2011

Correlating IPv6 addresses for network situational awareness

Jason Froehlich

Follow this and additional works at: https://scholarworks.rit.edu/theses

Recommended Citation

This Thesis is brought to you for free and open access by RIT Scholar Works. It has been accepted for inclusion in Theses by an authorized administrator of RIT Scholar Works. For more information, please contact ritscholarworks@rit.edu.
Correlating IPv6 Addresses for Network Situational Awareness

By

Jason Froehlich

Committee Members

Daryl Johnson
Bruce Hartpence
Yin Pan

Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Networking and Systems Administration

Rochester Institute of Technology

B. Thomas Golisano College of Computing and Information Sciences

May 17, 2011
Abstract

The advent of the IPv6 protocol on enterprise networks provides fresh challenges to network incident investigators. Unlike the conventional behavior and implementation of its predecessor, the typical deployment of IPv6 presents issues with address generation (host-based autoconfiguration rather than centralized distribution), address multiplicity (multiple addresses per host simultaneously), and address volatility (randomization and frequent rotation of host identifiers). These factors make it difficult for an investigator, when reviewing a log file or packet capture ex post facto, to both identify the origin of a particular log entry/packet and identify all log entries/packets related to a specific network entity (since multiple addresses may have been used).

I have demonstrated a system, titled IPv6 Address Correlator (IPAC), that allows incident investigators to match both a specific IPv6 address to a network entity (identified by its MAC address and the physical switch port to which it is attached) and a specific entity to a set of IPv6 addresses in use within an organization's networks at any given point in time. This system relies on the normal operation of the Neighbor Discovery Protocol for IPv6 (NDP) and bridge forwarding table notifications from Ethernet switches to keep a record of IPv6 and MAC address usage over time. With this information, it is possible to pair each IPv6 address to a MAC address and each MAC address to a physical switch port. When the IPAC system is deployed throughout an organization's networks, aggregated IPv6 and MAC addressing timeline information can be used to identify which host caused an entry in a log file or sent/received a captured packet, as well as correlate all packets or log entries related to a given host.
Acknowledgements

Thank You:

To my family, who have always supported me through the years;

To my mentors, advisors, professors, colleagues, and friends, whose encouragement and inspiration have propelled me forward;

And to my Lord, whose gracious blessings have provided me many wonderful opportunities.

– Jason Froehlich
# Table of Contents

1. Introduction .................................................................................................................. 1
2. Problem ......................................................................................................................... 1
3. Background .................................................................................................................... 5
   3.1 What is IPv6? ............................................................................................................. 5
   3.2 Why IPv6? ................................................................................................................ 6
   3.3 Current State of IPv6 Deployment .......................................................................... 7
   3.4 Use of Multiple Addresses ..................................................................................... 7
   3.5 Privacy Extensions for Address Autoconfiguration in IPv6 .................................... 9
4. Related Work ................................................................................................................. 10
5. Methodology ................................................................................................................ 12
   5.1 Foundation Procedure ............................................................................................ 13
   5.2 IPAC Modules ......................................................................................................... 16
      5.2.1 Agent Module .................................................................................................. 17
      5.2.2 Trap Handler Module ...................................................................................... 17
      5.2.3 Server Module ................................................................................................. 18
      5.2.4 Interface Module ............................................................................................. 19
   5.3 Other Considerations ............................................................................................... 19
      5.3.1 Sending Unicast vs. Multicast Solicitations ..................................................... 19
      5.3.2 Handling of Conflicts and Spoofing ................................................................ 20
      5.3.3 Time Synchronization ...................................................................................... 23
      5.3.4 Security Issues in SNMP ................................................................................. 23
6. IPv6 Operations Survey ............................................................................................... 24
   6.1 Testing Methodology ............................................................................................... 24
      6.1.1 Neighbor Discovery Protocol Behavior ......................................................... 24
      6.1.2 Use of Multiple IPv6 Addresses ..................................................................... 27
      6.1.3 IPv6 Privacy Extensions Implementation ...................................................... 28
   6.2 Survey Observations ............................................................................................... 29
      6.2.1 NDP Behavior Test .......................................................................................... 29
      6.2.2 Multiple IPv6 Addresses Test ......................................................................... 30
      6.2.3 Privacy Extensions Test ................................................................................... 31
7. Implementation ............................................................................................................. 32
   7.1 Agent Module .......................................................................................................... 32
   7.2 Trap Handler Module ................................................................................................ 41
   7.3 Server Module ......................................................................................................... 42
   7.4 Interface Scripts ...................................................................................................... 44
      7.4.1 Database Query Interface ................................................................................ 44
      7.4.2 Interactive Timeline Graph .............................................................................. 46
      7.4.3 Text Log Processing ......................................................................................... 46
      7.4.4 Libpcap File Processing ................................................................................... 47
   7.5 Databases ................................................................................................................. 48
   7.6 Time Synchronization .............................................................................................. 48
   7.7 Use of IPv4 and IPv6 ............................................................................................... 49
   7.8 MAC Notifications ................................................................................................. 50
8. Results .......................................................................................................................... 51
8.1 Demonstration Network ................................................................. 51
8.2 NDP Traffic Processing / IPAC System Operation .......................... 53
8.3 Interface Scripts Operation .......................................................... 55
9. Future Work .................................................................................. 59
  9.1 Bind port to user (802.1x) ............................................................ 59
  9.2 Multiple VLANs per Agent .......................................................... 60
  9.3 Additional Log Parsing Capabilities ............................................. 61
  9.4 Interactive Log Processing ........................................................... 62
  9.5 Impact on Wireless Networks ...................................................... 62
  9.6 Additional ICMPv6 Security Protections ...................................... 63
  9.7 Correlating Multiple IPv6/MAC/Port Pairs .................................... 64
  9.8 IPv4 .......................................................................................... 65
10. Conclusions ................................................................................... 65
11. Appendices ..................................................................................... 67
  11.1 Survey Results ............................................................................ 67
    11.1.1 Host System Information ....................................................... 67
    11.1.2 NDP Behavior Results .......................................................... 79
    11.1.3 Multiple IPv6 Addresses Results ......................................... 101
    11.1.4 Privacy Extensions Results .................................................. 106
  11.2 Database Design ......................................................................... 117
    11.2.1 Agent .................................................................................. 117
    11.2.2 Trap Handler ....................................................................... 118
    11.2.3 Server .................................................................................. 119
  11.3 IPAC Implementation Instructions ................................................. 122
    11.3.1 Environment ....................................................................... 122
    11.3.2 Database Implementation ..................................................... 122
    11.3.3 Agent Module Implementation .............................................. 124
    11.3.4 Trap Handler Module Implementation .................................... 124
    11.3.5 Server Module Implementation ............................................. 125
    11.3.6 Interface Scripts Implementation ......................................... 125
  11.4 IPAC Source Code ..................................................................... 127
    11.4.1 IPAC System ....................................................................... 127
    11.4.2 Survey Tools ....................................................................... 127
  11.5 Bibliography ................................................................................. 128
1. Introduction

Network forensics and incident response play a critical role in recovering from and preventing further attacks against enterprise networks. One of the key capacities required for the effective execution of these activities is network situational awareness – knowing what hosts, services and traffic exist on your organization’s network. Certain aspects of the behavior of Internet Protocol Version 6 (IPv6) make it difficult to precisely match an address observed in a log file or packet capture to an actual host at a later time, making complete network situational awareness difficult to obtain. With the gradually accelerating deployment of the IPv6 protocol on enterprise networks, investigators need a mechanism that will provide this capability. In this thesis, I propose and demonstrate a system – IPv6 Address Correlator (IPAC) – that allows network administrators and incident responders to track the use of IPv6 addresses within their organization’s network and identify which node utilized a specific IP address at any given time.

In this document, the term “IP” refers to IPv6 unless otherwise stated.

2. Problem

One of the requirements for successful network forensic investigation revolves around the topic of network node identification. In today's network architectures, identifiers usually come in the form of IP and MAC addresses. During the course of an investigation, investigators will likely encounter these identifiers in system and application logs, firewall logs, packet captures, and sometimes even in recovered executable code. In order for these information sources to be of any use, the investigator must be able to correlate the network identifiers to actual hosts and users. Specifically, the investigator must have two key capabilities: definitively linking an address back to a specific node at a specific point in time, and correlating all addresses used by a specific node over the course of time.

Linking an address back to a specific node at a specific time provides an investigator with the ability to identify the exact host which caused a log entry or originated/received a captured packet. In order to accomplish this, network address usage must be tracked over time by some mechanism that records the timing of address activation and deactivation. Specifically, this mechanism should record the times that each address became active on the network (as well as the ID of the associated host) and the time that it became no longer active (from either the host...
leaving the network or address rotation). It is not sufficient for the investigator to attempt to pair the identifier to a host at the time of the investigation, as it is increasingly likely (based on the amount of lapsed time) that the actual host has either left the network or adopted a different identifier by that time. If the identifier has been reused by another host during the course of that time interval, attempting to do this would also result in false attribution of the log entry/packet to the non-offending host.

Identifying all addresses used by a specific host provides an investigator with the ability to correlate all log entries and/or captured packets generated by the host in question. It is quite likely that a single host will utilize more than one network address over time, and in many cases simultaneously (in fact, with IPv6 this is required). Without this capability, each identifier seen in logs/packet captures will simply appear to be a unique host, providing an incomplete picture of the scenario. For example, let's consider a host that implements two IP addresses simultaneously, and uses each address to launch attacks against different hosts on its network. If an investigator identifies some evidence of the attacks originating from one of its addresses and only uses that address for the basis of his investigation, only a portion of the attack targets will be recognized. Only by knowing all addresses used by the attacking system can he identify all of the attacked targets.

Network forensic analysis is complicated by the use of IPv6 because of two aspects of the nature of IPv6 addressing: multiplicity and volatility.

As discussed in Section 3.4, hosts with IPv6 functionality will always have more than one active IP address if they are able to communicate beyond their local network segment. In some cases, the actual number of addresses used by the host could be upwards of ten, or even several dozen. In each case the exact number of addresses used by the host will depend on three factors: the number of network prefixes advertised on the network segment, whether or not Privacy Extensions for IPv6 are enabled, and (if Privacy Extensions are enabled) how long the host has been active on the network. For each network prefix used by the network segment, an autoconfigured EUI-64 address will be generated [20]. If Privacy Extensions are enabled, a pseudorandom temporary address will also be generated for each network prefix. For every Preferred Lifetime interval (usually one day) that the host has been active on the network beyond the first, up to the number of intervals allowed by the Valid Lifetime (usually seven days), the
host will have an additional pseudorandom temporary address for each network prefix. The formulas used to calculate how many autoconfigured addresses a host will have at any given time on a given network interface are provided in Figure 2.1. Note that these formulas do not take into consideration any addresses obtained by methods other than Autoconfiguration, such as DHCPv6 or static assignment.

<table>
<thead>
<tr>
<th>Privacy Extensions Disabled</th>
<th>$\alpha = \rho + \vartheta$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Privacy Extensions Enabled$^1$</td>
<td>$\alpha = \rho \left( \left\lfloor \frac{\tau}{\vartheta} \right\rfloor + 1 - \max \left( \left\lfloor \frac{\tau - \nu}{\vartheta} \right\rfloor, 0 \right) \right) + \vartheta$</td>
</tr>
</tbody>
</table>

Where:
- $\alpha$ = # of IPv6 Addresses in use
- $\rho$ = # of IPv6 Network prefixes on network segment
- $\tau$ = Time host has been active on network
- $\vartheta$ = Time the temporary address is preferred (lowest of Host and Router Advertisement values)
- $\nu$ = Time the temporary address is valid (lowest of Host and Router Advertisement values)
- $\vartheta$ = Single Link Local Address

As a result of the methods of generating autoconfigured addresses, the volatility of IPv6 addresses usually conforms to one of two diametric cases: either near-zero volatility or extreme volatility, with little middle ground in normal scenarios. The determining factor for this is the use of Privacy Extensions. If Privacy Extensions are not enabled on the host, then all autoconfigured addresses will reflect the network interface's MAC address (constructed in the EUI-64 format)$^2$. The only variation in addresses over time would be a result of either a change in MAC address (most likely through spoofing) or a change in network prefixes included in the network segment's Router Advertisements. In this case, correlating the IP address to a host may become somewhat easier, since the IP addresses can be reversed to provide what is most likely the host's MAC address. On the other hand, enabling Privacy Extensions on a host will result in

$^1$ Note that this formula assumes that the Network Prefixes advertised on the network segment remain consistent for the duration that the host is on the network. If prefixes are added or removed during this time, the number of temporary addresses would be based upon the time since the prefix's addition/removal rather than the time the host has been attached to the network. This also assumes that all generated addresses have the same Valid and Preferred lifetime values. If these lifetime values vary between network prefixes, each prefix would need to be considered individually.

$^2$ Except for Windows 7 and Windows Server 2008 hosts, as discussed in Section 6.2.2.
an extreme amount of volatility. In this case, the addresses used for all outbound connections from the host will be randomized each time its network interface is initialized, or daily (or other custom-configured Preferred Lifetime interval) if it is attached to a network for a long-term period. It is important to note that Privacy Extensions are enabled by default (when IPv6 functionality is enabled) on all Microsoft Windows desktop operating systems since Windows XP, a group that constituted over 90% of all personal computers used in North American and European businesses in 2010 [18].

Address multiplicity and volatility are by no means unique to the IPv6 protocol. Many operating systems that provide IPv4 functionality do include some mechanism for utilizing more than one IPv4 address on a single network interface, usually through static assignment. Address volatility can also become a concern with IPv4 when DHCP pools see high utilization (resulting in IP addresses being quickly re-assigned to different nodes) or when hosts are regularly mobile between networks. However, these cases are highly unusual (in the case of multiple addresses on a single interface) or largely avoidable with proper network planning (in the case of DHCP pool utilization), and are not as significant when compared to their impact in the IPv6 protocol.

When considering the issue of network addressing in relation to incident investigation, specifically in regard to IPv6, we recognize the need for a mechanism to track the use of network addresses. In order for such a system to be effective, it must satisfy several requirements:

1) It must be able to bind the address to a specific network entity. For the purposes of this thesis, a network entity could be a piece of hardware, a physical network port, or a user account. The system must be able to handle the attribution of multiple addresses to a single entity both simultaneously and over time. Additionally, it must be able to handle the attribution of a single address to multiple entities both over time and simultaneously (although the simultaneous use of a single address by multiple entities is a violation of protocol guidelines in most network scenarios, the system should be able to handle, recognize, and possibly provide an alert for such cases).

2) It must be able to provide precise timing of address initialization and termination on a network. This precision must be enough to prevent scenarios in which an address-entity relationship is not recognized or an address is attributed to an incorrect host. In situations when the precision is not enough to definitively point to a single entity, all possible entities should be recognized.

3) It must have an exhaustive view of address use on the network. Any address in use on the network, whether obtained through autoconfiguration, static assignment or DHCP, must be
Correlating IPv6 Addresses for Network Situational Awareness

Jason Froehlich

recorded by the system. This requires the use of some mechanism that sees all addresses, regardless of how they were acquired by the host.

4) It must not rely on host-initiated actions to obtain information about address use. If the host were to become compromised, any such updates could be modified or stopped completely, leaving the system with an incomplete, or even worse an incorrect, view of the network. One of the tenants of computer forensic investigations is to assume that all investigated hosts have been compromised and to not place complete trust in their logging mechanisms. This requirement results in the need for a system that operates apart from existing network nodes.

5) It must be resilient to record spoofing and other attack attempts. Instead of trusting a single indicator (whatever that may be for the system), address-entity bindings should be verified by some second-level attempt. This will prevent the system from recording inaccurate information.

In addition to these requirements, the system should also provide the following ideal (but not required for forensic validity) characteristics:

- It should not provide a noticeable impact on host or network performance. Any traffic generated by the system should be minimized and constrained to the local network. When possible, this traffic should be sent via unicast rather than broadcast or multicast to reduce network bandwidth and host resource consumption. Ideally, it should be based on a process or protocol already existing on the network.

In this thesis, I will constrain the scope of this problem to networks controlled by a single administrative authority (Local Area Networks within an organization), rather than the whole of the Internet.

3. Background

Although it has been around for over a decade and a half, the IPv6 protocol is still a foreign concept to many network professionals. In this section, I discuss what IPv6 is, why it is being implemented, how it is currently being deployed, and some issues that provide a challenge during its deployment.

3.1 What is IPv6?

Internet Protocol Version 6 (IPv6) is a network addressing protocol developed to facilitate communication between nodes on the “next generation” Internet. First introduced by IETF RFC 1883 [11] in 1995, IPv6 was designed as a replacement for the near ubiquitous IPv4 in use since
the early 1980's. Since its first introduction, IPv6 functionality has been touched on in some way by over 270 IETF RFCs. The most current definition of the protocol's core functionality can be found in IETF RFC2460 [12].

3.2 Why IPv6?

There have been several reasons cited as the motivation for the development of IPv6. Some of these motivations include the addition of security and serviceability features not found in the original IPv4 standards. Among the promised features were mechanisms for network level encryption and Quality of Service, both built into the protocol's core functionality. However, both of these elements were back-ported into IPv4 implementations in the form of protocol extensions, leaving the IPv6 protocol with little in terms of new functionality to encourage its adoption.

Perhaps the most substantive motivation for the development and implementation of IPv6 is the inevitable exhaustion of IPv4 addresses. Developed in the late 1970's, IPv4 was designed for small-scale research and defense industry networks [28]. At the time, the four billion unique values provided by 32-bit addresses seemed more than sufficient for the future growth of those types of systems. However, since the adoption of personal computing, commercialization of the Internet, and popularization of Internet-enabled mobile devices, four billion addresses has proven to be woefully insufficient. The development of Classless Inter-Domain Routing (CIDR) [17] and Network Address Translation (NAT) [41] in the mid 1990's has allowed network administrators to use their limited supply of IPv4 Addresses in a much more effective manner, postponing the expected exhaustion of addresses for several years. However, these two “tricks” do not provide long-term sustainability, and have almost reached the limit of their utility in terms of IPv4 address minimization.

As of January 31, 2011, the Internet Assigned Numbers Authority (IANA) has completely exhausted their global pool of IPv4 addresses, leaving regional registrars with only an estimated six months of available IPv4 addresses in reserve [37]. Once these regional pools have been exhausted, no new IPv4 addresses will be available. IPv6 mitigates the issue of IP address exhaustion by implementing an incredibly large address space unlikely to be substantially utilized in the next several hundred years. The new protocol's 128-bit addresses provide over 340 Undecillion \( (3.40 \times 10^{38}) \) unique values, which calculates to over 49 Octillion \( (4.9 \times 10^{29}) \)
addresses for each of the world's current occupants.³

3.3 Current State of IPv6 Deployment

Despite having existed for over 15 years, the current deployment of IPv6 in production networks is quite infinitesimal. As of mid February 2011, there are over 37,000 unique Autonomous Systems (AS) advertising over 386,500 IPv4 prefixes on the Internet [3]. At the same time, there only 3,200 Autonomous Systems advertising 5,000 IPv6 prefixes [4]. While the Internet's AS structure may not be a 1-to-1 match between the two protocols, these numbers do seem to indicate that less than ten percent of the world's networks implement any IPv6 functionality. Further, these numbers reflect the networks that implement IPv6 addressing, but speak nothing to service availability over IPv6. Even though these networks have IPv6 addressing enabled, it cannot be assumed that all of their services have been updated to support the protocol.

Although IPv6 deployment is still very small overall, we are starting to see an increase in both its public awareness and adoption rate. A graph tracking AS's with IPv6 prefixes over the last four years shows that its growth has been parabolic, roughly quadrupling in count over that period of time [2]. A number major web services have committed to participating in World IPv6 Day (June 8, 2011) [26], during which these services will enable IPv6 access to their main websites for 24 hours. The United States federal government has also taken steps to encourage its adoption among federal agencies. In 2005, the Office of Management and Budget issued a mandate that required Executive branch agencies to implement IPv6 functionality on their network backbones by 2008 [15], a goal that was met by all 24 of the targeted entities [32]. More recently (September 2010), the same office issued another mandate that all public-facing services use native IPv6 by June 30, 2012 and all internal applications implement the protocol by the same date in 2014 [29]. At least one major consumer Internet Service Provider (ISP) within the U.S. has begun public trials of native IPv6 connectivity to residential customers [8].

3.4 Use of Multiple Addresses

The shift to IPv6 brings to light an issue that isn't quite new to Network layer addressing, but is much more pronounced than in previous generations of Layer 3 protocols. In most host

³ Considering an estimated world population of 6.9 billion, as of February 2011 [43]
implementations of IPv4, it was usually possible to utilize more than one unicast IP address on a
single network interface, but in the majority of scenarios only one was necessary. However, with
IPv6 each Internet-connected host will have at least two, and in some cases nine or more, unique
network addresses per interface. This plethora of IP addresses can be the result of two different
mechanisms within the IPv6 protocol: Multiple Scopes and Privacy Extensions for Address
Autoconfiguration. (Privacy Extensions are discussed in the following section.)

When using IPv6, network administrators use the concept of scope to reflect the
“topological span within which the address may be used as a unique identifier” [10]. More
simply put, scope determines how far beyond the local system a host can communicate with the
given IP address. Current standards documents specify three different types of scopes in which
hosts will generate one or more addresses: link-local, unique-local, and global.

The link-layer scope consists of only a single Layer 2 network segment, and cannot be
routed between networks. Each link-layer scope uses the same set of addresses: the fe80::/64
subnet. Every IPv6-enabled host must take part in this scope, and thus will always have a
fe80::/64 address on each network interface.

The unique-local scope consists of network segments controlled by a single
administrative authority (e.g., a corporation) and usually within a single geographic site. Traffic
can be routed between networks within the scope, but cannot be routed beyond the scope (e.g., to
the Internet). Although communication within this scope is constrained by the organization's
network borders, unique-local scope addresses are intended to be globally unique. The current
method of obtaining a unique set of addresses is to randomly choose a /48 subnet from the
fd00::/8 range (through a procedure outlined in IETF RFC4193 [21]), though there is work being
done on establishing an authority to manage allocation of the fc00::/8 range [19]. Unique-local
addresses serve as the functional replacement for site-local addresses (deprecated by IETF
RFC3879), which did not have a provision for unique addressing between sites. Addresses
within this scope are not required for IPv6 operation, but are often encouraged to facilitate
administration of an organization's network [30].

The final scope of network addresses is known as the global scope. These addresses are
unique across the whole of the Internet, and are capable of being routed between all networks. In
order for a host to be able to communicate beyond its network segment or set of organization's
networks, it must have a valid global scope address. Allocation of these addresses is hierarchically managed by IANA and regional registrars.

In addition to these three scopes in which hosts generate their own address, there exist two other scopes of IPv6 addresses in which hosts do not generate their own unique address. The first is the loopback scope, ::1/128. Similar to the 127.0.0.1 address in IPv4, this scope is only used for intra-host communication. The second scope is used for Multicast addressing, and utilizes the ff00::/8 subnet. Hosts do not create their own address in this range, but will “join” multicast groups and listen on certain sets of addresses.

3.5 Privacy Extensions for Address Autoconfiguration in IPv6

Privacy Extensions for Address Autoconfiguration in IPv6 (hereafter “Privacy Extensions” or “Privacy Extensions for IPv6”) is a set of endpoint privacy enhancement features introduced with IETF RFC 4941 [33]. When “normal” Address Autoconfiguration occurs on IPv6-enabled hosts, the generated IP address is partially based on the interface's Layer 2 MAC address (known as EUI-64); the network identifier portion (the first 64 bits) is based on the prefix observed in Router Advertisements while the host identifier portion (the last 64 bits) is a modified form of the MAC address, mutated using a formula provided in the RFC. Since the interface's MAC address does not change in normal scenarios, this practice causes the host identifier portion to be exactly the same for each network the host joins. Because the MAC address is intended to be globally unique, it becomes possible to track the device as it moves between networks.

Privacy extensions attempt to resolve this issue by generating a pseudorandom interface identifier and using that as the basis of the host identifier portion of autoconfigured IPv6 addresses. Even though an IP address based on the MAC address is still activated on the network interface, this pseudorandom address (known as a “temporary” address) becomes the primary one for all outbound communication. These temporary addresses are regenerated regularly (the RFC recommends every 24 hours by default, which most implementations follow), keeping the host's addresses constantly revolving. Each temporary address is kept active on the interface but in a deprecated state for a certain amount of time after a new temporary address is generated (the RFC recommends six additional days by default, which most implementations follow). During this time, the host will still accept traffic destined for this address, but will not
use it for any new outgoing sessions. This set of temporary addresses is generated for each
network prefix distributed in Router Advertisement traffic. Considering this, it is possible for a
host to have up to eight different autoconfigured IP addresses for each network prefix on each
interface at a given point in time: one autoconfigured based on MAC address, one temporary
address in use as the primary, and six temporary addresses still active but in a deprecated state.

This functionality is supported by most popular host operating systems, and is activated
by default on several of the most common. As discussed in Section 6.2.3, Privacy Extensions are
enabled by default in Microsoft Windows 7 and XP Professional systems, while at the same time
they are supported but not enabled by default in Mac OS X, Ubuntu 10.04 and CentOS 5.5
Linux, and Microsoft Windows Server 2008 and 2003 systems. Despite their reassuring title, it
is important to note that Privacy Extensions do not prevent service and content providers from
tracking a mobile device through Application layer data (such as HTTP cookies), but only
prevents someone “in the middle” of the traffic from doing so through Network layer
information.

4. Related Work

Potential security issues within the IPv6 protocol have been well documented and
discussed by the academic and professional communities. Žagar et al. [47] outline several
methods in which routing headers, fragmentation, ICMPv6, and popular transition mechanisms
can be exploited. Martin et al. [31] expand on this to include issues regarding IPv6 flow labels,
neighbor discovery, address auto-configuration, router renumbering, and multicast addressing, as
well as a deeper discussion of ICMPv6 vulnerabilities. Practices for mitigating these and other
potential IPv6 security issues are discussed at length in IPv6 Security by Hogg et al. [22].

Despite the attention paid to security issues within the protocol itself, there has been
relatively little discussion on how the dynamics of IPv6 addressing will affect incident response
and forensic investigations. The problem provided by the constant regeneration of addresses
when Privacy Extensions are in use is briefly mentioned by IPv6 Security. The authors
acknowledge the need to correlate an IP address to a specific user at a given time during a
forensic investigation, and that the volatility of IPv6 privacy addresses makes this difficult using
traditional methods. The two strategies that they recommend to resolve this issue are to
implement DHCPv6 in combination with IEEE 802.1X (which simply side-steps the whole issue of privacy addressing and autoconfigured addresses) or implement “strong [Dynamic DNS] systems” for tracking address changes. Neither of these strategies satisfies the forensic requirements of incident response.

Dynamic Domain Name System (Dynamic DNS or DDNS) is a mechanism that allows for the automated updating of DNS records as host addresses change. Although there are several different ways existing DNS systems have implemented this functionality, a proposed standard for DDNS is outlined in IETF RFC2136 [44], and extended in IETF RFC3007 [45] to include support for encryption and authentication. Most DNS services provide some mechanism for logging DDNS updates, usually through Syslog. However, these systems alone cannot provide a provably complete depiction of the IP addresses used on a network, nor an accurate account of the timing of address use (when the address became active, and then deprecated). In order to get precise timing of address changes, the client DDNS software would have to send updates immediately upon an address generation or deletion. Successful and secure DDNS implementations are troubled by a major challenge: how can we trust the updates from hosts? Standard encryption and authentication mechanisms require either pre-distributed keys or a preestablished Public Key Infrastructure. Even when proper authentication and encryption solutions are in place, there is still the issue of trusting that the client's updates are accurate and complete. If a host was to become compromised and its DDNS software deactivated, then any further address changes will go unobserved by the system.

The issue of binding an IP address to a host for security considerations has been previously discussed by a group from Microsoft Research [46]. In this work, the researchers are able to track hosts as they move between IP addresses. While my proposed system focuses on hosts within an enterprise network, their research takes the perspective of a service provider (in this case, a web-based email service) tracking clients over the Internet. They accomplish this by observing and correlating application layer “unreliable IDs” (user IDs and cookies) used by clients to access the web service. Although I will focus on this issue at Layer 2, the concepts of matching IPs to hosts are very similar to those used by my proposed system. However, their work deals only with IPv4 addressing and the issue of host mobility (between IP addresses), and does not take into account other addressing issues presented by IPv6, such as the use of multiple
addresses simultaneously. Additionally, they outline a method of visualizing host-to-IP address bindings on a graph that is useful in reporting IP address “ownership” over time.

Kohno et al. [27] have proposed two methods of differentiating between devices on a network by calculating internal clock skews over time, based on observation of timestamps within either TCP or ICMP traffic. Although their more direct approach that uses ICMP timestamp requests cannot be of any help (since ICMPv6 does not provide for timestamp requests), it may still be possible to leverage TCP timestamp observation to pair a MAC address to a node. Once a MAC address is observed, their system would initiate a series of TCP connections to the node, calculating the clock skew. At regular intervals, the clock skews would be recalculated and compared to previous skews, revealing whether it is the same or a spoofed node. However, this system is not completely definitive either, as demonstrated by Arackaparambil et al. [1] with an attack that spoofs another host's clock skew. Additionally, implementing this method would introduce a substantial amount of additional traffic to the network, would only work for nodes that have an open TCP port, and would be affected by clock synchronization operations (such as through NTP).

Several open source and commercial tools are currently available that simplify IP Address Management (IPAM) for network administrators, but none attempt to track IP address use in real time. These utilities focus on address planning and obtaining quick “snapshots” of address utilization at a single point in time, and few have any support for IPv6. The current leading IPAM utility is provided by Infoblox [25]. Their IPAM tool is able to determine which IP addresses are active on the network by examining DHCP and DNS records, as well as performing manual ping scans. However, these records are not exhaustive or precise enough for use in scenarios in which all IP addresses and the exact times in which they became active must be known.

5. Methodology

This section explores the principles governing the IPAC system's operation and discusses several of the considerations behind the system's design. Here I focus on theory and architecture; details of the actual implementation of the system are provided in Section 7.
5.1 Foundation Procedure

In this thesis I propose and demonstrate a system that allows network administrators and incident responders to correlate an IPv6 address to a specific Media Access Control (MAC) address and physical network switch port at a specific point in time, based solely on existing networking mechanisms already in place on many corporate networks. This capability is provided by a three-step procedure: identify all IPv6 addresses active on the network, correlate each IPv6 address to a specific Layer 2 MAC address, and then correlate the MAC address to a specific Ethernet switch port.

Step 1: Identify IPv6 Addresses

Discovering IPv6 addresses on a local network is made simple by a mechanism already in place on all IPv6-enabled networks: the Neighbor Discovery Protocol (NDP) [34]. When a node wants to send traffic to a given IPv6 address, it must first find the target’s (or the appropriate gateway router’s) MAC address to use in its outbound Layer 2 frames. It accomplishes this by sending a NDP Solicitation to the target requesting its MAC address, which will respond with an NDP Advertisement containing the appropriate identifier. Because the source node does not know the target's Layer 2 information, these Solicitations are sent to the proper Solicited Nodes multicast group, meaning that every node on the network will receive the NDP Solicitation packet, as shown in Figure 5.1.

The behavior of NDP provides us with a simple mechanism for identifying every IPv6 address active on a local network. Before an address can have traffic forwarded to it, it must appear in an NDP Solicitation. By simply listening to all NDP traffic and parsing Source and Target addresses, we can compile a complete list of every IPv6 address in use on the network. Because NDP Solicitations are multicasted to everyone, this can be accomplished completely passively. This procedure may not identify secondary IPv6 addresses that exist but are unused on a host's interface, but such cases are irrelevant as these addresses would never be seen in log files or packet captures examined by an investigator.

The only exception to this is when both ends of a connection have static NDP records for the other. This is highly unlikely to occur in most enterprise networks, as it would require all hosts to have a manually assigned NDP record for all other nodes on its local network. If either of the two does not have the proper static entry set, both will be identified in the resulting NDP Solicitation (one as the Source, the other as the Target).
Step 2: IPv6 -> MAC

Once IPv6 addresses are identified, the next step in this system is to correlate each IPv6 address to a specific MAC address. This procedure is already performed by other IPv6-enabled nodes through NDP. By simply sending our own NDP Solicitations to the observed IPv6 addresses, we can elicit a valid NDP Advertisement from the target, which will contain the appropriate MAC address value, as shown in Figure 5.2. This also allows us to validate the presence of the IP address on the network, as a non-existent address will not generate any response.

The continuous monitoring of existing multicasted NDP traffic and the acquisition of MAC addresses through NDP Solicitations provides us with a mechanism for determining the complete Layer 2 state of a local network at any given time. As new nodes join the network or IPv6 addresses change, the event will be marked by NDP traffic and observed by our system. The list of IPv6 addresses currently active on the network will be kept fresh by periodically resending NDP Solicitations; if a valid NDP Advertisement is not returned, the target can be considered to no longer be a part of the network. Historical records of the network's state can be maintained by simply recording the times in which an address joined and left the network.

Step 3: MAC -> Port

MAC addresses are intended to provide globally-unique identification of individual Network Interface Cards (NICs); however, there are several widely known methods of changing or spoofing these identifiers in network traffic [38]. As a result of the non-immutability of MAC addresses, in order for the system to satisfy forensic requirements it cannot simply rely on a MAC address to identify a system or user. Instead, it must provide some method of associating the MAC address to a specific physical device or location. The system that I have demonstrated accomplishes this with the use of managed Ethernet switches and the Simple Network Management Protocol (SNMP).

In order to provide layer 2 switching, Ethernet switches maintain an internal table that matches MAC addresses to physical ports based on the traffic they observe. Often referred to as the Content Addressable Memory (CAM) or bridge forwarding table [9], this facility provides a near real-time view of MAC-Port pairings for the set of links served by the switch. Many popular managed access-layer switches marketed to enterprises provide a mechanism for
observing the state of (and any subsequent changes to) the CAM table by a remote manager through SNMP. If a MAC address joins the network, moves to a different port, or becomes inactive on the network, the switch will notify the manager of the event and the associated physical ports through SNMP Traps or Inform messages, as shown in Figure 5.3.

Based on this facility, it is possible to maintain a complete recording of the physical switch ports used by any MAC address over time. As MAC addresses join and leave the network, notifications from the switch will inform the manager of the arrival or departure. The manager can use these recorded notifications to build its own picture of the Layer 1-2 state of the network at any point in time. So long as all Ethernet switches within the organization's network produce these SNMP messages, the MAC can be tracked as it moves throughout the network.

![Figure 5.1. Identifying IPv6 Addresses in Multicasted NDP Solicitations.](image-url)
5.2 IPAC Modules

The system that I have demonstrated, IPAC, provides the functionality outlined in the previously discussed procedure through four separate modules. Communication between these modules is illustrated in Figure 5.4, while the actual implementation of each of these modules is discussed in detail in Section 7.
5.2.1 Agent Module

The first module, the IPAC Agent, takes care of the NDP-related functions of the system. It observes all multicast NDP traffic on the network and parses the packets for Source and Target IPv6 addresses. Once addresses are observed, the Agent will generate its own NDP Solicitation packets to validate and obtain the MAC addresses associated with each. Every time an IP address is observed and validated, a sighting record containing the appropriate MAC address and timestamp is stored for later processing by the Server module. The Agent also maintains a list of IP addresses it knows to be currently active on the network. Periodically it will check this list for addresses not recently seen in other NDP traffic and attempt to re-validate the address. If the address cannot be re-validated (no valid NDP Advertisements are received after several attempts), the address is considered to have left the network and its departure is logged for later processing by the Server module. Because NDP traffic operates at Layer 2, a separate Agent is required for each broadcast domain in the organization's network, although this limitation can be reduced with a few modifications discussed in Section 9.2.

5.2.2 Trap Handler Module

The second module takes care of all SNMP Trap-related functions of the system. It listens for messages from the organization's access-layer Ethernet switches, which signal the addition or removal of a MAC address from the switch's CAM table. Once these notifications are received, they are stored for later processing by the Server module. Each switch within the organization's network must be configured to send CAM table change notifications, as discussed
in Section 5.1. Since SNMP traffic is routable, it is sufficient to have only a single Trap Handler serving the organization's entire network. However, if server or traffic load could become an issue for an organization, multiple Trap Handlers can be implemented to serve different sets of Ethernet switches.

5.2.3 Server Module

The IPAC Server module handles most of the functions related to turning raw sighting and trap records into usable and queryable data. This set of scripts produces a timeline of all IPv6-MAC address and MAC-Port pairing sessions, summarizing each session into a single database record containing the addresses/ports involved and marking the session's start and end times.

As shown in Figure 5.5, the module begins by querying the Agent databases for unprocessed sighting records. Once obtained, it will condense these records into summarized sessions marking the beginning and end times for which an IP-MAC address pair was seen on the network. These sessions are added to the Server's database, with any existing sessions being updated to reflect the new data. After all Agent records have been processed, the Server performs the same function with any unprocessed SNMP trap records. These records are correlated to identify the start and end times of each session, and the resultant data set is merged into the existing Server database. Once processed by the Server module, sighting and trap records are removed from their respective databases and discarded to prevent the reprocessing of old data.

![Figure 5.5. The IPAC Server Obtains Records From the Agent and Trap Handler Databases, Correlates and Summarizes These Records, and Stores the Summary Records in the Server Database.](image)
5.2.4 Interface Module

The final module consists of a set of scripts that provide network administrators and incident responders with multiple methods of accessing and interacting with the data maintained by the Server module.

The first script generates a graph of IP address use within the network over time. This interactive visualization provides a timeline showing when an IP address was active, its associated MAC addresses, and the exact times in which it was first and last seen. This tool is also capable of highlighting instances in which IP-MAC conflicts occur (more than one MAC claims the same IP address).

Another script provides a terminal-based utility for manually querying the Server for specific data. With this script a user can pull a wide array of information from the Server, including the MACs used by an IP within specified time boundaries, the IPs used by a MAC within specified time boundaries, the physical switch port used by a given IP or MAC address, the current state of IP or MAC address use, or instances in which a MAC conflict has occurred.

The third script will process a text-based log file containing IPv6 addresses. For every IPv6 address in the file (if a valid timestamp is present), the script will query the Server and attempt to determine the MAC addresses and Ethernet switch ports associated with the address at the given time. Each line within the file that contains an IPv6 address will have a small summary of the results appended.

The final script provides the same functionality as the previous, but applied to binary packet captures instead of text logs. Since it is not possible to modify the packet capture file without corrupting the original data, the results are provided in a separate text file.

5.3 Other Considerations

5.3.1 Sending Unicast vs. Multicast Solicitations

One major point of contemplation during the design of the Agent system regarded whether to send NDP Solicitations from the Agent module via unicast or multicast packets. Unicast Solicitations have the advantage of reducing bandwidth and host resource consumption. If the Destination MAC address exists within the Ethernet switch's CAM table, then the frame containing that packet will only be forwarded on a single link and received by a single host. On
the other hand, multicast Solicitations provide greater visibility into the network state at the cost of bandwidth and host resources. Each multicasted packet will be forwarded on all Ethernet links and received by all hosts (though if the host correctly implements its NIC drivers, the packet will not be processed beyond the Destination MAC address). Since the multicast packet is received by all hosts, it becomes possible to detect scenarios in which two or more hosts claim to have the same IPv6 address (each will send its own NDP Advertisement). Unicast NDP solicitations, since they are only received by a single host in most cases, miss this opportunity. With these points in mind, I decided to implement the system's behavior in a manner that balances the needs for network visibility and minimization of resource consumption.

Multicast solicitations are generated whenever the Agent needs to perform address discovery: identifying all MAC addresses that claim to “own” the given IP address. Even when a MAC address is already known to use the IP address in question, a multicast Solicitation is necessary to ensure that no other MAC addresses claim to have that IP address as well. Scenarios in which this is implemented include instances where the Agent observes an IP address that it doesn't already know, as well as when it observes an IP/MAC association that differs from what it already knows. Additionally, a multicast Solicitation will be sent to the Target and/or Destination IP address of every NDP Solicitation observed on the network. As a result of this, the Agent should have the exact same view of the network as the originating host.

Unicast Solicitations are generated whenever the Agent needs to perform address verification. In these cases, the IP and MAC addresses are already known by the Agent. By sending a unicast Solicitation to this known address pair, the Agent verifies that the IP is indeed used by the MAC, and that it is [still] active on the network. Scenarios in which this is implemented include verifying that the host is still on the network after a period of inactivity and verifying an IP-MAC address pair seen as an NDP Solicitation Source.

5.3.2 Handling of Conflicts and Spoofing

5.3.2.1 Conflicts

In order for the IPAC system to satisfy forensic requirements, it must be resilient enough to handle instances in which conflicting indicators exist. In the context of this system, a conflict occurs when two entities claim to use or have control of the same identifier. In the IPAC system,
a conflict can occur in two ways: two different MAC addresses claim to 'own' the same IPv6 address or two physical switch ports claim to serve the same MAC address. Although such conflicts are a violation of IP and MAC addressing standards, they are still liable to exist in production networks as a result of either misconfiguration, software malfunction, or a spoofing attack (discussed next).

The IPAC system is not capable of preventing these conflicts from occurring. Instead, its function is to identify and record the instances in which they do appear. Whenever two MAC addresses claim the same IPv6 address or two ports claim the same MAC address, each session will be recorded individually. Once it comes time for an investigator to query the system for IP address use information, the scripts provided with the Interface module will detect any conflicts existing at the specified time and provide the IDs of both entities participating in the conflict. In such cases, the system will not be able to positively identify only a single MAC address or port as the owner, but will be able to reduce the scope of possibilities to the two in question. From there, it is up to the investigator to determine exactly which one they want through other means.

Detection of an IP address conflict by an Interface module script is somewhat complicated by the behavior of address detection timing (if an IPv6 address leaves the network immediately after it has been verified by the IPAC system, its departure will not be detected until 30 seconds later). To ensure no hosts are overlooked, the script will query the IPAC databases for all addresses matching the given IP within the window of 30 seconds before and 30 seconds after the given time. If two or more are found, each will be reported as the possible owner. If only one of these was known at the exact specified time (the others began or ended during the ±30 second window), it will be marked as the priority result.

Although this is the desired behavior, it may cause false positives in certain scenarios. If a conflict began/ended less than 30 seconds after/before after the queried time, both would be reported even though the conflict didn't exist at the exact requested time. False positives may even occur when no actual conflict occurred on the network. Examples of this might include a host being quickly unplugged and plugged into a different physical switch port (MAC-Port conflict) or an IP address being moved to another device as part of a high-availability mechanism (IP-MAC conflict). In such cases, querying the IPAC database for a timestamp within 30 seconds of these events would result in a report of a conflict. Despite these situations, this
behavior is actually preferred over that which could result in false negatives. A false negative would occur when a conflict existed on the network, but because of the IPAC system's address detection timing only one node (the wrong one) was seen and reported, resulting in the attribution of a log entry or captured packet to the incorrect host. A false positive does not implicate a specific host, but draws the investigators attention to a small group for further investigation.

5.3.2.2 Spoofing

One cause of conflicts in the IPAC system's data is the spoofing of IP and MAC addresses by a malicious party. IP address spoofing occurs when a host with a different MAC address claims to have ownership of an IPv6 address already in use by another host. The IPAC system does implement a mechanism to mitigate certain spoofing attacks. When the IPAC Agent observes an already known IP address paired with a different MAC address, it will send a multicast NDP Solicitation and wait for NDP Advertisements to return before accepting the new pair. This prevents the system's data from being spoiled by an attacker simply injecting spoofed NDP traffic onto the network; if the attacker does not respond to the NDP Solicitation, the new IP-MAC address pair will be simply ignored. However, if the spoofing system does reply to the Solicitation, the addresses will be recorded and conflicts will be handled as described above. In such cases, the location of the spoofing host can still be tracked through the system's physical switch port records.

MAC address spoofing occurs when a host uses a MAC address that is different from the one assigned by the hardware manufacturer. Although it will cause a switched network to operate incorrectly, it is possible for one host to utilize a MAC address already in use by another host on the same network. In such scenarios, the Ethernet switch will observe the address on a new port and generate a MAC Learned notification to the IPAC system, revealing the physical location of the spoofing host. Any subsequent traffic from the original host will cause the switch to generate another notification indicating it learned the MAC on the original port. The records will be recorded by the IPAC system, allowing for conflict handling as described above.

In some network topologies, it may be possible for an attacker to spoof SNMP notifications to the IPAC system, resulting in spoiled data. This can be prevented by segregating management traffic onto a separate network and using strong authentication of SNMP messages,
as discussed in Section 5.3.4.

5.3.3 Time Synchronization

In order for the IPAC system to provide accurate results, all devices that generate IPAC data must have synchronized clocks. This includes all Agents, SNMP trap-generating network devices (Ethernet switches), SNMP Trap Handlers, and any device that generates a log file or packet capture. If two devices have unsynchronized clocks, MAC-IP records, MAC-Port records, and log entries may all become misaligned, reducing the accuracy of the system. Scenarios in which this misalignment is greater than 30 seconds could result in the inability to identify a host or, even worse, a mis-attribution to another host.

A standardized method of providing synchronized time to network devices is provided by the Network Time Protocol (NTP) [36]. Implemented within the host or device operating system (not within the IPAC modules), this protocol uses a centralized time server to keep the internal clocks of all devices within the organization synchronized on the order of hundreds of microseconds. Details on how NTP was deployed within the demonstration IPAC network are provided in Section 7.6.

It is important to note that in the demonstration network, preference was given to simplicity over security during the implementation of NTP in order to minimize system complexity. With the basic NTP configurations shown in Section 7.6, the only consideration for security was the choice to use client-initiated updates rather than server-initiated broadcasts (preventing rogue NTP servers from affecting the times on our systems). In addition to this basic safeguard, NTP implementations in production networks should implement strong authentication of updates through symmetric or public key cryptography [35].

5.3.4 Security Issues in SNMP

The system that I have implemented utilizes SNMP Version 1 (SNMPv1) traps to relay MAC address table modification events from Ethernet switches. A number of security issues with this version of the protocol have already been widely discussed [6]. Despite these issues, SNMPv1 was chosen for this demonstration system because of its simplicity and straightforward implementation, as well as its compatibility with available laboratory hardware and software modules. However, in a production network it is highly recommended that SNMPv1 be
replaced with the latest version of the protocol, SNMPv3, which provides much stronger agent and manager authentication as well as message encryption. This would require that all of the organization's Ethernet switches have SNMPv3 functionality available, which could be an issue for certain organizations and those with older hardware (on some Cisco Catalyst models, SNMPv3 encryption functionality is only available on those with export-controlled 'crypto' software images). If compatible Ethernet switching hardware is available, implementing SNMPv3 with this system would only require modifying the code of the Trap Handler module.

6. IPv6 Operations Survey

Before completing construction of the IPAC system, I performed a survey to determine exactly how the IPv6 protocol operates between systems from various vendors on a production network. This survey was designed to examine three key aspects of IPv6 behavior on popular Operating Systems (OS's) that would directly affect the requirements of the IPAC system's operation: the behavior of the Neighbor Discovery Protocol, use of multiple IPv6 addresses, and use of Privacy Extensions for IPv6. In this section, I will outline the testing methodology used during the survey and discuss the results found for each examined OS. Detailed results for each tested OS are provided in Section 11.1.

6.1 Testing Methodology

6.1.1 Neighbor Discovery Protocol Behavior

The first portion of this survey had the goal of examining how each operating system generated NDP traffic and handled incoming NDP packets. This behavior was determined by injecting crafted NDP Solicitations and Advertisements destined to the targeted host, observing resultant IPv6 traffic on the network, and observing changes to the targeted host's NDP cache table.

The test network in this portion of the survey consisted of an Ethernet hub, a test workstation that generated NDP packets and observed all network traffic, and a host running the targeted operating system. The targeted host existed as either a physical workstation device or a VirtualBox 4.0.4 virtual machine; the specifics for each target OS are outlined in Section 11.1.1. When a virtual machine was used as the targeted host, the host's network interface card was
bridged directly to the wired network. Additionally, the physical host running the VM did not have any IPv6 functionality enabled, leaving the target system and the testing workstation as the only nodes generating IPv6 traffic. In each NDP testing scenario, a router was attached long enough to generate Router Advertisement packets containing the global and unique-local IPv6 address scopes for the testing network (causing the target host to generate addresses within those scopes), and then detached from the network to prevent the generation of any extraneous IPv6 traffic. This network topology is illustrated in Figure 6.1.

![Network Topology for NDP Behavior Testing](image)

Figure 6.1. Network Topology for NDP Behavior Testing.

This test was not intended to be a exhaustive examination of how each host reacted to all possible NDP scenarios. Instead, it focused on the normal operation of the protocol and simple cases in which incoming packets varied slightly from the norm. The traffic generation tool used to test the NDP behavior of the targeted operating systems was custom built using the Python standard library and the dpkt module. Since this tool was not intended to be a complete protocol fuzzer that attempted to test every possible combination of options within the generated NDP Solicitation and Advertisement packets, it only tested select packet field combinations as well as all possible scenarios in which only a single field differed from a correctly constructed packet, as determined by IETF RFC4861. The source code for this tool is included in Section 11.4.2.

The testing in this scenario attempted to answer the following questions about the targeted host OS:
Generation of NDP Solicitations
1. What are the values of the IPv6 and ICMPv6 options and flags?
2. What IPv6 Address is used as the Source?
   (Same scope as Target? Link-local scope address?)
3. What values are used for IPv6 and Ethernet Destinations in multicast Solicitations?
   (All-nodes multicast/broadcast? Solicited Node Multicast?)
4. Is the Source Link Layer Address option used in the ICMPv6 datagram?

Generation of NDP Advertisements
1. What are the values of the IPv6 and ICMPv6 options and flags?
2. What IPv6 Address is used as the Source?
   (Target address from Solicitation? Link-local scope address?)
3. Is the Target Link Layer Address option used in the ICMPv6 datagram?

