Design and Implementation of A Cooperative, Autonomous Anti-DDoS Network Using Intruder Detection and Isolation Protocol

Sarah Jelinek
University of Colorado, Colorado Springs
1420 Austin Bluffs Pkwy.
Colorado Springs, CO USA 80918
sjjelinek@gmail.com

This paper presents a design and implementation of an cooperative, enterprise wide, autonomous distributed denial of service(DDoS) defense infrastructure using well known DoS technologies along with a partial implementation of a protocol developed by Boeing Corporation called the Intruder Detection and Isolation Protocol(IDIP). This work is a follow on to the original autonomous distributed denial of service network(A2D2) developed by Angela Cearns, University of Colorado, Colorado Springs[C02]. As a result of the original project name, this project has been dubbed A2D2V2 and will be referred to by this name for the remainder of this paper.

This paper presents a discussion of DoS, DDoS and DDoS defense mechanisms along with a brief overview of A2D2. An overview of the IDIP Architecture and guiding principles along with other emerging technologies will be presented.

For this project a subset of the IDIP implementation was developed and tested in the A2D2V2 test bed. Along with this, a technique for discovering attacks by cooperating upstream nodes in the IDIP community was developed. A set of tests were developed to show the benefits of the cooperative intrusion defense.

There were several goals for A2D2V2:

1. To validate the enterprise effectiveness of the software implementation utilizing IDIP with regard to attack response.
2. Show that IDIP can provide a cooperative defense that efficiently notifies upstream routers of an attack.
3. To expand on A2D2 ideas to provide a cooperative defense against DDoS attacks.
4. To try to provide sustained performance for both clients in the A2D2V2 enterprise network with the full attack mitigation activated.
5. To show that A2D2V2 via IDIP provides a cooperative defense that efficiently notifies upstream routers of an attack, which enables the containment the attack in a short period of time.

The test results for A2D2V2 show that these goals were met and further show a clear benefit from using a protocol to communicate and coordinate with other nodes in a network to push back DDoS attacks.

TABLE OF CONTENTS

INTRODUCTION

The threat of Distributed Denial of Service (DDoS) attacks on Internet systems has not diminished. These attacks are insidious and difficult to handle. Research in to handling them, as well as tracing their inception focuses on both the end system and the infrastructure. The difficulty lies in distinguishing between an attack and a legitimate large number of attempted connections over a short period of time. The effect of these two is of course the same, but the response will most likely be different.

In general, a policy is set and implemented to respond to the DDoS attacks. This policy cannot always be enforced by the systems being attacked. Some of it requires human intervention. However, host and infrastructure systems should be configured in a way as to implement the parts of the response policy that can be enforced by technical means.

While the architecture of a local, private network, can indeed mitigate and sometimes even stop the DDoS attack, this solution is isolated and does not provide others with help against the same attack. We need to add the capability to push back the intrusion so that legitimate clients can continue to receive service.

This report is laid out as follows:

- Section 1 presents an overview of DoS attacks.
- Section 2 presents an overview of DDoS attacks.
- Section 3 discusses possible defenses against DDoS attacks.
- Section 4 gives an overview of IDIP and how it is intended to work
- Section 5 discusses the Cooperative Intrusion and Traceback Architecture known as CITRA
- Section 6 presents the basic IDIP Software Architecture
- Section 7 discusses this projects precursor, A2D2
- Section 8 presents the A2D2V2 architecture and implementation details
- Section 9 presents the test configuration and detailed analysis of the performance results gathered
- Section 10 discusses lessons learned during the course of this project
- Section 11 presents ideas for future work in this area
- Section 12 presents the authors final conclusions regarding this project and cooperative DDoS attack response
- Following Section 12 are the bibliography and appendices

1. DENIAL OF SERVICE ATTACKS(DOS)

There are many here are many manifestations of Denial of Service attacks but they ultimately have the same objective - to deny or degrade a user's ability to legitimately access network or host based services. DoS attacks accomplish this by exhausting the limited resources of network bandwidth by packet flooding or exhausting host resources by consumption of CPU cycles, random memory, static memory or data structures [T02].

DoS attacks can generally be classified as either a Flood Attack or a malformed (or crafted) Packet Attack. Attacks originate simultaneously from several compromised sources are classified as Distributed DoS attacks(DDoS).

Fundamental to the IP protocol every packet has a source and destination address field that is used to determine the originating and destination end points. The process of forwarding these packets by intermediate routers partly relies on the destination field; the source address will only be used when a response to the packet is required. This makes the implementation of DDoS flooding attacks easy to accomplish because fake or “spoofed” source addresses can be used, and packets will generally be forwarded unchallenged to the specified destination. This allows a DoS or DDoS attack to be carried out from any location and with total anonymity.

If an attack is underway from a single address then it is possible to arrange for a “block” of the offending source IP address at the ISP or the border router. However, when a DDoS attack occurs the problem is not as easy to resolve because packets appear to be coming from hundreds or even thousands of different hosts, there is absolutely no point trying to implement temporary Access Control Lists on routing devices or modify the border firewall rulebase, it is too late - you are left at the mercy of the attack under way. The types of attack can also take the form of a single “one shot” crafted packet originating from a single host to thousands of packets per second originating simultaneously from multiple hosts.

2. DISTRIBUTED DENIAL OF SERVICE ATTACKS

There are many types of DDoS attacks. However they all have the same signature. A DDoS is a Denial of Service (DoS) attack in which many unwitting participants have been unknowingly recruited to initiate attacks. The end goal is generally to disrupt the victims systems such that they are no longer able to provide the service expected.

The normal DDoS attack architecture works upon the basis that the required hosts to launch the attack from have already been identified and compromised via Trojans or “backdoors”[B02]. In a DDoS scenario the Intruder (also called the Attacker or Client) issues control traffic to the Master (also called the Handler) which, in turn then issues commands to the daemon (also called an Agent, Broadcast program or Zombie). The daemons that are at the end of this command chain finally initiate the attack traffic against the Victim. This distributed architecture increases the attack capability many times over and allows the Intruder the means to remain undetected as shown in figure 11 below.

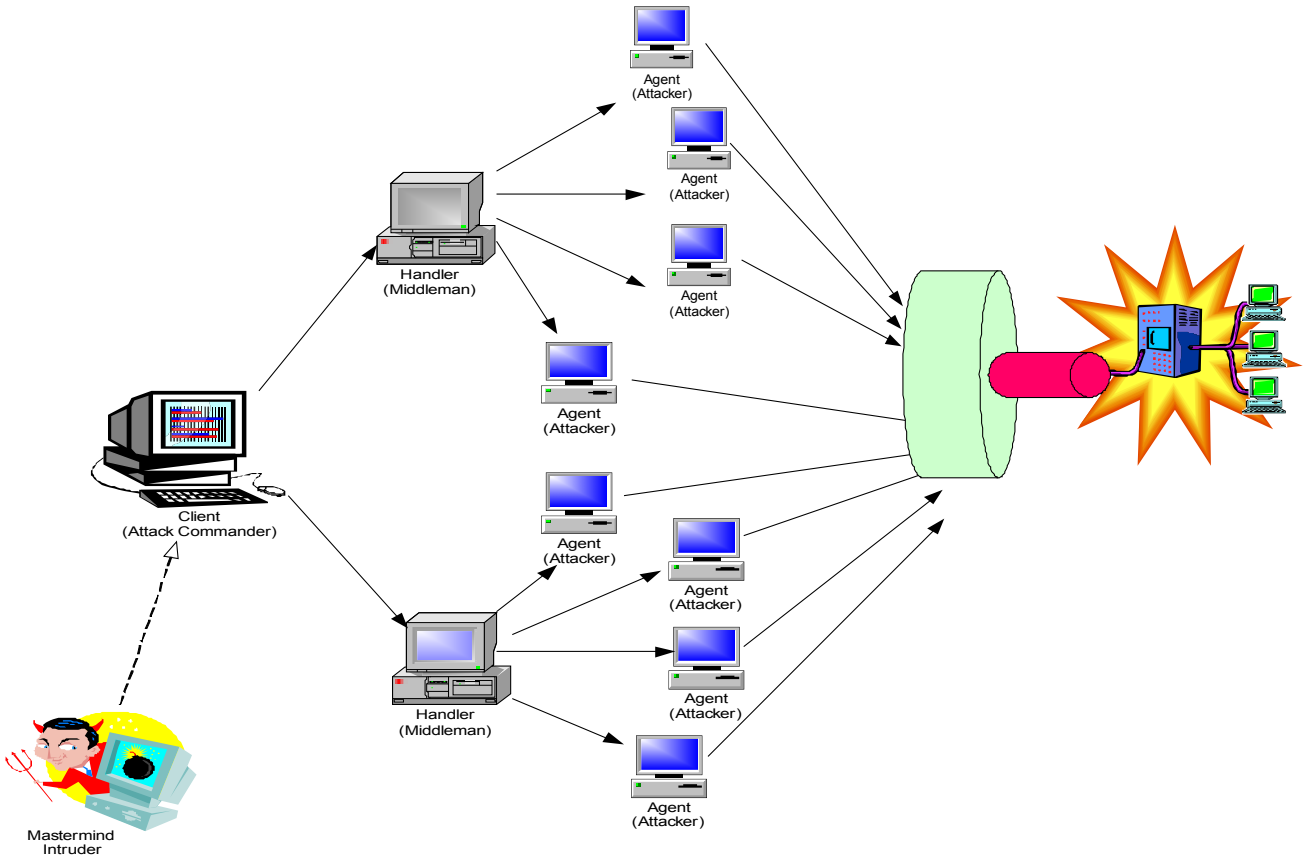


Figure 1.1 Typical DDoS Architecture[C03]

The distributed nature of these attacks makes it extremely difficult to trace and stop these kinds of attacks. While one attacker will most likely not be able to generate enough traffic to effectively shut down a large commercial site, the cooperative power of the diverse set of attack agents can easily make any network inoperable [C02].

3. DEFENSE AGAINST DDOS ATTACKS

In general, DDoS defense is broken down in to three areas: 1) Intrusion Detection 2) Intrusion Prevention and 3) Intrusion Response.

3.1 INTRUSION DETECTION

“Intrusion Detection (ID) is like chess, or a game of network cat-and-mouse. ID software to date commonly analyzes the actions of an attacker in more or less linear terms:

"this stream of packets matches a stream known to be a smurf, SYN, or other known *attack signatures.*" Signature-based ID systems are adequate to deal with misuse intrusions, but can't deal with out-of-the-box thinkers who pen-test, audit, or attack networks, purposely thinking non-linearly with the expectation of ultimately discovering code, policy, and logic flaws. They also can't adequately deal with *anomalous* behaviors and resulting intrusions, e.g., the disgruntled insider who abuses authorized access, the unwitting user who is victimized by a worm, the server that is back-doored." [T02]

The best defense against DDoS attacks is to prevent initial system compromises. Generally, this involves installing patches, anti virus software, using a firewall and monitoring for intruders. However, even vigilant hosts can become targets because of lesser prepared, less security aware hosts (especially if these hosts have always-on high-speed internet connections). Many systems are compromised because patches for vulnerabilities reported and fixed months beforehand were never installed. Similarly, such systems have anti-virus software that is not up to date.

3.2 INTRUSION PREVENTION

Intrusion Prevention goes beyond detection. The ultimate aim of Intrusion Prevention is to neutralize an attack before it reaches the firewall.

It is difficult to specifically defend against becoming the ultimate target of a DDoS attack but protection against being used as a daemon or master system is more easily attainable. To this end, the following measures should be met (Gary Flynn, 2000):

- Check for frequent patches and subscribe to automatic vendor notifications
- Attempt to understand the vulnerabilities in your software and configuration
- Disable unnecessary network software
- Only accept program files from trusted sources (or at least be cautious)

Another prevention technique is vulnerability or penetration testing. This is the act of determining which security holes and vulnerabilities may be applicable to the target network or hosts. The penetration tester or attacker will attempt to identify machines within the target network of all open port and the operating systems as well as running applications including the operating system, patch level, and service pack applied.

The vulnerability testing phase is started after some interesting hosts are identified via the nmap scans or another scanning tool and is preceded by the reconnaissance phase. Nmap will identify if a host is alive or not and what ports and services are available even if ICMP is completely disabled on the target network to a high degree of accuracy.

3.3 INTRUSION RESPONSE

Once an attack has been determined to be in progress, the immediate response is to identify the source of the attack and block traffic from that source. However, as noted in section 1.2, in a DDoS attack it is normally quite difficult to determine the true source of the attack. As a result of this, most Intrusion Response systems do whatever is necessary

to mitigate the affect that the attack has on the system resources, and generally do not go farther to trace the source, or notify others of potential attacks.

Generally the Intrusion Response system must provide a policy by which the affected systems can tolerate the attack. This is usually a traffic blocker or traffic rate limiting approach. In this way however, legitimate clients will have their service degraded as well since it is very difficult to know the true source.

4. IDIP PROTOCOL – A TECHNICAL PRIMER

IDIP was initially developed as part of DARPA's Dynamic, Cooperating Boundary Controllers program. It was later extended through the Automated Response to Intrusion, Adaptive System Security Policies, and Multi-Community Cyber Defense Contracts. This work was done in conjunction with Boeing, Network Associates(NAI labs) and the University of California, Davis.

Although IDIP was originally intended to be a published, standard protocol, the most recent efforts for this have been adopted by NAI labs, McAfee and Telcordia and is currently not available for public viewing. The architectural documents used for this project during implementation of the protocol were those found as part of the research done. The date of these documents is February, 2002.

IDIP was developed to support real-time tracking and containment of attacks that cross network boundaries. IDIP was developed to provide responses in two stages: (1) an initial immediate response that may be relatively harsh but is relatively short-lived, and (2) a more reasoned, optimal response that is more effective at meeting the system's overall operational needs while attempting to contain the attack.

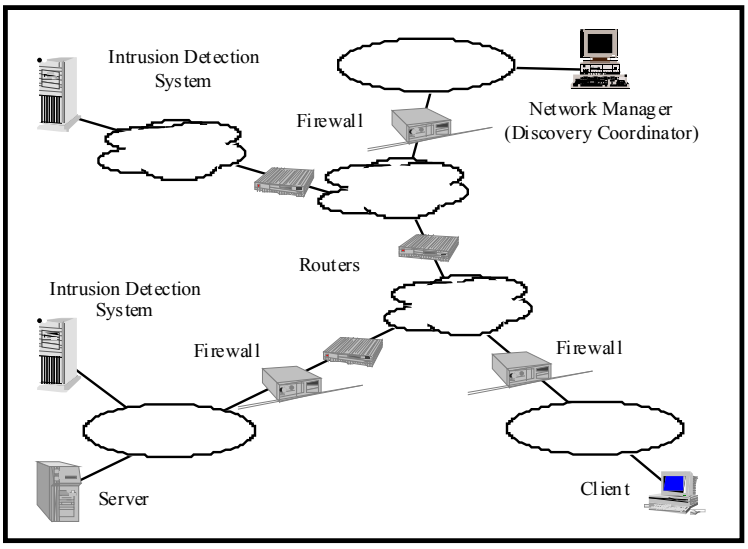


Figure 4.1 IDIP Nodes[NB02]

Figure 4.1 shows the various components that can participate in an IDIP-based response. Intrusion Detection components initiate IDIP response messages, and can

support damage assessment and recovery within the local environment. Boundary controllers provide network based response mechanisms by blocking the intruder's access to network resources. A centralized network management component, call the Discovery Coordinator, receives intrusion reports and audit data from other IDIP nodes, enabling it to 1) provide administrative personnel with a global picture of the system intrusion status and 2) coordinate the overall system response to attacks.

4.1 IDIP ARCHITECTURE

The IDIP Architecture was developed with the following principles in mind:

- An IDIP system must be able to respond to intrusions in real-time
- An IDIP system must support environments that span multiple administrative domains
- An IDIP system must have minimal impact on the systems performance
- An IDIP system must be capable of operating while the system is under attack
- The IDIP system components must be capable of responding autonomously to the attack

Figure 4.1.1 shows the IDIP enterprise architecture. As part of this architecture several new terms are important to note and understand.

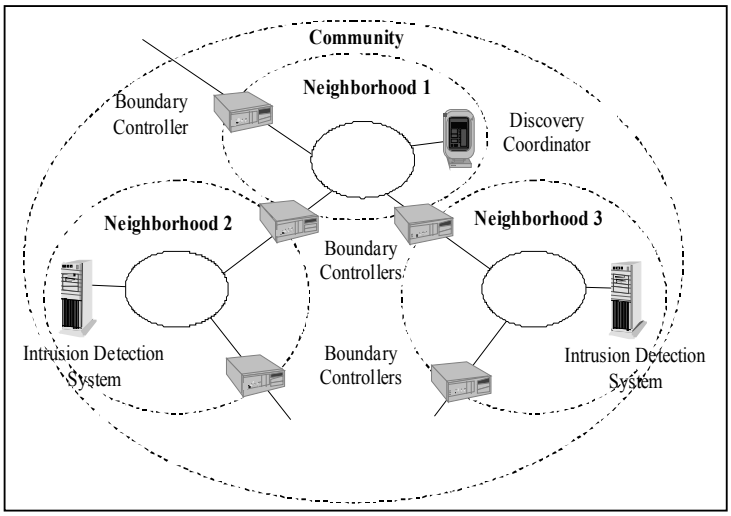


Figure 4.1.1 IDIP Enterprise Architecture

4.1.1 IDIP NEIGHBORHOODS

Each IDIP neighborhood is an administrative domain, with intrusion detection and response functions managed by a component called the Discovery Coordinator. Each administrative domain is capable of detection and response as it sees fit, without knowledge of or communication with other IDIP neighborhoods. These neighborhoods are the collection of components with no other IDIP node between them.

4.1.2 IDIP Communities

Neighborhoods are further organized in to IDIP communities. Each distinct neighborhood in a community are connected to other neighborhoods via boundary control devices. Boundary control devices are members of multiple IDIP neighborhoods.

The design of the IDIP neighborhood and community response allows for each neighborhood to respond autonomously to an attack. This is an important feature of the IDIP architecture.

4.2. IDIP PROTOCOL DEFINITIONS

IDIP is actually made up of several protocol definitions. The combination of the deployment of these protocols comprises a complete IDIP system. There are two distinct layers in IDIP and as such, the protocol definitions are separate across these layers. The two layers in IDIP are the Application Layer and the Message Layer.

4.2.1 IDIP Message Layer

The Message Layer is the lower layer, and acts, in part, as the transport layer for IDIP. The message layer consists of the following protocols:

1. HELLO protocol for neighborhood management[NB02-2]
2. Neighborhood Key information distribution protocol(NKID).[NB02-2]
3. IDIP authentication header-[NB02-2]
4. IDIP encapsulating security payload-[NB02-2]

The Message Layer is designed to provide secure, reliable messaging for IDIP applications between neighbors in an IDIP neighborhood as well as between IDIP communities. It is also designed to provide privacy and integrity/authentication. It uses the IDIP Authentication Header(AH) and Encapsulating Security Payload(ESP) to achieve this. (The IDIP AH and IDIP ESP protocol definitions are not available for public viewing at this time.).

The Discovery Coordinator functionality can be embedded within the message layer, or as a stand-alone entity. Most generally it is embedded as part of the message layer application.

One other major guiding principle in the IDIP Message Layer design is to have minimal performance impact on the protected systems. The IDIP Message Layer adds very little overhead for each message. The use of multicast, in networks where this is supported, reduces the message traffic for IDIP messages. The use of UDP minimizes the consumption of local host resources. Since there are potentially many neighbors in an IDIP neighborhood using TCP could potentially consume many of the network resources required for an application.

4.2.1.1 IDIP Hello Protocol

The IDIP HELLO Protocol is responsible for requests for inclusion as an IDIP neighbor in an IDIP neighborhood. Nodes that wish to register as a neighbor send a

HELLO message with the appropriate data included. The HELLO Protocol then determines if the request is valid and updates the neighborhood list appropriately. The HELLO Protocol maintains the master list of the IDIP neighbors and is the initialization point for the IDIP protocol.

4.2.2 IDIP Application Layer

IDIP's objective is to share the information necessary to enable intrusion tracking and containment. The Application Layer defines the messages and procedures used by IDIP applications to support intrusion isolation and containment. These messages are passed to neighboring IDIP devices to trace the patch of the intrusion, and provide the information necessary for each device along the path to determine the appropriate response.

A fundamental guiding principle in the design of IDIP was to minimize the size and number of messages required to support intrusion response. Application Layer messages are primarily sent only after an intrusion has been detected. Once the response has been initiated the Application Layer attempts to only send messages to components that could have been affected by part of the attack.

There are several types of IDIP Application devices. They are generally intrusion detection systems(IDS), firewalls and routers. IDIP Applications generally send an IDIP Trace message when it has determined that the IDIP node it resides on, or set of nodes it can see, are under attack. This IDIP node at the detector of the potential attack specifies which type of response is needed. Each IDIP node that gets the trace message can decide on whether or not to follow the suggested response. An IDIP node can also choose to take some other node-specific action based on local policy.

4.3 HOW IDIP MEETS THE KEY PRINCIPLES

As noted in Section 4.1 there are several key principles employed with the architecture of an IDIP system.

4.3.1 An IDIP system must be able to respond to intrusions in real-time

The potential for a large amount of data to be generated by the tracing and analyzing of network data by an IDIP node is great. This volume of data must somehow be disseminated quickly and a response formed appropriately, in real-time, for IDIP to meet this key principle. There is nothing in the IDIP protocols as they are defined today, that necessarily supports this principle automatically. However, the Discovery Coordinator capability could easily be modified to include a knowledge engine that could do this work. One of the key design principles for IDIP was the ability to plug in additional components to the nodes in an IDIP system. As long as the component communicates using the defined message format, the inclusion of such is supported.

4.3.2 An IDIP system must support environments that span multiple administrative domains

As noted in section 4.1, and IDIP neighborhood is a single administrative domain. The ability to span this administrative domain is supported via the inclusion of Boundary

Controllers as IDIP nodes. These Boundary Controllers enable the communication across IDIP neighborhoods and facilitate the ability to stop an attack on a more global scope.

An important thing to note with this design is that the trust from one neighborhood to another is important to establish and not automatic. The use of the Key Management protocol as defined for Section 4.2.1 is critical to establishing and maintaining this trust.

The distributed nature of the IDIP architecture deployment ensures that each IDIP Neighborhood can respond as it chooses and completely autonomously of any other IDIP Neighborhoods.

4.3.3 An IDIP system must have minimal impact on the systems performance

As noted in Section 4.2.1, the IDIP Architecture definition specifies that the network transport that must be used in sending and receiving IDIP messages is the User Datagram Protocol(UDP). UDP is a connection-less protocol that runs on top of IP networks. It provides very little in the way of error recovery services. It provides a direct way to send and receive datagram messages over the network.

The UDP protocol provides a procedure for applications to send messages to other applications with a minimum of overhead. UDP is transaction oriented, and as such delivery of messages and duplicate message protection are not guaranteed. However, UDP places little additional overhead on any system with applications using this as the transport mechanism.

4.3.4 An IDIP system must be capable of operating while the system is under attack

The IDIP Message Layer was designed to be simple and to minimize the likelihood that it would fail in the event of an attack. As a result, the IDIP Message Layer has minimal dependence on the network infrastructure and uses the User Datagram Protocol(UDP) rather than the Transmission Control Protocol(TCP) to achieve this. It also uses IP addresses in application-layer node identification fields to minimize the dependence on DNS. Both TCP and DNS are vulnerable to attack. This design decision forces the IDIP Message Layer to handle all of the acknowledgment and verification when sending IDIP messages. This puts some additional overhead on this layer, but ultimately provides survivability in the event of an attack. As a result of the use of UDP, the IDIP Message Layer has to incorporate the delivery acknowledgment and duplicate message functionality in to its functional area. This decision could potentially place a larger load on a system hosting an IDIP node and must be taken in to consideration when implementing the response capability.

4.3.5 The IDIP system components must be capable of responding autonomously to the attack

As shown in figure 4.1.1, each IDIP neighborhood is an administrative domain. Each IDIP node within a neighborhood must have the ability to respond autonomously, regard-

less of the status of other the other IDIP nodes. Each IDIP Community must be able to respond even if another IDIP Neighborhood within the community is compromised in some way and cannot respond appropriately.

The Discovery Coordinator represents a single point of failure in the IDIP system, making it a target for DoS attack. If the Discovery Coordinator is not available for directing an optimal response, IDIP nodes can take increasingly severe responses when attacks continue following the initial response, reducing the reliance of IDIP on Discovery Coordinator actions. This is fundamental to and IDIP system continuing its response in the even that a node or set of nodes have been compromised.

5. COOPERATIVE INTRUSION DETECTION AND TRACEBACK ARCHITECTURE (CITRA), IDIP'S GLOBAL RESPONSE ARCHITECTURE

CITRA is a framework for integration of intrusion detection systems, firewalls, routers, security management systems and other components in to an IDIP System. This framework enables tracing intrusions across network boundaries, preventing or mitigating subsequent damage from attacks, consolidating and reporting intrusion activities and coordination of intrusion responses on a system-wide basis.

Figure 5.1 illustrates how IDIP nodes can cooperate to generate a global response. The CITRA concept of operations has each response component(IDIP node) independently deciding on what is an appropriate response. The system's objective is to generate the response as close to possible to the attacker, minimizing the response impact on the critical functions of the system under attack. Each component of CITRA has an objective to allow this optimal response while protecting local resources as well.

CITRA was designed to facilitate low-cost integration of independently developed components. It was also designed for flexible adaptation of these components capabilities. IDIP is integral to the CITRA framework. IDIP defines the format of and information specification that CITRA-enabled components may exchange.

