An IP Address Management Solution for a Server Solution Provider

By
Matthew G. Doyle

A DISSERTATION

Submitted to

The University of Liverpool

in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

28th September 2005

ABSTRACT

An IP Address Management Solution for a Server Solution Provider

By
Matthew G. Doyle

Increasing demand for Internet access for businesses and individuals has seen an increasing demand on the IPv4 address space and made IP addresses a valuable asset. As with any asset, a company's IP address space must be managed well and protected from those who would damage or despoil it. A system to assist an organization with its efforts ought to be welcome indeed. The main goal of this project is to create such a system. The project analyzes some currently employed methods of IP address management and prevention of IP theft in particular. Methods such as layer-2-implemented theft prevention and passive reconnaissance tools are discussed and their shortcomings noted. A review of the pertinent literature is offered. The proposed solution, a combination of an active reconnaissance tool and record management system, is explained from both design and construction aspects. Finally, the system is evaluated and, based on its user reception and performance, recommendations are made for further work.

DECLARATION

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

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

Signed,

Matthew G. Doyle

ACKNOWLEDGEMENTS

First and foremost, I would like to thank the project sponsor for agreeing to allow me to work with them in the development of this project. Their openness to my questions, cooperation in getting employee feedback, and expertise at times when needed is greatly appreciated. It is not easy to allow someone to pick at weak spots (which were few and far between), probe security-sensitive topics, and get involved with the smallest details of day-to-day technical operations; diverting resources to assist in these activities must be even more difficult, and for that assistance I am deeply grateful. While much of the project was developed remotely, implementation does not happen without input from many people. I thank the technicians, network engineers, and system administrators who contributed to making this project a success. I must, of course, respect their request for anonymity, but I cannot let their assistance go unacknowledged.

I would also like to extend my sincerest thanks to my employer for assisting me with resources when necessary. Additionally, I thank them for time off when needed and other "sanity-enhancing" measures, which they have been very gracious in supplying whenever requested. The expertise and assistance supplied by my co-workers and colleagues has been invaluable. Many, many thanks to Jeremy Alons and Prescott Kulow for their assistance in brainstorming and troubleshooting problems in the code. Brandon Ewing is owed a debt of gratitude as well for help with certain particular sticky database issues. I am also grateful for the consistent help with testing supplied by Andrew Howard, Shannon Wittgreve, and Jeff Link. Last, but certainly not least, the input and support of Terrance Bush and Travis Schaffner—on professional, technical, and personal levels—is greatly appreciated. You are gentlemen and scholars, all.

On a final professional level, I absolutely must thank my dissertation advisor, Kathleen Kelm. Without her sage advice, firm grasp of the issues, and academic acumen, the entire project would have begun as blizzard of inconsistencies, languished in the doldrums of lack of focus, and, finally and tragically, disappeared in a puff of hardware and versioning problems. That it did not is due to her guidance, for which I am deeply grateful.

A journey not begun cannot come to a conclusion. For encouragement to take that first step three years ago, and consistent, unwavering support along the way, I thank my lovely wife, Mary. Without her determination to see me through the entire journey and, especially, without her support in its last stages, this project would simply not exist.

Finally, to my kids, Andrew, Nathalie, and Alanna, who never—not for the entire three years—complained that I was almost constantly perched in front of my computer, working on my degree, I have only this to say: Thanks for understanding.

Oh, and this: I'm back now.

TABLE OF CONTENTS

Page

LIST OF FIGURES

Figure Page

CHAPTER 1

INTRODUCTION

1.1 The Current Environment

The growth of network access in the past twenty-five years has seen remarkable developments in the technologies employed to make that access possible. Especially in the past fifteen years, this growth has expanded to, and then exploded upon, the public Internet. Increasing demand for Internet access for businesses and individuals has seen an increasing demand on the IPv4 address space and made IP addresses a valuable asset. As with any asset, a company's IP address space must be managed well and protected from those who would damage or despoil it. A system to assist an organization with its efforts ought to be welcome indeed. The main goal of this project to create such a system.

The goal set for the project is the somewhat narrower goal of assisting one particular organization, with the hope that the findings here can be expanded to assist other organizations. The organization at hand, the project sponsor, provides servers which customers rent on a monthly basis, and rack them at one of two high-bandwidth datacenters. For these rented servers (known as dedicated servers), the sponsor provides a range of service level agreements, ranging from completely managed (the customer does not even have administrative access) to completely unmanaged (the sponsor does not have administrative access). Additionally, it provides co-location service, whereby the customers provide their own servers and the sponsors charge for rack space, power consumption, and bandwidth. The sponsor does not have administrative access to these servers. The servers, as mentioned, are racked in one of two datacenters. Each datacenter has one aggregate switch and numerous branch switches. Each server is hooked to one of the branch switches which are in turn uplinked to the aggregate switch. This makes one broadcast domain at each aggregate switch.

Key to the smooth operation of a server solution provider with a large presence on the public Internet is the management of IP address space. The sponsor currently addresses this need through two Web-based interfaces into a MySQL backend database called the 'dedicated database'. When the sponsor receives an allocation of IP addresses from one of its upstream providers, a record for the address space (the 'base', usually a C-class) is added to the table base. This makes the addresses in that base available for assignment to a customer. When a server is ordered by a customer, a sales engineer allocates two IP addresses to it, which are bound to the server's adapter during the course of preparing the server for use by the customer. Records are added to the database

table `ipaddr` for those addresses, indicating that the addresses have been assigned. A customer can request additional IP addresses for which there is a monthly charge. A technician allocates requested IP addresses by using a Web interface known as ipmanage, finding a free IP address in the proper address range, and attempting to determine if the address is in fact unused.

To this end, the technician normally uses a tool known as arping. Written by Alexey Kuznetsov, this tool is widely distributed on many *NIX-type systems. With most Linux distributions, it comes as part of the iputils package. This particular tool sends out an Ethernet broadcast of ARP who-has packets, requesting the MAC address of the system with the IP address of interest. The arping utility is used rather than a simple ping; many of the sponsor's systems have disabled ICMP responses, especially the Linux systems, so a simple ping would be ignored too often to be useful in this regard.

If no response is received from the arping requests, it is assumed the IP address is actually free, and the technician completes a Web form which adds a record for the newly allocated IP address to the database table ipaddr. When an IP address is returned to the pool of those available for allocation (such as when a server is canceled or a customer no longer needs an extra IP address), this record is deleted and the Web interface shows that the address is available for reallocation.

1.2 Problems

A number of problems have become manifest within this particular environment. Potentially the most damaging--although certainly rarest--problem is the potential for a spammer or other network abuser to find an IP address not currently in use, bind it to his adapter, and violate any terms in the Acceptable Usage Policy he desires. The sponsor currently have no technical means of tracking down the source of such a stolen IP address easily; the only means of knowing the disposition of a given IP address is through the records kept in the dedicated database. Any need to track the location of an IP address outside of the records in the dedicated database means watching the traffic through one of up to 40 switches for the IP address--which would narrow the search down to 23 servers in the event that the IP address is found in the outbound traffic on the switch. Such a search can take hours or days, waste time, and allow the abusive activity to continue long enough to result in serious consequences.

A more common problem is allocating to a customer an IP address that has previously been allocated to another customer. Despite policies against it, it is not unheard of for a technician to simply rely on the

dedicated database when allocating an IP address. A related problem is failure to update the database when an IP address is allocated. This contributes to the previous problem and results in an IP address that is essentially 'lost'.

Finally, there is the possibility for any customer to take any unallocated IP address and bind it to their server. As long as the address is not involved in any network abuse, there is little chance that the sponsor would ever find out. While the IP address is, according to the ipmanage records, allocated, a technician getting an arping response from such an address simply marks the address as allocated (indicating '????' or something similar for the server label) and finds a different, non-responsive address to allocate. Given this situation, the sponsor essentially loses out on the revenue that the IP address should be generating, and valuable IPv4 address space is lost.

1.3 Potential Solutions

There are other approaches that can contribute to some of the aims of this project. One of the major aims is the prevention of 'stolen' IP addresses. To this end, the implementation of virtual LANs (VLANs) can prevent users from using unauthorized IP addresses. A more verbose description of this technology is presented in the next chapter. Briefly, however, in this environment, 'stolen' IP addresses will not work outside of the branch switch, as the packets would be stopped at the router.

A second current method is much closer to this project in its implementation. A tool known as arpwatch, authored primarily by Richard Leres from the Lawrence Berkeley National Laboratory, is widely used to monitor Ethernet traffic. From an examination of the code and the information in the man pages, arpwatch passively watches for ARP traffic on the network segment. A few events are considered significant and will trigger an email to an address supplied as an argument, including the first occurrence of a MAC on a network, and a change in IP-address-to-MAC pairings.

1.4 Proposed Solution

The current project proposes a different approach to those outlined. The project begins with a brief discussion of the motivation behind creating a new solution and then goes on to touch on the main features of the solution, which will be described in more detail in the ensuing chapters.

1.4.1 Why current solutions are inadequate

IP address management is clearly not a new idea or need, and the above potential solutions each have worth to contribute to a complete solution. However, for varying reasons each of them fail to provide an adequate solution for the provider.

Given that the main purpose of the project at hand is to prevent or quickly detect the misappropriation of an IP address, it is not surprising that the leading contender for a solution for this provider was the implementation of VLANs. The potential exists with this technology to make it technically impossible for IP addresses to be stolen. Since the address will not route unless it has been added to the VLAN at the switch, only authorized addresses will be able to send any kind of traffic outside of whatever smaller, private environment they might have.

There are a number of reasons why this approach would not be appropriate in the sponsor's current environment. The major drawback to this is that, to implement it in a sane fashion, all address space would need to be reclaimed and all devices renumerated with a new allocation schemata. Currently, this would mean reclaiming and reallocation over 12,000 addresses—a task clearly beyond the sponsor given current staffing levels.

Further, if the assignment of addresses is to be logical, it would be extremely desirable for the sponsor to have it own autonomous system number (ASN) from the regional Internet number registry (ARIN) and a direct allotment from ARIN of IPv4 address space. Currently, the sponsor receives its address space in allotments from upstream providers. Those allotments come in a hodge-podge of random subnets that would make intelligent reallotment to customers difficult at best; at worst, fractured and extremely difficult to manage.

Finally, address security is but on facet of the sponsor's current problem. Other problems exist in that the 'technical' aspect of allocating an address is divorced from the 'recordkeeping' aspect of the task. Given this division, it is entirely too frequent that only one of the tasks is accomplished. While the implementation of VLANs is, perhaps, the very best way to solve the IP address theft problem, it does nothing to alleviate the other issues facing the sponsor.

The existence of such tools as arpwatch, discussed above, have the virtue of being much simpler to implement. However, existent tools still fall short of meeting the sponsor's needs in the current environment. Tools such as arpwatch fall into the passive reconnaissance category of Schiffman's (2003, p. 5) taxonomy of network security tools. Clearly,

passive reconnaissance tools are not intrinsically flawed. However, the sponsor's belief was that the passive approach could easily lead to a situation in which the single alert sent when an event occurred in the passive paradigm would too frequently be ignored. The sponsor felt that an active tool would better suit its needs.

Another weakness to the arpwatch approach is the alerts themselves. These take the form of an email sent to an address supplied as an argument. The issue for the sponsor is that each occurrence of a new MAC, new IP-MAC pairing, and other events would lead to an emailed alert—whether or not the event was of interest, and without regard to whether the event was expected. As an example, consider the barrage of emails to the sponsor's ticket system that would have occurred when, as recently occurred, a new server with 30 pre-ordered IP addresses came online. The real concern here was that the technicians would be so used to 'the boy who cried wolf' and being peppered with arpwatch emails, that no action would be taken when the issue was one that required immediate attention.

Moreover, the 'databases' in which arpwatch stores the IP-Ethernet address pairings are flat files. Flat files are relatively inefficient when used to access the amount of data with which the sponsor is dealing. Further, the flat-file paradigm leads to a situation in which the information cannot easily be accessed for other purposes (such as IP address-to-customer name lookups). Finally, a solution based on arpwatch, like that of VLAN implementation, does nothing in the arena of improving recording keeping by making the allocation of addresses a technical as well as a recordkeeping process, centralizing the process, and making it more likely that both tasks will be completed.

An ideal solution for the sponsor will combine a relatively simple and low-cost implementation, elimination of the 'two-aspect' angle of the allocation process (recordkeeping and technical), and a robust method of preventing the theft of IP addresses. The next section discusses how, with a general approach similar to arpwatch, the project will be implementing the idea in a different fashion and thus overcome these inadequacies.

1.4.2 The proposed solution

This paper proposes a solution to the sponsor's problem that will combine the 'technical' and 'recordkeeping' aspects of IP address allocation into a single act. This will reduce (to the point of eliminating) the egregious errors that lead to permanently lost IP addresses. Rolled into this solution is an active reconnaissance tool that will periodically scan all of its assigned IP address space searching for anomalies in the

responses. A final feature is the ability to use the solution to scan for problems and immediately use the solution to query for further information. For example, given an IP address that is reporting an unauthorized MAC address, the technician could immediately use the solution to find out what MAC is authorized for that IP address; what other IP addresses are authorized for that particular MAC; and ultimately, if desired, all customers involved in the incident.

The central portion of the solution is a program known (for lack of a better name) as IPManager, or ipman (so as not to confuse things too much with some of the sponsor's current solution segments, namely ipmanage). A small program written in C, ipman has three core functions: IP querying, IP binding and unbinding, and accessing other information.

Additional pieces of the solution include two databases. One of these, ipman_db, has been created expressly for the ipman solution. The second of these, ipmanage_ipplan, has been in use for years as the core of the IP address management system. This has been taken from IPplan, an IP address management system available under the GNU General Public License at http://sourceforge.net/projects/iptrack.

1.4.2.1 IP querying

This function is really the core solution to what the sponsor saw as its main problem. The implementation of this function involves ipman scanning, in turn, each IP address in its list of IP addresses, kept in a database table. For each address that is marked as one to scan, the program sends an ARP request packet, to find out what MAC address will respond as having a given IP address. The session is filtered to capture only packets with a source IP address of the packet being scanned. If a packet is received, it is disassembled, and the MAC address of the respondent recorded. This is repeated five times for each address in the table. If the MAC address field in all received packets match and agree with the MAC address that should be responding according to the database, the next address is scanned. If the MAC address fields of all packets do not agree, or there is disagreement with the value in the database, then an IP-MAC mismatch will be reported for that IP address. If no responses are received for a given IP address, that address is written to a table of 'non-responders', which are scanned in the next two passes. Finally, a table containing IP addresses recently allocated to customers that have never responded to a scan is scanned and appropriate action taken.

1.4.2.2 IP binding and unbinding

At the heart of the entire system lies the server that actually runs ipman, known as the IP host server. On top of having ipman and the associated ipman_db database installed, the IP host server has all unallocated IP addresses bound to its adapter.

The process of allocating an IP address to a customer, then, consists of a technician unbinding the address from the IP host server. Before the IP address is actually unbound, ipman checks to see if is already unbound (and therefore, officially allocated to a customer or to internal usage). If it is not, ipman prompts the technician for information; to which server the IP address is being released, the server's location, and so on. Once all necessary information is obtained and everything checks out, ipman will unbind the address from the IP host server's adapter, and record the address as an unbound address.

The reverse process takes place when an address is being reclaimed from a customer. A technician will run ipman and have it bind an address to the IP host server. Again, ipman checks to ensure that the address is not already bound. If it is not, ipman performs a look-up to see to which server it is currently assigned. The technician is prompted for confirmation that the address currently assigned to a given server is to be unassigned and bound to the IP host server. Once confirmation is received, ipman updates necessary database tables and binds the IP address to the IP host server's adapter.

1.4.2.3 Accessing other information

This functionality is wide open, and can encompass as much as the sponsor (or any other user) would desire. The ability to query an IP address is already built in to ipman. With structures already imbued within the program, it is possible to extract a great deal of information. For example, given an IP-MAC address mismatch, a query of ipmanage on the IP address can yield the server label; given a MAC address, a query on ipman's native database can yield all IP addresses authorized to that MAC. In short, given either an IP address or a MAC address, the access provided by the functionality within the program leaves very little information that cannot be extracted from within the ipman environment.

1.4.3 Benefits of the proposed solution

The proposed solution has a number of advantages over the alternatives considered. It is relatively simple and inexpensive to implement. While it does not offer the iron-clad prevention of IP address theft that, for

examples, VLANs can assure, it is a robust solution that will always discover such theft—in acceptably short time, and with minimal effort.

Despite the strengths of any solutions discussed, none of them integrate the technical and recordkeeping aspects of the IP address allocation process. Both aspects are critical to IP address management; no other considered solution covers both.

Further, the proposed solution offers additional information accessibility within the same environment, affording technicians the ability to query the databases in the solution for a range of information. This functionality is simply not a part of the design of alternative solutions.

1.5 Conclusion

This chapter has reviewed the current environment for a server solution provider and discussed some of its issues with IP address management. It briefly discussed potential solutions for these problems and touched on their weaknesses. Finally, it outlined the proposed solution and its potential benefits. In the next chapter, it will review some of the literature on the various aspects of the alternative solutions, as well as available literature as it pertains to the design and construction of the proposed solution.

CHAPTER 2

Review of Literature

2.1 Review of Alternative Solutions

There is not a wealth of published material in refereed materials on the management of IP address space. The thrust of the review done was on materials that would explain how to actually design and construct the solution being implemented for the sponsor. Many sources indicated that the preferred method of IP address 'theft' was the usage of either VLANs or, where that was not feasible, the employment of a tool such as arpwatch. A review of the literature on these alternative solutions follows.

2.1.1 VLANs

The implementation of virtual LANs (VLANs) can prevent users from using unauthorized IP addresses. VLANs, are essentially logical collections of end stations that communicate directly without a router (Syngress Media, 1998, p.393). One of the more popular uses of VLANs is to enable the grouping of end users without regard to their physically connection to the network. A user participating in a VLAN can plug into a separate location and, without further configuration, continue participation in the same VLAN.

