN 978-1-4503-0409-2. Disponível em: <http://doi-acm-org.ez31.periodicos.capes.gov.br/10.1145/1868447.1868466>.

LYU, M. et al. Quantifying the reflective ddos attack capability of household iot devices. In: *Proceedings of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks*. New York, NY, USA: ACM, 2017. (WiSec '17), p. 46–51. ISBN 978-1-4503-5084-6. Disponível em: <http://doi.acm.org/10.1145/3098243.3098264>.

MCKEOWN, N. et al. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, ACM, New York, NY, USA, v. 38, n. 2, p. 69–74, mar. 2008. ISSN 0146-4833. Disponível em: <http://doi.acm.org/10.1145/1355734.1355746>.

MELO, L. P. de et al. Análise de malware: Investigação de códigos ma-liciosos através de uma abordagem prática. *SBSeg*, v. 11, p. 9–52, 2011.

MIRANDA, J. C. *Segurança cibernética com hardware reconfigurável em subestações de energia elétrica utilizando o padrão IEC 61850*. Tese (Doutorado) — Escola de Engenharia de São Carlos, Universidade de São Paulo, 7 2016. Doi:10.11606/T.18.2016.tde-11112016-090545.

PA, Y. M. P. et al. Iotpot: Analysing the rise of iot compromises. In: *9th USENIX Workshop on Offensive Technologies (WOOT 15)*. Washington, D.C.: USENIX Association, 2015. Disponível em: <https://www.usenix.org/conference/woot15/workshop-program/presentation/pa>.

POPERESHNYAK, S. et al. Intrusion detection method based on the sensory traps system. In: *2018 XIV-th International Conference on Perspective Technologies and Methods in MEMS Design (MEMSTECH)*. [S.l.: s.n.], 2018. p. 122–126. ISSN 2573-5373.

PROVOS, N. A virtual honeypot framework. In: *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*. Berkeley, CA, USA: USENIX Association, 2004. (SSYM'04), p. 1–1. Disponível em: <http://dl.acm.org/citation.cfm?id=1251375.1251376>.

SANTOS, B. P. et al. Internet das coisas: da teoriaa prática. *Minicursos SBRC-Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*, 2016.

SCHMIDT, J. L. da S.; GUARDIA, H. C. Utilizando a análise dinâmica para entender códigos maliciosos: Um estudo de caso. *Revista TIS*, v. 3, n. 1, 2014.

SEZER, S. et al. Are we ready for sdn? implementation challenges for software-defined networks. *IEEE Communications Magazine*, IEEE, v. 51, n. 7, p. 36–43, 2013.

SHIN, M.-K.; NAM, K.-H.; KIM, H.-J. Software-defined networking (sdn): A reference architecture and open apis. In: IEEE. *2012 International Conference on ICT Convergence (ICTC)*. [S.l.], 2012. p. 360–361.

SIMPSON, A. K.; ROESNER, F.; KOHNO, T. Securing vulnerable home iot devices with an in-hub security manager. In: IEEE. *Pervasive Computing and Communications Workshops (PerCom Workshops), 2017 IEEE International Conference on*. [S.l.], 2017. p. 551–556.

SONG, C.; HAY, B.; ZHUGE, J. *Qebex*. 2018. `http://honeynet.org/papers/KYT_qebek`. Acesso em: 23/07/2018.

SPAMHAUS. *Spamhaus Botnet Threat Report 2017*. 2018. `https://www.spamhaus.org/news/article/772/spamhaus-botnet-threat-report-2017`. Acessado em: 05/03/2018.

STALLINGS, W. *Arquitetura e Organização de Computadores 8a Edição*. [S.l.]: Pearson Prentice Hall, 2010.

STEVENS, W. R.; WRIGHT, G. Tcp/ip illustrated, vol. 1. addison-wesley. *Reading, MA*, 1994.

SYMANTEC. *Internet Security Threat Report, Volume 23*. 2018. `https://www.symantec.com/security-center/threat-report`. Acessado em: 20/03/2018.