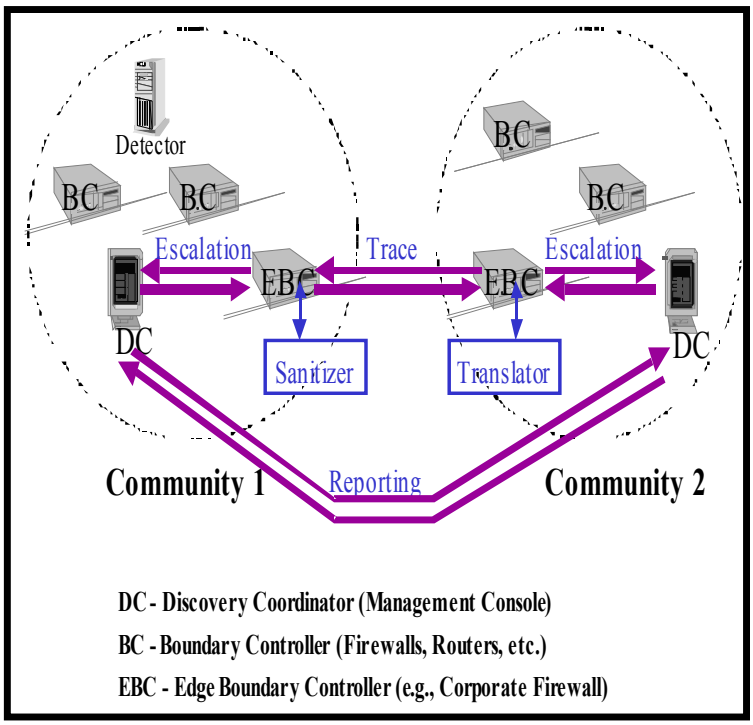


Figure 5.1 IDIP Global Response Architecture[NB02]

CITRA was extended under the Multi-Community Cyber Defense (MCCD) contract from DARPA to allow traceback and response to attacks to continue across multiple IDIP communities. Previously traceback and response would end when the last IDIP node in the community along the path of the attack was reached. With the extensions made to CITRA under MCCD attacks could now be traced into other IDIP communities and those communities could be requested to respond to attack. For MCCD, there are two terms that require special definition:

- Remote Neighborhood – A Remote Neighborhood is a collection of adjacent Edge Boundary Controllers. (I.e., two IDIP nodes are neighbors if they have no IDIP nodes between them).
- Edge Boundary Controller (EBC) – An Edge Boundary Controller is an IDIP boundary controller with one or more neighbors belonging to a differing IDIP community from itself.

5.1.2 COMMUNICATION BETWEEN IDIP COMMUNITIES

Communication between IDIP communities is handled through the edge boundary controllers. Trace requests are handled by the IDIP agents on the nodes. Messages are passed from one DC to the other DC through both EBCs using normal IDIP message layer protocols.

5.1.3 MULTI-COMMUNITY POLICIES

Inter-Community policies are established for the edge boundary controllers on how they will handle requests from other edge boundary controllers in their remote neighborhood. This policy determines if the request is continued, ignored or requires a human in the loop to authorize the request (Escalation). The policy also dictates if outgoing trace messages should be sanitized and if so which fields require sanitization. The policy specifies if incoming trace messages should be translated and if so which fields require translation. The policy also includes whether intrusion alert warnings or correlation events will be sent to another community.

5.1.4 CITRA REMOTE NEIGHBORHOOD TRUSTWORTHINESS AND LOCATION

As seen in Figure 5.1, there must be trust established between remote neighborhoods. This trust is developed using the key management principles and authentication mechanisms described in the IDIP Message Layer Protocol.

How does one remote neighborhood determine where another remote neighborhood resides? How does a remote neighborhood determine if another node along the network path is an IDIP Boundary Controller? These questions are not answered specifically in the CITRA specification. This project does list some possible technologies that could be used in the aide of answering these questions in Section 11.

6. IDIP SOFTWARE ARCHITECTURE

The IDIP software architecture has two primary objectives: 1) Ease of integration with various components and 2) flexibility in modifying the generic component behaviors for specific components. This concept supports the integration of many types of IDIP nodes, boundary controllers, network and host based IDS systems, clients, servers and network management components.

The Protocols were designed for portability and the intent was to make them platform independent. However, there are places where platform differences, particularly in the area of network interfaces, must be accounted for.

The IDIP software components comprise the IDIP Backplane and IDIP Applications. Figure 6.1 shows the software architecture intended for an IDIP protocol implementation. The IDIP Software was designed for portability.

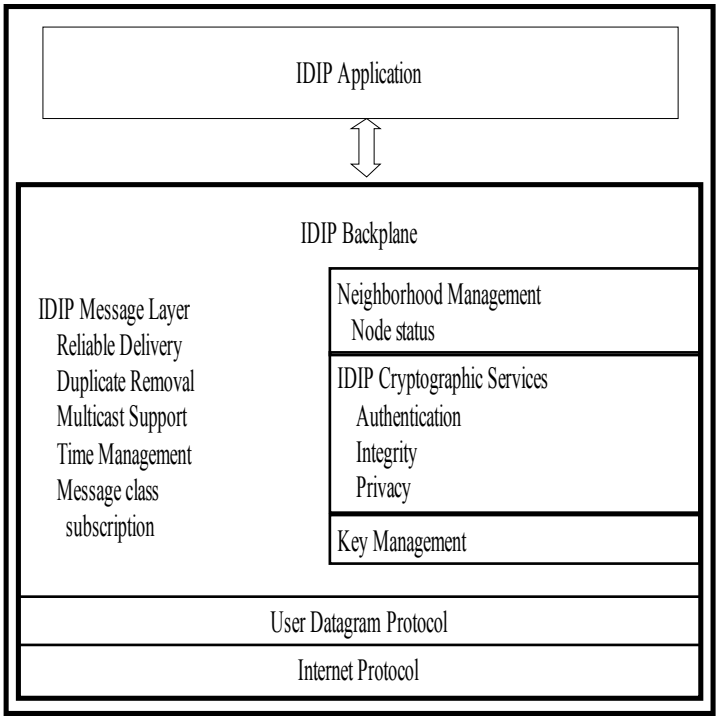


Figure 6.1 IDIP Software Architecture

7. A2D2

The Autonomous Anti-DDoS Network (A2D2) was designed and implemented by Angela Cearns as part of her University of Colorado, Colorado Springs Masters Thesis[C02]. The goal of the design of A2D2 was to combine various technologies and make necessary improvements to achieve autonomous attack mitigation similar to that attained by elaborate expensive architectures. The A2D2 network is specifically designed to enhance quality of service during bandwidth consumption DDoS attack. The A2D2 design follows four main guiding principles [A02]:

- Affordable
- Manageable
- Configurable
- Portable

To achieve these goals several well known technologies were used that make up the A2D2 system:

- Open Source Snort Intrusion Detection software
- Linux *iptables(8)* mechanism
- Linux Class based queuing mechanism

This paper will briefly discuss these technologies and the changes made for the A2D2 system.

7.1 A2D2 DESIGN-SNORT MODIFICATIONS

7.1.1 Snort Overview

Snort is a free, opensource, lightweight network instruction detection system. Snort is capable of real-time traffic analysis and packet logging on IP networks. It can be used to detect a variety of attacks and probes and can perform protocol analysis and do content searching/matching.

Snort uses a flexible rules language to describe traffic it has interest in and employs a detection engine that supports a modular plug-in architecture. This ability to add new modules makes Snort very adaptable to new threats. Snort has a real-time alerting mechanism as well. Snort can be used as a packet sniffer, a packet logger or a full intrusion detection system. Snort is available on multiple operating platforms.

For all of these reasons Snort was chosen as the IDS for this project as well as the Autonomous Anti-DDoS Network(A2D2) developed by Angela Cearns. The modifications made in Snort for A2D2 were carried forward for this project. Additional modifications were made to Snort as well.

More information about Snort can be found at: http://www.snort.org. Details regarding the changes made to Snort for A2D2V2 are listed in Section 9.1 below.

7.1.2 A2D2 Snort Specific Modifications

For A2D2, Snort was modified to include new module plug-ins. Of these, there are two which are important to the discussion of this project. First, a module was added which detects a generic flooding attack when traffic flooding is occurring, independent of the type of tool used to generate the flooding. This flood preprocessor evaluates 'x packets over y time' [C02] to determine if a flood is occurring. The user must set a value for what the threshold is that will signal an alert to be triggered by Snort with this new module. As noted in Angela Cearns[C02] thesis, normal traffic thresholds vary based on many things. As such, it is suggested that the user do some basic traffic analysis prior to setting the threshold value. The new flood preprocessor capability was added via two files, spp_flood.c and spp_flood.h

The packet rate is maintained by a linked list introduced with this new module. This list keeps track of the source IP address, the destination IP address, and connection information.

The flood preprocessor is only interested in the type of packet that is being sent, not the packet payload or contents. The types of traffic that it detects are ICMP, UDP, TCP-SYN or TCP-SYN-ACK packets. If the number of packets from a specific source, for a specific period of time exceeds the defined threshold, a flood alert is triggered.

It is important to note that the modifications made to Snort for A2D2 did include measures to counter IP host spoofing.

Another preprocessor was added to Snort to allow for ignoring packets from certain hosts in the flood detection. This allows the user to specify a host or set of hosts for exclusion from the general flood detection mechanism. The ability to do this gives the Snort flood detection mechanism more fine grained control.

7.2 A2D2 RATE LIMITER

As part of A2D2 a rate limiter configuration file was added that allowed the user to statically specify rate levels, rate values (number of packets per second) and duration for each level. This configuration file is used when a Snort Alert is triggered based on the flood detection, to enable automatic application of these limits. The rate limiter program applies the limits as part of its work when it receives a flood in progress notification.

To enable this feature, another preprocessor add-on was developed as well as a rate limiter program which runs on the firewall. After a source surpasses the defined threshold, an initial flood alert is sent to the firewall. The firewall then applies the rules that were defined in the configuration file via the rate limiter program. The FloodRate-Limiters role is to keep track of the incoming packet rate of the host after the rate limiting is applied. If the rate of incoming packets continues to exceed the threshold set, another Snort Alert is sent and the rate limiting is moved to the next level as defined by the user. When the incoming packet rate for a particular host has gone below the threshold limit, the rate limiting is automatically turned off.

The mechanism used to limit the rate of incoming packets on the firewall is the Linux[1] *iptables(8)* mechanism. Iptables is used to set and modify the packet filter rules in the Linux kernel. Several different tables may be defined and each table contains a number of build-in chains and may contain user defined chains. For A2D2 the built in chains were used, along with some user defined chains to set the thresholds for the number of allowed incoming packets at any point in time.

7.3 Q0S FIREWALL RULES

As part of the initial setup there were a set of rules applied to the firewall for the A2D2 network. These rules also utilized the *iptables(8)* mechanism and allowed certain types of traffic to be accepted no matter what the state of an attack. This was necessary to ensure that good traffic was allowed through. It was also done to set up the user defined chains described in the section above so that traffic from these sources would not be subject to the limitations applied for attack traffic.

7.4 A2D2 CLASS BASED QUEUEING(CBQ)

To enable the firewall to respond to the request from the Snort IDS the Linux CBQ mechanism was utilized to allow each type of traffic a percentage of the total bandwidth the system had available. Without the CBQ settings the firewall was so overwhelmed with incoming packets that it could not respond to the messages being sent by the modified Snort IDS preprocessor. The details of the rules applied for A2D2 will not be given, but the utility used to achieve the CBQ settings was *tc(8)*, specifically the *qdisc* option. *qdisc* is short for queueing discipline. This mechanism allows for creation of queues for a specific interface, and when packets come in they are queued to these as appropriate. Some qdiscs can't contain classes, which contain further qdiscs-traffic may then be enqueued in any of the inner qdiscs, which are within the classes. This nesting mechanism allows for the prioritization of traffic which leads to the CBQ implementation found in A2D2.

In figure 7.4.1, the full implementation of the A2D2 network is shown.

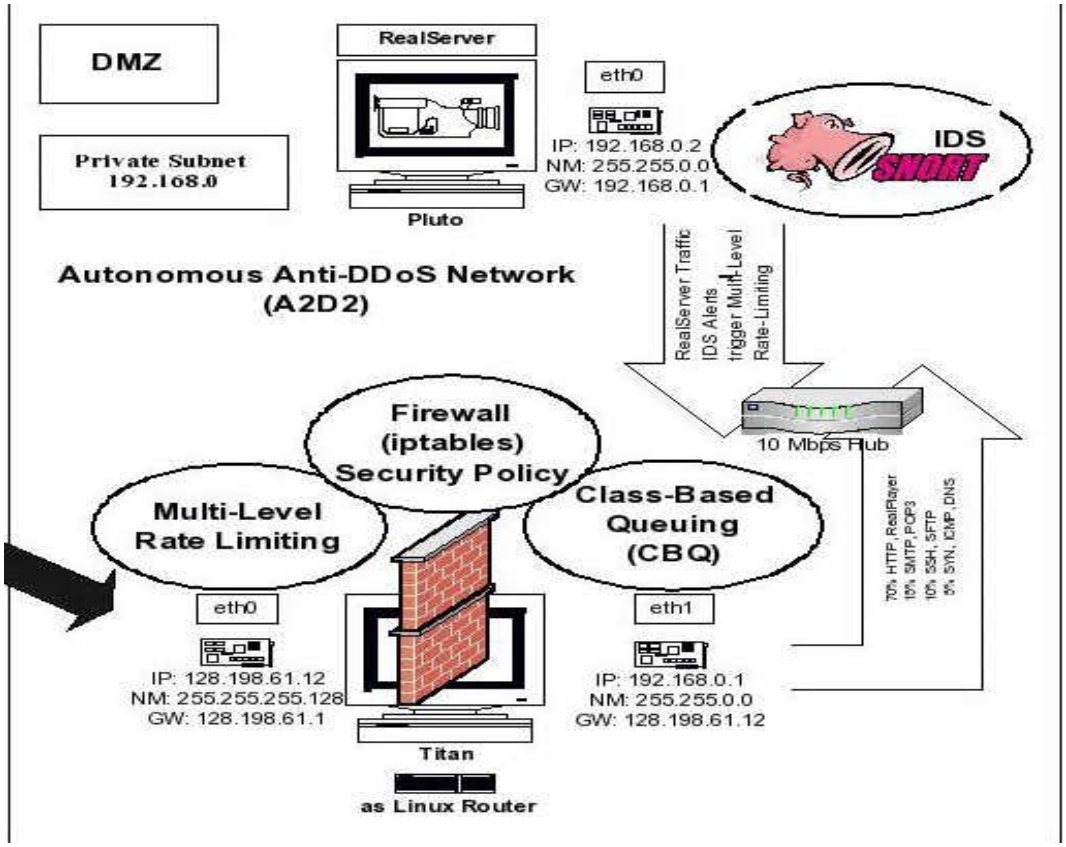


Figure 7.4.1 A2D2 Implementation[C02]

8. A2DV2 FEATURES, ARCHITECTURE AND IMPLEMENTATION

A2D2V2 builds on the features provided with A2D2. As noted above, the original A2D2 design was well suited for a local area network response to a DDoS attack. A2D2V2 provides an Enterprise wide network response to an attack via earlier attack detection and IDIP enabled node cooperation.

There are 7 key feature additions to A2D2 that encompass the features provided by A2D2V2 that will be described in this section:

- IDIP Additions to A2D2V2 Snort IDS
- IDIP enabled firewalls/routers
- Earlier detection and push back of an attack via traffic monitoring on systems not hosting intrusion detection software
- Notification of upstream routers via IDIP messaging regarding perceived attacks
- Notification to upstream routers of attack mitigation strategies taken by surrounding neighborhoods via IDIP messaging
- Upstream router response to notification of attack and strategies taken

Figure 8.1 below shows a communication diagram of how A2D2V2 is setup, with the specific IDIP features on each IDIP enabled node highlighted. The detailed architecture of A2D2V2 is shown figure 9.1.1.

A2D2V2 implements the enterprise wide attack response and coordination utilizing the main concepts of the IDIP protocol. As noted in Section 4.1.1 above, an IDIP neighborhood is an administrative domain, with intrusion detection and response functions managed by a component called the discovery coordinator. Each administrative domain is capable of detection and response as it sees fit, without knowledge of or communication with other IDIP neighborhoods. In the implementation of A2D2V2 each neighborhood has its own discovery coordinator, but in the only neighborhood that actively supports intrusion detection is Neighborhood 2. This Neighborhood is the one with the Snort IDS in the 13.x subnet. The other Neighborhoods, 1 and 3, have the ability to respond to IDIP messages and determine the best response, however they do not have an active intrusion detection mechanisms enabled for the purposes of testing the A2D2V2 implementation. They could easily be enabled this way, as the IDIP messaging and IDIP applications are the same for all Boundary Controllers(noted with BC) in figure 8.1 below.

The A2D2V2 configuration includes a subnet, 15.x, that is not a part of the IDIP community and is not IDIP enabled. The intent of this setup will be shown later, but the basic idea was to show that a non-IDIP enabled set of hosts could reap the benefits of IDIP communities when attacks cross both boundaries.

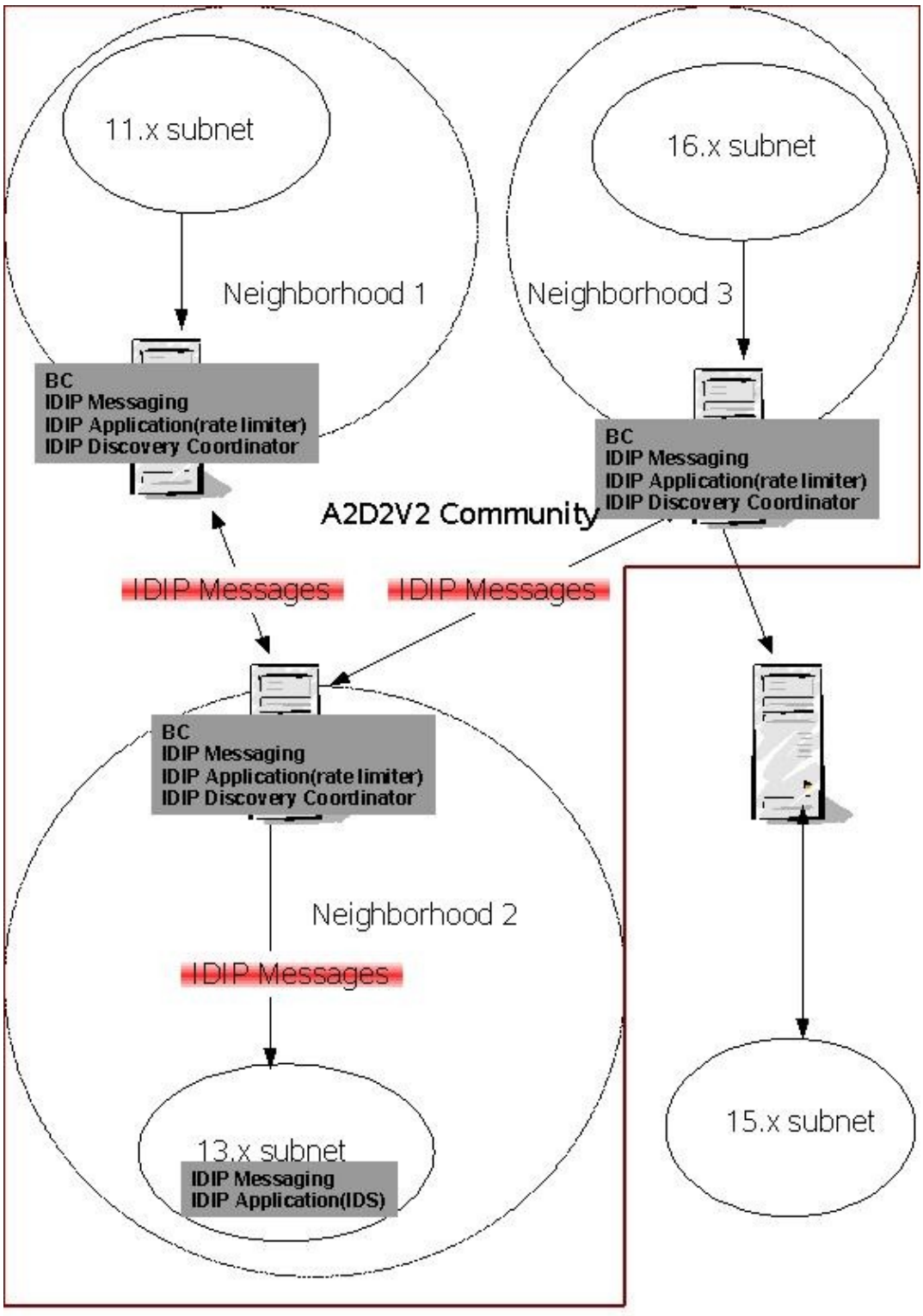


Figure 8.1 A2D2V2 Community and Neighborhood Overview

8.1 A2D2V2 AND IDIP

Unfortunately, there was no available open source implementation if the IDIP protocols. As a result, a partial implementation of the IDIP Message Protocol and IDIP Application Protocol was done for A2D2V2. For A2D2V2 the following pieces were completed:

IDIP Message Protocol:

- IDIP Neighborhood Management via the Discovery Coordinator
- Reliable Delivery of IDIP Messages
- All Message formatting
- Protocol Initialization
- Message Forwarding, including trace requests, and rate limiting requests
- Socket communication

IDIP Application Protocol:

- Modifications to Snort for IDIP Application protocol support
- Addition of an IDIP enabled firewall/router application to enable reception of IDIP messages from Snort IDIP application, initiate tracing of potential attackers and notification to upstream routers when attacks are discovered

8.1.1 A2D2V2 IDIP IDS IMPLEMENTATION

As seen in figure 7.4.1 the A2D2 architecture makes heavy use of an existing Intrusion Detection System (IDS) named Snort. A Snort overview was given in Section 7. The original features added by Angela Cearns work in A2D2 for the Snort IDS, specific to the flood detection and notification mechanism were left unchanged. These features are still provided by the flood preprocessor module noted as *spp_flood.c*. For A2D2V2 the mechanism for receiving the flood alert messages and response to them was modified to be IDIP enabled.

Any node in a network can become an IDIP node, and as noted above in Section 4 an IDS can be used as an IDIP Application Node. As with A2D2 the Snort IDS plays a central part in the intrusion detection feature. However, the actual response to any attack is now IDIP enabled, which results in a much different outcome than A2D2 provided. More details on this in Section 9.

The mechanism to intercept and respond to the flood messages sent by the Snort IDS flood preprocessor was added for A2D2V2 to enable the use of IDIP messages. This section details this work and highlights the pieces of IDIP that are pertinent to this functionality. It does not provide the full implementation details.

The messages for this new mechanism, which was encapsulated in a module named *report_idip.c*, were formatted using the IDIP application message format. Each of these messages has the appropriate application message header and subsequent application

message for use by the upstream IDIP message mechanisms. The definitions of the IDIP application message header and IDIP application message body follows.

The IDIP Application Message Header format is defined as follows:

0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1

Version	Class ID
Length (4 octets)	
Timestamp (4 octets)	
Thread ID (4 octets)	
Originator IDIP Address (16 octets)	
Flags	Pad
Timediff (4 octets)	

- Class ID: Class IDs from the CISL and additional IDIP classes. A list of the class IDs is found in [NB02-1]
- Length: Length of the following IDIP application message
- Version: Identifies the version of protocol.

  IDIP Version: 0x0010
  CIDF Version: 0x0100

After the IDIP application header comes the IDIP application message. The Class ID field in the application header corresponds to the message type that follows the header. For A2D2V2, the supported ClassID types are as follows:

a) Trace

b) Do

c) Undo

Which translated to the following C structure:

```
/*
    * 3.1 IDIP Application Message Header
    * p.4 Figure 2
    */
   struct IDIP_app_msg_hdr {
         uint8_t                    version;
         uint8_t                    class_id;
         uint32_t                    length;
         uint32_t                    timestamp;
         uint32_t                    thread_id;
         struct IDIP_app_orig_addr       orig_addr;
```

```
        uint8_t                 flags;
        uint8_t                 pad[3];
};

/*
 * 3.1 IDIP Application Message Header versions
 */
#define         IDIP_APP_VERS_IDIP              0x0010
#define         IDIP_APP_VERS_CIDF          0x0100

/*
 * 3.1 IDIP Application Message Header Class ID's
 */
#define         IDIP_APP_CLASS_TRACE     0x0020
#define         IDIP_APP_CLASS_DCUNDO  0x002a
#define         IDIP_APP_CLASS_DCDO           0x002e
```

An IDIP application communicates with the IDIP Message Layer through the socket on port 0xc1df. The IDIP Message Layer sender process listens on port 0xc1df for connections from applications. When an application connects to the port, the sender receives a registration message. The registration message includes the IDIP classes of messages the application wants to receive. When the IDIP Message Layer receives a message on the RCV mailbox, it sends the message using the socket connection to all applications who have registered for this type.

For A2D2V2 attack processing an IDIP enabled message creator and forwarding program, *report_idip.c,* was developed which runs on the host running the Snort IDS systems. This new module a) receives flood notification messages from the Snort IDS, b) creates and then forwards an IDIP TRACE message request to the IDIP message/DC coordinator node on the upstream router and c) creates and forwards an IDIP DO Message to the IDIP message node on the upstream router, requesting a rate limiting of the identified source address ***or*** d) creates and forwards an IDIP UNDO message to the IDIP message node on the upstream router requesting to shutoff rate limiting for an address or class of addresses. The decision as to which type of action is taken is dependent on the data received from the Snort IDS flood preprocessor.