VLANs can be assigned based on port, MAC address, user ID, or network address (Syngress Media, p. 395). Assignment by port is the most common method. This is the way VLANs would be assigned in the sponsor's environment, where, again, the largest concern is the prevention of IP address theft. Figure 1 illustrates the VLAN environment.

In the VLAN environment, each individual server would be allocated its own subnet, and addresses would be non-portable to other servers. Each branch switch would be a separate VLAN. If a packet is not bound to another port on the same branch switch, it gets forwarded to the aggregate switch. There the packet is tagged based on the port it came in one with an 802.1q tag (the VLAN ID) (Barnes and Sakandar, 2005, pp. 84-92). The packet would then get trunked to the router.

Trunking is a method of sending packets from multiple VLANs over a single physical connection. It has the benefit of obviating the need for multiple physical connections necessary to achieve the same effect without trunking. In a sense, trunking can be viewed as form of

multiplexing similar to multiple broadcast signals being multiplexed onto the public airwaves (Barnes and Sakandar, p.88). Because both the switch and the router are using a trunking protocol, they can use these connections to send trunking information and, therefore, information for multiple VLANs over the same wire.

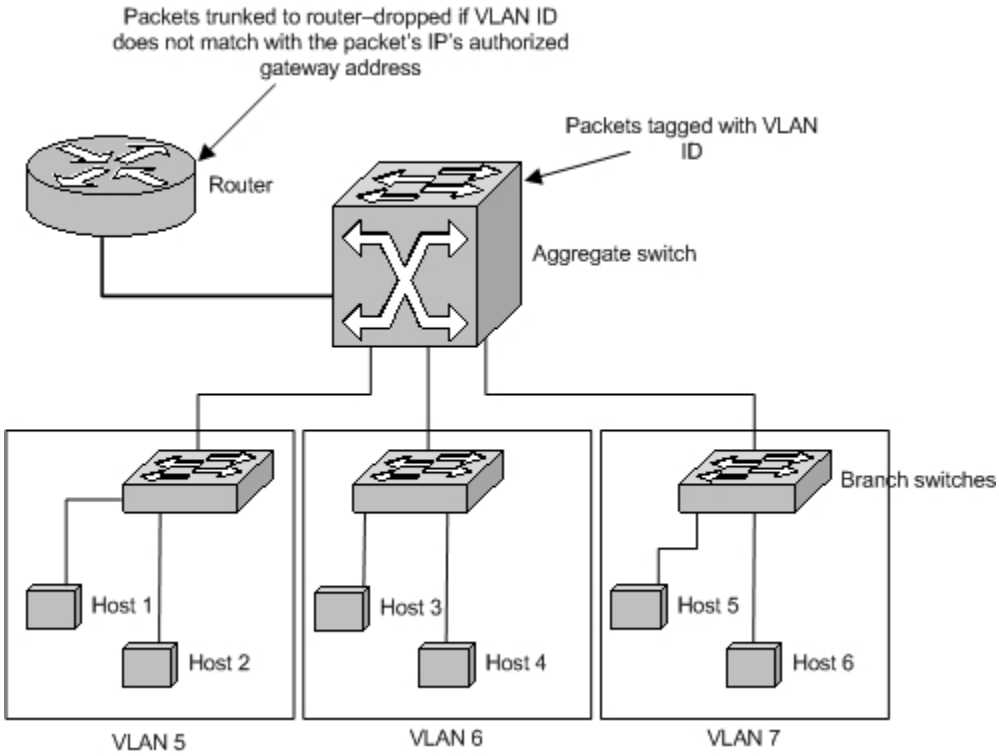

Figure 1. A VLAN implementation.

Once the packet is trunked to the router, the tag with the VLAN ID is removed and examined. (Barnes and Sakandar, 2005, pp. 84-92) At the router level, the gateway address of each address allocation has been 'authorized' for each VLAN. If the ID in the 802.1q tag does not match the authorized VLAN, the packet is ignored. In this environment, 'stolen' IP addresses will not work outside of the branch switch, as the packets would be stopped at the router.

As an example, assume Host 1 in Figure 1 is allocated the IP address block 64.38.0.4/29. The gateway address 64.38.0.5, 64.38.0.6 is the useable address, and 64.38.0.7 is the broadcast address. A subinterface on the router has the gateway address 64.38.0.5 bound to it, and it is tagged with the VLAN ID 5. Assume now that Host 3 binds that address to its adapter and tries to use it. When it sends a packet to a host not on the same branch switch, the packet gets forwarded to the aggregate switch. There it is tagged (as all packets coming to that port) with the VLAN ID 6, and the packet is trunked to the router. The router reads the

ID, sees that VLAN 6 is not authorized to use gateway address 64.38.0.5, and ignores the packet.

An added benefit to the VLAN approach is without VLAN implementation, all hosts on all branch switches are in one large broadcast domain: All hosts receive all broadcasts. A VLAN is a single broadcast domain—no broadcast traffic will pass to other VLANs, since VLAN-to-VLAN traffic must pass through the router. Thus, implementing one VLAN per branch switch means that the sponsor's environment has moved from one broadcast domain at the aggregate switch to one broadcast domain per branch switch. This would provide a marked improvement in the traffic on the network.

2.1.2 Arpwatch and similar programs

While any documentation on this tool is rare and there is little more to be found than the man pages, a look at the code (this code from version 2.1a13) indicates that, in normal mode, the tool runs as a daemon, judging from lines 232 to 250 of arpwatch.c:

```
if (!debug) {
                  pid = fork();
                  if (pid < 0) {
                        syslog(LOG_ERR, "main fork(): %m");
                        exit(1);
                  } else if (pid != 0)
                        exit(0);
                  (void)close(fileno(stdin));
                  (void)close(fileno(stdout));
                  (void)close(fileno(stderr));
#ifdef TIOCNOTTY
                  fd = open("/dev/tty", O_RDWR);
                  if (fd >= 0) {
                        (void)ioctl(fd, TIOCNOTTY, 0);
                        (void)close(fd);
                  }
#else
                  (void) setsid();
#endif
```

Based on the excellent explanation by Schiffman (2003, p. 14) of the libpcap packet-capture library, Line 272 opens a libpcap packet-capture session in which the host's adapter is set to promiscuous mode:

```
pd = pcap_open_live(interface, snaplen, 1, timeout, errbuf);
```

Finally, after setting some signals, the program enters a packet-capture loop on lines 334 or 338 (depending on the link type found for the packet-capture session):

```
status = pcap_loop(pd, 0, process_ether, NULL);
```

or

```
status = pcap_loop(pd, 0, process_fddi, NULL);
```

From an examination of the code and the information in the man pages, arpwatch passively watches for ARP traffic on the network segment. Certain events are considered significant and will trigger an email to an address supplied as an argument: The first occurrence of a MAC on the network; a change in the IP-MAC pairing recorded; a flip-flop (i.e., reversion to a just-previously-used MAC address); and the reappearance of a MAC after 6 months of dormancy are some of those events.

2.2 Review of the Proposed Solution

The heart of the current solution relies on the notion of being able to identify a machine using a given IP address. The sponsor's entire installed base is connected via Ethernet and uses TCP/IP as its sole communication protocol. Thus, the decision was made to use the Address Resolution Protocol (ARP) to determine the media access control (MAC) address of the machine using a given IP address.

2.2.1 Address Resolution Protocol

ARP was initially proposed in 1982, in RFC 826 by David C. Plummer. The motivation behind proposing ARP was the expanding usage of Ethernet as a transmission medium and the greater number of manufacturers supplying interfaces for the increasingly popular Ethernet standard. Given that higher-level protocols were independent of the physical transmission medium, some form of translating the high-level addresses to the low-level, physical address was needed. Plummer's observation was that, as more software was being written for Ethernet interfaces, the implementers could either all create their own version of address resolution, or they could all use a standard (Plummer, 1982). The purpose of RFC 826 was to create that standard.

Given the motivation, it is hardly surprising that the RFC does not actually discuss implementation. This has been left largely to the implementers. The only real discussion of the implementation details is the usage of fields in Ethernet packets to incorporate ARP into Ethernet functionality, as well as a general algorithm of how ARP might be implemented. A review of the structures involved in ARP follows.

2.2.2 Ethernet frames

As a brief review, an Ethernet frame has the following structure (Kurose and Ross, 2003, p.456):

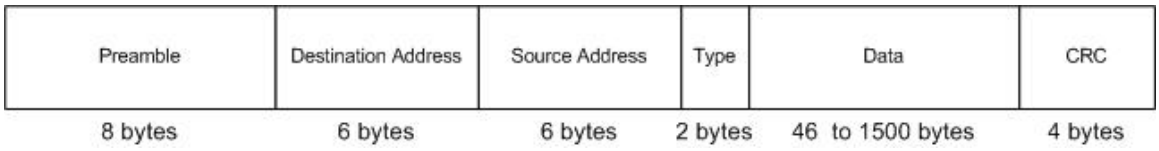

Figure 2. Ethernet frame.

The preamble is a field that simply serves to 'wake up' the receiving host and synchronize the transmission rates. The destination and source address fields contain the Ethernet (MAC) addresses of the hosts involved in the exchange. The type field allows Ethernet to carry data for a variety of network protocols. This field contains a code indicating that the higher-level protocol is IP, IPX, AppleTalk, or any number of other protocols. It will be sseen shortly how this value will be set to indicate that the protocol is ARP when packets are sent, and later, examine this field to determine whether a captured packet is an ARP packet. The data field contains the actual payload for which the Ethernet frame is being constructed. The final field, the CRC, is a 4-byte checksum used to determine whether the data has been corrupted during transmission.

It is the data field of the Ethernet into which the data discussed the next section, the ARP packet, will be packed.

2.2.3 ARP packets

The structure of an ARP packet is as follows (Plummer, 1982):

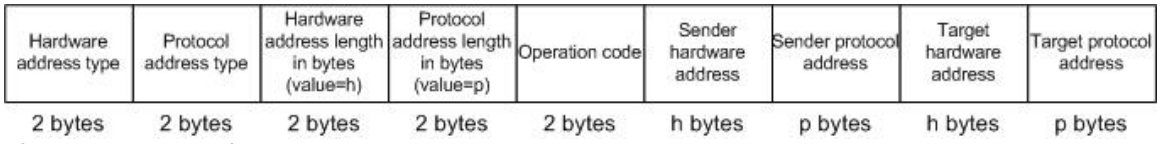

Figure 3. ARP packet

These fields are, for the most part, self-explanatory. The hardware address type in all examples will be Ethernet. ARP was originally proposed as a general solution for resolving high-level protocol addresses to physical addresses, so one could include in this field some other physical address type, such as Packet Radio Network. The protocol address type field specifies exactly that; in the present program it will contain IP, although other protocols are possible here as well in other applications. The next two fields contain the length in bytes of the hardware and protocol addresses. The next field contains the operation

code. This will be, for present purposes, a code that indicates the ARP packet is either an ARP request or an ARP reply.

The sixth and eighth fields contain the hardware addresses of the sender and receiver respectively. These fields are the same length as the value contained in the third field, the hardware address length. The seventh and ninth fields are the source and destination protocol addresses, with field size equal to the value of the fourth field, the protocol address length. For present purposes, the hardware address is the MAC address, and the protocol address is the IP address.

2.2.4 Packet injection

In order to query an IP address for the host's MAC address, an ARP request packet must be built with proper values and put on the wire. Then the program must capture the reply packets, and extract the necessary information from them.

Packet assembly at the lowest level can be a difficult, tedious, and error-prone process. Fortunately, there are libraries available to assist the application programmer in packet assembly and injection. One such library is libnet, an open-source project providing a C API to create packets and write them to the wire. Libnet is distributed under the BSD license. This library is available for download at the URL http://www.packetfactory.net/libnet/. The program will use this library to create an ARP request packet, which will broadcast a request to reply to the sender with the MAC address of the holder of the IP address the program is scanning.

The actual process of building the ARP request packet consists of, first, declaring a pointer to type `libnet_t`. `libent_t` is a typedef of the `libnet_context` structure. This structure is libnet's "main monolithic control data structure that describes a complete libnet packet shaping/injection session" (Schiffman, 2003, p. 40).

Generically, a packet creation and injection session would follow the general steps laid out here. First, the program creates a libnet session, and gets the IP addresses (source and destination) stored in the proper variables.

```
libnet_t *context;
char dest_mac[ ETH_ALEN ]={ 0xff, 0xff, 0xff, 0xff, 0xff, 0xff };
char src_mac[ ETH_ALEN ];
struct libnet_ether_addr *ptr_hwaddr;

context=libnet_init(LIBNET_LINK,"eth0",libnet_error_buf);
temp_ip = libnet_name2addr4( context, str_ip, LIBNET_DONT_RESOLVE );
```

```
memcpy( src_ip, ( char * ) &temp_ip, IP_ALEN );
temp_ip = libnet_name2addr4( context, ip_addr, LIBNET_DONT_RESOLVE );
memcpy( dest_ip, ( char * ) &temp_ip, IP_ALEN );
// get the source MAC address
ptr_hwaddr = libnet_get_hwaddr( context );
memcpy(src_mac, ptr_hwaddr, ETH_ALEN);
```

With the session created and the variables defined, it is time to create the packet to send.

```
arp = libnet_build_arp(
      ARPHRD_ETHER,   // Hardware address type
      ETHERTYPE_IP,   // Protocol address type
      ETH_ALEN,       // Hardware address length
      IP_ALEN,        // Protocol address length
      ARPOP_REQUEST,  // Operation code
      src_mac,             // Sender hardware address
      (u_int8_t *)src_ip,  // Sender protocol address
      dest_mac,            // Target hardware address
      (u_int8_t *)dest_ip, // Target protocol address
      NULL,                // Payload --  none in this case
      0,                   // Size of the payload -- size of nothing
      context,
      0);       // This last argument explained below
```

Compare this to the ARP packet depicted in Figure 3 on Page 20. We are filling in the ARP packet's fields, in order, with constants defined in libnet and values assigned earlier. The last argument is of type `libnet_ptag_t`. This is a protocol tag identifier. All libnet packet building functions return this data type, and all accept it as an argument. To change something slightly in this header (or in the data, if there were any), one could pass `arp` (declared as `static libnet_ptag_t` so it will be available next time around, if need be) as the final argument after modifying whatever needed modification. Since it is not necessary to modify packet parameters, 0 is passed for this last argument.

With the ARP packet assembled, the program now builds the Ethernet frame.

```
eth = libnet_build_ethernet(
      dest_mac,
      src_mac,
      ETHERTYPE_ARP,
      NULL,
      0,
      context,
      0);
```

Again, the correspondence to the depiction of an Ethernet frame in Figure 2, Page 20, is almost one-to-one. The preamble is taken care of by the libnet library, as is the CRC at the end of the packet. The data, here

the NULL argument, is actually the ARP packet just built above. The last three arguments are analogous to those in constructing the ARP packet.

Finally, the program writes the packet to the wire and cleans up the session:

```
libnet_write( context );
libnet_destroy( context );
```

2.2.5 Packet capture

Packet capture and disassembly can be just as difficult and error-prone as packet creation and injection. An analogous library, libpcap, offers an API to the application programmer and is included in most Linux distributions. The program employs libpcap to capture packets, filters them so examines only those that are of interest, and then disassembles them to extract the MAC address of the respondent.

The program first opens a pcap capture session as it did with libnet packet creation. `pkt_descriptor` is analogous to the `libnet_t` structure; it is the structure that contains all the necessary information for a packet-capture session. The variable `filter` allows us to set a filter on the packets that will actually be captured. In the present case, they need to be ARP packets, and the source host must be from the IP address in `ip_addr`. `pcap_compile` processes the filter and prepares it for employment. Finally, `pcap_setfilter` sets the filter on the session.

```
pcap_t *pkt_descriptor;        // like a file descriptor for packets
struct bpf_program prog_buff;  // space for compiled filter
char *filter;                  // the filter to pick out only
                               // interesting packets
char error_buf[ PCAP_ERRBUF_SIZE ];
const u_char *packet;
int MAX_SIZE_FILTER = strlen( "arp src host " ) + MAX_SIZE_IPADDR + 1;

pkt_descriptor = pcap_open_live(interface, BUFSIZ, 0, 500, error_buf);
```

In the following lines, a filter on captured packets is created, compiled and set in the packet capture session. The filter implements the *Berkley Packet Filter* (BFP) filter programs (Schiffman, 2005, p. 18). The filter is compiled into the area allotted to it in `prog_buff` and then set in the session.

```
sprintf( filter, "arp src host %s", ip_addr );
pcap_compile( pkt_descriptor, &prog_buff, filter, 1, 0 );
pcap_setfilter( pkt_descriptor, &prog_buff );
```

Finally, the packet capture session is initiated with a call to `pcap_dispatch`. This function takes as arguments the session structure; the number of packets to capture (0 indicates that it should keep capturing); a callback function that is called to handle each packet; and finally a pointer to `u_char`. This argument is actually passed by `pcap_dispatch` to the callback function specified by the previous argument.

```
pcap_dispatch( pkt_descriptor, 0, ( void * )pcap_callback_fct, NULL );
```

The callback function is used to process the packets. This can really be whatever the application programmer needs. The callback function receives as arguments the user-defined pointer to `u_char` describe above, as well as pointers to the `pcap_t` structure and the captured packet. We will discuss the callback function later, as it is an implementation detail.

2.2.6 Database access

The MySQL database server is ubiquitous in the sponsor's environment, as well as being one of the most widely-implemented open-source database platforms in the world. The MySQL implementation of an API for C is quite simple to use. There are essentially four steps to running a query on a MySQL database from within a C program (MySQL AB, 2004). First, a session is initiated. Secondly, a connection is made to the server. The next step is to construct and execute the query. Finally, the session is ended by destroying the resources allocated to the session.