Handling of Incoming NDP Solicitations
1. Does the system add the Source address to its own NDP table based on this Solicitation?
2. Does this Solicitation packet cause the target system to reciprocate with an NDP Solicitation to the test system?
3. Will the system respond to malformed Solicitations?
   a) ICMPv6 Code field not 0
   b) ICMPv6 Checksum incorrect
   c) ICMPv6 Source Link Layer Address option missing
   d) ICMPv6 Source Link Layer Address option incorrect
   e) IPv6 Destination to All Nodes (not Solicited Node)
   f) IPv6 Destination to incorrect Solicited Node Address
   g) IPv6 Hop Limit field not 255
   h) IPv6 Flow Label field not 0
   i) IPv6 Traffic Class field not 0
   j) Ethernet Destination to Broadcast
   k) Ethernet Destination to incorrect Solicited Node

Handling of Incoming NDP Advertisements
1. Does the system accept unsolicited NDP Advertisements when no cache entry exists?
2. Will the system accept malformed Advertisements (solicited, non-gratuitous)?
   a) ICMPv6 Code field not 0
   b) ICMPv6 Checksum incorrect
   c) ICMPv6 Source Link Layer Address option missing
   d) ICMPv6 Source Link Layer Address option incorrect
   e) ICMPv6 Flag field combinations (Router,Solicited,Override)
      1. Router only
      2. Override only
3. Not Solicited (none)
4. Override and Solicited
   f) IPv6 Multicast to All Nodes
   g) IPv6 Multicast to Solicited Node Address
   h) IPv6 Source not matching ICMPv6 Target
   i) IPv6 Hop Limit field not 255
   j) IPv6 Flow Label field not 0
   k) IPv6 Traffic Class field not 0

6.1.2 Use of Multiple IPv6 Addresses

The second portion of this survey sought to determine how each operating system utilized IPv6 addresses when the host was on a network segment with multiple IPv6 prefixes and scopes. This was accomplished by generating ICMPv6 Ping traffic from the targeted system and observing the Source IPv6 addresses used as the packets traversed the network.

As before, the network used in this scenario contained the targeted host (either as a physical workstation or a virtual machine) and the testing workstation that sniffed network traffic, both connected to an Ethernet hub. Additionally, the network contained a second segment connected by an IPv6 router, which contained virtualized CentOS 5.5 and Windows Server 2003 servers. As before, all virtual machines were run on physical hosts with no IPv6 functionality. Each network segment contained three IPv6 scopes with individual prefixes (public, unique-local and link-local), which were distributed by ICMPv6 Router Advertisements from the router. The topology of this network is shown in Figure 6.2.
The testing in this scenario attempted to answer the following questions:

1. Will the system generate an address for each prefix in the Router Advertisement, plus a link-local scope address?
2. When pinging nodes on the same network segment, what Source addresses are used?
   a) Does the Source address prefix match the Destination address prefix?
3. When pinging nodes on a different network segment, what Source addresses are used:
   a) For NDP Solicitations to the router?
   b) For ICMPv6 Ping traffic to destination?

### 6.1.3 IPv6 Privacy Extensions Implementation

The final portion of the survey examines how the host implements Privacy Extensions for Stateless Address Autoconfiguration in IPv6. This was determined by examining and adjusting Privacy Extensions-related settings in the targeted operating system, generating ICMPv6 Ping traffic from the targeted system, and observing the Source IPv6 addresses used as the packets traversed the network.

The network utilized in this section was exactly the same as the one used in the Multiple IPv6 Addresses portion of the survey, with a virtualized or physical target host, testing workstation, Ethernet hub, IPv6 router, and a separate segment containing virtualized servers.
The topology of this network is shown in Figure 6.2.

The testing in this scenario attempted to answer the following questions:

1. Are Privacy Extensions for IPv6 supported?
2. Are Privacy Extensions for IPv6 enabled by default?
3. What procedure is used to activate/deactivate the use of Privacy Extensions for IPv6?
4. What are the default parameters?
5. What parameters can be changed?
6. Are Temporary addresses created for each network prefix and scope?
7. Are the Temporary addresses used for all outbound traffic on all scopes?

6.2 Survey Observations

While each of the examined operating systems appeared to correctly implement the IPv6 and NDP protocols as outlined by IETF RFCs 4291, 4861, and 4941, each implementation had its own unique set of quirks in certain aspects not specifically mandated by the standards. Once each of these three tests was completed, no two OS's had performed exactly the same. In this section I provide a brief summary of some of my observations; detailed results for each OS are provided in Section 11.1.

6.2.1 NDP Behavior Test

During the NDP Behavior tests, I observed several instances in which host behavior differed from the recommendations (but not requirements) of IETF RFC4861. In each case, the discrepancy did not affect the protocol's operation or receiving host's ability to understand the NDP packet. When sending both unicast and multicast NDP Solicitations, each system used a Source address within the Target's subnet (as recommended) except for the CentOS and Ubuntu Linux hosts, which used their link-local address for all unicast Solicitations (but not multicast Solicitations). Additionally, the Windows Server 2003 host also appeared to use its link-local address as the Solicitation source when performing neighbor discovery after an upper-layer connection had failed.

When sending NDP Advertisements, there were also minor discrepancies in certain parts of the packets. On all systems in the Microsoft Windows family, every Advertisement packet had both the Solicited and Overwrite flags enabled. On the other hand, all other systems had both of these flags enabled when responding to multicast Solicitations, but only the Solicited flag
enabled when responding to a unicast Solicitation. Similarly, each Windows system included the Target Link Layer Address option in each of its Advertisements, while others included it only when replying to a multicast Solicitation. Additionally, each system used the address requested in the incoming Solicitation as the Advertisement Source address, except for Mac OS X which used its link-local address instead.

When handling incoming NDP Solicitations, the only two systems that behaved differently from the rest were Microsoft's Window XP and Windows Server 2003. After receiving the incoming packet, every system added the Solicitation's Source Link Layer Address to its own NDP cache table with a state of INCOMPLETE or STALE, as prescribed by the RFC. However, each system except for the aforementioned two immediately sent its own Solicitation in an attempt to complete the table entry, even when it had no actual data to send to the other system. Similarly, when receiving a Solicitation with no Source Link Layer Address option specified, every system except for Windows XP and Windows Server 2003 accepted the packet, performed its own neighbor discovery to find the other host's MAC address, and then sent its Advertisement with the newly discovered address. In the case of these two systems, the incorrectly formatted NDP packet was simply ignored.

In each of these cases, the observed discrepancy did not affect the protocol's operation or the receiving host's ability to understand the NDP packet. The instances in which these systems differed in behavior appeared to be limited to areas of the RFCs in which the terms “SHOULD” or “MAY” were used in place of “MUST”.

6.2.2 Multiple IPv6 Addresses Test

Several differences between operating systems' IPv6 protocol implementations were also discovered during the Multiple IPv6 Addresses tests. The first and possibly most significant of these is the autoconfigured addresses used by Microsoft's Windows 7 and Windows Server 2008 systems. Rather than basing these autoconfigured addresses on the interface's MAC address (as specified by IETF RFC4862 [42]), these two systems generated a separate pseudorandom interface identifier for use as the basis of the calculated address, similar to how Privacy Extensions operate. This pseudorandom identifier did persist through a system reboot, but it is unknown whether this value is permanent or on a long-term rotation. Although this behavior is enabled by default, the Windows 7 system can be configured to use the Address
Autoconfiguration procedures outlined in IETF RFC4862 with the netsh utility. Additionally, the Windows 7 system was observed to occasionally use its Unique Local-scope address as the Source address in traffic destined to a link-local address. However, this behavior could not be consistently replicated.

Finally, when sending traffic all systems, except for the CentOS and Ubuntu Linux hosts, appeared to use their own IP address that was numerically closest to the target's IP address as the packet Source. For example, consider a host with addresses in the 2001::/16 and FC00::/8 ranges. When trying to reach 9999::1 the host's address in the FC00::/8 range was used as the Source. Even though packets generated from the FC00::/8 address would not be able to leave the organization's network, it was still chosen as the Source since it is numerically closer to 9999::1 than an address starting with 2001::. While this should not cause any issues with current IPv6 deployments, potential problems could arise as address utilization expands. For example, if the 8000::/8 range or higher was ever allocated for use as Global-scope addresses, Microsoft Windows and MAC OS X hosts might attempt to reach these with their FC00:: or FE80:: addresses. Since these addresses correspond to the Unique Local and Link Local scopes respectively, such scope-bound packets would never reach their target. The CentOS and Ubuntu systems, on the other hand, used their Global scope address to reach all IP addresses except those in the FC00::/8 (Unique Local) and FE80::/16 (Link Local) ranges. In such cases, the appropriate addresses within those ranges were used.

### 6.2.3 Privacy Extensions Test

The largest difference between systems regarding their implementation of Privacy Extensions for Stateless Address Autoconfiguration was whether or not this feature was enabled by default when the IPv6 protocol was activated. While each of the tested systems provided support for this feature set, only the desktop family of Microsoft Windows systems (Windows XP and 7) had the extensions enabled by default. Each of these systems appeared to have the same set of configurable Privacy Extensions parameters (which could only be modified through command-line utilities in all cases) and used the default Preferred and Valid lifetimes of 1 and 7 days, respectively, as recommended by IETF RFC4941.
7. Implementation

In this section, I will discuss the details of implementing each module in the IPAC system, as well as the configuration several network management mechanisms it requires for proper operation.

7.1 Agent Module

The Agent process consists of ten separate threads operating in parallel. Each thread provides a vital function to the operation of the IPAC Agent system. In this section, I detail the operations and describe some of the design considerations of each thread (and one additional class used by the threads). Communication between these threads is illustrated in Figure 7.1.

This module is composed of five separate Python script files. agent.py contains the body of the main thread, ipac_threads.py contains the bodies of all other threads called by the main thread, ipac_daemon.py contains the code used to daemonize the Agent process on the host system, and ipac_functions.py contains several common functions shared by these scripts. Customization of the Agent process is provided by the ipac_agent_config.py file. This file declares each of the constants for Agent setup, operation, and debugging imported by the other scripts. Since the service does create a raw packet listening socket on the host system, privileged access is required to run this module.
7.1.1 Main Thread

The main Agent thread initializes all other threads required for IPAC Agent operation. This thread also creates the packet capturing interface and handles all incoming ICMPv6 packets. Each NDP Solicitation or Advertisement packet (if not sent from the Agent host) will have its IPv6 Source, IPv6 Destination, and ICMPv6 Target addresses processed, while all other types of ICMPv6 packets are discarded.

For each incoming NDP Solicitation packet:

If the packet is from the Agent host (Source MAC address is the system's own address), then the packet is ignored. Otherwise, if the packet is destined to the Agent host (Destination MAC is the system's own), the Source IPv6 address will be processed (address processing is discussed below). If the packet is neither from nor to the Agent host, the Source IPv6 address is
Correlating IPv6 Addresses for Network Situational Awareness  Jason Froehlich

processed and the Target IPv6 address is checked. If the Target IPv6 address is the Agent host's own (meaning the packet was a multicast Solicitation to the Agent), the rest of the packet is ignored. Otherwise, the packet's Target IPv6 address is processed, as well as the Destination IPv6 address (if it differs from the Target and is not a multicast address).

**For each incoming NDP Advertisement packet:**

If the Target Link Layer Address (TLLA) option is included in the ICMPv6 packet, it is verified against the packet's Source MAC address. Any inconsistencies will result in a warning. Otherwise (no TLLA option is included), the Source MAC address is used as the TLLA. If the packet is from the Agent host (Source MAC address is the system's own address), the packet is ignored. Otherwise, if the packet is not destined to the Agent host (Destination MAC is not the system's own), then the Source and Target IPv6 addresses are processed using the procedure outlined in the “Source Processing” section below, and the Destination IPv6 address is processed as discussed in the “Target and Destination Processing” section. If the packet is destined to for the Agent host (Destination MAC is the system's own), then the Agent will check its list of addresses awaiting Advertisements (maintained by the ReplyWaiting thread) for the Target IPv6 and TLLA addresses. If an Advertisement is expected from this IP-MAC address pair, the addresses will be added to the list of pending Advertisements (maintained by the PendingProcessor thread) for further processing. Additionally, if the Source IPv6 address differs from the Target IPv6 address, the Source address is processed as discussed below. If the Advertisement is not expected, the Source and Target IPv6 addresses are processed using the procedure outlined in the next section.

**Source Processing:**

For each observed Source IPv6 address, the system first checks to see if it exists in the system's list of active IP addresses. If the IP address is not already known to be active, a unicast Solicitation is sent to the observed IPv6 and MAC Source addresses for verification. On the other hand, if the IP address is already known to be active, the associated known MAC address is checked against the observed Source MAC address. If these MAC addresses match, the Source address is accepted as genuine; a sighting record is created for the IP-MAC pair and the last-seen time for the IP address is updated in the active list. Otherwise (the MAC address does not match what is on record), a multicast Solicitation is sent to verify which addresses are still active on the
network. Any conflicts that are discovered are handled by the PendingProcessor thread.

**Target and Destination Processing:**

For each valid Target and Destination IPv6 address, the system will generate a multicast NDP Solicitation. Unlike with Source addresses, the system does not attempt to determine if the address is already known to be active. The reasoning behind this is that the system should have the same view of the network as the host that sent the Solicitation. Since the Solicitation could potentially result in unicast Advertisements from more than one host (some of which may be presently unknown to the system), the Agent should always send its own multicast packet to ensure that all addresses are known.

The program does make an attempt to throttle the multicast Solicitations it sends to prevent overloading network and host resources. When a multicast Solicitation is needed, the system will check its record of recently sent multicast Solicitations for the Target/Destination address. If the address was the target of a multicast Solicitation sent in the last MC_SOLICIT_DUPLICATE_INTERVAL seconds (default: 5), then the new Solicitation will not be sent. This prevents the generation of extraneous multicast traffic in certain situations, such as when the system observes another host continuously soliciting for a non-existent IPv6 address. However, this could result in a scenario in which an address goes unseen for a certain amount of time. If a host begins using the specified address during the MC_SOLICIT_DUPLICATE_INTERVAL time window, it will not be confirmed by the system until it is seen in the Source address of an NDP Solicitation or the next time it is seen as the Target/Destination of a Solicitation after the threshold window has passed. As evidenced by the results of the IPv6 Implementation Survey (Section 6.2), the former should occur on most systems as soon as the system has data it needs to send from that address.

**7.1.2 ReplyWaiting Thread**

The ReplyWaiting thread maintains a list of IP and MAC addresses from which the Agent expects to receive an NDP Advertisement. This list is used to differentiate between incoming NDP Advertisements destined to the Agent node as a result of IPAC-generated Solicitations and other unsolicited Advertisements. To prevent the poisoning of the Agent’s records, any Advertisement that is not expected will not be considered valid. Although several threads need to retrieve data from and modify the contents of this list, direct access is only available to the
Correlating IPv6 Addresses for Network Situational Awareness

Jason Froehlich

ReplyWaiting thread. Other threads are able to access the list through a set of commands. Incoming access/modification commands are queued by the ReplyWaiting thread and executed sequentially to prevent any data integrity issues that might arise when the list is accessed or modified simultaneously by more than one thread.

The ReplyWaiting list consists of a dictionary with an IP and MAC address pair as the key, and a tuple containing a timestamp, reply count, and timeout count as the value. The timestamp identifies the last time a Solicitation was sent to the IP/MAC pair, the reply count records the number of Advertisements received from the IP/MAC pair since the last Solicitation was sent, and the timeout count records the number of times a Solicitation has been resent after receiving no Advertisement within the timeout threshold. When a multicast Solicitation is sent, the MAC address is recorded as 'NULL'; incoming Advertisements cause the system to check the list for both the IP-MAC and IP-'NULL' pairs.

Other threads interact with the ReplyWaiting thread and its internal list by calling the thread's action function and passing a command string (optionally with additional parameters for the command). The action function adds the command and any options to an internal Queue. When the command is processed by the body of the ReplyWaiting thread, any results are passed to the function through a results Queue and returned to the calling thread. To prevent results from being returned to the wrong thread when multiple threads interface with the ReplyWaiting thread simultaneously, the commands and results are tracked and organized by a timestamp value created when the action function was called.

The body of the ReplyWaiting thread monitors the incoming command queue and processes the commands serially. The 'add' command will cause an IP/MAC pair to be added to the list with null data values (or overwritten if the pair already exists), the 'get' command will return the data values for the given IP/MAC pair, the 'getall' will return the contents of the entire list, and the 'haskey' command will check the list for the given IP/MAC pair. The 'incomingAdv' command is used by the main thread when an Advertisement destined for the Agent system is observed. With this command, the thread will check the list for the given IP/MAC and IP-'NULL' pairs. If either is found, the entry's timestamp is checked to see if it is within the SOLICIT_TIMEOUT_INTERVAL age threshold. If the entry has not timed out, the reply counter is incremented and a value of 'True' is returned to the main thread to signal that
the Advertisement is expected. The 'solResend' command is used by the SolicitationResender thread to identify the IP/MAC address pairs that have timed out without having received any Advertisement. With this command, the thread iterates the list and checks each entry for these conditions. If the entry is a match, the IP/MAC address pair is added to a list that will be returned to the calling thread. If the entry has reached the maximum number of retries specified by SOLICIT_RETRY_COUNT, it is removed from the list.

7.1.3 PendingProcessor Thread

The PendingProcessor thread maintains a list of IP/MAC address pairs from which an NDP Advertisement was recently received by the main Agent thread. This list of pending addresses allows the Agent to recognize scenarios in which more than one MAC address claims to 'own' the same IP address. Only after the SOLICIT_TIMEOUT_INTERVAL has passed (at which point all Advertisements that should be coming in will already have arrived) are the IP/MAC address pairs added to the database sighting records and Active IPs list.

When the main thread receives an expected NDP Advertisement, it informs the PendingProcessor thread through the add function. This function adds the IP address, MAC address, and timestamp values provided by the main thread to a queue. The body of the PendingProcessor thread periodically checks this queue and adds new addresses to its internal list. Once the list has been updated, the thread will iterate the list and examine each IP and MAC address. For each IP address that has reached its SOLICIT_TIMEOUT_INTERVAL, if it has only one associated MAC address, a record is created and the Active IP list updated with the new timestamp and a retry count of zero (or added if the IP/MAC doesn't already exist). If the IP has more than one associated MAC address, a record is created for each MAC, both are added to/updated in the Active IP list, and an alert is raised to warn of the conflict.

7.1.4 SolicitationResender Thread

The SolicitationResender thread handles the retransmission of NDP Solicitations to IP addresses for which an NDP Advertisement was expected but not received within the timeout threshold. This reduces the chances of the Agent process missing an IP address in instances when a Solicitation or Advertisement packet did not reach its destination due to link congestion. The body of the SolicitationResender thread will periodically query the ReplyWaiting thread for
a list of IP/MAC address pairs for which no reply was received within the timeout period. For each resulting IP/MAC pair, the SolicitationResender thread will direct the NDUnicastGenerator or (in cases where the MAC is 'NULL') NDMulticastGenerator threads to generate a new Solicitation to the Target IP address.

7.1.5 ActiveIPRefresher Thread

The ActiveIPRefresher thread allows the Agent system to verify that a previously observed IP/MAC address pair is still active on the network even when it is not continuously generating NDP traffic. If the Agent has not observed any NDP traffic from the IP/MAC pair in question during the last address aging interval, it will generate a unicast Solicitation, causing an Advertisement to return from the target.

The ActiveIPRefresher thread periodically queries the Agent database (through the DatabaseInterface thread) for IP/MAC address pairs in the 'active' table not seen in the last ACTIVE_IP_REFRESH_INTERVAL. If the IP/MAC pair has reached the ACTIVE_IP_REFRESH_COUNT threshold without the Agent receiving any Advertisements in return, the pair is removed from the 'active' database table and a record is generate to mark its termination. Otherwise, a unicast Solicitation is generated (through the NDUnicastGenerator thread) and the associated entry in the 'active' database table is updated to reflect the incremented retry count.

7.1.6 NDMulticastGenerator Thread

The NDMulticastGenerator thread handles the generation of all multicast NDP Solicitations sent by the Agent. Other threads interact with this thread through the generate function, which accepts the IP address of the target as a parameter. This function adds the address to an internal queue, which is continuously monitored by the body of the thread. For each address, it will construct an ICMPv6 NDP Solicitation packet with the help of the PacketConstructor class, establish a packet injection socket on the host's network interface, and inject the crafted packet onto the network. Before the packet is actually sent, the thread will add the target to the Reply Waiting list via the ReplyWaiting thread (it is done in this order to prevent the resulting Advertisement from arriving before it is expected). The body of the thread will then generate a timestamp that marks when the packet was sent and add it to an internal results queue,
causing the generate function to return the value to the calling thread.

7.1.7 NDUnicastGenerator Thread

The NDUnicastGenerator thread handles the generation of all unicast NDP Solicitations sent by the Agent. Other threads interact with this thread through the `generate` function, which accepts the IP and MAC addresses of the target as a parameter. This function adds the addresses to an internal queue, which is continuously monitored by the body of the thread. For each address, it will construct an ICMPv6 NDP Solicitation packet with the help of the PacketConstructor class, establish a packet injection socket on the system's network interface, and inject the crafted packet onto the network. Before the packet is actually sent, the thread will add the target to the Reply Waiting list via the ReplyWaiting thread (it is done in this order to prevent the resulting Advertisement from arriving before it is expected). The body of the thread will then generate a timestamp that marks when the packet was sent and add it to an internal results queue, causing the generate function to return the value to the calling thread.

7.1.8 DatabaseInterface Thread

The DatabaseInterface thread acts as a singular interface to the Agent's MySQL database. Although several threads have the need to obtain and modify records in the database, all queries and updates to the database tables are performed by the DatabaseInterface thread. This prevents any data integrity issues that might arise when multiple threads attempt to read or write simultaneously.

Upon initialization, the DatabaseInterface thread will establish a connection to the Agent database. If the FRESH_START testing mode is enabled, it will remove all existing records from the 'active' and 'records' database tables in preparation for new data. Other threads interact with the DatabaseInterface thread through the `execute` function. This function takes a text string containing a set of SQL commands as a parameter and adds the item to an internal Queue. The body of the DatabaseInterface thread continuously monitors the queue for incoming commands. Upon the receipt of a new item, it will execute the SQL command, obtain any results from the database, commit any changes to the database, and add any results to an internal results queue. The results queue is accessed by the `execute` function, which will return the results to the calling thread. If an exception occurs when executing the SQL commands (such as a
duplicate primary key), the exception is caught and an empty set will be returned instead.

7.1.9 QueuedPrinter Thread

The QueuedPrinter thread acts as a singular interface for writing messages to the Agent's output and error log. When multiple threads running in parallel use the same output file for printed text, it is possible (and very likely) that two or more will attempt to print text at the same time. When this occurs, the text of these messages will become interwoven, producing illegible text. The QueuedPrinter thread resolves this issue by queuing messages that are to be appended to the output log and writing them one at a time.

Other threads relay text to the QueuedPrinter through the `printq` function. This function takes the message text (and optionally the name of the originating thread for debugging purposes) as a parameter, and adds a tuple of these values plus a timestamp to an internal Queue. The body of the QueuedPrinter continuously monitors the queue for incoming messages and prints them individually. If the PRINT_DEBUGGING mode is enabled, the name of the originating thread and timestamp will also be added to the output file.

7.1.10 LogMarker Thread

The LogMarker thread generates regular MARK messages in the Agent's output and error log. These messages serve to inform the user that the Agent process continues to run when there may be no other messages within the log. In addition to the current time, these messages also include a count of the incoming NDP Solicitations and Advertisements processed by the Agent since the last MARK. These log messages are generated once every LOG_MARK_INTERVAL (default: 15 minutes), and are added to the Agent's log through the QueuedPrinter thread.

7.1.11 PacketConstructor Class

The PacketConstructor class contains a set of functions that assemble NDP Solicitation packets in preparation for their injection onto the network. The `unicastNDSol` and `multicastNDSol` functions are called by the NDUncastGenerator and NDMulticastGenerator threads, respectively. A third function, completePkt, is called by the other two to finalize the raw Ethernet frame.

The `unicastNDSol` function takes Source and Target IP and MAC addresses as parameters and converts these ASCII values into binary form for insertion into the raw packet
data. With the help of the dpkt Python module it constructs the ICMPv6 portion of the packet, using the Target IP as the Target value and the Source MAC as the Source Link Layer Address option value. The `completePkt` function is then called to construct the remainder of the packet, and then the entire packet is returned to the calling thread.

The `multicastNDSol` function takes Source IP, Target IP, and Source MAC addresses as parameters and converts these ASCII values into binary form for insertion into the raw packet data. Multicast Destination IP and MAC addresses are constructed based on the given Target IP address. With the help of the dpkt Python module, it constructs the ICMPv6 portion of the packet, using the Target IP as the Target value and the Source MAC as the Source Link Layer Address option value. The `completePkt` function is then called to construct the remainder of the packet, and then the entire packet is returned to the calling thread.

The `completePkt` function takes the ICMPv6 datagram generated by `unicastNDSol` or `multicastNDSol` as a parameter, as well as the binary form of the Source and Destination IP and MAC addresses. With the help of the dpkt Python module, it creates an IPv6 packet with the ICMPv6 datagram as its payload, and an Ethernet frame with the IPv6 packet as its payload. The finalized Ethernet frame is returned to the calling function, which will return it to the originating thread for injection onto the network.

### 7.2 Trap Handler Module

The Trap Handler process consists of a single thread that continuously listens for SNMP trap messages. This module is composed of two separate Python script files; `ipac_traphandler.py` contains the body of the process, while `ipac_daemon.py` contains the code used to daemonize the Trap Handler process on the host system. Customization of this process is done through constants initialized at the beginning of the `ipac_traphandler.py` script. The values of these constants determine the Trap Handler's initial actions, the database it uses for storage, and its debugging operations. Since the service does create a socket within the restricted range of UDP ports, privileged access is required to run this module.

The process starts by establishing a listening socket on UDP port 162, the default destination for SNMP traps and notifications. The socket functionality is provided by the PySNMP Python module. Each SNMP packet received by the socket, as well as each Variable
Binding pair contained within the packet, is processed individually. If the variable binding pair includes either a MAC Learned or MAC Removed notification, an appropriate record is created in the Trap Handler database's 'mac_notifications table'.

This module provides two special modes useful for testing and debugging the Trap Handler's operation: FRESH_START and TEST_MODE. When the FRESH_START mode is enabled, the Trap Handler process will purge any existing records from its database upon its initialization. This provides a clean slate for each testing session, and the Trap Handler operates as normal. TEST_MODE, on the other hand, processes each trap notification as normal but prevents any modification to the module's database. This allows us to debug the module's operation without affecting any existing data.

### 7.3 Server Module

The Server process consists of a single thread that periodically summarizes and cleans up the databases records created by the Agent and Trap Handler processes. This module is composed of four separate Python script files. ipac_server.py contains the body of the Agent record processing code, ipac_trapsum.py contains the body of the Trap Handler record processing code, ipac_daemon.py contains the code used to daemonize the Server process on the host system, and ipac_server_config.py contains a set of constants used to customize the operation of the Server module. The values of these constants determine the Server's initial actions, processing behavior, the database it uses for storage, the database from which it retrieves SNMP trap records, and its debugging operations. In this section, the term “session” is used to refer to the interval of time in which an IP and MAC address pair (IP-MAC) or MAC address and physical switch port pair (MAC-Port) was known to exist on the network.

The Server begins by obtaining a list of known Agents from its own database. For each Agent marked as active, it will query the corresponding Agent database for its IP-MAC sighting records. Once these records are obtained, they are deleted from the Agent database to prevent reprocessing old data in subsequent iterations of the Server process. It then iterates through the list of obtained records (sorted by the IPv6 address and then timestamp) with a goal of reducing a multitude of sighting records for the same IP and MAC address session to a single record that indicates the first and last times the pair was seen. Two adjacent Agent sighting records are considered to be a part of the same session if they contain the same IP and MAC values and have
a timestamp delta that is not larger than the allowed Timeout value. If a termination record is seen, or if the timeout threshold is reached between records, the session is considered to be ended. In this case, a summarization record is created in the Server's database, and processing of a new session begins again with the next Agent record. Each record created by the Server has an 'Ending Type' value that indicates whether the MAC-IP binding session is considered terminated or ongoing. An ending type value of 20 indicates that a termination record was seen and the pair had left the network, while a value of 10 indicates that the pair was still active at the last time the Agent updated its database. If the Server iterates through all of the entries with the same IP and MAC addresses without seeing a final termination record, the session is considered to be ongoing and a record is created in the Server's database with the ending type value of 10.

Despite its violation of the protocol standards, there may be instances in which the same IPv6 address is seen as paired with multiple MAC addresses simultaneously, as discussed in Section 5.3.2. The Server identifies instances of this by seeing consecutive records with the same IPv6 address, different MAC addresses, and a timestamp delta less than the value of the timeout threshold. In such cases, it will treat each conflicting pair individually, with each receiving its own summary record. However, the records identifying each of these pairs will be intertwined in the results obtained from the Agent database, making it difficult to identify the actual beginning and end of each session. In order to resolve this, each time the Server identifies a conflict it will move the second pair's records to a secondary list and temporarily ignore them, continuing to process the original pair's records as normal. Once all of the remaining records have been processed (and any other conflicts resolved), the Server will return to this secondary list, using the same procedure to summarize sessions and identify conflicts. The Server will continue to iterate through this list until all conflicting pairs have been processed.

Once all Agent records have been processed, the same procedures are used to process Trap Handler records. However, because Trap Handler records are generated only when the MAC-port pair is first and last seen (rather than every 30 seconds as with Agent records), each record is processed individually. For each record obtained from the Trap Handler's database, the Server determines whether it signifies an instance when a MAC address was learned by the switch or removed from the switch. If it is a 'Removed' record, the Server checks its database to see if a corresponding unclosed MAC-Port session already exists. If it does, that record is
updated with the proper ending time and closed; otherwise a new session record is both created and closed. If it is a 'Learned' record, the Server again checks its database to see if an unclosed session already exists for that MAC and port. If no unclosed record exists, a new session record is created and left unclosed (without an ending time). Otherwise, the existing record is terminated with an error code (as no Removed record was received) before a new record is created.

To prevent overloading the Server host or the Agent databases, the Server process implements a waiting period of 15 minutes between each record acquisition and processing iteration.

Like the Trap Handler, this module provides two special modes for testing and debugging its operation: FRESH_START and TEST_MODE. Just as before, the FRESH_START mode causes the Server to purge all existing binding and trapBinding records from its own database, leaving a fresh environment for the testing session. However, TEST_MODE on the Server operates differently. Instead of preventing records from being written to its own database, the Server will prevent the deletion of records it retrieves from Agent and Trap Handler databases. This allows the records to be reprocessed during each iteration of the Server, obviating the need to repopulate the Agent and Trap Handler databases between testing sessions.

### 7.4 Interface Scripts

The final module consists of a set of scripts that provides an administrator or investigator with an interface for obtaining addressing information from the IPAC system. These four scripts each provide a different view of the stored data: one provides a tool for querying the system’s raw summary records, another utilizes a visualization tool to create a visual representation of the IPv6 addressing timeline, and the last two process text-based application logs and binary packet captures and provide detailed information about each identified IPv6 address. Within the output of each script, physical switch port identifiers consist of the switch’s IPv4 address, port number, and VLAN ID (e.g., “192.168.0.20/0003/0010” would be VLAN 10 on Port 3 on the switch with IPv4 address of 192.168.0.20).

#### 7.4.1 Database Query Interface

The ipac_query.py script provides several methods of obtaining specific information from
the IPAC Server database via a command line utility. This tool can perform variations of five different operations: listing all IPv6 addresses used on the network, listing all MAC addresses used on the network, listing all physical switch ports used by a given IPv6 or MAC address, listing all IPv6 and/or MAC addresses currently in use on the network, or listing all instances in which an IPv6-MAC address conflict occurred. Each of these operations (except for the listing of currently active addresses) can be limited to a specific point in time, or a starting and/or ending times (matches records ending after and beginning before the specified times, respectively). Additionally, the operation that identifies all IPv6 addresses used on the network can be constrained to those paired with a specific MAC address, and the operation that finds all MAC addresses on the network can be constrained to those paired with a specific IPv6 address.

Operations and limiting parameters are specified by the user when running this script from a terminal. These options are shown in the tool's usage guide provided in Figure 7.2, and an example of its output can be seen in Figure 7.3.

```
Usage: python ipac_query.py --agent [Agent ID] {Operations} {Options}
Operations:
  --findips : Find all IPs in use, constricted by --mac
  --findmacs : Find all MACs in use, constrained by --ip
  --findport : Find ports used by given IP or MAC
  --now [ip|mac| ]: Find all IPs and/or MACs currently in use
  --conflicts : Find times when multiple MACs used the same IPv6 address
Options:
  --ip [IPv6 Address] : Find records with this IPv6 Address
  --mac [MAC Address] : Find records with this MAC address
  --time [Time] : Exact time to find records for
  --start [Start Time] : Find records existing after this time
  --end [End Time] : Find records existing before this time
  -b : Brief output
  -v : Verbose output (includes timestamps when relevant)
Time Format: 'Year-Month-Day_Hour-Min-Sec' Ex: '2011-02-14_16:32:45'
```

Figure 7.2. `ipac_query.py` Usage Guide.
Correlating IPv6 Addresses for Network Situational Awareness

Jason Froehlich

| 00:0C:29:39:2B:49 |
| FE80:0000:0000:0000:020C:29FF:FE39:4C39 |
| 00:0C:29:C1:A4:70 |
| 2001:0DB8:1111:1111:0000:0000:0000:0001 |
| FC00:AAAA:AAAA:1111:0000:0000:0000:0001 |
| FE80:0000:0000:0000:020C:29FF:FE31:A470 |
| 00:17:31:B8:E6:58 |
| FC00:AAAA:AAAA:1111:0217:31FF:FEB8:F21C |
| FE80:0000:0000:0000:0217:31FF:FEB8:F21C |
| 00:60:08:27:4A:FC |
| FC00:AAAA:AAAA:1111:0260:08FF:FEC7:4BFC |
| FE80:0000:0000:0000:0260:08FF:FEC7:4BFC |
| 40:61:86:C7:DA:24 |
| FE80:0000:0000:0000:4261:86FF:FEC7:3B4C |

Figure 7.3. Example ipac_query.py script output

7.4.2 Interactive Timeline Graph

The ipac_graph.py script provides network operators with an interactive visual timeline of IPv6 address utilization on the organization's networks. Based on the BrokenBarH graph provided by the matplotlib Python module, this dynamic graph represents each IP-MAC session obtained from the IPAC Server database as a separate timeline segment. The graph's Y-axis identifies each IPv6 address observed on the network, while the X-axis represents time. Each segment is marked with the MAC address associated with the session, as well as a red or green tab indicating whether the session has ended (red) or is still considered to be active (green). Any IP-MAC conflicts (overlapping segments) are identified with a red underlining bar and an orange starburst. The graph is interactive in that it allows the user to pan and zoom along the timeline in order to view the information from different time scales. Additionally, it allows the user to click on a segment to reveal the session's exact start and ending times. A screenshot of this graph is provided in Figure 8.9 in Section 8.3.

7.4.3 Text Log Processing

The ipac_textlog.py provides a tool that allows for the automated processing of text-based log files containing IPv6 addresses. For each IPv6 address identified in the log file, this script will provide information about the MAC addresses and switch ports associated with the address at the time identified by the log entry's timestamp (if available in the IPAC Server's database records). The ipac_textlog.py script utilizes regular expressions to locate IPv6
addresses and timestamps within each log entry. While it is capable of finding IPv6 addresses in any log format, it currently only has the ability to identify timestamps within files using the Syslog format, as well as Apache access and error logs; additional log formats can be accommodated by adding the proper regular expression to match their timestamp structures.

After locating the proper IPv6 addresses and timestamp in each log entry, the script must decide which Agent records it should query to find the proper MAC address information. For any Link-Local scope addresses, it will use the Agent of the network from which the log file originated (requested from the user when the script initializes). For all other address types, it uses the IPAC Server database's list of known subnets to identify the proper Agent to query. If the address does not match any subnet prefix stored by the Server, the script will not know which set of records to query and will not attempt any further processing of the address.

Once the associated MAC addresses and physical switch ports have been identified, a summary of this information is appended to the log entry and output to a separate file, encapsulated by `<[ and ]>` brackets to delimit it from the log's original content. Because of the system's timing of address identification, as discussed in Section 5.3.2.1, each MAC address and port ID is marked with a sequence of asterisks or carets to show the system's confidence in the identifier. This confidence rating is based on whether duplicates were found (more than one MAC/Port for the queried IP/MAC) and whether the result was an exact match or was within a ±30 second window of the given timestamp (referred to as a “close” match). The meanings of these sequences are explained in Figure 7.4, and an example of the script's output can be seen in Figures 8.5 and 8.6 in Section 8.3.

| ** = No Duplicates, Exact match |
| *** = Duplicates, Exact Match |
| ** = Duplicates, More Than One Exact Match |
| * = All Are Close Matches (May Be Only One) |
| [none] = Duplicates, Close Match |

Figure 7.4. Confidence Indicators for MAC addresses in Text Logs Reports. (Values also apply to the '^^' marks for Switch Port identifiers)

### 7.4.4 Libpcap File Processing

The final Interface script, `ipac_pcaplog.py`, provides a function similar to the previous text log processor, but for binary packet capture files in the Libpcap format. This utility utilizes the dpkt Python module to disassemble and processes each packet in the provided capture file.
For each IPv6 packet in the file, the script will attempt to correlate its Source and Destination IPv6 addresses (except any Loopback or Multicast addresses) to a MAC address and physical switch port by querying the Server database. In the current version of the script, no attempt is made to identify any IPv6 addresses existing beyond the IP header. Just as with the previous interface script, determining which Agent records to query is accomplished by examining the list of known subnets in the Server database. Rather than appending to the packet capture file, all summaries are written to a separate text report and identified by the number of the corresponding packet in the original file. An example of the script's report output can be seen in Figure 8.8 in Section 8.3.

7.5 **Databases**

The IPAC system uses three different types of databases for storing records and processed data. Hosts implementing the Agent module use a database to store information about IP addresses the Agent considers as currently active on its network, as well as a historical recording of each instance in which an IP address is seen in NDP traffic. Trap Handlers utilize a database for storing each MAC Learned or MAC Removed notification from an Ethernet switch. The Server module uses a database to hold information about the IPAC infrastructure and summaries of the Agent's and Trap Handler's records. In the demonstration network, each host utilizes its own local MySQL server. The structure and implementation of these databases are discussed further in Section 11.2.

7.6 **Time Synchronization**

In the demonstration network, the centralized NTP service is provided by the IPAC Server host, which keeps its clock synchronized with a Stratum 1 or Stratum 2 time source at pool.ntp.org. Sample NTP configuration files and commands for ntpd servers, ntpd clients, and Cisco IOS clients are provided in Figures 7.5, 7.6, and 7.7, respectively.

```
driftfile /var/lib/ntp/ntp.drift
server pool.ntp.org
restrict -4 default kod notrap nomodIFY nopeer
restrict -6 default kod notrap nomodIFY nopeer
restrict 127.0.0.1
restrict ::1
restrict 10.100.111.0 mask 255.255.255.0
```

*Figure 7.5. Sample Linux NTP Server Configuration File*
7.7 Use of IPv4 and IPv6

The IPAC system and demonstration network utilize the IPv6 protocol for most inter-node communication. However, there are two portions of the system that currently require the use of the IPv4 protocol. The first place in which IPv4 is required is the scripts' connections to their respective databases. Even though the MySQL servers implemented on this network fully support remote access over IPv6, current versions of the MySQLdb Python module can only establish connections to IPv4 addresses. Thus, the only method of accessing the databases without heavy modification of the MySQLdb module is to connect to the server's IPv4 address.

The other area in which IPv4 use is currently required is the handling of SNMP traps. Although the SNMP protocol is fully capable of transporting traps and notifications over IPv6 and the PySNMP Python module is able to listen for notifications on IPv6 addresses, the Ethernet switching hardware that was available to me for testing did not support any IPv6 functionality. This issue is also likely to be faced by many organizations still utilizing older hardware in their network infrastructure. Cisco switching hardware did not begin to provide IPv6 support until the release of the Catalyst 2960 and 3560 families of Ethernet switches running IOS releases 12.0(22) or 12.2.(2) and later (actual IOS versions providing IPv6 functionality vary between release branches). In such scenarios where IPv6 management capabilities are not provided by the network hardware, the operations of the NTP clock synchronization mechanism must also occur over IPv4.
7.8 MAC Notifications

In order for the IPAC system to know which physical ports are used by each MAC address, each access-layer Ethernet switch on the organization's network must be configured to provide the system with information about each MAC addresses it knows. The mechanism that the IPAC uses to accomplish this is SNMP CAM table event notifications, as discussed in Section 5.2.2. Enterprise-grade Ethernet switches from many different vendors provide this functionality, but in this demonstration system I have focused only on Cisco's Catalyst series of switches running IOS 12.0 and later. A sample configuration containing the commands related to enabling these MAC address notifications on a Cisco Catalyst 2950 switch running IOS release 12.1(22)EA14 is provided in Figure 7.8.

```plaintext
!--- Enable MAC Notification on all non-uplink/non-trunk ports
interface FastEthernet0/1
   name Uplink
!
interface FastEthernet0/2
   snmp trap mac-notification added
   snmp trap mac-notification removed
!
interface FastEthernet0/3
   snmp trap mac-notification added
   snmp trap mac-notification removed
!--- Rest of interfaces omitted
...
!
interface Vlan1
   ip address 10.100.111.12 255.255.255.0
   no ip route-cache
!
   snmp-server community public RO
   snmp-server enable traps MAC-Notification
   snmp-server host 10.100.111.114 public
!
!--- Send notifications immediately after event
   mac-address-table notification interval 0
   mac-address-table notification
!--- Inactive MACs removed from CAM table after 60 seconds
   mac-address-table aging-time 60
!
end
```

Figure 7.8. Sample SNMP Notification Configuration for a Cisco Catalyst 2950 Switch.

It is important to ensure that only switch ports providing direct access to end devices are configured to send MAC address-related notifications. If an uplink port were configured to send these updates, the switch would inform the IPAC system of MAC addresses connected to other switches, causing duplicate messages. Since we are only concerned with finding the physical
port to which the device is directly connected, messages from these uplink ports would compromise the accuracy of the system. However, there may be instances in which a port connecting to another switch would actually need to be configured to send these notifications. In cases where the second device is not capable of sending its own CAM table updates to the IPAC system, such a configuration would provide at least some visibility into the network state. While such scenarios would not allow for the identification of the exact physical segment to which a node was connected, the system could at least reduce the scope of possibility to the attached switch.

8. Results

I have demonstrated the capabilities and operations of the final IPAC system in a platform that attempts to replicate a corporate network environment on a small scale. In this section, I describe the network and procedures used to demonstrate this system and provide examples of how administrators and investigators can utilize its IPv6 address tracking capabilities.

8.1 Demonstration Network

The network I used to demonstrate the operation of the IPAC system attempted to provide a small-scale replication of what an enterprise might implement within a single office. The network consisted of three subnets: one containing host systems (“LAN A”), another containing servers (“LAN B”), and a third providing connectivity to networks external to the organization (“WAN”). Each network was assigned a separate Global scope subnet (within 2001::/16), while the two internal networks also utilized a Unique-Local scope subnet (within FC00::/8). The two networks considered to be internal to the organization each had their own IPAC Agents, with a single IPAC Server processing all Agent records and handling SNMP trap messages from the Ethernet switches. Web servers were also provided on the LAN B and WAN portions on the network for use in the generation of test traffic. This network topology is shown in Figure 8.1.

The IPAC Agents and Servers were implemented on Ubuntu 10.10 Desktop operating systems running Python 2.6.6 and MySQL Server 5.1.49. The web servers were provided by CentOS 5.5 hosts running Apache 2.2.3. Routing between the networks was provided by a host using the Vyatta Core 6.0 routing platform. The hosts attached to LAN A consisted of several Windows XP Professional SP3 and Ubuntu 10.10 systems.
This demonstration network heavily relied on virtualization to provide routing and network services. All virtualized systems were provided by a single physical server running Ubuntu 10.04 Desktop LTS 64-bit as its host operating system. Each of the IPAC Agents and the IPAC Server existed as an Oracle VirtualBox 4.0.6 image, while the web servers and router existed as VMware Workstation 7.1.3 images. The use of both virtualization software packages was necessitated by behavior observed during the IPAC system testing phase. Initially deployed completely within the VMware Workstation environment, the IPAC Agents and Servers were migrated to VirtualBox images after observing instances in which VMware did not properly forward ICMPv6 traffic to the virtual systems. The other virtualized services were left on the initial platform after determining that their operation would not be affected by this behavior. The systems contained within these two virtualized environments (on the same physical host) are shown within the dashed blue line in Figure 8.1.

![Diagram of the demonstration network](image.png)

**Figure 8.1. Demonstration Network Topology**
8.2 NDP Traffic Processing / IPAC System Operation

The IPAC demonstration began with the Agents, Servers, and network infrastructure in place but host nodes detached from the network. These host nodes were then individually attached to the Ethernet switches. After each physical connection was created, the switches generated an SNMP notification telling the Trap Handler module that it had learned about the new host’s MAC address. Upon receiving these messages, the Trap Handler process created a MAC notification record as shown in Figure 8.2 (table structure is explained in Section 11.2.2).

```
mysql> select * from mac_notifications;
+---------------+--------------+------+------+-----------+------------------+
| switchIPv4    | macAddress   | port | vlan | operation | timestamp        |
|---------------+--------------+------+------+-----------+------------------+
| 10.100.111.21 | 006008274afc | 000a | 0001 |         1 | 1304029177.94567 |
+---------------+--------------+------+------+-----------+------------------+
1 row in set (0.00 sec)
```

Figure 8.2. Sample SNMP trap record obtained from the IPAC Trap Handler database.

After being attached to the switch, each host would establish a connection to the web servers in the LAN B and WAN portions of the network. As part of this process, NDP traffic containing the hosts’ Unique-Local and Global addresses was exchanged between these nodes and the router. Upon observing this traffic, the Agent verified the existence of the addresses, added them to its list of known active IP addresses, and created a sighting record marking the time the addresses were observed. Examples of these records are shown in Figure 8.3 (table structures are explained in Section 11.2.1).

```
mysql> select * from active;
+----------------------------------+--------------+------------------+---------------+
| ipv6Address                      | macAddress   | lastSeen         | retryFailures |
+----------------------------------+--------------+------------------+---------------+
| 20010DB81111111111020C29FFFE394C39 | 000C29394C39 | 1304029615.33424 |             0 |
| 20010DB81111111111000000000000001 | 000C29C1A470 | 1304029603.19496 |             1 |
| FC00AAAAAAA1111111111111111111 | 000C29C1A470 | 1304029627.85339 |             0 |
| FE80000000000000020C29FFEE1A470 | 000C29C1A470 | 1304029612.45889 |             0 |
| FE80000000000000020C29FFFE394C39 | 000C29394C39 | 1304029620.5273  |             0 |
+----------------------------------+--------------+------------------+---------------+
5 rows in set (0.00 sec)
```

```
mysql> select * from records;
+----------------------------------+--------------+------------------+------+
| ipv6Address                      | macAddress   | timestamp        | type |
|----------------------------------+--------------+------------------+------+
| 20010DB81111111111111020C29FFFE394C39 | 000C29394C39 | 1304029603.19494 |   11 |
| 20010DB811111111111020C29FFFE394C39 | 000C29394C39 | 1304029578.41214 |   11 |
| 20010DB811111111111020C29FFFE394C39 | 000C29394C39 | 1304029578.53811 |   11 |
| FE80000000000000020C29FFE1A470   | 000C29C1A470 | 1304029620.5273  |   10 |
+----------------------------------+--------------+------------------+------+
```

- 53 -
At regular intervals, the IPAC Server would obtain the address sighting and SNMP trap records from the Agents and Trap Handler databases. These records were processed to generate records summarizing IP-MAC and MAC-Port binding sessions (as discussed in Section 7.3). Examples of the records generated and stored by the IPAC Server are shown in Figure 8.4 (table structures are explained in Section 11.2.3).
Correlating IPv6 Addresses for Network Situational Awareness

Figure 8.4. Sample IPv6-MAC (top) and MAC-Port (bottom) summary records obtained from the IPAC Server database.

8.3 Interface Scripts Operation

During the IPAC system operation demonstration, several application logs and packet captures were created to demonstrate the IPAC Interface scripts. These include an Apache access log from HTTP Server 1 on LAN B, an iptables log (via Syslog) from the central Router, and a short packet capture from LAN A. Details of the operations and outputs of each of these scripts are provided in Section 7.4.

The ipac_textlog.py script was capable of processing the log files from the Apache webserver and iptables firewall. As shown in Figures 8.5 and 8.6, this script appended a report of IPv6 address usage at the end of each individual log entry. Enclosed in “<” and “>” brackets, this report includes each IPv6 address found in the log entry, any MAC addresses associated with the IPs, and any switch ports associated with MACs. If an IPv6 did not have an associated MAC address or switch port, that text was replaced with “Unknown” or “Port Unknown”, respectively. In addition, each MAC address and switch port ID was marked with a confidence rating in one to four asterisks (MAC addresses) or carats (switch ports), as discussed in Section 7.4.3. In cases where multiple IPv6 addresses were found within a log entry, each address report was separated by “ | | ”. The report output was structured in the following format:

```plaintext
< [ IPv6 Address -> MAC Address*Confidence (Switch ID/Switch Port/VLAN*Confidence) | | ... ] >
```

---

```
< [ FC00:AAAA:AAAA:1111:20731B8F21C -> 001731B8F21C****
10.100.111.21/0007/0001^^^ ] >

< [ FC00:AAAA:AAAA:1222:20c:29ff:fe7a:32D5 -> 001731B8F21C****
10.100.111.21/0007/0001^^^ ] >
```

---

- 55 -
Correlating IPv6 Addresses for Network Situational Awareness

Jason Froehlich

Figure 8.5. Output of Apache access log processed by ipac_textlog.py.
Bold portions represent text added by the IPAC processing script.

<table>
<thead>
<tr>
<th>Port Unknown</th>
<th>00:0C:29:39:2B:49 (Port Unknown)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00:0C:29:39:2B:49 (Port Unknown)</td>
<td>00:0C:29:39:2B:49 (Port Unknown)</td>
</tr>
</tbody>
</table>

Figure 8.6. Output of iptables log processed by ipac_textlog.py.
Bold portions represent text added by the IPAC processing script.

The ipac_pcaplog.py script was capable of processing binary packet capture files in the libpcap format (shown in Figure 8.8) and producing a separate text report of IPv6 address usage within these packets (shown in Figure 8.7). For each IPv6 packet in the file, the script identified each unicast source and destination address and attempted to match it to a MAC address and physical switch port. If an IPv6 did not have an associated MAC or switch port, that text was replaced with “Unknown” or “Port Unknown”. The script output was structured in the following format:

<table>
<thead>
<tr>
<th>Packet #</th>
<th>Source IPv6 Address</th>
<th>MAC Address (Switch ID/Switch Port/VLAN)</th>
<th>Destination IPv6 Address</th>
<th>MAC Address (Switch ID/Switch Port/VLAN)</th>
</tr>
</thead>
</table>
Figure 8.7. Report produced by ipac_pcaplog.py, generated from the packets shown in Figure 8.8.