In the new module, *report_idip.c* the following section of code does the main part of the work described above.

report_idip.c:

```
analyzePacket(Alertpkt *alert, FILE *log, char *hostname, int portno)
{
        Packet  *p = NULL; /* This is used for logging data only */
        time_t  now;
        int     snd;
        int     created_trace = 0;
        char    *time_string = NULL;
        struct  sockaddr_in     snd_addr;
        idip_message_t idip_trace_msg; /* request for tracing a source */
```

```
idip_message_t idip_do_msg;     /* request for rate limiter */
in_addr_t       haddr;

memset (&idip_trace_msg, 0, sizeof (idip_message_t));
memset (&idip_do_msg, 0, sizeof (idip_app_msg_t));


....
....
/*
 * main part of code to generate IDIP trace and do requests to send to message
 * receiver on upstream IDIP discovery coordinator firewall/router.
 *
 *
 * When a new attack is discovered, create a new trace message to send to
 * upstream firewall/router IDIP enabled message receiver.
 */
 if ((p != null) && (strstr((char *)alert->alertmsg, "end") == null)) {
       if (create_trace_msg(&idip_trace_msg, (char *)alert->alertmsg,
          inet_ntoa(p->iph->ip_src), hostname, 0) != 0) {
             fprintf(stderr, "failed to create trace request \n");
             return (-1);
       }
       created_trace = 1;
} else if (strstr((char *)alert->alertmsg, "end") == null){
       if (create_trace_msg(&idip_trace_msg, (char *)alert->alertmsg,
          "192.168.11.2", hostname, 0) != 0) {
             fprintf(stderr, "failed to create trace request \n");
             return (-1);
       }
       created_trace = 1;
}
/*
 * send the idip trace message to the firewall. the firewall
 * will then begin a tcpdump process and gather data
 * from the host we have identified in this message.
 * the firewall will notify upstream routers of the attack
 *
 */
if (created_trace) {
       if (sendto(snd, (idip_message_t *)&idip_trace_msg,
          sizeof(idip_message_t), 0,
          (struct sockaddr *)&snd_addr, sizeof (snd_addr)) < 0) {
             perror("could not send message to idip message layer");
       }
       created_trace = 0;
}


/*
 * Create the IDIP 'do' message for the upstream IDIP enabled firewall/router
 * message receiver/ The IDIP enabled application on the firewall/router will then
 * process this request to adjust the rate, and forward this action taken on to the
 * upstream routers for their consideration.
 *
```

```
         */
        if (p != null) {
                if (strstr((char *)alert->alertmsg, "end")) {
                        if (create_do_msg(&idip_do_msg,
                           (char *)alert->alertmsg,
                           hostname, inet_ntoa(p->iph->ip_src),
                           idip_restore_rate) != 0) {
                              fprintf(stderr, "failed to create_do_msg \n");
                              return (-1);
                        }
                } else {
                        /*
                         * this is a rate limiting request.
                         */
                        if (create_do_msg(&idip_do_msg,
                           (char *)alert->alertmsg, hostname,
                           inet_ntoa(p->iph->ip_src),
                           idip_limit_rate) != 0) {
                           fprintf(stderr, "failed to create_do_msg \n");
                              return (-1);
                        }
                }
        } else { /* end if p != null */
                if (strstr((char *)alert->alertmsg, "end")) {
                        if (create_do_msg(&idip_do_msg,
                           (char *)alert->alertmsg,
                           hostname,
                           "192.168.11.2", idip_restore_rate) != 0) {
                              fprintf(stderr, "failed to create_do_msg \n");
                              return (-1);
                        }
                } else {
                        if (create_do_msg(&idip_do_msg,
                           (char *)alert->alertmsg, hostname,
                           "192.168.11.2", idip_limit_rate) != 0) {
                                   fprintf(stderr,
                                    "failed to create_do_msg \n");
                                    return (-1);
                        }
                }
        }

        if (sendto(snd, (idip_message_t *)&idip_do_msg,
                sizeof(idip_message_t), 0,
                   (struct sockaddr *)&snd_addr, sizeof (snd_addr)) < 0) {
                      perror("could not send message to idip message layer ");
                      exit (1);
 }
....
....
```

8.1.2 A2D2V2 IDIP ENABLED FIREWALL/ROUTER(S)

A2D2V2 provides the IDIP message protocol mechanism via a module named *idip_firewall_receiver.c* which runs and listens on the appropriate nodes to intercept and process the IDIP messages sent. In A2D2V2 the basic communication flow is as follows:

Snort IDS ->generates flood report when attack is detected

report_IDIP -> intercepts flood report message

report_IDIP->creates three classes of IDIP messages:

IDIP DO

IDIP UNDO

IDIP TRACE

report_IDIP->forwards IDIP message to upstream firewall/router

IDIP_firewall_receiver->receives IDIP message and processes according to request

The *idip_firewall_receiver* module is the central IDIP message application module in A2D2V2. This application is responsible for all of the IDIP message processing. This section will highlight the IDIP Message Layer source code that was developed for A2D2V2. The full implementation will not be shown, but the location of the full source is shown in Appendix B. The IDIP Message Layer application is the central piece in an IDIP enabled network.

As noted in [NB02-2] the IDIP Message Header is defined to be as follows:

1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1

Version	Flags	Length
Next Type	Pad	Checksum
Sequence Number (4 octets)		
Time-Stamp (4 octets)		
Priority (4 octets)		
Destination Address (4 octets)		
Destination Process ID Number (4 octets)		
Destination Boot Time (4 octets)		
Pad (4 octets)		

This translated in to the following C structure definition:

```
struct  idip_header {
        uint18_t    version;
          uint8_t   flags;
          uint16_t  length;
          uint8_t   next_type;
          uint8_t   pad;
```

```
        uint16_t  checksum;
        uint32_t  seq_num;
        uint32_t  time_stamp;
        uint32_t  priority;
        uint32_t  dest_addr;
        uint32_t  dest_proc_id;
        uint32_t  dest_boot_time;
        uint32_t  pad_extra;
    };
```

Next type field values

```
#define IDIP_MESSAGE                0x0
#define IDIP_APPLICATION_DATA  0x01
#define IDIP_HELLO_DATA             0x02
#define IDIP_NKID_DATA              0x03
#define IDIP_CRED_DATA              0x05
#define IDIP_STARTUP_DATA           0x06
#define IDIP_ESP                        0x32
#define IDIP_AH                             0x33
```

Flag values

```
#define IDIP_ACK                    0x1
#define IDIP_ND_ACK                     0x2
#define IDIP_NCI_ACK                    0x4
#define IDIP_NR_ACK                     0x8
#define IDIP_NN_ACK                     0x10
#define IDIP_NC_ACK                     0x11
```

This header information is what is used to determine the appropriate response by the IDIP Message Layer when it receives an incoming message. The following rules are used for Inbound Message Processing:

- If the version number is not the proper version number the message is discarded. For A2D2V2 this version number is static and is not checked
- If the next type is valid application data respond to the sender using the same IDIP header, with no data, with the control field set to ACK as an acknowledgment.

For outbound message processing when an IDIP Application requests IDIP Message transmission, the IDIP Message Layer does the following:

- Builds the Header
- Records the list of recipients as specified by source node
- Transmits the message to the list of recipients
- Waits for acknowledgment or time-out from recipients

The IDIP Message Layer consists of two functions, the sender and receiver. For A2D2V2 both sender and receiver were implemented in one process and communication

was managed via sockets. The IDIP protocol allows for making these 2 processes with multiple mailboxes to manage communication. For A2D2V2 this was not utilized for testing, only because it was not central to proving the reliability and functional pieces of this project. This was implemented however, and is a part of the final source found as noted in Appendix B.

The central message module in A2D2V2, *idip_firewall_receiver,* listens on the IDIP_APP_PORT, number noted above, for incoming IDIP messages. As shown below in figure 9.1.1, the A2D2V2 test bed had 4 routers, 3 of which were IDIP enabled. These routers served multiple roles as the IDIP Boundary Controllers(BC) between IDIP neighborhoods, firewalls, IDIP application nodes and IDIP Discovery Coordinators. IDIP allows for the BC's to set policies with regard to acceptance of an processing of incoming messages from other boundary controllers. In A2D2V2 this policy was set to 'accept' at all times, that is all IDIP messages incoming to any BC from another BC were accepted and processed. In a real-life network this would not likely be allowed as all traffic coming in from other IDIP enabled nodes would have to be validated in some way prior to acceptance and processing. Within the A2D2V2 test bed, all traffic from each BC was known to be legitimate, and thus no additional validation was required.

idip_firewall_receiver has several key areas that implement the capabilities noted above. Implementation of the message receiving mechanism is done in the main() part of the module as shown below. It simply listens on the *gen_mbx, IDIP_MSG_PORT*, for incoming IDIP message. The *process_idip_message()* function handles the bulk of the functionality once a message is received.

IDIP_firewall_receiver.c:

```
void
main() {

        int             length;
        int             n;
        idip_message_t      i_message;
        struct sockaddr_in    toaddr;

       /* set up our listening socket */
        if ((gen_mbx = socket(af_inet, sock_dgram, 0)) < 0) {
              fprintf(stderr, "unable to set up receiver socket.\n");
              perror(strerror(errno));
              return;
        }
        /*
         * listen for messages from any host, on the idip_app_port
```

```
        */
       (void) memset(&gen_from, 0, sizeof (gen_from));
       gen_from.sin_family = AF_INET;
       gen_from.sin_addr.s_addr = INADDR_ANY;
       gen_from.sin_port = htons(IDIP_APP_PORT);

       if (bind(gen_mbx, (struct sockaddr *) &gen_from,
            sizeof (struct sockaddr_in)) < 0) {
            fprintf(stderr, "%s", "Could not bind to port\n");
            perror(strerror(errno));
       }

       length = sizeof (gen_from);

       if (getsockname(gen_mbx, (struct sockaddr *) &gen_from, &length)) {
           perror("getting socket name");
            exit(1);
       }
       while (1) {
            n = recvfrom(gen_mbx, &i_message,
               sizeof (IDIP_message_t),
               0, (struct sockaddr *)&gen_from, &length);
....
....
            /*
             * Process this message. It is possible that there has
             * been a transmission problem or data is garbled. Move on
             * if this is the case.
             */
            if (process_idip_message(&i_message) != 0) {
                 perror("error processing IDIP message");
                 continue;
            }
       }
}
```

Processing of the incoming messages depends on the type of data enclosed in the message as pointed to by the next_type member of the *i_hdr.next_type* member of the *idip_message_t* structure. The *class_id* member of this structure indicates the type of application request this message contains. This data tells the message layer module, *idip_firewall_receiver* how to handle the message it has just received.

idip_firewall_receiver: process_idip_message():

```
static int
process_idip_message(idip_message_t *msg)
{
    int error = 0;
    idip_app_msg_t app_msg = msg->p.app_msg;

    if (msg->i_hdr.next_type == IDIP_APPLICATION_DATA) {
        if (msg->p.app_msg.app_hdr.class_id == IDIP_APP_CLASS_TRACE) {
            error = do_trace_request(&(msg->p.app_msg));
            if (error) {
               perror("executing trace request");
               return (-1);
            }
        } else if (msg->p.app_msg.app_hdr.class_id ==
            IDIP_APP_CLASS_DCDO) {
            error = do_request(&(msg->p.app_msg));
            if (error) {
               perror("executing do request");
               return (-1);
            }
        } else if (msg->p.app_msg.app_hdr.class_id ==
            IDIP_APP_CLASS_DCUNDO) {
            error = undo_request(&(msg->p.app_msg));
            if (error) {
               perror("executing undo request");
               return (-1);
            }
        }
    } else {
        printf("Unrecognized app header type\n");
```

```
            return (-1);
        }
        return (0);
    }
```

As you can see, there are the three types of messages that are recognized and implemented for A2D2V2 DO, UNDO and TRACE message types.

8.2 A2D2V2 DYNAMIC TRACING AND ENTERPRISE NOTIFICATION TO ACHIEVE COOPERATION

One important feature of A2D2V2 is the ability to dynamically start a proactive response to a perceived attack farther up in the network hierarchy away from the network under attack. This is part of the IDIP trace request response implemented in *idip_firewall_receiver.c* The idea is that the farther removed from the network under attack an attack can be detected and mitigated the smaller burden imposed each individual network within the enterprise.

In A2D2V2 the tracing mechanism used was *tcpdump* and the push back was achieved by the IDIP Discovery Coordinator utilizing a static table of upstream router addresses to send out the messages required to push the response to the attack farther upstream in the network. There were several choices considered for use as 1st the tracing mechanism and 2nd the discovery of upstream notification of upstream routers for this project. A description of those, along with the implementation details of the *tcpdump* usage is given in this section.

Within a IDIP Neighborhood, multiple hosts can be enlisted to trace the traffic from incoming sources. This data is then sent to the Discovery Coordinator embedded in the IDIP Message node for archival. Ultimately, the Discovery Coordinator would formulate a more fine grained response to an attack based on the data from the trace messages. For A2D2V2 this capability was limited to one node in the neighborhood and a coarse grained attack response was implemented in the A2D2 to limit the rate of incoming packets from those hosts discovered during the tracing exercise. A notification from Snort to the Discovery Coordinator via the *report_idip* module triggers a '*Trace'* request first, and then a subsequent *'Do'* request to the firewall to begin traffic limiting and bandwidth management.

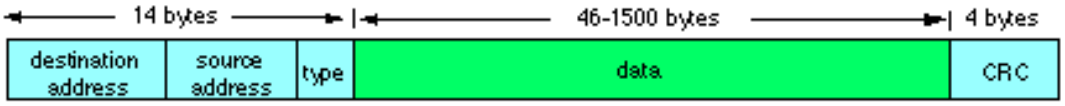
8.2.1 Considerations For Dynamic Tracing Mechanism

8.2.1.1 IP Link Level Header Parsing and Address Resolution Protocol

Within an IP packet there is a header, called the Link Level header, which represents the data link level information that is available about the source of the packet. The data link layer in a network is the layer immediately below the IPV4/V6 layer.

The link level header information is obtained with utilities such as *tcpdump* and consists of the data link header data that is sent from the data link layer to the protocol

layer when packets are transmitted. In the case of A2D2V2 which is an Ethernet based network, the data link packet would look as follows:



In an Ethernet network the Medium Access Control(MAC) protocol is used to provide the data link layer of the Ethernet LAN. For the purposes of extracting the header only the following parts of the above packet are interesting:

The header consists of three parts:

- A 6-byte destination address, which specifies either a single recipient node (unicast mode), a group of recipient nodes (multicast mode), or the set of all recipient nodes (broadcast mode).
- A 6-byte source address, which is set to the sender's globally unique node address. This may be used by the network layer protocol to identify the sender, but usually other mechanisms are used (e.g. arp). Its main function is to allow address learning which may be used to configure the filter tables in a bridge. This is the system MAC address.
- A 2-byte type field, which provides a Service Access Point (SAP) to identify the type of protocol being carried (e.g. the values 0x0800 is used to identify the IP network protocol, other values are used to indicate other network layer protocols). In the case of IEEE 802.3 LLC, this may also be used to indicate the length of the data part. The type field is also be used to indicate when a Tag field is added to a frame.

Utilizing a utility like *tcpdump* and parsing the output from the link level headers to obtain the system MAC address was considered as the mechanism by which to trace packets coming in and identify the source of the potential attack. To make use of the MAC address we would have to then resolve the MAC address in to its known IP address.

When a device needs to send an IP packet to another device on the local network, the IP software will first check to see if it knows the hardware address associated with the destination IP address. If it has this data it will simply transmits the data to the destination system, using the protocols and addressing appropriate for whatever network medium is used between the two devices. However, if the destination system's hardware address is unknown, then the IP software must locate it before any data can be sent. At this point IP will call on the Address Resolution Protocol(ARP) to locate the hardware address of the destination system. This resolution is achieved by a low-level broadcast onto the network, requesting that the system that is using the specified IP address respond with its hardware address. When the requesting system gets an ARP response, it will store the hardware and IP address pair of the requested device into a local cache. The next time the system needs to send data it will consult with the local cache first, prior to issuing an ARP request.

You might ask why it would be necessary in A2D2V2 to do this address resolution when the IP packets coming in to the router have a source IP address specified? IP

addresses are filled in by the application at the time of packet generation. It is easy to 'spoof' IP addresses, that is create a packet which contains an IP address which is really not the IP address of the source, thus making it difficult to detect the real source of a packet. In a DDoS attack mitigation scheme this could result in stopping legitimate client traffic. And, in many DDoS attacks IP spoofing is done to make it more difficult to find the legitimate sources of the attacks.

With this in mind and with the knowledge that while it is possible to spoof the MAC address it isn't as likely as spoofing an IP address, so the question became how can we resolve the MAC level address from an incoming packet to the router to its IP address to ensure we limited traffic from the appropriate source address? There are 2 possibilities that were considered for A2D2V2:

The 1st one was to use the Reverse Address Resolution protocol, at the firewall/router to determine the source IP of an incoming packet. The *rarp(8)* utility can be used to map dynamically between the IP and network interface MAC addresses. Machines that boot over the network use *rarp(8)* to discovery their own internet protocol address. This mechanism can be generalized for use for queries about specific MAC addresses. The main limitation with this approach is that there must be a Reverse Address Resolution Protocol server that responds to these requests and can resolve the Ethernet address in to its corresponding IP address. A secondary limitation *rarp(8)* is not supported in Linux after 2.3 making it difficult to use.

The 2nd possibility explored was to intercept packets coming in to the firewall/routers and record the <Ethernet address, IP address pair> from those packets utilizing the *tcpdump* utility. This list would then be consulted when an attack was detected from a specified source IP address to find the MAC address corresponding to the specified IP address. Again, due to IP spoofing we would need to take one additional step to ensure we had the correct source address, that is to use ARP to send a broadcast on the network to obtain the correct IP address. There are several drawbacks with this approach:

1. The list of <Ethernet address, IP Address> pairs cannot be infinitely long. During a heavy attack it is likely that this list would be rolled over many times, resulting in the high probability of not being able to obtain the Ethernet MAC address for a specified IP address. This would result in either dropping this IP address as an attacker or blocking traffic from this address and possibly blocking legitimate client traffic
2. The time it would take to maintain and consult this list could be significant depending on attack load, which wold result in a significant delay in attack response
3. It is important to remember that the scope of ARP is a single IP link, that is the only address resolution the system is able to maintain would be one link away. If the network is comprised of many interim routers the use of ARP would not be sufficient to resolve the IP address of the source. To resolve the source completely one would have to traverse up the list of addresses that were resolved until the end was found.

8.2.1.2 TCPDUMP

Another way to use *tcpdump* is to use it to monitor traffic on each interface known on a system and to record this traffic for a period of time while collecting the incoming source IP addresses. *tcpdump* has several options that allow for fined grained control of monitoring of incoming traffic. It can be monitored for each individual interface and allowing for specifications of specific source addresses to be watched. This utility is very flexible and powerful. The main limitation with using this is something mentioned in the section above, that is IP source address spoofing. With *tcpdump* the source IP address is recorded literally from the IP Packet with no modification. So, if an attacker is spoofing this address *tcpdump* does not catch this. Based on all the data from the previous sections and realizing that even MAC addresses can be spoofed via software the use of *tcpdump* as the active tracing mechanism was chosen. *tcpdump* was used to monitor each known interface on the firewall/router system, to track the number of packets for the specified period of time, while also tracking the IP addresses for each of those packets. This decision was partly based on the decision to reuse the A2D2 Rate limiter program which also has provisions specifically encoded for IP spoofing issues[C02].

For A2D2V2 *tcpdump* is invoked when an IDIP Trace request is received from the IDIP IDS application. The data gathered during this tracing was then used to determine if an attack was underway and if the IDIP enabled firewall/router should issue an IDIP message with this attack data to the known upstream routers.

The implementation details of how the dynamic tracing was achieved are shown below. A set of scripts, utilizing both shell commands and *awk(1)* pattern and processing language were developed to achieve the dynamic tracing and subsequent recording and archiving of this trace data.

tcpdump.sh:

```
#!/bin/sh

# set time limit based on what caller specified. Exec script that will send
# SIGTERM to tcpdump to force this script to run the END block. Background
# this so it doesn't interrupt gawk processing below.

# Invoke tcpdump with options and pipe through gawk to gather data. The
# running of tcpdump is limited to the time specified by the caller. I
# am only interested in the ip protocol packets. I will get the source
# and destination addresses with the "ip' specifier at $3 and $5 respectively.
# Do not track outgoing packets from this host as part of tracing data. This is
# achieved by the 'src host not loghost' qualifier.

#
# I need to dump on every interface I find on system. so, call ifconfig -a
```

```
# first, to get interface name. Call tcpdump on these.

INTERFACES=`/sbin/ifconfig | gawk ' {
        # Get the interface name
        x = split($1, ifname)
        newif[i]=ifname[1]
        if (match(newif[i], "eth") && newif[i] != "lo") {
                printf("%s ", newif[i])
 }
        i = i + 1
} '`
for i in $INTERFACES
do
# for each interface check number of packets , if over threshold, report
./dumper.sh $i $1 > /tmp/o_$i &
done
# kill this process in $1 amount of time
./trace_kill $2
sleep 3
/bin/cat /tmp/o_*
#rm /tmp/o_*
```

This script loops through every known interface on the system, discards the loop back interface, and calls another shell script named *dumper.sh* to invoke the actual call to *tcpdump* with the appropriate options. The process running the *tcpdump* command is killed in a set amount of time based on the original flood message received by the Snort IDIP enabled IDS.

The dumper.sh script utilizes the *awk* programing language to keep track of the number of packets received and which interface the packet arrived on for each source IP address found. It also invokes the *tcpdump* utility with the appropriate options and logic to monitor the interfaces on that system. The code shown below is specific to the R99 router. The rules for each firewall/router differ in the specific configuration.

dumper.sh:

```
# This is the dumper program for host R99. Each of these is slightly different
# based on the /etc/hosts file.
/usr/sbin/tcpdump -i $1 -lnq ip src host not loghost and not localhost  2>/dev/null | \
gawk -v threshold=$2 -v interface=$1 '
```

```
{
        split($3, ip, ".")
        x=sprintf("%d.%d.%d.%d", ip[1], ip[2], ip[3], ip[4])
        source[x,interface] += 1
}
END {
        for (name in source) {
                if (source[name] >= threshold) {
                        split(name, ar, SUBSEP);
                        printf("%s %s %s\n", ar[1], ar[2], source[name])
                }
        }
} '
```

8.2.2 Considerations For Discovery of Upstream Routers To Notify When Attack is Discovered

8.2.2.1 Traceroute

traceroute(8) is an application that tracks the routes packets can take across a TCP/IP network on their way to a given host. It utilizes the IP protocol time to live (TTL) field and attempts to elicit a ICMP TIME_EXCEEDED response from each gateway along the path to the host. The general use of this is from the host you wish to trace packets from, specifying the host for to which you want to get the packets route. So, for example, if we wanted to trace a potential route from host 128.198.61.99 to google.com, the command would look like:

Run on 128.198.61.99:

```
traceroute google.com
```

Yields for first run:

```
traceroute to google.com (64.233.167.99), 30 hops max, 40 byte packets
 1  128.198.60.1 (128.198.60.1)  0.206 ms   0.176 ms   0.175 ms
 2  * * *
 3  * * *
 4  dvr-edge-03.inet.qwest.net (65.121.122.205)  5.577 ms   7.865 ms   8.084 ms
 5  dia-core-02.inet.qwest.net (205.171.10.77)  5.874 ms   6.059 ms   6.023 ms- ******
 6  cer-core-02.inet.qwest.net (67.14.8.22)  30.103 ms   29.992 ms   29.965 ms
 7  chx-edge-01.inet.qwest.net (205.171.139.166)  29.954 ms   30.310 ms   30.328 ms
 8  65.112.69.202 (65.112.69.202)  30.675 ms   29.404 ms   29.970 ms
```

9 216.239.46.1 (216.239.46.1) 29.482 ms 30.242 ms 29.855 ms

10 66.249.95.121 (66.249.95.121) 29.737 ms 72.14.232.53 (72.14.232.53) 31.438 ms 31.879 ms

11 72.14.232.57 (72.14.232.57) 32.110 ms 30.955 ms 31.013 ms

Yields for 2nd run:

traceroute to google.com (72.14.207.99), 30 hops max, 40 byte packets

1 128.198.60.1 (128.198.60.1) 0.315 ms 0.198 ms 0.179 ms

2 * * *

3 * * *

4 dvr-edge-03.inet.qwest.net (65.121.122.205) 5.107 ms 4.956 ms 5.379 ms

5 dia-core-01.inet.qwest.net (205.171.10.33) 5.217 ms 5.540 ms 5.064 ms- ******

6 svl-core-02.inet.qwest.net (67.14.12.10) 30.848 ms 30.862 ms 31.779 ms

7 pax-edge-01.inet.qwest.net (205.171.214.34) 32.120 ms 30.820 ms 31.503 ms

8 72.165.46.18 (72.165.46.18) 31.690 ms 32.369 ms 30.950 ms

9 66.249.95.66 (66.249.95.66) 33.042 ms 31.844 ms 66.249.94.19 (66.249.94.19) 32.262 ms

10 216.239.46.45 (216.239.46.45) 57.162 ms 56.209 ms 56.852 ms

11 72.14.233.146 (72.14.233.146) 67.170 ms 66.909 ms 67.177 ms

12 66.249.94.94 (66.249.94.94) 67.793 ms 68.030 ms 66.734 ms

13 66.249.94.118 (66.249.94.118) 73.306 ms 72.460 ms 74.715 ms

14 72.14.207.99 (72.14.207.99) 67.639 ms 67.360 ms 67.137 ms

There is one major limitation to note with the use of *traceroute(8)* as the output above shows. Note the ******, line 5 for each output. Prior to this, the addresses shown in the potential route are identical, but beginning with line 4 these addresses diverge, thus leading to a totally unique route that the packet could take to the specified destination. The routes shown at any given invocation of *traceroute(8)* are 'possible' routes, not deterministically probable. Thus utilizing this mechanisms to determine the upstream routers to notify of an ongoing attack would be non-deterministic. This would result in delayed attack mitigation response and possibly no mitigation response at all with the upstream routers.