As in the previous libraries, the API for MySQL contains one monolithic structure that is the repository for all the major data and components of a MySQL session. This is the MYSQL structure, and it represents a handle to a database connection (MySQL AB, 2004, p. 777). It is used is almost every function in the mysqlclient library. Another very common structure in MySQL sessions is the `MYSQL_RES` structure. This represents the result set of a query that returns rows (for example, SELECT or DESCRIBE). Below is an example of a short MySQL session that returns one row and prints the results. Memory allocation and error checking, while essential, have been omitted for brevity.

```
MYSQL conn;
MYSQL_RES *res;
MYSQL_ROW row;
char *server = "localhost";
char *mysqlUser = "my_user";
char *mysqlPass = "my_pass";
char *mysqlDB = "my_database";
char *mysqlQuery;
```

```
mysql_init( &conn );
mysql_real_connect( &conn, server, mysqlUser,
      mysqlPass, mysqlDB, 0, NULL, 0 );
sprintf( mysqlQuery, “SELECT field2, field3 FROM table1” );
mysql_real_query( &conn, mysqlQuery, strlen( mysqlQuery ));
```

These are the first three steps mentioned. The following four lines are not discussed above. These lines handle the returned information. There are clearly as many variations on this type of code as there are queries and users. What follows is just a very simple sample.

```
res = mysql_use_result( &conn );
row = mysql_fetch_row( res );
printf( “field1 = %s\n”, row[ 0 ] );
printf( “field2 = %s\n”, row[ 1 ] );

mysql_free_result( res );
mysql_close( &conn );
```

2.2.7 Final considerations

An attempt has been made in this project to follow the best programming practices—although, due to lack of experience, this is not always apparent.

Whenever possible, local variables have been used, rather than external variables. The usage of external variables is “fraught with peril…[external] variables can be changed in unexpected and even inadvertent ways.” (Kernighan and Ritchie, 1988, p. 34). A couple of external variables, as exceptions to this rule, have been employed, but for good reason. Further, the small number of external variables—and the prominence of the information they store—make “inadvertent” changes very unlikely.

The program has been broken down into functions whenever this was determined to be a real possibility. In general, an attempt has been made to strike a balance between code readability, ease of making changes to the program, and performance. Large numbers of function calls can have a negative impact on a program’s performance (Deitel and Deitel, 2003, p.208). However, a large number of small, concise functions aid in program maintenance, debugging, and readability. In the present case, both of these considerations carries equal weight, and the functionalizing of the solution should be evaluated accordingly.

2.3 Conclusion

This chapter briefly reviewed the literature on some alternative solutions to the sponsor’s problem. It discussed how VLANs work to prevent IP address misallocation as well as bringing other benefits. The chapter

then reviewed available materials on arpwatch, a passive tool that watches for certain predefined key events to occur. Finally, some of the literature was reviewed which contained insight into designing and constructing the proposed solution. The next chapter discusses that design.

CHAPTER 3

Analysis and Design

3.1 Overview

The current solution has two major goals. The first of these, and the one which the sponsor considered to be the main goal of the project, is a system that will periodically scan for unauthorized usage of IP addresses. 'Unauthorized usage' in this context means that an IP address is bound to a server to which that IP address was not allocated.

The second is to consolidate the two aspects of IP address management—the technical aspect, and the recordkeeping aspect. The sponsor's technical staff members are much better at carrying out whatever technical challenges face them than they are at keeping good records. This was a major finding of the analysis of the current environment—that many of the issues associated with address management stem from a failure to carry through on the 'clerical' aspects of things.

With these two goals in mind, the design of the project became fairly clear. The sponsor needed a solution that could combine and centralize, to the extent possible, all functions associated with IP address management. The review of the environment made it clear that there are three functions involved in IP address management. Those functions are allocation to the customer, reclaiming from the customer, and auditing the usage of all addresses, whether allocated or not.

The centralization of these three functions necessitated a specialized database to help maintain the records needed. Auditing the usage of IP addresses, allocated or not, implies keeping records of not only to whom addresses are allocated; it is also necessary to keep track of which addresses are not allocated, and to make sure that those are not used.

Indeed, the sponsor already has deployed a database that is used to track IP allocation. The database is ipmanage_ipplan, and is derived largely (if not entirely) from an open source project called IPplan. IPplan is an IP address management system available under the GNU General Public License at http://sourceforge.net/projects/iptrack. This database uses one table to track allocated IP addresses. The usage of this table is telling: When an IP address is allocated, a record is inserted into the table for it, and when the address is reclaimed, the record is deleted: There is no way to track unallocated addresses within this schema.

This database is, all the same, central to the sponsor's tracking of IP address allocation, and therefore—for the near future, at any rate—the database needs to remain part of the solution. A brief description of the tables that touch on the present solution, then, is in order.

The main table that concerns us is `ipaddr`, described below:

```
+-----------+------------------+------+-----+---------+-------+
| Field     | Type             | Null | Key | Default | Extra |
+-----------+------------------+------+-----+---------+-------+
| ipaddr    | int(11) unsigned |      | PRI | 0       |       |
| userinf   | varchar(80)      |      |     |         |       |
| location  | varchar(80)      |      |     |         |       |
| telno     | varchar(15)      |      |     |         |       |
| descrip   | varchar(80)      |      |     |         |       |
| baseindex | int(11)          |      | PRI | 0       |       |
| lastmod   | timestamp(14)    | YES  |     | NULL    |       |
| userid    | varchar(40)      |      |     |         |       |
+-----------+------------------+------+-----+---------+-------+
```

The field `ipaddr` is an integer representation of an IP address. `userinf` is a field that represents a description of the 'user' to whom the address is assigned. The sponsor uses this for the label put on the server when it is racked, which sometimes serves as a key. The `location` field identifies the datacenter and rack in which the server is located. The field `descrip` is a generic field for further information—the sponsor uses it to describe the relation of the assignee to the organization. Examples of data in this field include 'Dedicated server', 'Co-location', 'Internal', and 'Assigned to (staff member)—testing'. The field `baseindex` serves as a tie-in to the table base; in a current and properly designed database, this would actually be a foreign key referencing the primary key `baseindex` in the table base. The field `lastmod` is a timestamp that is updated by the MySQL engine each time there is an INSERT or UPDATE performed on the row. `userid` is the login name of the staff member that last updated the row (via INSERT or UPDATE).

The table `base` has little real impact on the present project. Essentially, each entry in the table represents an IP address net block allocation to the sponsor from the upstream provider. Almost all of these are C-class address blocks. There are a handful of allocations that are not C-class blocks, but these are considered legacy. As they are already permanently allocated to internal devices, they do not concern this project. The table `base` will only be used when an additional C-class address is allocated to the sponsor. In this event, the address block needs to be provisioned on the IP host server and added to the ipmanage_ipplan database. We will touch on the specifics of adding address blocks later.

Central to the workings of the current solution is the database ipman_db. While ipman does interact with ipmanage_ipplan, ipman_db is the core data store for the solution. Therefore, an understanding of the structure of ipman_db is indispensable to understanding the design of the program itself.

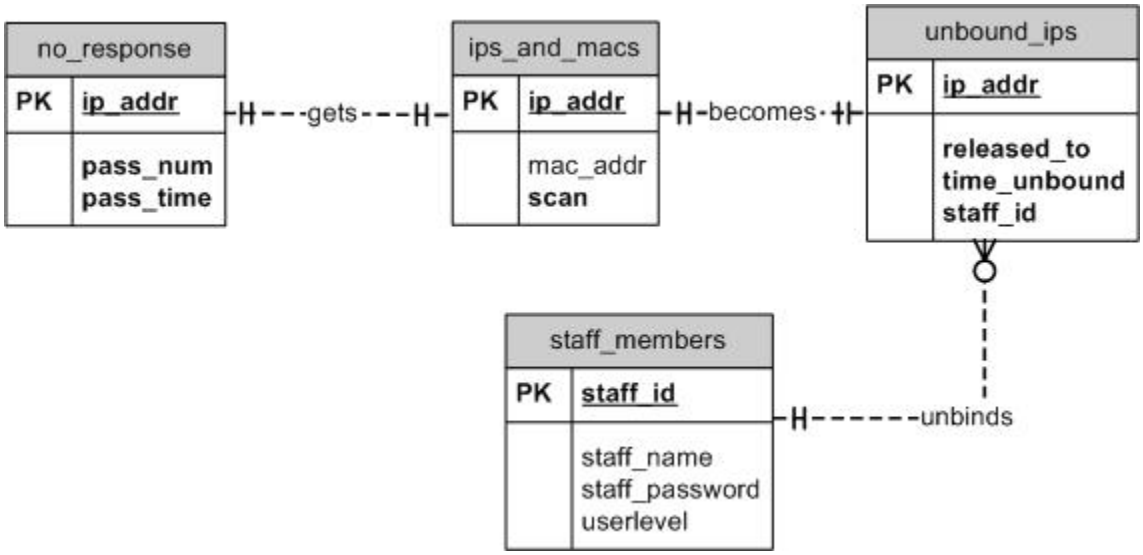

Figure 4. Entity relationship diagram for ipman_db.

The central table is `ips_and_macs`. This table contains each IP address in the IP host server's broadcast domain. The address is stored as an unsigned integer in the field `ip_addr`. The field `mac_addr` holds the MAC address of the IP address once it is known; when an IP address is first added to the database, the MAC address may not be known. This is the case, for example, when the entire system is first being initialized. The final field, `scan`, is essentially a Boolean representation of whether the IP address ought to be scanned.

The table `no_response` is used as a sort of temporary table during a scan to track which IP addresses did not responded during the previous pass. The fields are self-explanatory.

The table `unbound_ips` contains IP addresses that have recently been allocated to a customer, but have not yet responded to an ARP query since being unbound from the IP host server. The primary key `ip_addr` is, of course, the address as an unsigned integer. The field `released_to` contains information on the individual using the IP address. Under all but the most unusual circumstances, this would be the server label, the same information that is stored in the `ipaddr` table's `userinf` field in the ipmanage_ipplan database.

The final table contains information on the users of the system. The field `staff_id` serves as the primary key. `staff_name` is used as the username, and `staff_password` stores the user's password as an MD5 hash. Finally, `userlevel` can be used as a value to limit access to certain functions, as the administrators of the system choose.

An overview of the entire system is provided in the context diagram below.

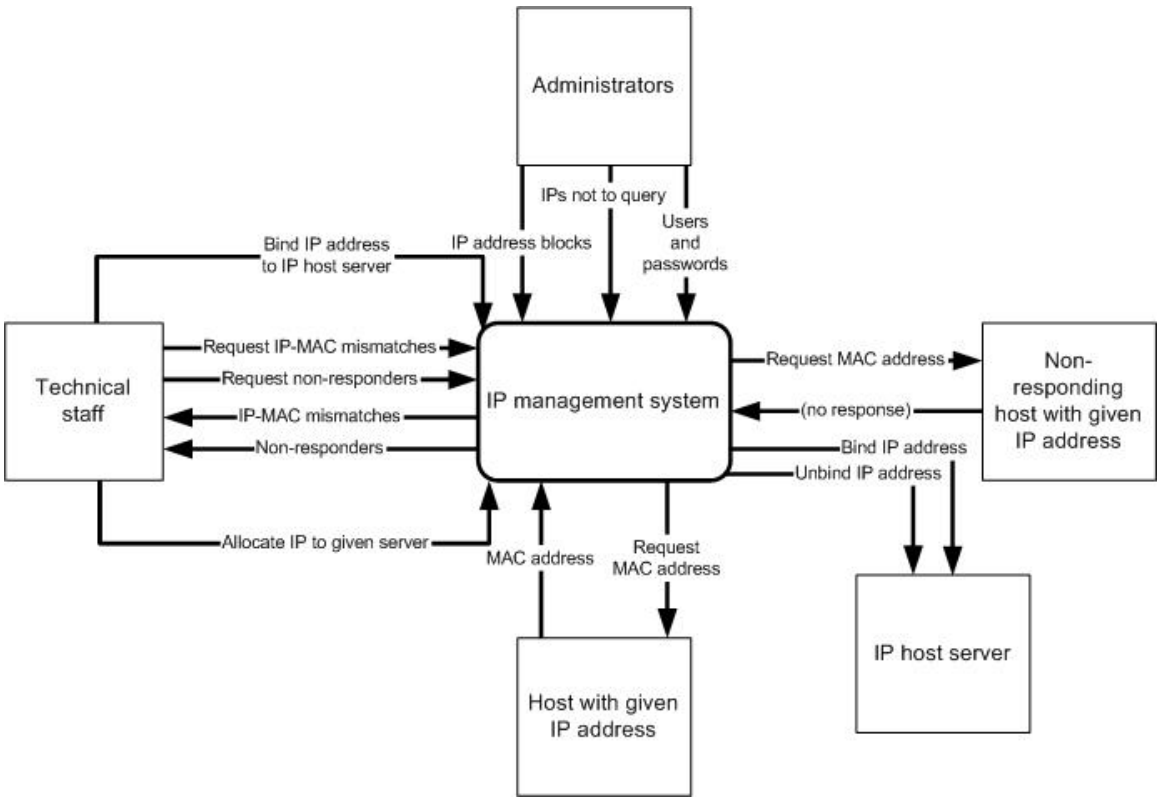

Figure 5. IP address management system context diagram.

The bulk of communication occurs between the technical staff and the IP management system. The technical staff will periodically request IP-MAC address mismatches, as well 'non-responders'—addresses that do not elicit a reply to an ARP request. In response to this request, the IP management system sends ARP requests to all the IP addresses in its `ips_and_macs` table, provided that the value of the scan field for a given address indicates it ought to be scanned. After a number of passes, the IP management system reports to the technical staff the IP addresses elicited a reply from an unauthorized MAC address as well as those which elicited no response at all.

Additionally, the technical staff can request that the system bind an address to the IP host server. This will be done when reclaiming an IP address that was allocated to a customer but is being returned. In a

similar fashion, the technical staff can request that the system unbind an address from the IP host server. This will be done when allocating an IP address to a customer.

As Figure 5 shows, administrators have their own unique requests to make to the system. Generally, it would be an administrative task to enter new addresses into the system in the event that the sponsor received a new address block allocation from the upstream provider. This is indicated in Figure 5 with the administrators communicating IP address blocks to the system. Also, the system is designed in such a fashion that it would be very simple to add a function to the program to add additional users to the system. It would be equally simple to add a function that allows an administrator to remove an IP address from the normal scan. This might be done if the administrator knew that the IP address would be assigned to a 'non-responder' for a time, or to remove it from normal scanning procedures for some other reason.

From the context diagram, it is apparent that there are three central functions to the IP management system: IP address binding; IP address unbinding; and IP address querying. The design of each of these functions will be clarified in the following sections.

3.2 Binding IP Addresses

The sponsor has found that the addresses most susceptible to 'theft' are those in its address space, but not bound to any interface. To reduce the window of opportunity for theft, then, includes minimizing the amount of time an address is unbound. This observation lead to two design decisions. The first is that all addresses not allocated to a customer should be bound to the IP host server. The second design decision is that addresses allocated to customers need to be tracked as 'unbound' until they respond to an ARP request. Thus, the process of reclaiming an IP address from a customer involves binding it to the IP host server's interface and adjusting the system records to indicate its new status as an unallocated address.

Thus, the process of binding an IP address to the IP host server is invoked when an IP address needs to be reclaimed from a customer. This situation normally comes about when a server is cancelled and all its addresses are returned to the pool of addresses available for assignment to a customer. Occasionally a situation occurs when a customer no longer has use for an address that was previously allocated to him.

The process of binding an IP address through the IP address management system is straightforward. It is depicted in the data flow diagram below.

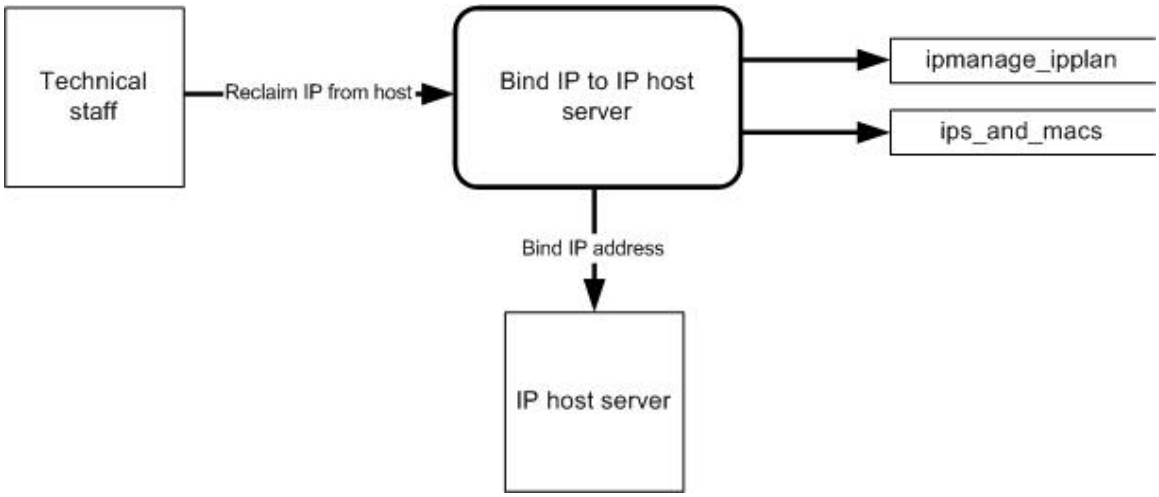

Figure 6. IP address binding data flow diagram.

Briefly, a member of the technical staff issues the command to the system to bind an IP address. The system checks that IP address is not already bound to the IP host server; if it is, the system informs the staff member and exits. Otherwise, the system looks up the address record in the ipmanage_ipplan database's `ipaddr` table. If the record exists, the staff is asked whether to really reclaim the IP address from the server in the record. If the staff member indicates to do so, the system performs the following steps:

- deletes the record from ipmanage_ipplan's `ipaddr` table;
- updates the address's record in ipman_db's `ips_and_macs` table, setting the `scan` field to true and the `mac_addr` field to the MAC address of the IP host server, and finally deleting any record for the address that might be in `unbound_ips` (in the unlikely event that one would exist);
- adds the IP address to a text file on the IP host server (/etc/ips) which lists all the IP addresses bound to the server;
- makes system calls on the IP host server to shut down and then bring up the network interface, reloading the IP addresses listed in /etc/ips.

The system call that reloads the IP address in /etc/ips is to execute a script commonly used by the sponsor to simplify IP aliasing; it is not written by the author.