GLOSSÁRIO

APIs – *Application Programming Interface*

CPU – *Central Process Unit*

C&C – *Command & Control*

DDoS – *Distributed Denial of Service*

DNS – *Domain Name System*

DVR – *Digital Video Record*

DoS – *Denial of Service*

ICS – *Sistema de Controle Industrial*

IDS – *Instrusion Detection Systems*

IP – *Internet Protocol*

IoT – *Internet of Things*

LXD – *Linux Contrainer Daemon*

OSINT – *Open Source Inteligence*

SDN – *Software Defined Network*

TCP – *Transmission Control Protocol*

TI – *Tecnologia da Informação*

Apendice A

IMPLEMENTAÇÃO DO CONTROLADOR

Neste apêndice demonstramos o código implementado no controlador, este é responsável por criar regras e pela criação dos containers, os quais vão ser utilizados como internet Emulada.

```
# Honeynet Switch
# Container dinamico
# criacao de regras no switch
# Version: 1.0

from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER, MAIN_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet
from ryu.lib.packet import ipv4
from ryu.lib.packet import ipv6
from ryu.lib.packet import tcp
from ryu.lib.packet import udp
from ryu.lib.packet import icmp
from ryu.lib.packet import icmpv6
from paramiko import SSHClient
import threading
import paramiko
import time

class HoneynetSwitch(app_manager.RyuApp):
```

```
25      OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
26
27      def __init__(self, *args, **kwargs):
28          super(HoneynetSwitch, self).__init__(*args, **kwargs)
29
30          # mac do destino para porta do switch
31          self.mac_to_port = {}
32
33          # informa o IP do honeypot
34          self.HONEYPOT_IP = '10.0.0.2'
35
36          # informa o MAC do Gateway da internet
37          self.GATEWAY_MAC = ''
38
39          # template default (generico) para container
40          self.DEFAULT_CONTAINER_TEMPLATE = 'servidor'
41
42          # semaforo para exitar problemas de concorrencia na atualizacao das
        regras do switch
43          self.switch_flows_semaphore = threading.Semaphore()
44
45          # cria uma conexao ssh com o host
46          #self.ssh = SSH()
47
48          # cria um mapa para porta -> template de container
49          self.map_port_to_container_template = Map()
50
51          # cria um mapa para IP -> container (nome, mac)
52          self.map_ip_to_container = Map()
53
54          # cria uma lista de IPs de comando e controle de Botnets
55          self.command_ip_list = IPList()
56
57          # cria uma whitelist para liberar conexoes iniciadas por hosts
        externos
58          self.whitelist = WhiteList()
59
60          # inicializa o mapeamento
61          #self.init_map_container_template()
62
63      # inicializa o mapeamento dos portas para os container
64      # mapeia porta:(nome do container, MAC do container)
65      def init_map_container_template(self):
```

```
66          # mapeia a porta da aplicacao para um template de container
    especifico
67          # parametros: porta, nome do template
68
69          '''
70          # NAO UTILIZADO NO MOMENTO
71          self.map_port_to_container_template.add(80,'servidor')
72          self.map_port_to_container_template.add(23,'servidor')
73          '''
74
75          # imprime a mensagem de conclusao
76          print('Mapping (containers template) successful!')
77
78      @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
79      def switch_features_handler(self, ev):
80          datapath = ev.msg.datapath
81          ofproto = datapath.ofproto
82          parser = datapath.ofproto_parser
83
84          # install table-miss flow entry
85          #
86          # We specify NO BUFFER to max_len of the output action due to
87          # OVS bug. At this moment, if we specify a lesser number, e.g.,
88          # 128, OVS will send Packet-In with invalid buffer_id and
89          # truncated packet data. In that case, we cannot output packets
90          # correctly.  The bug has been fixed in OVS v2.1.0.
91          match = parser.OFPMatch()
92          actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER, ofproto.
    OFPCML_NO_BUFFER)]
93          self.add_flow(datapath, 0, match, actions)
94
95      def add_flow(self, datapath, priority, match, actions, buffer_id=None):
96          ofproto = datapath.ofproto
97          parser = datapath.ofproto_parser
98
99          inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