8.2.2.2 Netstat -rn

netstat(8) is a utility that allows for printing network connections, routing tables and interface statistics. The *-rn* options specifically state that we are interested in routing table information showing numerical addresses instead of the host name. The thought was that we could use this to see the directly connected upstream routers with this utility.

There are several concerns with using *netstat* as the mechanism to determine a packets route. Consider the following example:

windom.uccs.edu> netstat -rn

Kernel IP routing table

Destination	Gateway	Genmask	Flags	MSS	Window	i rtt	Iface
172.16.29.0	0.0.0.0	255.255.255.	U	0	0	0	vmnet8
172.16.223.0	0.0.0.0	255.255.255.0	U	0	0	0	vmnet1
128.198.160.0	0.0.0.0	255.255.224.	U	0	0	0	eth0
169.254.0.0	0.0.0.0	255.255.0.0	U	0	0	0	eth0
0.0.0.0	128.198.160.1	0.0.0.0	UG	0	0	0	eth0

The router default router shown is 128.198.60.1. So, utilizing *netstat* we would have chosen this address to send the IDIP messages to for enabling push back of the attack traffic.

However, in the router, you may have more routing entries such as:

169.100.160.0	0.0.0.0	255.255.224.0	U	0	0	0 eth1
100.198.160.0	0.0.0.0	255.255.224.0	U	0	0	0 eth2
128.198.160.0	0.0.0.0	255.255.224.0	U	0	0	0 eth3

The upstream router 1 may have 169.100.162.33 as IP address as indicated by subnet 169.100.160.0/19 in the entry.

The upstream router 2 may have 100.198.162.33 as IP address as indicated by subnet 100.198.160.0/19 in the entry.

The upstream router 3 may have 128.198.162.33 as IP address as indicated by subnet 128.198.160.0/19 in the entry.

Assume we received an intrusion packet from 12.0.0.10, how can you tell via which upstream router the packet arrived? We can't determine this exactly and it we would either have to send the attack notification to all possible upstream routers, or traverse farther upstream from these routers to try to determine how the 12.0.0.10 host is attached and how its packets may be routed. This could be potentially time prohibitive and we could never really be assured we would get the correct router information.

8.2.2.3 Static Routing Configuration Files

With the A2D2V2 test bed as shown in figure 9.1.1, the routing for the 4 routers, R97, R98, R99 and R102 was setup as static routing tables. For example, the routing tables for R99 look as follows:

netstat -rn

Kernel IP routing table

Destination	Gateway	Genmask	Flags	MSS	Window	irtt	Iface
128.198.61.0	0.0.0.0	255.255.255.128	U	0	0	0	eth2
192.168.16.0	192.168.14.102	255.255.255.0	UG	0	0	0	eth3
192.168.15.0	192.168.14.98	255.255.255.0	UG	0	0	0	eth3
192.168.14.0	0.0.0.0	255.255.255.0	U	0	0	0	eth3
192.168.13.0	0.0.0.0	255.255.255.0	U	0	0	0	eth1
192.168.12.0	0.0.0.0	255.255.255.0	U	0	0	0	eth0
192.168.11.0	192.168.12.97	255.255.255.0	UG	0	0	0	eth0
169.254.0.0	0.0.0.0	255.255.0.0	U	0	0	0	eth3
0.0.0.0	128.198.61.1	0.0.0.0	UG	0	0	0	eth2

Each router in the A2D2V2 test bed has a similar set of static routing defined. With this in mind it was decide that a file that contained the specific routing information for each interface on a system would be used to determine where to send the IDIP messages for attack notification to upstream routers. With no dynamic utility to definitively determine the exact route a packet has taken this solution seemed like the most efficient and consistent way to deal with this discovery. In a real world scenario this would likely have to be changed and is the subject of future work as discussed in section 11.

8.3 A2D2V2 PORTABILITY

All of the initial work for A2D2V2 Snort modifications was done using the Solaris[2] Operating System. Every effort was made to ensure that the components introduced in A2D2V2 were portable to other platforms. To achieve this the same tools used in A2D2 were used with A2D2V2. The compilation and initial testing was done on Solaris.

As a more robust testing criteria was needed the test bed as described in Section 9.1 was developed. All hosts used in this test bed are running Linux and as such a re-compilation and resting was done.

The original A2D2 firewall capability was retained and it is used when the firewall receives an IDIP message for a rate limit request. The Class Based Queuing(CBQ) portion of A2D2 was used as well.

Along with the use of open source tools, the IDIP infrastructure as developed for A2D2V2 was designed with portability in mind so that it could easily be used on any UNIX[3]/Linux operating system. It was written in C and uses standard socket interfaces. A recompile should be all that is necessary to port this to another operating platform. It could be argued that to make this component even more portable, Java should have been used for the language of implementation. However, the resident set size and footprint of a Java Virtual Machine, would make it very difficult to provide for a lightweight, resource minimal solution, as the IDIP specification notes.

One of the intended outcomes of the A2D2V2 design was to make this system available in heterogeneous environments. With the exception of Win32, I believe this goal was met.

9. A2D2V2 TEST BED SPECIFICATIONS AND PERFORMANCE RESULTS

9.1 TEST BED CONFIGURATION

To test the assumptions of A2D2V2 a specific test configuration was setup to enable a remote notification scheme with cooperating IDIP firewall/router nodes. The test bed is presented in figure 9.1.1. There are a total of 11 nodes in the test bed. There are 5-100MB switches and 1-10MB switch. Routers are denoted with R<number>, attackers are denoted with A<number> and clients are denoted with C<number>.

7 of the nodes in the test bed are HP Vectra, 600 MHz with 256MB of memory. These machines are the following:

- A1, A2,, C1, C2, S1, S2, R102

The 4 remaining nodes are Dell Optiplex GX150's, 1GHz with 512K of memory. These machines are the following:

- R97, R98, R99 and A3

Each of them is installed with the Fedora Core 5 release of Linux. In consideration of the differing capabilities of these machines, specifically the apparently faster machines used for the routers, it is important to note that this setup was not done this way to try to skew the performance results in any specific way. These were simply the machines that were available to me for testing. It is not expected that the difference in machines will make a discernible difference in the performance results. The faster routers are offset by a faster attack client, and slow servers. Ultimately the performance measurements do not come down to a specific packet processing speed but the overall way in which the non-attack clients themselves recover.

There are 4 LANS in the A2D2V2 test bed, each attached to their own firewall/router. This setup was done to show the enterprise capability of A2D2V2 in detecting and mitigating DDoS attacks. These LAN's are numbered 11.X, 13.X, 15.X and 16.X. DDoS attackers are contained in both the 11.X LAN and the 16.X LAN. This is done to show the distributed nature of a DDoS attack, and to show the ability to push back the attacks detected across this attack distribution.

The IDIP enabled firewall/routers are the R97, R99, R102 routers. R98 is not enabled with any special software and no pre-set firewall rules or class based queueing applied. As you may note, the routers in the A2D2V2 test bed are addressed in a non-traditional way, that is not utilizing the standard router addressing of a .1 or .254 for the last part of the IP address. The choice to use non-traditional addresses for the A2D2V2 test bed was done to enable ease in identifying the servers and routers during for setup of the software

and test configuration. The naming scheme chosen was to utilize the routers name as the last part of the IP address.

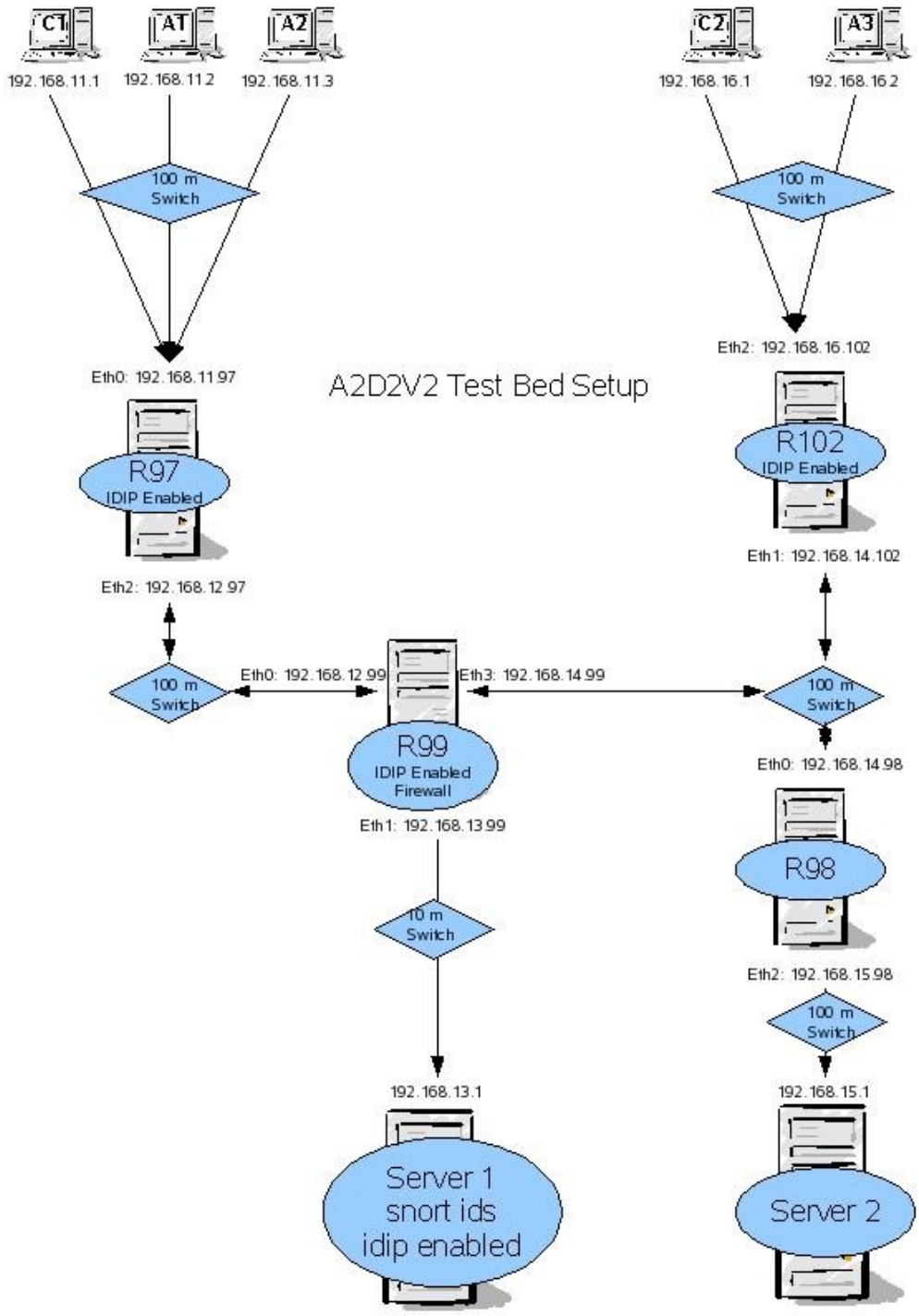


Figure 9.1.1 A2D2V2 test bed

The essential software contained on each node will be listed below on a per node type basis. This is the general list of software. Each of these have been specifically modified as needed for the specific host configuration on which they are running. For example, the cbq.sh script must know which network interfaces that the rules must be applied to. For each router/firewall in the A2D2V2 test bed configuration this is potentially different.

Router/firewall software, R97, R99, R102:

- idip_firewall_receiver – IDIP message application and discovery coordination
- tcpdump.sh – dynamic tracing script
- dumper.sh – dynamic tracing script
- trace_kill – script that dumper.sh calls to kill the tcpdump process
- cbq.sh – Class Based Queueing setup and initial firewall rule setup
- rateif.pl – rate limiter perl program
- rateif.conf – rate limiter configuration file
- topo.txt – static router topology file

Server software for S1 only:

- snort v1.8.6 with additional spp_flood.c preprocessor module
- report_idip.c – Flood notice receiver application and IDIP message creator and forwarding agent
- tcp_snd – basic tcp server software developed to gather performance data. It streams a message to any client connected.

Clients, C1 and C2:

- tcp_rcv – basic tcp client software developed to gather performance data. Connects to named server and accepts message sent continuously by tcp_snd program on server.
- Plot.pl – traffic statistics gathering program

Attackers, A1, A2 and A3:

- Stacheldraht Version 4- attack tool

9.2 A2D2V2 TEST SCENARIOS

Three main test scenarios are deployed to test the feasibility and functionality of the A2D2V2 system. These are:

1. Normal tcp_rcv traffic running on C1 and C2 and tcp_snd running on S1 with no attack. This is used for baseline packet performance data.
2. Normal tcp_rcv traffic running on C1 and C2, tcp_snd running on S1 with the TCP SYN flood attack running on A1, A2 and A3 targeting S1,

192.168.13.1 and S2, 192.168.15.1. No IDIP or IDS software running nor class based queueing has been applied. This is to show the affect on the clients with no DDoS attack mitigation. Results shown are for C1 only. C2 exhibited exact symptoms as C1 in this test scenario, that is the near total loss of packet transmission.

3. Normal tcp_rcv traffic running on C1 and tcp_snd running on S1 with a 3 1/2[1] minute non-stop TCP SYN attack running on A1 and A2 with R97 and R99 running IDIP enabled software, and S1 running IDIP enabled Snort IDS. Class based queueing and other QoS techniques have been applied to each participating router/firewall as discussed in Section 8.1.2. This scenario is intended to show the attack response within 2 LAN's only. Cooperation happens between the R97 and R99 firewall/routers.
4. Normal tcp_rcv traffic running on C1 and C2, tcp_snd running on S1 and S2 with the non-stop TCP SYN flood attack running on A1, A2 and A3 targeting both S1 and S2 for 3 ½ minutes, along with the A2D2V2 IDIP enabled Snort running on S1, and IDIP firewall/router software running on R97, R99 and R102. Class based queueing and other QoS techniques have been applied to each participating A2D2V2 router/firewall as discussed in Section 8.1.2. This is to show the results of a full enterprise wide cooperative DDoS attack response and mitigation scenario. This test was run several times, with 2 graphs per client being displayed to show the consistency of response for each client.

1 The length of the test runs were determined to be sufficient for proving the capability of A2D2V2

For testing and performance results gathering, C1 was a client of S1 and C2 was a client of S2. A1, A2 and A3 attacked both S1 and S2 simultaneously. Stacheldraht allows you to do this by setting the IP addresses of the machines you want to attack. During test #4 the command to run the TCP SYN attack from Stacheldraht looked as follows:

stacheldraht(status: a!3 d!0)>.showalive

waiting for ping replies...

showing the alive bcasts... ***---> shows the active attack agents***

192.168.16.1

192.168.11.2

192.168.11.3

----

alive bcasts: 3

stacheldraht(status: a!3 d!0)>

****

.msyn 192.168.13.1:192.168.15.1 ***---> shows the target addresses for receiving attack***

The ICMP and UDP flood tests that were a part of the original A2D2 masters work were not run as part of the performance analysis done for this project. None of the code that mitigated these attacks has changed and those attacks are actually managed via the CBQ and firewall rules applied. One other change from the A2D2 system is that with the A2D2V2 test bed the attackers are within the same subnet. The A2D2 setup allowed for full subnet blocking of attack traffic. With A2D2V2 the setup allows for both legitimate clients and attackers to be within the same subnet. So, full subnet blocking as provided in A2D2 is not appropriate. This feature was disabled and attackers are blocked on IP address only. The Stacheldraht attack tool randomly chooses IP addresses as the source of the attack packets and thus could choose the IP address of a legitimate client in the A2D2V2 test bed. This was considered and allowed. The performance data does show the consequence of this decision and it will be discussed below specific to the set of results.

The first three scenarios were run once, and data gathered one time. For the 4th scenario several sets of runs were performed. It was observed that C1 and C2 reacted consistently for each run. The data shown below is for 2 of the runs for test #4.

The traffic type that is used and measured for the above scenarios is TCP via the *tcp_snd* and *tcp_rcv* programs developed for this project. TCP was chosen so that both incoming and outgoing traffic would be measured.

With A2D2 the goal was to minimize the attack affect on the servers, and thus on the legitimate clients by allowing them to operate at a steady state, but sub-optimal perfor-

mance. The test results for A2D2 indicate that this goal was met. For A2D2V2 the goal is slightly different. There are, in fact, several goals for A2D2V2:

6. To mitigate the attack affect on the legitimate clients much like was seen in A2D2.
7. To validate the enterprise effectiveness of the A2D2V2, software implementation utilizing IDIP, with regard to attack response.
8. To show that even clients who are in a subnet with no IDIP enabling and no attack detection or mitigation mechanism, and that are affected by a DDoS attack within its path, can reap the benefits of the A2D2V2 enterprise network cooperation when the attackers are stopped farther up in the network configuration.
9. To try to provide sustained, if sub-optimal performance for both clients in the A2D2V2 network with the full attack mitigation activated.

9.3 RESULTS ANALYSIS

Figures 9.3.1 – 9.3.8 show the test results per the test scenarios performed above. A program is run on each client that reads the data from /proc/net/dev/<interface> to read the number of packets sent and received. This is then calculated every second for the period of time in the plots shown.

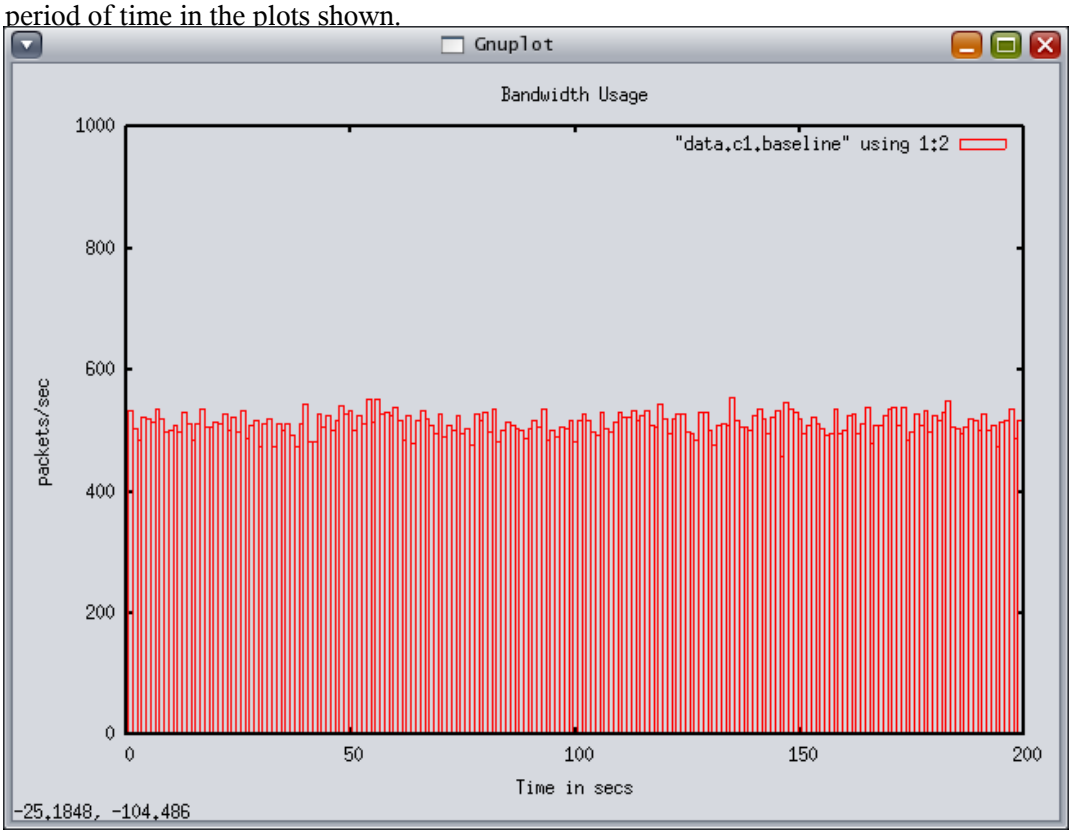


Figure 9.3.1 Client 1 baseline packet rate, Test #1

Figure 9.3.2 Client 2 baseline packet rate, Test #1

This test just sets the baseline traffic expected from the client systems in the A2D2V2 network. What is important to note about these is that C1 has a much more stable sustained rate of about 550 packets per second throughput. Where as C2 is much more inconsistent even in normal client/server activity. This is likely due to several factors: 1) C2 appears more slow than C1. This is anecdotal data but it is an observation seen over many test runs, re-configuration and rebooting of C2 2) The routers that are supporting the link between C2 and S2 are different in their speed and memory size. R102 is the older HP Vectra model which is much slower and has much less memory than the other routers. C1 is on a path with 2 fast routers which helps it sustain a steady rate.

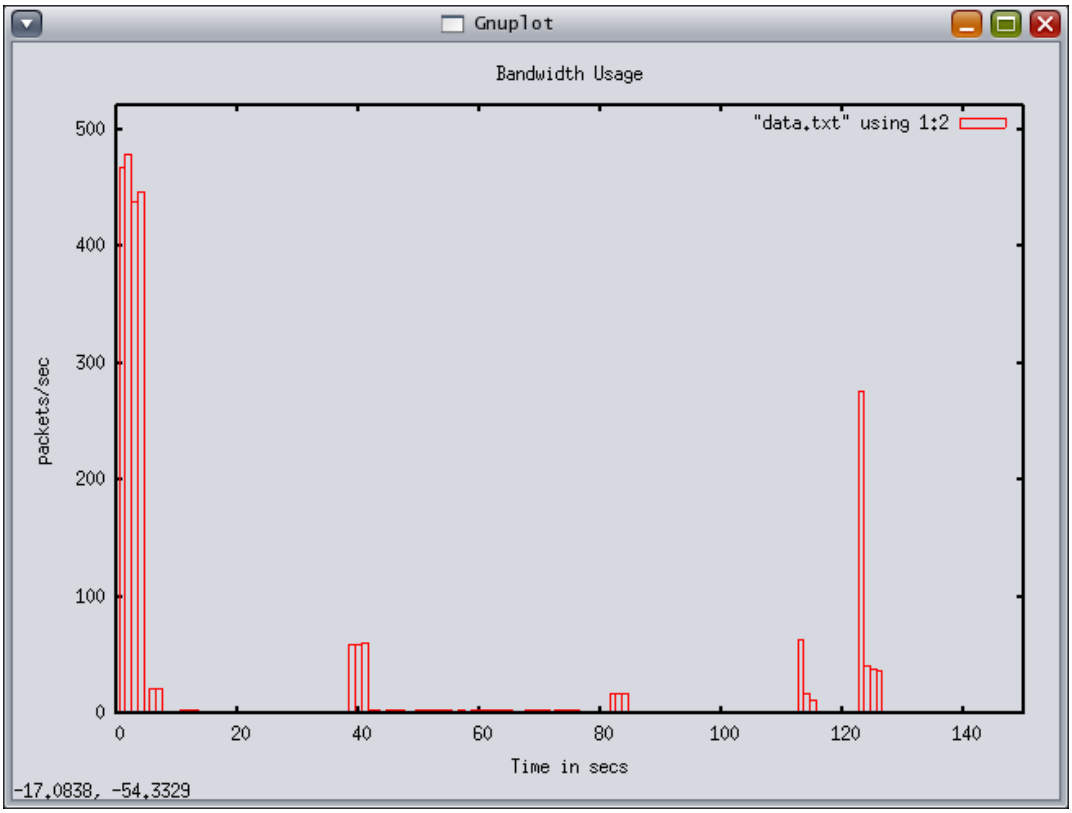


Figure 9.3.3 Client 1 baseline packet rate under attack, no attack mitigation, Test #2

The client under attack with no attack mitigation has almost no packet activity at all.

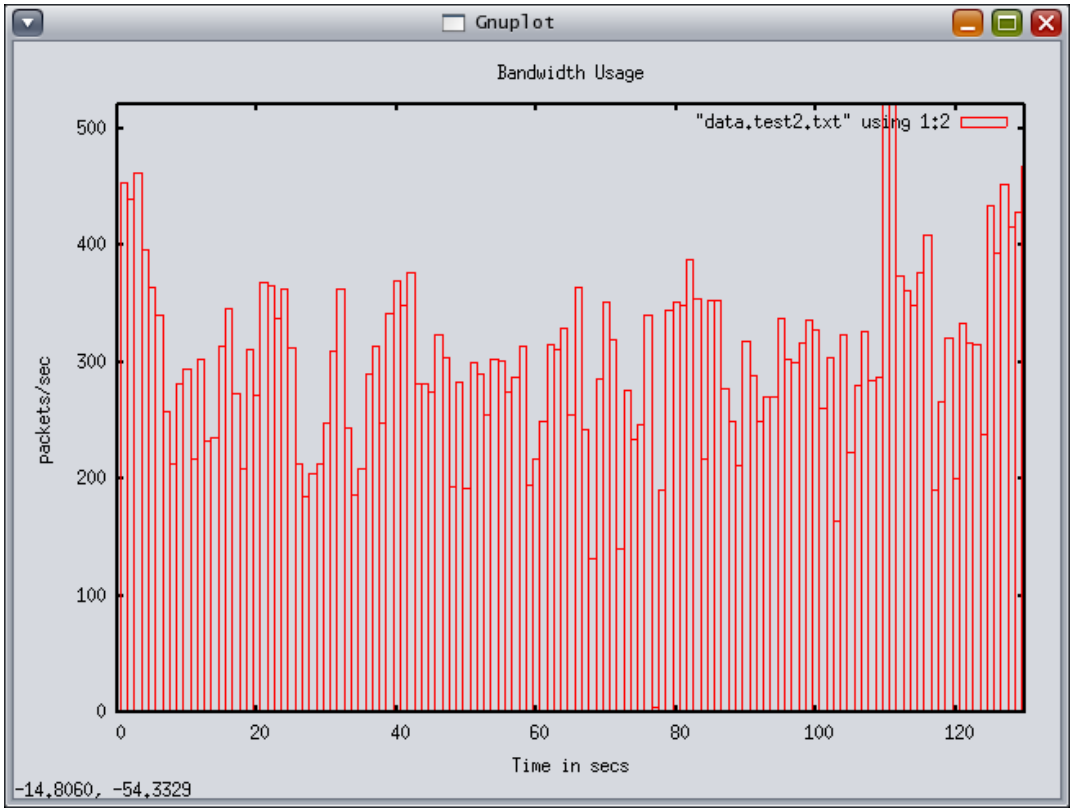


Figure 9.3.4 Client 1 packet rate under attack, 2-LAN full cooperative attack mitigation,Test #3

In this test scenario only R99 and R97 are IDIP enabled, and attacks are only coming from the 11.x subnet. C1 realizes very little performance degradation. There are short bursts of small performance losses due the removal of subnet blocking during the attack,and since 2 of the attackers are within C1's subnet. Overall the client traffic is running at a reasonably steady state however, lower than the baseline run of 550 pkts/second. This is expected and intentional. With A2D2V2 as with A2D2 the class based queueing as described in section 7.4 limits the amount of normal traffic coming through for each interface. The idea is to allow a steady state of performance even during an attack while not starving out other legitimate clients.