3.3 Unbinding IP Addresses

IP address unbinding occurs in the opposite situation; namely, a customer requests an additional IP address. This entails the process illustrated in the following figure and described below.

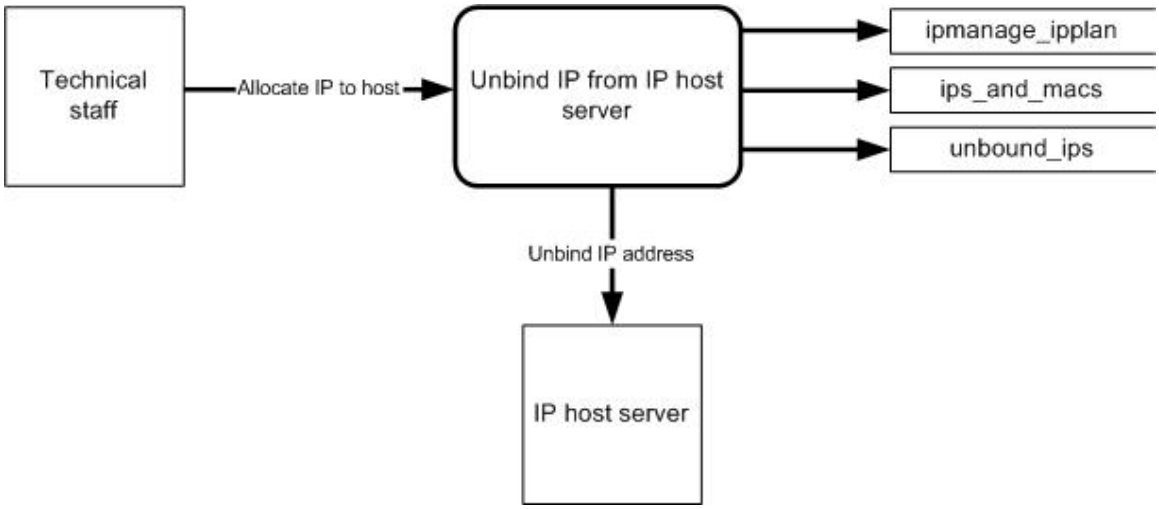

Figure 7. IP address unbinding data flow diagram.

The flow of data in this process is similar to the flow of data in the binding process, with one difference—the flow to the `unbound_ips` table. A member of the technical staff issues the command to the system to unbind an IP address from the IP host server. The system first checks to see if the address is already unbound; if it is, the staff member is notified and the program exits. Otherwise, the staff member is prompted for information about the server to which the address is being assigned. This information includes the server label, the datacenter and cabinet where the server is racked, and the type of server (dedicated, co-located, etc.). The input is echoed to the staff member for confirmation. If it is confirmed, the system performs the following actions:

- adds a record for the address to ipmanage_ipplan's `ipaddr` table;
- updates the address's record in ipman_db's `ips_and_macs` table by setting the `scan` field to false;
- adds a record to the `unbound_ips` table for the address, noting the staff ID for the staff member issuing the unbind command and the date and time the command was issued;
- deletes the IP address from the /etc/ips text file;
- makes system calls on the IP host server to shut down and then bring up the network interface, reloading the IP addresses still listed in /etc/ips.

At the end of the process, the table `ips_and_macs` has the IP address's record with with the IP host server's MAC address in the `mac_addr` field, the `scan` field set to false, and a record for the address in the table `unbound_ips`. The design decision to minimize the time that an IP address is unbound necessitated the inclusion of the table `unbound_ips`. We will see in the next section the reasoning behind the table and the role it plays in the IP querying process.

3.4 Querying IP Addresses

The process of going through all the IP addresses in one of the sponsor's broadcast domains, sending ARP requests to each to determine the MAC address on the interface to which the addresses are bound, analyzing the results, and reporting key events is the heart of the IP address management system. This is technically the most challenging part of the project, and its design the most involved. An overview of the process is illustrated in its top-level data flow diagram below.

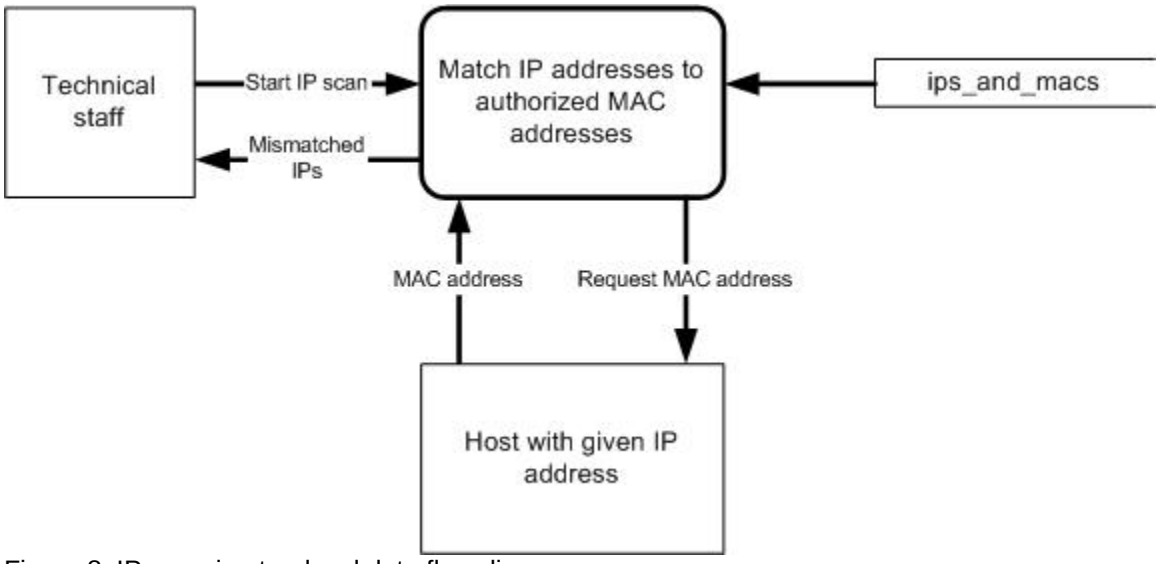

Figure 8. IP querying top-level data flow diagram.

The technical staff issues the command to start the scan. This could either be directly from the command line, or (as will more frequently be the case), indirectly by scheduling scans to run regularly as cron jobs.

The command invokes the program ipman, which starts by reading all the IP addresses from the ipman_db table `ips_and_macs` where the `scan` field is set to true. Ipman will then go through the list one address at a time. For each address, ipman will send ARP request packets asking for the IP address of the holder of the address being queried. It then captures packets, filtering specifically for ARP packets where the source host is the IP address being queried. When packets are received, it

makes sure that all of the responses match each as well as the `mac_addr` field in ips_and_macs. If they do not match each other as well as the `mac_addr` field, the IP address is reported as stolen at the end of the scan. However, the description of the top-level data flow diagram actually hides a great deal of the process involved in IP address querying. The scan actually consists of three passes. The first pass is illustrated in the second-level data flow diagram. It proceeds as described in the previous

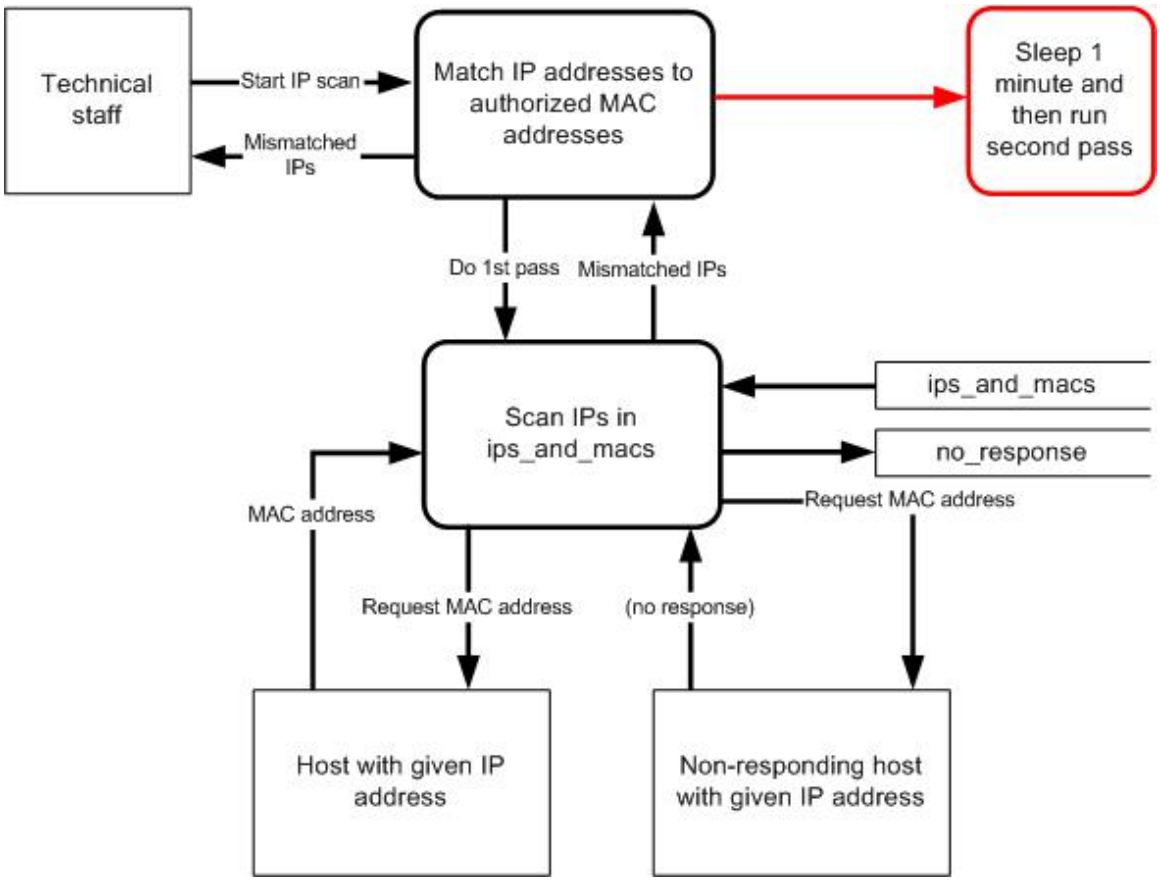

Figure 9. IP address querying second-level data flow diagram—first pass.

paragraph, with the exception that addresses that do not elicit a response have a record added to the table `no_response` which includes the address, the pass, and a timestamp. At the end of the first pass, ipman optionally sleeps for a short period of time, and then begins the second pass. The second pass is illustrated in the second-level data flow diagram in Figure 10.

The major difference between the first pass and the second and third passes is that while the first pass queries addresses in `ips_and_macs`, the second and third passes query only addresses that are in `no_response`. The program gets a list of all the addresses in the table and queries them as in the first pass. If a response is elicited on this pass, the record for

the address is deleted from `no_response` and the reply is analyzed as in the first pass.

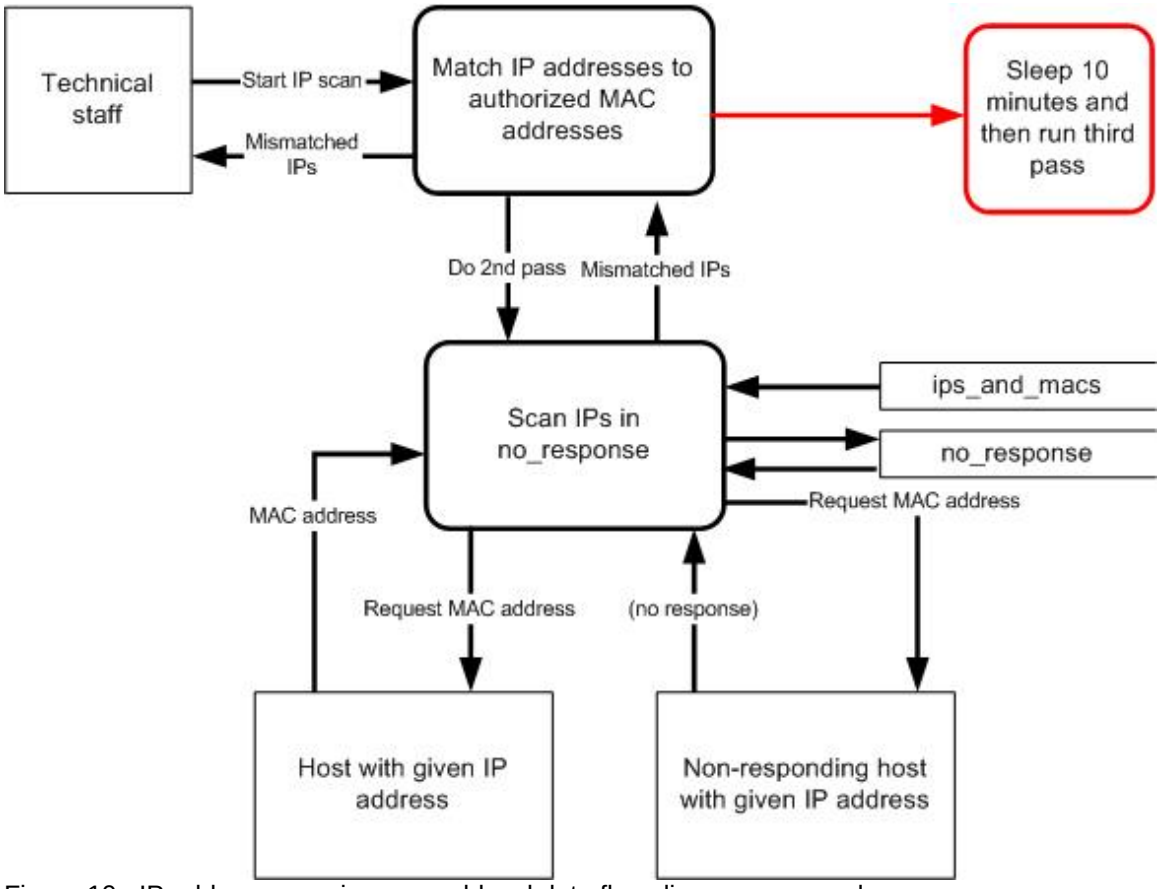

Figure 10. IP address querying second-level data flow diagram—second pass.

The third pass is virtually identical to the second pass; in the interests of brevity, the second-level data flow diagram illustrating it will not be included here.

With the three passes completed, the scan process finishes up with some recordkeeping activities. The remaining IP addresses in `no_response` are noted in the report log and then the records are deleted. The IP addresses in `unbound_ips` are scanned in a fashion similar to the first pass above. If a response is received, the record in `ips_and_macs` for the address is updated—the `mac_addr` field is updated with the MAC address received in the response, and `scan` is set to true—the record is deleted from `unbound_ips`, and the address is written to the report as getting a first response since being allocated to the customer.

3.5 Conclusion

This chapter covered the basic design of the proposed solution. It briefly explained the reasoning for the design's three central functions, and provided an overview of the processes involved in those functions. The next chapter will show how these processes are implemented as the construction of the system is discussed.

CHAPTER 4

Construction

4.1 Overview

This chapter discusses the actual implementation of the project. It begins with a very brief discussion of the resources to build the project and the factors upon which it relies, and then turns to some of the more salient points of the implementation. Emphasis is placed on those portions of the solution that are central to its functioning—such as the actual code for the functions involved in IP address querying, binding, and unbinding.

4.2 Resources

The project construction work was carried out on two servers. The first of these was a low-end PC on the author's home network. This machine is running Red Hat Linux 9 with the 2.4.20-37.9 kernel for an operating system. The code was written using vi (VIM 6.1.320), and compiled using the GNU C Compiler (gcc 3.2.2). The ipman_db database was created on this server, using MySQL server version 4.0.20-standard.

The basic structure of the program was created on this server—most of the functions that could be tested in any environment were created here. The `main` method was created on this machine, along with almost all other functions. Very few functions were initially written on the testing server, although a few were as it became clear during the testing phase that more work was needed.

Development continued in this environment until it was necessary to move the project to an environment more suitable for testing. The testing server also belongs to the author, but is racked at the sponsor's datacenter. This server is running CentOS 3.5 with the 2.4.21-32.0.1.EL kernel for an operating system. Code written on this machine also employed vi (VIM 6.3.81), and was compiled using gcc version 3.2.3. The databases were moved to MySQL server version 4.0.25-standard.

4.3 IP Address Querying

The basics of querying an IP address involve building an ARP request packet, putting it on the wire, and then capturing responses (if any) and storing the response's source protocol address for comparison with the information stored in the database.

The entire process of build and sending ARP request packets, capturing replies, and storing the results is performed in the ipman function `get_mac`. This function is defined as follows:

```
void get_mac( char *ip_addr )
```

`get_mac` actually stores the retrieved MAC address in an array of character strings, since multiple functions need to have access to the extracted information. Thus, there is no return type. `get_mac` takes one argument, which is the IP address for which the MAC address is being requested.

The actual process of building the ARP request packet consists of, first, declaring a pointer to type libnet_t. libent_t is a typedef of the libnet_context structure. This structure is libnet's "main monolithic control data structure that describes a complete libnet packet shaping/injection session" (Schiffman, 2003, p. 40).