100                                               actions)]
101         if buffer_id:
102             mod = parser.OFPFlowMod(datapath=datapath, buffer_id=buffer_id,
103                                     priority=priority, match=match,
104                                     instructions=inst)
105         else:
106             mod = parser.OFPFlowMod(datapath=datapath, priority=priority,
```

```
107                                     match=match, instructions=inst)
108         datapath.send_msg(mod)
109
110     @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
111     def _packet_in_handler(self, ev):
112         # If you hit this you might want to increase
113         # the "miss_send_length" of your switch
114         if ev.msg.msg_len < ev.msg.total_len:
115             self.logger.debug("packet truncated: only %s of %s bytes",
116                               ev.msg.msg_len, ev.msg.total_len)
117         msg = ev.msg
118         in_port = msg.match['in_port']
119         pkt = packet.Packet(msg.data)
120
121         #informacoes do protocolo ethernet - L2
122         pkt_ethernet = pkt.get_protocol(ethernet.ethernet)
123         src_ethernet = pkt_ethernet.src
124         dst_ethernet = pkt_ethernet.dst
125
126         dpid = msg.datapath.id
127         self.mac_to_port.setdefault(dpid, {})
128
129         # learn a mac address to avoid FLOOD next time.
130         self.mac_to_port[dpid][src_ethernet] = in_port
131
132         #informacoes do ip
133         if pkt.get_protocol(ipv4.ipv4):
134             pkt_ip = pkt.get_protocol(ipv4.ipv4)
135             ip_version = 'IPV4'
136         else:
137             pkt_ip = pkt.get_protocol(ipv6.ipv6)
138             ip_version = 'IPV6'
139
140         #enderecos IP
141         if pkt_ip:
142             src_ip = pkt_ip.src
143             dst_ip = pkt_ip.dst
144
145         #informacoes do tcp
146         pkt_tcp = pkt.get_protocol(tcp.tcp)
147
148         #se for tcp
149         if pkt_tcp:
```

```
150             #portas
151             src_port = pkt_tcp.src_port
152             dst_port = pkt_tcp.dst_port
153 
154             #se bits for igual a 2, entao eh um pacote SYN
155           #bits = pkt_tcp.bits
156 
157             #payload tcp
158             payload = pkt.protocols[-1] if pkt.protocols[-1] != pkt_tcp
    else ''
159 
160             #trata o pacote tcp
161             self.handle_packet(msg, 'TCP', src_ethernet, dst_ethernet,
    ip_version, src_ip, dst_ip, src_port, dst_port, payload)
162 
163         else:
164             #informacoes do udp
165             pkt_udp = pkt.get_protocol(udp.udp)
166 
167             if pkt_udp:
168                 #portas
169                 src_port = pkt_udp.src_port
170                 dst_port = pkt_udp.dst_port
171 
172                 #payload udp
173                 payload = pkt.protocols[-1] if pkt.protocols[-1] != pkt_udp
     else ''
174 
175                 #trata o pacote udp
176                 self.handle_packet(msg, 'UDP', src_ethernet, dst_ethernet,
    ip_version, src_ip, dst_ip, src_port, dst_port, payload)
177 
178             else:
179                 self.send_packet_out(msg)
180 
181     '''
182         encaminha o pacote para o destino
183     '''
184     def send_packet_out(self, msg):
185 
186         datapath = msg.datapath
187         ofproto = datapath.ofproto
188         parser = datapath.ofproto_parser
```

```
189         in_port = msg.match['in_port']
190         pkt = packet.Packet(msg.data)
191 
192         #informacoes do protocolo ethernet - L2
193         pkt_ethernet = pkt.get_protocol(ethernet.ethernet)
194         dst = pkt_ethernet.dst
195         #src = pkt_ethernet.src
196 
197         '''
198         dpid = datapath.id
199         self.mac_to_port.setdefault(dpid, {})
200 
201         # learn a mac address to avoid FLOOD next time.
202         self.mac_to_port[dpid][src] = in_port
203 
204         if dst in self.mac_to_port[dpid]:
205             out_port = self.mac_to_port[dpid][dst]
206         else:
207             out_port = ofproto.OFPP_FLOOD
208         '''
209 
210         # obtem a porta de saida
211         out_port = self.get_out_port(msg, dst)
212 
213         actions = [parser.OFPActionOutput(out_port)]
214 
215         data = None
216         if msg.buffer_id == ofproto.OFP_NO_BUFFER:
217             data = msg.data
218 
