Strengthening MT6D Defenses with Darknet and Honeypot Capabilities

Dileep Kumar Basam

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Engineering

Joseph G. Tront, Chair

Randolph C. Marchany

J. Scot Ransbottom

November 13, 2015

Blacksburg, Virginia

Keywords: Moving Target Defense, MT6D, Honeypot, Dionaea, Darknet, IPv6

Strengthening MT6D Defenses with Darknet and Honeypot Capabilities

Dileep Kumar Basam

(ABSTRACT)

With the ever increasing adoption of IPv6, there has been a growing concern for security and privacy of IPv6 networks. Mechanisms like the Moving Target IPv6 Defense (MT6D) leverage the immense address space available with the new 128-bit addressing scheme to improve security and privacy of IPv6 networks. MT6D allows participating hosts to hop onto new addresses, that are cryptographically computed, without any disruption to ongoing conversations. However, there is no feedback mechanism in the current MT6D implementation to substantiate the core strength of the scheme i.e., to find an attacker attempting to discover and target any MT6D addresses.

This thesis proposes a method to monitor the intruder activity targeting the relinquished addresses to extract information for reinforcing the defenses of the MT6D scheme. Our solution identifies and acquires IPv6 addresses that are being discarded by MT6D hosts on a local network, in addition to monitoring and visualizing the incoming traffic on these addresses. This is essentially equivalent to forming a darknet out of the discarded MT6D addresses. The solution's architecture also includes an ability to deploy a virtual (LXC-based) honeypot on-demand, based on any interesting traffic pattern observed on a discarded address.

With this solution in place, we can become cognizant of an attacker trailing an MT6D-host along the address changes, as well as understanding the composition of attack traffic hitting the discarded MT6D addresses. With the honeypot deployment capabilities, the solution can take the conversation forward with the attacker to collect more information on attacker methods and delay further tracking attempts. The solution architecture also allows an MT6D host to query the solution database for network activity on its relinquished addresses as a JavaScript Object Notation (JSON) object. This feature allows the MT6D host to identify any suspicious activity on its discarded addresses and strengthen the MT6D scheme parameters accordingly. We have built a proof-of-concept for the proposed solution and analyzed the solution's feasibility and scalability.

Acknowledgments

I want to thank my committee members Dr. Joseph G. Tront, Dr. J. Scot Ransbottom, and Prof. Randy Marchany for their guidance and mentorship along this path. I also thank Chris Morrell, Reese Moore, Mike Cantrell, Dr. David Raymond and everyone in the IT Security Office and Lab, for their support.

I thank my grandmother Rukminamma, my parents Annapurna and Nageswara Rao Basam and my brothers Dinesh, Ram and Lakshman for their continued love and encouragement over the years. I also thank my wife, Sirisha, for her patience, understanding and companionship.

Contents

List of Figures

Chapter 1

Introduction

Many cybersecurity attacks start with reconnaissance that involves passive monitoring of target network traffic and further network scanning to find potential victims with vulnerable services. Unfortunately, IPv4's address space is so small that the state-of-the-art attacker tools can scan a whole class-C subnet in about 4 minutes[1]. Even if hosts dynamically change addresses in IPv4, the limited address space and ease of scanning makes it easy to track the host across address changes for exploitation. Therefore IPv4 networks are vulnerable to network-correlation and reconnaissance attacks.

IPv6[2] with it's 128-bit address size offers a larger address pool, making reconnaissance attacks harder. An IPv6 enabled host can use either Dynamic Host Configuration Protocol (DHCP)[3] or Stateless Address Auto-Configuration (SLAAC)[4] protocol to acquire a routable IP address. In SLAAC, the host's physical MAC address forms the basis of the IPv6

address. The 48-bit MAC address contributes to the 64-bit interface-identifier(IID) portion of the IPv6 address. This mechanism poses privacy concerns, as a host can be tracked across address-hops using the IID portion that remains constant during the address changes. Even though the larger address space mitigates the threat vector of reconnaissance, static IPv6 addresses are still vulnerable to network-correlation attacks and the use of SLAAC approach opens up privacy concerns.

MT6D[5][6] addresses the above issues by taking advantage of the vast address space available in IPv6 to allow hosts to periodically change their network identity without interruption to the ongoing conversations. The address morphing behavior denies any advantage to an attacker passively monitoring target network traffic and increases the work-factor for him/her to compute the host's next address in order to be able to exploit the host. The concept of MT6D is discussed in greater detail in Section 2.1.3.

1.1 Problem Statement and Proposal

1.1.1 Problem Statement

Nodes participating in MT6D periodically relinquish IP addresses and hop on to new addresses. This address-hopping trait of MT6D improves the security and privacy of IPv6 networks, however, there is no feedback-mechanism in the current implementation i.e., a node engaged in a MT6D conversation have no means of realizing if there is an attacker

uncovering its MT6D addresses and trailing the MT6D node along its address hops.

1.1.2 Threat Model and Assumptions

We assume a threat model involving an an attacker with infinite resources (compute cycles and time), with an ability to brute-force the scheme parameters e.g., the secret-key involved in address-computation (hashing scheme), to discover the MT6D addresses. But the attacker validating these uncovered MT6D addresses will generate some traffic, leaving a trail of his/her activity on these discarded MT6D addresses.

1.1.3 Challenge

- To come up with a mechanism to use relinquished MT6D addresses in order to discover any anomalies, and gather intelligence on attacker methods
- To build a proof-of-concept solution to impart darknet and honeypot capabilities to MT6D scheme

1.1.4 Proposal

Our approach to solve this problem is to design a solution that identifies and acquires the addresses discarded by the MT6D hosts and enumerating incoming traffic on these relinquished addresses in addition to providing the ability to deploy a virtual container-based honeypot

configured with a specific discarded address upon detecting a suspicious traffic pattern. Our approach[7] includes building a Central-Node (CN) that passively listens to local-link traffic in promiscuous mode, identifying and acquiring discarded addresses by MT6D nodes and performing traffic enumeration on these addresses. The CN uses typical IPv6 neighbor discovery protocol (NDP) messages like Neighbor Solicit (NS) and Multicast Listener Discovery (MLD) messages to re-construct who is relinquishing which addresses and when to acquire them. We discuss the dissection of NS and MLD messages and how they can be used in the solution for identification and acquisition of discarded addresses in Section 2.

The solution's ability to deploy a honeypot tied to a discarded MT6D address of interest can potentially take up the conversation forward with the attacker to gather intelligence on attacker methods besides collecting attack traffic samples for further analysis. The solution also allows an MT6D node to query the database for incoming traffic on it's discarded addresses, thus giving the MT6D node the ability to analyze the activity to identify malicious traffic or an attacker persistently trailing on its discarded addresses, to change the scheme parameters e.g., move to a stronger secret key or a faster hopping interval to evade the attacker. Such a solution equipping MT6D with darknet and honeypot capabilities will serve as a feedback mechanism to strengthen defenses for the involved MT6D hosts along their address hopping journey.

1.2 Goals

In this thesis, we seek to impart darknet and honeypot capabilities to Moving Target IPv6 Defense (MT6D) in order to provide a feedback-mechanism for evaluating the strengths of MT6D scheme by analyzing the activity on discarded addresses. Among the goals of this effort, the first goal is to implement the Central-Node (CN) logic, comprised of traffic-parsing, address-acquisition, traffic enumeration, analysis, and visualization modules. A second goal is to implement the database-integration in to Central-Node along with developing a way to build web-server to offer real-time visualization of the local MT6D addresses that are being relinquished and incoming traffic on these discarded addresses. Our next goal is to implement a capability of deploying honeypot-service on a virtual-container on-demand bound to a discarded MT6D address of interest with minimal interruption observable to an attacker. Lastly, we want to analyze the feasibility and scalability of the proposed solution by analyzing the the CPU and memory overhead trends while scaling to multiple honeypot-containers.

1.3 Structure of Thesis

This thesis is organized by first discussing the background in Chapter 2. A survey on related work is provided in Chapter 3. We describe our approach and pre-work experiments in Chapter 4. Then we explain our testbed and implementation details in Chapter 5. The results of our evaluation and observations are reported in Chapter 6. Finally, in Chapter 7

we provide our conclusions and directions for future work.

Chapter 2

Background

In this chapter we will present background on IPv6, Neighbor Discovery Protocol (NDP), and other components that were used in building the proposed solution.

2.1 IPv6

On 24 September 2015, the American Registry for Internet Numbers (ARIN) announced that it allocated the final IPv4 address blocks in its free pool and the IPv4 address space is completely exhausted. The IPv6 protocol by design overcomes the address exhaustion limitation with its address length of 128 bits. This new address size allows for 2^{128} possible addresses, approximately $7.92X10^{28}$ addresses for every possible IPv4 address. This has been a driving force behind IPv6 adoption and the trend can be witnessed in the Figure 2.1, which depicts IPv6 adoption statistics among Google users.

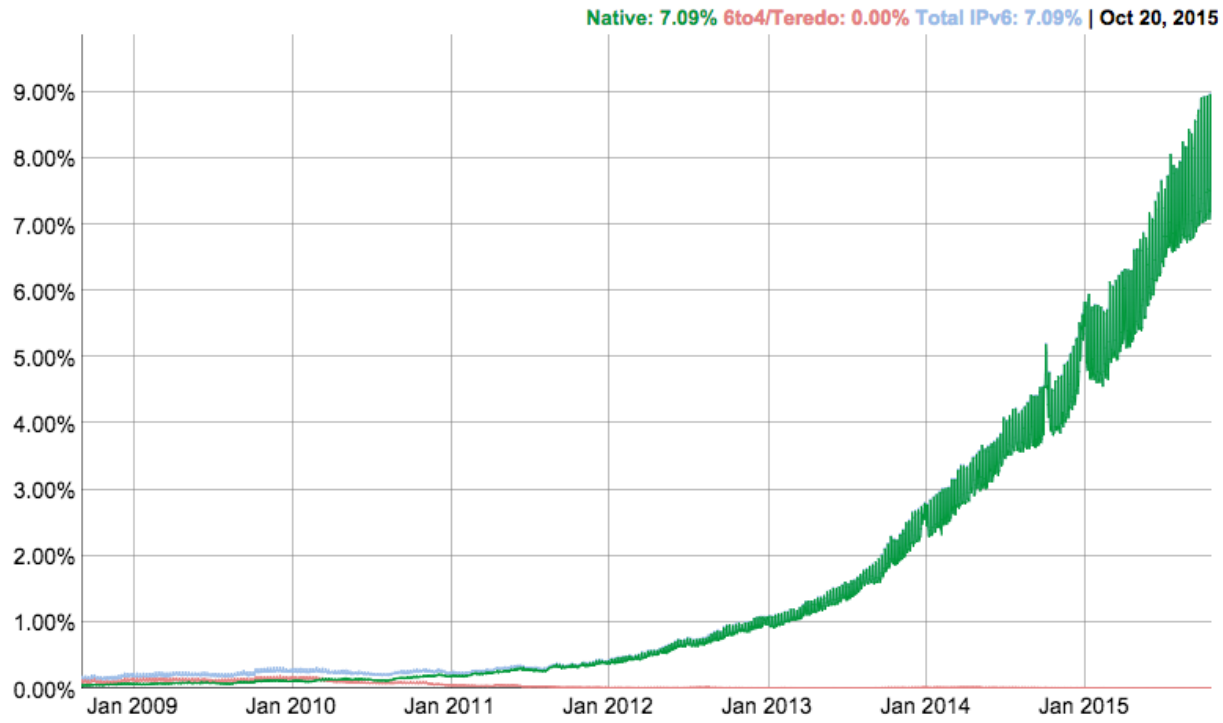

Figure 2.1: IPv6 adoption statistics among Google users [8]. [Google. Ipv6 adoption statistics among google users. http://www.google.com/intl/en/ipv6/statistics.html. Used under fair use, 2015.]

To understand IPv6 address design, Figure 2.2 depicts various IPv6 header fields and we can observe that the IPv6 address field is designed for 128-bit length. As shown in Figure 2.3, each IPv6 address is made up of subnet and interface-identifier(IID) portions of 64 bits each.

2.1.1 Neighbor Discovery in IPv6

The process of Neighbor Discovery (ND)[9] in IPv6, replaces Address Resolution Protocol (ARP) in solving the problem of discovering other nodes on the local link. It also has a mechanism to discover local routers on the network. Neighbor Discovery Protocol (NDP)

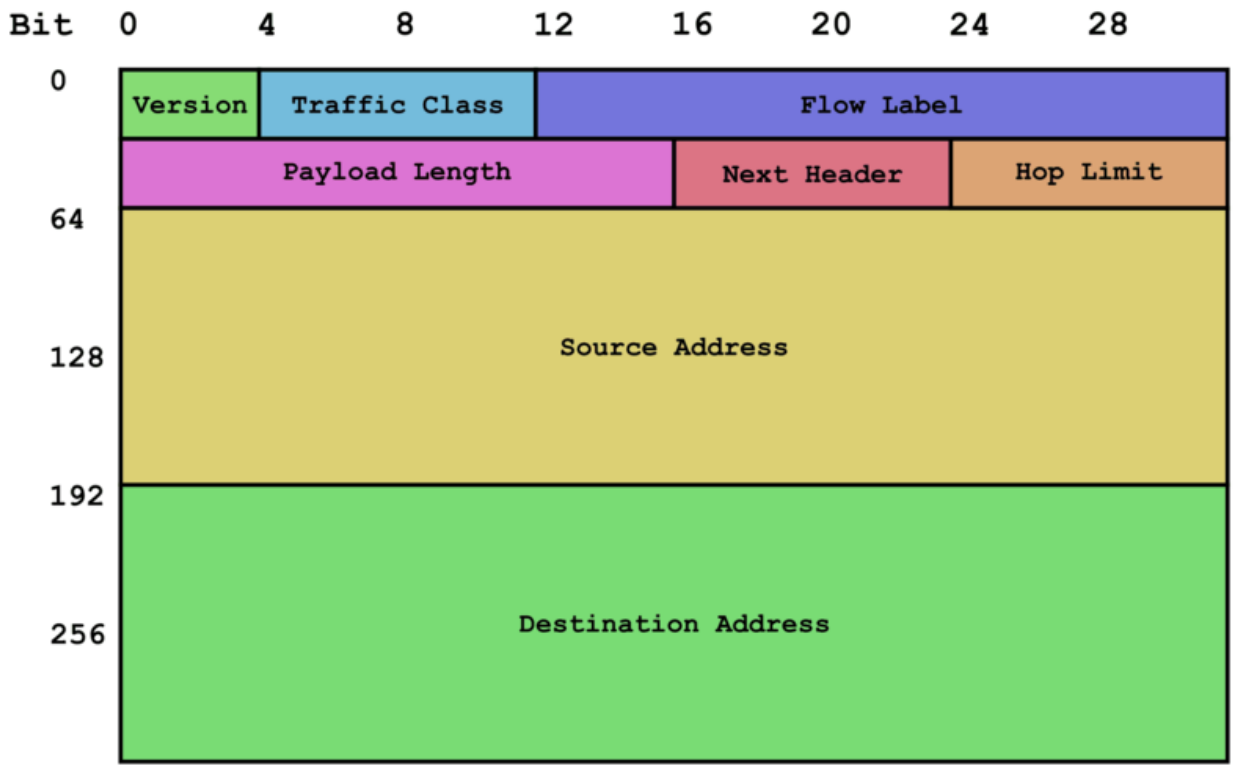

Figure 2.2: IPv6 header format[IPv6 Header. https://commons.wikimedia.org/wiki/File:Ipv6_header.svg. Used under GNU Free Documentation License.]

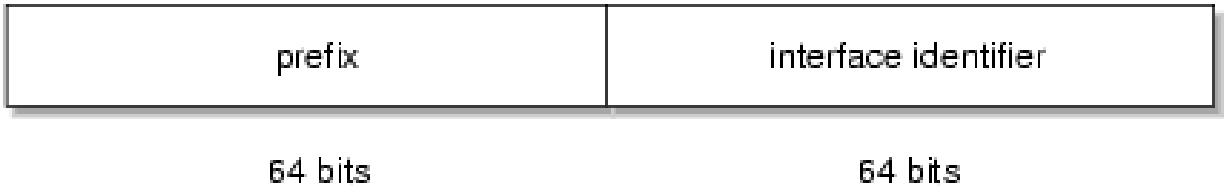

Figure 2.3: IPv6 address composition

can also facilitate duplicate address detection, and can redirect nodes to another router when needed. The following sections explain address resolution and Duplicate Address Detection(DAD) mechanisms in more detail.