In `get_mac`, the program declares the pointer to libnet_t as context, and a few other variables, the most important of which are included below:

```
libnet_t *context;

//The following holds the integer version of the IP address
u_int32_t temp_ip = 0;
struct libnet_ether_addr *ptr_hwaddr;
char src_ip[ IP_ALEN ], dest_ip[ IP_ALEN ];

// We don't know the destination MAC address, so need to broadcast.
char dest_mac[ ETH_ALEN ]={ 0xff, 0xff, 0xff, 0xff, 0xff, 0xff };
char src_mac[ ETH_ALEN ];

// Protocol tag
static libnet_ptag_t arp = 0, eth = 0;
```

With the variables defined, the program initializes some values. First, it will create the libnet session by calling `libnet_init` with parameters that specify the injection type (it uses the constant LIBNET_LINK to specify link-layer injection), the device over which the packets will be sent, and a buffer to which libnet can write errors if necessary.

```
//build the ARP request packet

// create the libnet session handle
if((context=libnet_init(LIBNET_LINK,"eth0",libnet_error_buf))==NULL) {
      printf( "libnet_init(): %s", libnet_error_buf );
}
```

Next, it assigns values to the source and destination IP variables. Note that once the libnet_t structure has been initialized with `libnet_init`, a

pointer to it is passed to all libnet function calls. Even when not strictly required otherwise, the functions need to have the libnet_t pointer passed to them because any error messages will be stored within this structure. Below, the program gets the integer values of the IP addresses' dotted decimal formats and stores them in the src_ip and dest_ip character arrays.

```
// pack the source and destination IPs—the source IP address is
// hard-coded, which needs to be changed
temp_ip = libnet_name2addr4( context, "147.202.49.28",
      LIBNET_DONT_RESOLVE );
memcpy( src_ip, ( char * ) &temp_ip, IP_ALEN );
temp_ip = libnet_name2addr4( context, ip_addr, LIBNET_DONT_RESOLVE );
memcpy( dest_ip, ( char * ) &temp_ip, IP_ALEN );
```

Now that all necessary variables are initialized, it is time to build the packet. The program will start at the top of the OSI stack and build the ARP packet first.

```
//build the ARP request packet
arp = libnet_build_arp(
      ARPHRD_ETHER,   // Hardware address type
      ETHERTYPE_IP,   // Protocol address type
      ETH_ALEN,       // Hardware address length
      IP_ALEN,        // Protocol address length
      ARPOP_REQUEST,  // Operation code
      src_mac,             // Sender hardware address
      (u_int8_t *)src_ip,  // Sender protocol address
      dest_mac,            // Target hardware address
      (u_int8_t *)dest_ip, // Target protocol address
      NULL,                // Payload --  none in this case
      0,                   // Size of the payload -- size of nothing
      context,
      0);       // This last argument explained below
```

Compare this to the ARP packet depicted in Figure 3 on Page 20. The program is filling in the ARP packet's fields, in order, with constants defined in libnet and values that were assigned earlier. The last argument is of type libnet_ptag_t. This is a protocol tag identifier. All libnet packet building functions return this data type, and all accept it as an argument. If one wanted change something slightly in this header (or in the data, if there were any), one could pass `arp` (declared as `static libnet_ptag_t` so it will be available next time around, if need be) as the final argument after modifying whatever needed modification. Since the modification of packet parameters is unnecessary, 0 is passed for this last argument.

With the ARP packet assembled, the program now builds the Ethernet frame.

```
eth = libnet_build_ethernet(
      dest_mac,
      src_mac,
      ETHERTYPE_ARP,
      NULL,
      0,
      context,
      0);
```

Again, the correspondence to the depiction of an Ethernet frame in Figure 2, Page 20 is almost one-to-one. The preamble is taken care of by the libnet library, as is the CRC at the end of the packet. The data, here the NULL argument, is actually the ARP packet that was just built above. The last three arguments are analogous to those in constructing the ARP packet.

All is set to write the packet to the wire, but first it is necessary to set things up to receive the replies. Here the program will employ the libpcap library, designed to make packet capture as transparent to the application programmer as libnet makes packet creation.

4.3.1 pcap_open_live

The following code has the error checking code removed to ease readability. The program first opens a pcap capture session as was done with libnet packet creation. `pkt_descriptor` is analogous to the `libnet_t` structure; it is the structure that contains all the necessary information for a packet-capture session. The variable `filter` allows the program to set a filter on the packets that are actually worth capturing. In the present case, they need to be ARP packets, and the source host must be from the IP address in `ip_addr`. `pcap_compile` processes the filter and prepares it for employment. Finally, `pcap_setfilter` sets the filter on the session.

```
pkt_descriptor = pcap_open_live(interface, BUFSIZ, 0, 500, error_buf);
sprintf( filter, "arp src host %s", ip_addr );
pcap_compile( pkt_descriptor, &prog_buff, filter, 1, 0 );
pcap_setfilter( pkt_descriptor, &prog_buff );
```

With everything set for both packet sending and capture, the program is almost ready to enter the loop that will send the ARP request five times and wait for the response. First, it is necessary to deal with a problem Linux has with `pcap_open_live` implementation.

4.3.2 Problem with pcap_open_live on Linux

At this point, the project hit a major problem. Taking another look at the `pcap_open_live` call, it is apparent that the fourth argument (passed here

as the 'magic number' 500), represents the read timeout. On different systems, this has different meanings. The intent is that after waiting that number of milliseconds for a packet, a read timeout occurs and the function returns. The read timeout functions this way on some systems. On other systems, the 'read timeout' clock does not start ticking until the first packet is captured. On Linux systems such as the development and testing servers, the read timeout is not implemented at all. The end result is that if no packets are captured that meet the packet filter constraints, the process will block indefinitely.

The only apparent way around this limitation in Linux was to ensure that there was an 'overseeing' process that prevented the process from blocking indefinitely. The two methods that are, perhaps, the most obvious are the implementation of threads, or the forking of child processes. The decision was made to fork child processes.

4.3.3 Implementation of forking

The idea of forking the child is simple in concept. The child process would be forked by the parent and begin the packet capture, while the parent went to sleep for 100 milliseconds. Captured packets would be disassembled, the MAC address extracted and written to the global array of character strings, and then the child would exit. If the child had not filled the array by the time the parent had awoken, the assumption would be that the child process is blocking indefinitely as it waits to capture replies from an unbound IP address. Thus, if the array had not been filled when the parent awoke, it would kill the child process.

The problem with this design should be apparent. Upon forking, the child is an exact replica of the parent process, with all the same variables, environment, and so forth. The problem is that these variables are copies of the parent's; the child has its own memory space (Robbins, 2004, p. 286). Thus, were the child to write the MAC address from captured packets to the global array declared for that purpose, it would be to *its own copy* of the array; the parent would no longer have any access to it.

In order to implement the design put forth, it was necessary to implement some form of interprocess communication (IPC). Of the five major methods of IPC (shared memory, mapped memory, pipes, FIFOs, and sockets), the decision was made to implement shared memory—it is relatively simple and allows for very fast communication between local processes (Mitchell et al., 2001, p. 96).

The implementation is as simple as promised. Once process allocates a shared memory segment. Each process that wishes to use it must attach

it. When finished, the processes detach the shared segment. Finally, one process must deallocate it. It seems clear in the present program that the parent process will allocate the shared memory segment, the child will attach it and write to it, and the parent will then deallocate it.

First, the program needs to declare some extra variables for the fork and shared memory implementation.

```
pid_t child_pid;
char *shared_mem; // This will be shared for getting the MAC into
                  // target_mac[]
int shmseg_id;
```

Next, it allocates the shared memory.

```
shmseg_id = shmget( IPC_PRIVATE, MAX_SIZE_ETHADDR,
             IPC_CREAT | IPC_EXCL | S_IRUSR | S_IWUSR );
```

The first parameter is an integer key indicating which segment to create. Use of the constant IPC_PRIVATE guarantees that a new segment will be created which no other process has specified (Mitchell, p. 98). The second argument is the amount of space to allocate. The third argument is a bitwise OR of flags specifying more options to `shmget`. Here it is specified that a new segment be created; that failure to allocate a unique segment means that `shmget` fails; the last two flags specify read and write permissions for the owner of the segment.

Now, with an overseeing process to make sure that no child blocks indefinitely and a way for the child to communicate captured MAC addresses to the parent through IPC, the program is ready to enter the loop that will send the ARP request five times and wait for the response.

4.3.4 Writing the packet and capturing the reply

With all prepared, the program entered a loop that repeats five times. In each iteration of the loop, the prepared packet is written to the wire, and the child is forked and listens for the reply. Any captured packet that makes it through the filter is passed to the pcap callback function, `pcap_callback_fct`. That function will be discussed shortly; briefly, it extracts from the captured packet the source MAC address and stores it in the global character string variable `temp_mac`.

```
for( x = 0; x < 5; x++ ) {
      t = libnet_write( context );
      if( t == -1 ) {
            printf( "libnet_write error: %s\n", libnet_geterror(
            context ));
      }
```

```
      child_pid = fork();
      if( child_pid == 0 ) {  // this is the child process
            // attach to shared memory
            shared_mem = ( char * ) shmat( shmseg_id, 0, 0 );
            pcap_dispatch( pkt_descriptor, 1,
                  ( void * )pcap_callback_fct, NULL );
            //convert to upper case
            to_upper( temp_mac );
            memcpy( shared_mem, temp_mac, MAX_SIZE_ETHADDR );
            exit( 0 );
      }
```

The call to `shmat` passes the segment ID to the function for attaching the shared memory. `pcap_dispatch` begins the packet capture, and specifies that the capture will stop once a packet meeting the filter requirements is captured. At this point, the child process has either performed its task and exited, or has blocked. If it has blocked, nothing will have been written to the shared memory, so the program can check the string length there.

Now the code listing continues with the parent:

```
      else {
            sleep( .1 );
            shared_mem = ( char * ) shmat( shmseg_id, 0, 0 );
            if( strlen( shared_mem ) == 0 ) {
                  // the child has blocked, kill it
                  kill( child_pid, SIGTERM );
            }
      }
      memcpy( target_mac[ x ], shared_mem, MAX_SIZE_ETHADDR );
}//  end of for( x = 0; x < 5; x++ )
```

Here ends the loop that sends the five ARP requests and captures the reply, if any. It is necessary to briefly examine the pcap callback function to see exactly what transpires with the captured packet. The function is defined as:

```
void pcap_callback_fct( u_char *what, struct pcap_pkthdr *pkt_header,
      u_char *packet )
```

The program first defines a structure for the Ethernet header, the ARP header, and the source MAC address.

```
struct ether_header *eth_header;  //  net/ethernet.h
struct ether_arp *arp_header;   // linux/if_arp.h
u_char src_ha[ 19 ]; //source hardware address
```

Next, it extracts the Ethernet header from the packet that was passed in to the function.

```
eth_header = ( struct ether_header * ) packet;
```

It then checks to see if it is an ARP packet. If it is not an ARP packet, the program really should have gotten here, as the filter set in the calling function specified only ARP packets.

```
// If it's an ARP packet, get some ARP info from it:
if( ntohs( eth_header->ether_type ) == ETHERTYPE_ARP ) {
```

If it is an ARP packet, the program next extracts the ARP header, and then examines the operation code field. If the op code indicates an ARP reply, it extracts the hardware (MAC) address from the packet, copies it into the global character string `temp_mac`, and returns.

```
      arp_header=(struct ether_arp *)
            (packet+sizeof(struct ether_header));
      if( arp_header->ea_hdr.ar_op == ntohs( ARPOP_REPLY ) ) {
            snprintf( src_ha, sizeof( src_ha ) - 1,
                  "%02x:%02x:%02x:%02x:%02x:%02x:",
                  arp_header->arp_sha[ 0 ], arp_header->arp_sha[ 1 ],
                  arp_header->arp_sha[ 2 ], arp_header->arp_sha[ 3 ],
                  arp_header->arp_sha[ 4 ], arp_header->arp_sha[ 5 ] );

                  memcpy( temp_mac, src_ha, 19 );
      }
}
```

Thus, the pcap callback function writes the source MAC address of a single packet to `temp_mac`. The child, before exiting, writes the contents of `temp_mac` to the shared memory segment for the parent to access.

4.4 IP Address Binding and Unbinding

The binding and unbinding of IP address is much simpler and requires little explanation. When the binding function is invoked by a user, the program prompts the user for the dotted-decimal notation of the address and its subnet mask. The input is passed to the function `bindips`, defined in the following manner:

```
int bindips( const char *addr, char *mask )
```

`bindips` first validates that the input is a valid IP address, a valid range of IP addresses, or a valid C-class. If the input is a single IP address, it checks to make sure that the IP address is not already bound with a call to `ip_is_bound`. If it is not, the function calls `query_ipmanage` to delete the IP address record. Next, it updates the `ips_and_macs` table in the ipman_db database with calls to `add_iphostserver_mac` and `set_scannable`. The function then adds the address to the list of locally bound address, /etc/ips. A call to `delete_unboundip` deletes any potential record for the IP

address in the table `unbound_ips`. A similar process is followed if the function finds that the user input is a range of IP addresses.

Should the input represent a valid C-class address block, `bindips` calls `add_new_addr_block`, which adds a new base index for the block to the table `base` in the database ipmanage_ipplan adds the addresses to the ipman_db table `ips_and_macs` with the IP host server's MAC address in `mac_addr` and `scan` set to true for each record, and then adds the addresses to /etc/ips.

Finally, the function calls `reload_ipaliases`, which simply makes a few simple system calls to dump the current aliased IP addresses and reload them all, including the recent addition to /etc/ips.

Unbinding IP addresses works in much the same way. The function responsible is `unbindips`. A major difference is that `unbindips` accepts only a single IP address for unbinding. The removal of a C-class allocation has never happened with the sponsor. Unbinding a range of IP addresses has happened, but is quite rare and thus was not implemented.

4.5 Conclusion

This chapter has shown the specifics of implementing the design covered in the previous chapter. The construction of the key functions of the solution was covered, and some of the more esoteric pieces involved in getting the design to function properly as a whole were discussed. The next chapter will evaluate the solution's strengths and weaknesses.

CHAPTER 5

Findings

5.1 Overview

This chapter discusses the actual implementation of the project. It begins with a brief discussion of the functionality ultimately offered by the solution, and the extent to which the users considered it an improvement over the previous system. It concludes with a discussion of the solution's performance in the sponsor's environment.

5.2 Functionality

5.2.1 IP address unbinding

As may be recalled from the initial discussion of the environment in Chapter 1, the technical staff members went through a series of steps to allocate an IP address to a customer. First, a web interface into ipmanage_ipplan was consulted to determine addresses allegedly free for customer allocation. Next, an internal server was used to arping the address to ensure that, regardless of what the records indicated, it was in fact free. Finally, the web interface was used again—clicking the IP address brought up an HTML form in which the needed information was supplied.

The implementation of the solution has not done anything to simplify this process for the members of the technical staff. It is still necessary for them to consult the web interface to determine which IP addresses are allegedly free. At that point, they log in to the IP host server to arping the free address. They then execute the command 'ipman –u' and are prompted for the IP address to allocate and the same information as previously entered in the HTML form. The technicians were unanimous in agreeing that the new solution did not make allocating addresses to customers easier.

In fact, in some situations allocation to customers was made more difficult. Specifically, when a customer requests multiple addresses, those addresses are frequently allocated from the same Web page, by control-clicking from a list box of the subnet's addresses. In this way, multiple addresses can be assigned by submitting a single form. There is no such functionality in ipman. Each address allocated involves a separate run of the program and the re-entry of the required information.

5.2.2 IP address binding

Unbinding IP addresses normally only happens when a server is cancelled. When this happens, the addresses must be reclaimed. This was previously done by means of the Web interface; when an IP address is clicked, there is a button to delete record.

Again, the verdict was unanimous that the solution did not make IP deallocation any simpler. There was no complaint, however, that deallocation had been made any more difficult, either. Repetitive clicking to delete IP address records was not considered any more difficult than repetitive runs of ipman.

5.2.3 IP querying

The tests of IP address querying went very well. The staff members felt that the scan was simple to run and produced the needed results in a timely and efficient fashion. The only suggestion as far as the interface and functionality was concerned was that, since the state of the system was not being changed by the scan, there was no real reason to have the function protected by a login. Additionally, removal of the login requirement from the scanning function would facilitate scheduling the scan as a cron job.

5.2.4 Logging

The logging aspect of the program was very well received. In the current environment, it is very difficult for the technical staff to recover from any error made in adding or deleting address records. Records in ipmanage_ipplan have a timestamp indicating when they were last modified; however, very few staff members have both the access and the knowledge to extract the needed data to undo the mistake in a timely fashion. Further, deleted records, of course, have no timestamp at all. If a record is accidentally deleted, there is no way to recover the proper state of the system easily. The logging facilities of ipman record the time, date, user, address, and server label involved in every instance of IP address binding or unbinding. This makes recovering from a mistake much simpler than in the previous environment.

5.3 Performance

The solution performed solidly in most aspects. It is robust, with ample error checking and error logging in the event things went poorly. Once it was fully debugged and implemented, there were no problems with the program crashing.

The database accesses performed well, and there were no issues with errors or incomplete transactions. Logging also exhibited good performance. The log messages are concise, so do not threaten to overwhelm disk space or lead to the creation of overly large files.

The only performance issue involved IP address querying. Recall from chapter 4 that the parent sleeps while each of the child processes are given an opportunity to capture a response packet. In the tested version of the program, the parent process sleeps for one-tenth of a second. Since each address has five ARP requests sent to it, each address takes a half of a second for each pass.

With 38 subnets and approximately 253 addresses per subnet to scan, the total number of IP addresses on the first pass is 9614. This means that the first pass would take approximately 80 minutes.

The sleep value for the parent process was arbitrarily chosen. One hundred milliseconds seemed ample time to receive a reply packet, and in the testing environment the entire scan completed in an acceptably short amount of time. In the live environment, 80 minutes is too long. However, there is no reason that the parent sleep time cannot be shortened. Given that the environment is a 100 Mbps Ethernet network, replies to an ARP packet can be expected in 10 msec or less. Thus, the sleep time for the parent could easily be reduced to .01 seconds, bringing the total time for the first pass down to a little more than eight minutes.

5.4 Conclusion

This chapter demonstrated that, on balance, the solution as presented did not bring about the desired benefits as far as combining the 'technical' and 'recordkeeping' aspects of IP address management. The present solution did little to make IP address deallocation easier, and actually made address allocation more difficult in certain situations. However, the logging features added by the solution were very beneficial. Moreover, the solution performed well, although minor, easily-made adjustments to the program parameters could improve the IP address query run time considerably. The next chapter will briefly discusses potential avenues for further development of this solution, as well as other areas of study.

CHAPTER 6

Conclusions and Recommendations for Further Work

Overall, the finding has been that the desired improvements to IP address allocation and deallocation have not been realized. The IP querying function, however, has performed as expected, and with minor modifications is ready for the sponsor's live environment. However, these failures to meet desired outcomes can be easily remedied.