The ipac_graph.py script provided an interactive visualization of the IPv6 addressing information stored by the IPAC system, as shown in Figure 8.9. This timeline graph plotted each known IPv6 address along the Y-axis and time along the X-axis. Every IPv6-MAC pair in the IPAC Server's records was represented by a blue horizontal bar positioned next to the proper IPv6 label on the Y-axis and spanning the proper starting and ending times on the X-axis. Each of these bars were marked with its associated MAC address, and ended with a colored marker indicating the ending status of the IPv6-MAC pair: green when the system though the IPv6-MAC session was still active and red when the system knew the session had ended. If a user clicked on a timeline bar, the graph would also show the exact starting and ending times for the session (shown on the bottom bar in Figure 8.9). Any MAC conflicts were indicated by overlapping bars and highlighted by an underlining red box, as shown on the 4th bar from the bottom in Figure 8.9.
Figure 8.9. Interactive timeline graph of IPv6 and MAC address usage, produced by ipac_graph.py.
9. Future Work

The IPAC system that I have created was meant to be a proof of concept demonstration of the viability of using NDP for tracking IPv6 address usage within an organization's networks. Before being deployed in a production network for forensic purposes, additional intensive code reviews should be performed to ensure that all cases in which erroneous information could be recorded are precluded. Additionally, there are numerous small modifications that could be made to the system to provide for additional functionality and easier deployment in enterprise networks.

9.1 Bind port to user (802.1x)

The IPAC system outlined in this document provides the capability to correlate an IPv6 address to a specific network switch port at a given point in time. This is useful in narrowing the field of possible machines to a specific physical subset of the network, but still leaves the incident investigator with the task of tracking down which machines are attached to the identified link. It may be possible to further extend this system to allow for identification of a specific user account in addition to the physical switch port through the use of IEEE 802.1x [24].

IEEE 802.1x is a standardized mechanism for providing port-based network access control to wired or wireless networks. This utility allows the network to authenticate each physically attached device before any of its traffic is allowed to pass. An 802.1x system consists of three separate entities: the supplicant accessing the network (the host with 802.1x software), the authenticator providing or denying network access to the host (the Ethernet switch), and the authentication server verifying the credentials provided by the host and making the decision of whether to grant or deny access. When a supplicant connects to the authenticator, it is prompted for the host's credentials (a username/password pair, certificate, or pre-shared key). Once received by the authenticator, this information is forwarded to the authentication server for validation. Based on the information provided, the server will reply to the authenticator with a success or failure message. If the authentication succeeded, the switch will activate the appropriate physical port and allow the host's traffic to pass freely. Otherwise, the physical port will either stay in a deactivated state or allow traffic to pass only to a segregated guest VLAN (according to the policies configured on the switch). Although many authentication servers are available, several of the most popular are based on the open RADIUS protocol. Because the
Correlating IPv6 Addresses for Network Situational Awareness

Before an 802.1x and RADIUS implementation can be considered suitable for use with the IPAC system, it must satisfy several requirements that aren't mandated by the IEEE standard. First, the authentication server must be capable of logging all information it receives when authenticating a user. These logs will be used by the IPAC system to correlate a username to a specific port. FreeRADIUS [16], a popular open source RADIUS implementation, does provide this functionality. Additionally, the network switch attempting to authenticate the user must inform the RADIUS server of the MAC address of the authenticating host as well as the physical port to which it is connected. The RADIUS protocol allows for this information to be transferred by the 'NAS-Port' and 'Calling-Station-ID' Attribute Value Pairs (AVPs) within the packet, respectively. While these specific AVPs are not required for RADIUS authentication to occur, it appears that they are included in the authentication messages from Cisco Catalyst 2950 series Ethernet switches running IOS release 12.1(22)EA14 and later.

Finally, the network switch should ideally have the capability of authenticating each device that joins its physical ports. In traditional 802.1x implementations, the port is activated after the first node is authenticated. If additional nodes join the port (such as through a hub), their traffic will be passed without having been authenticated. Many implementations provide the option of allowing only the authenticated node's traffic to pass and blocking all others (based on Source MAC address), but in these cases only one device can be authenticated at a time. Certain Cisco Catalyst switches provide the ability to authenticate up to eight devices at a time through a feature they call “802.1x Multiple Authentication Mode”, though this appears to only be available on Catalyst 2960 series and later switches running IOS release 12.2(50)SE and later.

9.2 Multiple VLANs per Agent

In the system I demonstrated, each IPAC Agent was capable of handling NDP traffic from only a single VLAN on a single interface. This limitation makes it necessary for a different IPAC Agent to exist on each VLAN controlled by the organization, a requirement that can be financially and logistically difficult for those with a large number of local networks. This limitation can be largely obviated by modifying the IPAC system to support what I have called “Agent-on-a-stick.” Similar to the well-known “Router-on-a-Stick” network topology [5], this
Correlating IPv6 Addresses for Network Situational Awareness

system consists of an Agent connected to a single physical network link but serving multiple VLANs. On that host, an instance of the Agent software would be active for each VLAN served over the link. Rather than process all incoming NDP traffic, each instance would only pay attention to those packets tagged with their respective VLAN.

To implement this, a small amount of modification to the IPAC system's code would be required, but no large changes to the system's architecture. Upon initialization of the IPAC Agent daemon, the list of VLAN IDs active on the network link would need to be read from the Agent's configuration file, and then a set of Agent threads generated for each ID. A single packet capturing thread could handle all incoming packets, sort the packets by their VLAN ID, and relay them to the proper Agent thread through Queues. The structure of the Agent database would need to be modified to have a VLAN ID column in the active and records tables, which would be included in the tables' primary keys. All queries to the database from the Agent process would also need to include the thread's respective VLAN ID. The IPAC Server process would need to separate all obtained record entries by VLAN ID, and then process each of these sets just as it currently processes each Agent.

On a typical network, a single Agent implemented in this manner should be able to easily handle five or more VLANs. In such cases network bandwidth on the Agent's physical link should not be an issue. Since only multicast and broadcast traffic should be traversing the link, there is only a minimal possibility that the link would become saturated. In normal scenarios, such traffic should only constitute a small portion of the network's total bandwidth capabilities. Another consideration, although unlikely to become an issue with any hardware purchased within the last server replacement cycle, would be Agent processing resources. Each VLAN would require its own set of three threads running simultaneously on the Agent system in addition to the ten already in use by the Agent process. The actual system requirements for an Agent in this scenario would depend on the number of VLANs being monitored, the number of hosts on each VLAN, and the amount of ICMPv6 traffic on each VLAN.

9.3 Additional Log Parsing Capabilities

With this IPAC system, I demonstrated a script for processing IPv6 addresses within Apache access and error logs and Syslog files. As IPv6 compatibility continues to be extended to an increasing number of applications, we will see IPv6 addresses appear in many more types
of log files. Because log files often utilize different formats for representing and organizing their information, this processing script will likely need to be modified to accommodate to each individual log source. Specifically, the log parsing script would need to be updated with the proper regular expressions for identifying timestamps within each log entry.

9.4 Interactive Log Processing

The IPAC system that I demonstrated provides several scripts for processing different types of text and packet capture logs. However, the output produced by these utilities can be immense and cumbersome to work with when processing large log files. As a result, this system could benefit from the development of a utility that provides dynamic interaction with the processing results.

One example of such a utility would be a web browser-based view of the processed file. Instead of having the MAC address and switch port summaries appended to each line, this information could be provided in a dynamic callout box (through Javascript) when the user clicks or hovers over an IPv6 address. Additionally, this could allow for the highlighting of addresses within each entry, sorting and filtering of entries, and summarization of contiguous entries.

9.5 Impact on Wireless Networks

While this IPAC system was designed and demonstrated on a wired Ethernet network, it could also prove useful if implemented on wireless local area networks (WLANs). Further research is necessary to study its impact on these networks in terms of its affect on wireless network performance. Unlike modern switched Ethernet LANs, WLANs utilize a single collision domain (or a small few, depending on the frequency sets in use). Because of this, all wireless nodes must contend for the ability to transmit data without causing collisions. Since the IPAC system will increase the network’s NDP traffic by a factor of 2-3, its operation may result in degraded wireless network performance when a high level of NDP traffic already exists. Additionally, further research should examine the wireless network's impact on the forensic validity of the IPAC system. It may be possible that the system's view of the network could be affected by a substantial amount of collisions, or low power or sleep mechanisms employed on wireless nodes. These areas of concern should be investigated in both WiFi-only networks and mixed environments (WLAN bridged to a wired Ethernet network).
9.6 Additional ICMPv6 Security Protections

The functionality provided by the IPAC system provides a base that can be used to implement many other operational and security protection mechanisms for the functions of ICMPv6. Since it already listens to and processes all multicasted ICMPv6 traffic on the local network, the system could be modified to alert administrators or perform certain actions upon seeing certain types of packets or traffic patterns. While the implementation of each of these features would only require minor modifications to the IPAC system, they were beyond the scope of this thesis. Examples of possible additional mechanisms include:

NDP Spoofing – Since the system already maintains a database of known IPv6 and MAC address pairs, it can recognize when a host claims to own the IP address used by another. Such scenarios could be the result of a Man-in-the-Middle attack that causes traffic to be forwarded through a malicious device. To provide this functionality, the IPAC system would simply need to be modified to send an alert to administrators when such conditions are observed.

Rogue Router Advertisements – ICMPv6 Router Advertisement packets are used to inform nodes of the routers available on their network. With this mechanism, it is possible for a malicious host to pretend to be a valid router in order to intercept other nodes' traffic. If the rogue device claims to have a higher preference than all other routers on the network, all traffic destined for subnets beyond the local network will be forwarded to the attacker. To provide this, the IPAC system would need to be modified to examine all multicasted Router Advertisement packets and compare them to a list of known valid routers.

Rogue Network Prefixes and Darknets – Each ICMPv6 Router Advertisement packet contains a list of network address prefixes that the router considers to be active on the local network. Beyond generating Routing Advertisements for an unauthorized router, a malicious device could also generate an Advertisement for a valid router that includes extra network prefixes not authorized by administrators. Upon receiving these packets, nodes will automatically generate an address in this new subnet, all without the knowledge of the network's administrators. Such unknown networks within the existing LAN are often known as Darknets [7], and can facilitate inter-host communication that bypasses some network protection mechanisms. The IPAC system could identify these scenarios just as it would identify rogue routers. In addition to a list of valid routers, the system would also need to know the network
prefixes associated with each. Additionally, the IPAC system could also recognize unauthorized address ranges created by mechanisms other than spoofed Router Advertisements. Since it sees every IPv6 address in use on the network, it could easily verify that each address is part of a known subnet and produce an alert if any are not.

Duplicate Address Detection Denial of Service – Before a host configures an IPv6 address on one of its interfaces, it sends a series of ICMPv6 Neighbor Solicitation packets to verify that the address is not already in use on the network (Duplicate Address Detection). This behavior creates the opportunity for an attacker to cause a denial of service on the local network by responding to all of these Solicitations, preventing any host from creating a valid address. The IPAC system could be modified to identify the traffic patterns caused by this attack (a large volume of NDP Advertisements from many different IPv6 addresses coming from the same Source MAC address or physical switch port) and produce an alert. The system could even provide the administrator with the exact location of the attack source if modified to correlate these Advertisements to a single switch port.

9.7 Correlating Multiple IPv6/MAC/Port Pairs

While this IPAC System is able to establish relationships between IPv6 addresses, MAC addresses, and physical switch ports, it is unable to correlate one IPv6/MAC/Port pair to another. For example, if a host is attached to a network (or multiple networks) through two interfaces and sends a packet from each interface, the IPAC system is unable to determine that the two packets originated from the same host. Since both interfaces have unique IPv6 and MAC addresses and are connected to different switch ports, the host will appear to be two separate nodes. In order to accomplish this correlation, some additional host identification mechanism must be used to augment the IPAC system. Although they each have limitations, potential solutions include the use of Dynamic DNS, upper-layer identifiers (cookies), or host clock skew calculation (each discussed in Section 4). Such a system could also aid in correlating spoofed IPv6/MAC pairs when they originate from a physical port connected to an Ethernet hub or unmanaged switch that does not provide bridge forwarding table notifications. While these scenarios would be somewhat atypical in most conventional networks, having this capability would help to increase our network situational awareness.
9.8 IPv4

Although the IPAC system I demonstrated focuses on IPv6, the issues that it confronts are by no means unique to that protocol. While I focused solely on IPv6 because of the magnitude of its address correlation issues, networks running IPv4 may also benefit from similar functionality. IPv4 networks utilize the Address Resolution Protocol (ARP) to match IP addresses to MAC addresses. Just as with NDP on IPv6 networks, a Solicitation-Advertisement exchange (in ARP parlance, request-reply) must occur before a host can send traffic to another IP address, providing a mechanism for tracking IPv4 address use. Since the fundamental principles behind tracking IPv6 and IPv4 addresses on local networks are the same, implementing the functionality of the IPAC system on an IPv4 network would only require a limited amount of modification to the system code. First, the Agent would need to be modified to listen for and generate ARP traffic. While the functionality of ARP and NDP are very similar, there are differences in their implementation that would necessitate a reevaluation of how packets are handled. Additionally, all parts of the system would need to be modified to handle IPv4 address formats.

10. Conclusions

The deployment of the IPv6 protocol on enterprise networks provides administrators and incident investigators with the pronounced challenge of identifying which node utilized an IP address at a given time. This challenge is introduced by the fact that IPv6 addressing can be highly volatile, nodes use a multiplicity of addresses simultaneously, and address generation and assignment is usually performed by individual nodes rather than centrally. Because of this, it may be difficult for investigators to identify the actual node that caused a specific application or firewall log entry when IP addresses are the only host identifier being recorded.

I have proposed and demonstrated a system that allows administrators and investigators to correlate an IPv6 address in use on their organization's network to a specific MAC address and physical Ethernet switch port at any given point in time. Based on the IPv6 Neighbor Discovery Protocol and switch CAM table information, the IPAC system satisfies all of the identified requirements for effectively maintaining a complete and accurate record of IPv6 address usage over time (as described in Section 2). This system is capable of attributing multiple IPv6 addresses to a single network entity as well as multiple entities to a single address, it provides
precise timing of network address initialization and termination, it maintains an exhaustive view of addresses in use on each network, it is capable of handling spoofing and other attack attempts, and it does not rely on any host-initiated action for obtaining addressing information. In 'normal' scenarios this system will be able to precisely identify a single host for a given IPv6 address; in instances in which the NDP protocol or MAC addresses are spoofed or implemented incorrectly, the system is able to narrow the scope of possible sources to a limited set of nodes: the legitimate source and any attacking/misbehaving nodes. Based on these results, it is possible to maintain a complete and accurate account of IPv6 addressing on enterprise networks via the Neighbor Discovery Protocol for use in security incident investigation or network management functions.
11. Appendices

11.1 Survey Results

Survey testing was completed on CentOS 5.5 Linux, Mac OS X 10.6, Ubuntu 10.10 Linux, Windows 7, Windows Server 2003, Windows Server 2008, and Windows XP SP3 systems. The results of each of these tests are outlined within the following sections. Observations drawn from these tests are discussed in Section 6.2.

11.1.1 Host System Information

11.1.1.1 CentOS 5.5

IPv6 functionality within CentOS 5.5 was tested on a VirtualBox virtual machine running on a Windows XP Professional SP3 host. This test system used a “factory default” installation of the OS provided by RIT's CentOS mirror, without any system updates or additional software installed. The VM's network interface was bridged directly to the host's wired Ethernet port, and was able to observe all traffic coming into the host. IPv6 functionality was enabled on the system's network interfaces by default. The Ethernet MAC address for the system's Local Area Connection interface was 08:00:27:f5:00:87. Detailed system and network information is shown in Figure 11.1.1.

```
[root@localhost ~]# uname -a
Linux localhost.localdomain 2.6.18-194.el5 #1 SMP Fri Apr 2 14:58:35 EDT 2010 i686 i686 i386 GNU/Linux

[root@localhost ~]# lsb_release -a
Distributor ID: CentOS
Description: CentOS release 5.5 (Final)
Release: 5.5
Codename: Final

[root@localhost ~]# ifconfig -a
eth0     Link encap:Ethernet  HWaddr 08:00:27:F5:00:87
          inet addr:10.100.111.12  Bcast:10.100.111.12  Mask:255.255.255.255
          inet6 addr: fc00:aaaa:aaaa:2222:a00:27ff:fef5:87/64 Scope:Global
          inet6 addr: 2001:db8:1111:2222:a00:27ff:fef5:87/64 Scope:Global
          inet6 addr: fe80::a00:27ff:fef5:87/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:75 errors:0 dropped:0 overruns:0 frame:0
```
Correlating IPv6 Addresses for Network Situational Awareness  
Jason Froehlich

<table>
<thead>
<tr>
<th>lo</th>
<th>TX packets:45 errors:0 dropped:0 overruns:0 carrier:0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RX bytes:10746 (10.4 KiB) TX bytes:7351 (7.1 KiB)</td>
</tr>
<tr>
<td>Link encap:Local Loopback</td>
<td></td>
</tr>
<tr>
<td>inet addr:127.0.0.1 Mask:255.0.0.0</td>
<td></td>
</tr>
<tr>
<td>inet6 addr: ::1/128 Scope:Host</td>
<td></td>
</tr>
<tr>
<td>UP LOOPBACK RUNNING MTU:16436 Metric:1</td>
<td></td>
</tr>
<tr>
<td>RX packets:1604 errors:0 dropped:0 overruns:0 frame:0</td>
<td></td>
</tr>
<tr>
<td>TX packets:1604 errors:0 dropped:0 overruns:0 carrier:0</td>
<td></td>
</tr>
<tr>
<td>collisions:0 txqueuelen:0</td>
<td></td>
</tr>
<tr>
<td>RX bytes:3754148 (3.5 MiB) TX bytes:3754148 (3.5 MiB)</td>
<td></td>
</tr>
<tr>
<td>sit0</td>
<td></td>
</tr>
<tr>
<td>Link encap:IPv6-in-IPv4</td>
<td></td>
</tr>
<tr>
<td>NOARP MTU:1480 Metric:1</td>
<td></td>
</tr>
<tr>
<td>RX packets:0 errors:0 dropped:0 overruns:0 frame:0</td>
<td></td>
</tr>
<tr>
<td>TX packets:0 errors:0 dropped:0 overruns:0 carrier:0</td>
<td></td>
</tr>
<tr>
<td>collisions:0 txqueuelen:0</td>
<td></td>
</tr>
<tr>
<td>RX bytes:0 (0.0 b) TX bytes:0 (0.0 b)</td>
<td></td>
</tr>
</tbody>
</table>

Figure 11.1.1. CentOS 5.5 System and Network Information

11.1.1.2 Mac OS X

IPv6 functionality within Mac OS X was tested with a physical Apple MacBook Pro 3,1 laptop running OS X 10.6 Build 10A432 (Snow Leopard). This test system used a factory default installation of the OS, without any system updates or additional software installed. The OS's IPv6 functionality is provided by a customized version of the network stack provided by the KAME project; this system identified its IPv6 network stack as version '20010528/apple-darwin'. IPv6 functionality was enabled on the system's network interfaces by default. The Ethernet MAC address for the system's wired network port was 00:1b:63:9a:1b:67. Detailed system and network information is shown in Figure 11.1.2.

bash-3.2$ sw_vers
ProductName: Mac OS X
ProductVersion: 10.6
BuildVersion: 10A432

bash-3.2$ sysctl net.inet6.ip6.kame_version
net.inet6.ip6.kame_version: 20010528/apple-darwin

bash-3.2$ ifconfig en0
en0: flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX,MULTICAST> mtu 1500
inet6 fe80::21b:63ff:fe9a:2454%en0 prefixlen 64 scopeid 0x4
    inet6 2001:db8:1111:2222:21b:63ff:fe9a:2454 prefixlen 64 autoconf
    inet6 fc00:aaaa:aaaa:2222:21b:63ff:fe9a:2454 prefixlen 64 autoconf
    inet 169.254.166.50 netmask 0xffff0000 broadcast 169.254.255.255
    ether 00:1b:63:9a:1b:67
    media: autoselect (100baseTX <half-duplex>) status: active
    supported media: autoselect 10baseT/UTP <half-duplex> 10baseT/UTP <full-duplex>
            10baseT/UTP <full-duplex,hw-loopback> 10baseT/UTP <full-duplex,flow-control> 100baseTX <half-duplex> 100baseTX <full-duplex>
IPv6 functionality within Ubuntu 10.10 was tested on a VirtualBox virtual machine running on a Windows XP Professional SP3 host. This test system used a “factory default” installation of the OS provided by RIT’s Ubuntu mirror, without any system updates or additional software installed. The VM’s network interface was bridged directly to the host’s wired Ethernet port, and was able to observe all traffic coming into the host. IPv6 functionality was enabled on the system's network interfaces by default. The Ethernet MAC address for the system's Local Area Connection interface was 08:00:27:a4:89:65. Detailed system and network information is shown in Figure 11.1.3.

```
root@Survey-Ubuntu1010:~# uname -a
Linux Survey-Ubuntu1010 2.6.35-22-generic #33-Ubuntu SMP Sun Sep 19
20:34:50 UTC 2010 i686 GNU/Linux

root@Survey-Ubuntu1010:~# lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description: Ubuntu 10.10
Release: 10.10
Codename: maverick

root@Survey-Ubuntu1010:~# ifconfig -a
eth0      Link encap:Ethernet  HWaddr 08:00:27:a4:89:65
inet6 addr: fc00:aaaa:aaaa:2222:a00:27ff:fea4:8965/64 Scope:Global
inet6 addr: fe80::a00:27ff:fea4:8965/64 Scope:Link
UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
RX packets:102 errors:0 dropped:0 overruns:0 frame:0
TX packets:49 errors:0 dropped:0 overruns:0 carrier:0
 collisions:0 txqueuelen:1000
RX bytes:15546 (15.5 KB)  TX bytes:9611 (9.6 KB)
lo      Link encap:Local Loopback
inet addr:127.0.0.1  Mask:255.0.0.0
inet6 addr: ::1/128 Scope:Host
UP LOOPBACK RUNNING  MTU:16436  Metric:1
RX packets:12 errors:0 dropped:0 overruns:0 frame:0
TX packets:12 errors:0 dropped:0 overruns:0 carrier:0
 collisions:0 txqueuelen:0
RX bytes:720 (720.0 B)  TX bytes:720 (720.0 B)
```

Figure 11.1.3. Ubuntu 10.10 System and Network Information
11.1.1.4 Microsoft Windows 7

IPv6 functionality within Windows 7 Professional was tested on a VirtualBox virtual machine running on a Windows XP Professional SP3 host. This test system used a factory default installation of the OS provided by the Microsoft Developer Network Academic Alliance (MSDNAA), without any system updates, service packs, or additional software installed. The VM's network interface was bridged directly to the host's wired Ethernet port, and was able to observe all traffic coming into the host. IPv6 functionality was enabled on the system's network interfaces by default. The Ethernet MAC address for the system's Local Area Connection interface was 08:00:27:05:9a:35. Detailed system and network information is shown in Figure 11.1.4.

C:\Users\jason>systeminfo
Host Name:                 Survey-Win7
OS Name:                   Microsoft Windows 7 Professional
OS Version:                6.1.7600 N/A Build 7600
OS Manufacturer:           Microsoft Corporation
OS Configuration:          Standalone Workstation
OS Build Type:             Multiprocessor Free
Registered Owner:          jason
Registered Organization:
Product ID:                00371-839-1630263-85121
Original Install Date:     3/11/2011, 1:31:16 PM
System Boot Time:          3/11/2011, 1:18:58 PM
System Manufacturer:       innotek GmbH
System Model:              VirtualBox
System Type:               X86-based PC
Processor(s):              1 Processor(s) Installed.
                          [01]: x86 Family 15 Model 3 Stepping 4
GenuineIntel ~ 3423 Mhz
BIOS Version:              innotek GmbH VirtualBox, 12/1/2006
Windows Directory:         C:\Windows
System Directory:          C:\Windows\system32
Boot Device:               \Device\HarddiskVolume1
System Locale:             en-us;English (United States)
Input Locale:              en-us;English (United States)
Time Zone:                 (UTC-05:00) Eastern Time (US & Canada)
Total Physical Memory:     512 MB
Available Physical Memory: 292 MB
Virtual Memory: Max Size:  1,536 MB
Virtual Memory: Available: 1,212 MB
Virtual Memory: In Use:    324 MB
Page File Location(s):     C:\pagefile.sys
Domain:                    WORKGROUP
Logon Server:              \WIN-79B7RLDF1C
Hotfix(s):                 N/A
Network Card(s):           1 NIC(s) Installed.
                          [01]: Intel(R) PRO/1000 MT Desktop Adapter
Connection Name: Local Area Connection
Correlating IPv6 Addresses for Network Situational Awareness

Jason Froehlich

C:\Users\jason>ipconfig /all
Windows IP Configuration
  Host Name . . . . . . . . . . . . : Survey-Win7
  Primary Dns Suffix . . . . . . . :
  Node Type . . . . . . . . . . . . : Hybrid
  IP Routing Enabled . . . . . . . : No
  WINS Proxy Enabled . . . . . . . : No
  Ethernet adapter Local Area Connection:
    Connection-specific DNS Suffix . :
    Description . . . . . . . . . . . : Intel(R) PRO/1000 MT Desktop Adapter
    Physical Address . . . . . . . . : 08-00-27-05-9A-35
    DHCP Enabled . . . . . . . . . : Yes
    Autoconfiguration Enabled . . . : Yes
    IPv6 Address . . . . . . . . . : 2001:db8:1111:2222:edf4:4de5:3f0d:5eed(Preferred)
    IPv6 Address . . . . . . . . . :
    Temporary IPv6 Address . . . . :
    Link-local IPv6 Address . . . . : fe80::edf4:4de5:3f0d:5eed
    IPv4 Address . . . . . . . . . : 10.0.2.15(Preferred)
    Subnet Mask . . . . . . . . . : 255.255.255.0
    Lease Obtained . . . . . . . . : Tuesday, February 02, 1875 10:01:24 AM
    Lease Expires . . . . . . . . . : Saturday, March 12, 2011 2:14:35 PM
    Default Gateway . . . . . . . : fe80::219:2fff:fe8e:db79%11
                                 10.0.2.2
    DNS Servers . . . . . . . . . : fec0:0:0:ffff::1%
                                 fec0:0:0:ffff::2%
                                 fec0:0:0:ffff::3%
    NetBIOS over Tcpip . . . . . . : Enabled
    Tunnel adapter isatap.{A239B84D-905B-444D-9A4A-4A1EC3A35178}:

Figure 11.1.4. Windows 7 System and Network Information (isatap interface information truncated)

11.1.1.5 Microsoft Windows Server 2003

IPv6 functionality within Windows Server 2003 was tested on a VirtualBox virtual machine running on a Windows XP Professional SP3 host. This test system used a factory default installation of the OS provided by the Microsoft Developer Network Academic Alliance (MSDNAA), without any system updates or additional software installed. The VM's network interface was bridged directly to the host's wired Ethernet port, and was able to observe all traffic coming into the host. The Ethernet MAC address for the system's Local Area Connection interface was 08:00:27:9c:78:bc. Detailed system information is shown in Figure 11.1.5.
In all versions of Windows Server 2003, IPv6 functionality is not enabled by default. The protocol can be installed and enabled using either the Windows command prompt or Network Connections properties within the Control Panel GUI. From the command line, the only command needing to be executed is `netsh interface ipv6 install`. If using the Control Panel, this can be accomplished by opening the network interface's properties, choosing the “Install” button, selecting “Protocol”, and selecting “Microsoft TCP/IP version 6”, as shown in Figure 11.1.5.
Figure 11.1.6. Once installed, “Microsoft TCP/IP version 6” will be included in the network interface's protocol list, as shown in Figure 11.1.7. Detailed network information for the test system is shown in Figure 11.1.8.

Figure 11.1.6. Windows Server 2003 IPv6 Protocol Installation

Figure 11.1.7. Windows Server 2003 IPv6 Protocol Installed
Correlating IPv6 Addresses for Network Situational Awareness  
Jason Froehlich

C:\Documents and Settings\Administrator>ipconfig /all
Windows IP Configuration
- Host Name: survey-win2k3
- Primary Dns Suffix: 
- Node Type: Unknown
- IP Routing Enabled: No
- WINS Proxy Enabled: No
Ethernet adapter Local Area Connection:
- Connection-specific DNS Suffix: 
- Description: AMD PCNET Family PCI Ethernet Adapter
- Physical Address: 08-00-27-9C-78-BC
- DHCP Enabled: Yes
- Autoconfiguration Enabled: Yes
- Autoconfiguration IP Address: 169.254.86.7
- Subnet Mask: 255.255.0.0
- IP Address: fc00:aaaa:aaaa:2222:a00:27ff:fe9c:78bc
- IP Address: 2001:db8:1111:2222:a00:27ff:fe9c:78bc
- Default Gateway: fe80::219:6ff:fe23:8569%5
- DNS Servers: fec0:0:0:ffff::1%1
  fec0:0:0:ffff::2%1
  fec0:0:0:ffff::3%1
Tunnel adapter Teredo Tunneling Pseudo-Interface:
  ...
Tunnel adapter Automatic Tunneling Pseudo-Interface:
  ...

Figure 11.1.8. Windows Server 2003 Network Information

11.1.1.6 Microsoft Windows Server 2008

IPv6 functionality within Windows Server 2008 was tested on a VirtualBox virtual machine running on a Windows XP Professional SP3 host. This test system used a factory default installation of the OS provided by the Microsoft Developer Network Academic Alliance (MSDNAA), without any system updates or additional software installed. The VM's network interface was bridged directly to the host's wired Ethernet port, and was able to observe all traffic coming into the host. IPv6 functionality was enabled on the system's network interfaces by default. The Ethernet MAC address for the system's Local Area Connection interface was 08:00:27:f0:ae:2c. Detailed system and network information is shown in Figure 11.1.9.

C:\Users\Administrator>systeminfo
Host Name: WIN-DGQB215TLFX
OS Name: Microsoft Windows Server 2008 Standard
OS Version: 6.0.6001 Service Pack 1 Build 6001
OS Manufacturer: Microsoft Corporation
OS Configuration: Standalone Server
OS Build Type: Multiprocessor Free
Registered Owner: Windows User

- 74 -
Correlating IPv6 Addresses for Network Situational Awareness

<table>
<thead>
<tr>
<th>Registered Organization:</th>
<th>92573-082-2500115-76772</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original Install Date:</td>
<td>3/11/2011, 9:33:55 AM</td>
</tr>
<tr>
<td>System Boot Time:</td>
<td>3/13/2011, 4:49:44 PM</td>
</tr>
<tr>
<td>System Manufacturer:</td>
<td>innotek GmbH</td>
</tr>
<tr>
<td>System Model:</td>
<td>VirtualBox</td>
</tr>
<tr>
<td>System Type:</td>
<td>X86-based PC</td>
</tr>
<tr>
<td>Processor(s):</td>
<td>1 Processor(s) Installed.</td>
</tr>
<tr>
<td></td>
<td>[01]: x86 Family 15 Model 3 Stepping 4</td>
</tr>
<tr>
<td>GenuineIntel ~</td>
<td>3.399 Mhz</td>
</tr>
<tr>
<td>BIOS Version:</td>
<td>innotek GmbH VirtualBox, 12/1/2006</td>
</tr>
<tr>
<td>Windows Directory:</td>
<td>C:\Windows</td>
</tr>
<tr>
<td>System Directory:</td>
<td>C:\Windows\system32</td>
</tr>
<tr>
<td>Boot Device:</td>
<td>\Device\HarddiskVolume1</td>
</tr>
<tr>
<td>System Locale:</td>
<td>en-us;English (United States)</td>
</tr>
<tr>
<td>Input Locale:</td>
<td>en-us;English (United States)</td>
</tr>
<tr>
<td>Time Zone:</td>
<td>(GMT-08:00) Pacific Time (US &amp; Canada)</td>
</tr>
<tr>
<td>Total Physical Memory:</td>
<td>511 MB</td>
</tr>
<tr>
<td>Available Physical Memory:</td>
<td>214 MB</td>
</tr>
<tr>
<td>Page File: Max Size:</td>
<td>1,494 MB</td>
</tr>
<tr>
<td>Page File: Available:</td>
<td>1,256 MB</td>
</tr>
<tr>
<td>Page File: In Use:</td>
<td>238 MB</td>
</tr>
<tr>
<td>Page File Location(s):</td>
<td>C:\pagefile.sys</td>
</tr>
<tr>
<td>Domain:</td>
<td>WORKGROUP</td>
</tr>
<tr>
<td>Logon Server:</td>
<td>\WIN-DGQB215TLFX</td>
</tr>
<tr>
<td>Hotfix(s):</td>
<td>N/A</td>
</tr>
<tr>
<td>Network Card(s):</td>
<td>1 NIC(s) Installed.</td>
</tr>
<tr>
<td></td>
<td>[01]: Intel(R) PRO/1000 MT Desktop Adapter</td>
</tr>
<tr>
<td></td>
<td>Connection Name: Local Area Connection</td>
</tr>
<tr>
<td></td>
<td>DHCP Enabled: Yes</td>
</tr>
<tr>
<td></td>
<td>DHCP Server: 255.255.255.255</td>
</tr>
<tr>
<td></td>
<td>IP address(es)</td>
</tr>
<tr>
<td></td>
<td>[01]: 169.254.119.21</td>
</tr>
<tr>
<td></td>
<td>[02]: fe80::f9ca:6411:b670:7715</td>
</tr>
<tr>
<td></td>
<td>[03]:</td>
</tr>
<tr>
<td></td>
<td>fc00:aaaa:aaaa:2222:f9ca:6411:b670:7715(Preferred)</td>
</tr>
<tr>
<td></td>
<td>[04]:</td>
</tr>
</tbody>
</table>

C:\Users\Administrator>ipconfig /all

Windows IP Configuration

    Host Name . . . . . . . . . . . . : WIN-DGQB215TLFX
    Primary Dns Suffix . . . . . . . . : 
    Node Type . . . . . . . . . . . . : Hybrid
    IP Routing Enabled. . . . . . . . : No
    WINS Proxy Enabled. . . . . . . . : No

Ethernet adapter Local Area Connection:

    Connection-specific DNS Suffix . . : 
    Description . . . . . . . . . . . . : Intel(R) PRO/1000 MT Desktop Adapter

    Physical Address . . . . . . . . . : 08-00-27-F0-AE-2C
    DHCP Enabled. . . . . . . . . . . : Yes
    Autoconfiguration Enabled . . . . : Yes
    IPv6 Address . . . . . . . . . . . : 
    IPv6 Address . . . . . . . . . . . : 
    fc00:aaaa:aaaa:2222:f9ca:6411:b670:7715(Preferred)
IPv6 functionality within Windows XP Professional Service Pack 3 was tested on a VirtualBox virtual machine running on a Windows XP Professional SP3 host. This test system used a factory default installation of the OS provided by the Microsoft Developer Network Academic Alliance (MSDNAA), without any system updates or additional software installed. The VM's network interface was bridged directly to the host's wired Ethernet port, and was able to observe all traffic coming into the host. The Ethernet MAC address for the system's Local Area Connection interface was 08:00:27:b9:92:8b. Detailed system information is shown in Figure 11.1.10.
In all versions of Windows XP, IPv6 functionality is not enabled by default. The protocol can be installed and enabled using either the Windows command prompt or Network Connections properties within the Control Panel GUI. From the command line, the only command needing to be executed is `netsh interface ipv6 install`. If using the Control Panel, this can be accomplished by opening the network interface's properties, choosing the “Install” button, selecting “Protocol”, and selecting “Microsoft TCP/IP version 6”, as shown in Figure 11.1.11. Once installed, “Microsoft TCP/IP version 6” will be included in the network interface's protocol list, as shown in Figure 11.1.12. Detailed network information for the test system is shown in Figure 11.1.13.
Correlating IPv6 Addresses for Network Situational Awareness  

Jason Froehlich

Figure 11.1.11. Windows XP SP3 IPv6 Protocol Installation

Figure 11.1.12. Windows XP SP3 IPv6 Protocol Installed

```
C:\Documents and Settings\jason> ipconfig /all
Windows IP Configuration
 Host Name . . . . . . . . . . . . : survey-winxpsp3
 Primary Dns Suffix . . . . . . . : 
 Node Type . . . . . . . . . . . . : Hybrid
 IP Routing Enabled . . . . . . . : No
 WINS Proxy Enabled . . . . . . : No
```
Correlating IPv6 Addresses for Network Situational Awareness

Jason Froehlich

<table>
<thead>
<tr>
<th>Ethernet adapter Local Area Connection:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connection-specific DNS Suffix :</td>
</tr>
<tr>
<td>Description . . . . . . . . . . . . . : AMD PCNET Family PCI Ethernet</td>
</tr>
<tr>
<td>Adapte r</td>
</tr>
<tr>
<td>Physical Address . . . . . . . . . . . : 08-00-27-B9-92-8B</td>
</tr>
<tr>
<td>Dhcp Enabled. . . . . . . . . . . . . . : Yes</td>
</tr>
<tr>
<td>Autoconfiguration Enabled . . . . . . : Yes</td>
</tr>
<tr>
<td>Autoconfiguration IP Address . . . . . : 169.254.8.253</td>
</tr>
<tr>
<td>Subnet Mask . . . . . . . . . . . . . : 255.255.0.0</td>
</tr>
<tr>
<td>IP Address . . . . . . . . . . . . . :</td>
</tr>
<tr>
<td>fc00:aaaa:aaaa:2222:d95a:d00c:7a3:41 38</td>
</tr>
<tr>
<td>IP Address . . . . . . . . . . . . . :</td>
</tr>
<tr>
<td>fc00:aaaa:aaaa:2222:a00:27ff:feb9:92 8b</td>
</tr>
<tr>
<td>IP Address . . . . . . . . . . . . . :</td>
</tr>
<tr>
<td>2001:db8:1111:2222:d95a:d00c:7a3:413 8</td>
</tr>
<tr>
<td>IP Address . . . . . . . . . . . . . :</td>
</tr>
<tr>
<td>2001:db8:1111:2222:a00:27ff:feb9:928 b</td>
</tr>
<tr>
<td>IP Address . . . . . . . . . . . . . : fe80::a00:27ff:feb9:928b%5</td>
</tr>
<tr>
<td>Default Gateway . . . . . . . . . . : fe80::219:2fff:fe8e:db79%5</td>
</tr>
<tr>
<td>DNS Servers . . . . . . . . . . . . . : fec0:0:0:ffff::1%1</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Tunnel adapter Teredo Tunneling Pseudo-Interface:</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Tunnel adapter Automatic Tunneling Pseudo-Interface:</td>
</tr>
</tbody>
</table>

Figure 11.1.13. Windows XP SP3 Network Information

11.1.2 NDP Behavior Results

11.1.2.1 Generation of NDP Solicitations

1) What are the values of the IPv6 and ICMPv6 options and flags?

a) CentOS 5.5

NDP Solicitation packets sent by the CentOS 5.5 system used null values for the Traffic Class and Flow Label fields in the IPv6 packet, 255 for the Hop Limit field in the IPv6 packet, and a null value for the Code field in the ICMPv6 datagram. This is consistent with the guidelines outlined in IETF RFC4861.

b) Mac OS X

NDP Solicitation packets sent by the Mac OS X system used null values for the Traffic Class and Flow Label fields in the IPv6 packet, 255 for the Hop Limit field in the IPv6 packet, and a null value for the Code field in the ICMPv6 datagram. This is consistent with the guidelines outlined in IETF RFC4861.

c) Ubuntu 10.10

NDP Solicitation packets sent by the Ubuntu 10.10 system used null values for the Traffic Class and Flow Label fields in the IPv6 packet, 255 for the Hop Limit field in the IPv6 packet, and a null value for the Code field in the ICMPv6 datagram. This is consistent with the guidelines outlined in IETF RFC4861.
d) **Windows 7**

NDP Solicitation packets sent by the Windows 7 system used null values for the Traffic Class and Flow Label fields in the IPv6 packet, 255 for the Hop Limit field in the IPv6 packet, and a null value for the Code and Flags fields in the ICMPv6 datagram. This is consistent with the guidelines outlined in IETF RFC4861.

e) **Windows Server 2003**

NDP Solicitation packets sent by the Windows Server 2003 system used null values for the Traffic Class and Flow Label fields in the IPv6 packet, 255 for the Hop Limit field in the IPv6 packet, and a null value for the Code and Flags fields in the ICMPv6 datagram. This is consistent with the guidelines outlined in IETF RFC4861.

f) **Windows Server 2008**

NDP Solicitation packets sent by the Windows Server 2008 system used null values for the Traffic Class and Flow Label fields in the IPv6 packet, 255 for the Hop Limit field in the IPv6 packet, and a null value for the Code and Flags fields in the ICMPv6 datagram. This is consistent with the guidelines outlined in IETF RFC4861.

g) **Windows XP**

NDP Solicitation packets sent by the Windows XP system used null values for the Traffic Class and Flow Label fields in the IPv6 packet, 255 for the Hop Limit field in the IPv6 packet, and a null value for the Code and Flags fields in the ICMPv6 datagram. This is consistent with the guidelines outlined in IETF RFC4861.

2) **What IPv6 Address is used as the Source?**

a) **CentOS 5.5**

Multicast NDP Solicitation packets sent by the CentOS 5.5 system used a Source IPv6 address that existed in the same subnet as the Target address; e.g., when soliciting for a node in the 2001:db8:1111:2222::/64 subnet, 2001:db8:1111:2222:a00:27ff:fef5:87 was used as the Source address. Unicast NDP Solicitations, on the other hand, used the system's link-local address as the Source IPv6 address.

b) **Mac OS X**

All NDP Solicitation packets sent by the Mac OS X system used a Source IPv6 address that existed in the same subnet as the Target address; e.g., when soliciting for a node in the 2001:db8:1111:2222::/64 subnet, 2001:db8:1111:2222:21b:63ff:fe9a:2454 was used as the Source address.

c) **Ubuntu 10.10**

Multicast NDP Solicitation packets sent by the Ubuntu 10.10 system used a Source IPv6 address that existed in the same subnet as the Target address; e.g., when soliciting for a node in the 2001:db8:1111:2222::/64 subnet, 2001:db8:1111:2222:a00:27ff:fea4:8965 was used as the Source address. Unicast NDP Solicitations, on the other hand, used the system's link-local address as the Source IPv6 address.
Correlating IPv6 Addresses for Network Situational Awareness  

Jason Froehlich

---

d) Windows 7
NDP Solicitation packets sent by the Windows 7 system used the IPv6 address existing in the same subnet as the target as the IPv6 Source address; e.g., when soliciting for a node in the 2001:db8:1111:2222::/64 subnet, 2001:db8:1111:2222:9c9:47da:29a7:d834 was used as the Source address.

e) Windows Server 2003
NDP Solicitation packets sent by the Windows Server 2003 system usually used the IPv6 address existing in the same subnet as the Target as the IPv6 Source address; e.g., when soliciting for a node in the 2001:db8:1111:2222::/64 subnet, 2001:db8:1111:2222:a00:27ff:fe9c:78bc was used as the Source address. However, a few cases showed that the link-local scope address may be used as the Source after an upper-layer connection has failed.

f) Windows Server 2008
NDP Solicitation packets sent by the Windows Server 2008 system used the IPv6 address existing in the same subnet as the Target as the IPv6 Source address; e.g., when soliciting for a node in the 2001:db8:1111:2222::/64 subnet, 2001:db8:1111:2222:f9ca:6411:b670:7715 was used as the Source address.

g) Windows XP
NDP Solicitation packets sent by the Windows XP system used the IPv6 address existing in the same subnet as the Target as the IPv6 Source address; e.g., when soliciting for a node in the 2001:db8:1111:2222::/64 subnet, 2001:db8:1111:2222:d95a:d00c:7a3:4138 was used as the Source address.

---

3) What values are used for IPv6 and Ethernet Destinations in multicast Solicitations?

a) CentOS 5.5
Multicast NDP Solicitation packets sent by the CentOS 5.5 system used the proper Solicited Node Multicast addresses in the Destination field for both the IPv6 and Ethernet headers, rather than the All Nodes Multicast IPv6 address (FF01::1) or broadcast Ethernet address (FF:FF:FF:FF:FF:FF). For example, a multicast Solicitation to fc00:aaaa:aaaa:2222:217:31ff:feb8:F21C used the IPv6 Destination address of ff02::1:ff8:F21C and Ethernet Destination address of 33:33:ff:b8:F21C.

b) Mac OS X
Multicast NDP Solicitation packets sent by the Mac OS X system used the proper Solicited Node Multicast addresses in the Destination field for both the IPv6 and Ethernet headers, rather than the All Nodes Multicast IPv6 address (FF01::1) or broadcast Ethernet address (FF:FF:FF:FF:FF:FF). For example, a multicast Solicitation to fc00:aaaa:aaaa:2222:217:31ff:feb8:F21C used the IPv6 Destination address of ff02::1:ff8:F21C and Ethernet Destination address of 33:33:ff:b8:F21C.

c) Ubuntu 10.10
Multicast NDP Solicitation packets sent by the Ubuntu 10.10 system used the proper...
Solicited Node Multicast addresses in the Destination field for both the IPv6 and Ethernet headers, rather than the All Nodes Multicast IPv6 address (FF01::1) or broadcast Ethernet address (FF:FF:FF:FF:FF:FF). For example, a multicast Solicitation to fc00:aaaa:aaaa:2222:217:31ff:feb8:F21C used the Destination IPv6 address of ff02::1:ffb8:F21C and Ethernet Destination address of 33:33:ff:b8:F21C.

**d) Windows 7**
Multicast NDP Solicitation packets sent by the Windows 7 system used the proper Solicited Node Multicast addresses in the Destination field for both the IPv6 and Ethernet headers, rather than the All Nodes Multicast IPv6 address (FF01::1) or broadcast Ethernet address (FF:FF:FF:FF:FF:FF). For example, a multicast Solicitation to fc00:aaaa:aaaa:2222:217:31ff:feb8:F21C used the IPv6 Destination address of ff02::1:ffb8:F21C and Ethernet Destination address of 33:33:ff:b8:F21C.

**e) Windows Server 2003**
Multicast NDP Solicitation packets sent by the Windows Server 2003 system used the proper Solicited Node Multicast addresses in the Destination field for both the IPv6 and Ethernet headers, rather than the All Nodes Multicast IPv6 address (FF01::1) or broadcast Ethernet address (FF:FF:FF:FF:FF:FF). For example, a multicast Solicitation to fc00:aaaa:aaaa:2222:217:31ff:feb8:F21C used the IPv6 Destination address of ff02::1:ffb8:F21C and Ethernet Destination address of 33:33:ff:b8:F21C.

**f) Windows Server 2008**
Multicast NDP Solicitation packets sent by the Windows Server 2008 system used the proper Solicited Node Multicast addresses in the Destination field for both the IPv6 and Ethernet headers, rather than the All Nodes Multicast IPv6 address (FF01::1) or broadcast Ethernet address (FF:FF:FF:FF:FF:FF). For example, a multicast Solicitation to fc00:aaaa:aaaa:2222:217:31ff:feb8:F21C used the IPv6 Destination address of ff02::1:ffb8:F21C and Ethernet Destination address of 33:33:ff:b8:F21C.

**g) Windows XP**
Multicast NDP Solicitation packets sent by the Windows XP system used the proper Solicited Node Multicast addresses in the Destination field for both the IPv6 and Ethernet headers, rather than the All Nodes Multicast IPv6 address (FF01::1) or broadcast Ethernet address (FF:FF:FF:FF:FF:FF). For example, a multicast Solicitation to fc00:aaaa:aaaa:2222:217:31ff:feb8:F21C used the IPv6 Destination address of ff02::1:ffb8:F21C and Ethernet Destination address of 33:33:ff:b8:F21C.

**4) Is the Source Link Layer Address option used in the ICMPv6 datagram?**

**a) CentOS 5.5**
All unicast and multicast NDP Solicitation packets sent by the CentOS 5.5 system did correctly implement the ICMPv6 Source link-layer address option, as outlined by IETF RFC4861.
b) Mac OS X
All unicast and multicast NDP Solicitation packets sent by the Mac OS X system did correctly implement the ICMPv6 Source link-layer address option, as outlined by IETF RFC4861.

c) Ubuntu 10.10
All unicast and multicast NDP Solicitation packets sent by the Ubuntu 10.10 system did correctly implement the ICMPv6 Source link-layer address option, as outlined by IETF RFC4861.

d) Windows 7
All unicast and multicast NDP Solicitation packets sent by the Windows 7 system did correctly implement the ICMPv6 Source link-layer address option, as outlined by IETF RFC4861.

e) Windows Server 2003
All unicast and multicast NDP Solicitation packets sent by the Windows Server 2003 system did correctly implement the ICMPv6 Source link-layer address option, as outlined by IETF RFC4861.

f) Windows Server 2008
All unicast and multicast NDP Solicitation packets sent by the Windows Server 2008 system did correctly implement the ICMPv6 Source link-layer address option, as outlined by IETF RFC4861.

g) Windows XP
All unicast and multicast NDP Solicitation packets sent by the Windows XP system did correctly implement the ICMPv6 Source link-layer address option, as outlined by IETF RFC4861.