Address Resolution through NS and NA messages

An IPv6 host uses Internet Control Message Protocol Version 6 (ICMPv6) messages like Neighbor Solicit (NS) and Neighbor Advertisement (NA) to accomplish neighbor discovery. An IPv6 host sends a Neighbor Solicitation (NS) message to a multicast address as specified by the target address to learn the link-layer address of another node on local link. The target node which should be listening to the multicast address, upon receiving the solicitation, should reply with a Neighbor Advertisement (NA) message. NA message includes flags like the override (ovr) flag, to tell the receiver to replace any information that the recipient (which may be a local router) may already have in it's routing cache.

Figure 2.4 shows a Wireshark capture dissection of a Neighbor Solicit (NS) message. NS message is an ICMPv6 message with payload consisting of fields like 'target-address' containing the IPv6 address that the host is interested in discovering/acquiring and 'Source link-layer address' that has a corresponding MAC address. As you can observe in Figure 2.4 the IPv6 host with address 2001:468:c80:c111:a693:bdfa:3b28:3a12 is sending out the concerned NS message with the target IPv6 address, 2001:468:c80:c111:9c29:f227:6717:a2b7, to be acquired along with the MAC address, 00:50:56:96:ac:7e, associated with the target IPv6 address.

Duplicate Address Detection (DAD)

DAD is built into IPv6 as part of its address acquisition mechanism. When a node comes up on a network and wants to be assigned a new address, it must first validate that no other

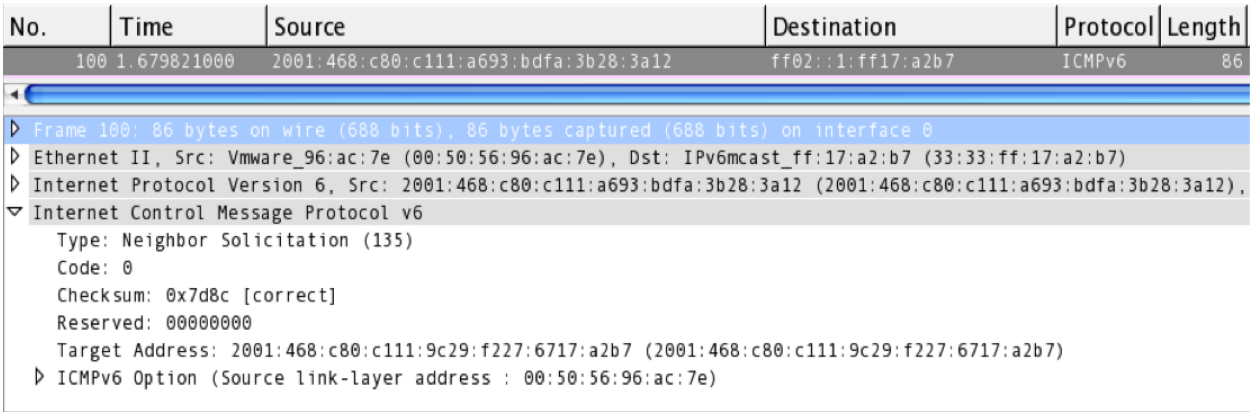

Figure 2.4: Wireshark capture dissection of a Neighbor Solicit (NS) message

node on the local-link uses this particular address. In other words, the node should ensure that there is no other host with a duplicate of this IP address already live on the network. This is accomplished by sending out a series of NS messages and looking for NA replies. If some other node is already using the address, that node will send a NA message as a reply. Upon seeing a NA message, the first host must select a new tentative address and repeat the process. If no other host replies with a NA, the first host is free to use the address of concern.

2.1.2 MLD

MLD stands for Multicast Listener Discovery. We leverage MLD messages in order to identify the time instant a host acquires or relinquishes an IPv6 address. MLD is used by an IPv6 host to indicate specifically which multicast addresses are of interest to listen on. Each MLD message consists of a set of records, with each record containing an IPv6 multicast address. Each message will have records that are segregated based on filter mode.The filter mode

can be either INCLUDE or EXCLUDE. In INCLUDE mode, reception of packets sent to the specified multicast address is enabled only from the source addresses listed in the source list. In EXCLUDE mode, reception of packets sent to the given multicast address is enabled from all source addresses except those listed in the source list.

Whenever an IPv6 host acquires a new address, it will send out a MLD message with multicast address record containing last three-octets of acquired address prefixed with ff02::1:ff with the EXCLUDE filter-label. Similarly when a node relinquishes an IP address, it sends out an MLD message with the relevant multicast address record under INCLUDE filter-label[10].

Figure 2.5 shows Wireshark capture dissection of a MLD message. We can observe from ICMPv6's payload consisting of multicast address records with include and exclude labels and infer that the host at IPv6 address fe80::250:56ff:fe96:871d is dropping two IPv6 addresses with (last 3-byte) suffixes 45:42d0 and 22:bee5, and has acquired an IPv6 adddress with suffix da:6ad2.

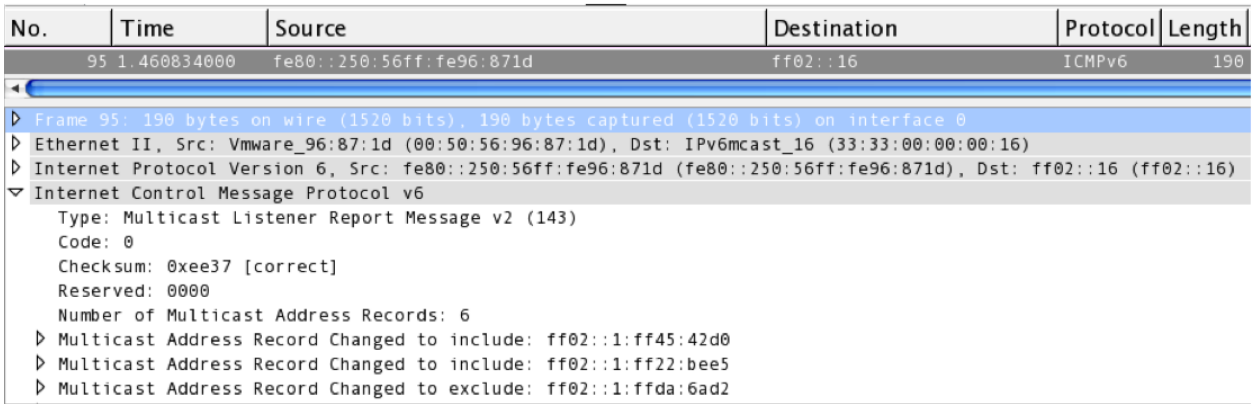

Figure 2.5: Wireshark capture dissecting a MLD message

2.1.3 Moving Target IPv6 Defense(MT6D)

Moving Target IPv6 Defense[5][6] is a technique that aims to prevent attackers from correlating network traffic, to probe, or to pinpoint a host for exploitation. Instead of having static addresses that allow attackers to easily target and exploit systems, employing a moving target defense system involves nodes periodically hopping to new addresses while dumping the old addresses. This achieves an effect similar to frequency hopping in radio networks, for an attacker passively listening to the conversation of two MT6D hosts will see multiple hosts communicating to each other rather than a pair of hosts as shown in figure 2.6. This kind of consistent address changing behavior gives additional security at the network layer, as the attacker needs to start with reconnaissance all over again and re-acquire the target host. The MT6D protocol changes IP addresses for communicating hosts in synchronization. MT6D achieves synchronization between the involved nodes by having both ends of a communication pair compute both their own IP-address and their remote-peer's IP address during the

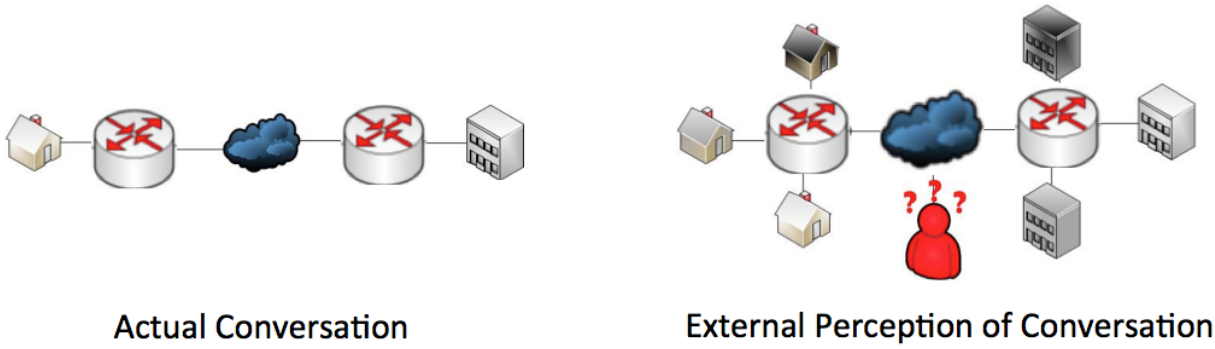

Figure 2.6: Moving Target IPv6 Defense[11] [Stephen Groat, Matthew Dunlop, W. Urbanksi, Randy Marchany, and Joseph Tront. Using an ipv6 moving target defense to protect the smart grid. In Innovative Smart Grid Technologies (ISGT), 2012 IEEE PES, pages 1-7, Jan 2012. doi: 10.1109/ISGT. 2012.6175633. Used under fair use, 2015.]

particular time window 't'. The scheme involves computation of the Interface-Identifier part of the address i.e., IID' extracted from first 64 bits from a hash-digest (H) of concatenated string made of a symmetric Secret Key (SK), initial IID, and current time window 't'.

$$IID' = H[IID\_initial||SK||t]_{0 \rightarrow 63}$$

2.1.4 Dionaea

Dionaea is a low-interaction honeypot offering IPv6 support with an ability to capture malware in addition to logging suspicious traffic. Dionaea offers flexibility to configure services and interfaces to listen on. Dionaea uses Sqlite as the backend database allowing custom queries to be made to extract interesting information from logs stored in the database[12].

2.1.5 DionaeaFR

DionaeaFR is an open-source library that offers front-end visualization of Dionaea's logs that are stored in the sqlite database. DionaeaFR's web-server is implemented in Python and Django framework. This front-end console allows retrieving statistics of traffic hitting the honeypot in addition to offering a way to download attack traffic samples e.g., malware[13].

2.1.6 LXC

LXC is a light-weight virtualization technology that relies on isolated userspaces to create virtual containers that share same kernel. LXC uses Linux kernel features like namespaces, Chroots, CGroups etc., to isolate the container processes. LXC differs from the concept of standard virtual-machines (VM) by sharing the same kernel and underlying hardware which obviates the need for a hypervisor[14].

2.1.7 Other components used in building the Solution

Scapy, Pcapy, Impacket and Pyroute2

Scapy is a Python module for packet crafting and manipulation tool for computer networks, written in Python by Philippe Biondi. We used scapy during the experiments to simulate attacker (probing) traffic by forging packets and also for crafting forced Neighbor Advertisements (NA) packets for expediting the address take over[15].

Pcapy is another Python extension module that interfaces with the libpcap packet capture library to capture packets on the network. We used it for capturing incoming packets on the acquired addresses on the central node (CN).

Impacket is a collection of Python classes to decode the raw network traffic in terms of protocols. We used Impacket along with Pcapy to parse the raw network traffic to glean the IP address and Port numbers for both developing IP-MAC associations and Traffic

enumeration on the acquired addresses.

Pyroute2 is a Python netlink and Linux network configuration library. We used it to bind the IP addresses to the Linux host.

Wireshark

Wireshark is a popular packet capture and analysis tool. We used Wireshark extensively in pre-work experiments to analyze captures in order to understand the address transition times and packet loss involved during the address take-over[16].

MongoDB and PyMongo

MongoDB is a NoSQL database, that stores each record in the database as a JSON object thus deviating from SQL's traditional table-based relational database structure. A typical Mongo database is composed of collections and each collection is made up of set of documents/records.

PyMongo is a Python library that acts as an application programming interface (API) to interact with MongoDB from Python code.

D3.js

D3.js is a JavaScript library that leverages HTML, SVG and CSS to produce complex visualizations on a web browser. We forked an existing repository of a radial-diagram(a kind of

D3.js visualization) and adapted it for building the visualization module[17] [18].

Dstat

Dstat is a free tool that combines vmstat, iostat, netstat and ifstat to view system resources in real-time. The tool is used in this work to collect real-time CPU (usr/sys) and free-memory (RAM) statistics during the experiments[19].

This chapter presented background on IPv6, MT6D, and components like Dionaea, LXC and various python Libraries etc., used in building the solution. The next section presents details on the related work and how our work extends the prior art in this area.

Chapter 3

Related Work

There has been lot of past work in the area of designing darknets, honeypots, honey-nets and leveraging these implementations for collecting information about attacker and attack techniques. Finally, we will describe how work presented in this thesis extends the previous research.

3.1 Darknets a.k.a Network Telescopes

As per [20], "A darknet is an advertised and routed portion of Internet address space that contains no advertised services". Therefore any traffic that is incoming on a darknet is deemed illegitimate. Darknets have been leveraged in IPv4 networks to uncover malicious traffic and malware trends. Work by Ronan et al.,[20] created the first IPv6 darknet to detect the backscatter present on the IPv6 Internet and they also studied how this traffic

analysis differs from that of the IPv4 Internet.

In a similar effort, Researchers at UCSD came up with a Network Telescope, a globally routed /8 IPv4 subnet with a few provider allocated addresses. This network carries almost no legitimate traffic. After discarding the legitimate traffic from the incoming packets, the remaining data represent a continuous view of anomalous unsolicited traffic, or Internet Background Radiation (IBR). It was pointed out that IBR or the backscatter may have sources in spoofed source denial-of-service attacks, Internet worms and viruses, scanning of address space by attackers or malware looking for vulnerable targets, and various misconfigurations. [21]. For example a /24 network telescope was used for Conficker analysis, where the research team [22] analyzed dataset of some 16 million packets targeting port 445/tcp in the South African IP address space. This is followed by a detailed analysis of the packet characteristics observed, including size and TTL and intricacies of observed target selection and the flaw in the Conficker worm's propagation algorithm.

3.2 Honeypots and Honeynets

A honeypot is a security resource that delivers insights by getting probed, attacked, or compromised by malicious entities [23] and subsequently reporting the characteristics of the attack. Honeypots are classified as either low or high interaction based on their designed level of interaction with the potential attacker. Low-interaction honeypots, as their name suggests, are often limited by the degree of interaction. Examples of low interaction honeypots include

Honeyd, Dionaea, etc. On the other hand high-interaction honeypots support complex behavior by emulating a real operating system with full suite of applications. High-interaction honeypots carry more risk by allowing full compromise by the attacker. Honeypots have no legitimate production use for incoming traffic, so in theory any traffic hitting a honeypot can be deemed suspicious and warrant inspection.

On the other hand, a honeynet is a basic network of commonly used operating systems, such as Red Hat Linux or Windows NT in their default configurations. Despite not broadcasting the identity of the Honeynet, nor luring attackers, it collects illegitimate incoming traffic which is further investigated[24].

Kuwatly's work [25] discusses a dynamic honeypot solution built from the fingerprinting tool (p0f) and Nmap to dynamically configure and leverage Honeyd-based emulated virtual hosts and a physical high-interaction honeypot cluster to capture and analyze attacker traffic. Hecker et al.'s paper [26] proposes a solution that uses nmap for fingerprinting the local network (topology, hosts, ports, etc.) and honeyd-configuration manager for dynamically building honeypots.

Kishimoto et al.'s work [27] on dynamic honeypot commissioning involves detection of incoming address scans targeting an unallocated IP address. Research work done by [27] on commissioning honeypots based on incoming address scans, shows that disabling Duplicate Address Detection (DAD) makes the address acquisition faster. During the pre-work tests, we analyzed the relevant network captures and found that we can significantly reduce the delay further by enabling a node to pro-actively claim ownership of the acquired address

without waiting for NS messages from Router. More details are presented in the section 4.2. Hieb [28] employs dynamic honeypots and monitoring network activity of deployed honeypots to setup anomaly-based intrusion detection for the network. Brzeczko's work [29] on Turnkey Honeynet framework involved automatic commissioning of Honeypots (Dionaea, Kippo, Glastopf) depending on the composition of live attack traffic.