A few modifications to the IP address allocation and deallocation process could decidedly turn things around and make this process much easier. These modifications did not manifest in the solution as tested due to design flaws (requested functionality that was simply not implemented in the design) and the lack of request for certain functionality.

As an example of the latter, consider the IP deallocation process. The ipman program already has all the essential abilities required to implement a function that will reclaim all IP addresses from a given server. A function could very easily be added that would prompt the technician for the server label. It could then perform a lookup of the server label in the ipmanage_ipplan database `ipaddr` table, find all IP address records tied to that label, extract the addresses to an array in memory, delete the records in the `ipaddr` table, and bind the addresses in the array to the IP host server's adapter. Thus, a single command and entry of the server's label can perform all necessary actions to reclaim an IP address from a server.

The solution could also easily make IP address allocation much simpler. Without a great deal of extra work, functions could be added to make address allocation simpler as well. There is no need for the technician to search through a Web interface for allegedly free IP addresses, and then arping those addresses to find verify that information. By simply entering the server label and the number of addresses to allocate, ipman could perform database lookups to find potentially free addresses, use its `get_mac` function to verify that they are free, and perform all the necessary functions to update tables and unbind the addresses from the IP host server.

While the implementation of arguments to the program to perform a single function was found to be efficient, the necessity to immediately log in again if more than one function was being performed was found to be onerous. It would be worthwhile to implement a menu system if the program is invoked with no arguments. With such a menu system, the

user can log in a single time, and continue to run functions without the need for authenticating each time.

Essentially, a potentially powerful system has been barely uncovered in this project. By adding more database functionality, options, and some performance enhancements, it would be possible to perform all IP address management from this single solution. While the solution does not solve all problems associated with IP address management, it takes some serious first steps; the investigation of the next steps is worthy of consideration.

References Cited

Barnes, D. and Sakandar, B. (2005) Cicso LAN Switching Fundamentals. Indianapolis: Cisco Press.

Deitel, H. and Deitel, P. (2003) C++ How to Program. 4th ed. Upper Saddle River, NJ: Prentice Hall.

Kernighan, B. W., and Ritchie, D. M. (1988) The C Programming Langauage. 2nd ed. Upper Saddle River, NJ: Prentice Hall PTR.

Kurose, J. and Ross, K. (2003) Computer Networking: A Top-Down Approach Featuring the Internet. 2nd ed. Boston: Addison Wesley.

Mitchell, M., Oldham, J. and Samuel, A. (2001) Advanced Linux Programming. Indianapolis: New Riders Publishing.

MySQL AB. (2004) MySQL Reference Manual. [Internet] Madison: MySQL AB. Available from: <http://mirror.services.wisc.edu/mysql/Downloads/Manual/manual.pdf> [Accessed 28 September 2005]

Plummer, David C. (1982) An Ethernet Address Resolution Protocol. [Internet] Rockville: Internet Engineering Task Force. Available from: <http://www.ietf.org/rfc/rfc0826.txt?number=826> [Accessed 28 September 2004]

Robbins, A. (2004) Linux Programming by Example. Upper Saddle River, NJ: Prentice Hall PTR.

Schiffman, M. (2003) Building Open Source Network Security Tools: Components and Techniques. Indianapolis: Wiley Publishing.

Syngress Media, Inc. (1998) CCNA Cisco Certified Network Associate Study Guide. Berkely: Osborne/McGraw-Hill.