The performance seen is due to several factors:

1. R99 is only tracing traffic coming in from the 11.x subnet due to the limited attack, thereby reducing the processing overhead required to do this tracing.
2. The attack traffic is lessened allowing R99 more general processing time.
3. The cooperative nature of A2D2V2 in notifying R97 thereby pushing back the attack nearer the source, and the subsequent response by R97 to limit the packets coming from the attack traffic from within its own subnet reduces the load on R99 considerably.

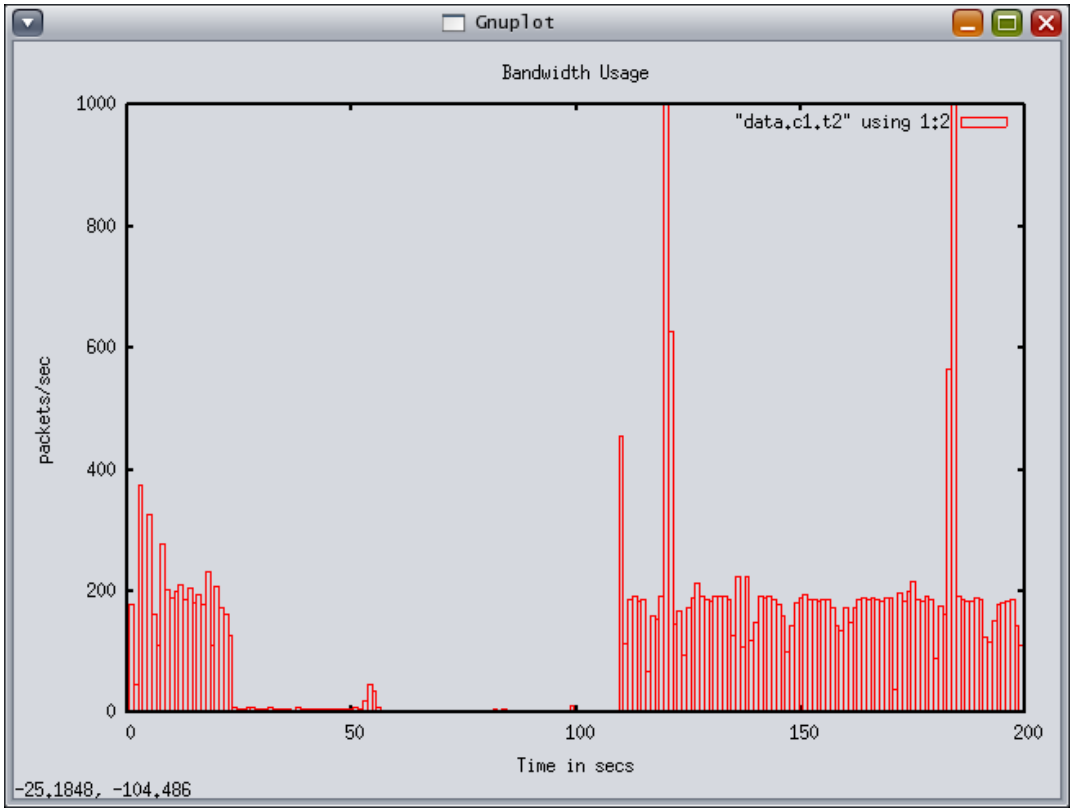


Figure 9.3.5 Client 1 packet rate under attack, enterprise wide attack mitigation, Test #4, a

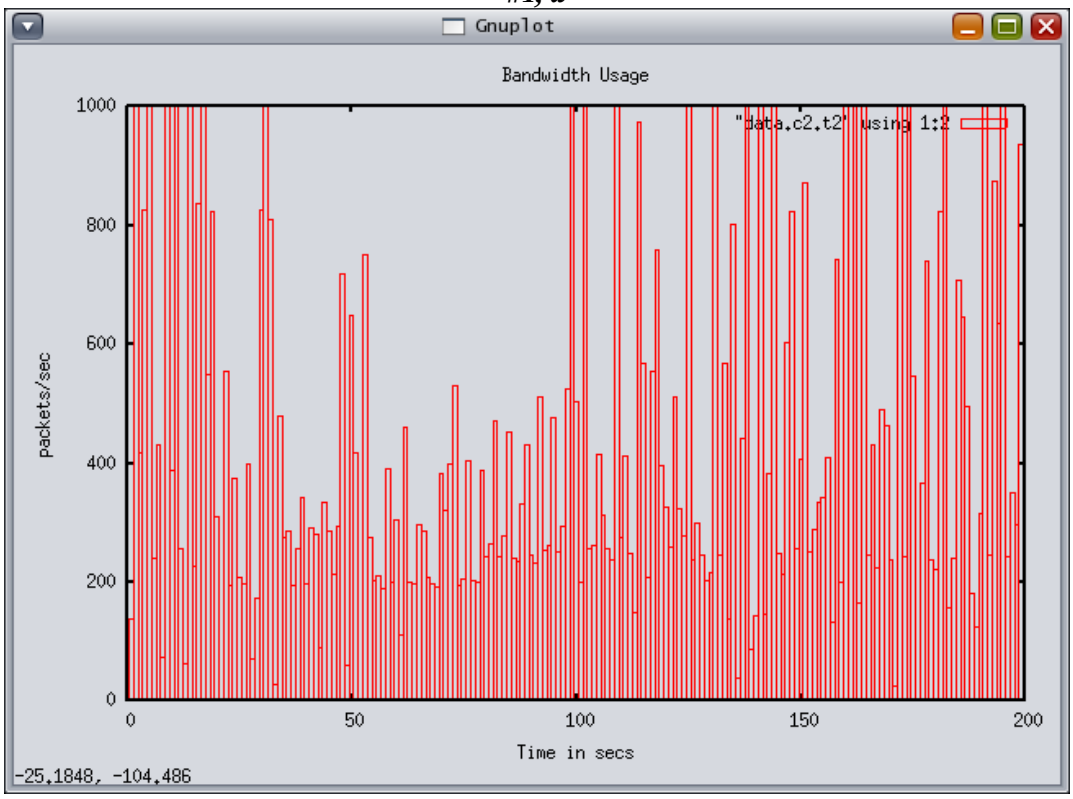


Figure 9.3.6 Client 2 packet rate under attack, enterprise wide attack mitigation, Test #4, a

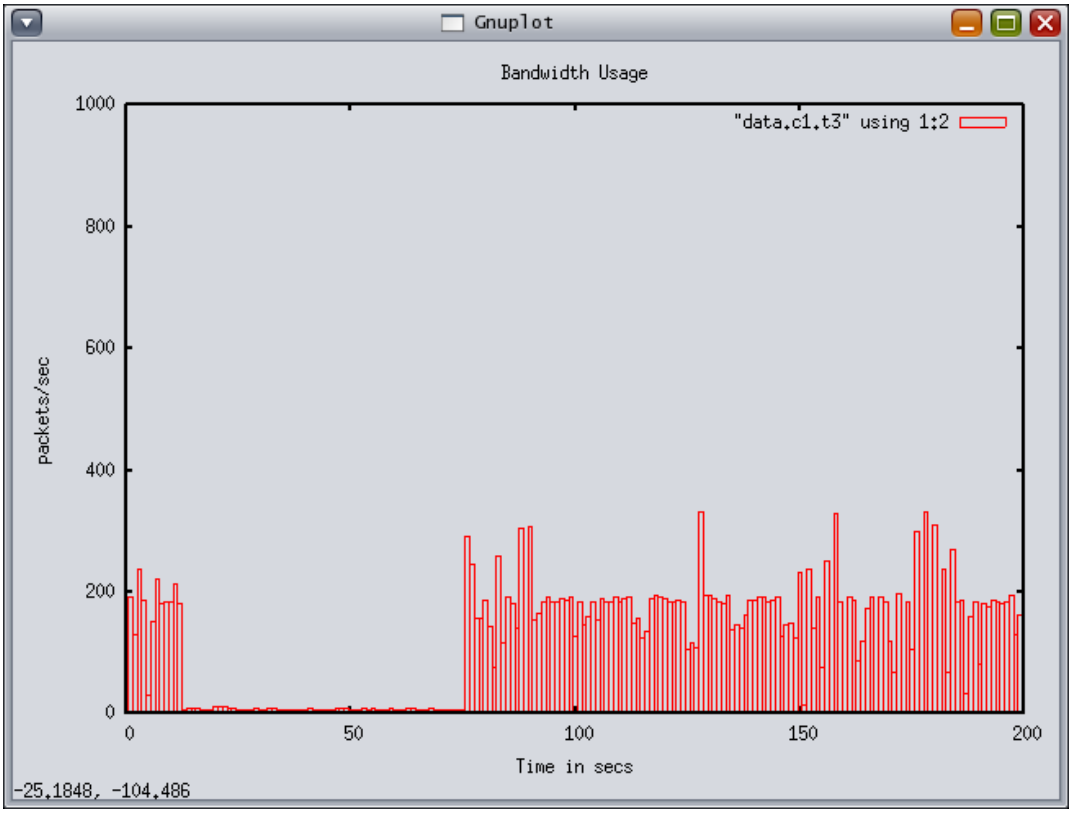


Figure 9.3.7 Client 1 packet rate under attack, enterprise wide attack mitigation, Test #4, b

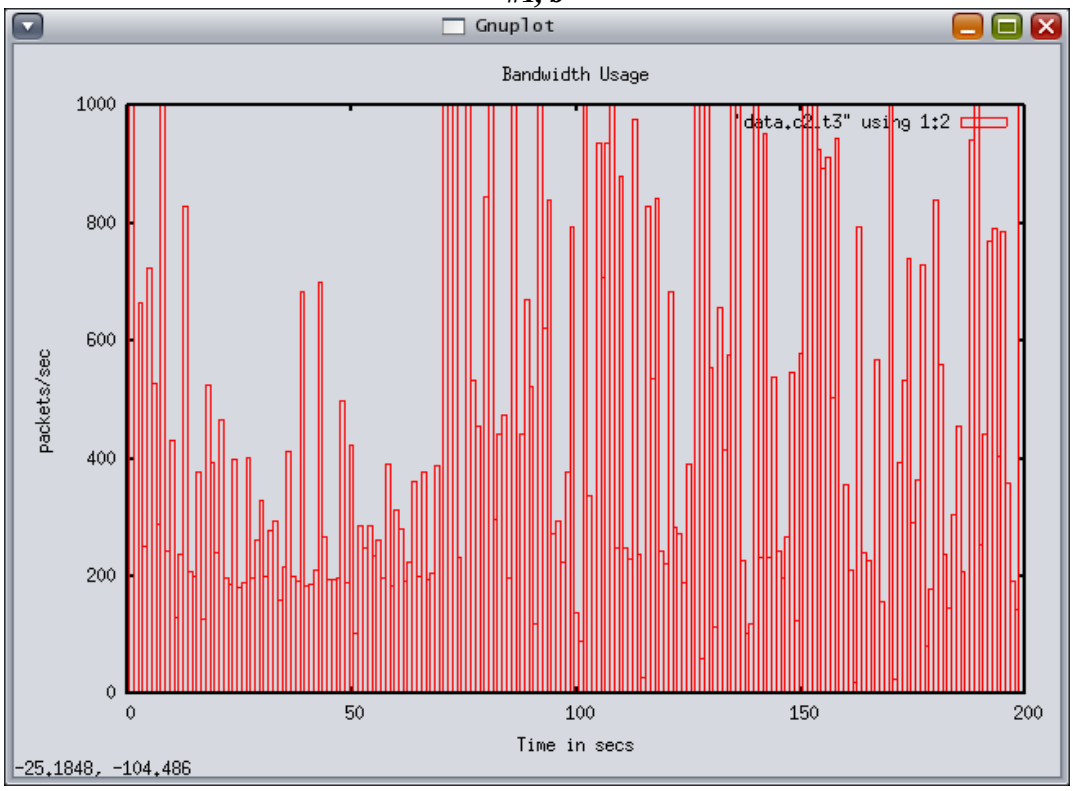


Figure 9.3.8 Client 2 packet rate under attack, enterprise wide attack mitigation, Test #4,b

Clearly the data shows that both clients have some initial and subsequent sustained loss of performance from baseline during the full attack. Specifically at time t + approx. 20 seconds when the attack was started. C1 shows a longer initial performance degradation in all runs of a full attack and full A2D2V2 attack mitigation.

There are many reasons for what is observed with regard to C1 during an attack:

1. R99 must do the requested dynamic tracing prior to applying any rate limiting rules to itself from the time it receives the initial notification that an attack is underway. It searches the static route table for the routers associated with the attack traffic, 1st choosing the router that is in the direct line of the initial attackers, and sends the flood indicated message to each of these routers. Therefore the upstream router in the direct line of attack will apply any attack mitigation rules first.
2. R99 has additional processing overhead with the dynamic tracing process running which limits its ability to respond to attack mitigation rules being sent from the snort IDS.
3. The attack agent A3 is much faster than the attack agents A1 and A2. It was observed that A3, thus the 16.x subnet addresses, were almost always first in terms of a flood being detected by the Snort IDS. With this in mind the R102 router would have been notified first by R99 since it is in the 16.x path, and it would have applied the rate limiting rules first, therefore resulting in a much shorter time for the attack to run from the 16.x subnet.
4. There are 2 attack agents in the path of R97, and R99. This results in a higher number of attack packets going through R97 and R99. Looking at tables 9.3.9a-c below, which show the resultant Linux *iptables* rules after the test runs, you will see a higher number of 11.x addresses for which the rate limiting rules are applied.

Another data point to discuss is the fact that each client continues to experience some short bursts of performance degradation during the whole attack, although C1 is able to achieve a more sustained performance. This is likely due to the removal of the subnet blocking rules that were applied in A2D2, which was intended to better manage a steady state of client performance. With A2D2V2, as new attackers are found they are stopped and more of the legitimate traffic is allowed to proceed, even from within the same subnet. The degradation is not considered to be significant though and good progress is being made.

C1 is able to achieve approx. 50% of its original performance, which is similar to the results seen with A2D2. C2 has much higher levels of overall activity which is surprising given that its speed during normal operations is slower than C1. This is somewhat unexpected, but if you look at the shading that is most prominent in the graphs above, you see that it averages about the same packets per second as C1 during an attack. Some of this data is unexplained except to note that C2's baseline is also erratic in terms of the number of packets per second it transmits so this outcome during an attack is not wholly unexpected.

A critical data point to note is that C2 is being served by S2 which is attached to router R98. Remember from above that R98 is not IDIP enabled in any way and is not participating in any of the A2D2V2 attack mitigation strategies. Also, remember that A1, A2 and A3 are attacking both the S1 and S2 servers. S1 is running the IDIP enabled Snort IDS. S2 does not have any attack detection mechanism running so must bear the full brunt of any attack coming in.

With the above data in mind, we note that without the cooperative affects of the enterprise wide attack response from the A2D2V2 system, C2 would be starved out during the attack. If R102 was not notified of the attack coming from the 16.x subnet to S1 then all attack traffic coming from the 16.x subnet would continue to be allowed through R102 to S2. Even if the local attack response of A2D2 was in place, that is R99 stopped the attack traffic at its doorstep and S1 was relieved, S2 would still bear the full weight of the attack. This is a critical point and the main goal of the A2D2V2 system. If we only notified R99 of the attack as was done in the A2D2 system the legitimate client C2 would reap none of the benefits of the attack response. By also notifying the upstream routers of this attack other clients can be helped in the event of an attack. The fact that C2 recovers as well as it does and is able to maintain a state of reasonable performance is the true measure of the successful outcome of the A2D2V2 system.

The A2D2V2 attack response notifies the upstream routers of attacks to any machine from any attacker. In this way it is generic, that is R102 responded to the attack notification by limiting the rate for the 16.x attackers, with no concern about who they are attacking. It is simply at attack which is coming from within a subnet it serves. In this way the attack is stopped in a more generic way than with A2D2. This type of cooperation helps protect networks in an enterprise even if they are not protected from within.

It is important to note that the class based queueing and *iptables* rules applied to each router/firewall prior to the attack starting is the reason for the lower initial packet rate from baseline, seen in the test results above in figures 9.3.5 - 9.3.8.

During the attacks a set of times were taken for each router to determine the average response times after an attack notification was received and subsequent attack mitigation started. These were started when router R99 received the first notification from the Snort IDS that an attack was detected was measured. The measurements were taken as follows:

- Start time when R99 received first notice of attack and started dynamic tracing
- Time when R99 send out first attack notification message to upstream router
- Time when each upstream router received first notification of attack

The idea was to get an idea of the average response time for each router during the attacks. Three separate runs were monitored and these were averaged as shown below in table 9.3.1 below. However, it is important to note that this data is specific to this test bed. The result of the time it would take for routers to notify upstream routers of an attack would vary greatly on the Internet. The amount of traffic, the number of hops between the victim and the attackers network, and the length of time to trace the source of the attack

are all factors in this response time. Since no two computers are in perfect clock sync the times taken were done using a stop watch.

Time 0 is considered the start time. Subsequent deltas are show as T + X where X indicates the delta in time from time 0.

Event	***Time***
R99 Receives first attack notification and starts tracing	0
R99 Sends out first attack notification to upstream router R102	T + 6 seconds
R102 Receives attack notification from R99	T + 9 seconds
R99 Sends out first attack notification to R97	T + 62 seconds
R97 Receives first attack notification from R99	T + 64 seconds
R99 Applies first attack rule to itself	T + 65 seconds

Table 9.3.1 Router response times during attack

Some observations about this data::

1. R99 does not apply any rules to itself until it has done tracing on all interfaces and all upstream notifications. This slows down is response time to itself considerably and accounts for a large part of C1's performance degradation.
2. The deltas for when R97 receives its first notification of attack from time 0 varied greatly in the 3 runs measured. They were: 56 seconds, 30 seconds and 61 seconds. It is unclear why there is this discrepancy. One theory is that depending on where in the attack cycle R99 started its tracing it could take more or less time to trace all interfaces based on the load it was facing from attack traffic itself. It seemed that when R99 received very early notification of an attack from the Snort IDS the time it took to do the tracing was minimized thus allowing for faster notifications. Another thought is simply that the link between R99 and R97 appears to be somewhat less reliable and resulted in delays getting the notification across the wire. A final thought is that since R97 is on the route of two of the attackers it had more traffic to deal with initially and could not respond as quickly to an incoming attack notification from R99. It is likely a combination of these factors that contributed to this.

These measurements made clear the limitations in the way the dynamic tracing was implemented for A2D2V2. Blocking R99 from applying rules to itself during the tracing resulted in much slower attack response on this router. This contributed to the slower recovery seen by C1 as shown in figure 9.3.7.

These measurements also show the ability of the cooperative defense to efficiently notify upstream routers of an attack, and to contain the attack in a short period of time. Even the longest interval shown for notifying the last upstream router, in this case, R97, was only 1 minute, 5 seconds in length on average. It would take much longer for system administrators to manually intervene to stop attack traffic. Particularly when the attack is distributed and not necessarily contained within their administrative domain. Automatic coordination means that the system administrators do not have to try to figure out who to contact to shut off an attack, or even to trace where the attack is coming from.

Additional supporting data for the C2 client traffic seen above is the *iptraf* output in table 9.3.2 below run on S2 when running the full A2D2V2 system:

```
Wed Jul  5 14:13:05 2006; ******** Detailed interface statistics
started ********

*** Detailed statistics for interface eth0, generated Wed Jul  5
14:18:52 2006

Total:  1565701 packets, 210432861 bytes
        (incoming: 716189 packets, 45786214 bytes; outgoing:
849512 packets, 164646647 bytes)
IP:     1565701 packets, 186996595 bytes
        (incoming: 716189 packets, 34243116 bytes; outgoing:
849512 packets, 152753479 bytes)
TCP: 1565433 packets, 186978371 bytes
        (incoming: 715921 packets, 34224892 bytes; outgoing:
849512 packets, 152753479 bytes)
UDP: 0 packets, 0 bytes
        (incoming: 0 packets, 0 bytes; outgoing: 0 packets, 0
bytes)
ICMP: 268 packets, 18224 bytes
        (incoming: 268 packets, 18224 bytes; outgoing: 0 packets,
0 bytes)
Other IP: 0 packets, 0 bytes
        (incoming: 0 packets, 0 bytes; outgoing: 0 packets, 0
bytes)
Non-IP: 0 packets, 0 bytes
        (incoming: 0 packets, 0 bytes; outgoing: 0 packets, 0
bytes)
Broadcast: 0 packets, 0 bytes

Average rates:
  Total:        4851.48 kbits/s, 4512.11 packets/s
  Incoming:     1055.59 kbits/s, 2063.95 packets/s
  Outgoing:     3795.89 kbits/s, 2448.16 packets/s
```

```
Peak total activity: 7028.49 kbits/s, 8184.80 packets/s
Peak incoming rate: 2118.14 kbits/s, 4075.20 packets/s
Peak outgoing rate: 5706.25 kbits/s, 4901.00 packets/s
IP checksum errors: 0

Running time: 347 seconds
Wed Jul  5 14:18:52 2006; ******** Detailed interface statistics
stopped *******
```

Table 9.3.2 – iptraf output from S2 server during test run

The TCP traffic shown above is 849512 packets for 347 seconds. The average TCP packet rate is 2448 packets per second. This is both incoming and outgoing packets. It is hard to tell from the graph above, but in looking at the raw output for the plotting data C2 shows a large variation in packets per second. An average of 2448 per second, both outgoing and incoming is not unreasonable based on this data. A snipped of this log is included here to show this:

```
189 1699
190 254
191 441
192 770
193 792
194 404
195 787
196 358
197 191
198 143
199 1153
200 293
201 292
202 213
203 38
204 1085
243 245
244 1183
245 305
246 492
247 387
248 361
249 721
250 524
251 632
252 1659
```

There are several large spikes in packet activity during the test run. It is important to keep in mind that *iptraf* logging processes are very resource intensive. This puts load on the system which could also account for some of the erratic behavior in packet spikes on C2 during the test run. This data is presented for completeness only but should be viewed with caution. The results shown in the graphical form above indicate the real performance of the clients during the non stop attacks.

After each run of the full attack and complete attack mitigation the iptables was checked to see the rules that were applied.

R102 iptables -v -L after test run,4a:

```
Chain INPUT (policy DROP 0 packets, 0 bytes)
pkts bytes target     prot opt in  out     source   destination
0     0 level3     all  --  any    any     192.168.11.72 anywhere
0     0 level3     all  --  any    any     192.168.11.48 anywhere
0     0 level3     all  --  any    any     192.168.11.114 anywhere
0     0 level3     all  --  any    any     192.168.11.51 anywhere
0     0 level3     all  --  any    any     192.168.11.18 anywhere
0     0 level3     all  --  any    any     192.168.11.134 anywhere
3544  450K ACCEPT     all  --  any    any     anywhere anywhere

Chain FORWARD (policy DROP 0 packets, 0 bytes)
pkts bytes target     prot opt in   out    source     destination
0     0 level3     all  --  any     any    192.168.11.72 anywhere
0     0 level3     all  --  any    any     192.168.11.48 anywhere
0     0 level3     all  --  any    any     192.168.11.114 anywhere
0     0 level3     all  --  any    any     192.168.11.51 anywhere
0     0 level3     all  --  any    any     192.168.11.18 anywhere
0     0 level3     all  --  any    any    192.168.11.134 anywhere
1799K  253M ACCEPT     all  --  any    any     anywhere anywhere

Chain OUTPUT (policy DROP 0 packets, 0 bytes)
pkts bytes target     prot opt in     out   source destination
3487  363K ACCEPT     all  --  any    any  anywhere anywhere

Chain level0 (0 references)
pkts bytes target     prot opt in     out   source destination
0     0 DROP       all  --  any    any     anywhere   anywhere

Chain level1 (0 references)
pkts bytes target     prot opt in     out  source  destination
0     0 DROP       all  --  any    any     anywhere anywhere

Chain level2 (0 references)
pkts bytes target     prot opt in     out   source destination
0     0 ACCEPT     all  --  any    any      anywhere   anywhere
```

```
        limit: avg 50/sec burst 5
0     0 DROP       all  --  any    any     anywhere    anywhere

Chain level3 (14 references)
pkts bytes target     prot opt in     out     source
destination
1243 1861K ACCEPT     all  --  any    any     anywhere anywhere
      limit: avg 151/sec burst 5
500  749K DROP       all  --  any    any     anywhere anywhere
```

Table 9.3.3a – R102 iptables -v -L output

The important thing to note from the above output is that there were many rate limited and dropped packets in the Level 3 chain. The reasons we do not have the L1 and L2 chains populated is that the attack was fairly short in duration(approx. 3 ½ minutes) and we only have 3 attackers in this test bed attacking 2 servers at one time. Compare that with 5 attackers targeting one server in the A2D2 network.