219         out = parser.OFPPacketOut(datapath=datapath, buffer_id=msg.
    buffer_id,
220                                   in_port=in_port, actions=actions, data=data)
221 
222         datapath.send_msg(out)
223 
224     '''
225         obtem a porta de saida para a mensagem no switch
226     '''
227     def get_out_port(self, msg, eth_dst):
228 
229         datapath = msg.datapath
230         ofproto = datapath.ofproto
```

```
231         dpid = datapath.id
232 
233         if eth_dst in self.mac_to_port[dpid]:
234             out_port = self.mac_to_port[dpid][eth_dst]
235         else:
236             out_port = ofproto.OFPP_FLOOD
237 
238         return out_port
239 
240     '''
241         msg: eh a estrutura completa da mensagem recebida pelo controlador
242         protocol: eh o protocolo da mensagem (TCP, UDP ou ICMP)
243         src_ethernet: eh o MAC de origem
244         dst_ethernet: eh o MAC de destino
245         ip_version: eh a versao do IP (IPV4 ou IPV6)
246         src_ip: eh o IP de origem
247         dst_ip: eh o IP de destino
248         src_port: eh a porta de origem (apenas para TCP e UDP)
249         dst_port: eh a porta de destino: (apenas para TCP e UDP)
250         payload: eh a carga util do pacote
251     '''
252     def handle_packet(self, msg, protocol, src_ethernet, dst_ethernet,
    ip_version, src_ip, dst_ip, src_port, dst_port, payload):
253 
254         pkt = packet.Packet(msg.data)
255 
256         pkt_tcp = pkt.get_protocol(tcp.tcp)
257 
258         bits = pkt_tcp.bits if pkt_tcp else ''
259 
260         '''
261             faz as insercoes na whitelist
262             sao conexoes oriundas da rede externa
263         '''
264         if dst_ip == self.HONEYPOT_IP:
265 
266             if (protocol == 'TCP' and bits == 2) or protocol != 'TCP':
267                 self.whitelist.insert( (protocol, dst_port, src_port,
    src_ip) )
268 
269                 # obtem a porta de saida
270                 out_port1 = self.get_out_port(msg, dst_ethernet)
271                 out_port2 = self.get_out_port(msg, src_ethernet)
```

```
272
273             # define o padrao
274             if protocol == 'TCP':
275                 match1 = msg.datapath.ofproto_parser.OFPMatch(eth_type
=2048, ip_proto=6, tcp_src=src_port, tcp_dst=dst_port, ipv4_src=src_ip,
ipv4_dst=dst_ip)
276                 match2 = msg.datapath.ofproto_parser.OFPMatch(eth_type
=2048, ip_proto=6, tcp_src=dst_port, tcp_dst=src_port, ipv4_src=dst_ip,
ipv4_dst=src_ip)
277             else:
278                 match1 = msg.datapath.ofproto_parser.OFPMatch(eth_type
=2048, ip_proto=17, udp_src=src_port, udp_dst=dst_port, ipv4_src=src_ip,
 ipv4_dst=dst_ip)
279                 match2 = msg.datapath.ofproto_parser.OFPMatch(eth_type
=2048, ip_proto=17, upd_src=dst_port, udp_dst=src_port, ipv4_src=dst_ip,
 ipv4_dst=src_ip)
280
281             # define a acao
282             action1 = [msg.datapath.ofproto_parser.OFPActionOutput(
out_port1)]
283             action2 = [msg.datapath.ofproto_parser.OFPActionOutput(
out_port2)]
284
285             # adiciona as regras
286             self.switch_flows_semaphore.acquire()
287             self.add_flow(msg.datapath, 1, match1, action1)
288             self.add_flow(msg.datapath, 1, match2, action2)
289             self.switch_flows_semaphore.release()
290
291     # verifica se parte do honeypot e eh ataque
292     # neste caso, direciona o pacote para o container
293     if src_ip == self.HONEYPOT_IP and self.is_attack(protocol, src_port
, dst_port, dst_ip):
294
295         # verifica se ja existe um container para o IP de destino
296         container = self.map_ip_to_container.get(dst_ip)
297
298         # se nao existir, cria
299         if not container:
300
301             # ja insere na estrutura de controle para evitar multiplas
302             # tentativas de criacao do container para um
303             self.map_ip_to_container.add(dst_ip, ('-','-'))
```

```
304
305             # define o nome para o novo container
306             if ip_version == 'IPV4':
307                 new_container_name = 'net-cont-%s-%d-IP-%s' % (protocol
    , dst_port, dst_ip.replace('.','-'))
308             else:
309                 new_container_name = 'net-cont-%s-%d-IP-%s' % (protocol
    , dst_port, dst_ip.replace(':','-'))
310
311             # cria o container e insere as regras no switch