Appendix A

SOURCE CODE FOR THE IPMAN PROGRAM

```
/*  Serial 2005092003 */
/*  Source code available for download at
http://zaichik.org/msc/ipman.c
 */

#define _GNU_SOURCE
#include <arpa/inet.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
#include <malloc.h>
#include <unistd.h>
#include <getopt.h>
#include <openssl/md5.h>
#include <pcap.h>
#include <mysql/mysql.h>
#include <netinet/in.h>
#include <netinet/if_ether.h>
#include <sys/shm.h>
#include <sys/socket.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <ctype.h>
#include <libnet.h>

/*** Global variables
 */

char *target_mac[ 5 ];   // This has to be global, since
                         // pcap_callback_fct can't be sent params and
                         // has no return
char *temp_mac;         // Ditto
char *myName;  /* The name under which the program is running. Used in
                  most functions */
char *username;  /* The username from login function; used in many
                  functions for logging */


enum e_query { INSERT, DELETE };
typedef enum e_query query;

enum e_boolean { FALSE, TRUE };
typedef enum e_boolean boolean;

/*** Functions
 */

int add_iphostserver_mac( char *ip ); /* updates
                  ipman_db.ips_and_macs.mac_addr with the IP host
                  server's MAC address
                   on a binding operation, called from bindips(). */

int add_new_addr_block( int octet1, int octet2, int octet3 ); /* adds a
                             new C-class to the local IPs database */
```

```
int add_new_base( int oct1, int oct2, int oct3 ); /* adds a new base
                        record for a new C-class to ipmanage_ipplan */

void add_report( char *msg );  /*  Adds item to
                               /var/log/ipman/ipman.report  */

int bindips( const char *ip, char *netmask );

void delete_unboundip( char *ip );

unsigned int dotted_to_int( char *addr );  /* returns the integer value
                               of a IPv4 dotted decimal notation */

int get_base( const char *ip );/* returns the baseindex from
                               ipmanage_ipplan.base for inserting a
                               record into ipmanage_ipplan.ipaddr.
                        Returns -1 on error, baseindex otherwise */
void get_mac( char *ip_addr ); /* arpings the IP address, puts the MAC
                               address in target_mac[ ] or packs it full
                               of 0.0.0.0 if a non-responder */
int get_user_id( void ); /* gets ipman_db.staff_members.staff_id based
                               on login name */

char* int_to_dotted( unsigned int ipv4 );

int ip_is_bound( const char *addr ); /* Checks /etc/ips to see if IP
                                     already bound to IP host server */

int is_numeric( char *str ); /* Checks to see if a string is all
                                     numeric */

int logEntry( const char *message, const char *file ); /* Passed the
                        message and the log file, does the logging. */

int login( void ); /* compares username-password combo; returns 1 on
                        failure */

int open_report( void ); /* Creates the report file
                        /var/log/ipman/ipman.report and adds the date.
                        Returns -1 if unable to. */

int pass_one( void ); // Runs the scan on ipman_db.ips_and_macs

int pass_two_three( void ); // Runs the scan on ipman_db.no_response

void pcap_callback_fct( u_char *what, struct pcap_pkthdr *pkt_header,
      u_char *packet );

void print_usage( void );

int query_ipmanage( char *ip, query action );
      /* ip is the IP to add/delete, and int (enum queryType instead?)
            to indicate INSERT or DELETE
            Will ask for the server label in the event of INSERT,
            lookup label and verify action in
            event of DELETE.  Of course, bindips() entails DELETE
            (reclaiming an IP from
```

```
            a customer) and unbindip() entails INSERT (assigning them
            to a customer). */
int reload_ipaliases( void );  /* Calls WHM ipaliases reload, returns 0
                                  on failure */

int report_non_responders( void );

int report_old_unbound( void );

int report_unbound_response( void );

int scan( int report ); /* Scans IPs in local database, returns 0 on
                           any error */

int send_report( char *sendMsg ); /* Emails report digest */

int set_scannable( char *ip, boolean scan ); /* Updates
            ipman_db.ips_and_macs.scan to 0 on bind, 1 on unbind */

void to_upper( char *the_string ); /* changes a string to all
                                      uppercase, used for MACs */

int unbindip( const char *addr ); /* Unbinds a single IP address */

int update_unbound( char *ip, query queryType ); /* INSERT from
            unbindip(), DELETE from scan when response received */

int validate_addr( const char *addr, int *lower, int *upper ); /*
            Indicates if input is invalid, range, or single IP. If
            a range, the function sets lower and upper bounds */

#define MAX_SIZE_IPADDR 20 /* xxx.xxx.xxx.xxx-xxx\0 */
#define MAX_SIZE_NETMASK 16 /* xxx.xxx.xxx.xxx\0 */
#define MAX_SIZE_MSG 2048 /* Maximum characters in message sent to
                             logEntry() function. */

#ifndef ETH_ALEN
#define ETH_ALEN 6
#endif

#ifndef IP_ALEN
#define IP_ALEN 4
#endif
#define MAX_SIZE_IPADDR 20
#define MAX_SIZE_ETHADDR 20 // xx:xx:xx:xx:xx:xx\0

int main( int argc, char **argv) {

      char o; /* return from getopt_long() */
      char *ip;
      char *netmask;
      char *mac;
      int charsRead; /* for the bind operation */
      char *tempch;
      int success;
      int MAX_IPADDR_SIZE =  MAX_SIZE_IPADDR;
      int MAX_NETMASK_SIZE = MAX_SIZE_NETMASK;
```

```
      // Define the flags below.
      int bind = 0, checkip = 0, lookupMAC = 0, noalert = 0, runscan =
0, unbind = 0;
      struct option longopts[] = {
            { "bind",   no_argument,            NULL, 'b' },
            { "check",  required_argument,      NULL, 'c' },
            { "help",   no_argument,            NULL, 'h' },
            { "lookup", required_argument,      NULL, 'l' },
            { "noalert",      no_argument,            NULL, 'n' },
            { "scan",       no_argument,          NULL, 's' },
            { "unbind", no_argument,            NULL, 'u' },
            { 0, 0, 0, 0 }
      };

      myName = argv[ 0 ];

      if( argc == 1 ) {
            print_usage();
            return( 1 );
      }

      while(( o = getopt_long( argc, argv, "bc:hl:nsuW;", longopts,
NULL )) != -1 ) {

            switch( o ) {
                  case 'b': // bind
                        if( bind || checkip || lookupMAC || noalert ||
runscan || unbind ) {
                              print_usage();
                              exit( 0 );
                        }
                        bind = 1;
                        break;
                  case 'c':
                        if( bind || checkip || lookupMAC || noalert ||
runscan || unbind ) {
                                          print_usage();
                                          exit( 0 );
                                    }
                        checkip = 1;
                        ip = optarg;
                        break;
                  case 'h':
                        print_usage();
                        exit( 0 );
                        break;
                  case 'l':
                        if( bind || checkip || lookupMAC || noalert ||
runscan || unbind ) {
                                          print_usage();
                                          exit( 0 );
                                    }
                        lookupMAC = 1;
                        mac = optarg;
                        break;
                  case 'n':
```

```
                        if( bind || checkip || lookupMAC || noalert ||
unbind ) {
                                print_usage();
                                exit( 0 );
                        }
                        noalert = 1;
                        break;
                case 's':
                        if( bind || checkip || lookupMAC || runscan ||
unbind ) {
                                print_usage();
                                exit( 0 );
                        }
                        runscan = 1;
                        break;
                case 'u':
                        if( bind || checkip || lookupMAC || noalert ||
runscan || unbind ) {
                                print_usage();
                                exit( 0 );
                        }
                        unbind = 1;
                        break;
                case '?':
                default:
                        printf( "%s: option `-%c' is invalid.\n",
myName, optopt );
                        print_usage();
                        return( 1 );
            } /* switch */
      } /* while */

      /*****************************************************************
*********
         *  Once we are here, only one--possibly two--flags have been
set (two in
       *  the case of -n, which can double with -s).  Otherwise, we can
assume
       *  if a flag has been set, we run the associated action.  We do
need to
       *  check when we get to if( noalert ) that the flag for runscan
has been
       *  set, because the above will not catch ONLY -n being set.
First we need
       *  to get logged in, or quit with error.

*****************************************************************
**/

      if( !login() ) {
                return( 1 );
        }

      if( bind ) {
            ip = ( char * ) malloc ( MAX_SIZE_IPADDR + 1 );

            if( ip == NULL ) {
```

```
                  printf( "%s: Out of memory in main() ln 40.", argv[ 0
] );
                  return( -1 );
            }
            netmask = ( char * ) malloc( MAX_SIZE_NETMASK + 1 );
            if( netmask == NULL ) {
                  fprintf( stderr, "%s: Out of memory in main() ln
45.", argv[ 0 ] );
                  return( -1 );
            }

            printf( "Enter the IP address: " );
            charsRead = getline( &ip, &MAX_IPADDR_SIZE + 1, stdin );
            printf( "Enter the subnet mask: " );
            charsRead = getline( &netmask, &MAX_NETMASK_SIZE + 1, stdin
);
            bindips( ip, netmask );
            free( ip );
            free( netmask );
      }

// The arg below was assigned in the switch.
      if( checkip ) {
            printf( "IP = %s\n", ip );
      }

//  The arg below was assigned in the switch.
      if( lookupMAC ) {
            printf( "MAC = %s\n", mac );
      }

      if( noalert ) {
            if( ! runscan ) {
                  fprintf( stderr, "--noalert ( -n ) option only
allowed with the --scan ( -s ) option.\n" );
                  print_usage();
                  exit( 1 );
            }
      }

      if( runscan ) {
            int x = 0; // just a counter
            for( x = 0; x < 5; x++ ) {
                        target_mac[ x ]  = ( char * ) malloc(
MAX_SIZE_ETHADDR + 1 );
                  if( target_mac[ x ] == NULL ) {
                        printf( "%s:  Out of memory in main().\n",
myName );
                        exit( -1 );
                  }
            }
            scan( 1 ); // means report
            for( x = 0; x < 5; x++ ){
                  free( target_mac[ x ] );
            }
            return( 0 );
      }
```

```
      if( unbind ) {

            ip = ( char * ) malloc ( MAX_SIZE_IPADDR + 1 );

            if( ip == NULL ) {
                  fprintf( stderr, "%s: Out of memory in main() ln
96.\n", argv[ 0 ] );
                  return( -1 );
            }

            printf( "Enter the IP address: " );
            // as in bind(): charsRead = getline( &ip, &MAX_IPADDR_SIZE
+ 1, stdin );
            charsRead = getline( &ip, &MAX_IPADDR_SIZE + 1, stdin );
            if ((tempch = strchr( ip ,'\n')) != NULL) {
                  *tempch = '\0';
            }

            if( unbindip( ip ) )  {
                  fprintf( stderr, "%s: Error in unbindip().\n",
myName);
                  return( 1 );
            }
            free( ip );

      }
      return( 0 );

} /* main() */

int add_iphostserver_mac( char *ip ) {

      MYSQL conn;
        char *server = "localhost";
        char *mysqlUser = "user";
        char *mysqlPass = "pass";
        char *mysqlDB = "ipman_db";
        int MAX_SIZE_QUERY = strlen( "UPDATE ips_and_macs SET mac_addr
= '00:10:DC:37:D1:74' WHERE ip_addr = 1234567890" );
      char *mysqlQuery;
      unsigned int ipv4 = dotted_to_int( ip );
      int return_code = 0;

      mysqlQuery = ( char * ) malloc( MAX_SIZE_QUERY + 1 );
      if( mysqlQuery == NULL ) {
            printf( "%s: Out of memory in add_iphostserver_mac().\n",
myName );
            exit( -1 );
      }

      mysql_init( &conn );
      sprintf( mysqlQuery, "UPDATE ips_and_macs SET mac_addr =
'00:10:DC:37:D1:74' WHERE ip_addr = %u", ipv4 );
      mysql_real_connect( &conn, server, mysqlUser, mysqlPass, mysqlDB,
0, NULL, 0 );
      mysql_real_query( &conn, mysqlQuery, strlen( mysqlQuery ));
```

```
      if( mysql_errno( &conn )) {
            printf( "Error updating ipman_db.ips_and_macs.mac_addr for
IP address %s\n", ip );
            printf( "MySQL error is %s\n", mysql_error( &conn ));
            printf( "Contact an administrator.\n" );
            return_code = -1;
      }
      mysql_close( &conn );
      free( mysqlQuery );
      return( return_code );

} // add_iphostserver_mac

int add_new_addr_block( int octet1, int octet2, int octet3 ) {

      /* Takes first three octets of a class C address and adds to
local database.
         Returns 1 if all is good, 0 otherwise.
       */

      MYSQL conn;
      char *server = "localhost";
      char *mysqlUser = "user";
      char *mysqlPass = "pass";
      char *mysqlDB = "ipman_db";
      int MAX_SIZE_QUERY = strlen( "INSERT INTO ips_and_macs
VALUES('xxx.xxx.xxx.xxx', '255.255.255.0', '', x)" );
      int x = 0;  // just a counter
      char *mysqlQuery;
      char *logMsg;
      unsigned int baseaddr = 0;
      char *ipString;
      add_new_base( octet1, octet2, octet3 );

      ipString = ( char * ) malloc( MAX_SIZE_IPADDR );
        if( ipString == NULL ) {
                printf( "%s:  Out of memory in add_new_base().\n" );
                exit( -1 );
        }

        sprintf( ipString, "%d.%d.%d.0", octet1, octet2, octet3 );
        baseaddr = dotted_to_int( ipString ) + 2; // baseaddr starts at
integer equiv of x.x.x.2
      mysqlQuery = ( char * ) malloc ( MAX_SIZE_QUERY + 1 );
      if( mysqlQuery == NULL ) {
            fprintf( stderr, "%s: Out of memory in add_new_addr_block()
ln 181.\n", myName );
            exit( -1 );
      }

      logMsg = ( char * ) malloc( MAX_SIZE_MSG + 1 );
      if( logMsg == NULL ) {
            fprintf( stderr, "%s: Out of memory in add_new_addr_block()
ln 181.\n", myName );
            free( mysqlQuery );
      }
```

```
      mysql_init( &conn );
      mysql_real_connect( &conn, server, mysqlUser, mysqlPass, mysqlDB,
0, NULL, 0 );
      for( x = 2; x <= 254; x++ ) { // adding x.x.x.2 - x.x.x.254, 253
addresses

            sprintf( mysqlQuery, "INSERT INTO ips_and_macs VALUES(%u,
'255.255.255.0', '', 0)",
                  baseaddr );
            mysql_real_query( &conn, mysqlQuery, strlen( mysqlQuery )
);
            if( mysql_errno( &conn ) ) {
                  fprintf( stderr, "%s: Error on INSERT: %s.\n",
myName, mysql_error( &conn ) );

                  free( mysqlQuery );
                  if( x == 2 ) {
                        sprintf( logMsg, "IP address %d.%d.%d.2 added
to list of local IPs by user %s when an error occurred.  The IP may not
have been added successfully to the database.\n",
                              octet1, octet2, octet3, username );
                  }
                  else {
                        sprintf( logMsg, "IP addresses %d.%d.%d.2-%d
added to list of local IPs by user %s when an error occurred. The last
IP may not have been added successfully to the database.\n", octet1,
octet2, octet3, x );
                  }
                  printf( "%s: %s", myName, logMsg );
                  logEntry( logMsg, "/var/log/ipman/ipman.err" );
                  free( logMsg );
                  return( -1 );
            }
            baseaddr++;
      }

      free( mysqlQuery );
      free( logMsg );
      return( 0 );

} /* add_new_addr_block() */

int add_new_base( const int oct1, const int oct2, const int oct3 ) {

      MYSQL conn;
        char *server = "localhost";
        char *mysqlUser = "user";
        char *mysqlPass = "pass";
        char *mysqlDB = "ipmanage_ipplan";
        int MAX_SIZE_QUERY = strlen( "INSERT INTO base (baseaddr,
subnetsize, admingrp, customer, lastmod, userid, swipmod) VALUES
(1234567890, 256, 'NET-XXX-XXX-XXX-0', 'IPADMINs', 2, NOW(),
'XXXXXXXXXXXXXXXX', '0000-00-00 00:00:00')" );
        int x = 0;  // just a counter
        char *mysqlQuery;
        char *logMsg;
```

```
      // database variables
      char *ipString;
      unsigned int baseaddr;  // will be dotted_to_int(
"oct1.oct2.oct3.0" );
      char *descrip; // 'NET-OCT1-OCT2-OCT3-0'
      int subnetsize = 256;
      char *admingrp = "IPADMINs";
      int customer = 2;
      // lastmod will be NOW()
      // userid will be username
      char *swipmod = "0000-00-00 00:00:00";

      int returnValue; // for mysql_affected_rows() return value

      // assign field values
      ipString = ( char * ) malloc( MAX_SIZE_IPADDR );
      if( ipString == NULL ) {
            printf( "%s:  Out of memory in add_new_base().\n" );
            exit( -1 );
      }

      sprintf( ipString, "%d.%d.%d.0", oct1, oct2, oct3 );
      baseaddr = dotted_to_int( ipString );

      descrip = ( char * ) malloc( strlen( "NET-XXX-XXX-XXX-0" ) + 1 );
      if( descrip == NULL ) {
                printf( "%s:  Out of memory in add_new_base().\n" );
                exit( -1 );
        }
      sprintf( descrip, "NET-%d-%d-%d-0", oct1, oct2, oct3 );

      mysql_init( &conn );
      if( !mysql_real_connect( &conn, server, mysqlUser, mysqlPass,
mysqlDB, 0, NULL, 0 )) {
            printf( "%s\n", mysql_error( &conn ));
            free( ipString );
            free( descrip );
      }
      mysqlQuery = ( char * ) malloc( MAX_SIZE_QUERY );
      if( mysqlQuery == NULL ) {
            printf( "%s: Out of memory in add_new_base().\n" );
            free( ipString );
            free( descrip );
      }
      sprintf( mysqlQuery, "INSERT INTO base ( baseaddr, subnetsize,
descrip, admingrp, customer, lastmod, userid, swipmod ) VALUES (%u, %d,
'%s', '%s', %d, NOW(), '%s', '%s')", baseaddr, subnetsize, descrip,
admingrp, customer, username, swipmod );

      free( descrip ); // Still need ipString for logging.
      mysql_real_query( &conn, mysqlQuery, strlen( mysqlQuery ));

      returnValue = mysql_affected_rows( &conn );
      if( returnValue == 1 ) {
            logMsg = ( char * ) malloc( MAX_SIZE_MSG );
            if( logMsg == NULL ) {
```

```
                  printf( "%s: Out of memory in add_new_base().\n",
myName );
                  free( ipString );
                  exit (-1 );
            }
            sprintf( logMsg,
                  "New base added to ipmanage_ipplan.base by user %s.
New base address is %s.\n",
                  username, ipString );
            logEntry( logMsg, "/var/log/ipman/ipman.bind" ); //
ipman.bind because this is part of adding a new C-class
                                                      // and therefore
involves binding new IPs to the server.
            printf( "%s: %s", myName, logMsg );
      }
      else {
            logMsg = ( char * ) malloc( MAX_SIZE_MSG );
                if( logMsg == NULL ) {
                        printf( "%s: Out of memory in
add_new_base().\n", myName );
                  free( ipString );
                        exit (-1 );
                }
                sprintf( logMsg, "New C-class bound to IP host server
and added to ipman_db.ips_and_macs, but adding new base to
ipmanage_ipplan.base by user %s failed.  Base address that needs to be
added is %s.\n", username, ipString );
            logEntry( logMsg, "/var/log/ipman/ipman.err" );
            printf( "%s: %s", myName, logMsg );
      }
      free( ipString );
      free( logMsg );
      return( 0 );
}

void add_report( char *msg ) {

        FILE *out;

        if(( out = fopen( "/var/log/ipman/ipman.report", "a" )) == NULL
) {
                printf( "%s: Could not open /var/log/ipman/ipman.report
for append.\n", myName );
                printf( "%s: Scan aborting.  Contact an
administrator.\n", myName );
                exit( -1 );
        }

        fprintf( out, "%s", msg );
        fclose( out );
}

int bindips( const char *addr, char *mask ) {

      char *octet1, *octet2, *octet3, *octet4;  /* Octets as strings */
      int oct1, oct2, oct3, oct4;            /* Octets as ints */
      FILE *fd;     /* /etc/ips */
```

```
      int i;  /* all-purpose counter */
      char *output; /* for output to file /etc/ips */
      char *tempch; /* temp char for removing \n from strings */
      char *tempAddr; /* holds value of addr */
      int lower = 0; /* lower bound of range of IPs to bind */
      int upper = 0; /* upper bound of IP range */
      enum e_input { INVALID, SINGLE_IP, IP_RANGE, C_BLOCK };
      typedef enum e_input input;
      input inputType;
      char *logMsg;
      int MAX_SIZE_OCTET = 4;
      query queryType = DELETE;
      char bind_anyway;
      int returnValue = 0; // for return values
      boolean scan = TRUE; // value to set on
ipman_db.ips_and_macs.scan

      /* Tokenize the string */
      /* Replace \n with \0 */
        if(( tempch = strchr( addr, '\n' )) != NULL )
                *tempch = '\0';

      if(( tempch = strchr( mask, '\n' )) != NULL )
                *tempch = '\0';

      tempAddr = ( char * ) malloc( MAX_SIZE_IPADDR + 1 );
      if( tempAddr == NULL ) {
            fprintf( stderr, "%s: Out of memory in bindips() on ln
172.\n", myName );
                free( tempAddr );
            return( -1 );
      }
      strcpy( tempAddr, addr );
      inputType = validate_addr( tempAddr, &lower, &upper );

      logMsg = ( char * ) malloc( MAX_SIZE_MSG );
      if( logMsg == NULL ) {
                fprintf( stderr, "%s: Out of memory in bindips() ln
299.\n", myName );
                exit( -1 );
        }

      output = (char * ) malloc(  MAX_SIZE_IPADDR + 1 +
MAX_SIZE_NETMASK  + 1 );
      if( output == NULL ) {
            fprintf( stderr, "%s: Out of memory in bindips() on ln
172.\n", myName );
                return( -1 );
        }

      switch( inputType ) {

            case INVALID:
                  printf( "%s: The address or range %s is invalid.\n",
myName, addr );
                  free( tempAddr );
                  free( logMsg );
```

```
                  free( output );
                  return( 1 );
                  break;

            case SINGLE_IP:

                  if(( fd = fopen( "/etc/ips", "a" )) == NULL ) {
                        printf( "%s: Could not open /etc/ips for append
in bindip() ln 177.\n", myName );
                        free( tempAddr );
                              free( logMsg );
                              free( output );
                        return ( -1 );
                  }

                  strcpy( output, addr );
                  if (( tempch = strchr( output,'\n' )) != NULL ) {
                        *tempch = '\0';
                        }

                  if( ip_is_bound( output )) {
                        printf( "%s: IP %s is already bound.\n",
myName, output );
                        free( tempAddr );
                              free( logMsg );
                              free( output );
                        return( 1 );
                  }

                  returnValue = query_ipmanage( tempAddr, queryType );
                        if( returnValue != 0 ) {
                                printf( "%s: Record was not deleted
from ipmanage_ipplan\n", myName );
                        printf( "Continue with bind operation for IP
address %s [y/n]: ", output );
                        bind_anyway =  getchar(); getchar();
                        if( bind_anyway == 'n' ) {
                              printf( "%s: Bind operation for IP %s
cancelled.\n", myName, output );
                              return( 1 );
                        }
                        }
                  add_iphostserver_mac( tempAddr );
                  set_scannable( tempAddr, scan );
                        strcat( output, ":" );
                        strcat( output, mask );

                        printf( "%s\n", output );
                        fprintf( fd, "%s\n", output );

                  sprintf( logMsg, "IP address %s bound to IP host
server by user %s.\n", tempAddr, username );
                  logEntry( logMsg, "/var/log/ipman/ipman.bind" );
                  printf( "%s: %s", myName, logMsg );
                  delete_unboundip( tempAddr );
                  break;
            case C_BLOCK:
```

```
                  // This entails adding individual IPs to ips_and_macs
and /etc/ips, adding baseindex to
                  // ipmanage_ipplan.base
                  octet1 = ( char * ) malloc( MAX_SIZE_OCTET );
                  if( octet1 = NULL ) {
                        printf( "Out of memory in bindips().\n" );
                                free( tempAddr );
                                free( logMsg );
                                free( output );
                                return( -1 );
                  }
                  octet2 = ( char * ) malloc( MAX_SIZE_OCTET );