Also, important to note pertaining the differences between this output and the output shown in Table 4 of the A2D2 analysis, is that for A2D2 subnet blocking rules were in affect during the attack. For A2D2V2 subnet blocking was disabled due to the address configuration chosen for the A2D2V2 test bed. The A2D2V2 test bed allows for legitimate clients in both of the attack subnets.

R99 iptables -v -L after test run, 4a:

```
Chain INPUT (policy DROP 25 packets, 3604 bytes)
pkts bytes target     prot opt in     out     source destination
0     0 level3     all  --  any    any     192.168.11.72 anywhere
0     0 level3     all  --  any    any     192.168.11.48 anywhere
0     0 level3     all  --  any    any     192.168.11.114 anywhere
0     0 level3     all  --  any    any     192.168.11.51 anywhere
0     0 level3     all  --  any    any     192.168.11.18 anywhere
0     0 level3     all  --  any    any     192.168.11.134 anywhere
512K  134M ACCEPT     all  --  any    any  anywhere anywhere

Chain FORWARD (policy DROP 0 packets, 0 bytes)
pkts bytes target     prot opt in out     source destination
0     0 level3     all  --  any    any     192.168.11.72  anywhere
0     0 level3     all  --  any    any     192.168.11.48  anywhere
0     0 level3     all  --  any    any     192.168.11.114 anywhere
0     0 level3     all  --  any    any     192.168.11.51  anywhere
0     0 level3     all  --  any    any     192.168.11.18  anywhere
0     0 level3     all  --  any    any     192.168.11.134 anywhere
894K  170M ACCEPT     all  --  any    any     anywhere anywhere

Chain OUTPUT (policy DROP 1 packets, 52 bytes)
pkts bytes target     prot opt in     out     source  destination
286K  102M ACCEPT     all  --  any    any     anywhere anywhere
```

```
Chain level0 (0 references)
pkts bytes target     prot opt in     out     source anywhere
0     0 DROP       all  --  any    any     anywhere  anywhere

Chain level1 (0 references)
pkts bytes target     prot opt in     out     source destination
0     0 DROP       all  --  any    any     anywhere anywhere

Chain level2 (0 references)
pkts bytes target     prot opt in     out     source destination
0     0 ACCEPT     all  --  any    any     anywhere anywhere
        limit: avg 50/sec burst 5
0     0 DROP       all  --  any    any     anywhere anywhere

Chain level3 (14 references)
pkts bytes target     prot opt in     out     source anywhere 0
0 ACCEPT     all  --  any    any     anywhere anywhere
      limit: avg 151/sec burst 5
0     0 DROP       all  --  any    any     anywhere anywhere
```

Table 9.3.3b - R99 iptables -v -L output

For R99 there were no drops, and a lot of accepts. The reason for this is straightforward, the upstream routers R102 and R97 did the bulk of the packet processing once they were notified of an attack. The rules state that the sustained packet rate for L3 is 151 per second. We never reached this level once the upstream routers were notified which means most of the burden to stop the attack was pushed to the routers closest to the source of the attack.

R97 iptables -v -L after test run, 4a:

```
Chain INPUT (policy DROP 1 packets, 100 bytes)
pkts bytes target     prot opt in  out     source   destination
0     0 level3     all  --  any    any    192.168.11.115 anywhere
0     0 level3     all  --  any    any     192.168.11.74 anywhere
0     0 level3     all  --  any    any     192.168.11.107 anywhere
0     0 level3     all  --  any    any     192.168.11.139 anywhere
0     0 level3     all  --  any    any     192.168.11.112 anywhere
0     0 level3     all  --  any    any     192.168.11.60  anywhere
4643  696K ACCEPT     all  --  any    any   anywhere      anywhere

Chain FORWARD (policy DROP 0 packets, 0 bytes)
pkts bytes target     prot opt in     out     source   destination
0     0 level3     all  --  any    any     192.168.11.115 anywhere
```

```
0     0 level3     all  --  any    any     192.168.11.74 anywhere
0     0 level3     all  --  any    any     192.168.11.107 anywhere
0     0 level3     all  --  any    any     192.168.11.139 anywhere
0     0 level3     all  --  any    any     192.168.11.112 anywhere
0     0 level3     all  --  any    any     192.168.11.60 anywhere
5530K  368M ACCEPT     all  --  any    any     anywhere anywhere

Chain OUTPUT (policy DROP 1 packets, 132 bytes)
pkts bytes target     prot opt in     out     source destination
5009  544K ACCEPT     all  --  any    any     anywhere anywhere

Chain level0 (0 references)
pkts bytes target     prot opt in     out     source dest
    0     0 DROP       all  --  any    any     anywhere anywhere

Chain level1 (0 references)
 pkts bytes target     prot opt in     out     source dest
    0     0 DROP       all  --  any    any     anywhere anywhere

Chain level2 (0 references)
 pkts bytes target     prot opt in     out     source destination
0     0 ACCEPT     all  --  any    any     anywhere anywhere
        limit: avg 50/sec burst 5
0     0 DROP       all  --  any    any     anywhere anywhere

Chain level3 (14 references)
pkts bytes target     prot opt in     out     source destination
2233 3349K ACCEPT     all  --  any    any     anywhere   anywhere
        limit: avg 151/sec burst 5
766   1149K DROP       all  --  any    any     anywhere anywhere
```

Table 9.3.3c - R97 iptables -v -L output

Again, R97 shows a large number of accepts and drops in the Level 3 chain. This shows, along with the data for R102, that the attack was pushed to the upstream routers where the packets were managed.

A general note about the data in the above tables, it clearly shows many IP addresses are being managed via the *iptables* mechanism for all routers in the A2D2V2 test bed during and after an attack. You might ask why we have IP addresses that are not a part of the original A2D2V2 test bed setup appearing and why IP addresses that are not within the subnet a router is attached are showing up in that routers *iptables*? This is due to the way the Stacheldraht tool mounts an attack an spoofs the source IP addresses as discussed in section 9.2. This is fully expected behavior with the IP spoofing that Stacheldraht utilizes during its attack sequences.

During each run some of the IDIP Message/Discovery Coordinator output was captured. Below is some of the output from the R99 IDIP Message/DC run:

```
idip_firewall_receiver.c do_trace_request: UNDER ATTACK:<-- trace
request being processed
idip_firewall_receiver.c do_trace_request: from source
192.168.16.133
idip_firewall_receiver.c do_trace_request: on interface eth3
idip_firewall_receiver.c do_trace_request: number of packets 308
idip_firewall_receiver.c do_request: message received FLOOD
DETECTED on r993 from 192.168.16.133 (THRESHOLD 50 connections
exceeded in 10 seconds)<--creation of IDIP FLOOD message
idip_firewall_receiver.c do_request: Connected to rate limiter
idip_firewall_receiver.c do_request: Sent msg FLOOD DETECTED on
r993 from 192.168.16.133 (THRESHOLD 50 connections exceeded in 10
seconds) to rate limiter
idip_firewall_receiver.c do_trace_request: alertmsg sent to
192.168.14.102: FLOOD DETECTED on r993 from 192.168.16.133
(THRESHOLD 50 connections exceeded in 10 <-- alertmsg sent to
upstream router, 14.102
seconds)
idip_firewall_receiver.c do_trace_request : Checking for other
upstream routers
to notify
idip_firewall_receiver.c do_trace_request(): alertmsg sent to
192.168.12.97: FLOOD DETECTED on r993 from 192.168.16.133  <--same
message sent to other upstream router, 12.97
```

Table 9.3.4 IDIP message output

This test run output shows the sequencing of actions when a flood is detected by the IDIP Snort IDS, and subsequent tracing actions invoked and additional flood detected. It also shows the discovery of the upstream routers and subsequent notification to those routers of the newly discovered flood.

10. LESSONS LEARNED

During the course of this project I learned a great deal. In the end I have a much better understanding of routing, DDoS intrusions, Linux firewalls, IDIP, tracing packets, class based queuing, Snort, network traffic measurement tools and push back of DDoS attacks.

10.1 Network Routing Tables

During setup of the A2D2V2 test bed there was difficulty encountered in setting up the communication between each of the subnets separated by the firewall/routers systems. The default routing as defined when the systems were brought up was not sufficient for all of the hosts to communicate past their router boundaries. This is due to the fact that the default routers only know about the subnets attached within one link hop on each interface. To manage the complexity of the A2D2V2 test bed manual routing tables needed to be added for each of the routers in the test bed. With Linux using the '*route add*' command line utility will update the routing tables but this data will not be persistent across reboot. You must go in the system network utility and manually save the configuration for it to be retained.

The routing table for the R99 router looked like:

```
[sjelinek@r99 ~]$ netstat -rn
Kernel IP routing table
Destination     Gateway         Genmask         Flags   MSS Window
irtt Iface
128.198.61.0    0.0.0.0         255.255.255.128 U         0 0
0 eth2
192.168.16.0    192.168.14.102  255.255.255.0   UG        0 0
0 eth3
192.168.15.0    192.168.14.98   255.255.255.0   UG        0 0
0 eth3
192.168.14.0    0.0.0.0         255.255.255.0   U         0 0
0 eth3
192.168.13.0    0.0.0.0         255.255.255.0   U         0 0
0 eth1
192.168.12.0    0.0.0.0         255.255.255.0   U         0 0
0 eth0
192.168.11.0    192.168.12.97   255.255.255.0   UG        0 0
0 eth0
169.254.0.0     0.0.0.0         255.255.0.0     U         0 0
0 eth3
0.0.0.0         128.198.61.1    0.0.0.0         UG        0 0
0 eth2
```

Table 10.1.1.1 R99 Routing Table

10.2 iptables FORWARD Chain firewall rules

Part of learning about Linux *iptables* and the subsequent firewall rules had to be done to understand how A2D2 worked. However, since the test bed setup for A2D2V2 is much different than that which was used in A2D2 this necessitated a different implementation of the *iptables* FORWARD chain rules for setting up the firewalls. Research on *iptable*s chain rules was done to get a clear understanding of how all of this works.

What is different? In the A2D2 test bed as shown in figure 7.4.1 it had only one incoming router with one outgoing interface in to the private LAN. This meant that the firewall rules for the FORWARD chain which allows packets to be forwarded by the router to destinations within its subnet were only defined on that one output interface. With A2D2V2 there are several routers with multiple input and output interfaces which had to be accounted for in the firewall rule setup. For R99 the OUTPUT chain has to account for two interfaces, both the eth1 and eth3 interfaces. A subset of these rules is shown:

Rules for the eth1 interface:

```
#mark incoming mail traffic from smtp and pop3 with mark value 2
$IPTABLES -A FORWARD -p tcp -o $OUT1 -s 0/0 --dport smtp -d 0/0 -t
mangle -j MARK --set-mark 2
  $IPTABLES -A FORWARD -p tcp -o $OUT1 -s 0/0 --dport pop3 -d 0/0
-t mangle -j MARK --set-mark 2

  ...
```

Rules for the eth3 interface:

```
#mark incoming mail traffic from smtp and pop3 with mark value 2
$IPTABLES -A FORWARD -p tcp -o $OUT3 -s 0/0 --dport smtp -d 0/0 -t
mangle -j MARK --set-mark 2
  $IPTABLES -A FORWARD -p tcp -o $OUT3 -s 0/0 --dport pop3 -d 0/0
-t mangle -j MARK --set-mark 2
```

The full CBQ firewall script for R99 is shown in Appendix C.

10.3 Linux Class based queuing

One of the issues discussed in A2D2 was the need for limiting the traffic to a host based on traffic type. The reason this is needed is that when an attack is started and directed at a host, the ability for the DDoS system to respond and get legitimate traffic through to the appropriate hosts is stopped due to the high volume of attack traffic. Class Based Queuing is a mechanism by which you specify, based on speed of connection, a percentage of that connections bandwidth to types of traffic. For A2D2 and A2D2V2 several types of traffic were classified and 'marked' as part of the *iptables* rules setup on each firewall/router. The classes defined were:

- tcp/syn and icmp – class 1
- smtp and pop3 – class 2
- telnet, ftp and ssh - class 3

- www, RealServer and A2D2V2 test server app(on port 7654) – class 4

Each of these classes were marked when a packet came in to the router and this marking was later used to determine the percentage of total bandwidth they were allowed. For A2D2V2 each of the above types were allowed the following percentages of bandwidth:

- tcp/syn and icmp traffic – 5%
- smtp and pop3 traffic – 15%
- ssh, telnet and ftp traffic – 10%
- www , RealServer and A2D2V2 test server traffic – 70%

For A2D2V2, as with A2D2, the application hosted by the server was RealServer along with the *tcp_srv* application used for testing. The rules applied for class based queueing reflect this. If this was an *ftp* download site then it would make sense to allow more of the bandwidth to be assigned to the ftp traffic. These rules are set per site and per policy determined by the applications being served.

10.4 IDIP

As detailed in Sections 4, 5 and 6 a lot of time was spent researching IDIP and implementing the pieces of the protocol to enable A2D2V2 cooperative DDoS tracing and push back. IDIP as a protocol is not difficult to understand, but it is complicated to implement as it has a number of data structures and rules that must be followed to be compliant with the protocols.

The location and layout of the source tree and build instructions are shown in Appendix B. Much of the code for this project is new, in particular the IDIP message and application layer code. Some of the code developed for this project was modifications to existing code provided in the A2D2 project. The IDIP code was all written in C to allow for as much interoperability as possible. However, in heterogeneous environments it is likely that the building of the source will be different. This has only been fully tested using the Linux Fedora Core 5 release, but in theory should operate correctly on any UNIX operating system. The specific implementation of the *iptables* and class based queuing is specific to Linux and cannot be used on other UNIX systems unless supported by them.

10.5 Snort

As part of this project I had to learn about how to install, configure and modify Snort modules. The Snort IDS was used to detect the flood attack and to send the appropriate IDIP messages. For A2D2 Snort was modified to include a new preprocessor module for the detection of flood attacks. I enhanced the reporting module to enable the creation and forwarding of IDIP messages. The initial flood detection message is sent to the UNIX socket on the IDS host. The additions I made to snort were in the alerting mechanism. I wrote a standalone program that intercepts the messages sent by the flood preprocessor to the UNIX socket and formats these messages using IDIP definitions.

10.6 Pushback/Tracing Techniques for DDoS attacks

The technique that I employed for this project with regard to tracing of incoming traffic was *tcpdump.* Details of this are in section 8.2.1. With regard to my first implementation of the *tcpdump.sh* script, I didn't realize that *tcpdump* snoops the interfaces it is monitoring in promiscuous mode, meaning every packet that comes by is counted, even if the packet isn't intended for that router. I also didn't realize that the traffic coming from S2 and C2 would be found using *tcpdump* on R99 via the eth3 interface. The reason for this is straightforward, but it took my testing and noticing that S2 was showing up as an attacker on R99 to understand what was happening. R99, R102 and R98 have interfaces on the same network, the 192.168.14.x network. This means that all packets coming from the C2 client or S2 server would be seen across that network. Since R99 has eth3 on the 192.168.14.x network, *tcpdump* running on this interface would see packets from these two machines, even though they were legitimate client/server communications. In terms of a packet my *tcpdump* program would assume this meant an attack was coming from these machines. The initial testing I did showed both 192.168.15.1(S2) and 192.168.16.1(C2) IP addresses in the *iptables* for R99, R102 and R97 because of this situation. This caused anomalous behavior for the C2 client since the router R102 was blocking some of this traffic in the FORWARD chain.

The implementation of the dynamic tracing for A2D2V2 is autonomous. Once the firewall/router receives the notice that an attack has been detected by the Snort IDS it starts the dynamic tracing and interprets the results independent of what the Snort IDS has found. This is critical to the A2D2V2 goals. To move the attack mitigation farther up in the network away from the attack recipients and closer to the source requires this autonomy. With this in mind, I had two thoughts with regard to handling this within the restrictions of the A2D2V2 implementation:

1. I could have left things as they were and allowed the traffic coming from the legitimate server and client to be counted as attack traffic. This would have resulted in a larger degradation of performance.
2. I considered adding rules to my *tcpdump* invocation to ignore these hosts when counting the number of packets coming to an interface. Thus, they would not be counted as attack traffic. In the real world this might work, and could be applied much like the IDS rules are applied with regard to ignoring specific hosts. This would also result in less performance degradation.

In the end I chose number 2. I added the rule when calling *tcpdump* to exclude the 192.168.16.1 and 192.168.15.1 IP addresses. See Appendix C for the script that invokes *tcpdump* for dynamic tracing. This is an area of much future work however, as discussed in section 11.3.1 Correlation Engine.

The push back of the attack was implemented by identifying the attackers on the upstream routing with the dynamic tracing detailed in Section 8.2.2, and then once identified to consult the static routing information file to identify which router must be notified. This is a very simplistic approach to a very complicated problem. However, in my research I learned many things with regard to current research in the area of IP tracing and

push back. Much of this research is discussed in Section 11 and is considered a large area of further research and development. Without accurate, real-time tracing capability effective push back of the attack is hampered. What this project offers is a proof of concept that the ability to trace the source of an attack and push back the attack to the source succeeds in restoring traffic to legitimate clients in a much shorter time frame and more fully than without it. The analysis of my data in Section 9 discusses this.

11. FUTURE WORK

11.1 EXISTING TECHNOLOGIES THAT COMPETE IN THIS PROBLEM SPACE

As part of the work on this project, research of existing technologies was done. When this project was started, many of these technologies were in the beginning stages and were not available for consideration. With this in mind, a survey of the emerging technologies that are applicable to this type of application are listed in this section.

11.1.1 Intrusion Detection Message Exchange Format(IDMEF)

The purpose of the IDMEF is to define data formats and exchange procedures for sharing information of interest to intrusion detection and response systems[IDMEF]. It is intended to standardize the data format that automated IDS systems can use to report alerts about events of interest. The idea is that this will enable interoperability among commercial, open source and research IDS's. A mix and match of these systems can be used to obtain an optimal IDS implementation.

IDMEF can be utilized many ways. For instance, a single database system could store the results in this standard format for all of the IDS's employed. This would allow the management and data analysis activities to operate on a more global picture. Another way to use this would be to deploy an event correlation mechanism that took all of the alert data, in IDMEF format, from the various IDS systems. This would enable a more sophisticated cross-correlation and cross-confirmation response.

The IDMEF data model is an object oriented representation of the alert data sent by an IDS. Since alert information is inherently heterogeneous, with many differences in the data based on the IDS used. The data model allows for these differences. The object oriented model is naturally extensible.

The IDMEF data model allows for significant flexibility in reporting alert data. Since operating environments vary within a LAN and across the Internet, this flexibility is critical.

The goal of the IDMEF data mode was to provide a standard representation of alerts in an unambiguous manner. The IDMEF implementation was done using the Extensible Markup Language(XML). The IDMEF model is shown in Figure 11.1.1.1

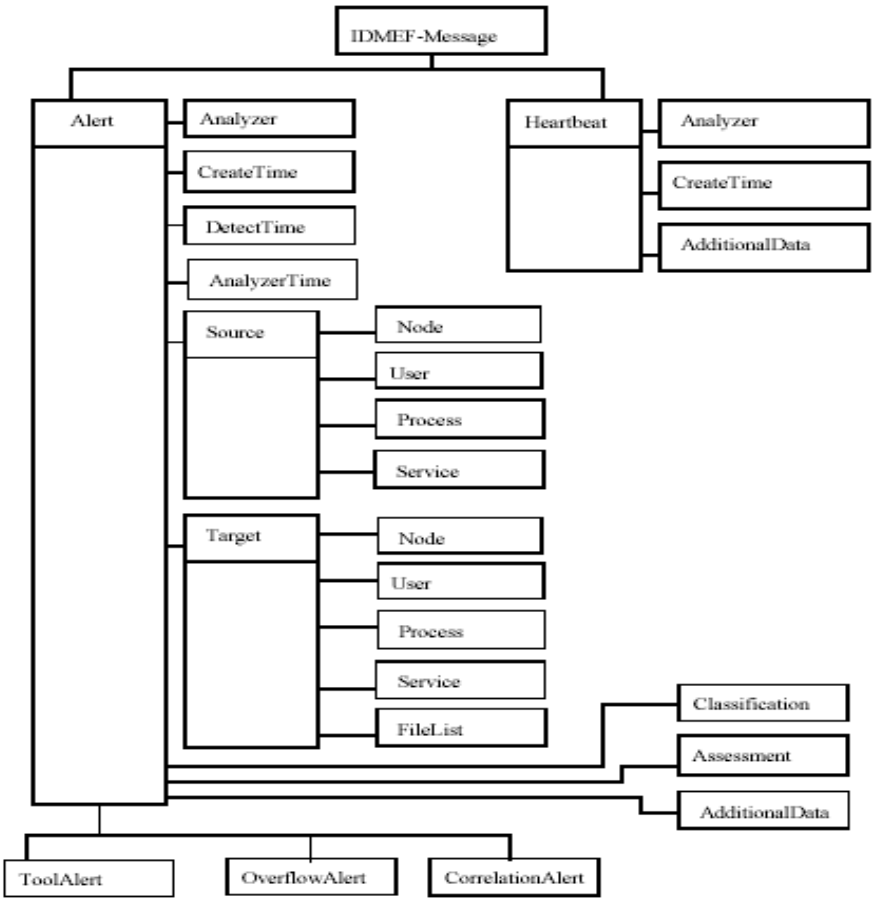


Figure 5. A simplified version of the IDMEF model as of January 30, 2003 [7].

Figure 11.1.1.1 IDMEF Model[GO03]

11.1.1.1 XML

A brief overview of XML will be given in this section.

XML is a simplified version of the Standard Generalized Markup Language(SGML). SGML is a syntax for specifying text markup defined by the ISO 8879 standard[XML].

XML is a metalanguage and enables an application to define its own markup. XML allows the definition of custom markup languages for different types of applications. This is different from HTML in that with HTML there is a fixed set of identifiers with preset meanings that must be adapted for special use.

XML provides both a syntax for declaring document markup and a structure for defining elements and attributes, specifying the order in which they should appear, etc.

11.1.1.2 Why IDMEF is implemented in XML

The details about this are important to this paper, as the discussion later in this section about future work will take this in to consideration.

XML based applications are being used or developed for many purposes. EXEMPT's flexibility makes it a good choice for these disparate applications and that same flexibility makes it a good choice for IDMEF.

XML allows a custom language to be developed for the purpose of describing intrusion detection alerts. It also allows for a standard way to extend this language.

Software tools for processing XML are widely available in both commercial and open-source forms. The ability to parse XML is ubiquitous on any platform.

XML message formats support full internationalization and localization.

XML is free, with no license, no license fees no royalties.

11.1.2 Common Intrusion Specification Language (CISL)

As part of the CIDF, as described in Section 5, a language was developed that could be used to disseminate analysis results and countermeasure directives among intrusion detection and response systems. While this is not specifically an emerging technology as it was available at the time of the start of this project, it is worth noting as a possible alternative or add on to this work.

Much like IDMEF, CISL is a common language in which to exchange IDS information. The idea is that sharing of this information with other IDS systems, potentially in other networks, could enable the more global response to a perceived attack.

Under the CIDF model many components could be operating at one time, all of whom are generating important statistical data that should be shared. CISL aims to define a language that allows for a common understanding and exchange of this data. Much like IDMEF CISL's intent is to describe this data in an unambiguous way.

CISL's solution to this is similar to that used for English. A general language construct, called S-expressions are used. S-expressions are simply recursive groupings of tags and data. S-expressions provide an explicit grouping of two terms. The interpretation of these is left up to the language definition.

11.1.3 Intrusion Detection and Exchange Protocol(IDXP)

IDXP defines a protocol to exchange data between IDS entities. It supports mutual authentication, integrity and confidentiality over a connection oriented protocol[IDXP].

The specification is a Blocks Extensible Exchange Protocol[IDXP]. It provides for exchange of IMDEF messages, unstructured data and binary data between IDS systems. IDXP is an open, published standard.

11.1.4 Intrusion Detection and Exchange Architecture

The intrusion detection and exchange architecture is an opensource project that allows for interpretation of data from many disparate IDS systems. The source is available on http://sourceforge.net. This project presents a unified view of the IDS data translated in to network activity.

XML is used as data transfer and correlation protocol. It is not a standards based solution and is implemented using Java.

11.2 COMPARISONS

IDIP was chosen for this project due to the fact that the IDMEF and IDXP efforts were just beginning at the time this project was started. This said, there are some compelling reasons to use IDIP and perhaps some for not using it as well.

11.2.1 IDIP vs. IDMEF

Both IDIP and IDMEF define data formats and exchange procedures for sharing data from IDS systems to other IDS systems. IDIP goes one step further, by allowing the standardization of the communication between IDS systems and other IDIP nodes. IDIP is a general protocol for which existing IDS systems can be adapted to conform.

Both IDIP and IDMEF enable interoperability between opensource, commercial and research IDS systems. IDIP uses a message protocol for transfer of data, IDMEF uses XML. Both IDIP and IDMEF require additional infrastructure to be placed on each node that is to be enabled.

IDMEF uses XML. XML is a standards based mechanism, and IDMEF has standard XML schema's published for use. This standard allows wider adoption of the protocol.

The original intent of IDIP was that it was to be an open, standard protocol. However, this status changed during the implementation of this project. IDIP is currently being worked on by Telcordia, Inc. and McAfee Software via Network Associates Incorporated Labs. This privatization of IDIP makes it less viable as a long term solution.