312             #self.create_container(msg, src_ethernet, ip_version,
    src_ip, dst_ip, new_container_name)
313             threading.Thread(target=self.create_container, args=(msg,
    src_ethernet, ip_version, src_ip, dst_ip, new_container_name,)).start()
314
315         else:
316             return
317
318
319     #if src_ip == self.HONEYPOT_IP or dst_ip == self.HONEYPOT_IP:
320         #self.logger.info('%s\n', pkt)
321         #self.logger.info('[%s - %s] De %s[%s] para %s[%s]', protocol,
    ip_version, src_ip, src_port, dst_ip, dst_port)
322
323     #envia a mensagem
324     self.send_packet_out(msg)
325
326 # cria o container
327 def create_container(self, msg, src_ethernet, ip_version, src_ip,
    dst_ip, container_name):
328
329     # obtem o template
330     template = self.DEFAULT_CONTAINER_TEMPLATE
331
332     # obtem o timestamp antes a criacao da LXC
333     first_time = time.time()
334
335     # cria uma conexao ssh com o host
336     ssh = SSH()
337
338     # cria o novo container
339     cmd = 'lxc launch %s %s' % (template, container_name)
340     ssh.exec_cmd(cmd)
```

```
341
342         # inicia o novo container
343         cmd = 'lxc network attach switch %s eth0' % (container_name)
344         ssh.exec_cmd(cmd)
345
346         # configura o IP do novo container
347         cmd = 'lxc exec %s -- ifconfig eth0 %s' % (container_name, dst_ip)
348         ssh.exec_cmd(cmd)
349
350         # adiciona uma rota default para outras redes
351         cmd = 'lxc exec %s -- route add default dev eth0' % (container_name
    )
352         ssh.exec_cmd(cmd)
353
354         # inicializa um web server em python
355         #cmd = 'lxc exec %s -- python -m SimpleHTTPServer 80 &' % (
    new_container_name)
356         #self.ssh.exec_cmd(cmd)
357
358         # obtem o endereco MAC do novo container
359         cmd = 'lxc exec %s -- cat /sys/class/net/eth0/address' % (
    container_name)
360         new_container_mac = ssh.exec_cmd(cmd).strip()
361
362         # fecha a conexao SSH
363         ssh.close()
364
365         # obtem o timestamp apos a criacao da LXC e antes da criacao da
    regra
366         second_time = time.time()
367
368         '''
369             INICIO: criacao da regra no switch - na ida e na volta
370         '''
371
372         # obtem a porta de saida
373         out_port1 = self.get_out_port(msg, new_container_mac)
374         out_port2 = self.get_out_port(msg, src_ethernet)
375
376         # define o padrao
377         match1 = msg.datapath.ofproto_parser.OFPMatch(eth_type=2048,
    ipv4_src=src_ip, ipv4_dst=dst_ip)
378         match2 = msg.datapath.ofproto_parser.OFPMatch(eth_type=2048,
```

```
    ipv4_src=dst_ip, ipv4_dst=src_ip)
379
380         # define a acao
381         action1 = [msg.datapath.ofproto_parser.OFPActionSetField(eth_dst=
    new_container_mac),
382                    msg.datapath.ofproto_parser.OFPActionOutput(out_port1)]
383         action2 = [msg.datapath.ofproto_parser.OFPActionOutput(out_port2)]
384
385         # adiciona as regras
386         self.switch_flows_semaphore.acquire()
387         self.add_flow(msg.datapath, 2, match1, action1)
388         self.add_flow(msg.datapath, 2, match2, action2)
389         self.switch_flows_semaphore.release()
390
391         '''
392             FIM: criacao da regra no switch
393         '''
394
395         # obtem o timestamp apos a criacao da regra
396         third_time = time.time()
397
398         print("%s criado com MAC %s em %f segundos" % (container_name,
    new_container_mac, (second_time - first_time)))
399         #print("%s criado com MAC %s" % (new_container_name,
    new_container_mac))
400
401         # define a tupla do container
402         container = (container_name, new_container_mac)
403
404         # atualiza a estrutura de controle
405         self.map_ip_to_container.set(dst_ip, container)
406
407         # obtem os dados da mensagem
408         pkt = packet.Packet(msg.data)
409
410         # altera o MAC do destino para o container
411         pkt[0].dst = new_container_mac
412
413         # informa que o checksum do transporte deve ser recalculado
414         pkt[2].csum = 0
415
416         #serializa o pacote
417         pkt.serialize()
```

```
418
419         #altera os dados da mensagem
420         msg.data = pkt.data
421
422         #envia a mensagem
423         self.send_packet_out(msg)
424
425