Memaris[30] builds a virtual honeynet and compares LXC virtualisation with other virtualisation methods including VMware, VirtualBox and KVM and concluded that the LXC approach provides better performance. Work by Pisarcik et al., [31] on Distributed Virtual Honeynets framework talks about a master control center and a network of high-interaction virtual honeynets based on OpenVZ and LXC virtualization.

Previous research work in this area involved creating Darknets, Network Telescopes and Honeynets, with no advertised services and listen for illegitimate traffic. There are no current implementations adapting such a scheme to moving target defense schemes like MT6D. Also unlike a static darknet that passively listens on unallocated address space, our solution is dynamic as it learns IP-address and MAC address associations from live network traffic to actively acquire the addresses being purged in order to gather intelligence on attacker methods. Work presented in this thesis[7][32] also extends the related work discussed above in terms of a full IPv6 implementation framework of on-demand honeypot commissioning using Linux-containers (LXC) in addition to proposing a method to leverage such a solution to fortify MT6D defenses. This solution presents a simplified visualization of attacker traffic by ignoring the MT6D nodes that don't have interesting incoming traffic to enable

an uncluttered view of the suspicious traffic. This work also presents a memory and CPU overhead analyses of the proposed solution while scaling to multiple honeypot-containers.

Chapter 4

Approach and Pre-work

4.1 Approach

The central idea of the research effort is to devise a mechanism for the nodes on an MT6D network to gather intelligence on attacker methods by watching for suspicious traffic on relinquished MT6D addresses. To develop a system that can acquire the addresses being released by MT6D nodes, a straight-forward approach is to have all of the keying material (secret keys being used by MT6D nodes for address generation) and other parameters like the IID for all the participating nodes hosted on the CN. This makes the CN capable of computing the next address hop for any MT6D node on the network. But this idea may invite an inherent vulnerability as such a hosting server would become a single point of failure. To avoid this, we developed an alternate approach after analyzing the network traces. Our

novel approach uses NDP and MLD message traffic that is common to any node running on a IPv6 network.

To implement such a scheme, as shown in figure 5.1, we have a MT6D-host that periodically hops onto new addresses while relinquishing old addresses. A Central-Node (CN) passively listens to the network traffic in promiscuous mode to parse the NS and MLD messages from MT6D hosts to populate a database collection of who's relinquishing what address, identifying the right time-instant to acquire these discarded addresses and subsequently binding these addresses to its local network interface. After the CN binds these discarded addresses to its network interface, it analyzes the incoming traffic on these discarded addresses and may place a request to the honeypot-host for a honeypot-container to be deployed on a specific IPv6 address. The challenge is to migrate the IP address from the CN to virtual LXC container on the honeypot-host (honeypot-container) with minimal interruption observable to an external attacker who is sending the traffic to the discarded address. The steps involved in this process are unbinding a specific IP address on the CN, binding this address on honeypot-container, and finally invoking and binding Dionaea and DionaeaFR services to this IP address on the honeypot-container.

We propose an efficient and quick approach for honeypot deployment by maintaining hot-spares of honeypot-containers. To facilitate the hot-spares approach, we packaged an LXC container with all prerequisites such as Dionaea, DionaeaFR, and supporting python libraries to act as a honeypot LXC-template. The honeypot-commissioning-module on honeypot-host clones a fixed number of containers from the honeypot LXC-template in advance and main-

tains a list of available containers. By keeping honeypot-containers waiting in a queue, deploying a honeypot configured with a target IPv6 address requires that the only configuration needed on the container-side is binding the IP address to the network interface of an available container and starting Dionaea honeypot and DionaeaFR (management web-server) services. This minimal configuration requirement ensures minimal delay in bringing up honeypot service bound to the desired IPv6 address.

In order to minimize the interruption window observable by an attacker during the migration of IP address from CN to honeypot-container, we followed the approach of holding the address for a fixed-period after receiving the message that a honeypot is being deployed from the honeypot-host. For a smoother IP transition from CN to the honeypot-container, based on our initial tests, we implemented the heuristic of CN holding the address for one second. This address-holding-period compensates for the time it takes the IP address to be bound on container's network interface and to start Dionaea and DionaeaFR services on the honeypot-container.

We went with the approach of using the honeypot-host instead of running a honeypot service on the Central-Node(CN) itself for two reasons. First, running honeypot on the CN may not be a secure approach as the Central-Node (CN) not only maintains a database of the discarded addresses, but also actively collects the addresses that are currently active on MT6D hosts from the parsing of NS and MLD messages. Secondly, a design including a dedicated honeypot host with its own database allows flexibility for such a honeypot-host running virtual machines with vulnerable-services to sit in a segregated zone like a DMZ

in the future while its partner Central-Node can still be on the local-network with MT6D nodes.

Even though we used a low-interaction honeypot, Dionaea, for experiments, by the virtue of using LXC containers in our solution, the solution allows high-interaction honeypot deployment on the container. So we are not limited to emulated services or virtual hosts in future experiments.

For a solution that involves learning and acquiring addresses from MT6D nodes on local-link, it is crucial to understand whether a host can acquire these addresses without major delays. If there were to be a delay, incoming traffic from a potential attacker may get lost in transit and this interruption may give cues to the attacker about the IP address migration. To achieve this, we conducted initial experiments to demonstrate faster address migration of IP address from one machine to another through the concepts of disabling DAD and forced Neighbor-Advertisement to the local router. We present this experiments in the next section on pre-work experiments.

4.2 Pre-work experiments on address migration

We ran experiments to test address migration times, or the time it takes for CN to identify (through parsing of NS/MLD conversations) and acquire a MT6D discarded address. Some of the observations from these experiments were used for faster address transition between the Central-Node and deployed honeypot-container for a minimal interruption window to an

external attacker.

For this test, we generated a continuous ICMP pingflood from a machine on a remote Internet connection, targeting the MT6D node's current address. After the stipulated address change interval when the MT6D node relinquishes this address, the CN tries to acquire it by binding the address to local network interface and sending a (Scapy) crafted packet advertising to the local router reflecting the new owner. In the following sections we will review the significance of Duplicate Address Detection (DAD) and forced neighbor-advertisement (NA) in speeding up the address acquisition process.

4.2.1 Concept of disabling DAD

As per work done by [27] in typical IPv6 address acquisition, a host must perform DAD in order to make sure it is not going to use an already-in-use address, but their honeypot system skips DAD and hands it off to address-management-system. For the solution presented in this work, the CN doesn't need to perform DAD as the address was in use by one of the legitimate MT6D nodes and the functionality of DAD would have been completed by the MT6D node during the address acquisition phase.

For the address takeover with DAD disabled on the CN, we observed that the onus being pushed on to the local router to send a NS message to solicit the new owner of the address to be able to route incoming pingflood traffic. Only when the router gets hit with traffic destined to the address of concern will the router send out a NS message seeking the address owner. In

No.	Time	Source	Destination	Protocol	Sequence	Sequence	Info
14055	55.832077000	External_pinghost	pc2-44.local	ICMPv6	36879	36879	Echo (ping) request id=0x1a42,
14056	55.841647000	External_pinghost	pc2-44.local	ICMPv6	36880	36880	Echo (ping) request id=0x1a42,
14057	55.852455000	External_pinghost	pc2-44.local	ICMPv6	36881	36881	Echo (ping) request id=0x1a42,
14058	55.862126000	External_pinghost	pc2-44.local	ICMPv6	36882	36882	Echo (ping) request id=0x1a42,

(a)

Figure 4.1: Wireshark capture showing initial ICMP pingflood hitting MT6D-host

No.	Time	Source	Destination	Protocol	Sequence	Info
2833	*REF*	router_local	ff02::1:ffcd:abcd	ICMPv6		Neighbor Solicitation for 2001:468:c80:c111:baac:abcd:abcd:abcd
2834	0.000046000	pc2-44.local	router_local	ICMPv6		Neighbor Advertisement 2001:468:c80:c111:baac:abcd:abcd:abcd (s
2835	0.011388000	External_pinghost	pc2-44.local	ICMPv6	36884	Echo (ping) request id=0x1a42, seq=36884, hop limit=48 (reply i
2837	0.022266000	External_pinghost	pc2-44.local	ICMPv6	36885	Echo (ping) request id=0x1a42, seq=36885, hop limit=48 (reply i
2839	0.031807000	External_pinghost	pc2-44.local	ICMPv6	36886	Echo (ping) request id=0x1a42, seq=36886, hop limit=48 (reply i
2843	0.042584000	External_pinghost	pc2-44.local	ICMPv6	36887	Echo (ping) request id=0x1a42, seq=36887, hop limit=48 (reply i

(a)

Figure 4.2: Wireshark capture showing ICMP pingflood hitting the CN indicating successful address transition

response to the NS message, the CN sends out a NA message claiming the address. Once the local router receives the NA message, it updates its routing table entries to reflect the address transition. Any packets that arrive during this owner-finding transition get dropped at the router. This can be observed in the Wireshark packet captures on MT6D-host and CN shown in figures 4.1 and 4.2. In figure 4.1 we can see the pingflood from a machine on Internet (External_pinghost) hitting the MT6D node (pc2_44.local) initially, and in figure 4.2 the pingflood traffic moves on to the CN(pc2_44.local) once CN completes address acquisition. As we can observe, one ICMP packet with sequence number 36883 is getting dropped at the local-router during the process of finding the new address owner (router sending out NS message seeking address-owner and waiting for NA reply). We identified a way to further optimize this mechanism by eliminating the need for the router to send out NS messages in the process of seeking new address-owner. We discuss the solution in the following section.

4.2.2 Concept of Forced NA for quick address acquisition

Once we understood that the address hand-off delay lies with router waiting on an NA reply from the new address owner (the CN) to establish the updated routing table entry, we sped up the process by having the CN craft the NA packet soon after the address binding and send it to local-router in an unsolicited manner. This forced NA message pro-actively advertises the new address that has been assigned to the local-router. The router, upon seeing a NA message with Override (ovr) flag set, updates its routing table entries to reflect the CN as the new address-owner, completing the address takeover.

In this chapter we have discussed our approach in designing the solution and pre-work experiments that were performed to evaluate mechanisms for faster address migration from one host to another host. We present the details on testbed used in our experiments along with solution architecture and implementation details in the next section.

Chapter 5

Testbed and Implementation

5.1 Testbed

The test setup as shown in Figure 5.1 involves three Linux desktops running the Ubuntu 12.04 operating system connected to a layer-2 switch with an uplink to the university network router. Virginia Tech has fully operational production IPv6 network. The MT6D host periodically acquires new addresses while relinquishing old addresses. The Central-Node (CN) identifies and acquires the discarded addresses on the local network and is also responsible for performing traffic enumeration, analysis, and visualization. Third Linux machine is the honeypot-host system that is installed with LXC software to support on-demand creation of containers to act as individual honeypots. To set context for the memory and CPU overhead analyses in section 6, the honeypot-host system is a Dell Optiplex Desktop running 64-bit

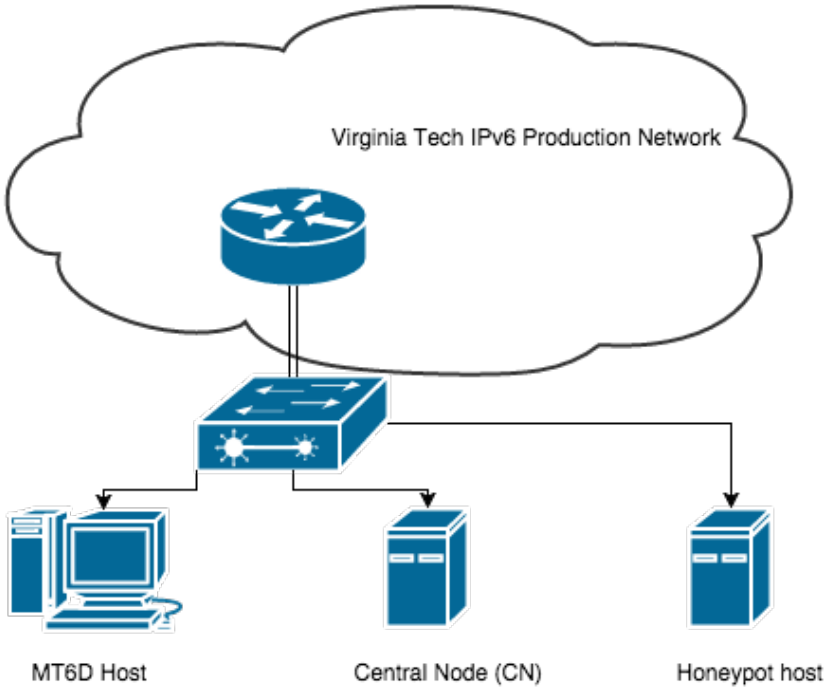

Figure 5.1: Block diagram of test setup

12.04 Ubuntu operating system installed with Intel Core-2 6700 2.66GHz CPU, 4GB RAM and 80GB of secondary memory.

Figure 5.2 depicts the architecture of our solution. The solution consists of Central-Node (CN) block integrated with the database (MongoDB) and web-server (Node.js) modules for inter-module communication and real-time visualizations. The honeypot-host block is implemented with on-demand honeypot deployment capability integrated with its dedicated (MongoDB) database.

In this section we have discussed the solution architecture and testbed to accomplish the goals of imparting darknet and honeypot deployment capabilities to MT6D networks. In the next section we discuss more details on the implementation specifics and various components

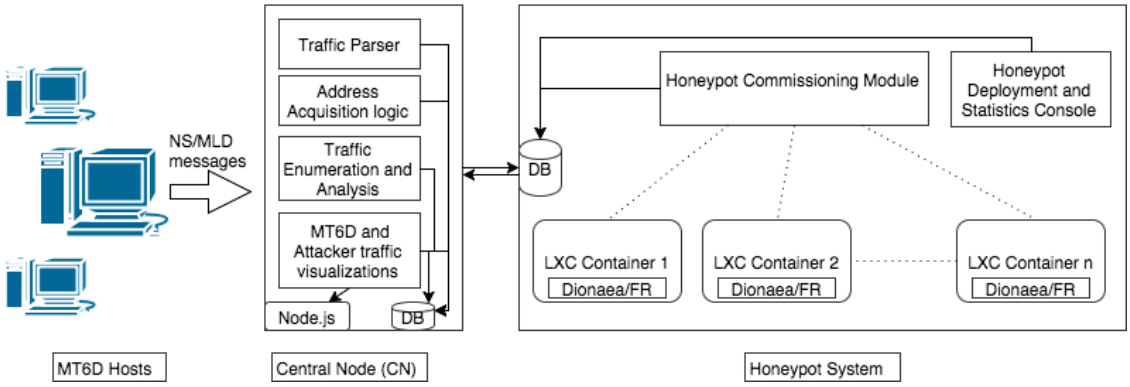

Figure 5.2: Solution architecture

of the solution. The implementation involves building the Central-Node(CN) and honeypot-host blocks. Each of these blocks is integrated with a dedicated database, so that one module can communicate its output to another module through the database.

5.2 Implementation

The CN includes traffic-parsing, address-acquisition, enumeration, analysis, and visualization modules. The CN uses Pcapy and Impacket libraries in its traffic-parsing module to listen to neighbor discovery (NS and MLD messages) conversations of all MT6D nodes on local link to learn the addresses that are being relinquished by the MT6D nodes and stores these associations in a database. It employs the Pyroute library in address-acquisition module to acquire and bind these learned addresses to local host and Scapy to send out a forced NA claiming ownership of the bound address as an unicast to the local-router. The Central-Node's traffic-enumeration module uses the same Pcapy and Impacket libraries to parse the incoming traffic on these acquired addresses and does traffic enumeration (Source IP

address, Destination Port, Packet Count), storing this data in the form of a native python dictionary. These python-dictionaries holding enumerated traffic data get converted to JSON objects for facilitating the visualizations. A node.js web-server has been built on the CN to offer visualizations based on real-time traffic (MT6D addresses that are being spawned and incoming traffic on acquired addresses). The visualized data offers two views, MT6D_view and attacker_view (figures 6.1, 6.4, 6.5), allowing an administrator to see whether an attacker is able to trail the MT6D node along the spawned addresses. One can also look at the port-number and packet-count of the incoming data from the attacker on the visualization to understand the composition of the incoming traffic. The algorithm in figure 5.3 explains the intricacies of traffic-parsing and enumeration modules of the CN.

```
procedure PacketProcessing(incoming_pkt)

    if incoming_pkt.type = NS then
        Learn IP <-> MAC association
        Store this association in neighbor_table

    else if incoming_pkt.type = MLD then
        Identify Multicast addresses with INCLUDE
        Store this Multicast address record as Key
        Use this key to retrieve the actual purged IP
                    from the learned neighbor_table
        Acquire this IP address
        Craft a NA message with override flag set
        Send this NA message as unicast to local router
        Delete the utilized IP <-> MAC association
                                from neighbor_table

    else if incoming_pkt.type = TCP or UDP or ICMP
then
        Capture the Source IP address, Dest port
            and maintain a count of such packets
    end if

end procedure
```

Figure 5.3: CN's packet processing algorithm

We also had to build some intelligence in to the Central-Node for an efficient implementation. We had to ensure that the CN doesn't learn IP-MAC associations and relinquished addresses from itself, as the CN itself may be dropping and acquiring temporary IPv6 addresses. We don't want these addresses to create any confusion during the neighbor_table look-ups for retrieving the full IP address for the multicast record suffix parsed from the MLD message. The CN also doesn't learn from the local Router's NS messages, because the router might be broadcasting the NS messages to see who owns a particular address, while finding a route to that address and not to claim ownership of the concerned address. We also included code that corroborates the MAC address the IP was learned from, to ensure better accuracy while matching the multicast address suffix to retrieve the full IP address from the neighbor_table.