IDMEF has some correlation protocol definitions. IDIP has none,and only provides for the ability to send and record trace data. It is important to note that both require a knowledgeable correlation engine to be built to fully utilize the data gathered.

Finally, at the outset of this project, the IDIP documentation was expected to be fully available. As mentioned above, this expectation was changed mid-stream and the only documents currently available are those listed in the bibliography. Many of the important documents, such as the cryptographic extensions and the key distribution protocols are not available.

11.2.2 IDIP, CISL and CIDF

The intent for use of CISL is in conjunction with IDIP and the CIDF mechanism described in Section 5. CISL appears to be a bit cumbersome to use, and is not as portable as the IDMEF XML format.

CIDF is an effort to develop protocols and application programming interfaces so that IDS research projects can share information and resources to enable sharing of IDS components. In my opinion for a global, distributed and autonomous network DDoS network it is not sufficient to provide all of the capability required.

IDIP itself provides an easy to understand protocol for exchanging data and initiating a response. And, for global notification of attacks. This feature is important, but as the protocol is not fully available, it limits the ability to use this in a real-world application.

11.3 FUTURE WORK RECOMMENDED

11.3.1 Correlation Engine

Regardless of the decision to use IDIP, IDMEF, CISL or a combination thereof, the most important next step to this work is to develop a correlation engine for disseminating and understanding the data that becomes available. There are many ways to approach this, for example using Artificial Intelligence learning techniques. Another way might be to employ the use of a Java rules based system, called JESS. JESS is a rule engine built in Java. There are many other types of rule engines available as well, and one could be developed specifically for this application domain.

Currently, the A2D2V2 IDIP implementation does not include a correlation engine. This makes the response mechanism coarse and broad, and it really only adopts the same response as built in to the A2D2 Flood preprocessor rate limiting mechanism. However, with the addition of IDIP Trace capability, and the inclusion of a comprehensive correlation engine, the IDIP Trace data could be utilized much more effectively.

Some emerging research has begun on the correlation of attack alert data. Dan Gorton[GO03] talks in this doctoral thesis of the need to correlate data from multiple IDS systems within a network. This idea could easily be extended to include network IDS systems that are not contained within the same LAN. Correlation is one of the key outcomes expected with the CISL language along with the IDXP protocol specification.

Another area that would benefit from correlation is the situation described in section 10.1.6. The ability to gather data autonomously from both the IDS and dynamic tracing mechanism and then correlate the data would help to better determine legitimate traffic and reduce the need for specific rules that may allow attack traffic to penetrate.

11.3.2 IDIP enhancements

With the current status of the IDIP effort and standardization, the continued use of IDIP for an application like this is doubtful. However, if IDIP is chosen as the continuing technology and protocol, an effort must be made for collaboration with Telcordia, Inc. and McAfee Software, Inc. The push must be made to make IDIP protocols open, standard and available.

The pieces that are currently unavailable are the cryptographic extensions and the IDIP key distribution protocol. These two are essential to making any implementation of IDIP robust.

For this project, there were several areas of the IDIP protocol that were not developed. These should continue to be developed.

IDIP Message Layer enhancements:

- Full implementation of the IDIP message sequence numbering and timing checks
- Support for proxy Discovery Coordinator
- Retransmission time-out support
- Full versioning support
- Message checksumming

- Multicast Message support

IDIP Application Layer Enhancements:

- More Application Message ClassID support
- Response Extension support
- Discovery Coordinator communication support
- Full Edge Boundary Controller implementation

Along with this, standards body work must be done to publish and ratify the IDIP protocol definitions. Without this, the likelihood of adoption of this protocol is small.

11.3.3 Redundant/Cooperative Discovery Coordinators

Currently the implementation of IDIP in this project has a single point of failure, namely the IDIP Message Layer/Discovery Coordinator node. The CIDF specification does not indicate multiple Discovery Coordinator nodes in the architecture published. However, my opinion is that the ability to failover in the event of a failure is critical to maintaining the autonomy wanted with an IDIP system. If the Discovery Coordinator becomes unresponsive, the current action is that all other local nodes will take their own action to the perceived attack. Without the ability to see the global picture that the Discovery Coordinator has, this renders this system as effective as a non-IDIP enabled system.

Redundant and perhaps cooperative Discovery Controllers would reduce the single point of failure scenario and the potential for cooperation among these Discovery Coordinators could speed up the response effort in the IDIP system. Enabling this feature would require more protocol definitions as to how the need for a failover is detected, perhaps just an extension to the already defined IDIP neighbor notification. The cooperation would require the introduction of a database of some sort that was synchronous with regard to updates. There are provisions in the IDIP Application protocol for DC to DC communication. This could be extended to include a cooperative communication protocol. As it stands today, this protocol only allows for sending and acknowledging messages.

11.3.4 Incorporate OpenSLP

Service Location Protocol (SLP) is an IETF standards track protocol that provides a framework to allow networking applications to discover the existence, location, and configuration of networked services in enterprise networks. Traditionally, in order to locate services on the network, users of network applications have been required to supply the host name or network address of the machine that provides a desired service. Ensuring that users and applications are supplied with the correct information has, in many cases, become an administrative nightmare. [OpenSLP]

Protocols that support service location are often taken for granted, mostly because they are already included (without fanfare) in many network operating systems. For example, without Microsoft's SMB service location facilities, "Network Neighborhood" could not discover services available for use on the network and Novell NetWare would be unable to locate NDS trees. Nevertheless, an IETF service location protocol was not standardized

until the advent of SLP. Because it is not tied to a proprietary technology, SLP provides a service location solution that could become extremely important (especially on UNIX) platforms.

Like all Internet Engineering Task Force (IETF) standards, Service Location Protocol (SLP) is described in great detail by documents called Request For Comments (RFC).

For these, the reader is referred the following RFCs:

RFC 2608 - Service Location Protocol, Version 2

RFC 2609 - Service Templates and Service Schemes

RFC 2610 - DHCP Options for Service

Location Protocol

RFC 2614 - An API for Service Location Protocol

SLP can eliminate the need for users to know the names of network hosts. With SLP, the user only needs to know the description of the service he is interested in. Based on this description, SLP is then able to return the URL of the desired service.

In many cases, SLP can eliminate the need for software applications to prompt users for host names, or to read host names from configuration files.

SLP is a unicast and a multicast protocol. This means that the messages can be sent to one agent at a time (unicast) or to all agents (that are listening) at the same time (multi-cast). A multicast is not a broadcast. In theory, broadcast messages are "heard" by *every* node on the network. Multicast differs from broadcast because multicast messages are only "heard" by the nodes on the network that have "joined the multicast group".

One of the most important parts of the SLP specification is the standard Application Programmers Interface (API). The SLP API is an interface that allows programmers to use SLP in their applications to locate services. Without the API, SLP would be little more than a specification. With the API, developers can easily add SLP based features to their programs.

11.3.4.1 A more dynamic global response using OpenSLP

At the heart of this suggestion is the combination of IDIP and OpenSLP in to future project work. The IDIP Discovery Coordinator will assume responsibility for the global response to the network attack. The Discovery Coordinator will make use of its ability to communicate with non-IDIP nodes. It will also be modified to support SLP. In this way, the Discovery Coordinator will be able to dynamically discovery any available proxy servers that have registered their services with SLP. The services provided by the proxy servers would be things like IDIP message forwarding services or correlation services. Upon detection of these servers, the Discovery Coordinator may issue a directive to the clients or client-DNS servers to redirect traffic through the newly discovered proxy servers.

This idea is based on research done by Dr. C. Edward Chow, University of Colorado, Colorado Springs title the Secure Collective Internet Defense Network[C03].

11.3.5 IDMEF, IDXP, CISL and IDIP

As noted in Section 11.1, there are a few new technologies and protocols that have been full developed and published since the start of this project work. In particular, the IDMEF and IDXP protocols have been published and approved by the IETF body.

Had these alternative choices been available at the start of this project, or even during its completion prior to the point of no return, the likelihood of the adoption of these in conjunction with, or in lieu of IDIP would have been fairly great. During the research for these technologies it has become fairly evident that in particular, IDMEF and IDXP offer a more stable, approved standardized and ubiquitous way to model and exchange data between disparate IDS's and even between non-local network entities.

As a result of this work, a suggestion is made to more fully explore the use of IDMEF and IDXP in the following ways:

1. Add the XML data model support to the existing IDIP protocol. Currently, IDIP uses CISL, but XML is a more standard way to model and format data. A comparison of these two should be made.
2. Explore the use of IDXP as opposed to IDIP. Since IDXP is fully available and is standardized the potential for adoption of this is much higher than the current state of IDIP.
3. Explore extensions to IDXP that would enable a more global, full response to a perceived attack. One thing that IDIP does well is the ability to request and track multiple host information and responses during an attack. This ability allows a system to see the more global view of the attack and in theory should enable a better, more accurate response.

11.3.6 CIDF work

The intent of A2D2V2 was to show that a more global response mechanism could be deployed from a DDoS system like A2D2. However, this work is only the beginning of the cooperative defense against DDoS attacks.

CIDF defines an architecture for cooperative intrusion detection. CIDF relies on IDIP as the basis for the cooperation. Depending on the future direction of IDIP the actual use of the CIDF defined architecture may not be appropriate. However, future work more generally on the cooperative capability of a DDoS attack identification and response mechanism is important. The ability to stop DDoS attacks more globally would be a huge win.

11.3.7 Performance Enhancements to Firewall Code

It was noted many times during test runs that the performance of the dynamic tracing, and subsequent notification to the upstream routers was poor depending on attack load. There were many occasions of what appeared to be hung firewalls and slow moving messages. This contributed to the slow recovery of both clients during attacks. There are several areas for potential performance work to the IDIP firewall code:

1) Dynamic Tracing Enhancements:

- The dynamic tracing is very basic and no performance enhancements were done. One possibility for improvement in this area is to modify the code to give the dynamic tracing higher priority and higher bandwidth than other activity on the firewall/router.
- From the response data in section 9 it is clear that the routers inability to apply attack rules to itself while running the dynamic tracing results in a longer sustained performance degradation during attack. One area for performance improvement would be to modify this to run in the background while allowing the router to proceed forward with its own response to the attack by applying the attack rules simultaneously.
- Another possibility is provide multiple threads for tracing, each monitoring a separate interface so as to not stall response until all interfaces have been polled.

IDIP Messaging enhancements:

- The general IDIP messaging mechanisms is also slow during attacks and could be improved as well. Multiple threads of message queues would enable a much faster send and response mechanisms than A2D2V2 provides.

11.3.8 IDIP Tracing and Real-Time Locating of other IDIP networks

As it is today with IDIP Trace requests an audit log is expected to be kept and when an audit request is received, the IDIP node is to do an audit of the specific traffic from the indicated source IP address. This data is then sent back to the IDIP Discovery Coordinator for further correlation and action updates. This scheme relies heavily on the ability of the Discovery Coordinator to gather the incoming data, correlate it and determine the best action to take all in near real-time. As noted in Section 11.3.1, a sophisticated correlation and response engine is clearly an area of need for a full, distributed IDIP architecture to work.

Beyond the correlation of data and response to the data, the ability to accurately identify the true source of a packet or set of packets is critical. To notify another IDIP Boundary Controller with a suggested response for their network based on inaccurate data is potentially a road block for real IDIP adoption.

IP Spoofing is common in DDoS type attacks. The lack of security features in TCP/IP specifications facilitates IP Spoofing. In IP Spoofing the source address of an incoming packet has been updated with an address that is not the true source of the packet. Since the Internet's routing mechanism is stateless and mostly based on destination addressing no entity is responsible for ensuring that source addresses are correct.[AL03].

There are several new research avenues being pursued to eliminate or at least minimize the ability of attackers to add bogus source address data to an IP packet.

11.3.8.1 IP Traceback

One of the emerging research areas IP Traceback technology. A2D2V2 utilizes a static router configuration file to determine the route a specific packet has taken as identified from the source IP field in the packet. This works well in a well understood test environment but provides no dynamic capability to determine the route for any packet on the network.

There are several areas of research in to how to best achieve accurate IP traceback data. Current approaches today include link testing which consists of testing network links between routers to determine the origin of an attackers traffic. Most techniques start from the router closest to the victim and interactively test its incoming links to determine which one carries the attack traffic. This continues on recursively until reaching the traffic's source. This is a reactive method and requires the attack to remain active during the testing. [AL03].

Another approach is to log the packets at key routers throughout the network and then use a data mining technique to extract information about the attack traffic's source. This solution can work, but the overhead for storage and processing time is generally thought too large to make it a viable solution.

ICMP traceback which uses ICMP traceback router generated messages contain path information that indicates where the packet came from, when it was sent and its authentication. However, on major drawback to this solution is the fact that in a DDoS attack when a particular zombie is responsible for only a small amount of the total traffic, the sampling rate used in this scheme introduces the likelihood that getting an attack packet has a much lower probability.

Packet marking is one of the newest methods proposed for IP traceback. Traceback data is inserted in to the IP packet to be traced, thus marking the packet along the way through the various routers on the network to the destination host. This allows the destination host machine to use the markings in a particular packet to deduce the path a packet has taken.

Currently, there are no commercial off-the-shelf products that can perform effective traceback in real time or across multiple hops. To do this it would mean changes to the existing routing protocols. This would require new hardware.

From the literature read, it would seem that the most promising solution for IP traceback is the packet marking along with a modified logging scheme. The logging could be achieved with hash buckets of data on the routers, in which the routers log only partial packet information and only a sampling of packets would get logged.

For IDIP this could be used and incorporated on to the IDIP Boundary Controllers so that when an IDIP Trace or Request message is received the Boundary Controller could then make more informed decisions based on the data gathered.

This is clearly an area that requires future work.

12. FINAL CONCLUSIONS

As part of this project, the analysis of the data collected has been given in Section 10.3 This section will expand on this analysis and give some conclusions developed as a result of this analysis for the reader to review.

12.1 WAS A2D2V2 A SUCCESS?

The ability to notify and request response from an outside network is a success for A2D2V2. The difficulty lies in that the IDIP technology adds a bit more overhead in general. Along with this, the incomplete documentation available for IDIP makes it difficult to fully implement and to produce a full, cooperative response system.

However, as the numbers show in Section 9 A2D2V2 has the ability to mitigate traffic from attack sources and can do so in much less time than it would take a system administrator. Along with the pushback to the source of the attack as shown, the ideas incorporated in A2D2V2 prove to be successful.

In the end, the use of IDIP may be proven to be the wrong choice. At the time of the start of this project it was the only available protocol for cooperative intrusion detection and response. But, the emergence of the other technologies discussed in Section 11 make the continued use of IDIP doubtful. A more interoperable choice may exist in the newer protocols and should be examined in further work.

12.2 IDIP AS A FUTURE TECHNOLOGY

The theory behind the development of IDIP was valid, that is to develop an open, standard and published set of protocols for use in intrusion detection and response. This is why IDIP was chosen for use in this project.

Since IDIP has been privatized to some extent, it is my belief that the further use of IDIP for research in DDoS attack response and detection should be evaluated carefully. With the emergence of IDMEF and IDXP, IDIP, unless it becomes available generally very soon, will most likely be adopted as a proprietary solution.

Clearly a standard, published protocol for modeling attack data and exchanging data between disparate systems is critical. That was the intent of IDIP. Work in this area must continue.

12.3 WHERE THE REAL WORK LIES

It is obvious from the development and deployment of this project that the real work lies in the ability to quickly gather, correlate and form a response to a perceived DDoS attack. This is clearly an areas of deficiency in general for DDoS attacks.

All of the existing protocols do make mention of a centralized coordinator, where all data will be archived and where all responses to a perceived attack will begin. However, none of them go in to detail about the functionality for this centralized coordinator. One

area of potential revenue is in the development of a sophisticated correlation and response engine for cooperative DDoS response. It would be important to make this work deployable in multiple types of environments, and to ensure that it follows the standard protocols adopted by the industry in DDoS defense.

BIBLIOGRAPHY

[AL03] Aljifri, Hassan. 2003. IP Traceback: A new Denial-of-Service Deterrent?
http://cs.uccs.edu/~chow/pub/master/sjelinek/doc/iptrace.pdf

[BAL-etal98] Balasubramaniyan, J. et al. 1998. An Architecture for Intrusion Detection using Autonomous Agents.
https://www.cerias.purdue.edu/tools_and_resources/bibtex_archive/archive/98-05.pdf

[BEAN03] Belenky, Andrey & Ansari, Nirwan. 2003. On IP Traceback.
http://web.njit.edu/~ansari/papers/03COMM_belenky.pdf

[B02] Brindley, Adrian. 2002. Denial of Service Attacks and the Emergence of Intrusion Prevention Systems.
http://www.sans.org/rr/firewall/prevention.php

[BU00] Buchholz, Florian et al. 2000. CERIAS Technical Report. Purdue University.
http://cs.uccs.edu/~chow/pub/master/sjelinek/doc/cerias.ps

[C02] Cearns, Angela. 2002. Autonomous Anti-DDoS Network.
http://cs.uccs.edu/~chow/pub/master/acearns/doc/angThesis-final.doc

[CHA-etal00] Chang, H. et.all 2000. Towards Hidden Attackers on Untrusted IP Networks. NCSU, June 2000.

[CHI-etal04] Chien-Lung, W. et al. 2004 On network-layer packet traceback: tracing denial-of-service (dos) and distributed denial-of-service (ddos) attacks.
http://cs.uccs.edu/~chow/pub/master/sjelinek/doc/research/3112814.pdf

[CHO-etal99] Chong, R. et al. 1999. Deciduous, A Decentralized Source Identification for Network- Based Intrusions. In 6th IEEE/IFIP International Symposium on Integrated Network Management (IM '99), M. Sloman, S. Mazumdar, and E. Lupu, eds., Boston, MA, May 1999, pp. 701-714

[C03] Chow, Edward C. 2003. SCOLD.
http://cs.uccs.edu/~chow/talk/ChowTPACForum.ppt

[CIDF] Common Intrusion Detection Framework and Specification Language. 2002.
http://www.ietf.org/html.charters/idwg-charter.html

[DAN00] Daniels, Thomas E., Spafford, Euguene H. Subliminal Traceroute in TCP/IP. 2000.
http://httpL//cs.uccs.edu/~chow/pub/master/sjelinek/602.pdf

[DAR02] DARPA. 2002. Common Intrusion Detection Framework.
http://www.isi.edu/gost/cidf/

[CRO95] Crosbie, G. et al. 1995. Defending a Computer System using Autonomous Agents.
http://ftp.cerias.purdue.edu/pub/papers/mark-crosbie/mcrosbie-spaf-NISC.pdf

[DW03] Wilkinson, David. 2003. Enhanced DNS, Masters Thesis.
http://cs.uccs.edu/~chow/pub/dbwilkinson/doc/dwilkinson_thesis.doc

[FE-etal03] Feinstein, Laura et al. 2003. Statistical Approaches to DDoS Attack Detection and Response. DARPA Information Survivability Conference and Exposition, 2003.
http://ieeexplore.ieee.org/xpl/RecentCon.jsp?punumber=8503

[GH-etal99] Ghosh, A. et al. 1999. Learning Program Behavior Profiles for Instrusion Detection. Proceedings of the Workshop on Intrusion Detection and Network Monitoring. USENIX, April, 1999.
http://www.google.com/url?sa=U&start=9&q=http://www.cigital.com/papers/download/usenix_id99.ps&e=15206

[GO-etal01] Goldman, Robert et al. 2001. Information Modeling for Intrusion Report Aggregation. Honeywell Labs.
http://citeseer.ist.psu.edu/cache/papers/cs/24225/http:zSzzSzwww.geocities.comzSzrpgoldmanzSzpaperszSzdiscex01irm.pdf/information-modeling-for-intrusion.pdf

[GO03] Gorton, Dan. 2003. Extending Intrusion Detection with Alert Correlation and Intrusion Tolerance. Chalmers University of Technology. Doctoral Thesis.
http://unbolted.llarian.net/ids-docs/extending-ids-with-alert-correlation-and-intrusion-tolerance.pdf

[IDMEF] Intrusion Detection Message Exchange Format.
http://www3.ietf.org/proceedings/05nov/IDs/draft-ietf-idwg-idmef-xml-14.txt

[**IDXP] Intrusion Detection Exchange Protocol**
http://www.ietf.org/internet-drafts/draft-ietf-idwg-beep-idxp-07.txt

[KO02] Kothari, Pravin. 2002. Intrusion Detection Interoperability and Standardization. 2002.
http://www.sans.org/reading_room/whitepapers/detection/356.php

[LY04] Lydon, Andrew. 2004. COMPILATION FOR INTRUSION DETECTION SYSTEMS. Masters Thesis.
http://www.ohiolink.edu/etd/send-pdf.cgi?ohiou1088179093

[NB02] Network Associates Labs. **2002.** Boeing Phantom Works.
http://zen.ece.ohiou.edu/~inbounds/DOCS/reldocs/IDIP_Architecture.doc

[NB02-1] Network Associates Labs. 2002. Boeing Phantom Works.
http://zen.ece.ohiou.edu/~inbounds/DOCS/reldocs/IDIP_Application_Layer.doc

[NB02-2] Network Associates Labs. 2002. Boeing Phantom Works.
http://zen.ece.ohiou.edu/~inbounds/DOCS/reldocs/IDIP_Message_Layer.doc

[OpenSLP] Open SLP Project.
http://www.openslp.org/

[PA-etal03] Papadopoulos, Christos et al . 2003. COSSACK: Coordinated Suppression of Simultaneous Attacks.
http://www.isi.edu/~hussain/pubs/Papadopoulos03a.pdf

[PE00] Petkac, M. and Badger, L. Security Agility in Response to Intrusion Detection
http://cs.uccs.edu/~chow/pub/master/sjelinek/research/43.pdf

[QI-VA-SU01] Qui, L., Varghese, G. and Suri, S. 2001. Fast Firewall Implementations for Software and Hardware-based Routers.
http://www.cs.ucsd.edu/~varghese/PAPERS/ICNP2001.pdf

[SC-DJ03] Schnackenberg, Dan and Djahandari, Kelly , Network Associates Labs. 2003. Infrastructure for Intrusion Detection and Response.
http://www.isso.sparta.com/research/documents/iidr_abstract.pdf

[SC-etal01]Schnackenberg, Dan et al. 2001. Cooperative Intrusion Traceback and Response Architecture (CITRA). Phantom Works, Boeing, Co.
http://cs.uccs.edu/chow/pub/master/sjelinek/doc/research/12120056.pdf

[SPA-ZA00] Spafford, E. and Zamboni, D. 2000. Data Collection Mechanisms for Intrusion Detection Systems.
http://homes.cerias.purdue.edu/~zamboni/pubs/2000-08.pdf

[ST-etal01] Stern, Daniel et al. 2001. Autonomic Response to Distributed Denial of Service Attacks. Recent Advances in Intrusion Detection. 4th International Symposium, RAID 2001 Davis, CA, USA, October 10-12, 2001 Proceedings.
http://cs.uccs.edu/~chow/pub/master/sjelinek/research/22120134.pdf

[TA02] Tanase, Matt. 2002. Barbarians at the Gate: An Introduction to Distributed Denial of Service Attacks. 2002.
http://www.securityfocus.com/infocus/1647

[TO02] Toplayer.com. 2002. Intrusion Protection Systems.
http://www.toplayer.com/bitpipe/IPS_Whitepaper_112602.pdf

[WO-DU99] Wood, Bradley J. and Duggan, Ruth A., Sandia National Labs. 1999. Red Teaming of Advanced Information Assurance Concepts
http://stinet.dtic.mil/cgi-bin/GetTRDoc?AD=ADA407925&Location=U2&doc=GetTRDoc.pdf

[XML] Extensible Markup Language. W3C.
http://www.w3.org/XML/

APPENDIX A

A.1 SETUP OF A2D2V2 TEST BED CONFIGURATION

STEP 1 – INITIAL TEST BED SETUP

All hosts have been installed with the Fedora Core 5 release of Linux. The hosts are given private IP addresses, with the exception of the routers, to allow access remotely to the A2D2V2 test bed for testing and debugging.

STEP 2-ROUTING TABLE SETUP

Routing tables must be setup manually,so that all hosts can talk with each other. This can achieved by developing and running a set of shell scripts to initialize each of the hosts routing table or by manually updating the routing table information for each router in the system. The routing table for R97 was setup as:

Kernel IP routing table

Destination	Gateway	Genmask	Flags	MSS Window	irtt Iface
128.198.61.0	0.0.0.0	255.255.255.128	U	0 0	0 eth1
192.168.16.0	192.168.12.99	255.255.255.0	UG	0 0	0 eth2
192.168.15.0	192.168.12.99	255.255.255.0	UG	0 0	0 eth2
192.168.14.0	192.168.12.99	255.255.255.0	UG	0 0	0 eth2
192.168.13.0	192.168.12.99	255.255.255.0	UG	0 0	0 eth2
192.168.12.0	0.0.0.0	255.255.255.0	U	0 0	0 eth2
192.168.11.0	0.0.0.0	255.255.255.0	U	0 0	0 eth0
169.254.0.0	0.0.0.0	255.255.0.0	U	0 0	0 eth2
0.0.0.0	128.198.61.1	0.0.0.0	UG	0 0	0 eth1

The setup must include routes to all subnets that need to be accessible by the specified router. A script can be used to set this up but utilizing this method will not make the routes permanent so upon reboot all of this data will be lost. To make this data permanent in the system configuration files you must use the system network modification utilities and choose to save the configuration.

STEP 3-FIREWALL RULES SETUP

Firewall security is on by default for all the hosts. Clients 1, 2 the Attack agents and the Server must have firewall security disabled. This is achieved by running the *redhat-config-securitylevel, or system-config-securitylevel* tool and selecting the *Disable* feature. The name of the tool is dependent on the version of RedHat Linux that is running.