11.1.2.2 Generation of NDP Advertisements

1) What are the values of the IPv6 and ICMPv6 options and flags?

a) CentOS 5.5
NDP Advertisement packets sent by the CentOS 5.5 system used null values for the Traffic Class and Flow Label fields in the IPv6 packet, 255 for the Hop Limit field in the IPv6 packet, and a null value for the Code field in the ICMPv6 datagram. When responding to a unicast Solicitation, the Advertisement had the Solicited flag set, while a response to a multicast Solicitation had both the Solicited and Override flags set.

b) Mac OS X
NDP Advertisement packets sent by the Mac OS X system used null values for the Traffic Class and Flow Label fields in the IPv6 packet, 255 for the Hop Limit field in the IPv6 packet, and a null value for the Code field in the ICMPv6 datagram. When responding to a unicast Solicitation, the Advertisement had the Solicited flag set, while a response to a multicast Solicitation had both the Solicited and Override flags set.
c) **Ubuntu 10.10**

   NDP Advertisement packets sent by the Ubuntu 10.10 system used null values for the Traffic Class and Flow Label fields in the IPv6 packet, 255 for the Hop Limit field in the IPv6 packet, and a null value for the Code field in the ICMPv6 datagram. When responding to a unicast Solicitation, the Advertisement had the Solicited flag set, while a response to a multicast Solicitation had both the Solicited and Override flags set.

d) **Windows 7**

   NDP Advertisement packets sent by the Windows 7 system used null values for the Traffic Class and Flow Label fields in the IPv6 packet, 255 for the Hop Limit field in the IPv6 packet, and a null value for the Code field in the ICMPv6 datagram. The Solicited and Override flags were set in the ICMPv6 portion of the packet for responses to both Unicast and Multicast Solicitations.

e) **Windows Server 2003**

   NDP Advertisement packets sent by the Windows Server 2003 system used null values for the Traffic Class and Flow Label fields in the IPv6 packet, 255 for the Hop Limit field in the IPv6 packet, and a null value for the Code field in the ICMPv6 datagram. The Solicited and Override flags were set in the ICMPv6 portion of the packet for responses to both Unicast and Multicast Solicitations.

f) **Windows Server 2008**

   NDP Advertisement packets sent by the Windows Server 2008 system used null values for the Traffic Class and Flow Label fields in the IPv6 packet, 255 for the Hop Limit field in the IPv6 packet, and a null value for the Code field in the ICMPv6 datagram. The Solicited and Override flags were set in the ICMPv6 portion of the packet for responses to both Unicast and Multicast Solicitations.

g) **Windows XP**

   NDP Advertisement packets sent by the Windows XP system used null values for the Traffic Class and Flow Label fields in the IPv6 packet, 255 for the Hop Limit field in the IPv6 packet, and a null value for the Code field in the ICMPv6 datagram. The Solicited and Override flags were set in the ICMPv6 portion of the packet for responses to both Unicast and Multicast Solicitations.

2) **What IPv6 Address is used as the Source?**

   a) **CentOS 5.5**

      All NDP Advertisements (unicast and multicast) sent by the Ubuntu 10.10 system used the Target address from the Solicitation packet as the IPv6 Source address.

   b) **Mac OS X**

      All NDP Advertisements (unicast and multicast) sent by the Mac OS X system used the interface's link-local scope address as the IPv6 Source rather than the requested Target address.
3) **Is the Target Link Layer Address option used in the ICMPv6 datagram?**

a) **CentOS 5.5**
   The CentOS 5.5 system included the Target link-layer address option in the ICMPv6 datagram only when generating a response to a multicast NDP Solicitation. All responses to unicast NDP Solicitations excluded all ICMPv6 options.

b) **Mac OS X**
   The Mac OS X system included the Target link-layer address option in the ICMPv6 datagram only when generating a response to a multicast NDP Solicitation. All responses to unicast NDP Solicitations excluded all ICMPv6 options.

c) **Ubuntu 10.10**
   The Ubuntu 10.10 system included the Target link-layer address option in the ICMPv6 datagram only when generating a response to a multicast NDP Solicitation. All responses to unicast NDP Solicitations excluded all ICMPv6 options.

d) **Windows 7**
   The Windows 7 system included the Target link-layer address option in the ICMPv6 datagram when generating a response to both multicast and unicast NDP Solicitations.

e) **Windows Server 2003**
   The Windows Server 2003 system included the Target link-layer address option in the ICMPv6 datagram when generating a response to both multicast and unicast NDP Solicitations.

f) **Windows Server 2008**
   The Windows Server 2008 system included the Target link-layer address option in the ICMPv6 datagram when generating a response to both multicast and unicast NDP Solicitations.
ICMPv6 datagram when generating a response to both multicast and unicast NDP Solicitations.

\( g \) \textit{Windows XP}

The Windows XP system included the Target link-layer address option in the ICMPv6 datagram when generating a response to both multicast and unicast NDP Solicitations.

11.1.2.3 \textbf{Handling of Incoming NDP Solicitations}

1. \textbf{Does the system add the Source address to its own NDP table based on this Solicitation?}
   
   \( a \) \textit{CentOS 5.5}
   
   Yes, the Source IPv6 and Source Link Layer addresses of the received NDP Solicitation were added to the system's NDP cache with a Stale state. This behavior is consistent with the recommendations by IETF RFC4861, which says that an entry should be created if the Solicitation is not from an unspecified address and the Source Link Layer Address option is present.

   \( b \) \textit{Mac OS X}
   
   Yes, the Source IPv6 and Source Link Layer addresses of the received NDP Solicitation were added to the system's NDP cache with a Stale state. This behavior is consistent with the recommendations by IETF RFC4861, which says that an entry should be created if the Solicitation is not from an unspecified address and the Source Link Layer Address option is present.

   \( c \) \textit{Ubuntu 10.10}
   
   Yes, the Source IPv6 and Source Link Layer addresses of the received NDP Solicitation were added to the system's NDP cache with a Stale state. This behavior is consistent with the recommendations by IETF RFC4861, which says that an entry should be created if the Solicitation is not from an unspecified address and the Source Link Layer Address option is present.

   \( d \) \textit{Windows 7}
   
   Yes, the Source IPv6 and Source Link Layer addresses of the received NDP Solicitation were added to the system's NDP cache with a Stale state. This behavior is consistent with the recommendations by IETF RFC4861, which says that an entry should be created if the Solicitation is not from an unspecified address and the Source Link Layer Address option is present.

   \( e \) \textit{Windows Server 2003}
   
   Yes, the Source IPv6 and Source Link Layer addresses of the received NDP Solicitation were added to the system's NDP cache with a STALE state. This behavior is consistent with the recommendations by IETF RFC4861, which says that an entry should be created if the Solicitation is not from an unspecified address and the Source Link Layer Address option is present.
f) Windows Server 2008
   Yes, the Source IPv6 and Source Link Layer addresses of the received NDP Solicitation were added to the system's NDP cache with a Stale state. This behavior is consistent with the recommendations by IETF RFC4861, which says that an entry should be created if the Solicitation is not from an unspecified address and the Source Link Layer Address option is present.

g) Windows XP
   Yes, the Source IPv6 and Source Link Layer addresses of the received NDP Solicitation were added to the system's NDP cache with a STALE state. This behavior is consistent with the recommendations by IETF RFC4861, which says that an entry should be created if the Solicitation is not from an unspecified address and the Source Link Layer Address option is present.

2. Does this Solicitation packet cause the target system to reciprocate with an NDP Solicitation to the test system?
   a) CentOS 5.5
      Once the Stale entry was added to the NDP cache, the CentOS 5.5 system attempted to bring the entry to a Reachable state by sending its own unicast NDP Solicitation packets to the testing workstation. This reciprocating discovery occurred even when there was no other traffic to be sent to the workstation.

   b) Mac OS X
      Once the Stale entry was added to the NDP cache, the Mac OS X system attempted to bring the entry to a Reachable state by sending its own unicast NDP Solicitation packets to the testing workstation. This reciprocating discovery occurred even when there was no other traffic to be sent to the workstation.

   c) Ubuntu 10.10
      Once the Stale entry was added to the NDP cache, the Ubuntu 10.10 system attempted to bring the entry to a Reachable state by sending its own unicast NDP Solicitation packets to the testing workstation. This reciprocating discovery occurred even when there was no other traffic to be sent to the workstation.

   d) Windows 7
      Once the Stale entry was added to the NDP cache, the Windows 7 system attempted to bring the entry to a Reachable state by sending its own unicast NDP Solicitation packets to the testing workstation. This reciprocating discovery occurred even when there was no other traffic to be sent to the workstation.

   e) Windows Server 2003
      No traffic was generated as a result of the received NDP Solicitation packet. When the Stale cache entry was present, unicast NDP Solicitations were only sent when other packets needed to be sent to the testing workstation.
Once the Stale entry was added to the NDP cache, the Windows Server 2008 system attempted to bring the entry to a Reachable state by sending its own unicast NDP Solicitation packets to the testing workstation. This reciprocating discovery occurred even when there was no other traffic to be sent to the workstation.

No traffic was generated as a result of the received NDP Solicitation packet. When the Stale cache entry was present, unicast NDP Solicitations were only sent when other packets needed to be sent to the testing workstation.

3. **Will the system respond to malformed Solicitations?**
   a) ICMPv6 Code field not 0.
      1. CentOS 5.5
         No, an Advertisement was not sent.
      2. Mac OS X
         No, an Advertisement was not sent.
      3. Ubuntu 10.10
         No, an Advertisement was not sent.
      4. Windows 7
         No, an Advertisement was not sent.
      5. Windows Server 2003
         No, an Advertisement was not sent.
         No, an Advertisement was not sent.
      7. Windows XP
         No, an Advertisement was not sent.
   b) ICMPv6 Incorrect Checksum.
      1. CentOS 5.5
         No, an Advertisement was not sent.
      2. Mac OS X
         No, an Advertisement was not sent.
      3. Ubuntu 10.10
         No, an Advertisement was not sent.
      4. Windows 7
         No, an Advertisement was not sent.
      5. Windows Server 2003
         No, an Advertisement was not sent.
         No, an Advertisement was not sent.
7. Windows XP
   No, an Advertisement was not sent.

c) ICMPv6 Source Link Layer Address option missing.
1. CentOS 5.5
   Yes, a unicast Advertisement was sent but only after the system found the proper MAC address through its own multicast NDP Solicitation.
2. Mac OS X
   Yes, a unicast Advertisement was sent but only after the system found the proper MAC address through its own multicast NDP Solicitation.
3. Ubuntu 10.10
   Yes, a unicast Advertisement was sent but only after the system found the proper MAC address through its own multicast NDP Solicitation.
4. Windows 7
   Yes, a unicast Advertisement was sent but only after the system found the proper MAC address through its own multicast NDP Solicitation.
5. Windows Server 2003
   No, an Advertisement was not sent.
   Yes, a unicast Advertisement was sent but only after the system found the proper MAC address through its own multicast NDP Solicitation.
7. Windows XP
   No, an Advertisement was not sent.

d) ICMPv6 Source Link Layer Address Option incorrect.
1. CentOS 5.5
   Yes, a unicast Advertisement was sent to the Solicitation's Source Link Layer Address value (rather than the Source MAC address).
2. Mac OS X
   Yes, a unicast Advertisement was sent to the Solicitation's Source Link Layer Address value (rather than the Source MAC address).
3. Ubuntu 10.10
   Yes, a unicast Advertisement was sent to the Solicitation's Source Link Layer Address value (rather than the Source MAC address).
4. Windows 7
   Yes, a unicast Advertisement was sent to the Solicitation's Source Link Layer Address value (rather than the Source MAC address).
5. Windows Server 2003
   Yes, a unicast Advertisement was sent to the Solicitation's Source Link Layer Address value (rather than the Source MAC address).
6. **Windows Server 2008**
   Yes, a unicast Advertisement was sent to the Solicitation's Source Link Layer Address value (rather than the Source MAC address).

7. **Windows XP**
   Yes, a unicast Advertisement was sent to the Solicitation's Source Link Layer Address value (rather than the Source MAC address).

e) IPv6 Destination to All Nodes (not Solicited Node).
   1. **CentOS 5.5**
      Yes, a unicast Advertisement was sent as normal.
   2. **Mac OS X**
      Yes, a unicast Advertisement was sent as normal.
   3. **Ubuntu 10.10**
      Yes, a unicast Advertisement was sent as normal.
   4. **Windows 7**
      Yes, a unicast Advertisement was sent as normal.
   5. **Windows Server 2003**
      Yes, a unicast Advertisement was sent as normal.
   6. **Windows Server 2008**
      Yes, a unicast Advertisement was sent as normal.
   7. **Windows XP**
      Yes, a unicast Advertisement was sent as normal.

f) IPv6 Destination to incorrect Solicited Node.
   1. **CentOS 5.5**
      No, an Advertisement was not sent.
   2. **Mac OS X**
      No, an Advertisement was not sent.
   3. **Ubuntu 10.10**
      No, an Advertisement was not sent.
   4. **Windows 7**
      No, an Advertisement was not sent.
   5. **Windows Server 2003**
      No, an Advertisement was not sent.
   6. **Windows Server 2008**
      No, an Advertisement was not sent.
   7. **Windows XP**
      No, an Advertisement was not sent.
Correlating IPv6 Addresses for Network Situational Awareness  

Jason Froehlich

---

**g) IPv6 Hop Limit field not 255.**

1. *CentOS 5.5*
   
   No, an Advertisement was not sent.

2. *Mac OS X*
   
   No, an Advertisement was not sent.

3. *Ubuntu 10.10*
   
   No, an Advertisement was not sent.

4. *Windows 7*
   
   No, an Advertisement was not sent.

5. *Windows Server 2003*
   
   No, an Advertisement was not sent.

6. *Windows Server 2008*
   
   No, an Advertisement was not sent.

7. *Windows XP*
   
   No, an Advertisement was not sent.

---

**h) IPv6 Flow Label field not 0.**

1. *CentOS 5.5*
   
   Yes, a unicast Advertisement was sent as normal, with a null Flow Label field.

2. *Mac OS X*
   
   Yes, a unicast Advertisement was sent as normal, with a null Flow Label field.

3. *Ubuntu 10.10*
   
   Yes, a unicast Advertisement was sent as normal, with a null Flow Label field.

4. *Windows 7*
   
   Yes, a unicast Advertisement was sent as normal, with a null Flow Label field.

5. *Windows Server 2003*
   
   Yes, a unicast Advertisement was sent as normal, with a null Flow Label field.

6. *Windows Server 2008*
   
   Yes, a unicast Advertisement was sent as normal, with a null Flow Label field.

7. *Windows XP*
   
   Yes, a unicast Advertisement was sent as normal, with a null Flow Label field.

---

**i) IPv6 Traffic Class field not 0.**

1. *CentOS 5.5*
   
   Yes, a unicast Advertisement was sent as normal, with a null Traffic Class field.

2. *Mac OS X*
   
   Yes, a unicast Advertisement was sent as normal, with a null Traffic Class field.

3. *Ubuntu 10.10*
   
   Yes, a unicast Advertisement was sent as normal, with a null Traffic Class field.

4. *Windows 7*

---

- 91 -
Yes, a unicast Advertisement was sent as normal, with a null Traffic Class field.

5. **Windows Server 2003**
   Yes, a unicast Advertisement was sent as normal, with a null Traffic Class field.

6. **Windows Server 2008**
   Yes, a unicast Advertisement was sent as normal, with a null Traffic Class field.

7. **Windows XP**
   Yes, a unicast Advertisement was sent as normal, with a null Traffic Class field.

j) Ethernet Destination to Broadcast.
   1. **CentOS 5.5**
      Yes, a unicast Advertisement was sent as normal.

   2. **Mac OS X**
      Yes, a unicast Advertisement was sent as normal.

   3. **Ubuntu 10.10**
      Yes, a unicast Advertisement was sent as normal.

   4. **Windows 7**
      Yes, a unicast Advertisement was sent as normal.

   5. **Windows Server 2003**
      Yes, a unicast Advertisement was sent as normal.

   6. **Windows Server 2008**
      Yes, a unicast Advertisement was sent as normal.

   7. **Windows XP**
      Yes, a unicast Advertisement was sent as normal.

k) Ethernet Destination to incorrect Solicited Node.
   1. **CentOS 5.5**
      No, an Advertisement was not sent.

   2. **Mac OS X**
      No, an Advertisement was not sent.

   3. **Ubuntu 10.10**
      No, an Advertisement was not sent.

   4. **Windows 7**
      No, an Advertisement was not sent.

   5. **Windows Server 2003**
      No, an Advertisement was not sent.

   6. **Windows Server 2008**
      No, an Advertisement was not sent.

   7. **Windows XP**
      No, an Advertisement was not sent.
11.1.2.4 Handling of Incoming NDP Advertisements

1. Does the system accept unsolicited NDP Advertisements when no cache entry exists?
   a) CentOS 5.5
      No, the CentOS 5.5 system will not accept an unexpected unsolicited NDP Advertisement packet when an appropriate entry is not found in the system's NDP cache.
   b) Mac OS X
      Unknown, procedures used to test this system did not properly generate unsolicited Advertisements as outlined by IETF RFC4861.
   c) Ubuntu 10.10
      No, the Ubuntu 10.10 system will not accept an unexpected unsolicited NDP Advertisement packet when an appropriate entry is not found in the system's NDP cache.
   d) Windows 7
      No, the Windows 7 system will not accept an unexpected unsolicited NDP Advertisement packet when an appropriate entry is not found in the system's NDP cache.
   e) Windows Server 2003
      No, the Windows Server 2003 system will not accept an unexpected unsolicited NDP Advertisement packet when an appropriate entry is not found in the system's NDP cache.
   f) Windows Server 2008
      No, the Windows Server 2008 system will not accept an unexpected unsolicited NDP Advertisement packet when an appropriate entry is not found in the system's NDP cache.
   g) Windows XP
      No, the Windows XP system will not accept an unexpected unsolicited NDP Advertisement packet when an appropriate entry is not found in the system's NDP cache.

2. Will the system accept malformed Advertisements? (solicited, non-gratuitous)
   a) ICMPv6 Code field not 0.
      1. CentOS 5.5
         No, the advertised MAC was not added to system's NDP cache.
      2. Mac OS X
         No, the advertised MAC was not added to system's NDP cache.
      3. Ubuntu 10.10
         No, the advertised MAC was not added to system's NDP cache.
      4. Windows 7
         No, the advertised MAC was not added to system's NDP cache.
5. Windows Server 2003
   No, the advertised MAC was not added to system's NDP cache.

   No, the advertised MAC was not added to system's NDP cache.

7. Windows XP
   No, the advertised MAC was not added to system's NDP cache.

b) ICMPv6 Incorrect Checksum.
   1. CentOS 5.5
      No, the advertised MAC was not added to system's NDP cache.

   2. Mac OS X
      No, the advertised MAC was not added to system's NDP cache.

   3. Ubuntu 10.10
      No, the advertised MAC was not added to system's NDP cache.

   4. Windows 7
      No, the advertised MAC was not added to system's NDP cache.

   5. Windows Server 2003
      No, the advertised MAC was not added to system's NDP cache.

      No, the advertised MAC was not added to system's NDP cache.

   7. Windows XP
      No, the advertised MAC was not added to system's NDP cache.

c) ICMPv6 Target Link Layer Address option missing.
   1. CentOS 5.5
      No, the advertised MAC was not added to system's NDP cache.

   2. Mac OS X
      No, the advertised MAC was not added to system's NDP cache.

   3. Ubuntu 10.10
      No, the advertised MAC was not added to system's NDP cache.

   4. Windows 7
      No, the advertised MAC was not added to system's NDP cache.

   5. Windows Server 2003
      No, the advertised MAC was not added to system's NDP cache.

      No, the advertised MAC was not added to system's NDP cache.

   7. Windows XP
      No, the advertised MAC was not added to system's NDP cache.
d) ICMPv6 Target Link Layer Address option incorrect.

1. **CentOS 5.5**
   Yes, the Advertisement's Target Link Layer Address value was added to the system's NDP cache (rather than the Source MAC address).

2. **Mac OS X**
   Yes, the Advertisement's Target Link Layer Address value was added to the system's NDP cache (rather than the Source MAC address).

3. **Ubuntu 10.10**
   Yes, the Advertisement's Target Link Layer Address value was added to the system's NDP cache (rather than the Source MAC address).

4. **Windows 7**
   Yes, the Advertisement's Target Link Layer Address value was added to the system's NDP cache (rather than the Source MAC address).

5. **Windows Server 2003**
   Yes, the Advertisement's Target Link Layer Address value was added to the system's NDP cache (rather than the Source MAC address).

6. **Windows Server 2008**
   Yes, the Advertisement's Target Link Layer Address value was added to the system's NDP cache (rather than the Source MAC address).

7. **Windows XP**
   Yes, the Advertisement's Target Link Layer Address value was added to the system's NDP cache (rather than the Source MAC address).

e) ICMPv6 Flag field combinations (Router, Solicited, Override).

1. **Router only:**
   a) **CentOS 5.5**
      The CentOS 5.5 system did accept the NDP Advertisement packet and used the Target Link Layer Address value to send the ping request. However, the IPv6 and MAC addresses were added to the NDP cache with an INCOMPLETE state, forcing an immediate series of NDP Solicitations.

   b) **Mac OS X**
      Yes, the system does accept the NDP Advertisement packet and use the Target Link Layer Address value to send the ping request. However, the system's NDP cache was not checked, so it is unknown if the entry was stored.

   c) **Ubuntu 10.10**
      The Ubuntu 10.10 system did accept the NDP Advertisement packet and used the Target Link Layer Address value to send the ping request. However, the IPv6 and MAC addresses were added to the NDP cache with an INCOMPLETE state, forcing an immediate series of NDP Solicitations.
d) **Windows 7**  
The Windows 7 system did accept the NDP Advertisement packet and used the Target Link Layer Address value to send the ping request. However, the IPv6 and MAC addresses were added to the NDP cache with an INCOMPLETE state, forcing an immediate series of NDP Solicitations.

e) **Windows Server 2003**  
The Windows Server 2003 system did accept the NDP Advertisement packet and used the Target Link Layer Address value to send the ping request. However, the IPv6 and MAC addresses were added to the NDP cache with an INCOMPLETE state, forcing an immediate series of NDP Solicitations.

f) **Windows Server 2008**  
The Windows Server 2008 system did accept the NDP Advertisement packet and used the Target Link Layer Address value to send the ping request. However, the IPv6 and MAC addresses were added to the NDP cache with an INCOMPLETE state, forcing an immediate series of NDP Solicitations.

g) **Windows XP**  
The Windows XP system did accept the NDP Advertisement packet and used the Target Link Layer Address value to send the ping request. However, the IPv6 and MAC addresses were added to the NDP cache with an INCOMPLETE state, forcing an immediate series of NDP Solicitations.

2. Override only:
   a) **CentOS 5.5**  
The CentOS 5.5 system did accept the NDP Advertisement packet and used the Target Link Layer Address value to send the ping request. However, the IPv6 and MAC addresses were added to the NDP cache with an INCOMPLETE state, forcing an immediate series of NDP Solicitations.

   b) **Mac OS X**  
Yes, the system does accept the NDP Advertisement packet and use the Target Link Layer Address value to send the ping request. However, the system's NDP cache was not checked, so it is unknown if the entry was stored.

   c) **Ubuntu 10.10**  
The Ubuntu 10.10 system did accept the NDP Advertisement packet and used the Target Link Layer Address value to send the ping request. However, the IPv6 and MAC addresses were added to the NDP cache with an INCOMPLETE state, forcing an immediate series of NDP Solicitations.

   d) **Windows 7**  
The Windows 7 system did accept the NDP Advertisement packet and used the Target Link Layer Address value to send the ping request. However, the IPv6 and MAC addresses were added to the NDP cache with an INCOMPLETE state, forcing an immediate series of NDP Solicitations.
Correlating IPv6 Addresses for Network Situational Awareness

Jason Froehlich

3. Not Solicited (none):
   a) CentOS 5.5
      The CentOS 5.5 system did accept the NDP Advertisement packet and used the Target Link Layer Address value to send the ping request. However, the IPv6 and MAC addresses were added to the NDP cache with an INCOMPLETE state, forcing an immediate series of NDP Solicitations.

   b) Mac OS X
      Yes, the system does accept the NDP Advertisement packet and use the Target Link Layer Address value to send the ping request. However, the system's NDP cache was not checked, so it is unknown if the entry was stored.

   c) Ubuntu 10.10
      The Ubuntu 10.10 system did accept the NDP Advertisement packet and used the Target Link Layer Address value to send the ping request. However, the IPv6 and MAC addresses were added to the NDP cache with an INCOMPLETE state, forcing an immediate series of NDP Solicitations.

   d) Windows 7
      The Windows 7 system did accept the NDP Advertisement packet and used the Target Link Layer Address value to send the ping request. However, the IPv6 and MAC addresses were added to the NDP cache with an INCOMPLETE state, forcing an immediate series of NDP Solicitations.

   e) Windows Server 2003
      The Windows Server 2003 system did accept the NDP Advertisement packet and used the Target Link Layer Address value to send the ping request. However, the IPv6 and MAC addresses were added to the NDP cache with an INCOMPLETE state, forcing an immediate series of NDP Solicitations.

   f) Windows Server 2008
      The Windows Server 2008 system did accept the NDP Advertisement packet and used the Target Link Layer Address value to send the ping request. However, the IPv6 and MAC addresses were added to the NDP cache with an INCOMPLETE state, forcing an immediate series of NDP Solicitations.
f) **Windows Server 2008**
   The Windows Server 2008 system did accept the NDP Advertisement packet and used the Target Link Layer Address value to send the ping request. However, the IPv6 and MAC addresses were added to the NDP cache with an INCOMPLETE state, forcing an immediate series of NDP Solicitations.

g) **Windows XP**
   The Windows XP system did accept the NDP Advertisement packet and used the Target Link Layer Address value to send the ping request. However, the IPv6 and MAC addresses were added to the NDP cache with an INCOMPLETE state, forcing an immediate series of NDP Solicitations.

4. Override and Solicited:
   a) **CentOS 5.5**
      Yes, the NDP Advertisement was accepted and the IPv6 and MAC addresses added to the NDP cache with a Reachable state.

   b) **Mac OS X**
      Yes, the system does accept the NDP Advertisement packet and use the Target Link Layer Address value to send the ping request. However, the system's NDP cache was not checked, so it is unknown if the entry was stored.

   c) **Ubuntu 10.10**
      Yes, the NDP Advertisement was accepted and the IPv6 and MAC addresses added to the NDP cache with a Reachable state.

   d) **Windows 7**
      Yes, the NDP Advertisement was accepted and the IPv6 and MAC addresses added to the NDP cache with a Reachable state.

   e) **Windows Server 2003**
      Yes, the NDP Advertisement was accepted and the IPv6 and MAC addresses added to the NDP cache with a Reachable state.

   f) **Windows Server 2008**
      Yes, the NDP Advertisement was accepted and the IPv6 and MAC addresses added to the NDP cache with a Reachable state.

   g) **Windows XP**
      Yes, the NDP Advertisement was accepted and the IPv6 and MAC addresses added to the NDP cache with a Reachable state.

f) **IPv6 Multicast to All Nodes.**
   1. **CentOS 5.5**
      No, the advertised MAC was not added to system's NDP cache.

   2. **Mac OS X**
      No, the advertised MAC was not added to system's NDP cache.
3. *Ubuntu 10.10*
   No, the advertised MAC was not added to system's NDP cache.

4. *Windows 7*
   No, the advertised MAC was not added to system's NDP cache.

5. *Windows Server 2003*
   No, the advertised MAC was not added to system's NDP cache.

6. *Windows Server 2008*
   No, the advertised MAC was not added to system's NDP cache.

7. *Windows XP*
   No, the advertised MAC was not added to system's NDP cache.

g) IPv6 Multicast to Solicited Node Address.
1. *CentOS 5.5*
   No, the advertised MAC was not added to system's NDP cache.

2. *Mac OS X*
   No, the advertised MAC was not added to system's NDP cache.

3. *Ubuntu 10.10*
   No, the advertised MAC was not added to system's NDP cache.

4. *Windows 7*
   No, the advertised MAC was not added to system's NDP cache.

5. *Windows Server 2003*
   No, the advertised MAC was not added to system's NDP cache.

6. *Windows Server 2008*
   No, the advertised MAC was not added to system's NDP cache.

7. *Windows XP*
   No, the advertised MAC was not added to system's NDP cache.

h) IPv6 Source not matching ICMPv6 Target.
1. *CentOS 5.5*
   Yes, the Advertisement's Target Address value was added to the system's NDP cache with a Reachable state (rather than the Source IPv6 address).

2. *Mac OS X*
   Yes, the Advertisement's Target Address value was added to the system's NDP cache with a Reachable state (rather than the Source IPv6 address).

3. *Ubuntu 10.10*
   Yes, the Advertisement's Target Address value was added to the system's NDP cache with a Reachable state (rather than the Source IPv6 address).

4. *Windows 7*
   Yes, the Advertisement's Target Address value was added to the system's NDP cache with a Reachable state (rather than the Source IPv6 address).
5. **Windows Server 2003**  
   Yes, the Advertisement's Target Address value was added to the system's NDP cache with a Reachable state (rather than the Source IPv6 address).

6. **Windows Server 2008**  
   Yes, the Advertisement's Target Address value was added to the system's NDP cache with a Reachable state (rather than the Source IPv6 address).

7. **Windows XP**  
   Yes, the Advertisement's Target Address value was added to the system's NDP cache with a Reachable state (rather than the Source IPv6 address).

i) IPv6 Hop Limit field not 255.  
   1. **CentOS 5.5**  
      No, the advertised MAC was not added to system's NDP cache.

   2. **Mac OS X**  
      No, the advertised MAC was not added to system's NDP cache.

   3. **Ubuntu 10.10**  
      No, the advertised MAC was not added to system's NDP cache.

   4. **Windows 7**  
      No, the advertised MAC was not added to system's NDP cache.

   5. **Windows Server 2003**  
      No, the advertised MAC was not added to system's NDP cache.

   6. **Windows Server 2008**  
      No, the advertised MAC was not added to system's NDP cache.

   7. **Windows XP**  
      No, the advertised MAC was not added to system's NDP cache.

j) IPv6 Flow Label field not 0.  
   1. **CentOS 5.5**  
      Yes, the IPv6 and MAC addresses were added to the system's NDP cache with a Reachable state.

   2. **Mac OS X**  
      Yes, the IPv6 and MAC addresses were added to the system's NDP cache with a Reachable state.

   3. **Ubuntu 10.10**  
      Yes, the IPv6 and MAC addresses were added to the system's NDP cache with a Reachable state.

   4. **Windows 7**  
      Yes, the IPv6 and MAC addresses were added to the system's NDP cache with a Reachable state.
5. **Windows Server 2003**
   Yes, the IPv6 and MAC addresses were added to the system's NDP cache with a Reachable state.

6. **Windows Server 2008**
   Yes, the IPv6 and MAC addresses were added to the system's NDP cache with a Reachable state.

7. **Windows XP**
   Yes, the IPv6 and MAC addresses were added to the system's NDP cache with a Reachable state.

k) IPv6 Traffic Class field not 0.
   1. **CentOS 5.5**
      Yes, the IPv6 and MAC addresses were added to the system's NDP cache with a Reachable state.

   2. **Mac OS X**
      Yes, the IPv6 and MAC addresses were added to the system's NDP cache with a Reachable state.

   3. **Ubuntu 10.10**
      Yes, the IPv6 and MAC addresses were added to the system's NDP cache with a Reachable state.

   4. **Windows 7**
      Yes, the IPv6 and MAC addresses were added to the system's NDP cache with a Reachable state.

   5. **Windows Server 2003**
      Yes, the IPv6 and MAC addresses were added to the system's NDP cache with a Reachable state.

   6. **Windows Server 2008**
      Yes, the IPv6 and MAC addresses were added to the system's NDP cache with a Reachable state.

   7. **Windows XP**
      Yes, the IPv6 and MAC addresses were added to the system's NDP cache with a Reachable state.

11.1.3 **Multiple IPv6 Addresses Results**

1. **Will the system generate an address for each prefix in the Router Advertisement, plus a link-local scope address?**
   a) **CentOS 5.5**
      Yes, the system did generate a Stateless Autoconfiguration address for each of the subnet prefixes included in the Router Advertisement packets. The addresses generated by this system are shown in Figure 11.1.1 in Section 11.1.1.1.
b) *Mac OS X*
Yes, the system did generate a Stateless Autoconfiguration address for each of the subnet prefixes included in the Router Advertisement packets. The addresses generated by this system are shown in Figure 11.1.2 in Section 11.1.1.2.

c) *Ubuntu 10.10*
Yes, the system did generate a Stateless Autoconfiguration address for each of the subnet prefixes included in the Router Advertisement packets. The addresses generated by this system are shown in Figure 11.1.3 in Section 11.1.1.3.

d) *Windows 7*
Yes, the system did generate an IPv6 address for each of the subnet prefixes included in the Router Advertisement packets. However, these were not Stateless Address Autoconfiguration addresses based on the interface's Ethernet MAC address. Instead, the system generates a separate persistent randomized interface identifier that it uses in place of the MAC address in the SAA configuration process. The addresses generated by this system are shown in Figure 11.1.4 in Section 11.1.1.4.

e) *Windows Server 2003*
Yes, the system did generate a Stateless Autoconfiguration address for each of the subnet prefixes included in the Router Advertisement packets. The addresses generated by this system are shown in Figure 11.1.8 in Section 11.1.1.5.

f) *Windows Server 2008*
Yes, the system did generate an IPv6 address for each of the subnet prefixes included in the Router Advertisement packets. However, these were not Stateless Address Autoconfiguration addresses based on the interface's Ethernet MAC address. Instead, the system generates a separate persistent randomized interface identifier that it uses in place of the MAC address in the SAA configuration process. The addresses generated by this system are shown in Figure 11.1.9 in Section 11.1.1.6.

g) *Windows XP*
Yes, the system did generate a Stateless Autoconfiguration address for each of the subnet prefixes included in the Router Advertisement packets. The addresses generated by this system are shown in Figure 11.1.13 in Section 11.1.1.7.

2. **When pinging nodes on the same network segment, what Source addresses are used?**

a) *CentOS 5.5*
The host system always used an address in the same subnet as the intended destination when sending traffic to a node on the local network segment; e.g., when sending to 2001:db8:1111:2222::1, it used its address in the 2001:db8:1111:2222::/64 subnet. The same Source address was used for both the NDP Solicitation packet and the Ping Request packet sent to each destination.

b) *Mac OS X*
The host system always used an address in the same subnet as the intended destination...
when sending traffic to a node on the local network segment; e.g., when sending to 2001:db8:1111:2222::1, it used its address in the 2001:db8:1111:2222::/64 subnet. The same Source address was used for both the NDP Solicitation packet and the Ping Request packet sent to each destination.

c) **Ubuntu 10.10**

The host system always used an address in the same subnet as the intended destination when sending traffic to a node on the local network segment; e.g., when sending to 2001:db8:1111:2222::1, it used its address in the 2001:db8:1111:2222::/64 subnet. The same Source address was used for both the NDP Solicitation packet and the Ping Request packet sent to each destination.

d) **Windows 7**

During my observations, the host system usually used an address in the same subnet as the intended destination when sending traffic to a node on the local network segment; e.g., when sending to 2001:db8:1111:2222::1, it used its address in the 2001:db8:1111:2222::/64 subnet. However, in a few cases when pinging a link-local address, the system's Unique-Local address was used as the Source for the NDP Solicitation (this behavior could not be reliably reproduced). In all cases, the Ping Request packet used an address in the same subnet as the Destination address.

e) **Windows Server 2003**

The host system always used an address in the same subnet as the intended destination when sending traffic to a node on the local network segment; e.g., when sending to 2001:db8:1111:2222::1, it used its address in the 2001:db8:1111:2222::/64 subnet. The same Source address was used for both the NDP Solicitation packet and the Ping Request packet sent to each destination.

f) **Windows Server 2008**

The host system always used an address in the same subnet as the intended destination when sending traffic to a node on the local network segment; e.g., when sending to 2001:db8:1111:2222::1, it used its address in the 2001:db8:1111:2222::/64 subnet. The same Source address was used for both the NDP Solicitation packet and the Ping Request packet sent to each destination.

g) **Windows XP**

The host system always used an address in the same subnet as the intended destination when sending traffic to a node on the local network segment; e.g., when sending to 2001:db8:1111:2222::1, it used its address in the 2001:db8:1111:2222::/64 subnet. The same Source address was used for both the NDP Solicitation packet and the Ping Request packet sent to each destination.
3. When pinging nodes on a different network segment, what Source addresses are used?

a) CentOS 5.5

The CentOS 5.5 host used its Global-scope address 2001:db8:1111:2222:a00:27ff:fe5:87 as the Source IPv6 address for all traffic to destinations beyond its local network, except those beginning with FC00. For destinations beginning with FC00, its Unique-local address fc00:aaaa:aaaa:2222:a00:27ff:fe5:87 was used as the Source IPv6 address. In each case, the same address was used for both the NDP Solicitation packets to the gateway and the Ping Request Packets to the destination.

b) Mac OS X

The host system appeared to use the non-link-local address that was numerically closest to the intended Destination address as the Source address when sending traffic to a node not on the local network segment. For example, when sending Ping Requests to 9999::1 (not actually active on the network, so no reply traffic) the host's address in the fc00:aaaa:aaaa:2222::/64 subnet was used as the Source. However, when sending Ping Requests to 0099::1 (again, not active on the network) the host's address in the 2001:db8:1111:2222::/64 subnet was used as the Source. In both of these sample cases, the routable address numerically closest to the Destination address was used (2001:: is closer to 0099:: than FC00::, and FC00:: is closer to 9999:: than 2001::). The same Source address was used for both the NDP Solicitation packet to the gateway and the Ping Request packet to the destination.

c) Ubuntu 10.10

The Ubuntu 10.10 host used its Global-scope address 2001:db8:1111:2222:a00:27ff:fea4:8965 as the Source IPv6 address for all traffic to destinations beyond its local network, except those beginning with FC00. For destinations beginning with FC00, its Unique-local address fc00:aaaa:aaaa:2222:a00:27ff:fea4:8965 was used as the Source IPv6 address. In each case, the same address was used for both the NDP Solicitation packets to the gateway and the Ping Request Packets to the destination.

d) Windows 7

The host system appeared to use the non-link-local address that was numerically closest to the intended Destination address as the Source address when sending traffic to a node not on the local network segment. For example, when sending Ping Requests to 9999::1 (not actually active on the network, so no reply traffic) the host's address in the FC00:AAAA:AAAA:2222::/64 subnet was used as the Source. However, when sending Ping Requests to 0099::1 (again, not active on the network) the host's address in the 2001:DB8:1111:2222::/64 subnet was used as the Source. In both of these sample cases, the routable address numerically closest to the Destination address was used (2001:: is closer to 0099:: than FC00::, and FC00:: is closer to 9999::.
than 2001::). However, in a few cases the system did use its link-local address as the Source of the NDP Solicitation to the gateway. At all other times, the same Source address was used for both the NDP Solicitation packet to the gateway and the Ping Request packet to the destination.

e) **Windows Server 2003**
The host system appeared to use the non-link-local address that was numerically closest to the intended Destination address as the Source address when sending traffic to a node not on the local network segment. For example, when sending Ping Requests to 9999::1 (not actually active on the network, so no reply traffic) the host's address in the FC00:AAAA:AAAA:2222::/64 subnet was used as the Source. However, when sending Ping Requests to 0099::1 (again, not active on the network) the host's address in the 2001:DB8:1111:2222::/64 subnet was used as the Source. In both of these sample cases, the routable address numerically closest to the Destination address was used (2001:: is closer to 0099:: than FC00::, and FC00:: is closer to 9999:: than 2001::). The same Source address was used for both the NDP Solicitation packet to the gateway and the Ping Request packet to the destination.

f) **Windows Server 2008**
The host system appeared to use the non-link-local address that was numerically closest to the intended Destination address as the Source address when sending traffic to a node not on the local network segment. For example, when sending Ping Requests to 9999::1 (not actually active on the network, so no reply traffic) the host's address in the FC00:AAAA:AAAA:2222::/64 subnet was used as the Source. However, when sending Ping Requests to 0099::1 (again, not active on the network) the host's address in the 2001:DB8:1111:2222::/64 subnet was used as the Source. In both of these sample cases, the routable address numerically closest to the Destination address was used (2001:: is closer to 0099:: than FC00::, and FC00:: is closer to 9999:: than 2001::). The same Source address was used for both the NDP Solicitation packet to the gateway and the Ping Request packet to the destination.

g) **Windows XP**
The host system appeared to use the non-link-local address that was numerically closest to the intended Destination address as the Source address when sending traffic to a node not on the local network segment. For example, when sending Ping Requests to 9999::1 (not actually active on the network, so no reply traffic) the host's address in the FC00:AAAA:AAAA:2222::/64 subnet was used as the Source. However, when sending Ping Requests to 0099::1 (again, not active on the network) the host's address in the 2001:DB8:1111:2222::/64 subnet was used as the Source. In both of these sample cases, the routable address numerically closest to the Destination address was used (2001:: is closer to 0099:: than FC00::, and FC00:: is closer to 9999:: than 2001::). The same Source address was used for both the NDP Solicitation packet to the gateway and the Ping Request packet to the destination.
11.1.4 Privacy Extensions Results

1. Are Privacy Extensions for IPv6 supported?
   a) CentOS 5.5
      Yes, Privacy Extensions are supported by the CentOS 5.5 system.
   b) Mac OS X
      Yes, Privacy Extensions are supported by the Mac OS X 10.6 system.
   c) Ubuntu 10.10
      Yes, Privacy Extensions are supported by the Ubuntu 10.10 system.
   d) Windows 7
      Yes, Privacy Extensions are supported by the Windows 7 system.
   e) Windows Server 2003
      Yes, Privacy Extensions are supported by the Windows Server 2003 system.
   f) Windows Server 2008
      Yes, Privacy Extensions are supported by the Windows Server 2008 system.
   g) Windows XP
      Yes, Privacy Extensions are supported by the Windows XP system.

2. Are Privacy Extensions for IPv6 enabled by default?
   a) CentOS 5.5
      No, the Privacy Extensions functionality is not enabled by default. Instead, hosts will only generate a single IPv6 address per observed network prefix following the Stateless Address Autoconfiguration guidelines.
   b) Mac OS X
      No, the Privacy Extensions functionality is not enabled by default. Instead, hosts will only generate a single IPv6 address per observed network prefix following the Stateless Address Autoconfiguration guidelines.
   c) Ubuntu 10.10
      No, the Privacy Extensions functionality is not enabled by default. Instead, hosts will only generate a single IPv6 address per observed network prefix following the Stateless Address Autoconfiguration guidelines.
   d) Windows 7
      Yes, Privacy Extensions functionality is enabled by default on Windows 7 systems. In addition to the addresses created by Stateless Address Autoconfiguration, temporary addresses will be created for each of the network prefixes in observed Router Advertisements.
   e) Windows Server 2003
      No, Privacy Extensions functionality is not enabled by default. Instead, hosts will only generate a single IPv6 address per observed network prefix following the Stateless Address Autoconfiguration guidelines.
f) **Windows Server 2008**  
No, Privacy Extensions functionality is not enabled by default. Instead, hosts will only generate a single IPv6 address per observed network prefix following the Stateless Address Autoconfiguration guidelines.

g) **Windows XP**  
Yes, Privacy Extensions functionality is enabled by default on Windows XP systems. In addition to the addresses created by Stateless Address Autoconfiguration, temporary addresses will be created for each of the network prefixes in observed Router Advertisements.

3. **What procedure is used to activate/deactivate the use of Privacy Extensions for IPv6?**

a) **CentOS 5.5**

Enabling Privacy Extensions functionality on CentOS 5.5 is a multi-step process. First, modify the `/etc/sysctl.conf` file to include the following lines:

```plaintext
net.ipv6.conf.wlan0.use_tempaddr = 2  
net.ipv6.conf.eth0.use_tempaddr = 2  
net.ipv6.conf.all.use_tempaddr = 2  
net.ipv6.conf.default.use_tempaddr = 2
```

Second, execute the following commands in the terminal:

```plaintext
sudo sysctl set net.ipv6.conf.wlan0.use_tempaddr=2  
sudo sysctl set net.ipv6.conf.eth0.use_tempaddr=2  
sudo sysctl set net.ipv6.conf.all.use_tempaddr=2  
sudo sysctl set net.ipv6.conf.default.use_tempaddr=2
```

Finally, after a reboot each network interface will automatically generate a pseudorandom temporary address in addition to the Stateless Address Autoconfiguration addresses, as shown in Figure 11.1.14. Similarly, Privacy Extensions functionality can be disabled by removing the lines from `/etc/sysctl.conf`, issuing the same commands with “=0” instead of “=2”, and rebooting the system. No graphical utilities are provided by the OS to modify Privacy Extensions settings.

```
[root@localhost ~]# ifconfig eth0  
eth0   Link encap:Ethernet  HWaddr 08:00:27:F5:00:87  
inet addr:10.100.111.12  Bcast:10.100.111.12  
       Mask:255.255.255.255  
       inet6 addr: fc00:aaaa:aaaa:2222:3cf0:7d0c:fb7e:519a/64  
       Scope:Global  
       inet6 addr: fc00:aaaa:aaaa:2222:a00:27ff:fe5f:87/64  
       Scope:Global  
       inet6 addr: 2001:db8:1111:2222:3cf0:7d0c:fb7e:519a/64  
       Scope:Global  
       inet6 addr: 2001:db8:1111:2222:a00:27ff:fe5f:87/64  
       Scope:Global  
       inet6 addr: fe80::a00:27ff:fe5f:87/64  
       Scope:Link  
       UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
```
Correlating IPv6 Addresses for Network Situational Awareness

Jason Froehlich

RX packets: 75 errors: 0 dropped: 0 overruns: 0 frame: 0
TX packets: 45 errors: 0 dropped: 0 overruns: 0 carrier: 0
collisions: 0 txqueuelen: 1000
RX bytes: 10746 (5.4 KiB) TX bytes: 7351 (8.3 KiB)

Figure 11.1.14 CentOS 5.5 IPv6 Network Addresses with Privacy Extensions enabled

b) Mac OS X
Privacy Extensions functionality on Mac OS X 10.6 can be enabled by issuing the command `sudo sysctl -w net.inet6.ip6.use_tempaddr=1` in the terminal. Once this command has been issued, the network interfaces must be reinitialized by either rebooting the system or issuing the `sudo ifconfig en0 down` and `sudo ifconfig en0 up` commands in the terminal. After the network has been reinitialized, the interface will automatically generate a pseudorandom temporary address in addition to the Stateless Address Autoconfiguration address, as shown in Figure 11.1.15. Similarly, Privacy Extensions can be disabled by issuing the `sudo sysctl -w net.inet6.ip6.use_tempaddr=0` command in the terminal and reinitializing the network. No graphical utilities are provided by the OS to modify Privacy Extensions settings.

bash-3.2$ ifconfig en0
en0: flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX,MULTICAST> mtu 1500
inet6 fe80::21b:63ff:fe9a:2454%en0 prefixlen 64 scopeid 0x4
inet 169.254.166.50 netmask 0xffff0000 broadcast 169.254.255.255
inet6 2001:db8:1111:2222:21b:63ff:fe9a:2454 prefixlen 64 autoconf
temporary
inet6 fc00:aaaa:aaaa:2222:21b:63ff:fe9a:2454 prefixlen 64 autoconf
temporary
inet6 fc00:aaaa:aaaa:2222:3152:db6b:49ed:d324 prefixlen 64 autoconf
temporary
ether 00:1b:63:9a:1b:67
media: autoselect (100baseTX <half-duplex>) status: active
supported media: autoselect 10baseT/UTP <half-duplex> 10baseT/UTP <full-duplex>
<full-duplex> 10baseT/UTP <full-duplex,hw-loopback> 10baseT/UTP <full-duplex,
flow-control> 100baseTX <half-duplex> 100baseTX <full-duplex>
100baseTX <full-duplex,hw-loopback> 100baseTX <full-duplex,flow-control>
1000baseT <full-duplex> 1000baseT <full-duplex,hw-loopback> 1000baseT
<full-duplex,flow-control> none

Figure 11.1.15 Mac OS X Network Information with Privacy Extensions enabled

c) Ubuntu 10.10
Enabling Privacy Extensions functionality on Ubuntu 10.10 is a multi-step process. First, modify the `/etc/sysctl.conf` file to include the following lines:

```plaintext
net.ipv6.conf.wlan0.use_tempaddr = 2
net.ipv6.conf.eth0.use_tempaddr = 2
net.ipv6.conf.all.use_tempaddr = 2
net.ipv6.conf.default.use_tempaddr = 2
```

Second, execute the following commands in the terminal:
Finally, after a reboot each network interface will automatically generate a pseudorandom temporary address in addition to the Stateless Address Autoconfiguration addresses, as shown in Figure 11.1.16. Similarly, Privacy Extensions functionality can be disabled by removing the lines from /etc/sysctl.conf, issuing the same commands with “=0” instead of “=2”, and rebooting the system. No graphical utilities are provided by the OS to modify Privacy Extensions settings.

```bash
root@Survey-Ubuntu1010:~# ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 08:00:27:a4:89:65
          inet6 addr: fc00:aaaa:aaaa:2222:79bf:21ed:1cb7:521c/64 Scope:Global
          inet6 addr: fc00:aaaa:aaaa:2222:a00:27ff:fea4:8965/64 Scope:Global
          inet6 addr: fe80::a00:27ff:fea4:8965/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:41 errors:0 dropped:0 overruns:0 frame:0
          TX packets:50 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:6282 (6.2 KB)  TX bytes:10696 (10.6 KB)
```

Figure 11.1.16. Ubuntu 10.10 IPv6 Network Addresses with Privacy Extensions enabled

d) Windows 7

Privacy Extensions functionality on Windows 7 can be disabled by issuing the command `netsh interface ipv6 set privacy state=disabled` in the command prompt. After a system reboot, the network interfaces should automatically disable the use of any existing temporary addresses. Similarly, Privacy Extensions functionality can be re-enabled by issuing the `netsh interface ipv6 set privacy state=enabled` command in the command prompt. No graphical utilities are provided by the OS to modify Privacy Extensions settings.

e) Windows Server 2003

Privacy Extensions functionality on Windows Server 2003 can be enabled by issuing the command `netsh interface ipv6 set privacy enabled` in the command prompt. After a system reboot, the network interfaces should automatically generate temporary addresses for all known network prefixes, as shown in Figure 11.1.17. Similarly, Privacy Extensions functionality can be disabled by issuing the `netsh interface ipv6 set privacy disabled` command in the command prompt. No graphical utilities are provided by the OS to modify Privacy Extensions settings.
Correlating IPv6 Addresses for Network Situational Awareness

Jason Froehlich

C:\Documents and Settings\Administrator>ipconfig
Windows IP Configuration
Ethernet adapter Local Area Connection:
    Connection-specific DNS Suffix . :
    Autoconfiguration IP Address...: 169.254.86.7
    Subnet Mask ...............: 255.255.0.0
    IP Address................:
fco0:aaaa:aaaa:2222:f5db:f025:5abc:67e3
    IP Address................:
fco0:aaaa:aaaa:2222:a00:27ff:fe9c:78bc
    IP Address................:
    IP Address................:
2001:db8:1111:2222:a00:27ff:fe9c:78bc
    IP Address................:
    Default Gateway ............: fe80::219:6ff:fe23:8569%4
Tunnel adapter Teredo Tunneling Pseudo-Interface:
    ...
Tunnel adapter Automatic Tunneling Pseudo-Interface:
    ...