426     # verifica se um pacote eh ataque ou nao
427     def is_attack(self, protocol, src_port, dst_port, dst_ip):
428         # verifica a WhiteList ou
429         # verifica se o IP de destino esta na lista de servidores de
    comando e controle
430         if self.whitelist.in_list( (protocol, src_port, dst_port, dst_ip) )
     or self.command_ip_list.in_list(dst_ip):
431             return False
432         else:
433             return True
434
435 '''
436 ##########################################################################
437 Classe para comunicacao ssh com o host
438 Necessaria para criacao de manipulacao de containers
439 ##########################################################################
440 '''
441 class SSH:
442
443     def __init__(self):
444         self.ssh = SSHClient()
445         self.ssh.load_system_host_keys()
446         self.ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())
447         self.ssh.connect(hostname='10.0.2.1',username='asgard',password='')
448
449     def exec_cmd(self, cmd):
450         stdin,stdout,stderr = self.ssh.exec_command(cmd)
451         if stderr.channel.recv_exit_status() != 0:
452             return stderr.read()
453         else:
454             return stdout.read()
455
456     def close(self):
457         self.ssh.close()
458
```

```
'''
#############################################################################
Classe com estrutura de mapeamento (dicionario) entre chave -> valor
#############################################################################
'''
class Map:

    def __init__(self):
        # mapa
        self.my_map = {}

    # adiciona um item
    def add(self, key, value):
        if not self.my_map.has_key(key):
            self.my_map[key] = value

    # remove um item
    def remove(self, key):
        if self.my_map.has_key(key):
            del self.my_map[key]

    # obtem um item
    def get(self, key):
        if self.my_map.has_key(key):
            return self.my_map[key]
        else:
            return None

    # altera um item
    def set(self, key, value):
        if self.my_map.has_key(key):
            self.my_map[key] = value

    # imprime o dicionario
    def print_map(self):
        print self.my_map

'''
#############################################################################
Registra os IPs de servidores de Comando e Controle de Botnets
#############################################################################
'''
class IPList:
```

```
502
503     def __init__(self):
504         self.list = []
505
506         # realiza a leitura do arquivo para inicializacao da lista de IPs
507         try:
508             arq = open('ip_list.txt', 'r')
509             lines = arq.readlines()
510             for ip in lines:
511                 self.insert(ip.strip())
512             arq.close()
513             print('%d IPs of Command and Control Servers' % (self.size()))
514         except:
515             print('Error while reading file of Command and Control IP List'
    )
516
517     def size(self):
518         return len(self.list)
519
520     def in_list(self, ip):
521         return ip in self.list
522
523     def insert(self, ip):
524         if not self.in_list(ip):
525             self.list.append(ip)
526
527     def remove(self, ip):
528         if self.in_list(ip):
529             self.list.remove(ip)
530
531 '''
532 ##########################################################################
533 Registra os pacotes oriundos do honeypot que podem ser liberados para a
    internet
534 sem passar pela verificacao da etapa de machine learning
535 Se trata de conexoes iniciadas por hosts externos
536 ##########################################################################
537 '''
538 class WhiteList:
539
540     def __init__(self):
541         self.list = []
542
```

```
543     def size(self):
544         return len(self.list)
545 
546     def in_list(self, item):
547         return item in self.list
548 
549     # item = ('TCP', src_port, dst_port, dst_ip)
550     # item = ('UDP', src_port, dst_port, dst_ip)
551     # item = ('ICMP', 0, 0, dst_ip)
552     def insert(self, item):
553         if not self.in_list(item):
554             self.list.append(item)
555 
556     def remove(self, item):
557         if self.in_list(item):
558             self.list.remove(item)
```

Listing A.1: Honeynet Switch