The honeypot-host block includes the honeypot-commissioning module, the LXC, and a database. The honeypot-host database hosts two collections, hpot_requested_ collection and hpot_deployed_collections, for exchanging information on addresses on which the honeypot is being requested or deployed. Figure 5.4 depicts the flow-diagram that explains the sequence of messages exchanged between the blocks and actions performed during the operation. Step 1 involves the CN listening to and parsing local-link NS/MLD messages to keep a list of MT6D addresses being relinquished on the local network. In Step 2, upon seeing an MT6D node relinquish an address, the CN assigns the address to itself to facilitate traffic enumeration on this address. In Step 3, we have interesting incoming traffic hitting a discarded MT6D address (IPv6-address-of-interest) that is currently bound to the CN. Step 4 involves the CN requesting that a honeypot be deployed on a specific IPv6 address by writing the

IPv6-address-of-interest to hpot_requested_collection on honeypot-host's database. In Step 5, the honeypot-host periodically checks hpot_requested_collection, retrieves the IPv6-address-of-interest, and acknowledges the CN by writing the IP address to hpot_deployed_collection in honeypot-host database.

In Steps 6 & 7, the CN continually checks for records in hpot_deployed_collection and upon finding a record the CN waits for a fixed time (the address-holding-period) and unbinds the address retrieved from hpot_deployed_collection's record. In the mean-time, as shown in the Step 8, the honeypot-host checks for an available spare LXC container in the queue and invokes a shell-script to run lxc-attach commands to bind the IP-address on which honeypot is requested to the container's network interface, starts the Dionaea service to bind the honeypot service to the newly assigned IPv6 address and run the python web-server for launching the DionaeaFR front-end console. To speed up the address binding process, as specified in previous research effort [7] we disable Duplicate-Address-Detection (DAD) and use the concept of a forced Neighbor Advertisement (NA) thus advertising the new IPv6 address with the container's mac-address to the local router for quick address-acquisition. This completes the deployment of a honeypot tied to a desired IPv6 address and all future attacker traffic hits the honeypot-container.

The MT6D_view visualizations (Figures 6.1 and 6.4) show each MT6D host represented by its mac address and all the addresses relinquished by each MT6D host and incoming traffic on each of these spawned addresses by source IP address and Protocol. To simplify visualization of incoming traffic without getting cluttered by new addresses that are being

discarded by MT6D nodes and to observe the incoming traffic from the attacker perspective, the attacker_view visualization (as shown in Figure 6.5) view would show Source IP (potential

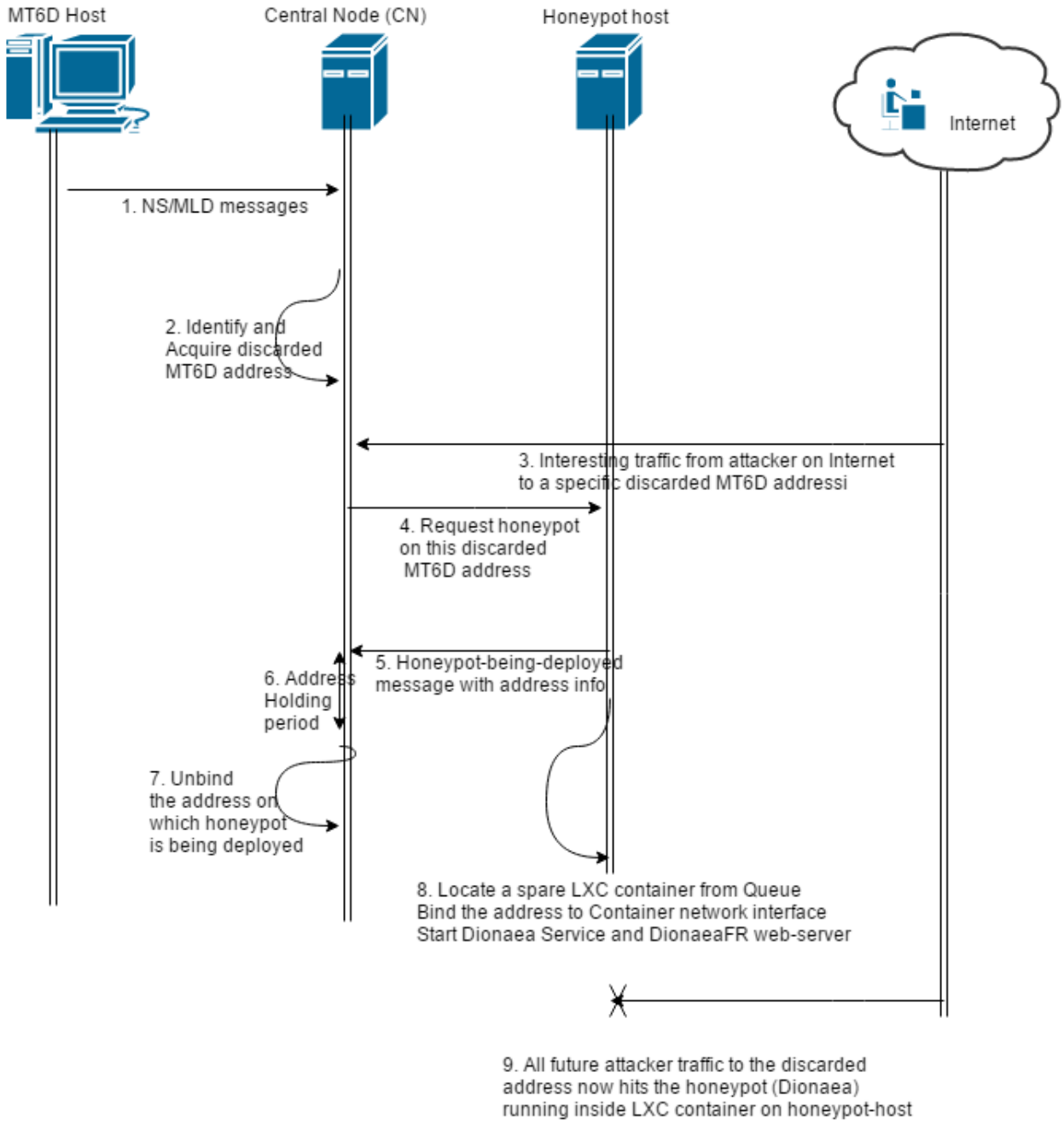

Figure 5.4: Flow Diagram - sequence of messages exchanged between various components

attacker) of incoming traffic at its root and the visualization would then branch from each of these source-IPs of incoming traffic to a MT6D mac address and then branching to MT6D IPv6 addresses and incoming traffic composition in terms of protocol and port numbers and corresponding packet-counts.

In this chapter we have discussed how we implemented various components of the proposed solution and explained how information flows between different modules. The next section presents evaluation of the solution in terms of our observations and results.

Chapter 6

Evaluation

Our solution at its heart involves efficiently spinning up honeypot-containers on the honeypot-host in addition to efficiently migrating the IP address from the Central-Node to a honeypot-container on the honeypot-host. To characterize such a solution, we will evaluate it in terms of interruption window observable by an attacker, as well as CPU and memory overheads. The interruption window observable to an attacker is a critical factor for the judging the feasibility of the solution, since the duration of interruption window may offer a cue to the attacker about migration of his/her traffic from one machine (the victim) to another(the honeypot). CPU and memory overhead analyses offer valuable insights in to scalability of the the solution i.e., the number of honeypot-containers that can be accommodated for a specific honeypot-host system(CPU and memory) configuration. We will also discuss a mechanism for MT6D nodes to leverage our solution. In this section we will explain the methodology of the testing and present our results.

6.1 Experiments and Results

6.1.1 Test Scenario

We first present a particular instance of potential attack-traffic hitting a discarded MT6D address and launch a honeypot-service bound to the specific address on an LXC container and supporting traffic visualizations. We then present results detailing the address-migration window times in terms of hold-packet-count and lost-packet-count for a ten honeypot-container deployment trial. We also present our observations from CPU and memory overhead analyses for single, five, and ten honeypot-container deployment trials. Hold-packet-count refers to the number of packets still hitting the Central-Node since the trigger that deploys the honeypot i.e., the first ICMP probe hitting the CN. Lost-Packet-Count refers to the number of packets lost as seen by the attacker during the migration of an IP address to the honeypot-container from the CN.

The experiment starts with an MT6D host with mac address 00:24:e8:42:c4:7a spinning off new addresses while dropping its old addresses. The CN listens to MT6D host's NS and MLD messages and parses them to identify and acquire the addresses that are being discarded by the MT6D host and visualizes the incoming traffic on these discarded addresses. Figure 6.1 shows the first level of a MT6D_view visualization with the MT6D host represented by its mac address,00:24:e8:42:c4:7a, at the center, with each newly generated MT6D address (e.g., 2001:468:c80:c111:8c23:9665:ae9a:c757) represented by the last 64 bytes (e.g., 8c23:9665:ae9a:c757). The /64 subnet prefix remains the same i.e., 2001:468:0c80:c111 for

all the MT6D address children nodes.

The next step involves sending ICMP echo probes(simulated attack traffic) every 50ms from the machine (at 2601:5c0:c000:773e:1565:8df2:1408:2077) on a remote Internet connection to a particular discarded MT6D address, 2001:468:c80:c111:8c23:9665:ae9a:c757, that is now assigned to the Central-Node (CN). The first ICMP echo request packet serves as a trigger (as per configuration in the traffic enumeration and analysis module) for the CN to flag this activity and request a honeypot be deployed on this specific address. The CN sends a 'honeypot-requested' message with the address information to the honeypot-host database. The honeypot-commissioning-module retrieves the 'honeypot-requested' message, acknowledges the CN with a 'honeypot-deployed' message, picks the IP address, and looks up the container-queue for an available honeypot-container. The honeypot-commissioning-module identifies an available container (in this case 'hpot-container1') and assigns the IPv6 address under attack, 2001:468:c80:c111:8c23:9665:ae9a:c757, to the local network interface of the honeypot-container and starts the Dionaea service and DionaeaFR python web-server on the container.

Figure 6.2 shows an ICMP ping (sending ICMP echo request probe every 50ms) results as seen by the attacker. Figure 6.3 shows the Wireshark capture running on the CN to identify the point when CN stops responding to the ICMP echo request packets incoming on the IPv6-address-of-interest indicating the unbinding. Figure 6.4 presents a second-level MT6D_view visualization of incoming attack traffic, showing an MT6D host with MAC address 00:24:e8:42:c4:7a receiving ICMP traffic of type 0x80 and a count of 24, on a

specific discarded address 8c23:9665:ae9a:c757 from an attacker at address 2601:5c0:c000 :773e:1565:8df2:1408:2077. Figure 6.5 presents attacker-view visualization depicting an attacker at address 2601:5c0:c000:773e:1565:8df2:1408:2077 sending traffic to MT6D host with MAC address 00:24:e8:42:c4:7a on a particular spawned MT6D address i.e., 8c23:9665:ae9a: c757 and the traffic composition is ICMP Type 0x80 and Code 0x00 (ICMP echo request) with a count of 24. MT6D_view and attacker_view present the visualization of the same traffic from two perspectives, the former shows the MT6D host at the center whereas the attacker_view puts the attacker-address at the center of the visualization and plots MT6D hosts that are getting hit from that specific attacker-address.

6.1.2 Observations on interruption window

From Figures 6.2, 6.3, 6.4 and 6.5 we can infer that the first 24 ICMP echo request (ICMP type=0x80 and code = 0) packets with sequence ids till '24', hit the CN and one ICMP echo request packet with sequence id '25' was lost, while packets (with sequence ids from '26', '27' ....) were routed to honeypot-container resulting in subsequent ICMP echo replies in figure 6.2 indicating successful IP address transition from CN to honeypot-container.

To verify the address-transition and Dionaea service binding we used a nmap NSE auth-spoof (generally used for testing authentication servers for malware infection) script as shown in Figure 6.6 to generate test-traffic on various ports targeting address of interest i.e., 2001:468:c80:c111:8c23:9665:ae9a:c757. Figure 6.7 shows the DionaeaFR console dis-

playing incoming connections for the address 2001:468:c80:c111:8c23:9665:ae9a:c757 on ports HTTPD, SMB, SIP, etc.

Figure 6.8 shows the hold-packet-counts and lost-packet-counts for the 10-honeypot deployment trial. We can also observe the IP address migration from the Central-Node to the honeypot-container on honeypot-host involves loss of one packet or none, from the perspective of an attacker probing a discarded MT6D address with an ICMP echo request packet every 50ms. From the Wireshark packet captures, we attribute this packet-loss i.e., unresponsive ICMP echo requests, to the router on Virginia Tech production network either routing the packet to the Central-Node (CN) even though the IP address has been migrated to the honeypot-container or the router forwarding the packet to honeypot-container while the container is still in the process of binding the address to its network interface. In either cases, even though the packet makes it to CN or the honeypot-container depending on routing-table entry, neither of them can respond with ICMP echo reply as the address is not active on their respective network interfaces.

6.1.3 Observations on CPU and memory overheads

Figures 6.9, 6.10 & 6.11 show CPU and memory (RAM) loading characteristics for three independent trials of single honeypot-container, five honeypot-container and ten honeypot-container deployment. CPU time comprises of Usr-CPU and Sys-CPU. Usr-CPU is CPU time spent in user-mode outside Kernel code and Sys-CPU is the CPU time spent in the kernel

within the process handling system calls and other kernel-space events. We observed that the trend of memory consumption with honeypot-service being deployed on each container is proportionally-linear (˜125MB for each honeypot-container launch) while CPU consumption peaks temporarily and then remains constant. The baseline free-memory consumption for the honeypot-host after a clean reboot was observed to be around 3025 MegaBytes. The baseline CPU consumption was observed to be nominal. From the observed memory and CPU consumption trend we substantiate the virtue of using the hot-spare LXC containers approach as the containers waiting as spares place nominal load on system resources and only request resources, Memory and CPU, on-demand when we invoke Dionaea and DionaeaFR management-server services on the container.

Figure 6.9 shows memory and CPU consumption for a single honeypot-container experiment. From the origin until the point 'A' on the free-memory curve represents the memory consumption on machine serving as baseline. At epoch time ˜1443048570, the LXC container 'hpot-container1' is cloned from honeypot-LXC-template and the 'hpot-container1' is started and lies waiting as hot-spare in the queue. We can observe a drop in free-memory at point 'A'. At epoch time ˜1443048801, the honeypot-commissioning-module runs a script to bind the requested IPv6 address, start the Dionaea service and, start the DionaeaFR python web-server on container 'hpot-container1'. This step brings up Dionaea service bound to desired IP address on 'hpot-container1'. We can observe a significant drop (˜125 MB) in free-memory at point 'B' on the curve in response to Dionaea and DionaeaFR service invocation. We can also observe the CPU (Usr and Sys CPU) consumption spike to a peak value

at the time-instant corresponding to point 'B' i.e., at the launch of honeypot-services on the virtual-container, and subsequently remain constant.