Figure 11.1.17. Windows Server 2003 IPv6 Network Addresses with Privacy Extensions enabled

f) Windows Server 2008

Privacy Extensions functionality on Windows Server 2008 can be enabled by issuing the command `netsh interface ipv6 set privacy state=enabled` in the command prompt. After a system reboot, the network interfaces should automatically generate existing temporary addresses for all known network prefixes, as shown in Figure 11.1.18. Similarly, Privacy Extensions functionality can be disabled by issuing the `netsh interface ipv6 set privacy state=disabled` command in the command prompt. No graphical utilities are provided by the OS to modify Privacy Extensions settings.

C:\Users\Administrator>ipconfig /all
Windows IP Configuration
    Host Name ......................: WIN-DGQB215TLFX
    Primary Dns Suffix ............:
    Node Type .....................: Hybrid
    IP Routing Enabled............: No
    WINS Proxy Enabled............: No
Ethernet adapter Local Area Connection:
    Connection-specific DNS Suffix . :
    Description ...................: Intel(R) PRO/1000 MT Desktop Adapter
    Physical Address ..............: 08-00-27-F0-AE-2C
    DHCP Enabled..................: Yes
    Autoconfiguration Enabled......: Yes
    IPv6 Address..................:
    IPv6 Address..................:
fco0:aaaa:aaaa:2222:f9ca:6411:b670:7715(Preferred)
    Temporary IPv6 Address........:
Correlating IPv6 Addresses for Network Situational Awareness

Temporary IPv6 Address. . . . . . :
fcc0:aaaa:aaaa:2222:e451:cb1d:1cbb:1b92(Preferred)
Link-local IPv6 Address . . . . . :
fe80::f9ca:6411:b670:7715%10(Preferred)
Autoconfiguration IPv4 Address. . . : 169.254.119.21(Preferred)
Subnet Mask . . . . . . . . . . . : 255.255.0.0
Default Gateway . . . . . . . . . : fe80::219:6ff:fe23:8569%10
DNS Servers . . . . . . . . . . . : fec0:0:0:ffff::1%1
fel0:0:0:ffff::2%1
fel0:0:0:ffff::3%1
NetBIOS over Tcpip. . . . . . . . : Enabled
Tunnel adapter Local Area Connection* 8:
...

Figure 11.1.18. Windows Server 2008 IPv6 Network Addresses with Privacy Extensions enabled

g) Windows XP
Privacy Extensions functionality on Windows XP can be disabled by issuing the command `netsh interface ipv6 set privacy disabled` in the command prompt. After a system reboot, the network interfaces should automatically disable the use of any existing temporary addresses. Similarly, Privacy Extensions functionality can be re-enabled by issuing the `netsh interface ipv6 set privacy enabled` command in the command prompt. No graphical utilities are provided by the OS to modify Privacy Extensions settings.

4. What are the default parameters relating to Privacy Extensions?

a) CentOS 5.5
Once Privacy Extensions functionality is enabled, the system will use a Valid Lifetime of seven days and a Preferred Lifetime of one day for each generated temporary address. This is consistent with the recommendations provided by IETF RFC 4861.

b) Mac OS X
Once Privacy Extensions functionality is enabled, the system will use a maximum Valid Lifetime of seven days and a maximum Preferred Lifetime of one day for each generated temporary address. This is consistent with the recommendations provided by IETF RFC 4861.

c) Ubuntu 10.10
Once Privacy Extensions functionality is enabled, the system will use a maximum Valid Lifetime of seven days and a maximum Preferred Lifetime of one day for each generated temporary address. This is consistent with the recommendations provided by IETF RFC 4861.

d) Windows 7
When Privacy Extensions functionality is enabled, the system will use a maximum Valid Lifetime of seven days and a maximum Preferred Lifetime of one day for each generated temporary address. This is consistent with the recommendations provided by IETF RFC 4861.
e) **Windows Server 2003**

When Privacy Extensions functionality is enabled, the system will use a maximum Valid Lifetime of seven days and a maximum Preferred Lifetime of one day for each generated temporary address. This is consistent with the recommendations provided by IETF RFC4861.

f) **Windows Server 2008**

When Privacy Extensions functionality is enabled, the system will use a maximum Valid Lifetime of seven days and a maximum Preferred Lifetime of one day for each generated temporary address. This is consistent with the recommendations provided by IETF RFC4861.

g) **Windows XP**

When Privacy Extensions functionality is enabled, the system will use a maximum Valid Lifetime of seven days and a maximum Preferred Lifetime of one day for each generated temporary address. This is consistent with the recommendations provided by IETF RFC4861.

5. **What parameters can be customized?**

a) **CentOS 5.5**

Aside from enabling and disabling Privacy Extensions functionality, the `sysctl` utility allows you to change the Valid and Preferred Lifetimes used by the generated temporary addresses. The `sysctl` parameters relating to Privacy Extensions for IPv6 are shown in Figure 11.1.19.

```
net.ipv6.conf.eth0.temp_prefered_lft = 86400
net.ipv6.conf.eth0.temp_valid_lft = 604800
net.ipv6.conf.eth0.use_tempaddr = 0
net.ipv6.conf.default.temp_prefered_lft = 86400
net.ipv6.conf.default.temp_valid_lft = 604800
net.ipv6.conf.default.use_tempaddr = 0
net.ipv6.conf.all.temp_prefered_lft = 86400
net.ipv6.conf.all.temp_valid_lft = 604800
net.ipv6.conf.all.use_tempaddr = 0
net.ipv6.conf.lo.temp_prefered_lft = 86400
net.ipv6.conf.lo.temp_valid_lft = 604800
net.ipv6.conf.lo.use_tempaddr = 0
```

**Figure 11.1.19. CentOS 5.5 `sysctl` parameters relating to Privacy Extensions for IPv6**

b) **Mac OS X**

Aside from enabling and disabling Privacy Extensions functionality, the `sysctl` utility allows you to change the Valid and Preferred Lifetimes used by the generated temporary addresses. The `sysctl` parameters relating to IPv6 are shown in Figure 11.1.20, with those relating to Privacy Extensions in bold type.

```
net.inet6.ip6.use_tempaddr: 1
net.inet6.ip6.temppltime: 86400
```
Correlating IPv6 Addresses for Network Situational Awareness

Jason Froehlich

Figure 11.1.20. Mac OS X's `sysctl` parameters relating to Privacy Extensions for IPv6

c) **Ubuntu 10.10**

Aside from enabling and disabling Privacy Extensions functionality, the `sysctl` utility allows you to change the Valid and Preferred Lifetimes used by the generated temporary addresses. The `sysctl` parameters relating to Privacy Extensions for IPv6 are shown in Figure 11.1.21.

```
net.ipv6.conf.all.use_tempaddr = 2
net.ipv6.conf.all.temp_valid_lft = 604800
net.ipv6.conf.all.temp_preferred_lft = 86400
net.ipv6.conf.default.use_tempaddr = 2
net.ipv6.conf.default.temp_valid_lft = 604800
net.ipv6.conf.default.temp_preferred_lft = 86400
net.ipv6.conf.lo.use_tempaddr = 2
net.ipv6.conf.lo.temp_valid_lft = 604800
net.ipv6.conf.lo.temp_preferred_lft = 86400
net.ipv6.conf.eth0.use_tempaddr = 2
net.ipv6.conf.eth0.temp_valid_lft = 604800
net.ipv6.conf.eth0.temp_preferred_lft = 86400
```

Figure 11.1.21. Ubuntu 10.10 `sysctl` parameters relating to Privacy Extensions for IPv6

d) **Windows 7**

Aside from enabling and disabling Privacy Extensions functionality, the `netsh` utility allows you to change the maximum Valid and Preferred Lifetimes and the number of Duplicate Address Detection attempts attempted for the generated temporary addresses, as well as the regenerate time, random time, and max random time parameters used for temporary address regeneration. The `netsh` parameters relating to Privacy Extensions for IPv6 are shown in Figure 11.1.22.

```
C:\Users\jason>netsh interface ipv6 show privacy
Querying active state...
Temporary Address Parameters
---------------------------------------------
Use Temporary Addresses             : enabled
Duplicate Address Detection Attempts: 5
Maximum Valid Lifetime              : 7d
Maximum Preferred Lifetime          : 1d
Regenerate Time                     : 5s
Maximum Random Time                 : 10m
Random Time                         : 0s
```

Figure 11.1.22. Windows 7 `netsh` parameters relating to Privacy Extensions for IPv6

e) **Windows Server 2003**

Aside from enabling and disabling Privacy Extensions functionality, the `netsh` utility allows you to change the maximum Valid and Preferred Lifetimes and the number of Duplicate Address Detection attempts attempted for the generated temporary
addresses, as well as the regenerate time, random time, and max random time parameters used for temporary address regeneration. The netsh parameters relating to Privacy Extensions for IPv6 are shown in Figure 11.1.23.

```
C:\Documents and Settings\Administrator>netsh interface ipv6 show privacy
Querying active state...
Temporary Address Parameters
---------------------------------------------
Use Temporary Addresses : enabled
Duplicate Address Detection Attempts: 5
Maximum Valid Lifetime   : 7d
Maximum Preferred Lifetime: 1d
Regenerate Time          : 5s
Maximum Random Time      : 10m
Random Time              : 2m47s
```

Figure 11.1.23. Windows Server 2003 netsh parameters relating to Privacy Extensions for IPv6

f) **Windows Server 2008**

Aside from enabling and disabling Privacy Extensions functionality, the netsh utility allows you to change the maximum Valid and Preferred Lifetimes and the number of Duplicate Address Detection attempts attempted for the generated temporary addresses, as well as the regenerate time, random time, and max random time parameters used for temporary address regeneration. The netsh parameters relating to Privacy Extensions for IPv6 are shown in Figure 11.1.24.

```
C:\Users\Administrator>netsh interface ipv6 show privacy
Querying active state...
Temporary Address Parameters
---------------------------------------------
Use Temporary Addresses : enabled
Duplicate Address Detection Attempts: 5
Maximum Valid Lifetime   : 7d
Maximum Preferred Lifetime: 1d
Regenerate Time          : 5s
Maximum Random Time      : 10m
Random Time              : 0s
```

Figure 11.1.24. Windows Server 2008 netsh parameters relating to Privacy Extensions for IPv6

g) **Windows XP**

Aside from enabling and disabling Privacy Extensions functionality, the netsh utility allows you to change the maximum Valid and Preferred Lifetimes and the number of Duplicate Address Detection attempts attempted for the generated temporary addresses, as well as the regenerate time, random time, and max random time parameters used for temporary address regeneration. The netsh parameters relating to Privacy Extensions for IPv6 are shown in Figure 11.1.25.
6. Are Temporary addresses created for each network prefix and scope?
   a) CentOS 5.5
      The CentOS 5.5 system will generate temporary addresses for all known network
      prefixes except the link-local scope prefix.
   b) Mac OS X
      The Mac OS X system will generate temporary addresses for all known network
      prefixes except the link-local scope prefix.
   c) Ubuntu 10.10
      The Ubuntu 10.10 system will generate temporary addresses for all known network
      prefixes except the link-local scope prefix.
   d) Windows 7
      The Windows 7 system will generate temporary addresses for all known network
      prefixes except the link-local scope prefix.
   e) Windows Server 2003
      The Windows Server 2003 system will generate temporary addresses for all known
      network prefixes except the link-local scope prefix.
   f) Windows Server 2008
      The Windows Server 2008 system will generate temporary addresses for all known
      network prefixes except the link-local scope prefix.
   g) Windows XP
      The Windows XP system will generate temporary addresses for all known network
      prefixes except the link-local scope prefix.

7. Are the Temporary addresses used for all outbound traffic on all scopes?
   a) CentOS 5.5
      Yes, when Privacy Extensions functionality is enabled, all outbound traffic and
      connections originating from the host will use the newest temporary address for the
      appropriate network prefix as the Source address.
   b) Mac OS X
      Yes, when Privacy Extensions functionality is enabled, all outbound traffic and
connections originating from the host will use the newest temporary address for the appropriate network prefix as the Source address.

c) Ubuntu 10.10
Yes, when Privacy Extensions functionality is enabled, all outbound traffic and connections originating from the host will use the newest temporary address for the appropriate network prefix as the Source address.

d) Windows 7
Yes, when Privacy Extensions functionality is enabled, all outbound traffic and connections originating from the host will use the newest temporary address for the appropriate network prefix as the Source address.

e) Windows Server 2003
Yes, when Privacy Extensions functionality is enabled, all outbound traffic and connections originating from the host will use the newest temporary address for the appropriate network prefix as the Source address.

f) Windows Server 2008
Yes, when Privacy Extensions functionality is enabled, all outbound traffic and connections originating from the host will use the newest temporary address for the appropriate network prefix as the Source address.

g) Windows XP
Yes, when Privacy Extensions functionality is enabled, all outbound traffic and connections originating from the host will use the newest temporary address for the appropriate network prefix as the Source address.
### 11.2 Database Design

The IPAC system I have demonstrated utilizes three different types of MySQL databases for storing information related to IPv6 address, MAC address, and Ethernet switch port use on the network. In this section I will discuss the purpose, structure, and design considerations of each of these databases. Sample commands for creating these databases on a MySQL server are provided in Section 11.3.2.

Within each of the following database tables, IPv6 addresses are stored as a 32-byte char value and MAC addresses are stored as 12-byte char values. During development, much consideration was placed on how to store these items. Unlike IPv4 addresses, the numeric value of an IPv6 address is too large to be contained by a single numeric SQL construct. Considering this, I chose to store the address in its 32-character string format (`char`), omitting all separators within the address and including all zeros that might be omitted when represented normally. For consistency, the same approach was taken in the storage of 12-character MAC addresses. All time values are stored as a `double` value in the POSIX time format.

It is possible for a single SQL server instance to provide all of the databases needed by the IPAC system. In such a case, a separate database would need to be created for each Agent, Server, and Trap Handler, and each process would need to be configured with the address of the SQL server and the name of its respective database. However, in my implementation each Agent, Server, and Trap Handler node implements a separate MySQL server locally. This was done to mitigate any potential bandwidth and latency issues caused by constantly sending SQL queries to remote servers.

#### 11.2.1 Agent

The first set of databases consists of those used by the Agent processes. Each Agent uses its own database to store information on the IP addresses it considers currently active on the subnet it monitors, as well as a historical recording of the instances in which an IP address is seen in NDP traffic. The structure of the Agent database is shown in Figure 11.2.1.

For each IPv6-MAC address pair active on the network, the `active` table holds the IPv6 and MAC addresses, the time the pair was last seen, and the number of times the Agent has tried to contact the pair without a receiving a valid response (stored as a `tinyint`). These values are
stored by the database, instead of being resident within the Agent process, to allow for quick resumption of the Agent service in instances when the Agent is offline for some period of time. Rather than having to relearn each address by observing NDP traffic over time, this table's information provides the Agent with a baseline of addresses to begin with. While it will not know any IPv6-MAC address pairings that have joined the network in the Agent's absence, it will have a record of those that persist from before it went offline.

The 'records' table in the Agent database provides a record of every instance in which an IPv6-MAC address pair was observed and verified by the Agent. It consists of the IPv6 and MAC addresses, sighting type (stored as a tinyint, with values of 10 and 11 indicating the Agent considers the IP-MAC pair active and 20 indicating the Agent considers the IP-MAC pair as having left the network), and the time that the observation occurred.

![Figure 11.2.1. IPAC Agent Database Structure](image)

### 11.2.2 Trap Handler

A separate database is used by the Trap Handler process. It consists of a single table: 'mac_notifications'. Each record stores the information obtained from a single CAM table notification sent by Ethernet switches. This includes the observed MAC address, the source switch's IPv4 address (stored as a 15-character varchar in dotted-quad notation), the corresponding switch port and VLAN (each stored as a 4-character char), the event's operation (stored as a tinyint, with value of 1 for MAC Learned and 2 for MAC Removed), and the time the notification was received by the Trap Handler. The structure of the Trap Handler database is shown in Figure 11.2.2.
Correlating IPv6 Addresses for Network Situational Awareness

Figure 11.2.2. IPAC Trap Handler Database Structure

11.2.3 Server

The final database is used by the Server process, and it consists of four separate tables that store information about the IPAC deployment as well as data summarizing address and port use. The structure of the Server database is shown in Figure 11.2.3.

The 'agents' table stores information about each of the IPAC Agents in the enterprise's network, including the Agent's unique ID (smallint), descriptive name (128-character varchar), a value indicating whether the Agent is currently active (tinyint, with 0 indicating inactive and 1 indicating active), IPv6 address of the Agent, IPv4 addresses of the SQL server with the Agent's database (stored as a 15-character varchar in dotted-quad notation), as well as the username, password, and database name used to access the Agent's database (each stored as a 128-character varchar). This information is used by the Server process to connect to and retrieve sighting records from each Agent's database. If an Agent is temporarily or permanently offline, it can be marked as such in this table to prevent the Server from attempting to query it.

The second table in the Server database is 'subnets', which contains information about the IPv6 subnets in the enterprise's network. Each record consists of the IPv6 prefix (32-character char) and mask (tinyint) of a different subnet, as well as the ID of the Agent serving its portion of the network (smallint, foreign key bound to the 'agents' table). This table is used by several of the provided Interface scripts to obtain binding information from the Server. These scripts will check the 'subnets' table for the prefix of the address they are attempting to resolve. If the prefix is found, they will know which Agent would have observed the address in question, and can apply filters to their queries of the 'bindings' table accordingly. If the address's prefix is not included in this table, then it is from a network beyond the
organization's control and cannot be resolved with the IPAC system.

The third table, 'bindings', contains summary information on the pairings of IPv6 to MAC addresses. Each record identifies a session in which a MAC address was known to be connected to a specific IPv6 address. Fields in this table include the IPv6 and MAC addresses, ID of the Agent observing the pairing (smallint, foreign key bound to the 'agents' table), times that the pairing was first and last seen, and a code that signifies whether the pairing continued at the point when the record was last updated (stored as a tinyint). A code value of 10 means that the pairings still existed when the Server last updated the database entry, while a value of 20 signifies that the Agent considers the IP-MAC pair to have left the network. In the latter case the ending time represents the last time the pair was observed by the agent before it no longer responded to NDP Solicitations, while in the former case the ending time represents the last positive sighting of the pair. As the Server processes Agent sighting records, it will update existing binding records' ending times to reflect those of the Agent databases' records. If the same IPv6-MAC address pair is seen after it has been marked as inactive by the Agent, a separate binding record will be created for the new session.

The last table in the Server database is 'portBindings'. Similar in function to the 'bindings' table, this facility identifies sessions in which a MAC address was known to be connected to a specific Ethernet switch port. Each record contains the MAC address, the IPv4 address of the switch (stored as a 12-character char in dotted-quad notation), the associated switch port and VLAN (each 4-character chars), the times the pairing was first and last seen, and a code that signifies the ending status of the pairing (tinyint). A code value of 10 is used when the record is generated as a result of a MAC Learned notification, signifying that the pairing still exists. A code value of 20 results from a MAC Removed notification, signifying that the MAC is no longer paired to that switch port. If a MAC Learned notification is received by the Trap Handler for the same MAC, switch, port, and VLAN as one already in the 'portBindings' table with a code of 10 (no MAC Removed notification was ever received), the existing record will be updated to have a code of 11 and an ending time of the new notification's timestamp, and a new record will be established with the new MAC Learned notification's timestamp as its starting time.
Figure 11.2.3. IPAC Server Database Structure
11.3 IPAC Implementation Instructions

11.3.1 Environment

The IPAC system was designed to be executed on a Linux system with the Python 2.6 scripting environment. Each module was tested and verified on Ubuntu 10.10 and CentOS 5.5 systems. Execution on a Microsoft Windows system with Python for Windows and the correct Python modules may be possible, but it has not been tested to any degree.

Non-Standard Python Modules

In addition to the Standard Library provided by Python 2.6, each IPAC module relies on several external Python modules to implement specific functionality. These modules must be installed and available in the module search path before the IPAC system will operate as intended. Because a single host may not implement each IPAC role, all of these modules may not be required on all systems. The Python modules required for each IPAC module are listed in Figure 11.3.1.

| IPAC Agent         | dpkt [13]                      |
|                   | MySQLdb [14]                  |
|                   | pypcap [39]                   |
| IPAC Trap Handler | MySQLdb [14]                  |
|                   | PySNMP [40]                   |
| IPAC Server       | MySQLdb [14]                  |
| IPAC Interface Scripts | dpkt [13] (ipac_pcaplog.py) |
|                   | Matplotlib [23] (ipac_graph.py) |
|                   | MySQLdb [14]                  |

Figure 11.3.1. External Python Module Requirements.

11.3.2 Database Implementation

A separate SQL database is required for each IPAC module. (If multiple instances of the same module are used, each instance will also require its own database.) The design and structure of these databases is discussed at length in Section 11.2. Sample commands for creating these databases on a MySQL 5.1 server are provided in Figures 11.3.2-11.3.4. Although implementing these databases on the same host that is running the module is not required, doing
so will prevent any latency issues provided by using a remote SQL server.

```sql
CREATE DATABASE IPAC_Agent;
CREATE USER 'ipac'@'localhost' IDENTIFIED BY 'T1gerCl@ws';
CREATE USER 'ipac'@'%' IDENTIFIED BY 'T1gerCl@ws';
GRANT ALL PRIVILEGES ON IPAC_Agent.* TO 'ipac'@'%' WITH GRANT OPTION;
GRANT ALL PRIVILEGES ON IPAC_Agent.* TO 'ipac'@'localhost' WITH GRANT OPTION;
USE IPAC_Agent;
CREATE TABLE records ( ipv6Address CHAR(32) NOT NULL, macAddress CHAR(12) NOT NULL, timestamp DOUBLE NOT NULL, type TINYINT NOT NULL, PRIMARY KEY (ipv6Address,timestamp) );
CREATE TABLE active ( ipv6Address char(32) NOT NULL, macAddress CHAR(12) NOT NULL, lastSeen DOUBLE NOT NULL, retryFailures TINYINT NOT NULL, PRIMARY KEY (ipv6Address,macAddress) );
```

**Figure 11.3.2. Sample MySQL Commands for IPAC Agent Database.**

```sql
CREATE DATABASE IPAC_SNMP;
CREATE USER 'ipac'@'localhost' IDENTIFIED BY 'T1gerCl@ws';
CREATE USER 'ipac'@'%' IDENTIFIED BY 'T1gerCl@ws';
GRANT ALL PRIVILEGES ON IPAC_SNMP.* TO 'ipac'@'%' WITH GRANT OPTION;
GRANT ALL PRIVILEGES ON IPAC_SNMP.* TO 'ipac'@'localhost' WITH GRANT OPTION;
USE IPAC_SNMP;
CREATE TABLE mac_notifications ( switchIPv4 VARCHAR(15) NOT NULL, macAddress CHAR(12) NOT NULL, port CHAR(4) NOT NULL, vlan CHAR(4) NOT NULL, operation TINYINT NOT NULL, timestamp DOUBLE NOT NULL, PRIMARY KEY (timestamp, operation, macAddress, switchIPv4) );
```

**Figure 11.3.3. Sample MySQL Commands for IPAC Trap Handler Database.**

```sql
CREATE DATABASE IPAC_Server;
CREATE USER 'ipac'@'localhost' IDENTIFIED BY 'T1gerCl@ws';
CREATE USER 'ipac'@'%' IDENTIFIED BY 'T1gerCl@ws';
GRANT ALL PRIVILEGES ON IPAC_Server.* TO 'ipac'@'%' WITH GRANT OPTION;
GRANT ALL PRIVILEGES ON IPAC_Server.* TO 'ipac'@'localhost' WITH GRANT OPTION;
CREATE TABLE agents ( agentID SMALLINT NOT NULL, name VARCHAR(128) NOT NULL, active TINYINT NOT NULL, ipv6Address CHAR(32), ipv4Address VARCHAR(15), username VARCHAR(128), password VARCHAR(128), dbName VARCHAR(128), PRIMARY KEY (agentID) );
CREATE TABLE bindings ( ipv6Address CHAR(32) NOT NULL, macAddress CHAR(12), agentID SMALLINT NOT NULL, startTime DOUBLE NOT NULL, endTime DOUBLE NOT NULL, endType TINYINT NOT NULL, PRIMARY KEY (ipv6Address, macAddress, agentID, startTime) );
CREATE TABLE portBindings ( macAddress CHAR(12) NOT NULL, switchID CHAR(15) NOT NULL, switchPort CHAR(4) NOT NULL, vlan CHAR(4) NOT NULL, startTime DOUBLE NOT NULL, endTime DOUBLE NOT NULL, endType TINYINT NOT NULL, PRIMARY KEY (macAddress, switchID, switchPort, vlan, startTime) );
CREATE TABLE subnets ( subnetID CHAR(32) NOT NULL, subnetMask TINYINT NOT NULL, agentID SMALLINT NOT NULL, PRIMARY KEY (subnetID, subnetMask), FOREIGN KEY (agentID) REFERENCES agents(agentID) );
```

**Figure 11.3.4. Sample MySQL Commands for IPAC Server Database.**
11.3.3 Agent Module Implementation

The IPAC Agent module is implemented as a daemonized process on the host system. The process is started, stopped, and checked through the `ipac_agent.py` script, as shown by the usage statement in Figure 11.3.5. Before the script is initialized, you must customize the constants in `ipac_agent_config.py` that specify the Agent's ID, the Agent database information (IPv4 address, user, password, and database name), its log and PID file locations, the network interface on which it will listen, and the IPv6 and MAC addresses it will use for generated traffic. The file paths specified for the log and PID files must lead to a directory for which the running user has read and write permissions. Because the process creates a raw socket listener on the network interface, the script must be run with superuser privileges. A separate IPAC Agent must be implemented on every local network within the organization.

**Required Files:**
- `ipac_agent.py`
- `ipac_agent_config.py`
- `ipac_daemon.py`
- `ipac_functions.py`
- `ipac_threads.py`

```
root@localhost:/IPAC$ python ipac_agent.py
usage: python ipac_agent.py start|status|stop|restart
```

Figure 11.3.5. IPAC Agent Usage.

11.3.4 Trap Handler Module Implementation

The IPAC Trap Handler module as implemented is a daemonized process on the host system. The process is started, stopped, and checked through the `ipac_traphandler.py` script, as shown by the usage statement in Figure 11.3.6. Before the script is initialized, you must customize the constants at the top of `ipac_traphandler.py` that specify the Trap Handler database information (IPv4 address, user, password, and database name) and its log and PID file locations. The file paths specified for the log and PID files must lead to a directory for which the running user has read and write permissions. Because the process creates a socket listener on a reserved UDP port, the script must be run with superuser privileges. A single IPAC Trap Handler can be used to handle messages from every Ethernet switch within the organization.

**Required Files:**
Correlating IPv6 Addresses for Network Situational Awareness

Jason Froehlich

**Figure 11.3.6. IPAC Trap Handler Usage.**

```bash
root@localhost:/IPAC$ python ipac_traphandler.py
usage: python ipac_traphandler.py start|status|stop|restart
```

**11.3.5 Server Module Implementation**

The IPAC Server module is implemented as a daemonized process on the host system. The process is started, stopped, and checked through the `ipac_server.py` script, as shown by the usage statement in Figure 11.3.7. Before the script is initialized, you must customize the constants in `ipac_server_config.py` that specify the Server database information (IPv4 address, user, password, and database name), the Trap Handler database information, and its log and PID file locations. The file paths specified for the log and PID files must lead to a directory for which the running user has read and write permissions. The Server database must also be pre-populated with information about every subnet and IPAC Agent on the organization's network. A single IPAC Server can be used to process the records from every Agent within the organization.

**Required Files:**
- `ipac_daemon.py`
- `ipac_functions.py`
- `ipac_server.py`
- `ipac_server_config.py`
- `ipac_trapsum.py`

```bash
root@localhost:/IPAC$ python ipac_server.py
usage: python ipac_server.py start|status|stop|restart
```

**Figure 11.3.7. IPAC Server Usage.**

**11.3.6 Interface Scripts Implementation**

The IPAC Interface scripts are implemented as standalone scripts that perform a single processing operation each time they are run. Before they are executed, you must modify the constants at the top of each script file to reflect the proper Server database information (IPv4 address, user, password, and database name) and year (`ipac_textlog.py` only).

When executing the `ipac_textlog.py` and `ipac_pcaplog.py` scripts, you must
specify as command line parameters the name of the file to be processed and the ID of the IPAC Agent serving the local network from which the log file/packet capture originated. The reports generated by these scrips will be written to the same directory as the specified file. When executing the ipac_query.py script, you must specify the IPAC Agent that created the records you want to examine, the operation you want to perform, and any optional parameters that would limit the query, as shown in the usage statement provided in Figure 7.2 in Section 7.4.1. The ipac_graph.py script does not require any additional parameters to be specified when executed.

**Required Files:**
- ipac_functions.py
- ipac_graph.py
- ipac_pcaplog.py
- ipac_query.py
- ipac_textlog.py
11.4 IPAC Source Code

11.4.1 IPAC System

11.4.1.1 ipac_agent.py

```python
#!/usr/bin/python
## IPAC Agent
## Jason Froehlich
## Rochester Institute of Technology
## Dept. of Network, Security, and System Administration
## May 17, 2011
## This program is a proof of concept being created for the partial
## fulfillment of the MS in Networking and Systems Administration from the
## Rochester Institute of Technology. All rights reserved.
## Daemon class based on the simple unix/linx daemon by Sander Marechal:
## http://www.jejik.com/articles/2007/02/a_simple_unix_linux_daemon_in_python/

## Module Imports
import binascii
import dpkt
import pcap
import string
import sys
import time
from ipac_agent_config import *
from ipac_daemon import Daemon
from ipac_functions import *
from ipac_threads import *
__author__ = "Jason Froehlich"
__copyright__ = "Copyright 2011, Jason Froehlich"
__version__ = "0.1.4"
__email__ = "jason.froehlich@alum.rit.edu"
__status__ = "Development"
class IPACAgentDaemon(Daemon):
    """
    Daemon class for the IPAC Agent process
    """
    def run(self):
        """
        Body of IPAC Agent process, called by start() or restart()
        """
        def processSource(src_ip, src_mac, ts):
            """
            Check and Update Database's Active List for IP/MAC, Generate Solicit
            """
            # Query list of active hosts for source IP address
            query = (""""SELECT * FROM active WHERE (ipv6Address = '%s');"") % (src_ip)
```

- 127 -
qResult = dbInterfaceThread.execute(query)
if qResult: ## IP exists in active list
    macs = []
    for entry in qResult:
        macs.append(entry[1])
        if macs.count(src_mac): ## Source MAC matches entry in list
            ## Generate Record
            recordQuery = ("""INSERT INTO records (ipv6Address, \nmacAddress, timestamp, type) values ('%s', '%s', %f, %d);"""
                % (src_ip, src_mac, ts, 10))
            recordResult = dbInterfaceThread.execute(recordQuery)
            ## Update Last Seen time in active list
            activeQuery = ("""UPDATE active SET lastSeen='%f', \nretryFailures=0 WHERE ipv6Address='%s' and \nmacAddress='%s'""" % (ts, src_ip, src_mac))
            activeResult = dbInterfaceThread.execute(activeQuery)
        else: ## Source MAC differs from all entries in active list
            qPrinter.printq("Warning: Source MAC different from",
                "known MAC")
            qPrinter.printq("IP: " + src_ip + ", Known MAC: " + `macs`
                + ",, Seen MAC: " + src_mac)
            ## Generate Multicast Solicit to IP
            results = ndSolicitMultiThread.generate(src_ip)
            mc_recent_solicits[result[results[0]]] = result[1]
    else: ## IP not in active list
        ## Generate Unicast Solicit to source MAC
        solicitResults = ndSolicitUniThread.generate(src_ip, src_mac)
startTime = time.time()
print "IPAC Agent Start"
print "  Start time: ", timeString(startTime)
sys.stdout.flush()

## Find IPv6 and MAC addresses of system
## Used to identify traffic going to/from us
## system_macs, system_ips : list of strings
system_macs = find_macs()
system_ips = find_ips()
print "  System info found"
sys.stdout.flush()

## Addresses with Recent Multicast Solicitations
## Used to throttle solicitations
## Dictionary: 'ipAddress':'lastSentTimestamp'
mc_recent_solicits = {} ## Counts of Received Packets by Types
## List: [ ND Solicitations, ND Advertisements ]
counts = [0, 0]

print "  Creating Threads ...",
sys.stdout.flush()
## Create Threads For Secondary Functions
## All defined in ipac_threads.py
## Queued Printer
qPrinter = QueuedPrinter()
## Maintains Records of Recently Sent Solicitations
replyWaiting = ReplyWaiting(qPrinter)
## Handles all queries to database
dbInterfaceThread = DatabaseInterface(FRESH_START, qPrinter)
## Generates Multicast ND Solicitations
ndSolicitMultiThread = NDMulticastGenerator(qPrinter, replyWaiting)
## Generates Unicat ND Solicitations
Correlating IPv6 Addresses for Network Situational Awareness

Jason Froehlich

ndSolicitUniThread = NDUnicastGenerator(qPrinter, replyWaiting)
## Maintains Records of Pending Incoming Advertisements
pendingProc = PendingProcessor(dbInterfaceThread, qPrinter)
## Resends Solicitations when Advertisements not received
solicitResender = SolicitationResender(replyWaiting,
            ndSolicitMultiThread,
            ndSolicitUniThread, qPrinter)
## Sends Solicitations to nodes not recently seen
activeRefresher = ActiveIPRefresher(dbInterfaceThread,
            ndSolicitUniThread, qPrinter)
## Regularly create MARKS in log file
logmarker = LogMarker(counts, qPrinter)

print "Threads Created"
sys.stdout.flush()

## Create Capture Interface
## Captures only ICMPv6 traffic
pc = pcap.pcap(SYSTEM_DEFAULT_IFACE, immediate=True)
pc.setfilter('icmp6') ## Berkely Packet Filter (BPF) format
pc.setnonblock()
print "  Capture interface created on", SYSTEM_DEFAULT_IFACE
sys.stdout.flush()

## PCAP information
if CAPTURE_INTERFACE_DEBUG:
    debugString = ( "Capture Interface Information:" +
    "\n  Interface: " + pc.name +
    "\n  PacketLength:" + pc.snaplen +
    "\n  L2HeaderLength:" + pc.dloff +
    "\n  Filter:" + pc.filter +
    "\n  FileDescriptor:" + pc.fd +
    "\n  NonBlocking:" + pc.getnonblock() )
qPrinter.printq(debugString)

print "  Capture interface created on", SYSTEM_DEFAULT_IFACE
sys.stdout.flush()

time.sleep(1) ## Wait for all threads to complete initialization
print "Waiting for packets...",
sys.stdout.flush()

## Process packets
for ts, pkt in pc:
    eth = dpkt.ethernet.Ethernet(pkt)
    type = `eth.data.data.type` ## ICMPv6 Packet Type
    src_mac = string.upper(binascii.b2a_hex(eth.src)) ## Source MAC Addr
    dst_mac = string.upper(binascii.b2a_hex(eth.dst)) ## Dest MAC Addr
    src_ip = string.upper(binascii.b2a_hex(eth.data.src)) ## Source IPv6
    dst_ip = string.upper(binascii.b2a_hex(eth.data.dst)) ## Dest IPv6

    ## NDP Solicitation (Request)
    if type == "$135$":
        counts[0] += 1
        payload = string.upper(binascii.b2a_hex(eth.data.data.data))
        target = payload[16:48] ## IPv6 address of Target

        if src_mac in system_macs: ## From us, ignore
            if (INCOMING_ICMPV6_TYPES_DEBUG or
                INCOMING_ICMPV6_FULL_DEBUG):
                qPrinter.printq("ND Solicit from us")
            if INCOMING_ICMPV6_FULL_DEBUG:
                debugString = ( "\n  Current Time: " + `time.time()` +
                "\n  Source MAC: " + src_mac +
                "\n  Target MAC: " + target +
                "\n  Source IPv6 Address: " + src_ip +
                "\n  Target IPv6 Address: " + target +
            else:
                qPrinter.printq("ND Solicit from us")
                debugString = ( "\n  Current Time: " + `time.time()` +
                "\n  Source MAC: " + src_mac +
                "\n  Target MAC: " + target +
                "\n  Source IPv6 Address: " + src_ip +
                "\n  Target IPv6 Address: " + target +

            else:
                qPrinter.printq("ND Solicit from us")
                debugString = ( "\n  Current Time: " + `time.time()` +
                "\n  Source MAC: " + src_mac +
                "\n  Target MAC: " + target +
                "\n  Source IPv6 Address: " + src_ip +
                "\n  Target IPv6 Address: " + target +

            else:
                qPrinter.printq("ND Solicit from us")
                debugString = ( "\n  Current Time: " + `time.time()` +
                "\n  Source MAC: " + src_mac +
                "\n  Target MAC: " + target +
                "\n  Source IPv6 Address: " + src_ip +
                "\n  Target IPv6 Address: " + target +

            else:
                qPrinter.printq("ND Solicit from us")
                debugString = ( "\n  Current Time: " + `time.time()` +
                "\n  Source MAC: " + src_mac +
                "\n  Target MAC: " + target +
                "\n  Source IPv6 Address: " + src_ip +
                "\n  Target IPv6 Address: " + target +

            else:
                qPrinter.printq("ND Solicit from us")
                debugString = ( "\n  Current Time: " + `time.time()` +
                "\n  Source MAC: " + src_mac +
                "\n  Target MAC: " + target +
                "\n  Source IPv6 Address: " + src_ip +
                "\n  Target IPv6 Address: " + target +

            else:
                qPrinter.printq("ND Solicit from us")
                debugString = ( "\n  Current Time: " + `time.time()` +
                "\n  Source MAC: " + src_mac +
                "\n  Target MAC: " + target +
                "\n  Source IPv6 Address: " + src_ip +
                "\n  Target IPv6 Address: " + target +

            else:
                qPrinter.printq("ND Solicit from us")
                debugString = ( "\n  Current Time: " + `time.time()` +
                "\n  Source MAC: " + src_mac +
                "\n  Target MAC: " + target +
                "\n  Source IPv6 Address: " + src_ip +
                "\n  Target IPv6 Address: " + target +

            else:
                qPrinter.printq("ND Solicit from us")
                debugString = ( "\n  Current Time: " + `time.time()` +
                "\n  Source MAC: " + src_mac +
                "\n  Target MAC: " + target +
                "\n  Source IPv6 Address: " + src_ip +
                "\n  Target IPv6 Address: " + target +

            else:
                qPrinter.printq("ND Solicit from us")
                debugString = ( "\n  Current Time: " + `time.time()` +
                "\n  Source MAC: " + src_mac +
                "\n  Target MAC: " + target +
                "\n  Source IPv6 Address: " + src_ip +
                "\n  Target IPv6 Address: " + target +

            else:
                qPrinter.printq("ND Solicit from us")
                debugString = ( "\n  Current Time: " + `time.time()` +
                "\n  Source MAC: " + src_mac +
                "\n  Target MAC: " + target +
                "\n  Source IPv6 Address: " + src_ip +
                "\n  Target IPv6 Address: " + target +

            else:
                qPrinter.printq("ND Solicit from us")
                debugString = ( "\n  Current Time: " + `time.time()` +
                "\n  Source MAC: " + src_mac +
                "\n  Target MAC: " + target +
                "\n  Source IPv6 Address: " + src_ip +
                "\n  Target IPv6 Address: " + target +

            else:
                qPrinter.printq("ND Solicit from us")
                debugString = ( "\n  Current Time: " + `time.time()` +
                "\n  Source MAC: " + src_mac +
                "\n  Target MAC: " + target +
                "\n  Source IPv6 Address: " + src_ip +
                "\n  Target IPv6 Address: " + target +

            else:
                qPrinter.printq("ND Solicit from us")
                debugString = ( "\n  Current Time: " + `time.time()` +
                "\n  Source MAC: " + src_mac +
                "\n  Target MAC: " + target +
                "\n  Source IPv6 Address: " + src_ip +
                "\n  Target IPv6 Address: " + target +

            else:
                qPrinter.printq("ND Solicit from us")
                debugString = ( "\n  Current Time: " + `time.time()` +
                "\n  Source MAC: " + src_mac +
                "\n  Target MAC: " + target +
                "\n  Source IPv6 Address: " + src_ip +
                "\n  Target IPv6 Address: " + target +

            else:
                qPrinter.printq("ND Solicit from us")
                debugString = ( "\n  Current Time: " + `time.time()` +
                "\n  Source MAC: " + src_mac +
                "\n  Target MAC: " + target +
                "\n  Source IPv6 Address: " + src_ip +
                "\n  Target IPv6 Address: " + target +

            else:
                qPrinter.printq("ND Solicit from us")
                debugString = ( "\n  Current Time: " + `time.time()` +
                "\n  Source MAC: " + src_mac +
                "\n  Target MAC: " + target +
                "\n  Source IPv6 Address: " + src_ip +
                "\n  Target IPv6 Address: " + target +

            else:
                qPrinter.printq("ND Solicit from us")
                debugString = ( "\n  Current Time: " + `time.time()` +
                "\n  Source MAC: " + src_mac +
                "\n  Target MAC: " + target +
                "\n  Source IPv6 Address: " + src_ip +
                "\n  Target IPv6 Address: " + target +

            else:
                qPrinter.printq("ND Solicit from us")
                debugString = ( "\n  Current Time: " + `time.time()` +
                "\n  Source MAC: " + src_mac +
                "\n  Target MAC: " + target +
                "\n  Source IPv6 Address: " + src_ip +
                "\n  Target IPv6 Address: " + target +

            else:
                qPrinter.printq("ND Solicit from us")
                debugString = ( "\n  Current Time: " + `time.time()` +
                "\n  Source MAC: " + src_mac +
                "\n  Target MAC: " + target +
                "\n  Source IPv6 Address: " + src_ip +
                "\n  Target IPv6 Address: " + target +

            else:
                qPrinter.printq("ND Solicit from us")
                debugString = ( "\n  Current Time: " + `time.time()` +
                "\n  Source MAC: " + src_mac +
                "\n  Target MAC: " + target +
                "\n  Source IPv6 Address: " + src_ip +
                "\n  Target IPv6 Address: " + target +
elif dst_mac in system_macs:  ## Unicast to us, only process src
    if (INCOMING_ICMPV6_TYPES_DEBUG or
        INCOMING_ICMPV6_FULL_DEBUG):
        qPrinter.printq("ND Solicit to us")
        if INCOMING_ICMPV6_FULL_DEBUG:
            debugString = ( " Timestamp: " + `ts` +
                " Current Time: " + `time.time()` +
                " Source MAC: " + src_mac +
                " Dest MAC: " + dst_mac +
                " IP Source: " + src_ip +
                " IP Dest: " + dst_ip +
                " Target: " + target)
            qPrinter.printq(debugString)
            processSource(src_ip, src_mac, ts)
        else:  ## Not from or to us (unless multicast), check target
            if (INCOMING_ICMPV6_TYPES_DEBUG or
                INCOMING_ICMPV6_FULL_DEBUG):
                qPrinter.printq("ND Solicit Other:")
                if INCOMING_ICMPV6_FULL_DEBUG:
                    debugString = ( " Timestamp: " + `ts` +
                        " Current Time: " + `time.time()` +
                        " Source MAC: " + src_mac +
                        " Dest MAC: " + dst_mac +
                        " IP Source: " + src_ip +
                        " IP Dest: " + dst_ip +
                        " Target: " + target)
                    qPrinter.printq(debugString)
                else:  ## Not from or to us (unless multicast), check target
                    if target in system_ips:  ## Multicast to us, process source
                        processSource(src_ip, src_mac, ts)
                    else:  ## Unicast or Multicast to other
                        ## Process Target
                        ## Check Recent Multicast Solicitations
                        lastSent = mc_recent_solicits.get(target)
                        if lastSent:  ## Target Exists in Recent Solicits list
                            if (lastSent < time.time() -
                                MC_SOLICIT_DUPLICATE_INTERVAL):
                                # Last sent > XX seconds ago, send new
                                results = ndSolicitMultiThread.generate(target)
                                mc_recent_solicits[results[0]] = results[1]
                                else:  ## Solicit sent < XX seconds ago, throttle
                                    if SOLICIT_THROTTLE_DEBUG:
                                        qPrinter.printq("Withholding MC Solicit to " + target)
                                    else:  ## Target IP is not in Recent Solicits list
                                        results = ndSolicitMultiThread.generate(target)
                                        mc_recent_solicits[results[0]] = results[1]
                                else:  ## Target IP is not in Recent Solicits list
                                    results = ndSolicitMultiThread.generate(target)
                                    mc_recent_solicits[results[0]] = results[1]
                            else:  ## Target exists in Recent Solicits list
                                results = ndSolicitMultiThread.generate(target)
                                mc_recent_solicits[results[0]] = results[1]
                            ## Process Dest IP if diff from Target and not Multicast
                            if not target == dst_ip:
                                if not dst_ip[0:2] == "FF":  ## Not to multicast
                                    ## Check Recent Multicast Solicitations
                                    lastSent = mc_recent_solicits.get(dst_ip)
                                    if lastSent:  # Dest is in Recent Solicits list
                                        if (lastSent < time.time() -
                                            MC_SOLICIT_DUPLICATE_INTERVAL):
                                            # Last sent > XX seconds ago, send new
                                            results = ndSolicitMultiThread.generate(target)
                                            mc_recent_solicits[results[0]] = results[1]
                                            else:  # Solicit sent < XX seconds ago, throttle
                                                if SOLICIT_THROTTLE_DEBUG:
                                                    qPrinter.printq("Withholding MC Solicit to " + target)
                                                else:
                                                    # Target IP is not in Recent Solicits list
                                                    results = ndSolicitMultiThread.generate(target)
                                                    mc_recent_solicits[results[0]] = results[1]
                        else:  ## Not from or to us (unless multicast)
                            if not dst_ip == dst_ip:
                                if not dst_ip[0:2] == "FF":  ## Not to multicast
                                    ## Check Recent Multicast Solicitations
                                    lastSent = mc_recent_solicits.get(dst_ip)
                                    if lastSent:  # Dest is in Recent Solicits list
                                        if (lastSent < time.time() -
                                            MC_SOLICIT_DUPLICATE_INTERVAL):
                                            # Last sent > XX seconds ago, send new
                                            results = ndSolicitMultiThread.generate(target)
                                            mc_recent_solicits[results[0]] = results[1]
                                            else:  # Solicit sent < XX seconds ago, throttle
                                                if SOLICIT_THROTTLE_DEBUG:
                                                    qPrinter.printq("Withholding MC Solicit to " + target)
                                                else:
                                                    # Target IP is not in Recent Solicits list
                                                    results = ndSolicitMultiThread.generate(target)
                                                    mc_recent_solicits[results[0]] = results[1]
MC_SOLICIT_DUPLICATE_INTERVAL):
    ## Last sent > XX seconds ago
    results = ndSolicitMultiThread.generate(dst_ip)
    mc_recent_solicits[results[0]] = {
        results[1]  ## Update list
    }
else:  ## Solicit sent < XX seconds ago
    if SOLICIT_THROTTLE_DEBUG:
        qPrinter.printq("Withholding MC " +
                     "Solicit to " + dst_ip)
    else:  ## Dest IP is not in Recent Solicits list
        results = ndSolicitMultiThread.generate(dst_ip)
        mc_recent_solicits[results[0]] = results[1]

    ## Process Source
    processSource(src_ip, src_mac, ts)

    ## NDP Advertisements (Reply)
elif type == "136":
    counts[1] += 1
    payload = string.upper(binascii.b2a_hex(eth.data.data.data))
    target = payload[16:48]  ## Target IPv6 Address
    if payload[52:64]:  ## Advertisement has Link Layer Address opt
        link_layer_addr = payload[52:64]  # Target Link-Layer Addr
        if (link_layer_addr != src_mac):
            qPrinter.printq("Warning: Adv Link Layer Address " +
                             "doesn't match source MAC")
            qPrinter.printq("IP: " + target + ", Source MAC: " +
                             src_mac + ", LLA: " + link_layer_addr)
    else:  ## Advertisement doesn't have Link Layer Address, use src
        link_layer_addr = src_mac

    if dst_mac in system_macs:  ## To us, process
        if (INCOMING_ICMPV6_TYPES_DEBUG or
            INCOMING_ICMPV6_FULL_DEBUG):
            qPrinter.printq("ND Adv to us")
        if INCOMING_ICMPV6_FULL_DEBUG:
            debugString = ("\n  Timestamp:    "+\`ts` +
                        "\n  Current Time: " + `time.time()` +
                        "\n  Source MAC: " + src_mac +
                        "\n  Dest MAC: " + dst_mac +
                        "\n  IP Source: " + src_ip +
                        "\n  IP Dest: " + dst_ip +
                        "\n  Target: " + target )
            qPrinter.printq(debugString)
        expected = replyWaiting.action("incomingAdv",
                                             (target,link_layer_addr))
        if expected:
            pendingProc.add(target, link_layer_addr, ts)
            if not target == src_ip:  ## Source and Target differ
                ## Process Source
                processSource(src_ip, src_mac, ts)
        else:  ## Not a result of our solicits, process src and tgt
            if UNEXPECTEDADV_DEBUG:
                qPrinter.printq("Unexpected Advertisement: "+
                                target + " from " + link_layer_addr)
Correlating IPv6 Addresses for Network Situational Awareness  

Jason Froehlich

```
301     + "\n\t received by " + dst_ip)
302
303     ## Process Target
304     processSource(target, link_layer_addr, ts)
305
306     ## Process Source if different from Target
307     if not target == src_ip:
308         processSource(src_ip, src_mac, ts)
309
310     elif src_mac in system_macs:  ## From us, ignore
311         if (INCOMING_ICMPV6_TYPES_DEBUG or
312             INCOMING_ICMPV6_FULL_DEBUG):
313             qPrinter.println("ND Advert from us")
314             if INCOMING_ICMPV6_FULL_DEBUG:
315                 debugString = ("  Timestamp: " + `ts` +
316                                "\n  Current Time: " + `time.time()` +
317                                "\n  Source MAC: " + src_mac +
318                                "\n  Dest MAC: " + dst_mac +
319                                "\n  IP Source: " + src_ip +
320                                "\n  IP Dest: " + dst_ip +
321                                "\n  Target: " + target )
322             qPrinter.println(debugString)
323
324     else:  ## Not to/from us, process source, target, and dest
325         if (INCOMING_ICMPV6_TYPES_DEBUG or
326             INCOMING_ICMPV6_FULL_DEBUG):
327             qPrinter.println("ND Advert Other")
328             if INCOMING_ICMPV6_FULL_DEBUG:
329                 debugString = ("  Timestamp: " + `ts` +
330                                "\n  Current Time: " + `time.time()` +
331                                "\n  Source MAC: " + src_mac +
332                                "\n  Dest MAC: " + dst_mac +
333                                "\n  IP Source: " + src_ip +
334                                "\n  IP Dest: " + dst_ip +
335                                "\n  Target: " + target )
336             qPrinter.println(debugString)
337             processSource(src_ip, link_layer_addr, ts)
338         if not target == src_ip:
339             processSource(target, link_layer_addr, ts)
340
341     ## Process Destination if not Multicast
342     if not dst_ip[0:2] == "FF":
343         lastSent = mc_recent_solicits.get(dst_ip)
344         if lastSent:  ## Target Exists in Recent Sol list
345             if (lastSent < time.time() -
346               MC_SOLICIT_DUPLICATE_INTERVAL):
347                 ## Last sent > XX seconds ago, send new
348                 results = ndSolicitMultiThread.generate(
349                     dst_ip)
350                 mc_recent_solicits[results[0]] = results[1]
351             else:  ## Solicit sent < XX seconds ago, throttle
352                 if SOLICIT_THROTTLE_DEBUG:
353                     qPrinter.println("Withholding MC " +
354                         "Solicit to " + target)
355                 else:  ## Target IP is not in Recent Solicits list
356                 results = ndSolicitMultiThread.generate(dst_ip)
357                 mc_recent_solicits[results[0]] = results[1]
358
359     ## Other ICMPv6 Type (Ping, Unreachable, etc.)
360     else:
361         if INCOMING_ICMPV6_TYPES_DEBUG or INCOMING_ICMPV6_FULL_DEBUG:
362             qPrinter.println("Other, Type: " + type)
```
Correlating IPv6 Addresses for Network Situational Awareness

Jason Froehlich

```python
if INCOMING_ICMPV6_FULL_DEBUG:
    debugString = ( "Timestamp: " + `ts` + 
                   "\n Current Time: " + `time.time()` + 
                   "\n Source MAC: " + src_mac + 
                   "\n Dest MAC: " + dst_mac + 
                   "\n IP Source: " + src_ip + 
                   "\n IP Dest: " + dst_ip )
    qPrinter.printq(debugString)

if __name__ == "__main__":
daemon = IPACAgentDaemon(AGENT_PID_FILE, time.time(),
                        stdout=AGENT_LOG_FILE, stderr=AGENT_LOG_FILE)
    if len(sys.argv) == 2:
        if 'start' == sys.argv[1]:
            daemon.start()
        elif 'stop' == sys.argv[1]:
            daemon.stop()
        elif 'status' == sys.argv[1]:
            daemon.status("IPAC Agent")
        elif 'restart' == sys.argv[1]:
            daemon.restart()
        else:
            print "Unknown command"
            sys.exit(2)
    else:
        print "usage: python %s start|status|stop|restart" % sys.argv[0]
        sys.exit(2)
```

11.4.1.2 *ipac_agent_config.py*

```
# /usr/bin/python
#
## IPAC Agent Configuration

## Jason Froehlich
## Rochester Institute of Technology
## Dept. of Network, Security, and System Administration
## May 17, 2011

## This program is a proof of concept being created for the partial
## fulfillment of the MS in Networking and Systems Administration from the
## Rochester Institute of Technology. All rights reserved.

###---------------------###
## Agent Information ##
###---------------------###

## Customize for each individual IPAC Agent process

## Agent Database's MySQL Name
## Format: String corresponding to database name
AGENT_DATABASE_DBNAME = "IPAC_Agent"

## Agent Database's IPv4 Address
## Format: String containing IPv4 address in "dotted quad" notation ("X.X.X.X")
AGENT_DATABASE_IP4 = "10.100.111.174"

## Agent Database's MySQL Password
## Format: String corresponding to AGENT_DATABASE_USER's password
AGENT_DATABASE_PASSWORD = "TigerCl@ws"
```
## Agent Database's MySQL Username
## User must have read, write and delete permissions to the 'active' and
## 'records' tables.
## Format: String corresponding to database's username
AGENT_DATABASE_USER = "ipac"

## Agent Identifier
## Must be unique within organization
## Format: Integer
AGENT_ID = 00001

## Log File Location
## All printed and error messages from IPAC Agent process will be written to
## the specified file. If file exists, text will be appended (user must have
## write permissions to file). If file doesn't exist, file will be created and
## text appended (user must have write permissions to directory).
## Format: String with full path of desired file location
AGENT_LOG_FILE = "/home/vortex/Desktop/ipacagentout.txt"

## PID File Location
## Holds PID and start time of the IPAC Agent process. These values are used
## for server daemon control (status/stop). User must have write permissions
## to the directory.
## Format: String with full path of desired file location
AGENT_PID_FILE = "/home/vortex/Desktop/ipacagent.pid"

## System Network Interface
## Used for packet listening, source interface for ND solicitations
## Format: String corresponding to system's newtork interface descriptor
SYSTEM_DEFAULT_IFACE = "eth1"

## System IPv6 Address
## Used as source IP for ND solicits
## Format: String of 32 Hexadecimal digits without colons
SYSTEM_DEFAULT_IP = "FE80000000000000426186FFFE123456"

## System MAC Address
## Used as source MAC for ND Solicitations
## Format: String of 12 Hexadecimal digits without colons
SYSTEM_DEFAULT_MAC = "406186123456"

##---------------------------##
##  Agent Program Constants  ##
##---------------------------##
## WARNING: Modification of these settings will affect IPAC Agent behavior.
## Do not change these values unless you are sure of their results.