Routers R97, R99 and R102 each have different firewall rules. The appropriate modifications to the CBQ.sh scripts must be made for the Iptables FORWARD chain on each router. See Appendix B for this script for the R99 router.

Step 4-Setup For Routers

Each router must have the following installed on the system:

- idip_firewall_receiver
- cbq.sh
- rateif.pl, rateif.conf
- tcpdump.sh
- dumper.sh

To start firewall software it must be done in the following order:

run *sh cbq.sh stop* – Reset all the existing firewall rules

run *perl rateif.pl* – Setup the rate limiter and iptables chain rules

run *sh cbq.sh start* – Start the firewall and CBQ rules

run *./ idip_firewall_receiver* – to start IDIP messaging and DC

The system will be in a state ready to accept and process IDIP messages.

Step 5-Setting up Client tests and traffic monitoring

To run the testing simple tcp receive program was developed and deployed on C1 and C2. The source for this is found as noted in Appendix B. Install the *tcp_rcv* module on each client.

To enable this:

run *./tcp_rcv*

This must be started before the server program, *tcp_snd*, which is detailed in Step 6.

On both C1 and C2 a traffic monitoring program was deployed to gather the throughput data. The Perl program used was the same one as provided in A2D2, *plot.pl*. This program outputs packets received and sent to standard out, as well as sending this to a file named data.txt. This data.txt file is then used to produce the plots shown in Section 9. A script is also provided that runs the plot.pl program, *runplot.sh.*

To invoke:

run *sh runplot.sh*

This script sets the time to record at 1 second. You can modify this by changing the option value associated with the -t option in this script. You can also run *plot.pl* without the script.

Step 6-Setting Up the Servers

Installation of the Snort IDS was done on Server S1. The HOME_NET value was changed in the snort.conf file to reflect the 192.168.13.1 network. This is the network that Snort is to monitor. The rules for the flood-ignorehosts has been changed as well. This rule indicates which hosts should be ignored in terms of flood traffic. In this scenario, there are no hosts coming in to this network that should be ignored. In general, you would want this to be set to the network addresses of legitimate incoming clients.

To start Snort:

run *./snort -A UNSOCK*.

This enables alerts to be sent to the UNIX socket on this host.

An additional binary is run on Server S1. As noted in A2D2 the alert binary[C02] was run to intercept the UNIX socket alerts and to enable the rate limiting feature. A similar scheme was deployed for A2D2V2, only the new binary is called *report_idip*, and it also intercepts the alert messages on the UNIX socket, creates the appropriate IDIP messages and forwards these on to the firewall/router for processing.

To enable the IDIP feature:

run *./report_IDIP -d -h <firewall host ip> -l <logfile name>*

Where -d enables debugging so it is not required. -l enables logging which is also not required.

Step 7-Setting up Server 1 test and traffic monitoring

Similar to Step 5, a simple tcp send program was developed and deployed on S1 and S2. This program sends a continuous stream of tcp packets to C1 and C2. To enable this you type *./tcp_snd <client ip address>*. For multiple clients this must be started individually.

For traffic throughput monitoring the *iptraf* tool was used on Server 1. This was enabled by typing *iptraf -d eth0 -t 10 -B*.

Step 8 -Setting up the Attackers

As noted in Section 10 the Stacheldraht version 4.0 tool was used to deploy the TCP SYN DDoS attack. This tool was installed on both attackers. Attacker A1 was the master node, and an agent as well. A2 and A3 were agents in this process.

1) On attacker A1, start the handler process. This is achieved by typing *./mserv*.

2) On attackers A1, A2 and A3, start an agent. This is achieved by invoking the *td* program contained in the *leaf* subdirectory of the tool. One small change must be made in the *td.c* file to enable this agent for A2D2V2. This file must be rebuilt when after this change is made. The default master server values must be changed to reflect the current environment. For A2D2V2 this was modified as follows:

```
/* default masterservers */

#define MSERVER1 "192.168.11.2"
#define MSERVER2 "192.168.11.2"
```

The attack agents must know what the address is of the master handler to be able to connect and send broadcast messages. The invocation of *./td* on both attackers starts the agents.

Step 9 – Starting the attack

Since attacker A3 is the master handler, this host is used to initiate the TCP SYN attack. To do this you must use the client program provided in the Stacheldraht version 4.0 tool. This is achieved by invoking the client program from the top level tool directory.

run *telnetc/client <master handler ip address>*.

Once initiated, you should see the following:

telnetc/client 192.168.11.2
[*] stacheldraht [*]
(c) in 1999 by randomizer

trying to connect...
connection established.
----
enter the passphrase :
----
entering interactive session.
****
welcome to stacheldraht
****
type .help if you are lame

stacheldraht(status: a!3 d!0)>

The passphrase is *'manager'*.

The value noted for *'a'* indicates the number of alive attack agents. The value for *'d'* indicates those that are no longer responding. To initiate an TCP SYN attack:

.msyn <host ip address to attack:host ip address to attack:...>

To stop the attack:

.mstop all.

To limit the attack duration:

.mtimer <time to limit>.

There are many more available types of attacks. To view the list type *.help*. To exit the tool type *.quit*

A.1.1 The A2D2V2 Attack Setup and run Recipe:

```
1. On R97 run the following in this order:
      sh cbqr97.sh stop
      perl rateifr97.pl start
      sh cbqr97.sh start
      ./idip_firewall_receiver_97

2. On R102 run the following in this order:
      sh cbqr102.sh stop
      perl rateifr102.pl start
      sh cbqr102.sh start
      ./idip_firewall_receiver_102

3. On R99 run the following in this order:
      sh cbqr99.sh stop
      perl rateifr99.pl start
```

```
      sh cbqr99.sh start
      ./idip_firewall_receiver_99

3. On each Client C1 and C2 run in this order:
      ./tcp_rcv
      sh runplot.sh – if you want to monitor packet rates

4. On Server S1 in separate xterms run in this order:
      ./report_idip -d -l <logname> -h <source ip address>
      ./snort -A UNSOCK
      ./tcp_snd 192.168.11.1

5. On Server S2 in separate xterms run in this order:
      ./tcp_snd 192.168.16.1

6. On A1 run:
      ./mserv 192.168.11.2
      ./leaf/td

7. On A2 and A3 run:
      ./td

8. On A1 run:
      ./telnetc/client 192.168.11.2
      Within the attack tool run:
      .showalive – to ensure all 3 attackers are alive
      .mtimer 200 – to set attack duration
      .msyn 192.168.13.1:192.168.15.1

Attack will be started and data will be gathered on client side
via the plotting program, plot.pl.
```

A.1.2 What to Look for To Verify Cooperative IDIP Defense

Router output:

The major observable point for verifying cooperative defense in action in the A2D2V2 system is at the router. The three IDIP enabled routers will output a lot of messages during an attack when attack detection and mitigation has started. You will also be able to observe some data from the Snort IDS IDIP message reporting mechanism.

R99 should output messages first as it is the first in line for notification from the Snort IDS. An example of its output is shown:

idip_firewall_receiver.c: Waiting for incoming idip messages on firewall
idip_message_receiver: sizeof idip_message_t 2232
idip_message_receiver: n bytes received :2232
idip_message_receiver: message received: spp_flood 101 :rate FLOOD DETECTED from 192.168.11.190/24 (THRESHOLD 100 connections exceeded in 0 seconds)
idip_message_receiver: received idip_message packet
idip_firewall_receiver.c do_trace_request: thishost is r991 received message spp_flood 101 :rate FLOOD DETECTED from 192.168.11.190/24 (THRESHOLD 100 connections exceeded in 0 seconds)
threshold packets 100
cmd = sh tcpdump.sh 100 3
idip_firewall_receiver.c do_trace_request: UNDER ATTACK:
idip_firewall_receiver.c do_trace_request: from source 192.168.11.187
idip_firewall_receiver.c do_trace_request: on interface eth0
idip_firewall_receiver.c do_trace_request: number of packets 806
Sent msg FLOOD DETECTED on r991 from 192.168.11.187 to rate limiter
idip_firewall_receiver.c do_trace_request: alertmsg sent to 192.168.12.97:

What this output shows is the initial 'Waiting on incoming...' message and the subsequent receiving of an alert from the Snort IDS. This triggers the tracing to being as seen by the 'do_trace_request' output. You then see the subsequent sending of the new flood detected message to the upstream router.

R97 will have received this message and will output something like:

idip_message_receiver: message received: FLOOD DETECTED on r991 from 192.168.11.187
idip_message_receiver: received idip_message packet
idip_firewall_receiver.c process_idip_message: Received DCDO request

This shows that R97 received the DCDO message from R99 and will be processing it. R102 will show similar output when it receives a message.

Snort IDS Reporting mechanisms output:

[Tue Jul 11 21:47:51 2006

```
] spp_floodindi: SubNet FLOOD DETECTED from 192.168.11.27/24 to
192.168.11.27 (THRESHOLD 101 connections exceeded in 0 seconds)
[1]
[Tue Jul 11 21:47:51 2006
] spp_floodindi: SubNet FLOOD DETECTED from 192.168.16.72/24 to
192.168.16.72 (THRESHOLD 104 connections exceeded in 0 seconds)
[2]
[Tue Jul 11 21:47:51 2006
] spp_flood 101 :rate FLOOD DETECTED from 192.168.11.27/24
(THRESHOLD 100 connections exceeded in 0 seconds) [3]
[Tue Jul 11 21:47:51 2006
] spp_flood 101 :rate FLOOD DETECTED from 192.168.16.72/24
(THRESHOLD 100 connections exceeded in 0 seconds) [4]
[Tue Jul 11 21:47:51 2006
] spp_flood 51 :rate FLOOD DETECTED from 192.168.11.27/24
(THRESHOLD 50 connections exceeded in 0 seconds) [5]
[Tue Jul 11 21:47:52 2006
] spp_flood 51 :rate FLOOD DETECTED from 192.168.16.72/24
(THRESHOLD 50 connections exceeded in 1 seconds) [6]
[Tue Jul 11 21:48:13 2006
] spp_flood: End of Flood from 192.168.16.72: TOTAL time(1s)
hosts(1) TCP(0) UDP(0) ICMP(0) [7]
```

This will just show what the Snort IDS and what the IDIP message reporting mechanism has sent to the upstream IDIP enabled routers.

APPENDIX B

B.1 A2D2V2 SOURCE AND BUILD RULES

A2D2V2 Source layout and Build Rules

The source for this project is located at:

a2d2v2.csnet.uccs.edu/home/sjelinek/src.tar.gz

Once unzipped and untarred the layout of the source is as follows:

/a2d2-2firewall – all the firewall source files.

idip_firewall_receiver.c, cbq.sh, dumper.sh, rateif/,tcpdump.sh and topo.txt* must modified as necessary for the appropriate firewall they are running on.

Makefile and build.sh will build this executable.

/idip_common

/idip_message

/idip_include

IDIP source files. Makefiles will build appropriate source

/snort

snort is built as per build rules in INSTALL file

report_idip is built using the *build.sh* script

/clientserv

Client/server source files and scripts. Makefile builds *tcp_srv, tcp_snd* targets.

/stach

stacheldrahtv4 attack tool source. Makefiles will build targets.

APPENDIX C

C.1 CLASS BASED QUEUING SCRIPT FOR A2D2V2 TEST BED

This is the CBQ and Firewall setup script for the R99 firewall. The other firewall scripts are similar except that R99 must be setup for the FORWARD chain rules for two output interfaces. This is due to the fact that R99 has two possible output interfaces, eth1 and eth3 to serve both S1 and S2.

```
# This is the location of the iptables command
IPTABLES="/sbin/iptables"
TC="/sbin/tc"


OUT1="eth1"
OUT1_IP=`/sbin/ifconfig eth1 |grep inet.addr | sed "s/:/ /g" | awk
'{print $3}'`OUT1_BCAST=`/sbin/ifconfig eth1|grep inet.addr | sed
"s/:/ /g" | awk '{print $5}'`
OUT1_MASK=`/sbin/ifconfig eth1 |grep inet.addr |sed "s/:/ /g" |awk
'{print $7}'`echo "INTERFACE: $OUT1 IP: $OUT1_IP BCAST:
$OUT1_BCAST MASK: $OUT1_MASK"

OUT3="eth3"
OUT3_IP=`/sbin/ifconfig eth3 |grep inet.addr | sed "s/:/ /g" | awk
'{print $3}'`OUT3_BCAST=`/sbin/ifconfig eth3|grep inet.addr | sed
"s/:/ /g" | awk '{print $5}'`
OUT3_MASK=`/sbin/ifconfig eth3 |grep inet.addr |sed "s/:/ /g" |awk
'{print $7}'`
case "$1" in
   stop)
 echo "Shutting down firewall..."
      $IPTABLES -F
      $IPTABLES -F -t mangle
      $IPTABLES -F -t nat
      $IPTABLES -X
      $IPTABLES -X -t mangle
      $IPTABLES -X -t nat

      echo "Setting default policy to ACCEPT"

      $IPTABLES -P INPUT ACCEPT
      $IPTABLES -P OUTPUT ACCEPT
      $IPTABLES -P FORWARD ACCEPT

        # Turn off cbq for all interfaces.
```

```
      echo "Turning off cbq for all interfaces"

$TC qdisc del dev $OUT1 root handle 10:0 cbq bandwidth 10Mbit
avpkt 1000
$TC qdisc del dev $OUT3 root handle 10:0 cbq bandwidth 10Mbit
avpkt 1000
      echo "...done"
      ;;
status)
      echo $"Table: filter"
      iptables --list
      echo $"Table: nat"
      iptables -t nat --list
      echo $"Table: mangle"
      iptables -t mangle --list
      ;;
    restart|reload)
      sh $0 stop
      sh $0 start
      ;;

   start)
    echo "Starting Firewall..."

    echo ""


##---------------Begin Firewall---------------------------------##
# Default policy is to drop packets
$IPTABLES -P INPUT DROP
$IPTABLES -P OUTPUT DROP
$IPTABLES -P FORWARD DROP


# Reduce DoS'ing ability by reducing timeouts
echo 30 > /proc/sys/net/ipv4/tcp_fin_timeout
echo 2400 > /proc/sys/net/ipv4/tcp_keepalive_time
echo 0 > /proc/sys/net/ipv4/tcp_window_scaling
echo 0 > /proc/sys/net/ipv4/tcp_sack

# Allow the kernel to forward packets and prevent ipspoof
echo 1 > /proc/sys/net/ipv4/ip_forward
echo 1 > /proc/sys/net/ipv4/ip_dynaddr
for f in /proc/sys/net/ipv4/conf/*/rp_filter; do echo 1 > $f; done

########### QOS GOES HERE #################
```

```
# The FORWARD chain controls all packets that are not destined for
this
# router. The rules below will mark packets with values that are
# later used in the QoS rules below starting with the $TC calls
below.
# allow icmp & syn traffic mark it with value 1
$IPTABLES -A FORWARD -p icmp -o $OUT1 -t mangle -j MARK --set-mark
1
$IPTABLES -A FORWARD -p tcp --syn -o $OUT1 -t mangle -j MARK --
set-mark 1

#mark incoming mail traffic from smtp and pop3 with mark value 2
$IPTABLES -A FORWARD -p tcp -o $OUT1 -s 0/0 --dport smtp -d 0/0 -t
mangle -j MARK --set-mark 2
$IPTABLES -A FORWARD -p tcp -o $OUT1 -s 0/0 --dport pop3 -d 0/0 -t
mangle -j MARK --set-mark 2

#mark incoming  telnet, ftp and ssh traffic with mark value 3
$IPTABLES -A FORWARD -p tcp -o $OUT1 -s 0/0 --dport 21 -d 0/0 -t
mangle -j MARK
--set-mark 3
$IPTABLES -A FORWARD -p tcp -o $OUT1 -s 0/0 --dport 22 -d 0/0 -t
mangle -j MARK
--set-mark 3
$IPTABLES -A FORWARD -p tcp -o $OUT1 -s 0/0 --dport 23 -d 0/0 -t
mangle -j MARK
--set-mark 3

#mark incoming www and Real Server traffic with mark value 4
$IPTABLES -A FORWARD -p tcp -o $OUT1 -s 0/0 --dport 80 -d 0/0 -t
mangle -j MARK
--set-mark 4
$IPTABLES -A FORWARD -p tcp -o $OUT1 -s 0/0 --dport 443 -d 0/0  -t
mangle -j MARK --set-mark 4
$IPTABLES -A FORWARD -p tcp -o $OUT1 -s 0/0 --dport 7070 -d 0/0 -t
mangle -j MARK --set-mark 4
$IPTABLES -A FORWARD -p tcp -o $OUT1 -s 0/0 --dport 554 -d 0/0  -t
mangle -j MARK --set-mark 4
$IPTABLES -A FORWARD -p tcp -o $OUT1 -s 0/0 --dport 8080 -d 0/0
-t mangle -j MARK --set-mark 4
$IPTABLES -A FORWARD -p tcp -o $OUT1 -s 0/0 --dport 2687 -d 0/0
-t mangle -j MARK --set-mark 4
#For a2d2v2 testing and data gathering, port 7654 is used to end
packets
#from server to client.
```

```
$IPTABLES -A FORWARD -p tcp -o $OUT1 -s 0/0 --dport 7654 -d 0/0
-t mangle -j MARK --set-mark 4
$IPTABLES -A FORWARD -p udp -o $OUT1 -s 0/0 --dport 554 -d 0/0  -t
mangle -j MARK --set-mark 4
$IPTABLES -A FORWARD -p udp -o $OUT1 -s 0/0 --dport 8080 -d 0/0
-t mangle -j MARK --set-mark 4
$IPTABLES -A FORWARD -p udp -o $OUT1 -s 0/0 --dport 2687 -d 0/0
-t mangle -j MARK --set-mark 4
$IPTABLES -A FORWARD -p udp --destination-port 6970:6999 -o $OUT1
-s 0/0 -d 0/0-t mangle -j MARK --set-mark 4
$IPTABLES -A FORWARD -p udp --dport 6970:6999 -o $OUT1 -s 0/0 -d
0/0 -t mangle -j MARK --set-mark 4

# allow icmp & syn traffic mark it with value 1
$IPTABLES -A FORWARD -p icmp -o $OUT3 -t mangle -j MARK --set-mark
1
$IPTABLES -A FORWARD -p tcp --syn -o $OUT3 -t mangle -j MARK --
set-mark 1

#mark incoming mail traffic from smtp and pop3 with mark value 2
$IPTABLES -A FORWARD -p tcp -o $OUT3 -s 0/0 --dport smtp -d 0/0 -t
mangle -j MARK --set-mark 2
$IPTABLES -A FORWARD -p tcp -o $OUT3 -s 0/0 --dport pop3 -d 0/0 -t
mangle -j MARK --set-mark 2

#mark incoming  telnet, ftp and ssh traffic with mark value 3
$IPTABLES -A FORWARD -p tcp -o $OUT3 -s 0/0 --dport 21 -d 0/0 -t
mangle -j MARK
--set-mark 3
$IPTABLES -A FORWARD -p tcp -o $OUT3 -s 0/0 --dport 22 -d 0/0 -t
mangle -j MARK
--set-mark 3
$IPTABLES -A FORWARD -p tcp -o $OUT3 -s 0/0 --dport 23 -d 0/0 -t
mangle -j MARK
--set-mark 3

#mark incoming www and Real Server traffic with mark value 4
$IPTABLES -A FORWARD -p tcp -o $OUT3 -s 0/0 --dport 80 -d 0/0 -t
mangle -j MARK
--set-mark 4
$IPTABLES -A FORWARD -p tcp -o $OUT3 -s 0/0 --dport 443 -d 0/0  -t
mangle -j MARK --set-mark 4
$IPTABLES -A FORWARD -p tcp -o $OUT3 -s 0/0 --dport 7070 -d 0/0 -t
mangle -j MARK --set-mark 4
$IPTABLES -A FORWARD -p tcp -o $OUT3 -s 0/0 --dport 554 -d 0/0  -t
mangle -j MARK --set-mark 4
```

```
$IPTABLES -A FORWARD -p tcp -o $OUT3 -s 0/0 --dport 8080 -d 0/0
-t mangle -j MARK --set-mark 4
$IPTABLES -A FORWARD -p tcp -o $OUT3 -s 0/0 --dport 2687 -d 0/0
-t mangle -j MARK --set-mark 4
#For a2d2v2 testing and data gathering, port 7654 is used to end
packets
#from server to client.
$IPTABLES -A FORWARD -p tcp -o $OUT3 -s 0/0 --dport 7654 -d 0/0
-t mangle -j MARK --set-mark 4
$IPTABLES -A FORWARD -p udp -o $OUT3 -s 0/0 --dport 554 -d 0/0  -t
mangle -j MARK --set-mark 4
$IPTABLES -A FORWARD -p udp -o $OUT3 -s 0/0 --dport 8080 -d 0/0
-t mangle -j MARK --set-mark 4
$IPTABLES -A FORWARD -p udp -o $OUT3 -s 0/0 --dport 2687 -d 0/0
-t mangle -j MARK --set-mark 4
$IPTABLES -A FORWARD -p udp --destination-port 6970:6999 -o $OUT3
-s 0/0 -d 0/0
-t mangle -j MARK --set-mark 4
$IPTABLES -A FORWARD -p udp --dport 6970:6999 -o $OUT3 -s 0/0 -d
0/0 -t mangle -j MARK --set-mark 4

# At the end of each of the chains, accept, if not traffic above.
$IPTABLES -A INPUT -j ACCEPT
$IPTABLES -A  FORWARD -j ACCEPT
$IPTABLES -A OUTPUT -j ACCEPT

#------End Ruleset------#

add_class() {
# $1=parent class $2=classid $3=hiband $4=lowband $5=handle
$6=style
$TC class add dev $OUT1 parent $1 classid $2 cbq bandwidth 10Mbit
rate $3 allot
1514 weight $4 prio 5  maxburst 20 avpkt 1000 $6
$TC qdisc add dev $OUT1 parent $2 sfq quantum 1514b perturb 15
$TC filter add dev $OUT1 protocol ip prio 3 handle $5 fw classid
$2
}

$TC qdisc add dev $OUT1 root handle 10: cbq bandwidth 10Mbit avpkt
1000
$TC class add dev $OUT1 parent 10:0 classid 10:1 cbq bandwidth
10Mbit rate 64kbi
$TC qdisc add dev $OUT3 root handle 10: cbq bandwidth 10Mbit avpkt
1000
$TC class add dev $OUT3 parent 10:0 classid 10:1 cbq bandwidth
```

```
10Mbit rate 64kbit allot 1514 weight 6.4kbit prio 8 maxburst 20
avpkt 1000 bounded

# we will give it a bounded bandwidth of 5% of our total incoming
bandwidth (10240*0.05=5120.0)
add_class 10:1 10:100 512kbit 51.2kbit 1 bounded

# second type of traffic SMTP,POP3 will be marked '2' by the
firewalling code
# we will give it a bandwidth of 15% of our total incoming
bandwidth (10240*0.15=1536.0)
add_class 10:1 10:200 1536kbit 153.6kbit 2

# third type of traffic ssh, telnet, ftp will be marked '3' by the
firewalling code
# we will give it a bandwidth of 10% of our total incoming
bandwidth (10240*0.1=1024.0)
add_class 10:1 10:300 1024kbit 102.4kbit 3

# last type of traffic is interactive traffic. It will be marked
'4' by the firewalling code
# we will give it a bandwidth of 70% of our total incoming
bandwidth (10240*0.70=7168.0)
add_class 10:1 10:400 7168kbit 716.8kbit 4

echo "...done"
echo ""

echo "--> IPTABLES firewall loaded/activated <--"

##-----------------End Firewall---------------------------------##
   ;;
   *)
      echo "Usage: firewall (start|stop|restart|status)"
      exit 1
esac

exit 0
```

C.2 TCPDUMP SCRIPT FOR DYNAMIC TRACING

```
#!/bin/sh

# set time limit based on what caller specified. Exec script that
will send
# SIGTERM to tcpdump to force this script to run the END block.
Background
# this so it doesn't interrupt gawk processing below.

# Invoke tcpdump with options and pipe through gawk to gather
data. The
# running of tcpdump is limited to the time specified by the
caller. I
# am only interested in the ip protocol packets. I will get the
source
# and destination addresses with the ''ip' specifier at $3 and $5
respectively.
# Do not track outgoing packets from this host as part of tracing
data. This is
# achieved by the 'src host not loghost' qualifier.

#
# I need to dump on every interface I find on system. so, call
ifconfig -a
# first, to get interface name. Call tcpdump on these.

INTERFACES=`/sbin/ifconfig | gawk ' {
        # Get the interface name
        x = split($1, ifname)
        newif[i]=ifname[1]
        if (match(newif[i], "eth") && newif[i] != "lo") {
                printf("%s ", newif[i])
}
        i = i + 1
} '`
for i in $INTERFACES
do
# for each interface check number of packets , if over threshold,
report
./dumper.sh $i $1 > /tmp/o_$i &
done
# kill this process in $1 amount of time
./trace_kill $2
sleep 3
/bin/cat /tmp/o_*
```

```
#rm /tmp/o_*

# This is the dumper program for host R99. Each of these is
slightly different# based on the /etc/hosts file.
/usr/sbin/tcpdump -i $1 -lnq ip src host not loghost and not
localhost and not r991 and not r992 and not r993 and not
192.168.13.1 and not 192.168.16.1 and not192.168.15.1 and not
192.168.11.1 2>/dev/null | \
gawk -v threshold=$2 -v interface=$1 '
{
        split($3, ip, ".")
        x=sprintf("%d.%d.%d.%d", ip[1], ip[2], ip[3], ip[4])
        source[x,interface] += 1
}
END {
        for (name in source) {
                if (source[name] >= threshold) {
                        split(name, ar, SUBSEP);
                        printf("%s %s %s\n", ar[1], ar[2],
source[name])
                }
        }
} '
```