Figure 6.10 depicts the CPU and memory consumption trend for a five honeypot-container trial. From origin until the point 'A' on the free-memory curve represents the baseline memory consumption. At epoch time ~1443023165 five containers ('hpot-container1', 'hpot-container2'...'hpot-container5') are cloned from honeypot-LXC-template, started and lie waiting as hot-spares. At epoch times corresponding to points 'B', 'C', 'D', 'E', and 'F' on the free-memory curve represent the drops in available-memory corresponding to instants of binding requested IP address and firing the Dionaea service and DionaeaFR web-server on each LXC honeypot-container. We can also observe the CPU consumption spike to peak value at the time-instants corresponding to points 'B', 'C', 'D', 'E', and 'F' i.e., at the launch of honeypot-services on the virtual-containers, and subsequently lie constant.

Figure 6.11 shows similar memory consumption behavior for a ten honeypot-deployment trial, at epoch time corresponding to point 'A' i.e., ~1443029196, ten containers ('hpot-container1', 'hpot-container2'...'hpot-container10') are cloned, started and lie waiting in hot-spares queue. Epoch times corresponding to points 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', and 'K' on the free-memory curve correspond to IP address binding and invocation of Dionaea services. We can also observe the CPU consumption spike to peak value at the time-instants corresponding to points 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', and 'K' i.e., at the launch of honeypot-services on the virtual-containers, and subsequently remain constant.

6.1.4 How can a MT6D node leverage our solution

The MT6D nodes can consume the CN's intelligence on attacker-activity using the solution's Mongo database API. The JSON object responsible for Figure 6.5, the attacker-view visualization, can be queried from the CN's database by any MT6D host to detect a trailing attacker by identifying all the attacker-address nodes that have its (MT6D host's) MAC address as child node. In this case the MT6D host with the MAC address 00:24:e8:42:c4:7a can analyze the JSON object responsible for the Figure 6.5 to understand its MAC address is a child node for the attacker address 2601:5c0:c000:773e:1565:8df2:1408:2077 and among its relinquished addresses particularly MT6D addresses ending with 8c23:9665:ae9a:c757 and f8a1:cc15:e13c:7d9d were hit with ICMP echo request traffic. Based on the attack traffic-composition, the MT6D host can communicate to its MT6D partner and change their scheme parameters (e.g., stronger secret-key and faster hopping-interval) to evade any trailing attackers.

With this solution in place, nodes participating in MT6D nodes can uncover any suspicious activity on their relinquished addresses and be able to use it in fine-tuning the scheme parameters. The solution's traffic enumeration and honeypot deployment capabilities offer insights in to attacker methods. In this chapter we evaluated our solution in terms of interruption window observable to an attacker, CPU and memory overhead analyses in addition to discussing a mechanism to offer gathered intelligence on relinquished addresses to MT6D nodes through an JSON API. In the next chapter we will conclude our findings and provide directions for future work.

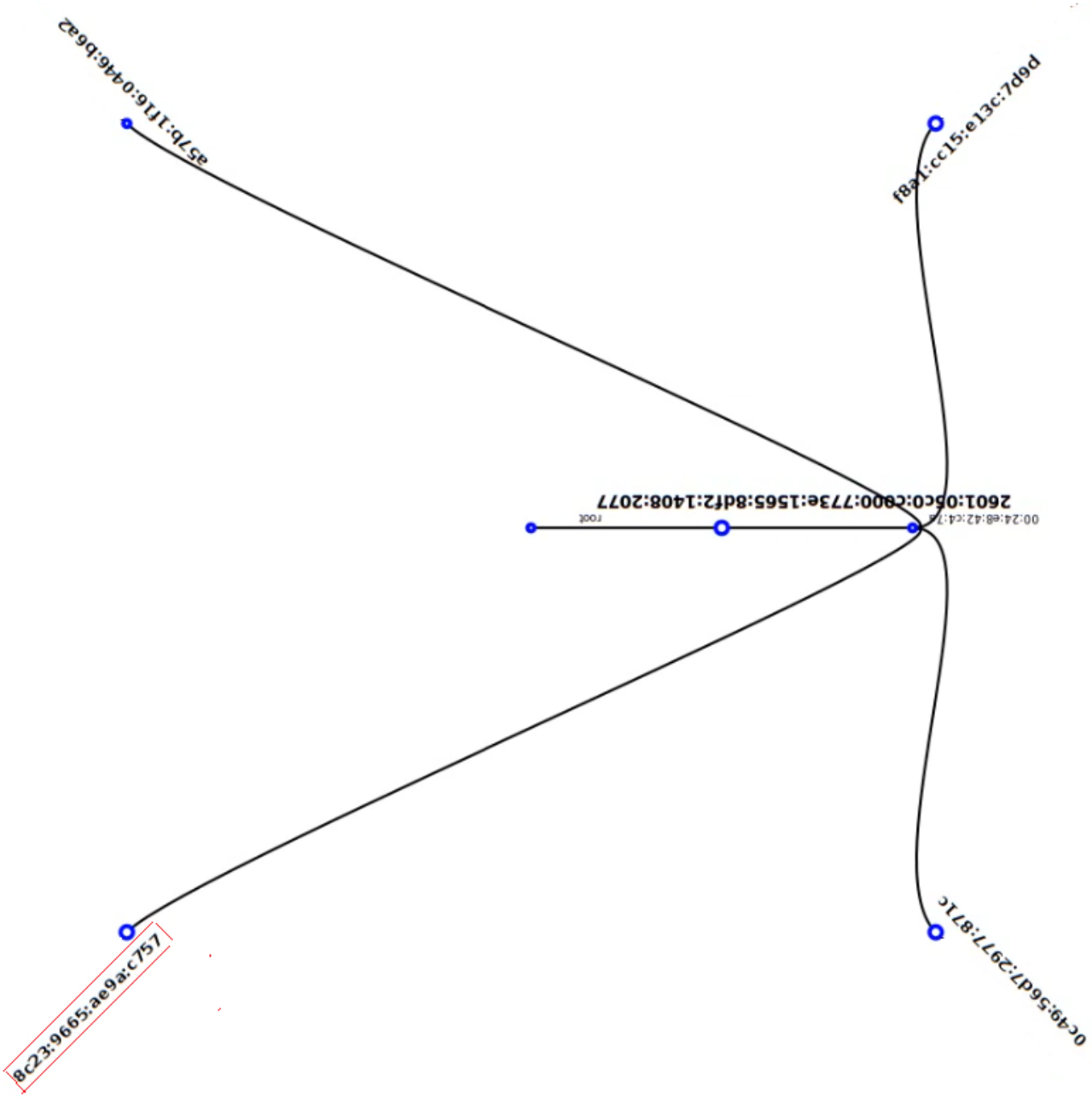

Figure 6.1: MT6D view depicting addresses spawned by a MT6D host

```
16 bytes from 2001:468:c80:c111:8c23:9665:ae9a:c757, icmp_seq=19 hlim=52 time=25.866 ms
16 bytes from 2001:468:c80:c111:8c23:9665:ae9a:c757, icmp_seq=20 hlim=52 time=28.564 ms
16 bytes from 2001:468:c80:c111:8c23:9665:ae9a:c757, icmp_seq=21 hlim=52 time=30.185 ms
16 bytes from 2001:468:c80:c111:8c23:9665:ae9a:c757, icmp_seq=22 hlim=52 time=25.166 ms
16 bytes from 2001:468:c80:c111:8c23:9665:ae9a:c757, icmp_seq=23 hlim=52 time=24.148 ms
16 bytes from 2001:468:c80:c111:8c23:9665:ae9a:c757, icmp_seq=24 hlim=52 time=34.412 ms
16 bytes from 2001:468:c80:c111:8c23:9665:ae9a:c757, icmp_seq=26 hlim=52 time=31.686 ms
16 bytes from 2001:468:c80:c111:8c23:9665:ae9a:c757, icmp_seq=27 hlim=52 time=32.178 ms
16 bytes from 2001:468:c80:c111:8c23:9665:ae9a:c757, icmp_seq=28 hlim=52 time=32.715 ms
16 bytes from 2001:468:c80:c111:8c23:9665:ae9a:c757, icmp_seq=30 hlim=52 time=23.711 ms
16 bytes from 2001:468:c80:c111:8c23:9665:ae9a:c757, icmp_seq=31 hlim=52 time=30.459 ms
16 bytes from 2001:468:c80:c111:8c23:9665:ae9a:c757, icmp_seq=32 hlim=52 time=24.892 ms
16 bytes from 2001:468:c80:c111:8c23:9665:ae9a:c757, icmp_seq=33 hlim=52 time=24.720 ms
16 bytes from 2001:468:c80:c111:8c23:9665:ae9a:c757, icmp_seq=34 hlim=52 time=26.199 ms
```

Figure 6.2: ICMP pingflood traffic from simulated attacker on a remote Internet connection at 50ms interval

Figure 6.3: Wireshark capture showing packets hitting CN before the IP address migration

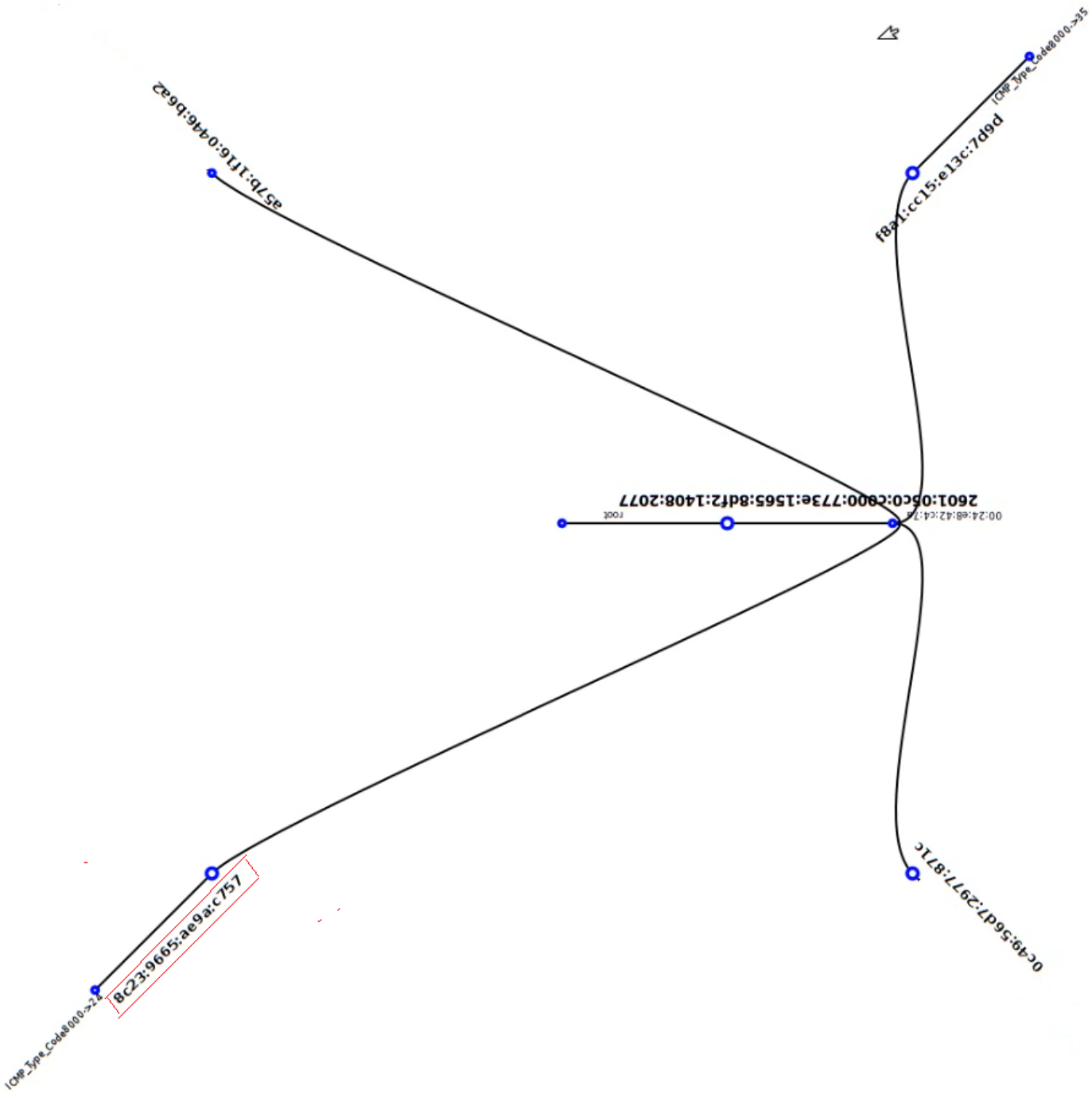

Figure 6.4: MT6D view of incoming traffic hitting the CN visualized

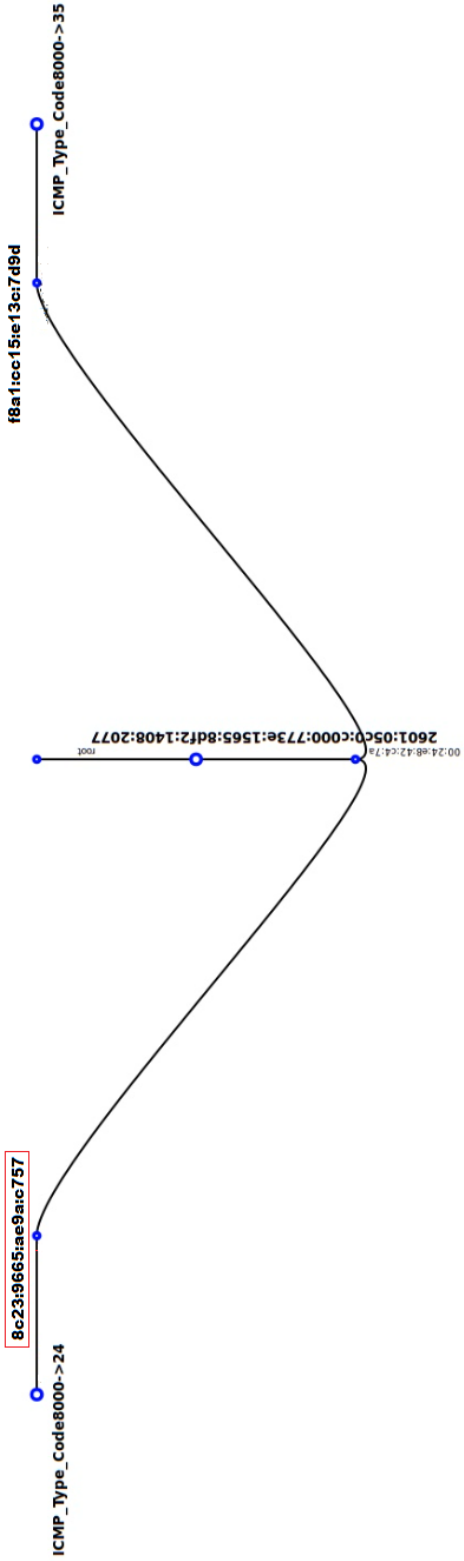

Figure 6.5: Attacker-view of incoming traffic hitting the CN visualized

```
pc1@pc1-OptiPlex-960:~$
pc1@pc1-OptiPlex-960:~$ sudo nmap -6 -sV --script=auth-spoof 2001:468:c80:c111:8c23:9665:ae9a:c757

Starting Nmap 6.40 ( http://nmap.org ) at 2015-10-02 16:51 EDT
```

Figure 6.6: Nmap script to send test traffic on various services to a deployed honeypot