## Clear database at IPAC Agent initialization
## Format: 'True'/'False'
FRESH_START = True

## Number of times a Refresh Solicitation will be sent before an IP is
## removed from the Active IPs list
## Format: Integer
ACTIVE_IP_REFRESH_COUNT = 2

## Interval between Active IP Refresher Thread Executions
## Format: Integer (seconds)
ACTIVE_IP_REFRESH_INTERVAL = 12
## Maximum amount of time ago since an IP in the Active IPs list was seen before a refresh Solicitation will be sent
## Format: Integer: (seconds)
ACTIVE_IP_REFRESH_TIMEOUT = 30

## Amount of time between MARK messages in IPAC Agent log
## Format: Integer (seconds)
LOG_MARK_INTERVAL = 900  ## 900 = 15 minutes

## Amount of time after a Multicast Solicitation has been sent before another will be sent to the same Target, Throttles Multicast solicitations to a target to 1 per XX seconds
## Format: Integer (seconds)
MC_SOLICIT_DUPLICATE_INTERVAL = 5

## Interval between Pending Processor Thread executions
## Format: Integer (seconds)
PENDING_PROCESS_INTERVAL = 5

## Number of times a solicitation is resent without a reply before the Target IP considered not active on the network and removed from Reply Waiting list
## Format: Integer
SOLICIT_RETRY_COUNT = 3

## Amount of time after a Solicitation has been sent in which an Advertisement should be received
## Format: Integer: (seconds)
SOLICIT_TIMEOUT_INTERVAL = 10

## Debugging Constants ##
## WARNING: These could produce a LOT of text in the log files.
## Set to 'False' unless you are debugging program execution.
## Format: 'True'/'False'

# Debug operations of ActiveIPRefresher thread
ACTIVE_REFRESHER_DEBUG = False

# Print capture interface information
CAPTURE_INTERFACE_DEBUG = False

# Display contents of database query results
DB_QUERY_RESULTS_DEBUG = False

# Print full description of incoming ICMPv6 packets
INCOMING_ICMPV6_FULL_DEBUG = False

# Print brief description of incoming ICMPv6 packets
INCOMING_ICMPV6_TYPES_DEBUG = False

# Identify IPAC-generated Solicitations in packet captures
## !-- WARNING --!
## PACKET_CAPTURE_DEBUG modifies the content of the ND Solicitation packets.
## When set, solicitation packets are sent with the Traffic Class of 255 rather than 0. Basic testing shows this does not affect how hosts respond, but this cannot be guaranteed for all host IPv6 stacks. Only set to TRUE ('True') if you need to differentiate between IPAC and regular system ND solicitations in packet captures when debugging.
## Set PACKET_CAPTURE_DEBUG to FALSE ('False') for normal use.
PACKET_CAPTURE_DEBUG = False
```python
#!/usr/bin/env python

## IPAC Daemon Class

## Jason Froehlich
## Rochester Institute of Technology
## Dept. of Network, Security, and System Administration
## May 17, 2011

## This program is a proof of concept being created for the partial
## fulfillment of the MS in Networking and Systems Administration from the
## Rochester Institute of Technology. All rights reserved.

## Based on the simple unix/linux daemon by Sander Marechal:
## http://www.jeik.com/articles/2007/02/a_simple_unix_linux_daemon_in_python/

## Module Imports
import atexit
import os
import sys
import time
from ipac_functions import prettyTime
from signal import SIGTERM

__author__ = "Jason Froehlich"
__copyright__ = "Copyright 2011, Jason Froehlich"
__version__ = "0.1.1"
__email__ = "jason.froehlich@alum.rit.edu"
__status__ = "Development"

class Daemon:
    ""
    A generic daemon class.
    ""
    Usage: subclass the Daemon class and override the run() method
    ""
    def __init__(self, pidfile, starttime, stdin='/dev/null',..."""
Correlating IPv6 Addresses for Network Situational Awareness

Jason Froehlich

```python
self.stdin = stdin
self.stdout = stdout
self.stderr = stderr
self.pidfile = pidfile
self.starttime = starttime

def daemonize(self):
    """do the UNIX double-fork magic, see Stevens' "Advanced
    Programming in the UNIX Environment" for details
    (ISBN 0201563177)
    http://www.erlenstar.demon.co.uk/unix/faq_2.html#SEC16
    ""
    try:
        pid = os.fork()
        if pid > 0:
            # exit first parent
            sys.exit(0)
        except OSError, e:
            sys.stderr.write("fork #1 failed: %d (%s)\n" % (e.errno,
                    e.strerror))
        sys.exit(1)
    # decouple from parent environment
    os.chdir("/")
    os.setsid()
    os.umask(0)
    # do second fork
    try:
        pid = os.fork()
        if pid > 0:
            # exit from second parent
            sys.exit(0)
        except OSError, e:
            sys.stderr.write("fork #2 failed: %d (%s)\n" % (e.errno,
                    e.strerror))
        sys.exit(1)
    # redirect standard file descriptors
    sys.stdout.flush()
    sys.stderr.flush()
    si = file(self.stdin, 'r')
    so = file(self.stdout, 'a+')
    se = file(self.stderr, 'a+', 0)
    os.dup2(si.fileno(), sys.stdin.fileno())
    os.dup2(so.fileno(), sys.stdout.fileno())
    os.dup2(se.fileno(), sys.stderr.fileno())
    # write pidfile
    atexit.register(self.delpid)
    pid = str(os.getpid())
    ## Write PID # and start time to PID file
    file(self.pidfile, 'w+').write("%s\n%f\n" % (pid, time.time()))

def delpid(self):
    os.remove(self.pidfile)

def start(self):
    """Start the daemon""
```

---

- 137 -
```python
# Check for a pidfile to see if the daemon already runs
try:
    pf = file(self.pidfile, 'r')
    pid = int(pf.readline().strip())
    pf.close()
except IOError:
    pid = None

if pid:
    message = """pidfile %s already exist. Daemon already \n    running?\n""
    sys.stderr.write(message % self.pidfile)
    sys.exit(1)

# Start the daemon
self.daemonize()
self.run()

def status(self, daemonName):
    ""
    Report status of daemon
    ""

    # Get the pid from the pidfile
    try:
        pf = file(self.pidfile, 'r')
        pid = int(pf.readline().strip())
        starttime = float(pf.readline().strip())
        pf.close()
    except IOError:
        pid = None

    if pid:
        # PID file contains PID # and start time
        message = (daemonName + " running for "+
                   prettyTime(time.time() - starttime) +
                   ", pid: "+ `pid`) 
    else:
        message = daemonName + " not running"
    print message

def stop(self):
    ""
    Stop the daemon
    ""

    # Get the pid from the pidfile
    try:
        pf = file(self.pidfile, 'r')
        pid = int(pf.readline().strip())
        pf.close()
    except IOError:
        pid = None

    if not pid:
        message = "pidfile %s does not exist. Daemon not \n        running?\n"" 
        sys.stderr.write(message % self.pidfile)
        return # not an error in a restart

    # Try killing the daemon process
    try:
```

    while 1:
        os.kill(pid, SIGTERM)
        time.sleep(0.1)
    except OSError, err:
        err = str(err)
        if err.find("No such process") > 0:
            if os.path.exists(self.pidfile):
                os.remove(self.pidfile)
            else:
                print str(err)
                sys.exit(1)
    def restart(self):
        ""
        Restart the daemon
        ""
        self.stop()
        self.start()
    def run(self):
        ""
        You should override this method when you subclass Daemon. It
        will be called after the process has been daemonized by start() or
        restart().
        ""

11.4.1.4 ipac_functions.py

    #! /usr/bin/python
    ## IPAC Shared Functions
    ## Jason Froehlich
    ## Rochester Institute of Technology
    ## Dept. of Network, Security, and System Administration
    ## May 17, 2011
    ## This program is a proof of concept being created for the partial
    ## fulfillment of the MS in Networking and Systems Administration from the
    ## Rochester Institute of Technology. All rights reserved.
    ## Module Imports
    import datetime
    import re  ## Regular Expressions
    import subprocess
    import time
    from ipac_agent_config import *
    __author__ = "Jason Froehlich"
    __copyright__ = "Copyright 2011, Jason Froehlich"
    __version__ = "0.1.2"
    __email__ = "jason.froehlich@alum.rit.edu"
    __status__ = "Development"
    def find_macs():
        ""
        Obtains MAC Addresses used by system on all interfaces
        Finds system's current MAC addresses on all network interfaces through the
        ifconfig command.
        ""
## Imports Required: re, subprocess

```python
# Obtain results of ifconfig, filter all except lines containing "HWaddr"
ifconfig1 = subprocess.Popen("ifconfig", stdout = subprocess.PIPE)
ifconfig2 = subprocess.Popen(["grep", "HWaddr"], stdin = ifconfig1.stdout,
                           stdout = subprocess.PIPE)
interfaces = ifconfig2.stdout.read()
```

## Regular Expressions to clean up ifconfig output

```python
macregex1 = re.compile('.*HWaddr')  # Match text before MAC
macregex2 = re.compile(\"\s+\")     # Match whitespace after MAC
macregex3 = re.compile('\:')        # Match colons within MAC
interfaces2 = macregex1.sub('', interfaces)  # Remove leading text
interfaces3 = macregex2.sub(\'', interfaces2)  # Remove trailing whitespace
interfaces4 = interfaces3.upper()     # Remove :, uppercase
```

## Return list of strings containing 12 chars of MAC address in upper case

```python
return interfaces4.split()
```

```python
def find_ips():
    ""
    Obtains IPv6 Addresses used by system on default interface
    Finds system's current IPv6 addresses on the network interface used by IPAC
    through the ifconfig command.
    """
    ## Imports Required: ipac_constants, re, subprocess
    ## Obtain results of ifconfig, filter all except lines with "inet6 addr"
    ifconfig1 = subprocess.Popen(["ifconfig", SYSTEM_DEFAULT_IFACE],
                                 stdout = subprocess.PIPE)
    ifconfig2 = subprocess.Popen(["grep", "inet6 addr"],
                                 stdin = ifconfig1.stdout,
                                 stdout = subprocess.PIPE)
    interfaces = ifconfig2.stdout.read()
    
    ## Regular Expressions to clean up ifconfig output
    fullIPs = []
    ipregex1 = re.compile('.*addr: ')  # Match text before IPv6 Addr
    ipregex2 = re.compile('/.*)')      # Match text after addr, including subnet mask
    interfaces2 = ipregex1.sub('', interfaces)  # Remove leading text
    interfaces3 = ipregex2.sub(\'', interfaces2)  # Remove trailing characters
    interfaces4 = interfaces3.split()     # Separate IPs

    ## Zero-fill shortened addresses
    for ip in interfaces4:
        fullIPs.append(expand_ip_address(ip))

    ## Return list of strings containing 32 chars of IP address in upper case
    return fullIPs
```

```python
def prettyMac(origMac):
    ""
    Formatted MAC Address String for Printing
    Create a properly formatted MAC address string, with bytes separated by
    colons. Takes in a 12 character string of Hex digits, produces a 17
    character string of uppercase Hex digits with colons.
    ""
    ol = list(origMac)
```
```python
def timeString(numtime):
    """
    Formatted Date-Time String
    Create a formatted date-time string. Takes in a float timestamp
    value, produces date-time string in format of "YYYY-MM-DD HH:MM:SS".
    """
    # Imports Required:  datetime
    return datetime.datetime.fromtimestamp(numtime).strftime("%Y-%m-%d %H:%M:%S")

def prettyIP(origIP):
    """
    Formatted IPv6 Address String for Printing
    Create a properly formatted IPv6 address string, with double bytes
    separated by colons. Takes in a 32 character string of Hex digits, produces
    a 39 character string of uppercase Hex digits with colons.
    """
    ol = list(origIP)
    p1 = "".join(ol[0:4])
    p2 = "".join(ol[4:8])
    p3 = "".join(ol[8:12])
    p4 = "".join(ol[12:16])
    p5 = "".join(ol[16:20])
    p6 = "".join(ol[20:24])
    p7 = "".join(ol[24:28])
    p8 = "".join(ol[28:32])
    return p1+":"+p2+":"+p3+":"+p4+":"+p5+":"+p6+":"+p7+":"+p8

def prettyTime(timestamp):
    """
    Modulated Date-Time String with English descriptors
    Returns a formatted string containing modulated number of days, hours,
    minutes, and seconds in given count of seconds. Seconds string is truncated
    to five digits (including ".") If days, hours or minutes field is zero, it
    will not be part of the final string.
    """
    seconds = timestamp % 60
    minutes = int(timestamp) // 60
    hours = minutes // 60
    days = hours // 24
    str = ""
    if days:
        if days == 1:
            str = str + 'day, '
        else:
            str = str + 'days` + " day, "
    else:
        str = str + `days` + " days, "
    if hours % 24:
```

- 141 -

Correlating IPv6 Addresses for Network Situational Awareness  Jason Froehlich
Correlating IPv6 Addresses for Network Situational Awareness

def expand_ip_address(compressed):
    
    Expands Shortened IPv6 Address to 32 Hexadecimal characters

    Expands shortened IPv6 Addresses with colons to 32 Hexadecimal characters.
    Removes all colon separators, fills in all missing leading zeros, fills in
    all "0000" double-bytes omitted by a "::". Accepts a string of hex and ":" chars, returns a string of 32 hex chars.
    NOTE: Assumes that input is a correctly formatted IPv6 address
    DOES NOT handle leading-zero addresses correctly (all addresses with
    leading ":" will be considered as "::1"
    
    ## Treat all leading "::" addresses as "::1"
    if compressed[0:2] == "::":
        return "00000000000000000000000000000001"

    complete = ""
    segments = compressed.split(":")
    missing = 9 - len(segments)
    for segment in segments:
        if len(segment) == 4:
            complete = complete + segment
        elif len(segment) == 3:
            complete = complete + "0" + segment
        elif len(segment) == 2:
            complete = complete + "00" + segment
        elif len(segment) == 1:
            complete = complete + "000" + segment
        elif len(segment) == 0:  # Results from "::", should be expanded
            complete = complete + ("0" * 4 * missing)
    return complete.upper()

def formatMacAddress(inMac):
    
    Formatted MAC Address String for Storage/Processing

    Accepts MAC Addresses with hexadecimal and ":" characters, produces a
    string of uppercase hex characters.
    
    ## Imports Required: re
    macregex = re.compile(\':\')
    return macregex.sub('',inMac.upper())

def formatIPAddress(inIP):
    
    Formatted IPv6 Address String for Storage/Processing

    Accepts IPv6 Addresses with hexadecimal and ":" characters, produces a
    string of uppercase hex characters. Calls expand_ip_address()s.
    
    return expand_ip_address(inIP)
def isValidDate(inTime):
    ""
    Verifies Date-Time String is a Valid Date
    Expected Date-Time String format: YYYY-MM-DD_HH:MM:SS
    Expects year to be between 2000 and 9999 (inclusive).
    Day value checks take into consideration months and leap years.
    Returns a Datetime tuple for valid dates, False for invalid dates
    ""
    ## Imports Required: datetime, time
    try:
        valid = True
        isLeapYear = False
        ## Split into components
        rawDate, rawTime = inTime.split("_")
        year,month,day = rawDate.split("-")
        hour,min,sec = rawTime.split(":")
        ## Convert components into integers
        year = int(year)
        month = int(month)
        day = int(day)
        hour = int(hour)
        min = int(min)
        sec = int(sec)
        ## Leap Year Test
        if (year % 400 == 0) or (year % 4 == 0 and year % 100 != 0):
            isLeapYear = True
        ## --Component Validation--
        ## Year must be 4 digits, later than/equal to 2000
        if not (year >= 2000 and year < 10000):
            valid = False
        ## Month must be 1-12
        if not (month > 0 and month < 13):
            valid = False
        ## Day must be more than 0
        if not (day > 0):
            valid = False
        ## Day must be less than or equal to 28, 29, 30 or 31
        if (month == 2): ## February, can have 28 or 29 days
            if isLeapYear: ## Leap Year, can have 29 days
                if not (day <= 29):
                    valid = False
            else: ## Can only have 28 days
                if not (day <= 28):
                    valid = False
        elif ([1,3,5,7,8,10,12].count(month)): ## Can have 31 days
            if not (day <= 31):
                valid = False
        else: ## Can only have 30 days
            if not (day <= 30):
                valid = False
        ## Hour must be 0-23
        if not (hour >= 0 and hour < 24):
            valid = False
        ## Minute must be 0-59
        return (datetime.datetime(1970, 1, 1, hour, min, sec), valid)
    except:
        return (datetime.datetime(1970, 1, 1, 0, 0, 0), False)
if not (min >= 0 and min < 60):
    valid = False
    ## Second must be 0-59
if not (sec >= 0 and sec < 60):
    valid = False
if valid:
    dt = datetime.datetime(year, month, day, hour, min, sec)
    return time.mktime(dt.timetuple())
else:
    return valid

except ValueError:
    ## Error when problem casting to integer, not valid date
    return False

def monthNameToNum(monthString):
    """
    Finds Numeral String for Month Name String
    Checks first three characters of provided Month Name string, returns two
    char Numerical value.
    """
    if monthString[0:3] == 'Jan':
        return "01"
    elif monthString[0:3] == 'Feb':
        return "02"
    elif monthString[0:3] == 'Mar':
        return "03"
    elif monthString[0:3] == 'Apr':
        return "04"
    elif monthString[0:3] == 'May':
        return "05"
    elif monthString[0:3] == 'Jun':
        return "06"
    elif monthString[0:3] == 'Jul':
        return "07"
    elif monthString[0:3] == 'Aug':
        return "08"
    elif monthString[0:3] == 'Sep':
        return "09"
    elif monthString[0:3] == 'Oct':
        return "10"
    elif monthString[0:3] == 'Nov':
        return "11"
    elif monthString[0:3] == 'Dec':
        return "12"
    else:
        return False

if __name__ == "__main__":
    pass

11.4.1.5 ipac_graph.py

#!/usr/bin/python
## IPAC Timeline Graph Generator
## Jason Froehlich
# Correlating IPv6 Addresses for Network Situational Awareness

**Jason Froehlich**

Rochester Institute of Technology  
Dept. of Network, Security, and System Administration  
May 17, 2011

This program is a proof of concept being created for the partial fulfillment of the MS in Networking and Systems Administration from the Rochester Institute of Technology. All rights reserved.

## Module Imports

```python
from matplotlib import dates, pyplot, patches
from matplotlib import pyplot
from matplotlib import patches
import MySQLdb
import sys
from ipac_functions import prettyIP
from ipac_functions import prettyMac
```

__author__ = "Jason Froehlich"  
__copyright__ = "Copyright 2011, Jason Froehlich"  
__version__ = "0.1.4"  
__email__ = "jason.froehlich@alum.rit.edu"  
__status__ = "Development"

### Server Database Information ###

```
DATABASE_HOST = "10.100.112.12"  
DATABASE_USER = "ipac"  
DATABASE_PASSWD = "T1gerCl@ws"  
DATABASE_DBNAME = "IPAC_Server"
```

```python
def onpick(event):
    ""
    Draw Starting/Ending Times For IP On Mouse Click
    ""
    Handles mouseclick event for BrokenBarHCollection objects. Redraws graph to include the exact starting and ending times of each broken bar segment for the selected container.
    ""
    thisItem = event.artist
    if labels.has_key(thisItem):  
        # Labels already displayed, remove
        # When text labels are added to the graph, they are added to the end of the axes's 'texts' array. To allow for removal, we must track the order in which text labels are added to the graph. This is done with the addedLabels array (stores IP address relating to each label). To remove labels for a specific container, find the positions in addedLabels for the container's related IP address and use those positions to remove the text objects from the axes's texts array.
        txtcount = labels[thisItem]  # Number of labels for this container
        n = 0  # Counts Labels Removed
        while n < txtcount:  # Repeat for all relevant labels
            loc = addedLabels.index(bars[thisItem])  # Find location in texts array
            del ax.texts[loc len(addedLabels)]  # Remove from graph through axes's texts array
            addedLabels.pop(loc)  # Remove from tracking array
            n += 1
        del labels[thisItem]  # Clear from dict tracking # of labels
    pyplot.draw()
```
else:    ## No labels displayed, add
    ip = bars[thisItem]
    times = ipData[ip][0]    ## Starting time and duration for each segment
    m = uniqueIPs.index(ip) + 1
    ## Tracks this BrokenBarH container's vertical location on graph
    txtcount = 0    ## Temporarily tracks number of labels added to graph
    for segment in times:    ## Add labels to all segments
        start = dates.num2date(segment[0])
        end = dates.num2date(segment[0] + segment[1])
        formStart = (start.strftime("%d %b %y %H:%M:%S.") +
                      start.strftime("%f")[0:3])
        formEnd = (end.strftime("%d %b %y %H:%M:%S.") +
                   end.strftime("%f")[0:3])
        ax.text(segment[0] + min((segment[1] / 40), 0.0005), m * 10 + 8.6, formStart,
                 horizontalalignment = 'left', verticalalignment = 'top', color = 'black')
        ## Add starting time label, proportional placement
        ax.text(segment[0] + segment[1] - min((segment[1] / 20), 0.001), m * 10 + 0.4, formEnd,
                 horizontalalignment = 'right', verticalalignment = 'bottom', color = 'black')
        ## Add ending time label, proportional placement
        addedLabels.append(ip)    ## Track starting time label
        addedLabels.append(ip)    ## Track ending time label
        txtcount += 2    ## 2 more labels created: starting and ending times
        labels[thisItem] = txtcount
    pyplot.draw()

## Main Program Execution
if __name__ == "__main__":

    ## Connect to Server database, get binding records
    serverDbConnection=MySQLdb.connect(DATABASE_HOST, DATABASE_USER,
                                         DATABASE_PASSWD, DATABASE_DBNAME)
    selectString = """SELECT * FROM bindings ORDER BY ipv6Address,startTime;""
    selectCursor = serverDbConnection.cursor()
    selectCursor.execute(selectString)
    results = selectCursor.fetchall()
    selectCursor.close()
    serverDbConnection.commit()

    ## Containers for graph data
    uniqueIPs = []    ## List of unique IPv6 Addresses in binding records
    haveConflicts = []    ## List of IPs with MAC Conflicts (Value: Vertical Pos.)
    ipData = {}    ## ipAddress :
                  ##       ([startTime, endTime], [macAddress], [endType])
    addedLabels = []    ## Tracks order in which text labels are added to graph
    ytickLoc = []    ## Stores vertical placements of Y axis tickmarks
    ytickLabels = []    ## Stores text labels (IPv6 addresses) of Y axis tickmarks
    bars = {}    ## Tracks IP address associated with each BrokenBarH container
                 labels = {}    ## Tracks number of labels for each BrokenBarH container (IP addr)

    ## If no binding records, nothing to show, exit
    if not results:
        print "No binding records, exiting..."
        sys.exit()
    ## Initialize min and max times to first binding record
    ## Will be used to track overall graph X axis scope
minTime = results[0][3]
maxTime = results[0][4]

## Find min and max times, number of unique IPs
for i in range(len(results)):
    if results[i][3] < minTime:  ## This entry is new minTime
        minTime = results[i][3]
    if results[i][4] > maxTime:  ## This entry is new maxTime
        maxTime = results[i][4]
    if uniqueIPs.count(results[i][0]):
        ## IP address already has binding records stored, append data
        ipData[results[i][0]][0].append((dates.epoch2num(results[i][3]),
                                           dates.epoch2num(results[i][4]) -
                                           dates.epoch2num(results[i][3])))
        ## Starting time and duration (ending time - starting time)
        ipData[results[i][0]][1].append(results[i][1])  ## MAC address
        ipData[results[i][0]][2].append(results[i][5])
        ## Ending type (20 = end observed;
        ## 10 = no more records, could still be active)
    else:  ## New IP Address
        uniqueIPs.append(results[i][0])  ## Add to list of unique IP addrs
        ipData[results[i][0]] = (([dates.epoch2num(results[i][3]),
                                   dates.epoch2num(results[i][4]) -
                                   dates.epoch2num(results[i][3]]),
          [results[i][1]], [results[i][5]])
        ## List of tuples containing starting time and duration
        ## (ending time - starting time), list of MAC address,
        ## list of Ending type (20 = end observed;
        ## 10 = no more records, could still be active)

## Get IP addresses and add into Y axis labels container
for ip in uniqueIPs:
    iptext = prettyIP(ip)
    ytickLabels.append(iptext[0:20] + " \n" + iptext[20:39])

## Determine vertical locations of Y axis tickmarks and labels
for i in range(len(uniqueIPs)):
    y = i * 10 + 14.5
    ytickLoc.append(y)

## Create graph object
fig = pyplot.figure()
fig.subplots_adjust(left = .2)
## Leave extra room on left of subplot for Y axis labels (IP addresses)
fig.subplots_adjust(right = .95)  ## Reduce extra room on right of subplot
ax = fig.add_subplot(111)

## Iterate through unique IP addrs, create BrokenBarH container and segments
j = 1  ## Tracks vertical location of current item
for ip in uniqueIPs:
    times = ipData[ip][0]  ## list of tuples containing starting time,
    ## duration (ending time - starting time) for each segment;
    ## used to specify X axis locations and lengths of segments
    macs = ipData[ip][1]  ## list of MAC addresses for each segment
    endTypes = ipData[ip][2]  ## list of Ending types for each segment
    ## (20 = end observed; 10 = no more records, could still be active)
    bh = ax.broken_barh(times, (j * 10, 9), facecolors=['#9bbdfd'],
                          picker = 5)  ## Create BrokenBarH container (X-axis
    ## loc. and len of segments specified by 'times')
    bh.set_alpha(0.5)
    bars[bh] = ip  ## Track IP addr associated with this BrokenBarH cont
# Find Conflicts and Indicate

conflicts = []  # Holds Start/End Times of Periods with MAC Conflicts
if len(times) > 1:  # >1 BrokenBarH Segments
    for i in range(len(times)):
        if i < len(times) - 1:  # Not last item
            l = i + 1
            while l < len(times):  # Iterate remaining segments
                if times[i][0] + times[i][1] > times[l][0]:  # End time is later than start of next, conflict
                    start = times[l][0]
                    if times[i][0] + times[i][1] > times[l][0] + times[l][1]:  # End of conflict is end of next
                        end = times[l][0] + times[l][1]
                    else:  # End of conflict is end of current
                        end = times[i][0] + times[i][1]
                    conflicts.append((start, end))
                l += 1

# Draw Conflict Indicators Below Segments
for set in conflicts:
    verts = ([set[0], j * 10 - .5], [segment[0], j * 10], [segment[1], j * 10 - .5])
    poly = patches.Polygon(verts, color='#d00000')
    ax.add_patch(poly)

if len(conflicts):
    haveConflicts.append(j)

# Add MAC address labels and Ending type indicators for each segment
k = 0  # Tracks current segment number, used to obtain MAC and EndType data from containers
for segment in times:
    # Create MAC address label for segment
    ax.text(segment[0] + segment[1] / 2, j * 10 + 5, prettyMac(macs[k]),
            horizontalalignment = 'center', verticalalignment = 'top',
            color = 'blue', weight = 'bold')

    # Create End type indicator at left end of segment
    if endTypes[k] == 20:  # End of IP address use observed, use Red
        xcolor = '#d00000'
    else:  # End not observed, could still be in use, use Green
        xcolor = '#009000'
    verts = ([segment[0] + segment[1] - min(segment[1] / 20, .0007),
              j * 10 + 3],
              j * 10 + 6],
              j * 10 + 7.5],
             [segment[0] + segment[1], j * 10 + 8],
             [segment[0] + segment[1], j * 10 + 1],
              j * 10 + 1.5])  # Rounded edge proportional
    poly = patches.Polygon(verts, color=xcolor)  # Create polygon
    ax.add_patch(poly)  # Add End type indicator polygon to graph
    k += 1  # Next segment
    j += 1  # Next BrokenBarH container

# Convert Final Min and Max Times
minT = dates.epoch2num(minTime)
maxT = dates.epoch2num(maxTime)

# Mark IPs that have MAC Address Conflicts
Correlating IPv6 Addresses for Network Situational Awareness  
Jason Froehlich

for ip in haveConflicts:
    verts = ([maxT+.001, ip * 10 + 4.5], [maxT + .003, ip * 10 + 3.5],
        [maxT + .005, ip * 10 + 4.5], [maxT+.003, ip * 10 + 5.5])
    verts2 = ([minT-.001, ip * 10 + 4.5], [minT - .003, ip * 10 + 3.5],
        [minT - .005, ip * 10 + 4.5], [minT-.003, ip * 10 + 5.5])
    poly = patches.Polygon(verts, color='orange')
    poly2 = patches.Polygon(verts2, color='orange')
    ax.add_patch(poly)
    ax.add_patch(poly2)

    ## Prepare graph structure
    ax.xaxis_date()  ## Use dynamic date formatting on X axis
    ax.set_ylim(5, (len(uniqueIPs) + 1) * 10 + 5)
    ## Set initial Y axis scope to reflect # of IP addresses
    if dates.epoch2num(minTime) - dates.epoch2num(maxTime) < 1:
        ## Records account for < 1 day, set initial X axis scope accordingly
        ax.set_xlim(dates.epoch2num(minTime) - .007,
                 dates.epoch2num(maxTime) + .007)
    else:
        ## Records account for > 1 day, set initial X axis scope to 1 day
        ax.set_xlim(dates.epoch2num(minTime) - .007,
                 dates.epoch2num(maxTime) + 1)
    ax.set_yticks(ytickLoc)  ## Create Y axis ticks
    ax.set_yticklabels(ytickLabels)  ## Label Y axis ticks
    ax.grid(True)  ## Show background grid

    ## Click event handler
    fig.canvas.mpl_connect('pick_event', onpick)
    ## Create and draw graph
    pyplot.show()
Correlating IPv6 Addresses for Network Situational Awareness

```python
DB_USER = "ipac"
DB_PASSWD = "T1gerCl@ws"
DB_DATABASE = "IPAC_Server"

def queryDatabase(queryString):
    
    databaseConnection=MySQLdb.connect(host=DB_HOST, user=DB_USER,
                                         passwd=DB_PASSWD, db=DB_DATABASE)
    dbCursor = databaseConnection.cursor()
    dbCursor.execute(queryString)
    results = dbCursor.fetchall()
    dbCursor.close()
    databaseConnection.commit()
    return results;

def addrType(ip6_addr):
    
    if ip6_addr[0:4] == "0000":
        type = "LB"
    elif ip6_addr[0:2] == "FF":
        type = "MC"
    elif ip6_addr[0:4] == "FE80":
        type = "LL"
    else:
        type = "--"
    return type

def processIP(ipAddr):
    returnVal = 
    type = addrType(ipAddr)
    macStrings = []

    if type == "LB" or type == "MC":  ## Address is Loopback or Multicast
        pass
    elif type == "LL":  ## Address is Link-Local
        queryString = "SELECT macAddress FROM bindings WHERE agentID=%i and ipv6Address='%s';" % (agent, ipAddr)
        results = queryDatabase(queryString)
        for mac in results:
            portQueryString = "SELECT switchID, switchPort, vlan FROM portBindings WHERE macAddress='%s';" % (mac)
            portResults = queryDatabase(portQueryString)
            for port in portResults:
                portStrings.append(port[0] + "/" + port[1] + "/" + port[2])
            if not portStrings:
                macStrings.append("Port Unknown")
                macStrings.append(prettyMac(mac[0]) + " (", ".join(portStrings) + ")")
    else:  ## Address is other non-local, translate if possible
        targetAgent = 0
        for subnet in subnets:
            if ipAddr[0:16] == subnet[0][0:16]:
                targetAgent = subnet[2]
```

- 150 -
89     if targetAgent:   ## Address is in organization's network
90         queryString = """SELECT macAddress FROM bindings WHERE agentID=%i
91             and ipv6Address='%s';""" % (targetAgent, ipAddr)
92         results = queryDatabase(queryString)
93         for mac in results:
94             portQueryString = """SELECT switchID, switchPort, vlan FROM
95                 portBindings WHERE macAddress='%s';""" % (mac)
96             portResults = queryDatabase(portQueryString)
97             portStrings = []
98             for port in portResults:
99                 portStrings.append(port[0] + "/" + port[1] + "/" + port[2])
100                if not portStrings:
101                    portStrings.append("Port Unknown")
102                    macStrings.append(prettyMac(mac[0]) + ": " +
103                        ", ".join(portStrings) + ")")
104     return macStrings
105
106     if __name__ == "__main__":
107         if len(sys.argv) < 2:
108             print "usage: %s filename [Agent ID]" % sys.argv[0]
109             sys.exit(2)
110     ## Obtain Agent ID used for Local subnets - parameter or prompt
111     if len(sys.argv) > 2: ## Agent argument provided
112         try:
113             agent = int(sys.argv[2])
114         except ValueError:  ## Error in conversion to int
115             try:
116                 agent = int(raw_input("Local Agent/VLAN: "))
117         except ValueError:
118             print "Valid Agent ID not given: Must be an Integer"
119             sys.exit()
120     else:  ## Agent argument not provided
121         try:
122             agent = int(raw_input("Local Agent/VLAN: "))
123         except ValueError:  ## Error in conversion to int
124             print "Valid Agent ID not given: Must be an Integer"
125             sys.exit()
126     ## Verify Agent ID exists in Server database
127     if agent > 0:  ## Must be a non-zero positive integer
128         agentSelect = "SELECT agentID FROM agents WHERE agentID=%i" % agent
129         results = queryDatabase(agentSelect)
130     if not results:  ## Agent not found
131         print "Valid Agent ID not given: Agent not known by Server"
132         sys.exit()
133     else:  ## Not a non-zero positive integer
134         print "Valid Agent ID not given: Must be a non-zero positive Integer"
135         sys.exit()
136
137     ## Open Input and Output files
138     pcapfile = sys.argv[1]
139     outfile = pcapfile + ".out"
140     try:
141         pf = open(pcapfile, 'r')
142         pcap = dpkt.pcap.Reader(pf)
143     except IOError:
144         print "Unable to open the pcap file"
145         sys.exit(2)
146     try:
147         of = open(outfile, 'w')
## Obtain Subnet info from Server
## Used to know which agent to query for IP address binding
subnetQuery = "SELECT * FROM subnets;"
subnets = queryDatabase(subnetQuery)

## Containers for Processing Results
pktCount = 0  ## Counts packets in pcap file
unmatched = {}  ## Holds IPv6 Addresses with no MAC at given time, counts
notLocal = {}  ## Holds IPv6 Addresses not in local subnets, counts

for timestamp, pkt in pcap:
    pktCount += 1
    eth = dpkt.ethernet.Ethernet(pkt)
    if eth.type == 34525:  ## Ethernet data is IPv6
        ip6 = eth.data
        src_ip = string.upper(binascii.b2a_hex(ip6.src))
        dst_ip = string.upper(binascii.b2a_hex(ip6.dst))
        of.write("Packet " + `pktCount` + "\n")
        newSrc = processIP(src_ip)
        if newSrc:
            of.write("    " + prettyIP(src_ip) + "\n")
            for pair in newSrc:
                of.write("        " + pair + "\n")
        else:
            if (addrType(dst_ip) != "LB" and addrType(dst_ip) != "MC"):
                of.write("    " + prettyIP(src_ip) + " (Unknown)" + "\n")
            newDst = processIP(dst_ip)
            if newDst:
                of.write("    " + prettyIP(dst_ip) + "\n")
                for pair in newDst:
                    of.write("        " + pair + "\n")
            else:
                if (addrType(dst_ip) != "LB" and addrType(dst_ip) != "MC"):
                    of.write("    " + prettyIP(dst_ip) + " (Unknown)" + "\n")

## Clean Up
pf.close()
of.close()
import sys
from ipac_functions import *
from ipac_functions import is_ValidDate

__author__ = "Jason Froehlich"
__copyright__ = "Copyright 2011, Jason Froehlich"
__version__ = "0.1.3"
__email__ = "jason.froehlich@alum.rit.edu"
__status__ = "Development"

DATABASE_HOST = "10.100.112.12"  # Database's IPv4 Address
DATABASE_USER = "ipac"           # Database's MySQL Username
DATABASE_PASSWD = "T1gerCl@ws"   # Database's MySQL Password
DATABASE_DBNAME = "IPAC_Server"  # Database's MySQL Name

def printUse():
    ""
    Prints Program Usage Guide and Exits
    ""

    print """
    Usage: python ipac_query.py --agent [Agent ID] {Operations} {Options}
    Operations:
    --findips : Find all IPs in use, contrained by --mac
    --findmacs : Find all MACs in use, contrained by --ip
    --findport : Find ports used by given IP or MAC
    --now [ip|mac| ]: Find all IPs and/or MACs currently in use
    --conflicts : Find times when multiple MACs used the same IPv6 address
    Options:
    --ip [IPv6 Address] : Find records with this IPv6 Address
    --mac [MAC Address] : Find records with this MAC address
    --time [Time] : Exact time to find records for
    --start [Start Time] : Find records existing after this time
    --end [End Time] : Find records existing before this time
    -b : Brief output
    -v : Verbose output (includes timestamps when relevant)
    Time Format: 'Year-Month-Day_Hour-Min-Sec'     Ex: '2011-02-14_16:32:45'
    ""
sys.exit()

def processRecords(records, op):
    data = {}
    for record in records:
        if op == "ip":
            key1 = record[0]
            key2 = record[1]
        elif op == "mac":
            key2 = record[0]
            key1 = record[1]
        recstart = record[3]
        recend = record[4]
        if time:  # Exact time specified
            if float(recstart) < time + 1 and float(recend) >= time:
                if data.has_key(key1):
                    if data[key1].has_key(key2):
                        data[key1][key2].append((recstart, recend))
                    else:
                        data[key1][key2] = [(recstart, recend)]
                else:
                    data[key1][key2] = [(recstart, recend)]
            else:
Correlating IPv6 Addresses for Network Situational Awareness

Jason Froehlich

```python
78     data[key1]={key2 : [(recstart,recend)]}
79     elif startTime and endTime: ## Start and End times specified
80         if float(recstart) < endTime + 1 and float(recend) >= startTime:
81             if data.has_key(key1):
82                 if data[key1].has_key(key2):
83                     data[key1][key2].append((recstart,recend))
84                 else:
85                     data[key1][key2] = [(recstart,recend)]
86             else:
87                 data[key1]={key2 : [(recstart,recend)]}
88     elif startTime: ## Start time specified
89         if float(recend) >= startTime:
90             if data.has_key(key1):
91                 if data[key1].has_key(key2):
92                     data[key1][key2].append((recstart,recend))
93                 else:
94                     data[key1][key2] = [(recstart,recend)]
95             else:
96                 data[key1]={key2 : [(recstart,recend)]}
97     elif endTime: ## End time specified
98         if float(recstart) < endTime + 1:
99             if data.has_key(key1):
100                if data[key1].has_key(key2):
101                    data[key1][key2].append((recstart,recend))
102                else:
103                    data[key1][key2] = [(recstart,recend)]
104            else:
105                data[key1]={key2 : [(recstart,recend)]}
106     else: ## No time constraints specified
107         if data.has_key(key1):
108             if data[key1].has_key(key2):
109                 data[key1][key2].append((recstart,recend))
110            else:
111                data[key1][key2] = [(recstart,recend)]
112        else:
113            data[key1]={key2 : [(recstart,recend)]}
114
115    return data

116 def queryDatabase(queryString):
117     """
118     Execute given query string on database.
119     """
120     databaseConnection = MySQLdb.connect(host=DATABASE_HOST, user=DATABASE_USER,
121                                          passwd=DATABASE_PASSWD, db=DATABASE_DBNAME)
122     dbCursor = databaseConnection.cursor()
123     dbCursor.execute(queryString)
124     results = dbCursor.fetchall()
125     dbCursor.close()
126     databaseConnection.commit()
127     return results;
128
129 if __name__ == "__main__":
130     ## Hold Exact, Start and End time constraints if given in arguments
131     time = 0
132     startTime = 0
133     endTime = 0
134
135     ## Check that more than one argument is provided
136     ## Program name is always 1st argument
137     if len(sys.argv) < 2: ## No arguments given
138     ```
## Check that Agent ID is provided

```python
if sys.argv.count('--agent'):
    loc = sys.argv.index('--agent')
    try:
        agent = int(sys.argv[loc + 1])
    except ValueError:
        # Error when casting arg to integer
        print("Error: Given Agent ID Not Numeric")
        printUse()
    except IndexError:
        # Error when no arg after '--agent'
        print("Error: Agent ID Not Specified")
        printUse()
else:
    print("Error: Agent ID Not Specified")
    printUse()
```

## Check that an operation argument is provided

```python
if not (sys.argv.count('--findips') or sys.argv.count('--findmacs') or
        sys.argv.count('--now') or sys.argv.count('--conflicts') or
        sys.argv.count('--findport')):
    print("Error: Operation not specified")
    printUse()
```

## Check that only one operation argument is provided

```python
if (sys.argv.count('--findips') + sys.argv.count('--findmacs') +
    sys.argv.count('--now') + sys.argv.count('--conflicts') +
    sys.argv.count('--findport')) > 1:
    print("Error: Only one operation can be specified")
    printUse()
```

## Verify that both Start/End and Exact times are not provided

```python
if sys.argv.count('--time') and (sys.argv.count('--start')
    or sys.argv.count('--end')):
    print("Error: Specify either Start/End or Exact times, but not both")
    printUse()
```

## Verify that both Brief and Verbose outputs are not specified

```python
if sys.argv.count('-b') and sys.argv.count('-v'):
    print("Error: Specify either Brief or Verbose output, but not both")
    printUse()
```

## Obtain time parameters if provided

```python
if sys.argv.count('--time'):
    loc = sys.argv.index('--time')
    try:
        inTime = sys.argv[loc + 1]
    except IndexError:
        # Error when no arg after '--time'
        print("Error: Time not specified")
        printUse()
    time = isValidDate(inTime)
    if not time:
        print("Error: Proper Time not given")
        printUse()
    else:
        pass
```

```python
if sys.argv.count('--start'):
    loc = sys.argv.index('--start')
    try:
        inStartTime = sys.argv[loc + 1]
    except IndexError:
        # Error when no arg after '--start'
        print("Error: Start Time not specified")
        printUse()
    startTime = isValidDate(inStartTime)
    if not startTime:
```
print "Error: Proper Start Time not given"
printUse()
else:
pass

if sys.argv.count("--end"):
    loc = sys.argv.index("--end")
    try: inEndTime = sys.argv[loc + 1]
    except IndexError: ## Error when no arg after '--end'
        print "Error: End Time not specified"
        printUse()
    endTime = isValidDate(inEndTime)
    if not endTime:
        print "Error: Proper End Time not given"
        printUse()
    else:
pass

## Verify that Start time is earlier than or equal to End Time if both given
if startTime and endTime and startTime > endTime:
    print "Error: End Time cannot be earlier than Start Time"
    printUse()

## Operation: Find IPs
if sys.argv.count("--findips"):
    targetMac = ""
    dbquerystring = "SELECT * FROM bindings WHERE agentID="" + `agent`

## If MAC constraint is specified
if sys.argv.count("--mac"):
    loc = sys.argv.index("--mac")
    try: rawmac = sys.argv[loc+1]
    except IndexError: ## Error when no arg after '--mac'
        print "Error: MAC Address not specified"
        printUse()
    if rawmac.count(":"):
        ## User input MAC is separated by colons
        fullmac = formatMacAddress(rawmac) ## Remove colons
    else:
        ## No colons
        fullmac = rawmac
    if len(fullmac) != 12:
        print "Error: Incorrectly formatted MAC Address Specified"
        sys.exit()
    targetMac = fullmac

## Execute database query
records = queryDatabase(dbquerystring + ";")
#Data Container: { IP : { MAC : (start,end)}
data = processRecords(records, "ip")
keys = data.keys()
keys.sort()

if targetMac: ## MAC Constraint specified
    print "IP Addresses used by MAC Address " + prettyMac(targetMac)
    for ip in keys:
        tscount = 0
        print " " + prettyIP(ip)
        if sys.argv.count("-v"): ## Verbose Output
            for mac in data[ip]:
                for times in data[ip][mac]:
                    if tscount:
                        print ("\n" + "\n" + timeString(times[0]) +
                            " -- " + timeString(times[1]),)
else:
    print(" "*8 + "( " + timeString(times[0]) + " -- " + timeString(times[1]),)
    tcount += 1
    print ")")
else:  ## MAC Constraint not specified
    print "Active IP Addresses"
    for ip in keys:
        print "  " + prettyIP(ip)
        if not sys.argv.count("-b"):
            ## Not Brief Output
            keys2 = data[ip].keys()
            keys2.sort()
            for mac in keys2:
                print "     " + prettyMac(mac),
                if sys.argv.count("-v"):
                    ## Verbose Output
                    print "     (",
                    for times in data[ip][mac]:
                        if (times == data[ip][mac][0] and
                            times == data[ip][mac][-1]):
                            print (timeString(times[0]) + " -- " +
                                timeString(times[1]) + " )")
                        elif times == data[ip][mac][0]:
                            print (timeString(times[0]) + " -- " +
                                timeString(times[1]) + ",")
                        elif times == data[ip][mac][-1]:
                            print (" "*30 + timeString(times[0]) +
                                " -- " + timeString(times[1]) + ",")
                        else:
                            print (" "*30 + timeString(times[0]) +
                                " -- " + timeString(times[1]) + ",")
                    else:
                        print ""
        else:
            print ""
## Operation:  Find MACs
elif sys.argv.count("--findmacs"):
    targetIP = ""
    dbquerystring = "SELECT * FROM bindings WHERE agentID=" + `agent` + ""
## If IP Constraint is specified
    if sys.argv.count("--ip"):
        loc = sys.argv.index("--ip")
        try:
            rawip = sys.argv[loc+1]
        except IndexError:
            ## Error when no arg after '--ip'
            print "Error: IP Address not specified"
            printUse()
        if rawip.count(":"):
            ## User input IP is separated by colons
            fullip = formatIPAddress(rawip)  ## Remove colons and expand
        else:
            ## No colons
            fullip = rawip
        if len(fullip) != 32:
            print "Error: Incorrectly formatted IP Address Specified"
            sys.exit()
        dbquerystring = dbquerystring + " and ipv6Address=" + fullip + "]"
    targetIP = fullip
## Execute Database Query
records = queryDatabase(dbquerystring + ");")
##Data Container: { IP : { MAC : (start,end)}
data = processRecords(records, "mac")
keys = data.keys()
keys.sort()
if targetIP:  ## IP Constraint Specified
    print "MAC Addresses owning IPv6 Address " + prettyIP(targetIP)
    for mac in keys:
        tcount = 0
        print " " + prettyMac(mac)
        if sys.argv.count("-v"):
            ## Verbose Output
            for ip in data[mac]:
                for times in data[mac][ip]:
                    if tcount:
                        print ("\n" + "*10 + timeString(times[0]) + " -- " + timeString(times[1]),)
                    else:
                        print (" "*8 + "( " + timeString(times[0]) + " -- " + timeString(times[1]),)
                    tcount += 1
                    print ")"
    else:  ## IP Constraint not specified
        print "Active MAC Addresses"
        for mac in keys:
            print " " + prettyMac(mac)
            if not sys.argv.count("-b"):
                keys2 = data[mac].keys()
                keys2.sort()
                for ip in keys2:
                    tcount = 0
                    print " " + prettyIP(ip)
            if sys.argv.count("-v"):
                ## Verbose Output
                for times in data[mac][ip]:
                    if tcount:
                        print ("\n" + "*12 + timeString(times[0])
                    + " -- " + timeString(times[1]),)
                    else:
                        print (" "*10 + "( " + timeString(times[0])
                    + " -- " + timeString(times[1]),)
                    tcount += 1
                    print ")"
        ## Operation:  Find Ports
    elif sys.argv.count("--findport"):
        targetMac = ""
        targetIP = ""
        ## If both IP and MAC constraints are specified
        if sys.argv.count("--ip") and sys.argv.count("--mac"):
            print "Error: Specify either IP or MAC, not both"
            printUse()
        ## If IP Constraint is specified
        elif sys.argv.count("--ip"):
            loc = sys.argv.index("--ip")
            try: rawip = sys.argv[loc+1]
            except IndexError:  ## Error when no arg after '--ip'
                print "Error: IP Address not specified"
                printUse()
            if rawip.count(":"):
                fullip = formatIPAddress(rawip)  ## Remove colons and expand
            else:
                fullip = rawip
            if len(fullip) != 32:
                print "Error: Incorrectly formatted IP Address Specified"
                sys.exit()
            targetIP = fullip
        elif targetIP:
            print "MAC Addresses owning IPv6 Address " + prettyIP(targetIP)
            for mac in keys:
                tcount = 0
                print " " + prettyMac(mac)
                if sys.argv.count("-v"):
                    ## Verbose Output
                    for ip in data[mac]:
                        for times in data[mac][ip]:
                            if tcount:
                                print ("\n" + "*10 + timeString(times[0]) + " -- " + timeString(times[1]),)
                            else:
                                print (" "*8 + "( " + timeString(times[0]) + " -- " + timeString(times[1]),)
                            tcount += 1
                            print ")"
## If MAC Constraint is specified

eif sys.argv.count("--mac"):
    loc = sys.argv.index("--mac")
    try:
        rawmac = sys.argv[loc+1]
    except IndexError:  # Error when no arg after '--mac'
        print "Error: MAC Address not specified"
        printUse()
    if rawmac.count(":"):
        fullmac = formatMacAddress(rawmac)  # Remove colons
    else:
        fullmac = rawmac
    if len(fullmac) != 12:
        print "Error: Incorrectly formatted MAC Address Specified"
        sys.exit()
    targetMac = fullmac
else:  # Neither IP nor MAC specified
    print "Error: IP or MAC address must be specified"
    printUse()

macs = []

if targetMac:
    macs.append(targetMac)
eif targetIP:
    dbquerystring = ('SELECT * FROM bindings WHERE ipv6Address="" +
                     targetIP + '"')
    if time:
        dbquerystring += ('" and startTime < " + `time` +
                           " and endTime > " + `time + 1`)
    if startTime:
        dbquerystring += " and endTime > " + `startTime`
    if endTime:
        dbquerystring += " and startTime < " + `endTime + 1`
    dbquerystring += ' ORDER BY startTime;'
    ipRecords = queryDatabase(dbquerystring)
    for rec in ipRecords:
        if not macs.count(rec[1]):
            macs.append(rec[1])
    for mac in macs:
        dbquerystring = ('SELECT * FROM portBindings WHERE macAddress="" +
                          mac + '"')
        if time:
            dbquerystring += ('" and startTime < " + `time` +
                               " and endTime > " + `time + 1`)
        if startTime:
            dbquerystring += " and endTime > " + `startTime`
        if endTime:
            dbquerystring += " and startTime < " + `endTime + 1`
        dbquerystring += ' ORDER BY startTime;'
        macRecords = queryDatabase(dbquerystring)
        if macRecords:
            print " " + prettyMac(mac)
        else:
            print " " + prettyMac(mac) + " [None]"
        uniqueMacs = {}
        for port in macRecords:
            if uniqueMacs.has_key(portID):
                uniqueMacs[portID] += 1
            else:
                uniqueMacs[portID] = 1
else:


uniqueMacs[portID] = 1
for mac in uniqueMacs.keys():
    if sys.argv.count("-b"):  
        print "    " + mac
    else:
        print "    " + mac + " Sawyer\n" + `uniqueMacs[mac]\` + "\n"
else:
    for port in macRecords:
        if int(port[4]):  
            start = timeString(int(port[4]))
        else:
            start = "        "
        if int(port[5]):  
            end = timeString(int(port[5]))
        else:
            end = "        
            " (" + start + " to " + end + ")")

## Operation: Find Conflicts
if sys.argv.count("--conflicts"):  
    targetMac = ""
    dbquerystring = ("SELECT * FROM bindings WHERE agentID=" + `agent` +  
            " ORDER BY ipv6Address,startTime")
## Execute database query
    records = queryDatabase(dbquerystring + ",")
    results = records

##Data Container: { IP : { MAC : (start,end)}
    data = {}  
    uniqueIPs = [] ## List of unique IPv6 Addresses in binding records
    ipData = {}  
    ## { ipAddress : ([(startTime, endTime)], [macAddress], [endType]) }

    for i in range(len(results)):  
        ## Find Unique IP Addresses
        if uniqueIPs.count(results[i][0]):  
            ## IP address already has binding records stored, append data
            ipData[results[i][0]][0].append((results[i][3],results[i][4]))
            ## Starting time and duration (ending time - starting time)
            ipData[results[i][0]][1].append(results[i][1])  
            ## MAC address
        else:
            uniqueIPs.append(results[i][0])  
            ## Add to list of unique IPs
            ipData[results[i][0]] = ([(results[i][3],results[i][4]),
                                      [results[i][1]])  
            ## list of tuples containing starting and ending time sets and list of MACs
    allconflicts = {}
    for ip in uniqueIPs:
        times = ipData[ip][0]
        macs = ipData[ip][1]
        conflicts = []  
        ## [ (start, end, [macs]) ]

        if len(times) > 1:  
            if startTime and endTime:
                for i in range(len(times)):
                    if i < len(times)-1:  
                        ## Not last item
                        if (times[i][0] < endTime + 1 and  
                            times[i][1] > startTime):
                            l = i+1
                        while l < len(times):
                            if (times[l][0] < endTime + 1 and  
                                times[l][1] > startTime):
                                l = l+1
if times[i][1] > times[l][0]:
    # End time is later than beginning of next, conflict
    start = times[l][0]
    if times[i][1] > times[l][1]:
        # End of conflict is end of nxt
        end = times[l][1]
    else:
        # End of conflict is end of cur
        end = times[i][1]
    conflicts.append((start, end, [macs[i], macs[l]]))

l += 1

elif startTime:
    for i in range(len(times)):
        if i < len(times)-1:  # Not last item
            if times[i][1] > startTime:
                l = i+1
        while l < len(times):
            if times[i][1] > times[l][0]:
                # End time is later than beginning of next, conflict
                start = times[l][0]
                if times[i][1] > times[l][1]:
                    # End of conflict is end of nxt
                    end = times[l][1]
                else:
                    # End of conflict is end of cur
                    end = times[i][1]
                conflicts.append((start, end, [macs[i], macs[l]]))
            l += 1

elif endTime:
    for i in range(len(times)):
        if i < len(times)-1:  # Not last item
            if times[i][0] < endTime + 1:
                l = i+1
        while l < len(times):
            if times[l][0] < endTime + 1:
                if times[i][1] > times[l][0]:
                    # End time is later than beginning of next, conflict
                    start = times[l][0]
                    if times[i][1] > times[l][1]:
                        # End of conflict is end of nxt
                        end = times[l][1]
                    else:
                        # End of conflict is end of cur
                        end = times[i][1]
                    conflicts.append((start, end, [macs[i], macs[l]]))
            l += 1

elif time:  # Exact time specified
    for i in range(len(times)):
        if i < len(times)-1:  # Not last item
            if times[i][0] < time + 1 and times[i][1] > time:
                l = i+1
        while l < len(times):
            if (times[l][0] < time + 1 and
                times[l][1] > time):
                if times[i][1] > times[l][0]:
                    # End time is later than beginning of next, conflict
                    start = times[l][0]
                    if times[i][1] > times[l][1]:
                        # End of conflict is end of nxt
                        end = times[l][1]
                    else:
                        # End of conflict is end of cur
                        end = times[i][1]
                    conflicts.append((start, end, [macs[i], macs[l]]))
Correlating IPv6 Addresses for Network Situational Awareness

Jason Froehlich

```python
end = times[i][1]
conflicts.append((start, end, [macs[i], macs[l]]))

1 += 1
else:  ## No time constraints
    for i in range(len(times)):
        if i < len(times) - 1:  ## Not last item
            l = i + 1
            while l < len(times):
                if times[i][1] > times[l][0]:
                    ## End time is later than beginning of next, conflict
                    start = times[l][0]
                    if times[i][1] > times[l][1]:
                        ## End of conflict is end of next
                        end = times[l][1]
                    else:  ## End of conflict is end of current
                        end = times[i][1]
                    conflicts.append((start, end, [macs[i], macs[l]]))
                    l += 1

if conflicts:
    allconflicts[ip] = conflicts
if allconflicts:
    print "MAC Conflicts Found:"
    for ip in allconflicts.keys():
        print "  ", prettyIP(ip) + ":"
        for cf in allconflicts[ip]:
            s = ", " + prettyIP(cf[2])
            print("  " + s, "from", timeString(cf[0]), "to",
                  timeString(cf[1]))
else:
    print "No Conflicts Found"

## Operation: Find Currently Active
elif sys.argv.count("--now"):
    loc = sys.argv.index("--now")
    ## Execute Database Query
    dbquerystring = ("SELECT * FROM bindings WHERE agentID=" + `agent` +
                    " and endType=10;")
    records = queryDatabase(dbquerystring)

if loc < len(sys.argv) - 1:  ## Not last argument
    if sys.argv[loc + 1] == "ip":  ## Only output IPs
        ips = {}
        for record in records:
            if not ips.has_key(record[0]):
                ips[record[0]] = record[3]
            elif ips[record[0]] < record[3]:
                ips[record[0]] = record[3]
        print "Currently Active IPs:"
        keys = ips.keys()
        keys.sort()
        for ip in keys:
            if sys.argv.count("-b"):  ## Brief Output, Only IP
                print "  ", ip
            else:  ## IP, Start Time
                print("  " + ip + " (Active since " +
                      timeString(ips[ip]) + ")")
        elif sys.argv[loc + 1] == "mac":  ## Only output MACs
            macs = {}
            for record in records:
                if not macs.has_key(record[1]):
```
Correlating IPv6 Addresses for Network Situational Awareness

Jason Froehlich

```python
macs[record[1]] = (record[3], 1)
```  
```python
elif macs[record[1]][0] > record[3]:
    macs[record[1]] = (record[3], macs[record[1]][1]+1)
else:
    macs[record[1]] = (macs[record[1]][0],
                       macs[record[1]][1]+1)
```

```python
print "Currently Active MACs:
keys = macs.keys()
keys.sort()
for mac in keys:
    if sys.argv.count("-b"):
        print "   " + mac  
    else:
        print("   " + mac + "(" + `macs[mac][1]` + 
              " IPs, active since " +
              "timeString(macs[mac][0]) + ")")
else:
    print "Currently Active:
for record in records:
    if sys.argv.count("-b"):
        print "   " + record[0] + " : " + record[1]  
    else:
        print("   " + record[0] + " : " + record[1] +
              " (Since " + timeString(int(record[3])) + ")")
else:
    print "Currently Active:
for record in records:
    if sys.argv.count("-b"):
        print "   " + record[0] + " : " + record[1]  
    else:
        print("   " + record[0] + " : " + record[1] +
              " (Since " + timeString(int(record[3])) + ")")
```

11.4.1.8 ipac_server.py

```
#!/usr/bin/python

## IPAC Server Daemon

## Jason Froehlich
## Rochester Institute of Technology
## Dept. of Network, Security, and System Administration
## May 17, 2011

## This program is a proof of concept being created for the partial
## fulfillment of the MS in Networking and Systems Administration from the
## Rochester Institute of Technology.  All rights reserved.

## Daemon class based on the simple unix/linux daemon by Sander Marechal:
## http://www.jejik.com/articles/2007/02/a_simple_unix_linux_daemon_in_python/

## Module Imports
import MySQLdb
import sys
import time
from ipac_daemon import Daemon
from ipac_functions import prettyTime
from ipac_server_config import *
from ipac_trapsum import clearPortBindings
from ipac_trapsum import summarizeTraps
```
class IPACServerDaemon(Daemon):
    
    Daemon class for the IPAC Server process
    
    def run(self):
        
        Body of the IPAC Server process, called by start() or restart()
        
        def bindingRecord(currentIP, currentMAC, agentID, firstSeen, lastSeen, code):
            
            Generate or Update IP-MAC Binding Records in Server Database
            
            # Codes:
            # 10 - No Ending Record, Node May Still Be Active
            # 11 - No Ending Record But Another MAC Claims IP
            # 20 - Ending Record Seen
            
            if code != 20:
                selectQuery = """SELECT * FROM bindings WHERE ipv6Address = \\
        '%%%s' and macAddress = '%%%s' and agentID = %d and endType=10 and \\
        endTime > %f ORDER BY endTime;""" % (currentIP, currentMAC, \\
                agentID, firstSeen-(ACTIVE_IP_REFRESH_COUNT+1))
                dbCursor1 = serverDbConnection.cursor()
                dbCursor1.execute(selectQuery)
                hasRecord = dbCursor1.fetchall()
                dbCursor1.close()
                
                if hasRecord:
                    if DEBUG_SERVER_RECORDS:
                        print currentIP, currentMAC, "has binding, updating"
                        sys.stdout.flush()
                    if code != 20:
                        updateQuery = """UPDATE bindings SET endTime=%f, endType=%d\\
            WHERE ipv6Address='%%%s' and macAddress='%%%s' and agentID=%d \\
            and startTime=%f;""" % (lastSeen, code, currentIP, \\
                                currentMAC, agentID, hasRecord[-1][3])
                    else:
                        updateQuery = """UPDATE bindings SET endType=%d WHERE \\
            ipv6Address='%%%s' and macAddress='%%%s' and agentID=%d and \\
            startTime=%f;""" % (code, currentIP, currentMAC, agentID, \\
                                hasRecord[-1][3])
                dbCursor3 = serverDbConnection.cursor()
                try:
                    dbCursor3.execute(updateQuery)
                except MySQLdb.IntegrityError:
print "Error executing database query, key doesn't exist
"
sys.stdout.flush()
dbCursor3.close()
serverDbConnection.commit()
else:
    if DEBUG_SERVER_RECORDS:
        print currentIP, currentMAC, "doesn't have binding, creating"
sys.stdout.flush()
    insertQuery = """INSERT INTO bindings (ipv6Address, macAddress,
agentID, startTime, endTime, endType) values ('%s', '%s', %d, 
%f, %f, %d);""" % (currentIP, currentMAC, agentID, firstSeen,
lastSeen, code)
dbCursor2 = serverDbConnection.cursor()
try:
    dbCursor2.execute(insertQuery)
except MySQLdb.IntegrityError:
    print "Error executing database query, key exists
"
sys.stdout.flush()
dbCursor2.close()
serverDbConnection.commit()
def getAgentRecords(agentID, agentName, agentIPv4, username, password,
    dbName):
    ""
    Obtain Record records from Agent database
    Once records have been obtained, the Agent records table will be
    cleared to prevent re-processing of old records.
    ""
    ## Connect to agent's database
    if DEBUG_AGENT_CONNECTIONS:
        print ("Connecting to Agent " + `agentID` + " (" + agentName + ") at " + agentIPv4)
sys.stdout.flush()
    agentDbConnection=MySQLdb.connect(host=agentIPv4, user=username,
    passwd=password, db=dbName)

    ## Obtain records, remove records to prevent re-processing
    if SERVER_TEST_MODE:
        selectString = """SELECT * FROM records ORDER BY ipv6Address, 
        timestamp;"""
    else:
        selectString = """SELECT * FROM records ORDER BY ipv6Address, 
        timestamp; DELETE FROM records"
    selectCursor = agentDbConnection.cursor()
    selectCursor.execute(selectString)
    results = selectCursor.fetchall() ## Retrieve records
    selectCursor.close()
    agentDbConnection.commit() ## Write changes to db, release lock

    ## Return tuple of records
    return results
initializeTime = time.time()
loopCount = 0
## Server Database connection
serverDbConnection=MySQLdb.connect(host=SERVER_DATABASE_IP4,
    user=SERVER_DATABASE_USER,
    passwd=SERVER_DATABASE_PASSWORD,
    db=SERVER_DATABASE_DBNAME)
## Clear existing bindings from server database

```python
if SERVER_FRESH_START:
    clearString = "DELETE FROM bindings;"
    clearCursor = serverDbConnection.cursor()
    clearCursor.execute(clearString)
    clearCursor.close()
    serverDbConnection.commit()
    print("Deleted existing binding records from server database")
    sys.stdout.flush()
```

## Loop until killed

```python
while 1:
    loopCount += 1
    print("IPAC Server, Iteration #" + `loopCount` + ", active for " +
          prettyTime(time.time()-initializeTime))
    sys.stdout.flush()
```

## Obtain information on active agents

```python
agentSelect = "SELECT * FROM agents WHERE active !=0;"
selectCursor = serverDbConnection.cursor()
selectCursor.execute(agentSelect)
agentInfo = selectCursor.fetchall()
selectCursor.close()
```

## Process each agent independently

```python
for agent in agentInfo:
    (agentID, agentName, agentActive, agentIPv6, agentIPv4,
     username, password, dbName) = agent
```

## Fetch records from agent

```python
results = getAgentRecords(agentID, agentName, agentIPv4,
                           username, password, dbName)
```

## Process records

```python
if not length:  # Agent database is empty, skip processing
    continue
```

## Agent Records Debugging

```python
if DEBUG_AGENT_RECORDS:
    for rec in results:
        print(rec[0], rec[1], rec[3])  # IPv6, MAC, Type
        sys.stdout.flush()
```

## Get content of first record

```python
currentIP = results[0][0]
currentMAC = results[0][1]
firstSeen = results[0][2]
lastSeen = results[0][2]
```

## Process records

```python
# Records already ordered by IP address and timestamp
# Processing logic:
# Iterate through agent records, finding point where node
# left network (Type = 20), agent has no more records for
# that node, or another MAC claims to have the same IPv6
```
## Correlating IPv6 Addresses for Network Situational Awareness

Jason Froehlich

```python
## address.
while results:
    duplicates = []
## Holds records that identify duplicate IPv6 address use
    length = len(results)
    for i in range(length):
## Iterate records
        ip = results[i][0]
        mac = results[i][1]
        timestamp = results[i][2]
        code = results[i][3]

        if ip == currentIP:
## Same IP address as previous
            if mac == currentMAC:
## Same MAC address as previous
                if i == length-1:
## Last item in list
                    ## Create binding record with lastSeen as current timestamp
                    if code != 20:
                        bindingRecord(currentIP, currentMAC,
                                       agentID, firstSeen, timestamp, 10)
                    else:
                        bindingRecord(currentIP, currentMAC,
                                       agentID, firstSeen, timestamp, 20)
                else:
## Not last item in list
                    if code != 20:
## Not ending record, continue
                        lastSeen = timestamp
                    else:
## An ending record
                        ## Create binding record
                        bindingRecord(currentIP, currentMAC,
                                       agentID, firstSeen, timestamp, 20)
                        ## Establish next record as current
                        k = i + 1
                        currentIP = results[k][0]
                        currentMAC = results[k][1]
                        firstSeen = results[k][2]
                        lastSeen = results[k][2]
            else:
## Different MAC address from previous
## Detect Duplicate
                if lastSeen > timestamp - (ACTIVE_IP_REFRESH_TIMEOUT *
                                            (ACTIVE_IP_REFRESH_COUNT+1)):
## Duplicate
                    duplicates.append(results[i])
                else:
## Create binding record for previous
                    bindingRecord(currentIP, currentMAC,
                                   agentID, firstSeen, lastSeen, 11)
## Special processing for last item
                if i == length-1:
## Last item in list
                    if code != 20:
## Not an ending record
                        ## Create binding record for this
                        bindingRecord(results[i][0],
                                       results[i][1], agentID,
                                       results[i][2], results[i][2],
                                       10)
                    else:
## Ending record
                        bindingRecord(results[i][0],
                                       results[i][1], agentID,
                                       results[i][2], results[i][2],
                                       20)
                else:
## Not last item in list
                    if code == 20:
## Ending record
```

- 167 -
bindingRecord(results[i][0],
    results[i][1], agentID,
    results[i][2], results[i][2],
    20)
    k = i + 1  ## Next is current
else:
    k = i  ## This is current
    ## Establish current
    currentIP = results[k][0]
    currentMAC = results[k][1]
    firstSeen = results[k][2]
    lastSeen = results[k][2]

else:  ## Different IP address from previous
    ## Create binding record for previous
    bindingRecord(currentIP, currentMAC, agentID,
                   firstSeen, lastSeen, 10)

    ## Special processing for last item
    if i == length-1:  ## Last item in list
        if code != 20:  ## Not and ending record
            ## Create binding record for this record
            bindingRecord(results[i][0], results[i][1],
                           agentID, results[i][2],
                           results[i][2], 10)
        else:  ## Ending record, ignore
            bindingRecord(results[i][0], results[i][1],
                           agentID, results[i][2],
                           results[i][2], 20)
    else:  ## Not last item in list
        ##> Establish this record as current
        if code == 20:
            bindingRecord(results[i][0], results[i][1],
                           agentID, results[i][2],
                           results[i][2], 20)
        else:
            k = i+1
            currentIP = results[k][0]
            currentMAC = results[k][1]
            firstSeen = results[k][2]
            lastSeen = results[k][2]

results = duplicates

summarizeTraps()  ## Process SNMP MAC Notification Records

time.sleep(SERVER_PROCESSING_INTERVAL)

if __name__ == "__main__":
    daemon = IPACServerDaemon(SERVER_PID_FILE, time.time(),
                               stdout=SERVER_LOG_FILE, stderr=SERVER_LOG_FILE)
    if len(sys.argv) == 2:
        if 'start' == sys.argv[1]:
            daemon.start()
        elif 'stop' == sys.argv[1]:
            daemon.stop()
        elif 'status' == sys.argv[1]:
            daemon.status("IPAC Server")
        elif 'restart' == sys.argv[1]:
            daemon.restart()
```
else:
    print "Unknown command"
    sys.exit(2)

else:
    print "usage: python %s start|status|stop|restart" % sys.argv[0]
    sys.exit(2)
```

### 11.4.1.9 ipac_server_config.py

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

## IPAC Server Configuration

## Jason Froehlich
## Rochester Institute of Technology
## Dept. of Network, Security, and System Administration
## May 17, 2011

## This program is a proof of concept being created for the partial
## fulfillment of the MS in Networking and Systems Administration from the
## Rochester Institute of Technology. All rights reserved.

## Customize for each individual IPAC Server process

## Server Database's MySQL Name
## Format: String corresponding to database name
SERVER_DATABASE_DBNAME = "IPAC_Server"

## Server Database's IPv4 Address
## Format: String containing IPv4 address in "dotted quad" notation ("X.X.X.X")
SERVER_DATABASE_IP4 = "10.100.111.170"

## Server Database's MySQL Password
## Format: String corresponding to SERVER_DATABASE_USER's password
SERVER_DATABASE_PASSWORD = "T1gerCl@ws"

## Server Database's MySQL Username
## User must have read and delete permissions to the 'bindings' table and read
## permissions to the 'agents' table.
## Format: String corresponding to database's username
SERVER_DATABASE_USER = "ipac"

## Log File Location
## All printed and error messages from IPAC Server process will be written to
## the specified file. If file exists, text will be appended (user must have
## write permissions to file). If file doesn't exist, file will be created and
## text appended (user must have write permissions to directory).
## Format: String with full path of desired file location
SERVER_LOG_FILE = "/home/vortex/Desktop/ipacserverout.txt"

## PID File Location
## Holds PID and start time of the IPAC Server process. These values are used
## for server daemon control (status/stop). User must have write permissions
## to the directory.
## Format: String with full path of desired file location
SERVER_PID_FILE = "/home/vortex/Desktop/ipacserver.pid"
```
## SNMP Trap Database's MySQL Name
## Format: String corresponding to database name
TRAP_DATABASE_DBNAME = "IPAC_SNMP"

## SNMP Trap Database's IPv4 Address
## Format: String containing IPv4 address in "dotted quad" notation ("X.X.X.X")
TRAP_DATABASE_IP4 = "10.100.111.170"

## SNMP Trap Database's MySQL Password
## Format: String corresponding to SERVER_DATABASE_USER's password
TRAP_DATABASE_PASSWORD = "T1gerCl@ws"

## SNMP Trap Database's MySQL Username
## User must have read and delete permissions to the 'bindings' table and read
## permissions to the 'agents' table.
## Format: String corresponding to database's username
TRAP_DATABASE_USER = "ipac"

### Server Program Constants ###
### WARNING: Modification of these settings will affect IPAC Server behavior.###
### Do not change these values unless you are sure of their results.###

## Number of times a Refresh Solicitation will be sent before an IP is
## removed from the Active IPs list
## Format: Integer
ACTIVE_IP_REFRESH_COUNT = 2

## Maximum amount of time ago since an IP in the Active IPs list was seen
## before a refresh Solicitation will be sent
## Format: Integer: (seconds)
ACTIVE_IP_REFRESH_TIMEOUT = 30

## Clear database of binding records at IPAC Server initialization
## Format: 'True'/'False'
SERVER_FRESH_START = True

## Interval between Agent Record Processing executions
## Format: Integer (seconds) (3600 = 1h)
SERVER_PROCESSING_INTERVAL = 3600

## Testing Mode: Does not clear Agent records after processing
## Will cause records to be re-processed each server iteration
## SERVER_FRESH_START must also be enabled to prevent MySQL key exists errors
## Format: 'True'/'False'
SERVER_TEST_MODE = True

## Testing Mode: Does not clear Trap records after processing
## Will cause records to be re-processed each trap summarization iteration
## SERVER_FRESH_START must also be enabled to prevent MySQL key exists errors
## Format: 'True'/'False'
TRAP_TEST_MODE = True

### Debugging Constants ###
### WARNING: These could produce a LOT of text in the log files.###
### Set to 'False' unless you are debugging program execution.###
Correlating IPv6 Addresses for Network Situational Awareness

## Format: 'True'/'False'

### Print Agent Information Each Server Iteration
DEBUG_AGENT_CONNECTIONS = False

### Print Records Retrieved From Agent
DEBUG_AGENT_RECORDS = False

### Print Descriptions of Modifications to Server Database
DEBUG_SERVER_RECORDS = False

### Print SNMP Trap Information Each Server Iteration
DEBUG_TRAP_CONNECTIONS = False

### Print Records Retrieved From Trap Database
DEBUG_TRAP_RECORDS = True

11.4.1.10  ipac_textlog.py

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

## IPAC Text Log Processor
## Jason Froehlich
## Rochester Institute of Technology
## Dept. of Network, Security, and System Administration
## May 17, 2011

## This program is a proof of concept being created for the partial
## fulfillment of the MS in Networking and Systems Administration from the
## Rochester Institute of Technology.  All rights reserved.

## Module Imports
import MySQLdb
import re
import sys
from ipac_functions import *

__author__ = "Jason Froehlich"
__copyright__ = "Copyright 2011, Jason Froehlich"
__version__ = "0.1.3"
__email__ = "jason.froehlich@alum.rit.edu"
__status__ = "Development"

##-------------------------------##
##  Server Database Information  ##
##-------------------------------##

SERVER_DATABASE_HOST = "10.100.112.12"  ## Database's IPv4 Address
SERVER_DATABASE_USER = "ipac"  ## Database's MySQL Username
SERVER_DATABASE_PASSWD = "T1gerCl@ws"  ## Database's MySQL Password
SERVER_DATABASE_NAME = "IPAC_Server"  ## Database's MySQL Name
DATE_YEAR = "2011"  ## Used in Syslog Time

def queryDatabase(queryString):
    ""
    Execute given query string on database.
    ""
    databaseConnection=MySQLdb.connect(host=SERVER_DATABASE_HOST,
                                        user=SERVER_DATABASE_USER,
                                        passwd=SERVER_DATABASE_PASSWD,
                                        ...
```python
dbCursor = databaseConnection.cursor()
dbCursor.execute(queryString)
results = dbCursor.fetchall()
dbCursor.close()
databaseConnection.commit()
return results;

if __name__ == "__main__":
    if len(sys.argv) < 2:
        print("usage: %s filename [Agent ID]" % sys.argv[0])
sys.exit(2)

    ## Obtain Agent ID used for Local subnets - parameter or prompt
    if len(sys.argv) > 2:  # Agent argument provided
        try:
            agent = int(sys.argv[2])
        except ValueError:
            try:
                agent = int(raw_input("Local Agent/VLAN: "))
            except ValueError:
                print("Valid Agent ID not given: Must be an Integer")
sys.exit()
    else:  # Agent argument not provided
        try:
            agent = int(raw_input("Local Agent/VLAN: "))
        except ValueError:
            print("Valid Agent ID not given: Must be an Integer")
sys.exit()

    ## Verify Agent ID exists in Server database
    if agent > 0:  # Must be a non-zero positive integer
        agentSelect = "SELECT agentID FROM agents WHERE agentID=%i" % agent
        results = queryDatabase(agentSelect)
        if not results:
            print(Valid Agent ID not given: Agent not known by Server")
sys.exit()
    else:  # Not a non-zero positive integer
        print("Valid Agent ID not given: Must be a non-zero positive Integer")
sys.exit()

    ## Open Input and Output files
    logfile = sys.argv[1]
    outfile = logfile + ".out"
    try:
        log = open(logfile, 'r')
    except IOError:
        print("Unable to open the log file")
sys.exit(2)
    try:
        out = open(outfile, 'w')
    except IOError:
        print("Unable to open the output file for writing")
sys.exit(2)

    ## Obtain Subnet info from Server
    ## Used to know which agent to query for IP address binding
    subnetQuery = "SELECT * FROM subnets;"
    subnets = queryDatabase(subnetQuery)

    ## Regular Expressions for parsing line text
```

Correlating IPv6 Addresses for Network Situational Awareness

Jason Froehlich
Correlating IPv6 Addresses for Network Situational Awareness

Jason Froehlich

```python
reg1 = re.compile('^\s*(((\[0-9A-Fa-f]{1,4}:){7}(\[0-9A-Fa-f]{1,4}|:))|((\[0-9A-Fa-f]{1,4}:){6}((\[0-9A-Fa-f]{1,4}:)(\[0-9A-Fa-f]{1,4}|:))|((\[0-9A-Fa-f]{1,4}:){5}((:|)(\[0-9A-Fa-f]{1,4}:)(\[0-9A-Fa-f]{1,4}|:))|((\[0-9A-Fa-f]{1,4}:){4}((|)(\[0-9A-Fa-f]{1,4}:)(\[0-9A-Fa-f]{1,4}|:))|((\[0-9A-Fa-f]{1,4}:){3}((|)(\[0-9A-Fa-f]{1,4}:)(\[0-9A-Fa-f]{1,4}|:))|((\[0-9A-Fa-f]{1,4}:){2}((|)(\[0-9A-Fa-f]{1,4}:)(\[0-9A-Fa-f]{1,4}|:))|(:|)))(%.+)?\s*$')

## Match IPv6 Addresses

reg3 = re.compile('[^a-zA-Z0-9\:\]')  # Splits by anything not alphanum or :
reg7 = re.compile('[a-zA-Z]+ [a-zA-Z]+ [0-3]\[0-9\] [0-2]\[0-9\]:[0-5]\[0-9\] 20[0-9]\[0-9\]')
reg8 = re.compile('[0-3]\[0-9\]/[a-zA-Z]+/[20[0-9]\[0-9]\[0-5]\[0-5\]:[0-5]\[0-5]\[0-9\][-+]?[0-9]\[0-9\]:[0-5]\[0-5\]]')
reg9 = re.compile('[a-zA-Z]+ [0-3]\[0-9\] [0-2]\[0-9\]:[0-5]\[0-5\]:[0-5]\[0-5\]')

### Apache Error Log Time Format
rawTime = reg7.search(line).group()
timeParts = reg6.split(rawTime)
timeNum = isValidDate(timeString)

### Apache Access Log Time Format
rawTime = reg8.search(line).group()
timeParts = reg6.split(rawTime)
timeString = (timeParts[2] + "-" + monthNameToNum(timeParts[0]) + "-" + timeParts[3] + ":" + timeParts[4] + ":" + timeParts[5])
timeNum = isValidDate(timeString)

### Syslog Time Format
rawTime = reg9.search(line).group()
timeParts = reg6.split(rawTime)
timeString = (DATE_YEAR + "-" + monthNameToNum(timeParts[0]) + "-" + timeParts[1] + ":" + timeParts[2] + ":" + timeParts[3] + ":" + timeParts[4])
timeNum = isValidDate(timeString)
```
else:  ## Timestamp not found
    missingTimestamp.append(lineCount)

## Only continue processing addresses if valid timestamp found
if timeNum:
## Find IPv6 Addresses
    for word in reg3.split(line):
        match = reg1.search(word)
        if match:
            addr = match.string
            if addresses.has_key(addr):
                addresses[addr] += 1
            else:
                addresses[addr] = 1
## Process each IPv6 Address found
    for addr in addresses:
        expanded = expand_ip_address(addr)
        macMatches = {}  ## Holds matching macs and certainty index (*)
        portAssociations = {}
        if expanded[0:4] == "0000":  ## Loopback Address, ignore
            pass
        elif expanded[0:2] == "FF":  ## Multicast Address, ignore
            pass
        elif expanded[0:4] == "FE80":  ## Link-Local Address
            queryString = "SELECT macAddress FROM bindings WHERE \ 
                       agentID=%i and ipv6Address='%s' and startTime < %f and \ 
                       endTime >= %f;" % (agent, addr, timeNum+1, timeNum)
            exactResults = queryDatabase(queryString)
            closeResults = queryDatabase(queryString2)

            if len(closeResults) < 1:  ## No matches found
                if unmatched.has_key(expanded):
                    unmatched[expanded] += 1
                else:
                    unmatched[expanded] = 1
            elif len(exactResults) >= 1:  ## At least one exact match
                if len(exactResults) == len(closeResults):  ## Only Exact
                    if len(exactResults) == 1:
                        macMatches[exactResults[0][0]] = "****"
                    else:  ## More than one exact match
                        for mac in exactResults:
                            macMatches[mac[0]] = "***"
                    else:  ## Also Close matches
                        if len(exactResults) == 1:
                            macMatches[exactResults[0][0]] = "***"
                        else:  ## More than one exact match
                            for mac in exactResults:
                                macMatches[mac[0]] = "***"
                        for mac in closeResults:
                            if exactResults.count(mac[0]):
                                ## Mac already in Exact matches, ignore
                                pass
                            else:
                                macMatches[mac[0]] = ""
                else:  ## No exact matches, only close
                    for mac in closeResults:
                        macMatches[mac[0]] = "*"
            else:  ## Other non-local address, translate if possible
### Find agent that observes that subnet

```python
def find_agent(subnets)
    target_agent = 0
    for subnet in subnets:
        if expanded[0:16] == subnet[0:16]:
            target_agent = subnet[2]

    if target_agent:
        # Address is in organization's network
        queryString = "SELECT macAddress FROM bindings WHERE agentID = %i and ipv6Address = '%s' and startTime < %f and endTime >= %f;" % (target_agent, expanded, timeNum + 1, timeNum)
        closeResults = queryDatabase(queryString)
        exactResults = queryDatabase(queryString2)

        if len(closeResults) < 1:  # No matches found
            if unmatched.has_key(expanded):
                unmatched[expanded] += 1
            else:
                unmatched[expanded] = 1
        elif len(exactResults) >= 1:  # At least one exact
            if len(exactResults) == len(closeResults):
                # Only Exact matches
                macMatches[exactResults[0][0]] = "****"
            elif len(exactResults) == 1:
                # Only one match
                macMatches[exactResults[0][0]] = "****"
            else:
                # More than one exact match
                for mac in exactResults:
                    macMatches[mac[0]] = "****"
                if exactResults.count(mac[0]) == 1:
                    # Mac already in Exact matches, ignore
                    pass
                else:
                    macMatches[mac[0]] = "*****"
            else:
                # No exact matches, only close
                for mac in closeResults:
                    macMatches[mac[0]] = "**"
        else:
            # Address outside of organization's network
            if notLocal.has_key(expanded):
                notLocal[expanded] += 1
            else:
                notLocal[expanded] = 1
```

### Find Ports Associated to MAC Addresses

```python
for mac in macMatches.keys():
    portMatches = {}
    # Holds matching ports and certainty index (*)
    queryString = "SELECT switchID, switchPort, vlan FROM portBindings WHERE macAddress = '%s' and startTime < %f and endTime >= %f;" % (mac, timeNum + 1, timeNum)
    closeResults = queryDatabase(queryString)
    queryString2 = "SELECT switchID, switchPort, vlan FROM portBindings WHERE macAddress = '%s' and startTime < %f and endTime >= %f;" % (mac, timeNum + 1, timeNum)
    exactResults = queryDatabase(queryString2)
    if len(closeResults) < 1:
        pass
    else:
        for mac in exactResults:
            portMatches[mac[0]] = "**"
        if exactResults.count(mac[0]) == 1:
            # Mac already in Exact matches, ignore
            pass
        else:
            for mac in closeResults:
                portMatches[mac[0]] = "*****"
```

---
```python
timeNum - 30)
exactResults = queryDatabase(queryString)
closeResults = queryDatabase(queryString2)

if len(closeResults) < 1:  # No matches found
    pass
elif len(exactResults) >= 1:  # At least one exact match
    if len(exactResults) == len(closeResults):  # Only Exact
        portInfo = (exactResults[0][0] + "/" +
                    exactResults[0][1] + "/" +
                    exactResults[0][2])
        portMatches[portInfo] = "^^^^"
    else:  # More than one exact match
        for port in exactResults:
            portInfo = (port[0] + "/" + port[1] + "/" +
                        port[2])
            portMatches[portInfo] = "^^"
        else:  # Also Close matches
            if len(exactResults) == 1:
                portInfo = (exactResults[0][0] + "/" +
                            exactResults[0][1] + "/" +
                            exactResults[0][2])
                portMatches[portInfo] = "^^^"
            else:  # More than one exact match
                for port in exactResults:
                    portInfo = (port[0] + "/" + port[1] + "/" +
                                port[2])
                    portMatches[portInfo] = "^^"
                for port in closeResults:
                    if exactResults.count(port):
                        # Port already in Exact matches, ignore
                        pass
                    else:
                        portInfo = (port[0] + "/" + port[1] + "/" +
                                   port[2])
                        portMatches[portInfo] = ""  
            else:  # > 1 Port
                strings = []
                for port in portMatches.keys():
                    portString = port + portMatches[port]
                    strings.append(portString)
                joined = ", " + joined + ")"
                tempstr = "^"  
    tempstr = ""
else:  # > 1 Port
    strings = []
    for port in portMatches.keys():
        portString = port + portMatches[port]
        strings.append(portString)
    joined = ", " + joined + ")"
    tempstr = "^"  
portAssociations[mac] = tempstr
```

---

- 176 -
mac = macMatches.keys()[0]
str = str + mac + macMatches[mac] + portAssociations[mac]
else:  ## > 1 MACs
    strings = []
    for mac in macMatches.keys():
        macString = (mac + macMatches[mac] + portAssociations[mac])
        strings.append(macString)
joined = ", " . join(strings)
str = str + joined
lineSum.append(str)

## Generate Final Line Summary
if lineSum:
    summary = " || ". join(lineSum)
    summary = " <[ " + summary + "]> "
    out.write(line.rstrip() + summary + 
"
"

## Report
out.write("\n")
if missingTimestamp:
    out.write("Unable to find timestamp on lines (lines not processed):
")
    out.write(" + ", ". join(missingTimestamp) + "\n")
if unmatched:
    out.write("No Matches found for:\n")
    for key in sorted(unmatched.keys()):
        out.write(" " + prettyIP(key) + " (" + `unmatched[key]` + " times)\n")
if notLocal:
    out.write("Not In Local Subnets:\n")
    for key in sorted(notLocal.keys()):
        out.write(" " + prettyIP(key) + " (" + `notLocal[key]` + " times)\n")

## Confidence Key
key = "" ":Confidence Indicators:
"**** = No Duplicates, Exact match
"*** = Duplicates, Exact Match
"** = Duplicates, More Than One Exact Match
"* = All Are Close Matches
[none] = Duplicates, Close Match"
out.write(key)

## Clean Up
log.close()
out.close()

11.4.11  ipac_threads.py

#!/usr/bin/python
## IPAC Agent Thread Classes
## Jason Froehlich
## Rochester Institute of Technology
## Dept. of Network, Security, and System Administration
## May 17, 2011
## This program is a proof of concept being created for the partial
## fulfillment of the MS in Networking and Systems Administration from the
## Rochester Institute of Technology. All rights reserved.
## Module Imports

```python
import binascii
import dpkt
import MySQLdb
import Queue
import socket
import sys
import time
import threading
from ipac_agent_config import *
from ipac_functions import timeString
```

__author__ = "Jason Froehlich"

__copyright__ = "Copyright 2011, Jason Froehlich"

__version__ = "0.1.4"

__email__ = "jason.froehlich@alum.rit.edu"

__status__ = "Development"

```python
class NDMulticastGenerator(threading.Thread):
    """
    Generates Multicast ND Solicitations, Updates Expected Advert List (Thread)
    """
    def __init__(self, qPrinter, replyWaiting):
        threading.Thread.__init__(self)
        self.setDaemon(1)
        self.incomingAddressQueue = Queue.Queue()
        self.resultQueue = Queue.Queue()
        self.qPrinter = qPrinter
        self.replyWaiting = replyWaiting
        self.start()
        def generate(self, ip6Target, updateRW = True):
            """
            Adds Packets to Generator Queue, Returns IP + Timestamp
            """
            self.incomingAddressQueue.put((ip6Target, updateRW))
            return self.resultQueue.get()
    def run(self):
        """
        Body of Multicast ND Solicit Generator Thread
        """
        while 1:
            incoming = self.incomingAddressQueue.get()
            target = incoming[0]
            updateRW = incoming[1]
            # Construct Solicitation Packet
            pcl = PacketConstructor()
            pkt = pcl.multicastNDSol(SYSTEM_DEFAULT_IP,
                                     SYSTEM_DEFAULT_MAC, target)
            # Create Socket
            soc = socket.socket(socket.AF_PACKET, socket.SOCK_RAW)
            soc.bind((SYSTEM_DEFAULT_IFACE, dpkt.ethernet.ETH_TYPE_IP6))
            timeSent = time.time()
            # Add to Expected Unicast Advertisements list
```
Correlating IPv6 Addresses for Network Situational Awareness

### Done before packet is sent to prevent reply from returning
### before agent is expecting it

```python
if updateRW:
    self.replyWaiting.action("add", (target, "NULL", timeSent))

## Send Packet
soc.send(str(pkt))

## Send Packet
if PACKET_SENT_DEBUG:
soc.send(str(pkt))

##  Add to Expected Unicast Advertisements list
##    Done before packet is sent to prevent reply from returning
```

```python
class NDUnicastGenerator(threading.Thread):
    
    """
    Generates Unicast ND Solicitations, Updates Expected Advert List (Thread)
    """

    def __init__(self, qPrinter, replyWaiting):
        threading.Thread.__init__(self)
        self.setDaemon(1)
        self.incomingAddressQueue = Queue.Queue()
        self.resultQueue = Queue.Queue()
        self.qPrinter = qPrinter
        self.replyWaiting = replyWaiting
        self.start()

    def generate(self, ip6Target, macTarget, updateRW = True):
        """
        Adds Packets to Generator Queue, Returns IP + Timestamp
        """

        self.incomingAddressQueue.put((ip6Target, macTarget, updateRW))

        while 1:
            target = self.incomingAddressQueue.get()
            updateRW = target[2]

            # Construct Solicitation Packet
            pc1 = PacketConstructor()
            pkt = pc1.unicastNDSol(SYSTEM_DEFAULT_IP, SYSTEM_DEFAULT_MAC,
                                target[0], target[1])

            # Create Socket
            soc = socket.socket(socket.AF_PACKET, socket.SOCK_RAW)
            soc.bind((SYSTEM_DEFAULT_IFACE, dpkt.ethernet.ETH_TYPE_IP6))
            timeSent = time.time()

            soc.send(str(pkt))

            if PACKET_SENT_DEBUG:
                soc.send(str(pkt))

            self.resultQueue.put((target, timeSent))
```

- 179 -
Correlating IPv6 Addresses for Network Situational Awareness

Jason Froehlich

138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199

## before agent is expecting it
if updateRW:
    self.replyWaiting.action("add",
        (target[0], target[1], timeSent))

## Send Packet
soc.send(str(pkt))

if PACKET_SENT_DEBUG:
    self.qPrinter.printq("Unicast ND Solicit sent to " + target[0] +
        " at MAC " + target[1], "UnicastGenerator")
    self.qPrinter.printq(binascii.b2a_hex(str(pkt)),
        "UnicastGenerator")

self.resultQueue.put((target[0], target[1], timeSent))

class DatabaseInterface(threading.Thread):
    
    Executes Queries to Agent Database (Thread)
    
    def __init__(self, fresh, qPrinter):
        threading.Thread.__init__(self)
        self.setDaemon(True)
        self.incomingRequestQueue = Queue.Queue()
        self.resultQueue = Queue.Queue()
        self.FRESH_START = fresh
        self.qPrinter = qPrinter
        self.start()

    def execute(self, requestString):
        
        Adds SQL Commands to Database Interface Queue, Returns Query Results
        
        Called By Other IPAC Agent Threads
        
        self.incomingRequestQueue.put(requestString)
        return self.resultQueue.get()

    def run(self):
        
        Body of Agent Database Interface Thread
        
        databaseConnection = MySQLdb.connect(host=AGENT_DATABASE_IP4,
            user=AGENT_DATABASE_USER,
            passwd=AGENT_DATABASE_PASSWORD,
            db=AGENT_DATABASE_DBNAME)

        if self.FRESH_START:
            cleanupString = """DELETE FROM active; DELETE FROM records;""
            cleanupCursor = databaseConnection.cursor()
            cleanupCursor.execute(cleanupString)
            cleanupCursor.close()
            databaseConnection.commit()
            self.qPrinter.printq(" Deleted old records from database",
                "DatabaseInterface")
        else:
            self.qPrinter.printq(" Keeping existing records in database",
                "DatabaseInterface")

        while 1:
Correlating IPv6 Addresses for Network Situational Awareness

Jason Froehlich

```python
import databaseConnection

dbCursor = databaseConnection.cursor()
request = self.incomingRequestQueue.get()
try:
    dbCursor.execute(request)
    self.resultQueue.put(dbCursor.fetchall())
except MySQLdb.IntegrityError:
    self.qPrinter.printq("Error executing database query, " +
"key exists:\n " + request)
    self.resultQueue.put([])
dbCursor.close()
databaseConnection.commit()

class PacketConstructor():
    """
    Constructs ICMPv6 NDP Solicitation Packets
    """

    def completePkt(self, icmp6, srcIP_h, srcMAC_h, dstIP_h, dstMAC_h):
        """
        Assembles IPv6 and Ethernet portions of packet
        """

        # IPv6 Packet
        ip6 = dpkt.ip6.IP6()
        ip6.src = srcIP_h
        ip6.dst = dstIP_h
        ip6.nxt = 58  # Next Header: ICMPv6
        ip6.hlim = 255  # Hop Limit, 255 for Neighbor Discovery packets
        ip6.plen = 32  # Payload Length
        ip6.data = icmp6

        if PACKET_CAPTURE_DEBUG:  # Allows easy differentiation between
            # IPAC and regular system solicits
            ip6._set_fc(255)  # Set Traffic Class

        # Ethernet Frame
        eth = dpkt.ethernet.Ethernet()
        eth.src = srcMAC_h
        eth.dst = dstMAC_h
        eth.type = 0x86dd  # Payload type: IPv6
        eth.data = ip6

        return eth

    def unicastNDSol(self, srcIP_a, srcMAC_a, targetIP_a, targetMAC_a):
        """
        Assembles ICMPv6 Portion of Unicast NDP Solicitation Packet
        """

        # _a: ASCII format, _h: HEX format

        # Convert ASCII Addresses to Hex
        srcIP_h = binascii.a2b_hex(srcIP_a)
        srcMAC_h = binascii.a2b_hex(srcMAC_a)
        targetIP_h = binascii.a2b_hex(targetIP_a)
        targetMAC_h = binascii.a2b_hex(targetMAC_a)