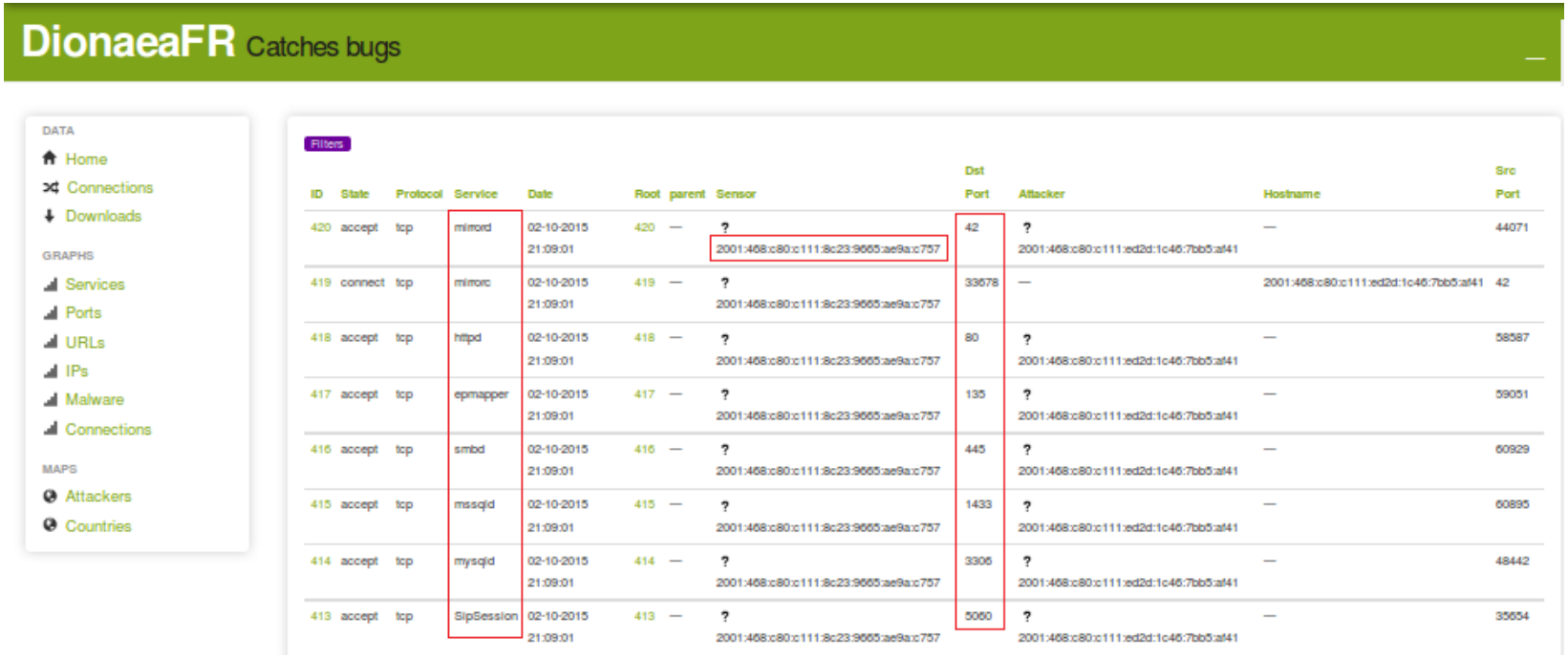

Figure 6.7: DionaeaFR console showing deployed honeypot connection statistics

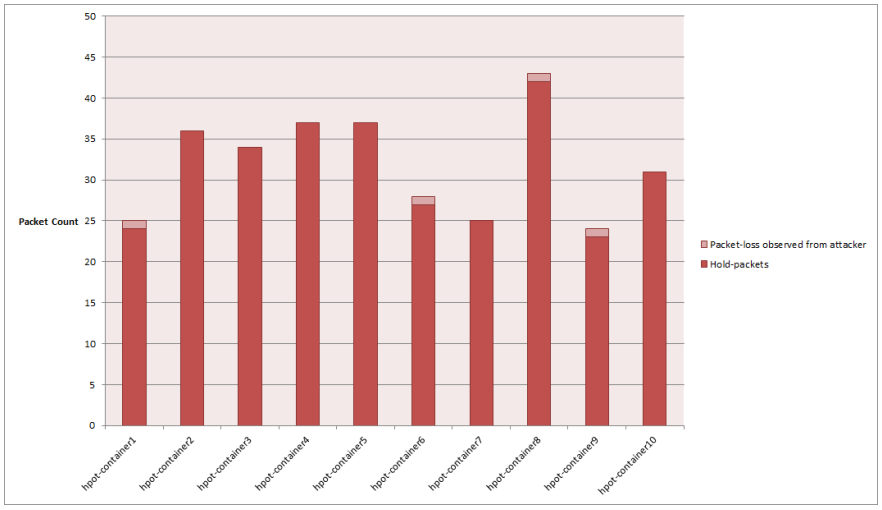

Figure 6.8: Chart giving out Hold-packet counts and packet-loss observed by an attacker during one of the 10-honeypot deployment trials involving sending an ICMP echo request every 50ms i.e., rate of 20 packets/second

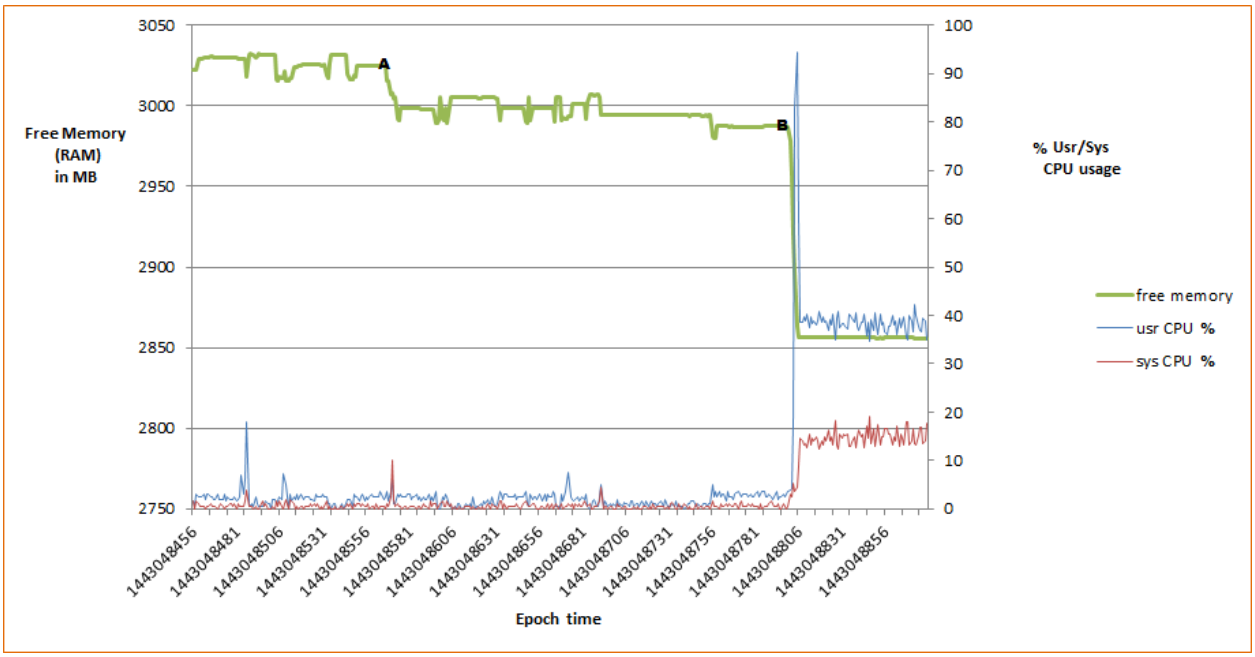

Figure 6.9: Memory and CPU loading characteristics for the scenario of 1 honeypot-container hotspare and subsequent deployment

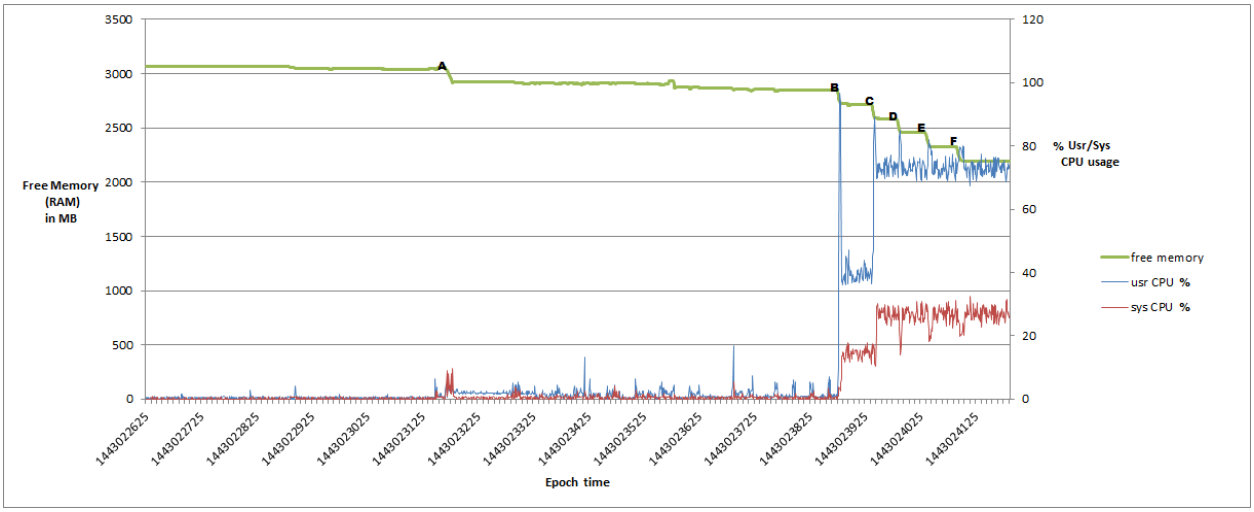

Figure 6.10: Memory and CPU loading characteristics for the scenario of 5 honeypot-container hotspares and subsequent deployment

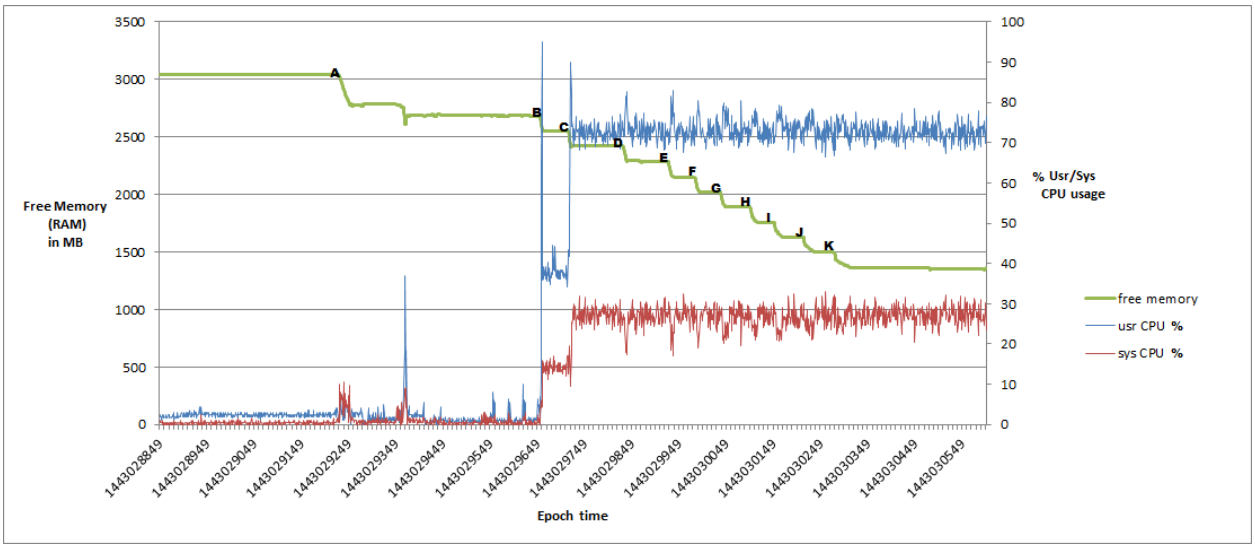

Figure 6.11: Memory and CPU loading characteristics for the scenario of 10 honeypot-container hotspares and subsequent deployment

Chapter 7

Conclusion and Future Work

We started by providing background on IPv6, Moving Target IPv6 Defense (MT6D) and it's extant threat vectors. In this work we developed a solution using darknet and honeypot capabilities to strengthen defenses of MT6D networks. By analyzing the activity on relinquished addresses, the proposed solution delivers valuable clues on attack traffic composition and attacker's methods for further analysis. We have reported our results and in this section we conclude our findings and observations.

7.1 Conclusion

The implemented solution offers two visualization views, MT6D-view and attacker-view. MT6D-view displays nodes representing discarded addresses from all MT6D hosts in a local network and their corresponding incoming traffic. Attacker-view displays nodes corre-

sponding to attackers addresses sending traffic and those specific MT6D hosts and discarded addresses that are receiving attack traffic. As we can observe from figures 6.4 and 6.5 attacker_view simplifies the visualization over MT6D_view by the virtue of ignoring MT6D nodes with no interesting traffic. MT6D nodes with no incoming traffic no longer appear on the visualization, thus resulting in less-cluttered visualization of interesting activity. This paves way for visualizing suspicious activity on large MT6D networks without getting burdened by the periodic address-hopping activity.

The solution with database-integration also allows any MT6D node to query for traffic activity on its discarded MT6D addresses as a JSON object. This can be integrated as a feedback mechanism into work that our fellow researchers Morrell et al., are doing in the area of MT6D Client-Server stack in terms of an MT6D Server looking up suspicious activity on discarded addresses and communicating new MT6D scheme parameters such as longer secret-key or faster address hopping interval to its MT6D client[33].

We evaluated the capability of dynamically deploying a LXC container-based honeypot upon detecting suspicious activity on discarded MT6D addresses. The evaluation included interruption observable to an external attacker, CPU and memory overhead analyses to demonstrate the feasibility and scalablility of such a solution. From our tests, we observed that the interruption observable to an attacker is one packet or none during the migration of the IPv6 address from the Central-Node to the honeypot-container. We also observed that the trend of the memory consumption with number of honeypot-containers being deployed to be proportionally-linear while CPU consumption peaks to a value and stays constant during

the launch of the honeypot-service on each container. This supports the argument that the presented solution in this effort is scalable provided that the memory and compute resources can be catered by the underlying honeypot-host hardware.

Previous work done by Morrell et al.[34] showed that a server can successfully bind up to 60000 randomly generated addresses, with each bound address communicating to an individual client. Based on these observations, the Central-Node (CN) should be able to bind and listen on a large number of discarded MT6D addresses. We would also like to point out the solution's honeypot-deployment ability is unaffected by the number of MT6D nodes on network, as the incoming traffic enumeration and analysis is delegated to the Central-Node(CN), and the honeypot-host resources are engaged only when the CN matches incoming traffic with a configured attack-traffic signature.

7.2 Future Work

Future work should include CPU and memory overhead analyses of the solution under complex attack scenarios and evaluation of the solution's performance (IP-address migration times and packet-loss observable to attacker) with the honeypot-host sitting in a segregated environment like a DMZ. It is also important to perform overhead analysis on network devices, as such a solution grabbing discarded addresses on a complex MT6D network would place overhead on the router in terms of persistent routing table entries. Future work can also include enhancing the solution with a new feature that would reclaim active honeypot-

containers based on no or low-priority incoming attacker traffic hitting a deployed honeypot-container.

Appendices

Please note that the code was edited to fit page boundaries and may not be in executable state.

Appendix A

Python script on CentralNode

A.1 CentralNode.py

```
#!/usr/bin/python

from pyroute2 import IPRoute

from scapy.all import *

import pcapy

from impacket.ImpactDecoder import *

import re, json

from pymongo import *

import ipaddr, time, threading, ast

#all byte values in HEX
```

```
TYPE_MLD_HEX = '8f'

TYPE_NS_HEX = '87'

INCLUDE_CODE='03'

EXCLUDE_CODE='04'

ICMPV6_CODE = '3a'

TCP_CODE = '06'

UDP_CODE = '17'

ICMP_CODE = '80'

TCP_CODE = '81'

ROUTER_INTERNAL_IF = 'fe80::2d0:1ff:fea6:c800'

LOCAL_SUBNET = '2001:0468:0c80'

SELF_MAC = 'b8:ac:6f:8b:35:fb'

ROUTER_MAC = '00:d0:01:a6:c8:00'

CONT1_MAC = '00:16:3e:65:26:76'

CONT2_MAC = '00:16:3e:75:01:14'

CONT3_MAC = '00:16:3e:c6:55:61'

CONT4_MAC = '00:16:3e:e3:7d:37'

CONT5_MAC = '00:16:3e:67:db:e3'

CONT6_MAC = '00:16:3e:9f:c2:dc'

CONT7_MAC = '00:16:3e:bf:42:d3'

CONT8_MAC = '00:16:3e:13:4f:20'
```

```
CONT9_MAC = '00:16:3e:bc:78:75'

CONT10_MAC = '00:16:3e:76:5d:07'

HONEYPOT_DB = '[2001:468:c80:c111:dbdb:dbdb:dbdb:dbdb]'

flag = 0

assigned = dict() #dictionary for acquired MT6D-discarded-addresses

neighbor = dict() #dictionary for IP <- -> MAC association

neighbor_backup = dict() #back-up data-structure for 'neighbor'

ipr = IPRoute()

pcapy.findalldevs()

max_bytes = 1024

promiscuous = False

read_timeout = 100 # in milliseconds

pc = pcapy.open_live("eth0", max_bytes, \

promiscuous, read_timeout)

pc.setfilter('ip6')

dev = ipr.link_lookup(ifname='eth0')[0]

#Instantiating MongoClient for local i.e., CentralNode database

client = MongoClient('localhost', 27017)

db = client.omni_database

neighbor_collection = db.neighbor_collection

assigned_collection = db.assigned_collection
```

```
mt6dView_collection = db.mt6dView_collection

attackerView_collection = db.attackerView_collection

#Instantiating MongoClient for remote i.e., Honeypot-host database

remoteClient = MongoClient(HONEYPOT_DB, 27017)

remoteDb = remoteClient.hpot

hpot_requested = remoteDb.hpot_requested

hpot_deployed = remoteDb.hpot_deployed

hpot = {'ip' : []}

already_unbinded = []

countt = 0

#Function that performs real-time Traffic-Parsing,

#Enumeration and Address acquisition operations

def packet_processing(hdr, data):

packet = EthDecoder().decode(data)

str_pkt= str(packet)

src_mac = str_pkt[7:24].split(' ')[0]

regex = re.compile('Source address: (.*)')

m= regex.search(str_pkt)

if m:

src_addr = str(m.group(1)).lower()

src_addr = ipaddr.IPv6Address(src_addr).exploded
```

```
regex = re.compile('Destination address: (.*)')

m= regex.search(str_pkt)

if m:

dest_addr = str(m.group(1)).lower()

dest_addr = ipaddr.IPv6Address(dest_addr).exploded

regex = re.compile('Next header: (.*)')

m= regex.search(str_pkt)

if m:

nxt_hdr = str(m.group(1))

regex = re.compile('.*-> (.*)')

m= regex.search(str(packet.child().child()))

if m:

dst_port = str(m.group(1))

payload = ''.join('%02x' % ord(b) \

for b in packet.child().get_data_as_string())

code = payload[0:2]

#Checking for Neighbor Solicit packets

if (code == TYPE_NS_HEX and (src_mac!=SELF_MAC) \

and (src_mac!=ROUTER_MAC)):

line = payload[16:48]

arr = [line[i:i+4] for i in range(0, len(line), 4)]
```

```
addrezz = ':'.join(arr)

if (addrezz not in neighbor) and ('fe80' not in addrezz):

neighbor[addrezz]=src_mac

neighbor_backup[addrezz]=src_mac

elif (code == ICMPV6_CODE and (src_mac!=SELF_MAC)):

#Checking for MLD Packets and their Include and Exclude labels

if payload[16:18] == TYPE_MLD_HEX:

index = 32

while True:

if payload[index:index+2]== EXCLUDE_CODE:

break

elif payload[index:index+2]== INCLUDE_CODE:

line = payload[index+8:index+8+32]

arr = [line[i:i+4] for i in range(0, len(line), 4)]

addrezz = ':'.join(arr)

addrezz = addrezz[index:index+8]

if index+40 < len(payload):

index=index+40

else:

break

regex = re.compile('[a-z0-9:]{32}'+addrezz+'$')
```

```
#Logic for finding the full IPv6 address

#from observed prefix in MLD using

#IP <--> MAC associations stored in 'neighbor'

new = [x for x in neighbor.keys() if regex.match(x)]

found_addr = ''

try:

if len(new)>0:

for item in new:

if neighbor[item]==src_mac:

found_addr = item

del neighbor[found_addr]

break

#MAC address corroboration logic

if found_addr== '':

print 'Exception - MAC address Mismatch'

break

#Store the discarded-MT6D address in

#'found_addr' in 'assigned' dictionary

if found_addr not in assigned.keys():

assigned[found_addr]={}

#Acquire the IP address
```

```
#Bind it to the local network interface

ipr.addr('add',dev,address=found_addr,mask=64)

print addrezz+'Prefix found in neighbor_list and \

about to advertize : '+found_addr,

#send a forced NA to the local router

#for a quicker routing-table update

sendNA(SELF_MAC)

#Update the database collections with the new dictionaries

try:

neighbor_collection.remove({})

assigned_collection.remove({})

Temp1_insert_id = neighbor_collection.insert_one(neighbor_backup)\

.inserted_id

Temp2_insert_id = assigned_collection.insert_one(assigned)\

.inserted_id

except:

'Exception in connecting to the Database'

else:

break

except:

print 'Exception in assigning this address'+found_addr
```

```
#Attack Traffic enumeration part for TCP/UDP

#and ICMP packets on acquired addresses coming

#from non-local subnets

elif nxt_hdr == TCP_CODE and  (LOCAL_SUBNET not in src_addr):

port = 'TCP_'+dst_port

if (dest_addr in assigned.keys()):

if src_addr not in assigned[dest_addr].keys():

assigned[dest_addr][src_addr]={}

if port in assigned[dest_addr][src_addr]:

assigned[dest_addr][src_addr][port]+=1

else:

assigned[dest_addr][src_addr][port]=1

elif nxt_hdr == UDP_CODE and  \

(LOCAL_SUBNET not in src_addr):

port = 'UDP_'+dst_port

if (dest_addr in assigned.keys()):

if src_addr not in assigned[dest_addr].keys():

assigned[dest_addr][src_addr]={}

if port in assigned[dest_addr][src_addr]:

assigned[dest_addr][src_addr][port]+=1

else:
```

```
assigned[dest_addr][src_addr][port]=1

elif code==ICMP_CODE1 or code==ICMP_CODE2 and  \

(LOCAL_SUBNET not in src_addr):

port = 'ICMP_Type_Code'+payload[0:4]

if (dest_addr in assigned.keys()):

if src_addr not in assigned[dest_addr].keys():

assigned[dest_addr][src_addr]={}

if port in assigned[dest_addr][src_addr]:

assigned[dest_addr][src_addr][port]+=1

else:

assigned[dest_addr][src_addr][port]=1

#Function that converts native python dictionaries

#in to JSON parent-child objects for D3.js visualization

def dictToJSONConverter_dbUpdate():

threading.Timer(1, dictToJSONConverter_dbUpdate).start ()

#MT6D View JSON object preparation

try:

for item in assigned.keys():

if item =='_id':

del assigned[item]

jsonList=[]
```

```
count1=count2=count3=0

for item1 in assigned.keys():

pos1=pos2=pos3=0

count=count1-1

flag=0

while count>=0:

if jsonList[count]['name']==neighbor_backup[item1]:

flag=1

pos1=count

break

else:

count=count-1

if flag==0:

jsonList.append({})

jsonList[count1]['name']=neighbor_backup[item1]

jsonList[count1]['parent']='root'

jsonList[count1]['children']=[]

jsonList[count1]['children'].append({})

jsonList[count1]['children'][0]['name']=item1

jsonList[count1]['children'][0]['parent']=neighbor_backup[item1]

jsonList[count1]['children'][0]['children']=[]
```

```
count2=0

for item2 in assigned[item1].keys():

jsonList[count1]['children'][0]['children'].append({})

jsonList[count1]['children'][0]['children'][count2]['name']=item2

jsonList[count1]['children'][0]['children'][count2]['parent']=item1

jsonList[count1]['children'][0]['children'][count2]['children']=[]

count2=count2+1

count3=0

for item3 in assigned[item1][item2].keys():

jsonDict={}

jsonDict['name']=item3+'->'+str(assigned[item1][item2][item3])

jsonDict['parent']=item2

jsonList[count1]['children'][0]['children'][count2-1]['children']\

        .append(jsonDict)

count1=count1+1

elif flag==1:

jsonList[pos1]['children'].append({})

pos2= len(jsonList[pos1]['children'])-1

jsonList[pos1]['children'][pos2]['name']=item1

jsonList[pos1]['children'][pos2]['parent']=neighbor_backup[item1]

jsonList[pos1]['children'][pos2]['children']=[]
```

```
for item2 in assigned[item1].keys():

jsonList[pos1]['children'][pos2]['children'].append({})

pos3=len(jsonList[pos1]['children'][pos2]['children'])-1

jsonList[pos1]['children'][pos2]['children'][pos3]['name']=item2

jsonList[pos1]['children'][pos2]['children'][pos3]['parent']=item1

jsonList[pos1]['children'][pos2]['children'][pos3]['children']=[]

for item3 in assigned[item1][item2].keys():

jsonDict={}

jsonDict['name']=item3+'->'+str(assigned[item1][item2][item3])

jsonDict['parent']=item2

jsonList[pos1]['children'][pos2]['children'][pos3]['children']\

        .append(jsonDict)

pubs = { "name" : "root", "parent" : "null", "children" :\

ast.literal_eval(json.dumps(jsonList)) }

mt6dView_collection.remove({})

Temp3_insert_id = mt6dView_collection.insert_one(pubs).inserted_id

dict = {}

for item in assigned.keys():

for key,value in assigned[item].iteritems():

if key not in dict.keys():

dict[key] = {neighbor_backup[item] : {item : assigned[item][key]}}
```

```
else:
if neighbor_backup[item] in dict[key].keys():
dict[key][neighbor_backup[item]].update({item : assigned[item][key]})
else:
dict[key].update({neighbor_backup[item] : \
{item : assigned[item][key]}})
#Attacker View JSON object preparation
jsonDict = {"name" : "root", "parent" : "null", "children" : []}
pos1=0;pos2=0;pos3=0
for item in dict.keys():
pos2=0
jsonDict["children"].\
append({"name" : item, "parent" : "root", "children" : [] })
for item2 in dict[item].keys():
pos3=0
jsonDict["children"][pos1]["children"].\
append({"name" : item2, "parent" : item, "children" : [] })
for item3 in dict[item][item2].keys():
jsonDict["children"][pos1]["children"][pos2]["children"]\
.append({"name" : item3, \
"parent" : item2, "children" : [] })
```

```
for item4 in dict[item][item2][item3].keys():

jsonDict["children"][pos1]["children"][pos2]["children"][pos3]["children"]\

.append({"name" : item4 + '->' \

+str(dict[item][item2][item3][item4]), "parent" : item3} )

pos3=pos3+1

pos2=pos2+1

pos1=pos1+1

attackerView_collection.remove({})

Temp4_insert_id = attackerView_collection.insert_one(jsonDict).\

inserted_id

hpot_requested.remove({})

for item in jsonDict["children"]:

for item2 in item["children"]:

for item3 in item2["children"]:

for item4 in item3["children"]:

#To configure ICMP ping traffic on a discarded address

#as trigger for deploying a honeypot

# on that discarded address

if 'ICMP' in item4['name']:

if (item4['parent'] not in hpot['ip']):

hpot['ip'].append(item4['parent'])
```

```
#Requesting honeypot to be deployed on \

#a specific discarded-MT6D address

Temp5_insert_id = hpot_requested.insert_one(hpot).\

inserted_id

except:

pass

#Invoking function dictToJSONConverter_dbUpdate

dictToJSONConverter_dbUpdate()

#forcedNeighborAdvtThread  to send a series of

#forced Neighbor Advertisement packets to router

#for quicker routing table update

def fakeNeighborAdvt(ip, countl):

i = 0

while i < 50:

ip = str(ip)

print ip

if countl == 0:

sendNA(ip,CONT1_MAC)

elif countl == 1:

sendNA(ip,CONT2_MAC)

elif countl == 2:
```

```
sendNA(ip,CONT3_MAC)

elif countl == 3:

sendNA(ip,CONT4_MAC)

elif countl == 4:

sendNA(ip,CONT5_MAC)

elif countl == 5:

sendNA(ip,CONT6_MAC)

elif countl == 6:

sendNA(ip,CONT7_MAC)

elif countl == 7:

sendNA(ip,CONT8_MAC)

elif countl == 8:

sendNA(ip,CONT9_MAC)

elif countl == 9:

sendNA(ip,CONT10_MAC)

i=i+1

#Scapy code for crafting a IPv6 Neighbor advertisement

#with desired link local address and ip address to the

def sendNA(ip,CONT_MAC):

send(IPv6(dst=ROUTER_INTERNAL_IF,src=ip)/\

ICMPv6ND_NA(type=136,tgt=ip,O=1,R=0)/\
```

```
ICMPv6NDOptDstLLAddr(lladdr=CONT_MAC))

#Function checkForDeployedHpots for looking up

#IPv6 addresses that are being processed by honeypot-host

def checkForDeployedHpots():

global countt

global already_unbinded

threading.Timer(0.1, checkForDeployedHpots).start ()

cursor = hpot_deployed.find({},{'_id' : 0})

for doc in cursor:

if (len(already_unbinded) == 0 or \

str(doc['ip']).strip() != already_unbinded[-1].strip()):

hpot_deployed.remove({})

sleep(1)#Address-Holding-Period

if len(already_unbinded) > 0:

print already_unbinded[-1]

try:

print doc['ip']

except:

pass

already_unbinded.append(str(doc['ip']))

#Invoking forcedNeighborAdvtThread  to send a series of
```

```
#forced Neighbor Advertisement packets to router

#for quicker routing table update

forcedNeighborAdvtThread  = threading.\

Thread(target=fakeNeighborAdvt, args=[doc['ip'], countt])

forcedNeighborAdvtThread .start()

try:

ipr.addr('delete',dev,address=doc['ip'],mask=64)

countt=countt+1

except:

pass

#Invoking checkForDeployedHpots function

checkForDeployedHpots()

packet_limit=-1

#Inovking packet_processing function

#packet_limt = -1 runs the packet_processing

#function in infinite loop

pc.loop(packet_limit, packet_processing)
```

Appendix B

Shell and Python scripts for honeypot-host

B.1 createHpots.sh

```
#! /bin/bash

name=$1

ip_internal=$2

sudo -S lxc-clone -o hpot -n $name

sudo lxc-start -d -n $name

sudo lxc-attach -q -n $name -- /sbin/route -A\

 inet6 add 2001:468:c80:c111::/64 dev eth0
```

```
sudo lxc-attach -n $name -- /sbin/ifconfig > $name.txt

sudo lxc-attach -q -n $name -- /sbin/ip addr add $ip_internal/24 dev eth0
```

B.2 deployHpots.sh

```
#! /bin/bash

name=$1

ip=$2

sudo lxc-attach -n $name -- /sbin/ip -6 \

addr add $ip/64 dev eth0

sudo lxc-attach -n $name -- /sbin/ifconfig

sudo lxc-attach -n $name -- /usr/bin/dionaea -c \

/etc/dionaea/dionaea.conf -w /var/dionaea \

-u nobody -g nogroup -D

sudo lxc-attach -q -n $name -- /bin/sh -c \

"cd /opt && /usr/bin/python /opt/DionaeaFR/manage.py \

runserver 0.0.0.0:8000"
```

B.3 HoneypotHost.py

```
#!/usr/bin/python
```

```
from time import *

import threading

import subprocess

from pymongo import*

import re, mmap

#Instantiating MongoClient and Collections

client = MongoClient('10.0.3.217', 27017)

db = client.hpot

hpot_requested = db.hpot_requested

hpot_deployed = db.hpot_deployed

hpot_being_deployed = db.hpot_being_deployed

ui_panel = db.ui_panel

hpot_macs = db.hpot_macs

hpotAvlbl = []

hpotAvlbl = ['hpot_img'+str(x) for x in range(1,11) ]

being_deployed = []

ui_panel_arr = []

deployed = {}

#createSpares function that invokes a shell script

#to create desired number of LXC containers

#and gather IPv4 addresses that were assigned to each container
```

```
#the createHpot.sh starts the contianers

#and these containers lie waiting as hot-spares

def createSpares():

ip_internal='10.0.3.'+str(100+int(re.findall(r'\d+', item)[0]))

for hpot_name in hpotAvlbl:

#Invoking createHpot.sh shell script

subprocess.check_call(['/home/pc5/Desktop/createHpot.sh',hpot_name,ip_internal])

#Inovking createSpares Function

createSpares()

#checkHpotRequests keeps checking the honeypot-host database's

#hpot_requested collection and invokes deployHpot.sh shell script

#to invoke Dionaea and DionaeaFR services on the lxc-container

def checkHpotRequests():

threading.Timer(0.1, checkHpotRequests).start()

cursor = hpot_requested.find({},{'_id':0})

for doc in cursor:

for ip in doc['ip']:

if ip not in being_deployed:

print 'got this ip as request for hpot'+ip

being_deployed.append(ip)

#hpotName = pop the first element of the hpotNames
```

```
try:
name = hpotAvlbl.pop(0)
print 'deploying '+name
deployed = {'ip' : ip}
hpot_deployed.remove({})
hpot_deployed.insert_one(deployed)
#Invoking deployHpot.sh shell script
subprocess.check_call(['/home/pc5/Desktop/deployHpot.sh',name, ip])
#extracting the internal IPv4 address from the text-file dump
#and populate a dictionary with hpot-container name and its IPv4 address
#to be able to launch DionaeaFR console from that IPv4 address
with open('/home/pc5/Desktop/'+name+'.txt','r+') as f:
data = mmap.mmap(f.fileno(),0)
mo=re.search('10.0.3.(.{1,3})', data)
if mo:
new_dict = {name : [ip, '10.0.3.'+mo.group(1).strip()]}
ui_panel_arr.append(new_dict)
print ui_panel_arr
ui_panel.remove({})
ui_panel.insert_one({'root' : ui_panel_arr})
except:
```

```
#Running out of honeypot-container hot-spares

print 'no hpot-containers available'

#Invoking function checkHpotRequests for \

continually checking for honeypot-deployment requests from CentralNode

checkHpotRequests()
```