        # ICMPv6 Datagram
        icmp6 = dpkt.icmp6.ICMP6()
        icmp6.type = 135  # Neighbor Solicitation (Discover)
        icmp6.data = "\x00\x00\x00\x00\x00\x00" + targetIP_h + "\x01\x01" + srcMAC_h

        pkt = self.completePkt(icmp6, srcIP_h, srcMAC_h, targetIP_h,
```

- 181 -
Correlating IPv6 Addresses for Network Situational Awareness

Jason Froehlich

```python
def multicastNDSol(self, srcIP_a, srcMAC_a, targetIP_a):
    
    """
    Assembles ICMPv6 Portion of Multicast NDP Solicitation Packet
    """
    ## _a: ASCII format, _h: HEX format
    
    ## Multicast Destination Addresses
    ND_MULTICAST_DST_IP_PRE = ("\xff\x02\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x01\xff")
    ND_MULTICAST_DST_MAC_PRE = "\x33\x33\xff"
    
    ## Convert ASCII Addresses to Hex
    srcIP_h = binascii.a2b_hex(srcIP_a)
    srcMAC_h = binascii.a2b_hex(srcMAC_a)
    targetIP_h = binascii.a2b_hex(targetIP_a)
    
    ## Generate Solicited-Node Addresses
    ## FF02:0:0:0:0:1:FF__:____ plus last 24 bits of target's unicast addr
    ## 33:33:ff:__:__:__ plus last 24 bits of target's unicast IPv6 addr
    dstIP_h = ND_MULTICAST_DST_IP_PRE + targetIP_h[13:16]
    dstMAC_h = ND_MULTICAST_DST_MAC_PRE + targetIP_h[13:16]
    
    ## ICMPv6 Datagram
    icmp6 = dpkt.icmp6.ICMP6()
    icmp6.type = 135  ## Neighbor Solicitation (Discover)
    icmp6.data = "\x00\x00\x00\x00" + targetIP_h + "\x01\x01" + srcMAC_h
    pkt = self.completePkt(icmp6, srcIP_h, srcMAC_h, dstIP_h, dstMAC_h)
    return pkt

class LogMarker(threading.Thread):
    """
    Creates Periodic MARKs in Agent Log, Summarizes Processing Stats (Thread)
    """
    def __init__(self, counts, qPrinter):
        threading.Thread.__init__(self)
        self.setDaemon(1)
        self.counts = counts
        self.qPrinter = qPrinter
        self.start()
        def run(self):
            """
            Body of Agent Log Marker Thread
            """
            initSolCount = self.counts[0]
            initAdvCount = self.counts[1]
            while 1:
                time.sleep(LOG_MARK_INTERVAL)
                printstring = ("MARK - " + timeString(time.time()) + " - " +
                                "solicits, " + "advert processed"")
                printstring = ("MARK - " + timeString(time.time()) + " - " +
                                "advert processed"")
                initSolCount = self.counts[0]
                initAdvCount = self.counts[1]
                self.qPrinter.printq(printstring, "LogMarker")
```

- 182 -
class PendingProcessor(threading.Thread):
    
    Processes Targets from Incoming Adverts (Thread)
    Delays Target Processing to Catch Duplicate Adverts
    
    __init__(self, dbInterfaceThread, qPrinter):
        threading.Thread.__init__(self)
        self.setDaemon(1)
        self.pendingAdv = {}
        ## Dictionary=  { [ipAddress] : { [macAddress] : (timestamps, ...) } }
        self.incomingPendingAdv = Queue.Queue()
        self.qPrinter = qPrinter
        self.start()
        self.dbInterfaceThread = dbInterfaceThread
    add(self, ip6Addr, macAddr, timestamp):
        
        Adds Received Addresses to List of Pending Addresses
        Called By Main IPAC Agent Thread
        
        self.incomingPendingAdv.put((ip6Addr, macAddr, timestamp))
    run(self):
        
        Body of Pending Processor Thread
        
        while 1:
            currentTime = time.time()
            emptyIPs = []
            
            ## Add new to list
            while not self.incomingPendingAdv.empty():
                ip6Addr, macAddr, timestamp = self.incomingPendingAdv.get()
                if self.pendingAdv.has_key(ip6Addr):  ## IP already in pending
                    if self.pendingAdv[ip6Addr].has_key(macAddr):
                        ## MAC already in IP, add timestamp
                        self.pendingAdv[ip6Addr][macAddr].append(timestamp)
                    else:  ## MAC not in IP, add MAC and timestamp
                        self.pendingAdv[ip6Addr][macAddr] = [timestamp]
                else:  ## IP not in pending, add IP, MAC and timestamp
                    self.pendingAdv[ip6Addr] = {macAddr:[timestamp]}
            
            ## Process items in list
            for ip6Addr in self.pendingAdv:
                emptyMACs = []
                ## Get number of macs with timestamps
                macCount = 0
                for macAddr in self.pendingAdv[ip6Addr]:
                    if (len(self.pendingAdv[ip6Addr][macAddr]) == 0):
                        macCount += 1
                    if macCount == 1:  ## Reply from only 1 MAC
                        ## Iterate to find MAC with timestamp
                        for macAddr in self.pendingAdv[ip6Addr]:
                            if len(self.pendingAdv[ip6Addr][macAddr]) == 0:
                                ## IP/MAC has pending timestamps
                                lastSeen = 0
                                for timestamp in self.pendingAdv[ip6Addr][macAddr]:
                                    if (timestamp < currentTime -
Correlating IPv6 Addresses for Network Situational Awareness
Jason Froehlich

386  SOLICIT_TIMEOUT_INTERVAL):
387       lastSeen = timestamp
388       #> Create Record
389       recordQuery = """INSERT INTO records \n390           (ipv6Address,macAddress,timestamp,type) \n391           values ('%s','%s',%f,%d);""" % (ip6Addr,
392           macAddr, timestamp, 11)
393       recordResult = (self.dbInterfaceThread.execute(recordQuery))
394       self.pendingAdv[ip6Addr][macAddr].remove(timestamp)
395
396       if lastSeen != 0:
397           #> Add to / Update Active
398           query = """SELECT * FROM active WHERE \n399           (ipv6Address = '%s' and macAddress='%s');""" % (ip6Addr, macAddr)
400           qResult = self.dbInterfaceThread.execute(query)
401           if qResult:
402               ## IP exists in active list, update
403               if qResult[0][2] < lastSeen:
404                   ## Newer than lastSeen timestamp in db
405                   activeQuery = """UPDATE active SET \n406                       lastSeen='%f', retryFailures=0 WHERE \n407                       ipv6Address='%s' and macAddress='%s';""" % (lastSeen, ip6Addr, macAddr)
408                   activeResult = self.dbInterfaceThread.execute(activeQuery)
409               else:
410                   ## Older than lastSeen timestamp in db
411                   activeQuery = """UPDATE active SET \n412                       retryFailures=0 WHERE ipv6Address='%s' \n413                       and macAddress='%s';""" % (ip6Addr, macAddr)
414                   activeResult = self.dbInterfaceThread.execute(activeQuery)
415           else:
416               ## IP not in active list, add
417               activeQuery = """INSERT INTO active \n418                       (ipv6Address, macAddress, lastSeen, retryFailures) values ('%s','%s',%f,%d);""" % (ip6Addr, macAddr, lastSeen, 0)
419               activeResult = self.dbInterfaceThread.execute(activeQuery)
420           else:
421               ## IP/MAC has no pending timestamps, clear MAC
422               emptyMACs.append(macAddr)
423               for mac in emptyMACs:
424                   self.pendingAdv[ip6Addr].pop(mac)
425
426       elif macCount > 1:  ## Reply from 2+ MACs, Problem
427           macs = []
428           for macAddr in self.pendingAdv[ip6Addr]:
429               if (len(self.pendingAdv[ip6Addr][macAddr]) != 0):
430                   macs.append(macAddr)
431                   lastSeen = 0
432                   for timestamp in self.pendingAdv[ip6Addr][macAddr]:
433                       if (timestamp < currentTime -
434                           SOLICIT_TIMEOUT_INTERVAL):
435                           lastSeen = timestamp
436                       #> Create Record
437                       recordQuery = """INSERT INTO records \n438                           (ipv6Address, macAddress, timestamp, type) \n439                           values ('%s','%s',%f,%d);""" % (ip6Addr,
440                           macAddr, timestamp, 15)
recordResult = (self.dbInterfaceThread.execute(recordQuery))
self.pendingAdv[ip6Addr][macAddr].remove(timestamp)

if lastSeen != 0:
    #> Add to / Update Active
    query = """SELECT * FROM active WHERE \n    (ipv6Address = '%s' and macAddress='%s');\""" % (ip6Addr, macAddr)
    qResult = self.dbInterfaceThread.execute(query)
if qResult:  # IP exists in active list, update
    if qResult[0][2] < lastSeen:
        ## Newer than lastSeen timestamp in db
        activeQuery = """UPDATE active SET \n        lastSeen='%f', retryFailures=0 WHERE \n        ipv6Address='%s' and macAddress='%s'\""" % (lastSeen, ip6Addr, macAddr)
        activeResult = self.dbInterfaceThread.execute(activeQuery)
    else:  # Older than lastSeen timestamp in db
        activeQuery = """UPDATE active SET \n        retryFailures=0 WHERE ipv6Address='%s' \n        and macAddress='%s'\""" % (ip6Addr, macAddr)
        activeResult = self.dbInterfaceThread.execute(activeQuery)
else:  # IP not in active list, add
    activeQuery = """INSERT INTO active \n    (ipv6Address, macAddress, lastSeen, \n    retryFailures) values ('%s', '%s', %f, %d);\""" % (ip6Addr, macAddr, lastSeen, 0)
    activeResult = self.dbInterfaceThread.execute(activeQuery)
else:  # last seen not past threshold
    pass
else:  # IP/MAC has no pending timestamps, clear MAC
    emptyMACs.append(macAddr)
    for mac in emptyMACs:
        self.pendingAdv[ip6Addr].pop(mac)
        self.qPrinter.printq("WARNING: More than one MAC active " + "for IP " + ip6Addr + ";\n" + `macs`)

else:  # IP has no pending timestamps, clear IP
    emptyIPs.append(ip6Addr)

    ## Clean up list
    for ip6Addr in emptyIPs:  # Clean up empty IP addresses
        self.pendingAdv.pop(ip6Addr)

if PENDING_DEBUG:
    self.qPrinter.printq("PendingProcessor has completed execution," + " sleeping for " + `(PENDING_PROCESS_INTERVAL / 2)`)
    self.qPrinter.printq("Pending List Contents:\n" + self.pendingAdv, "PendingProcessor")
    time.sleep(PENDING_PROCESS_INTERVAL/2)

class SolicitationResender(threading.Thread):
    """
Resends Solicitations to IPs in ReplyWaiting List With No Replies (Thread)

```
def __init__(self, replyWaiting, ndSolicitMultiThread,
         ndSolicitUniThread, qPrinter):
    threading.Thread.__init__(self)
    self.setDaemon(1)
    ## Dictionary= ('ipAddress','macAddress'):
    ##                 ('lastSentTimestamp','replyCount','timeoutCount')
    self.replyWaitingThread = replyWaiting
    self.ndSolicitMultiThread = ndSolicitMultiThread
    self.ndSolicitUniThread = ndSolicitUniThread
    self.qPrinter = qPrinter
    self.start()

def run(self):
    ""
    Body of Solicitation Resender Thread
    ""

    while 1:
        sendTo = self.replyWaitingThread.action("solResend")
        for ip_mac in sendTo:
            if ip_mac[1] == "NULL":
                resendResults = (  
                    self.ndSolicitMultiThread.generate(ip_mac[0], False))
            else:
                resendResults = self.ndSolicitUniThread.generate(ip_mac[0],
                ip_mac[1], False)
            if SOLICITATION_RESEND_DEBUG:
                self.qPrinter.printq("SolicitationResender has completed, " +
                    "sleeping for " + `SOLICIT_TIMEOUT_INTERVAL`,
                    "SolicitationResender")
            time.sleep(SOLICIT_TIMEOUT_INTERVAL)

class ActiveIPRefresher(threading.Thread):
    ""
    Checks Addresses in Active IP List to Verify Still Online (Thread)
    ""

def __init__(self,dbInterfaceThread,ndSolicitUniThread, qPrinter):
    threading.Thread.__init__(self)
    self.setDaemon(1)
    self.dbInterfaceThread = dbInterfaceThread
    self.ndSolicitUniThread = ndSolicitUniThread
    self.qPrinter = qPrinter
    self.startTime = time.time()
    self.start()

def run(self):
    ""
    Body of Active IP Refresher Thread
    ""

    while 1:
        selectString = "SELECT * FROM active;"
        activeIPs = self.dbInterfaceThread.execute(selectString)
        for ipRecord in activeIPs:
            ## ipRecord: ipv6Address, macAddress, lastSeen, retryFailures
            if ipRecord[2] < (time.time() - ACTIVE_IP_REFRESH_TIMEOUT -
            (ACTIVE_IP_REFRESH_TIMEOUT * ipRecord[3])):
                ## Host not seen in XX second
                if ACTIVE_IP_REFRESHER_DEBUG:
                    self.qPrinter.printq("IP " + ipRecord[0] +

Correlating IPv6 Addresses for Network Situational Awareness

Jason Froehlich

```python
572  " not seen in " + `(time.time() -
573  ipRecord[2])` + " seconds...",
574  "ActiveRefresher")
575  
576  timeThreshold = (time.time() - (ACTIVE_IP_REFRESH_TIMEOUT
577  * (ACTIVE_IP_REFRESH_COUNT + 1)))
578  if (ipRecord[2] < timeThreshold) and (time.time() >
579  (self.startTime + ACTIVE_IP_REFRESH_TIMEOUT)):
580      ## IP has not responded in XX refresh cycles and
581      ## program hasn't just started, remove
582      removeString = "DELETE FROM active WHERE (ipv6Address\n583                 = '%s' AND macAddress = '%s');" % (ipRecord[0],
584                ipRecord[1])
585      recordString = "INSERT INTO records (ipv6Address, \n586                        macAddress, timestamp, type) values ('%s', '%s', %f, \n587                        %d);" % (ipRecord[0], ipRecord[1], timeThreshold, 20)
588      self.dbInterfaceThread.execute(removeString)
589      self.dbInterfaceThread.execute(recordString)
590      if ACTIVE_REFRESHER_DEBUG:
591          tmpStr = ("Deleted " + ipRecord[0] + "-" +
592                 ipRecord[1] + " from active list, " +
593                 "inactive for " + `(time.time() -
594                 ipRecord[2])` + " seconds")
595          self.qPrinter.printq(tmpStr, "ActiveRefresher")
596  else:
597      ## Send solicitation to refresh
598      newSolicit = (self.ndSolicitUniThread.generate(ipRecord[0],
599                        ipRecord[1]))
600      updateString = "UPDATE active SET retryFailures=%d \n601                   WHERE (ipv6Address='%s' and macAddress='%s');" % (ipRecord[3]+1,ipRecord[0],ipRecord[1])
602      self.dbInterfaceThread.execute(updateString)
603      if ACTIVE_REFRESHER_DEBUG:
604          tmpStr = ("Active refresh solicitation sent to " +
605                  newSolicit[0] + " at " + newSolicit[1])
606          self.qPrinter.printq(tmpStr, "ActiveRefresher")
607          self.qPrinter.printq("Active IP Refresher completed, " +
608                  "sleeping for " + `ACTIVE_IP_REFRESH_INTERVAL`,
609                  "ActiveRefresher")
610      time.sleep(ACTIVE_IP_REFRESH_INTERVAL)

611  class QueuedPrinter(threading.Thread):
612      ""
613      Writes Messages to Agent Log in Order (Thread)
614      ""
615      Prevents simultaneous writing to Agent log, which could result in
616      spliced messages.
617      ""
618  def __init__(self):
619      threading.Thread.__init__(self)
620      self.setDaemon(1)
621      self.printQueue = Queue.Queue()
622      self.start()
623  def printq(self, message, origin="--"):  
624      ""
625      Adds Messages to Queued Printer Queue
```
Called By Other IPAC Agent Threads, expects message string and originator name string

```python
self.printQueue.put((message, origin, time.time()))
```

```python
def run(self):
    
    Body of Agent Log Queued Printer Thread
    
    while 1:
        message, origin, timestamp = self.printQueue.get()
        if PRINT_DEBUGGING:
            print "From", origin, "at", `timestamp` + ": ", message
            sys.stdout.flush()
        else:
            print message
            sys.stdout.flush()
```

```python
class ReplyWaiting(threading.Thread):
    
    Maintains List of Addresses the Agent Expects an Advertisement From
    
    def __init__(self, qPrinter):
        threading.Thread.__init__(self)
        self.setDaemon(1)
        self.data = {}  # Dictionary= ('ipAddress','macAddress') :
        # ('lastSentTimestamp','replyCount','timeoutCount')
        self.incomingCmds = Queue.Queue()
        self.outgoingResults = {}
        self.qPrinter = qPrinter
        self.start()

    def action(self, command, options="--"):
        
        Adds Processing Commands to Reply Waiting Command Queue
        
        Called By Other IPAC Agent Threads, commands are identified by
        timestamp to ensure the proper output is returned for each command.
        
        timestamp = time.time()
        if REPLY_WAITING_DEBUG:
            self.qPrinter.printq("ReplyWaiting: " + command + " (" +
                \"options\" + ")", "ReplyWaiting")
        self.incomingCmds.put((timestamp, command, options))
        if REPLY_WAITING_DEBUG:
            self.qPrinter.printq("InQueued: " + command + " (" + \"options\" +
                ")", "ReplyWaiting")
        while not self.outgoingResults.has_key(timestamp):
            time.sleep(.125)
            result = self.outgoingResults.pop(timestamp)
            if REPLY_WAITING_DEBUG:
                self.qPrinter.printq("Return: " + \"result\" + " (" + command + ")", "ReplyWaiting")
        return result
    
    def run(self):
        
        Body of Reply Waiting List Thread
        ```
while 1:
    timestamp, command, options = self.incomingCmds.get()
    if REPLY_WAITING_DEBUG:
        self.qPrinter.printq("Processing: " + command + " (" +
        'options' + ")", "ReplyWaiting")

    ## Add Address to List of Waiting Addresses
    if command == "add":
        ## Options: (ipAddr, macAddr, timestamp)
        self.data[(options[0], options[1])] = (options[2], 0, 0)
        self.outgoingResults[timestamp] = True

    ## Check List for Address, Update replyCount if Found
    elif command == "incomingAdv":
        ## Options: (ipAddr, macAddr)
        valid = False
        if self.data.has_key(options):
            ## Values: (lastSentTimestamp, replyCount, timeoutCount)
            values = self.data[options]
            if (values[0] < (time.time() - SOLICIT_TIMEOUT_INTERVAL)):
                valid = True
                self.data[options] = (values[0], values[1] + 1,
                                      values[2])
            if self.data.has_key((options[0], "NULL")):
                values = self.data[(options[0], "NULL")]
                if (values[0] < (time.time() - SOLICIT_TIMEOUT_INTERVAL)):
                    valid = True
                    self.data[(options[0], "NULL")] = (values[0],
                                                          values[1] + 1,
                                                          values[2])
        self.outgoingResults[timestamp] = valid

    ## Iterate List for Addresses Without Replies Within Threshold
    ## Remove Addresses Reaching Timeout Threshold,
    ## Return List of Addresses Needing Packet To Be Resent
    elif command == "solResend":
        sendTo = []
        timedOut = []
        for ip_mac in self.data:
            ## Values: (lastSentTimestamp, replyCount, timeoutCount)
            values = self.data[ip_mac]
            if (values[2] < SOLICIT_RETRY_COUNT):
                ## Not reached resend threshold
                if (values[0] < (time.time() -
                                SOLICIT_TIMEOUT_INTERVAL)) and (values[1] < 1):
                    ## last solicit > than XX seconds ago and no replies
                    sendTo.append(ip_mac)
                    self.data[ip_mac] = (time.time(), 0, values[2] + 1)
                else:
                    ## reply received or not reached timeout
                    pass
            else:
                ## already sent max number of retries
                timedOut.append(ip_mac)

        ## Cleanup data, remove entries reaching max number of retries
        for ip_mac in timedOut:
            self.data.pop(ip_mac)

        ## Return addresses needing retry
        self.outgoingResults[timestamp] = sendTo

    ## Return Data Stored for Given Address
elif command == "get":
    if self.data.has_key(options):
        self.outgoingResults[timestamp] = self.data[options]
    else:
        self.outgoingResults[timestamp] = False

## Return All Addresses and Data Values
elif command == "getall":
    self.outgoingResults[timestamp] = self.data

## Check List For Address, Returns True or False
elif command == "haskey":
    self.outgoingResults[timestamp] = self.data.has_key(options)

## Command Not Recognized, Warn
else:
    self.qPrinter.printq("ERROR in ReplyWaiting: No " + command)
    self.outgoingResults[timestamp] = False

## Main Execution, Do Nothing
if __name__ == "__main__":
    pass

11.4.1.12 ipac_traphandler.py
Correlating IPv6 Addresses for Network Situational Awareness

### Customize for each individual IPAC Agent process

DATABASE_HOST = "10.100.111.170"  ## Database's IPv4 Address
DATABASE_USER = "ipac"  ## Database's MySQL Username
DATABASE_PASSWD = "T1gerCl@ws"  ## Database's MySQL Password
DATABASE_DBNAME = "IPAC_SNMP"  ## Database's MySQL Name
FRESH_START = False  ## Clear database at Trap Handler initialization
TEST_MODE = False  ## Don't execute database commands
TRAPHANDLER_LOG_FILE = "/home/vortex/Desktop/ipactrapout.txt"  ## Log File Location
## All printed and error messages from IPAC Trap Handler process will be written to the specified file. If file exists, text will be appended (user must have write permissions to file). If file doesn't exist, file will be created and used for server daemon control (status/stop). User must have write permissions to the directory.

### Debugging Constants

WARNING: These could produce a LOT of text in the log files.
Set to 'False' unless you are debugging program execution.
Format: 'True'/'False'
DEBUG_MESSAGES = False
DEBUG_MESSAGES_DETAIL = False
DEBUG_OIDS = True
DEBUG_OTHER_OIDS = False

```python
class IPACTrapHandlerDaemon(Daemon):
    
    Daemon class for the IPAC Trap Handler process
```

```python
def run(self):
    
    Body of IPAC Trap Handler process, called by start() or restart()

    def cbFun(transportDispatcher, transportDomain, transportAddress, wholeMsg):
        
        Processes incoming SNMP Trap Notification
```

```python
while wholeMsg:
    inTime = time.time()
    msgVer = int(api.decodeMessageVersion(wholeMsg))
    if api.protoModules.has_key(msgVer):
        pMod = api.protoModules[msgVer]
    else:
        print 'Unsupported SNMP version %s' % msgVer
        sys.stdout.flush()
        return
    reqMsg, wholeMsg = decoder.decode(wholeMsg,
        asn1Spec=pMod.Message(),
    )
    if DEBUG_MESSAGES or DEBUG_MESSAGES_DETAIL:
        print ('Notification message from %s:%s at %s: ' %
            (transportAddress[0], transportAddress[1], inTime))
        sys.stdout.flush()
    reqPDU = pMod.apiMessage.getPDU(reqMsg)
    if reqPDU.isSameTypeWith(pMod.TrapPDU()):
        if msgVer == api.protoVersion1:
            if DEBUG_MESSAGES_DETAIL:
```

- 191 -
print '  Enterprise: %s' % (pMod.apiTrapPDU.getEnterprise(reqPDU).prettyPrint())
print '  Agent Address: %s' % (pMod.apiTrapPDU.getAgentAddr(reqPDU).prettyPrint())
print '  Generic Trap: %s' % (pMod.apiTrapPDU.getGenericTrap(reqPDU).prettyPrint())
print '  Specific Trap: %s' % (pMod.apiTrapPDU.getSpecificTrap(reqPDU).prettyPrint())
print '  Uptime: %s' % (pMod.apiTrapPDU.getTimeStamp(reqPDU).prettyPrint())
sys.stdout.flush()
varBinds = pMod.apiTrapPDU.getVarBindList(reqPDU)
else:
    varBinds = pMod.apiPDU.getVarBindList(reqPDU)
if DEBUG_MESSAGES_DETAIL:
    print '  Var-binds: '
    print varBinds
    sys.stdout.flush()
for oid, val in varBinds:
    if str(oid) == '1, 3, 6, 1, 4, 1, 9, 9, 215, 1, 1, 8, 1, 2, 0)'
        source = (pMod.apiTrapPDU.getAgentAddr(reqPDU).prettyPrint())
timestamp = pMod.apiTrapPDU.getTimeStamp(reqPDU)
if DEBUG_OIDS:
    print ("MAC Notification from %s at %.2f\n(%.2f):" % (source, float(timestamp)/100,
inTime))
    sys.stdout.flush()
    value = ''
    for byte in val[0][1]:
        value += binascii.b2a_hex(byte)
    op = value[0:2]
    vlan = value[2:6]
    mac = value[6:18]
    port = value[18:22]
    trailer = value[22:24]
    if op == '01':
        if DEBUG_OIDS:
            print ("  Learned %s on port %s, vlan %s" % (mac, port, vlan))
            sys.stdout.flush()
    qString = ("INSERT INTO mac_notifications \n(switchIPv4, macAddress, port, vlan, operation\n, timestamp) VALUES (%s, %s, %s, %s, %s, %i, %f);\n\n" % (source, mac, port, vlan,
int(op), inTime))
    queryDatabase(qString)
elif op == '02':
    if DEBUG_OIDS:
        print ("  Removed %s from port %s, vlan %s" % (mac, port, vlan))
        sys.stdout.flush()
    qString = ("INSERT INTO mac_notifications \n(switchIPv4, macAddress, port, vlan, operation\n, timestamp) VALUES (%s, %s, %s, %s, %s, %i, %f);\n" % (source, mac, port, vlan,
def queryDatabase(queryString):
    """Execute given query string on database."""
    if not TEST_MODE:
        databaseConnection = MySQLdb.connect(host=DATABASE_HOST,
                                               user=DATABASE_USER,
                                               passwd=DATABASE_PASSWD,
                                               db=DATABASE_DBNAME)
        dbCursor = databaseConnection.cursor()
        dbCursor.execute(queryString)
        results = dbCursor.fetchall()
        dbCursor.close()
        databaseConnection.commit()
        return results;
    else:
        print "Starting IPAC Trap Handler"
        sys.stdout.flush()
        if TEST_MODE:
            print "  TEST MODE -- Database will not be modified"
            sys.stdout.flush()
        if FRESH_START:
            cleanupString = """DELETE FROM mac_notifications;""
            queryDatabase(cleanupString)
            print "  Deleted old records from database"
            sys.stdout.flush()
        else:
            print "  Keeping existing records in database"
            sys.stdout.flush()
        transportDispatcher = AsynsockDispatcher()
        transportDispatcher.registerTransport(udp.domainName,
                                               udp.UdpSocketTransport().openServerMode(('0.0.0.0', 162)))
        transportDispatcher.registerRecvCbFun(cbFun)
        transportDispatcher.jobStarted(1)
        print "  Waiting for packets..."
        sys.stdout.flush()
        transportDispatcher.runDispatcher()

if __name__ == "__main__":
    daemon = IPACTrapHandlerDaemon(TRAPHANDLER_PID_FILE, time.time(),
                                          stdout=TRAPHANDLER_LOG_FILE,
                                          stderr=TRAPHANDLER_LOG_FILE)

    if len(sys.argv) == 2:
        if 'start' == sys.argv[1]:
            daemon.start()
        elif 'stop' == sys.argv[1]:
            daemon.stop()
        elif 'status' == sys.argv[1]:
            daemon.status("IPAC Trap Handler")
elif 'restart' == sys.argv[1]:
daemon.restart()
else:
    print "Unknown command"
sys.exit(2)
sys.exit(0)
else:
    print "usage: python %s start|status|stop|restart" % sys.argv[0]
sys.exit(2)

11.4.1.13 ipac_trapsum.py

# IPAC SNMP Trap Summarizer
#
# Jason Froehlich
# Rochester Institute of Technology
# Dept. of Network, Security, and System Administration
# May 17, 2011
#
# This program is a proof of concept being created for the partial
# fulfillment of the MS in Networking and Systems Administration from the
# Rochester Institute of Technology. All rights reserved.
#
# Module Imports
import MySQLdb
import sys
from ipac_server_config import *

__author__ = "Jason Froehlich"
__copyright__ = "Copyright 2011, Jason Froehlich"
__version__ = "0.1.2"
__email__ = "jason.froehlich@alum.rit.edu"
__status__ = "Development"

def getTrapRecords(trapIPv4, username, password, dbName):
    ""
    Obtain Record records from SNMP Trap database
    Once records have been obtained, the Trap records table will be
    cleared to prevent re-processing of old records.
    ""

    __ Connect to trap handler's database
    if DEBUG_TRAP_CONNECTIONS:
        print "Connecting to Trap Handler at", trapIPv4
        sys.stdout.flush()
        trapDbConnection=MySQLdb.connect(host=trapIPv4, user=username,
        passwd=password, db=dbName)

    # Obtain records, remove records to prevent re-processing
    if TRAP_TEST_MODE:
        selectString = """SELECT * FROM mac_notifications \" ORDER BY macAddress,timestamp;""
    else:
        selectString = """SELECT * FROM mac_notifications \" ORDER BY macAddress,timestamp; DELETE FROM mac_notifications""
    selectCursor = trapDbConnection.cursor()
    selectCursor.execute(selectString)
    results = selectCursor.fetchall() # Retrieve records
def queryServerDatabase(queryString):
    """Execute given query string on database."
    ""
    databaseConnection = MySQLdb.connect(host=SERVER_DATABASE_IP4,
                                          user=SERVER_DATABASE_USER,
                                          passwd=SERVER_DATABASE_PASSWORD,
                                          db=SERVER_DATABASE_DBNAME)
    dbc = databaseConnection.cursor()
    try:
        dbc.execute(queryString)
        results = dbc.fetchall()
    except MySQLdb.IntegrityError:
        print "Error executing database query, key error:\n" , queryString
    dbc.close()
    databaseConnection.commit()
    return results;

def clearPortBindings():
    """Delete existing records from Server's portBindings table""
    clearString = """DELETE FROM portBindings;""
    queryServerDatabase(clearString)
    print "Deleted existing port binding records from server database"
    sys.stdout.flush()

def summarizeTraps():
    """Body of SNMP Trap Summarizer""
    ""
    ## Fetch records from agent
    ## Obtain records, remove records to prevent re-processing
    results = getTrapRecords(TRAP_DATABASE_IP4, TRAP_DATABASE_USER,
                              TRAP_DATABASE_PASSWORD, TRAP_DATABASE_DBNAME)
    length = len(results)
    if DEBUG_TRAP_CONNECTIONS:
        print "", length, "port records"
        sys.stdout.flush()
    if not length: # Agent database is empty, skip processing
        #sys.exit()
        return
    for rec in results:
        if DEBUG_TRAP_RECORDS:
            print "Incoming record:", rec
            sys.stdout.flush()
        switch = rec[0]
        mac = rec[1]
        port = rec[2]
Correlating IPv6 Addresses for Network Situational Awareness  
Jason Froehlich

```python
vlan = rec[3]
op = rec[4]
time = rec[5]

if operation == 1:  ## MAC Learned
    queryString = "SELECT * FROM portBindings WHERE macAddress='%s' \ 
                 and switchID='%s' and switchPort='%s' and vlan='%s' ORDER BY \ 
                 startTime DESC;" % (mac, switch, port, vlan)
    existingRecords = queryServerDatabase(queryString)
    if existingRecords and existingRecords[0][6] == 10:
        ## Latest Record still active
        print "MAC not properly removed:", mac, switch, port, vlan
        sys.stdout.flush()
        updateString = "UPDATE portBindings SET endTime=%f, \ 
                       endType=%d WHERE switchID='%s' and macAddress='%s' and \ 
                       switchPort='%s' and vlan='%s' and startTime=%f;" % ( \
                       timestamp, 11, switch, mac, port, vlan, existingRecords[0][4])
        queryServerDatabase(updateString)
        insertString = "INSERT INTO portBindings (switchID, macAddress, \ 
                       switchPort, vlan, startTime, endTime, endType) values \ 
                       (%s,%s,%s,%s,%f,%f,%d);" % (switch, mac, port, vlan, 
                       timestamp, 0.0, 10)
        queryServerDatabase(insertString)

elif operation == 2:  ## MAC Removed
    queryString = "SELECT * FROM portBindings WHERE macAddress='%s' \ 
                 and switchID='%s' and switchPort='%s' and vlan='%s' ORDER BY \ 
                 startTime DESC;" % (mac, switch, port, vlan)
    existingRecords = queryServerDatabase(queryString)
    if existingRecords and existingRecords[0][6] == 10:
        ## Latest record is Add, update
        updateString = "UPDATE portBindings SET endTime=%f, \ 
                       endType=%d WHERE switchID='%s' and macAddress='%s' and \ 
                       switchPort='%s' and vlan='%s' and startTime=%f;" % ( \
                       timestamp, 20, switch, mac, port, vlan, existingRecords[0][4])
        queryServerDatabase(updateString)
        else:  ## Existing Add not found
            insertString = "INSERT INTO portBindings (switchID, \ 
                        macAddress, switchPort, vlan, startTime, endTime, endType) \ 
                        values (%s,%s,%s,%s,%f,%f,%d);" % (switch, mac, port, vlan, 
                        0.0, timestamp, 20)
            queryServerDatabase(insertString)

else:  ## Unknown Operation, Ignore
    pass

if __name__ == "__main__":
    summarizeTraps()
```

11.4.2 Survey Tools

11.4.2.1 ndfuzz.py

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

## IPv6/ICMPv6 Survey Tool
## Jason Froehlich
## Rochester Institute of Technology
## Dept. of Network, Security, and System Administration
```
Correlating IPv6 Addresses for Network Situational Awareness

May 17, 2011

This program is a proof of concept being created for the partial fulfillment of the MS in Networking and Systems Administration from the Rochester Institute of Technology. All rights reserved.

### Module Imports

```python
import binascii
import dpkt
import socket
import sys
```

__author__ = "Jason Froehlich"
__copyright__ = "Copyright 2011, Jason Froehlich"
__version__ = "0.1.2"
__email__ = "jason.froehlich@alum.rit.edu"
__status__ = "Development"

# Source and Destination IPs and MACs
# --Modify to match your system--

SYSTEM_IP6 = "FE80000000000000021731FFFEABCDEF"
SYSTEM_MAC = "AABBCCDDEEFF"
TARGET_IP6 = "20010470C2DD1010426186FFFE123456"
TARGET_MAC = "406186123456"

# Default Packet Header Values
IP6_SRC = SYSTEM_IP6
IP6_DST = TARGET_IP6
IP6_NXT = 58 # Next Header
IP6_HLIM = 255 # Hop Limit
IP6_PLEN = 32 # Payload length
ETH_SRC = SYSTEM_MAC
ETH_DST = TARGET_MAC
ETH_TYPE = 0x86dd

```python
def multicastSoliticNodeIP6(targetIP_a=TARGET_IP6):
    """
    Calculate Solicited Node IPv6 Address
    Generate an Multicast Solicited Node Address for the given Target IPv6 address, will use default Target IPv6 if none given.
    """
    ND_MULTICAST_DST_IP_PRE = "ff0200000000000000000001ff"
    newIP = ND_MULTICAST_DST_IP_PRE + targetIP_a[26:32]
    return newIP

def multicastSolicitNodeMAC(targetIP_a=TARGET_IP6):
    """
    Calculate Solicited Node MAC Address
    Generate a Multicast Solicited Node Address for the given Target MAC address, will use default Target MAC if none given.
    """
    ND_MULTICAST_DST_MAC_PRE = "3333ff"
    newMAC = ND_MULTICAST_DST_MAC_PRE + targetIP_a[26:32]
```
Correlating IPv6 Addresses for Network Situational Awareness

Jason Froehlich

```python
return newMAC

def ndSolicit(ethSrc=SYSTEM_MAC, ethDst=TARGET_MAC, ip6TrafClass=0,
ip6FlowLabel=0, ip6NextHead=58, ip6HopLim=255, ip6Src=SYSTEM_IP6,
ip6Dst=TARGET_IP6, icmp6Type=135, icmp6Code=0, icmp6Check=0,
icmp6Target=TARGET_IP6, icmp6LinkLayerAddr=SYSTEM_MAC):
    
    """
    Generate NDP Solicitation
    """

    Generate a NDP Solicitation packet using given field values, will use
    standards-default value if field value not specified.

    
    ## Convert ASCII Addresses to Hex
    srcIP6_h = binascii.a2b_hex(ip6Src)
srcMAC_h = binascii.a2b_hex(ethSrc)
dstIP6_h = binascii.a2b_hex(ip6Dst)
dstMAC_h = binascii.a2b_hex(ethDst)
icmp6Target_h = binascii.a2b_hex(icmp6Target)

    ## Construct ICMPv6 Datagram
    icmp6 = dpkt.icmp6.ICMP6()
icmp6.type = icmp6Type
icmp6.code = icmp6Code
icmp6.sum = icmp6Check
if icmp6LinkLayerAddr:  ## Link Layer Address Option is enabled, include
    icmp6LLA_h = binascii.a2b_hex(icmp6LinkLayerAddr)
icmp6.data = ("\00\00\00\00" + icmp6Target_h + "\01\01" +
icmp6LLA_h)
else:  ## Link Layer Address Option is disabled, exclude
    icmp6.data = "\00\00\00\00" + icmp6Target_h
print `icmp6`

## Construct IPv6 Packet
ip6 = dpkt.ip6.IP6()
ip6.src = srcIP6_h
ip6.dst = dstIP6_h
ip6.nxt = ip6NextHead
ip6.hlim = ip6HopLim
ip6._set_fc(ip6TrafClass)
ip6._set_flow(ip6FlowLabel)
if icmp6LinkLayerAddr:  ## Link Layer Address Option is included
    ip6.plen = 32
else:
    ip6.plen = 24  ## Link Layer Address Option is excluded
ip6.data = icmp6
print `ip6`

## Construct Ethernet Frame
eth = dpkt.ethernet.Ethernet()
eth.src = srcMAC_h
eth.dst = dstMAC_h
eth.type = 0x86dd
eth.data = ip6
print `eth`

## Establish network socket, inject packet onto network
soc = socket.socket(socket.AF_PACKET, socket.SOCK_RAW)
soc.bind(("eth1", dpkt.ethernet.ETH_TYPE_IP6))
soc.send(str(eth))
def ndAdvertise(ethSrc=SYSTEM_MAC, ethDst=TARGET_MAC, ip6TrafClass=0,
```
Correlating IPv6 Addresses for Network Situational Awareness  

Jason Froehlich

```

``"""  

Generate NDP Advertisement

Generate a NDP Advertisement packet using given field values, will use standards-default value if field value not specified.

"""

## Convert ASCII Addresses to Hex

srcIP6_h = binascii.a2b_hex(ip6Src)
srcMAC_h = binascii.a2b_hex(ethSrc)
dstIP6_h = binascii.a2b_hex(ip6Dst)
dstMAC_h = binascii.a2b_hex(ethDst)
icmp6Target_h = binascii.a2b_hex(icmp6Target)

## Construct ICMPv6 Datagram

icmp6 = dpkt.icmp6.ICMP6()
icmp6.type = icmp6Type
icmp6.code = icmp6Code
icmp6.sum = icmp6Check
if icmp6LinkLayerAddr:  ## Link Layer Address Option is enabled, include
    icmp6LLA_h = binascii.a2b_hex(icmp6LinkLayerAddr)
    icmp6.data = icmp6Flags + icmp6Target_h + "\x02\x01" + icmp6LLA_h
else:  ## Link Layer Address Option is disabled, exclude
    icmp6.data = icmp6Flags + icmp6Target_h
print `icmp6`

## Construct IPv6 Packet

ip6 = dpkt.ip6.IP6()
ip6.src = srcIP6_h
ip6.dst = dstIP6_h
ip6.nxt = ip6NextHead
ip6.hlim = ip6HopLim
ip6._set_fc(ip6TrafClass)
ip6._set_flow(ip6FlowLabel)
if icmp6LinkLayerAddr:  ## Link Layer Address Option is included
    ip6.plen = 32
else:  ## Link Layer Address Option is excluded
    ip6.plen = 24
ip6.data = icmp6
print `ip6`

## Construct Ethernet Frame

eth = dpkt.ethernet.Ethernet()
eth.src = srcMAC_h
eth.dst = dstMAC_h
eth.type = 0x86dd
eth.data = ip6
print `eth`

## Establish network socket, inject packet onto network
soc = socket.socket(socket.AF_PACKET,socket.SOCK_RAW)
soc.bind(("eth1",dpkt.ethernet.ETH_TYPE_IP6))
soc.send(str(eth))
```

if __name__ == "__main__":
    stored = 'stat'  ## initialize last run command placeholder
    while 1:
        """
    ## Prompt user for command
    ```
print "Enter code to send ND packet, or 'ip' or 'mac' to change target"
var = raw_input("Type: ")

## Program Control
if var == 'r': ## Redo last command
    var = stored
if var == 'ip':
    newip = raw_input("New IP: ")
    TARGET_IP6 = newip
elif var == 'mac':
    newmac = raw_input("New MAC: ")
    TARGET_MAC = newmac
elif var == 'stat':
    print "Target:", TARGET_IP6,TARGET_MAC
elif var == 'quit' or var == 'exit':
sys.exit()

#### Create NDP Solicitation #####

## Valid NDP Solicitations
elif var == '10': ## Correct Unicast Solicit
    print "Sending normal ND Solicit via Unicast"
    ndSolicit()
elif var == '11': ## Correct Multicast Solicit
    print "Sending normal ND Solicit via Multicast"
    ndSolicit(ethDst=multicastSolicitNodeMAC(),
              ip6Dst=multicastSolicitNodeIP6())

## Invalid NDP Solicitations - ICMPv6 Options Misconfigured
elif var == '20': ## ICMP6 Code not 0
    ndSolicit(icmp6Code=255)
elif var == '21': ## ICMP6 Incorrect Checksum
    ndSolicit(icmp6Check=0x325235)
elif var == '22': ## ICMP6 Source Link Layer Address Missing
    ndSolicit(icmp6LinkLayerAddr=0)
elif var == '23': ## ICMP6 Source Link Layer Address Wrong
    ndSolicit(icmp6LinkLayerAddr="FFEEDDCCBBAA")

## Invalid NDP Solicitations - IPv6 Options Misconfigured
elif var == '30': ## IP6 Multicast to all nodes
    ndSolicit(ip6Dst="ff0200000000000000000001",
              ethDst="333300000001")
elif var == '31': ## IP6 Multicast to incorrect IP6+MAC Solicited node
    ndSolicit(ip6Dst="ff0200000000000000000001ff123456",
              ethDst="3333ff123456")
elif var == '35': ## IP6 Multicast to incorrect IP6 Solicited node
    ndSolicit(ip6Dst="ff0200000000000000000001ff123456",
              ethDst=multicastSolicitNodeMAC())
elif var == '32': ## IP6 Hop Limit not 255
    ndSolicit(ip6HopLim=254)
elif var == '33': ## IP6 Flow Label not 0
    ndSolicit(ip6FlowLabel=255)
elif var == '34': ## IP6 Traffic Class not 0
    ndSolicit(ip6TrafClass=255)

## Invalid NDP Solicitations - Ethernet Options Misconfigured
elif var == '40': ## Eth Dest to Broadcast
    ndSolicit(ethDst="FFFFFFFFFFFF")
elif var == '41': ## Eth Multicast to incorrect MAC Solicited node
    ndSolicit(ethDst="3333ff123456", ip6Dst=multicastSolicitNodeIP6())
### Create NDP Advertisement ###

#### Valid NDP Advertisement ####

```python
## Correct Unicast Advert
def ndAdvertise():
    ## ICMP6 Code not 0
    ndAdvertise(icmp6Code=255)
## ICMP6 Incorrect Checksum
    ndAdvertise(icmp6Check=0x325235)
## ICMP6 Source Link Layer Address Missing
    ndAdvertise(icmp6LinkLayerAddr=0)
## ICMP6 Source Link Layer Address Wrong
    ndAdvertise(icmp6LinkLayerAddr="FFEEDDCBBAA")
## ICMP6 Flag - Router
    ndAdvertise(icmp6Flags="\x80\x00\x00\x00")
## ICMP6 Flag - Not Solicited
    ndAdvertise(icmp6Flags="\x00\x00\x00\x00")
## ICMP6 Flag - Override
    ndAdvertise(icmp6Flags="\x20\x00\x00\x00")
## ICMP6 Flag - Override and Solicited
    ndAdvertise(icmp6Flags="\x60\x00\x00\x00")
## IP6 Multicast to all nodes
    ndAdvertise(ip6Dst="FF020000000000000000000000000001",ethDst="333300000001")
## IP6 Multicast to IP6 + MAC Solicited node
    ndAdvertise(ip6Dst=multicastSoliticNodeIP6(),ethDst=multicastSolicitNodeMAC())
## IP6 Hop Limit not 255
    ndAdvertise(ip6HopLim=254)
## IP6 Flow Label not 0
    ndAdvertise(ip6FlowLabel=255)
## IP6 Traffic Class not 0
    ndAdvertise(ip6TrafClass=255)
## IP6 Source doesn't match ICMPv6 Target
    ndAdvertise(ip6Src="FE80000000000000CCCCDDDDEEEEFFFF")
## 'Normal'
    ndAdvertise(ip6Dst="FF020000000000000000000000000001",ethDst="333300000001",icmp6Flags="\x20\x00\x00\x00")
## Override flag not set
    ndAdvertise(ip6Dst="FF020000000000000000000000000001",ethDst="333300000001",icmp6Flags="\x00\x00\x00\x00")
## Destination is Solicited-Node Multicast address
    ndAdvertise(ethDst=multicastSolicitNodeMAC(),ip6Dst=multicastSoliticNodeIP6(),icmp6Flags="\x20\x00\x00\x00")
## Destination is Unicast Address
    ndAdvertise(icmp6Flags="\x20\x00\x00\x00")
## 'Normal'
    ndAdvertise(ip6Dst="FF020000000000000000000000000001",ethDst="333300000001",icmp6Flags="\x60\x00\x00\x00")
```

#### Invalid NDP Advertisement - ICMPv6 Options Misconfigured ####

```python
## IP6 Multicast to all nodes
    ndAdvertise(ip6Dst="FF020000000000000000000000000001",ethDst="333300000001")
## IP6 Multicast to IP6 + MAC Solicited node
    ndAdvertise(ip6Dst=multicastSoliticNodeIP6(),ethDst=multicastSolicitNodeMAC())
## IP6 Hop Limit not 255
    ndAdvertise(ip6HopLim=254)
## IP6 Flow Label not 0
    ndAdvertise(ip6FlowLabel=255)
## IP6 Traffic Class not 0
    ndAdvertise(ip6TrafClass=255)
## IP6 Source doesn't match ICMPv6 Target
    ndAdvertise(ip6Src="FE80000000000000CCCCDDDDEEEEFFFF")
## 'Normal'
    ndAdvertise(ip6Dst="FF020000000000000000000000000001",ethDst="333300000001",icmp6Flags="\x20\x00\x00\x00")
## Override flag not set
    ndAdvertise(ip6Dst="FF020000000000000000000000000001",ethDst="333300000001",icmp6Flags="\x00\x00\x00\x00")
## Destination is Solicited-Node Multicast address
    ndAdvertise(ethDst=multicastSolicitNodeMAC(),ip6Dst=multicastSoliticNodeIP6(),icmp6Flags="\x20\x00\x00\x00")
## Destination is Unicast Address
    ndAdvertise(icmp6Flags="\x20\x00\x00\x00")
## 'Normal'
    ndAdvertise(ip6Dst="FF020000000000000000000000000001",ethDst="333300000001",icmp6Flags="\x60\x00\x00\x00")
```

#### Invalid NDP Advertisement - IPv6 Options Misconfigured ####

```python
## IP6 Hop Limit not 255
    ndAdvertise(ip6HopLim=254)
## IP6 Flow Label not 0
    ndAdvertise(ip6FlowLabel=255)
## IP6 Traffic Class not 0
    ndAdvertise(ip6TrafClass=255)
## IP6 Source doesn't match ICMPv6 Target
    ndAdvertise(ip6Src="FE80000000000000CCCCDDDDEEEEFFFF")
## 'Normal'
    ndAdvertise(ip6Dst="FF020000000000000000000000000001",ethDst="333300000001",icmp6Flags="\x20\x00\x00\x00")
## Override flag not set
    ndAdvertise(ip6Dst="FF020000000000000000000000000001",ethDst="333300000001",icmp6Flags="\x00\x00\x00\x00")
## Destination is Solicited-Node Multicast address
    ndAdvertise(ethDst=multicastSolicitNodeMAC(),ip6Dst=multicastSoliticNodeIP6(),icmp6Flags="\x20\x00\x00\x00")
## Destination is Unicast Address
    ndAdvertise(icmp6Flags="\x20\x00\x00\x00")
## 'Normal'
    ndAdvertise(ip6Dst="FF020000000000000000000000000001",ethDst="333300000001",icmp6Flags="\x60\x00\x00\x00")
```

## Unsolicited NDP Advertisement

```python
elif var == '140':
    ## 'Normal'
    ndAdvertise(ip6Dst="FF020000000000000000000000000001",ethDst="333300000001",icmp6Flags="\x20\x00\x00\x00")
elif var == '141':
    ## Override flag not set
    ndAdvertise(ip6Dst="FF020000000000000000000000000001",ethDst="333300000001",icmp6Flags="\x00\x00\x00\x00")
elif var == '142':
    ## Destination is Solicited-Node Multicast address
    ndAdvertise(ethDst=multicastSolicitNodeMAC(),ip6Dst=multicastSoliticNodeIP6(),icmp6Flags="\x20\x00\x00\x00")
elif var == '143':
    ## Destination is Unicast Address
    ndAdvertise(icmp6Flags="\x20\x00\x00\x00")
elif var == '144':
    ## 'Normal'
    ndAdvertise(ip6Dst="FF020000000000000000000000000001",ethDst="333300000001",icmp6Flags="\x60\x00\x00\x00")
```

## Unrecognized Command, Ignore

```python
else:
    ```
pass

stored = var  ## Store last command
print "\n"
11.5 Bibliography


Correlating IPv6 Addresses for Network Situational Awareness

Jason Froehlich


Correlating IPv6 Addresses for Network Situational Awareness