Appendix C

Webserver and visualization code

C.1 server.js

```
//server.js: web-server in Node.js

var path = require('path'),

express = require('express'),

app = express(),

bodyParser = require('body-parser'),

request = require('request');

//Enabling bodyParser module for parsing POST requests

app.use(bodyParser.urlencoded({ extended : true}));

app.use(bodyParser.json());
```

```
//Setting directory to enable server find necessary files

app.use(express.static(path.join(__dirname, 'public')));

//Importing routes file and registering the routes

var routes = require('./routes/index.js');

app.use('/', routes);

//Catching any missed exceptions

process.on('uncaughtException', function (err) {

    console.log(err);

});

//Starting the SERVER

app.listen(3000).timeout = 500000;

console.log('Starting Server on port : 3000 ');
```

C.2 routes.js

```
/**@routes.js: Server-Side Application logic besides handling Routes in Node.js */

var express = require('express'),

router = express.Router(),

bodyParser = require('body-parser'),

request = require('request'),

MongoClient = require('mongodb').MongoClient;
```

```
//Logging a message on any incoming request

router.use(function(req,res,next){

console.log('Incoming.. !!');

next();

});

//Routes to handle incoming GET and POST requests

//Route for root folder /

router.get('/mt6d_view', function(req,res) {

res.sendfile('public/index.html');

});

router.get('/attacker_view', function(req,res) {

res.sendfile('public/index_att.html');

});

router.get('/omni_database/mt6dView_collection/', function(req,res) {

MongoClient.connect('mongodb://localhost:27017/omni_database',\

function(err, db){

if(err) throw err;

var cursor = db.collection('mt6dView_collection')\

  .find({}, {'_id' : 0}).toArray(function(err, docs){

if(err) throw err;

res.json(docs);
```

```
db.close();

});

});

});

router.get('/omni_database/attackerView_collection/', function(req,res) {

MongoClient.connect('mongodb://localhost:27017/omni_database', function(err, db){

if(err) throw err;

var cursor = db.collection('attackerView_collection')\

.find({}, {'_id' : 0}).toArray(function(err, docs)\

{

if(err) throw err;

res.json(docs);

db.close();

});

});

});

module.exports = router;
```

C.3 index.html

```
<!DOCTYPE html>
```

```
<html >

<head>

<meta charset="UTF-8">

<meta http-equiv="refresh" content="35">

<title>MT6DView/AttackerView Visualizer</title>

<link rel="stylesheet" href="css/style.css">

</head>

<body>

<script type="text/javascript" src="//

cdnjs.cloudflare.com/ajax/libs/d3/3.3.10/d3.min.js"></script>

<script type="text/javascript" src="//

cdnjs.cloudflare.com/ajax/libs/jquery/3.0.0-alpha1/jquery.min.js"></script>

<script src="js/index.js"></script>

</body>

</html>
```

C.4 index.js

This code is an adaptation of [17] D3.js repository

```
var vizUrl = 'omni_database/mt6dView_collection/';

//var vizUrl = 'omni_database/attackerView_collection/';
```

```
#Preparing an Ajax request

var xmlHttp = new XMLHttpRequest();

xmlHttp.open('GET', vizUrl, false);

xmlHttp.send(null);

var jsonObject = JSON.parse(xmlHttp.responseText);

jsonObject = jsonObject[0];

var diameter = 2600;

var margin = {top: 20, right: 120, bottom: 20, left: 120},

width = diameter,

height = diameter;

var i = 0,

duration = 350,

root;

var div = d3.select("body").append("div")

.attr("class", "tooltip")

.style("opacity", 1e-6);

var tree = d3.layout.tree()

.size([360, diameter / 2 - 80])

.separation(function(a, b)

{ return (a.parent == b.parent ? 1 : 1) / a.depth; });

var diagonal = d3.svg.diagonal.radial()
```

```
.projection(function(d)

{ return [d.y, d.x / 180 * Math.PI]; });

var svg = d3.select("body").append("svg")

.attr("width", width )

.attr("height", height )

.append("g")

.attr("transform", \

"translate(" + diameter / 2 + "," + diameter / 2 + ")");

root = jsonObject;

root.x0 = height / 2;

root.y0 = 3;

update(root);

d3.select(self.frameElement).\

style("height", "1600px");

function update(source) {

// Compute the new tree layout.

var nodes = tree.nodes(root),

links = tree.links(nodes);

// Normalize for fixed-depth. \

//fixed depth confines to smaller size.

//(*130) changes the size
```

```
nodes.forEach(function(d) { d.y = d.depth * 220; });

// Update the nodes

var node = svg.selectAll("g.node")

.data(nodes, function(d) \

{ return d.id || (d.id = ++i); });

// Enter any new nodes at the parent's previous position.

var nodeEnter = node.enter().append("g")

.attr("class", "node")

.attr("transform", function(d) { return "rotate(" \

      + (d.x - 90) + ")translate(" + d.y + ")"; })

.on("click", click)

.on("mouseover", onMouseOver)

nodeEnter.append("circle")

.attr("r", 1e-6)

.style("fill", function(d)

 { return d._children ? "DarkSalmon" : "#fff"; });

nodeEnter.append("text")

.attr("x", 10)

.attr("y", 10)

.attr("dy", ".35em")

.attr("text-anchor", "middle")
```

```
.attr("transform", function(d)
{ return d.x < 180 ? "translate(0)" : "rotate(180)translate(-" + \
(d.name.length * 20)  + ")"; })
.attr("pointer-events", "none")
.text(function(d) { if (d.name.length == 39 \
&& d.name.substring(0,20) == '2001:0468:0c80:c111:')
 { return d.name.substring(20,39);} return d.name; })
.style("fill-opacity", 1e-6)
.style("font", "13px sans-serif");
// Transition nodes to their new position.
var nodeUpdate = node.transition()
.duration(duration)
.attr("transform", function(d) { return "rotate(" + \
(d.x - 90) + ")translate(" + d.y + ")"; })
nodeUpdate.select("circle")
.attr("r", 4)
.style("fill", function(d) { return d._children ? "black" : "#fff"; });
nodeUpdate.select("text")
.style("fill-opacity", 1)
.attr("transform", function(d) { return d.x < 180 ?\
"translate(0)" : "rotate(180)translate(-" +\
```

```
(d.name.length + 75)  + ")"; });

var nodeExit = node.exit().transition()

.duration(duration)

.remove();

nodeExit.select("circle")

.attr("r", 1e-6);

nodeExit.select("text")

.style("fill-opacity", 1e-6);

// Update the links

var link = svg.selectAll("path.link")

.data(links, function(d) { return d.target.id; });

link.enter().insert("path", "g")

.attr("class", "link")

.attr("d", function(d) {

var o = {x: source.x0, y: source.y0};

return diagonal({source: o, target:o});

});

// Transition links to their new position.

link.transition()

.duration(duration)

.attr("d", diagonal);
```

```
// Transition exiting nodes to the parent's new position.

  link.exit().transition()

.duration(duration)

.attr("d", function(d) {

var o = {x: source.x, y: source.y};

return diagonal({source: o, target: o});

 })

.remove();

// Stash the old positions for transition.

nodes.forEach(function(d) {

d.x0 = d.x;

d.y0 = d.y;

});

}

// Toggle children on click.

function click(d) {

if (d.children) {

d._children = d.children;

d.children = null;

} else {

d.children = d._children;
```

```
d._children = null;
}
update(d);
}
// Collapse nodes
function collapse(d) {
if (d.children) {
d._children = d.children;
d._children.forEach(collapse);
d.children = null;
}
}
node.append("circle")
.on("mouseover", mouseover)
.on("mousemove", function(d){mousemove(d);})
.on("mouseout", mouseout)
.attr("fill","red")
.attr("r", 5.5);
node.append("text")
.attr("dx", 8)
.attr("dy", 3)
```

```
.text(function(d) { return d.name; })

function onMouseOver(d)

{

if (d.parent)

{

d3.select(this).select("circle").attr("r", 7);

d3.select(this).select("text").attr("x", 20)

d3.select(this).select("text").attr("y", 30)

d3.select(this).select("text").style("font", \

"20px sans-serif");

d3.select(this).select("text").\

style("fill-opacity", 1.0);

d3.select(this).select("text").\

style("font-weight", "bold");

}

printNodeInfo(d);

}

function onMouseOut(d)

{

if (d.parent)

{
```

```
d3.select(this).select("circle").attr("r", 4);

d3.select(this).select("text").attr("x", 10)

d3.select(this).select("text").attr("y", 10)

d3.select(this).select("text").style("font", \

"13px sans-serif");

 d3.select(this).select("text").style("fill-opacity",1.0);

d3.select(this).select("text").style("font-weight", "none");

}

printNodeInfo(d);

}
```

Bibliography

[1] Mat Ford. New internet security and privacy models enabled by ipv6. pages 2–5. IEEE, 2005.

[2] S. Deering and R. Hinden. Internet Protocol, Version 6 (IPv6) Specification. RFC 2460 (Draft Standard), December 1998. URL `http://www.ietf.org/rfc/rfc2460.txt`. Updated by RFCs 5095, 5722, 5871, 6437, 6564, 6935, 6946.

[3] D. Hankins and T. Mrugalski. Dynamic Host Configuration Protocol for IPv6 (DHCPv6) Option for Dual-Stack Lite. RFC 6334 (Proposed Standard), August 2011. URL `http://www.ietf.org/rfc/rfc6334.txt`.

[4] S. Thomson, T. Narten, and T. Jinmei. IPv6 Stateless Address Autoconfiguration. RFC 4862 (Draft Standard), September 2007. URL `http://www.ietf.org/rfc/rfc4862.txt`.

[5] Matthew Dunlop, Stephen Groat, William Urbanski, Randy Marchany, and Joseph Tront. Mt6d: A moving target ipv6 defense. In *Military Communications Conference, 2011-Milcom 2011*, pages 1321–1326. IEEE, 2011.

[6] Matthew Dunlop, Stephen Groat, William Urbanski, Randy Marchany, and Joseph Tront. The blind man's bluff approach to security using ipv6. *Security & Privacy, IEEE*, 10(4):35–43, 2012.

[7] Dileep Basam, Randy Marchany, and Joseph G Tront. Attention: moving target defense networks, how well are you moving? In *Proceedings of the 12th ACM International Conference on Computing Frontiers*, page 54. ACM, 2015.

[8] Google. Ipv6 adoption statistics among google users. *see http://www.google.com/intl/en/ipv6/statistics.html*, 2015.

[9] T. Narten, E. Nordmark, and W. Simpson. Neighbor Discovery for IP Version 6 (IPv6). RFC 2461 (Draft Standard), December 1998. URL `http://www.ietf.org/rfc/rfc2461.txt`. Obsoleted by RFC 4861, updated by RFC 4311.

[10] R. Vida and L. Costa. Multicast Listener Discovery Version 2 (MLDv2) for IPv6. RFC 3810 (Proposed Standard), June 2004. URL `http://www.ietf.org/rfc/rfc3810.txt`. Updated by RFC 4604.

[11] Stephen Groat, Matthew Dunlop, W. Urbanksi, Randy Marchany, and Joseph Tront. Using an ipv6 moving target defense to protect the smart grid. In *Innovative Smart Grid Technologies (ISGT), 2012 IEEE PES*, pages 1–7, Jan 2012. doi: 10.1109/ISGT.2012.6175633.

[12] The Honeynet Project's 2009 Google Summer of code. Dionaea. *see http://dionaea.carnivore.it/*, 2009.

[13] Ruben Espadas. Dionaeafr. *see https://github.com/rubenespadas/DionaeaFR*, 2014.

[14] Canonical. Lxc. *see https://linuxcontainers.org/lxc*, 2015.

[15] Philippe Biondi. Scapy. *see http://www. secdev. org/projects/scapy*, 2011.

[16] Gerald Combs et al. Wireshark. *Web page: http://www. wireshark. org/last modified*, pages 12–02, 2007.

[17] Anon Captain. Visualization code base. Lxc. *see http://www.codepen.io/anon*, 2014.

[18] Michael Bostock. D3. js. *Data Driven Documents*, 2012.

[19] Dag Wieers. Dstat. *see http://dag.wiee.rs/home-made/dstat/*, 2013.

[20] Matthew Ford, Jonathan Stevens, and John Ronan. Initial results from an ipv6 darknet13. In *Internet Surveillance and Protection, 2006. ICISP'06. International Conference on*, pages 13–13. IEEE, 2006.

[21] David Moore, Colleen Shannon, Geoffrey M Voelker, and Stefan Savage. *Network telescopes: Technical report*. Department of Computer Science and Engineering, University of California, San Diego, 2004.

[22] Barry Irwin. A network telescope perspective of the conficker outbreak. In *Information Security for South Africa (ISSA), 2012*, pages 1–8. IEEE, 2012.

[23] Lance Spitzner. Honeypots: Catching the insider threat. In *Computer Security Applications Conference, 2003. Proceedings. 19th Annual*, pages 170–179. IEEE, 2003.

[24] H PROJEC. Know your enemy: Statistics, 2002.

[25] Iyad Kuwatly, Malek Sraj, Zaid Al Masri, and Hassan Artail. A dynamic honeypot design for intrusion detection. In *Pervasive Services, 2004. ICPS 2004. IEEE/ACS International Conference on*, pages 95–104. IEEE, 2004.

[26] Christopher Hecker, Kara L Nance, and Brian Hay. Dynamic honeypot construction. In *10th Colloquium for Information Systems Security Education, University of Maryland, USA*. Citeseer, 2006.

[27] Kazuya Kishimoto, Kazuya Ohira, Yoshio Yamaguchi, Hirofumi Yamaki, and Hiroki Takakura. An adaptive honeypot system to capture ipv6 address scans. In *Cyber Security (CyberSecurity), 2012 International Conference on*, pages 165–172. IEEE, 2012.

[28] Jeff Hieb and James H Graham. *Anomaly Based Intrusion Detection for Network Monitoring Using a Dynamic Honeypot*. PhD thesis, University of Louisville, 2004.

[29] Albert Walter Brzeczko. Scalable framework for turn-key honeynet deployment. 2014.

[30] Nogol Memari and Shaiful Jahari Hashim Khairulmizam Samsudin. Design of a virtual hybrid honeynet based on lxc virtualisation for enhanced network security. *EDITORIAL BOARD*, page 120, 2014.

[31] Pavol Sokol and P Pisarcik. Digital evidence in virtual honeynets based on operating system level virtualization. *Proceedings of the Security and Protection of Information*, pages 22–24, 2013.

[32] Dileep Basam, J Scot Ransbottom, Randy Marchany, and Joseph G Tront. Strengthening mt6d defenses with honeypot capabilities. *Journal of Electrical and Computer Engineering [Special Issue: System and Network Security]*, 2015.

[33] Christopher Morrell, Reese Moore, Randy Marchany, and Joseph G. Tront. Dht blind rendezvous for session establishment in network layer moving target defenses. In *Proceedings of the Second ACM Workshop on Moving Target Defense*, MTD '15, pages 77–84, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3823-3. doi: 10.1145/2808475.2808477. URL `http://doi.acm.org/10.1145/2808475.2808477`.

[34] C. Morrell, J.S. Ransbottom, R. Marchany, and J.G. Tront. Scaling ipv6 address bindings in support of a moving target defense. In *Internet Technology and Secured Transactions (ICITST), 2014 9th International Conference for*, pages 440–445, Dec 2014. doi: 10.1109/ICITST.2014.7038852.