                  if( octet2 = NULL ) {
                                printf( "Out of memory in bindips().\n"
);
                                free( tempAddr );
                                free( logMsg );
                                free( output );
                        free( octet1 );
                                return( -1 );
                        }
                  octet3 = ( char * ) malloc( MAX_SIZE_OCTET );
                  if( octet3 = NULL ) {
                                printf( "Out of memory in bindips().\n"
);
                                free( tempAddr );
                                free( logMsg );
                                free( output );
                        free( octet1 );
                        free( octet2 );
                                return( -1 );
                        }
                  octet4 = ( char * ) malloc( MAX_SIZE_OCTET );
                  if( octet4 = NULL ) {
                                printf( "Out of memory in bindips().\n"
);
                                free( tempAddr );
                                free( logMsg );
                                free( output );
                        free( octet1 );
                        free( octet2 );
                        free( octet3 );
                                return( -1 );
                        }

                  strcpy( tempAddr, addr );
                  sscanf( tempAddr, "%d.%d.%d.%d", &oct1, &oct2, &oct3,
&oct4 );

                  add_new_addr_block( oct1, oct2, oct3 );
                  /* No break as we also do the stuff below.  When
setting logMsg we will have to
                     test value of inputType to differentiate between
adding a range or a C block.
                   */
            case IP_RANGE:
```

```
                  if(( fd = fopen( "/etc/ips", "a" )) == NULL ) {
                        printf( "%s: Could not open /etc/ips for append
in bindip() ln 177.\n", myName );
                        return ( -1 );
                  }

                  strcpy( output, addr );
                  if (( tempch = strchr( output,'\n' )) != NULL ) {
                        *tempch = '\0';
                        }
                  strcpy( tempAddr, addr );
                  /* Tokenize string to get octets 1 through 3. */
                  octet1 = strtok( tempAddr, "." );
                  octet2 = strtok( NULL, "." );
                  octet3 = strtok( NULL, "." );
                  for( i = lower; i <= upper; ++i ) { /* lower and
upper set by call to validate_addr() */

                        sprintf( output, "%s.%s.%s.%d", octet1, octet2,
octet3, i );
                        if( ip_is_bound( output ) )
                              printf( "%s: %s is already bound.\n",
myName, output );
                        else {
                              // If we are adding a new C-block, we
don't want to query
                              // ipmanage_ipplan; there will be no
recoid there
                              if( inputType != C_BLOCK ) {
                                    returnValue = query_ipmanage(
output, queryType );
                              }
                              if( returnValue == 0 ) {
                                    strcat( output, ":" );
                                    strcat( output, mask );
                                    strcat( output, "\n" );
                                    fprintf( fd, "%s", output );
                                    add_iphostserver_mac( tempAddr );
                                          set_scannable( tempAddr, scan
);
                              }
                              else {
                                    // it didn't go well, the record
was not deleted from ipmanage
                                    sprintf( logMsg, "The record for IP
address %s could not be deleted from ipmanage_ipplan.ipaddr.
query_ipmanage() returned %d.  The address has not been bound to the IP
host server.\n", output, returnValue );
                                          logEntry( logMsg,
"/var/log/ipman/ipman.err" );
                                          printf( "%s: %sContact an
admin.\n", myName,  logMsg );
                              }
                        }
                  }
                  if( inputType == C_BLOCK )
```

```
                        sprintf( logMsg, "Class C address block %s
added to database and bound to IP host server by user %s.\n", addr,
username );
                  else
                        sprintf( logMsg, "IP address range %s bound to
IP host \
                              server by user %s.\n", addr, username );
                  logEntry( logMsg, "/var/log/ipman/ipman.bind" );
                  printf( "%s: %s", myName, logMsg );
                  break;
            default:
                  printf( "%s:  Something horrible has happened in
bindips() ln 288", myName );
                  free( output );
                  return( -1 );
      }/* switch( inputType ) */
      fclose( fd );
      free( logMsg );
      free( output );
      if( reload_ipaliases()) { // returns 0 if no error, so if
reload_ipaliases() != 0 (false) something bad...
            printf( "Error reloading WHM ipaliases; contact an
administrator.\n" );
            return( -1 );
      }

      return( 0 );
}

unsigned int dotted_to_int( char *addr ) {

        unsigned int answer;
        unsigned int oct1 = 0, oct2 = 0, oct3 = 0, oct4 = 0;
        unsigned int FIRST_OCT_MULTIPLIER = 16777216;
        unsigned int SECOND_OCT_MULTIPLIER = 65536;
        unsigned int THIRD_OCT_MULTIPLIER = 256;

        sscanf( addr, "%d.%d.%d.%d", &oct1, &oct2, &oct3, &oct4 );
        answer = oct1 * FIRST_OCT_MULTIPLIER + oct2 *
SECOND_OCT_MULTIPLIER + oct3 * THIRD_OCT_MULTIPLIER + oct4;
        return( answer );
}

void delete_unboundip( char *ip ){
      MYSQL conn;
        char *server = "localhost";
        char *mysqlUser = "user";
        char *mysqlPass = "pass";
        char *mysqlDB = "ipman_db";
      char *mysqlQuery;
        int MAX_SIZE_QUERY = strlen( "DELETE FROM unbound_ips WHERE
ip_addr = 1234567890" ) + 1;
      unsigned int ipv4 = dotted_to_int( ip );

      mysql_init( &conn );
      if( !mysql_real_connect( &conn, server, mysqlUser, mysqlPass,
mysqlDB, 0, NULL, 0 )) {
```

```
                printf( "%s\n", mysql_error( &conn ));
            printf( "%s: Unable to delete potential record for %s from
ipman_db.unbound_ips.\n", myName );
            printf( "Contact an administrator.\n" );
                free( mysqlQuery );
                mysql_close( &conn );
                return;
        }
      mysqlQuery = ( char * ) malloc( MAX_SIZE_QUERY );
      if( mysqlQuery == NULL ) {
            printf( "%s: Out of memory in delete_unboundip().\n" );
            exit( -1 );
      }
      sprintf( mysqlQuery, "DELETE FROM unbound_ips WHERE ip_addr =
%u", ipv4 );
      mysql_real_query( &conn, mysqlQuery, strlen( mysqlQuery ) );
        free( mysqlQuery );
      mysql_close( &conn );
      return;
}

int get_base( const char *ip ) {
      MYSQL conn;
      MYSQL_RES *res;
        MYSQL_ROW row;
        char *server = "localhost";
        char *mysqlUser = "root";
        char *mysqlPass = "kimBall0508";
        char *mysqlDB = "ipmanage_ipplan";
      int MAX_SIZE_QUERY = strlen( "SELECT baseindex FROM base WHERE
descrip LIKE 'NET-XXX-XXX-XXX%'" );
      char *mysqlQuery;

      unsigned int answer;
        unsigned int oct1 = 0, oct2 = 0, oct3 = 0, oct4 = 0;
      int returnValue; // mysqlAffected_rows() return
      sscanf( ip, "%d.%d.%d.%d.", &oct1, &oct2, &oct3, &oct4 );

      mysqlQuery = ( char * ) malloc ( MAX_SIZE_QUERY + 1 );
        if( mysqlQuery == NULL ) {
                printf( "%s: Out of memory in get_base().\n", myName );
                exit( -1 );
        }

      sprintf( mysqlQuery, "SELECT baseindex FROM base WHERE descrip
LIKE 'NET-%d-%d-%d%%'", oct1, oct2, oct3 );
      mysql_init( &conn );
        if( !mysql_real_connect( &conn, server, mysqlUser, mysqlPass,
mysqlDB, 0, NULL, 0 )) {
                printf( "%s\n", mysql_error( &conn ));
            free( mysqlQuery );
            mysql_close( &conn );
            return( -1 );
      }
      mysql_real_query( &conn, mysqlQuery, strlen( mysqlQuery ) );
        free( mysqlQuery );
```

```
      if( !( res = mysql_store_result( &conn ))) {
                printf( "%s\n", mysql_error( &conn ));
            mysql_close( &conn );
            return( -1 );
      }
      returnValue = mysql_affected_rows( &conn );
      if( returnValue < 1 ) {
            printf( "%s: No base index for IP address %s in
ipmanage_ipplan.base. Bind operation cancelled.\n", myName, ip );
            return( -1 );
      }
      row = mysql_fetch_row( res );
      sscanf( row[ 0 ], "%d", &answer );
      mysql_close( &conn );
      return( answer );
} // get_base()

void get_mac( char *ip_addr ) {

      pcap_t *pkt_descriptor;        // like a file descriptor for
packets
        struct bpf_program prog_buff;  // space for compiled filter
        char error_buf[ PCAP_ERRBUF_SIZE ];
      char libnet_error_buf[ LIBNET_ERRBUF_SIZE ];
        const u_char *packet;
        char *interface = "eth0";
        int x; // all-purpose counter
        int MAX_SIZE_FILTER = strlen( "arp src host " ) +
MAX_SIZE_IPADDR + 1;
      int t; // test for libnet errors
      //stuff for forking the process and IPC
      pid_t child_pid;
      char *shared_mem; // This will be shared for getting the MAC into
target_mac[]
      int shmseg_id;

        //stuff for building the packet
        libnet_t *context;
        u_int32_t temp_ip = 0;
        struct libnet_ether_addr *ptr_hwaddr;
        char src_ip[ IP_ALEN ], dest_ip[ IP_ALEN ];
        char dest_mac[ ETH_ALEN ] = { 0xff, 0xff, 0xff, 0xff, 0xff,
0xff }; // We don't know this, need to broadcast.
        char src_mac[ ETH_ALEN ];  // Will get assigned later
        int c = 0, p = 0;  // just counters
        char *filter;   // the pcap filter so that we only pick up
packets of interest
        //build the ARP request packet

        // create the libnet session handle
        if((context = libnet_init( LIBNET_LINK, "eth0",
libnet_error_buf )) == NULL ) {
            printf( "libnet_init failed: %s", libnet_error_buf );
      }
        // pack the source and destination IPs
        temp_ip = libnet_name2addr4( context, "147.202.49.28",
LIBNET_DONT_RESOLVE );
```

```
        memcpy( src_ip, ( char * ) &temp_ip, IP_ALEN );
        temp_ip = libnet_name2addr4( context, ip_addr,
LIBNET_DONT_RESOLVE );
        memcpy( dest_ip, ( char * ) &temp_ip, IP_ALEN );

        // get the source MAC address
        ptr_hwaddr = libnet_get_hwaddr( context );
        memcpy(src_mac, ptr_hwaddr, ETH_ALEN);

        // create the header structures
        static libnet_ptag_t arp = 0, eth = 0;

        // start at the top of the stack--create the ARP headers
      arp = libnet_build_arp(
                ARPHRD_ETHER,
                ETHERTYPE_IP,
                ETH_ALEN,
                IP_ALEN,
                ARPOP_REQUEST,
                src_mac,
                (u_int8_t *)src_ip,
                dest_mac,
                (u_int8_t *)dest_ip,
                NULL,
                0,
                context,
                0);
      if( arp == -1 ) {
            printf( "Error in libnet_build_arp: %s", libnet_geterror(
context ));
      }

        // now add the Ethernet headers
        eth = libnet_build_ethernet(
                dest_mac,
                src_mac,
                ETHERTYPE_ARP,
                NULL,
                0,
                context,
                0);
      if( eth == -1 ) {
                printf( "Error in libnet_build_ethernet: %s",
libnet_geterror( context ));
        }
        // packet is built, let's get stuff read to capture the packets
we request
        // open for packet capture

        pkt_descriptor = pcap_open_live( interface, BUFSIZ, 0, 2000,
error_buf );
        if( pkt_descriptor == NULL ) {
                printf( "pcap function pcap_open_live(): %s", error_buf
);
                exit( 1 );
        }
```

```
        filter = ( char * ) malloc( MAX_SIZE_FILTER );
      if( filter == NULL ) {
            printf( "%s:  Out of memory in get_mac().\n", myName );
            exit( -1 );
      }

        sprintf( filter, "arp src host %s", ip_addr );

      if( pcap_compile( pkt_descriptor, &prog_buff, filter, 1, 0 ) < 0
) {
               printf( "pcap function pcap_compile(): %s", error_buf );
               exit( 1 );
        }

        if( pcap_setfilter( pkt_descriptor, &prog_buff ) < 0 ) {
                printf( "pcap function pcap_setfilter(): %s", error_buf
);
                exit( 1 );
        }
      temp_mac = ( char * ) malloc( MAX_SIZE_ETHADDR + 1 );
      if( temp_mac == NULL ) {
            printf( "%s: Out of memory in get_mac().\n", myName );
            free( filter );
            exit( -1 );
      }
        // Start sending and receiving packets
      // First allocate the shared memory
      shmseg_id = shmget( IPC_PRIVATE, MAX_SIZE_ETHADDR, IPC_CREAT |
IPC_EXCL | S_IRUSR | S_IWUSR );
        for( x = 0; x < 5; x++ ) {
                t = libnet_write( context );
            if( t == -1 ) {
                  printf( "libnet_write error: %s\n", libnet_geterror(
context ));
            }
            child_pid = fork();
            if( child_pid == 0 ) {  // this is the child process
                  // attach to shared memory
                  shared_mem = ( char * ) shmat( shmseg_id, 0, 0 );
                  pcap_dispatch( pkt_descriptor, 1, ( void *
)pcap_callback_fct, NULL );
                  //convert to upper case
                  to_upper( temp_mac );
                  memcpy( shared_mem, temp_mac, MAX_SIZE_ETHADDR );
                  exit( 0 );
            }
            else {
                  sleep( .1 );
                  shared_mem = ( char * ) shmat( shmseg_id, 0, 0 );
                  if( strlen( shared_mem ) == 0 ) {
                        // the child has blocked, kill it
                        kill( child_pid, SIGTERM );
                  }
            }
                memcpy( target_mac[ x ], shared_mem, MAX_SIZE_ETHADDR
);
        }//  for( x = 0; x < 5; x++ )
```

```
	// detach shared memory and deallocate
	  shmdt( shared_mem );
	  shmctl( shmseg_id, IPC_RMID, 0 );

	pcap_close( pkt_descriptor );
	  libnet_destroy( context );
	free( filter );
	free( temp_mac );
}

int get_user_id( void ) {

	MYSQL conn;
	MYSQL_RES *res;
	  MYSQL_ROW row;
	  char *server = "localhost";
	  char *mysqlUser = "user";
	  char *mysqlPass = "pass";
	  char *mysqlDB = "ipman_db";
	  int MAX_SIZE_QUERY = strlen( "SELECT MAX(staff_id) FROM
staff_members WHERE staff_name ='XXXXXXXXXXXXXXX'" );
	  int user_id = 0;
	  char *mysqlQuery;

	mysqlQuery = ( char * ) malloc ( MAX_SIZE_QUERY + 1 );
	  if( mysqlQuery == NULL ) {
		  fprintf( stderr, "%s: Out of memory in
get_user_id().\n", myName );
		  exit( -1 );
	  }
	// If there are duplicates for staff_members.staff_name, the
assumption is that the one with the
	// highest (most recent) staff_id is currently running the
program as the previous one is no longer
	// with the company

	sprintf( mysqlQuery, "SELECT MAX(staff_id) FROM staff_members
WHERE staff_name ='%s'", username );
	mysql_init( &conn );
	if( !mysql_real_connect( &conn, server, mysqlUser, mysqlPass,
mysqlDB, 0, NULL, 0 )) {
		  printf( "%s: Error on connect: %s\n", myName,
mysql_error( &conn ));
	      printf( "%s: Returning user_id = -1 from get_user_id().
Fix in ipman_db.unbound_ips.staff_id.\n" );
		  printf( "%s: Continuing execution normally.\n" );
		  free( mysqlQuery );
		  return( -1 );
	  }
	mysql_real_query( &conn, mysqlQuery, strlen( mysqlQuery ));
	free( mysqlQuery );
	if( mysql_errno( &conn ) ) {
	      printf( "%s: Error on SELECT: %s.\n", myName, mysql_error(
&conn ) );
	      printf( "%s: Returning user_id = -1 from get_user_id().
Fix in ipman_db.unbound_ips.staff_id.\n" );
	      printf( "%s: Continuing execution normally.\n" );
```

```
            return( -1 );
      }
      if( !( res = mysql_use_result( &conn ))) {
            printf( "%s: Error on mysql_use_result(): %s.\n", myName,
mysql_error( &conn ) );
                printf( "%s: Returning user_id = -1 from get_user_id().
Fix in ipman_db.unbound_ips.staff_id.\n" );
                printf( "%s: Continuing execution normally.\n" );
                return( -1 );
      }
      row = mysql_fetch_row( res );
      sscanf( row[ 0 ], "%d", &user_id );
      if( user_id < 1 ) {
            printf( "%s: Error getting user_id.\n", myName );
                printf( "%s: Returning user_id = -1 from get_user_id().
Fix in ipman_db.unbound_ips.staff_id.\n" );
                printf( "%s: Continuing execution normally.\n" );
                return( -1 );
        }
      return( user_id );
} // get_user_id()

char* int_to_dotted( unsigned int ipv4 ) {

        int remainder = 0;
        int octet1 = 0, octet2 = 0, octet3 = 0;
        unsigned int FIRST_OCT_MULTIPLIER = 16777216;
        unsigned int SECOND_OCT_MULTIPLIER = 65536;
        unsigned int THIRD_OCT_MULTIPLIER = 256;
        char *answer;

      answer = ( char * ) malloc( MAX_SIZE_IPADDR + 1 );
      if( answer == NULL ) {
            printf( "%s:  Out of memory in int_to_dotted().\n", myName
);
            exit( -1 );
      }

        octet1 = ipv4 / FIRST_OCT_MULTIPLIER;
        ipv4 %= FIRST_OCT_MULTIPLIER;
        octet2 = ipv4 / SECOND_OCT_MULTIPLIER;
        ipv4 %= SECOND_OCT_MULTIPLIER;
        octet3 = ipv4 / THIRD_OCT_MULTIPLIER;
        ipv4 %= THIRD_OCT_MULTIPLIER;
        sprintf( answer, "%d.%d.%d.%d", octet1, octet2, octet3, ipv4 );
        return( answer );
}

int is_numeric( char *str ) {

      while( *str ) {

            if( !isdigit( *str ) )
                  return( 0 );
            str++;
      }
      return( 1 );
```

```
}

int ip_is_bound( const char *addr ) {
      /*  Check /etc/ips to see if the ip is already bound to the IP
host host server.
            Here we are going to return false ( 0 ) or true ( 1 )
       */

      FILE *fd;     /* /etc/ips */
      int MAX_LINE_LENGTH = 32; // xxx.xxx.xxx.xxx:xxx.xxx.xxx.xxx\0
      char line[ MAX_LINE_LENGTH ];
      char tempAddr[ MAX_SIZE_IPADDR ];
      char *token;

      if(( fd = fopen( "/etc/ips", "r" )) == NULL ) {
            printf( "%s: Could not open /etc/ips for read in
ip_is_bound() ln 177.\n", myName );
                return ( -1 );
        }

      while( fgets( line, MAX_LINE_LENGTH, fd )) {
            token = strtok( line, ":" );
            strcpy( tempAddr, token );
            if( strcmp( addr, tempAddr) == 0 ) {
                  fclose( fd );
                  return( 1 );
            }
      } //while fgets

      fclose( fd );
        return( 0 );
}

int logEntry( const char *message, const char *file ) {

      FILE *out;
        char *time_buffer;
        time_t current_time;
        struct tm *local_time;
        int length;
      int MAX_TIME_BUFFER = 29;

        /*  Get time/date and create string */
      time_buffer = ( char * ) malloc( MAX_TIME_BUFFER );
      if( time_buffer == NULL ) {
            fprintf( stderr, "%s: Out of memory in logEntry() ln
368.\n", myName );
            exit( -1 );
      }

        current_time = time( NULL );
        local_time = localtime( &current_time );
        strcpy( time_buffer, asctime( local_time ));
        length = strlen( time_buffer );

        if( time_buffer[ length - 1 ] == '\n' )
                time_buffer[ length - 1 ] = ' ';
```

```
        if(( out = fopen( file, "a" )) == NULL ) {
                fprintf( stderr, "Could not open %s for append.\n",
file );
                return( -1 );
        }
        fprintf( out, "%s", time_buffer );
      free( time_buffer );
      fprintf( out, "%s", message );
      fclose( out );
        return( 0 );
}

int login( void ) {

      /*  Verifies login, returns 1 if OK and 0 on error.
          MD5 usage from http://www.openssl.org/docs/crypto/md5.html#
         */
      MYSQL conn;
        MYSQL_RES *res;
        MYSQL_ROW row;

        int MAX_PASSWORD = 16;
        int MAX_USERNAME = 16;

        char *server = "localhost";
        char *mysqlUser = "user";
        char *mysqlPass = "pass";
        char *mysqlDB = "ipman_db";
      char *mysqlQuery;
      int MAX_SIZE_QUERY = strlen( "SELECT password FROM staff_members
WHERE staff_name = 'xxxxxxxxxxxxxxx'" );

        char *password;

        char *tempChar;
        int charsRead;

        unsigned char hash[ MD5_DIGEST_LENGTH ];
        char *hashStr;
        int i;

      username = ( char * ) malloc( MAX_USERNAME + 1 );
      if( username == NULL ) {
            fprintf( stderr, "%s: Out of memory in login() ln 202.\n",
myName );
            free( username);
            exit( -1 );
      }

        printf( "Login as: " );
        charsRead = getline( &username, &MAX_USERNAME + 1, stdin );
      if( charsRead == 1 ) {
            fprintf( stderr, "%s: Username cannot be blank.\n", myName
);
            free( username );
            return( 0 );
```

```
      }

        /* Replace \n with \0 */
        if( (tempChar = strchr( username, '\n' )) != NULL )
                *tempChar = '\0';

        password = (char * ) malloc( MAX_PASSWORD + 1 );

      if( password == NULL ) {
            fprintf( stderr, "%s: Out of memory in login() ln 216.\n",
myName );
            free( username );
            free( password );
            exit( -1 );
      }

      printf( "Password for %s: ", username );

      strcpy( password, getpass( "" ));

      if( (tempChar = strchr( password, '\n' )) != NULL )
                *tempChar = '\0';

      if( strlen( password ) < 1 ) {
            fprintf( stderr, "%s: Password cannot be blank.\n", myName
);
            free( username );
            free( password );
            return( 0 );
      }

        MD5( password, strlen( password ), hash );
        free( password );

        hashStr = ( char * ) malloc ( MD5_DIGEST_LENGTH * 2 + 1 );
      if( hashStr == NULL ) {
            printf( "%s: Out of memory in login() ln 246.\n", myName );
            free( username );
            free( hashStr );
            exit( -1 );
      }

      for( i = 0; i < MD5_DIGEST_LENGTH; ++i ) {
                sprintf( hashStr + 2 * i, "%.2x", hash[ i ] );
      }

        mysqlQuery = ( char * ) malloc( MAX_SIZE_QUERY + 1 );
      if( mysqlQuery == NULL ) {
            printf( "%s: Out of memory in login() ln 257.\n", myName );
            free( username );
            free( hashStr );
            exit( -1 );
      }

        sprintf( mysqlQuery, "SELECT staff_password FROM staff_members
WHERE staff_name = '%s'", username );
        mysql_init( &conn );
```

```
        if( !mysql_real_connect( &conn, server, mysqlUser, mysqlPass,
mysqlDB, 0, NULL, 0 )) {
            printf( "%s\n", mysql_error( &conn ));
            free( username );
                free( hashStr );
                free( mysqlQuery );
            exit( -1 );
      }

      mysql_real_query( &conn, mysqlQuery, strlen( mysqlQuery ) );
        free( mysqlQuery );

      if( !( res = mysql_store_result( &conn ))) {
            printf( "%s\n", mysql_error( &conn ));
                free( username );
                free( hashStr );
                exit( -1 );
      }

      if( mysql_affected_rows( &conn ) < 1 ) {
            printf( "%s: Invalid login for user %s.\n", myName,
username );
            free( username );
            free( hashStr );
            mysql_close( &conn );
            return( 0 );
      }
        row = mysql_fetch_row( res );
      if( strcmp( hashStr, row[ 0 ] ) != 0 ) {
                printf( "%s: Invalid login for user %s.\n", myName,
username );
                free( hashStr );
            free( username );
                mysql_close( &conn );
            return( 0 );
        }

        printf( "user %s logged in.\n", username );
        free( hashStr );
        mysql_close( &conn );
        return( 1 );
}

int open_report( void ) {
      FILE *out;
      char *time_buffer;
        time_t current_time;
        struct tm *local_time;
      int length;
      int MAX_TIME_BUFFER = 29;
      /*  Get time/date and create string */

      if(( out = fopen( "/var/log/ipman/ipman.report", "w" )) ==  NULL
) {
                printf( "%s: Could not create report file
/var/log/ipman/ipman.report.\n", myName );
                return( -1 );
```

```
        }

      /*  Get time/date and create string */
        time_buffer = ( char * ) malloc( MAX_TIME_BUFFER );
        if( time_buffer == NULL ) {
                fprintf( stderr, "%s: Out of memory in logEntry() ln
368.\n", myName );
                exit( -1 );
        }
      current_time = time( NULL );
        local_time = localtime( &current_time );
        strcpy( time_buffer, asctime( local_time ));
        length = strlen( time_buffer );

        if( time_buffer[ length - 1 ] == '\n' )
                time_buffer[ length - 1 ] = ' ';

      fprintf( out, "%s:  ipman scan begun.\n\n", time_buffer );
      fprintf( out, "****************************************\n\n" );
      fclose( out );
      return( 0 );
}

int pass_one( void ) {

      // Scan ipman_db.ips_and_macs, move non-responders to
ipman_db.no_response,
      // report mismatches.
      // Return 0 for no mismatch, 1 on mismatch
      MYSQL conn;
        MYSQL conn_temp; // the xxx_temp vars here may be reset
immediately after use
        MYSQL_RES *res;
        MYSQL_RES *res_temp;
        MYSQL_ROW row;
        MYSQL_ROW row_temp;
        char *server = "localhost";
        char *mysqlUser = "user";
        char *mysqlPass = "pass";
        char *mysqlDB = "ipman_db";
        int MAX_SIZE_QUERY = strlen( "SELECT ip_addr, released_to FROM
unbound_ips WHERE ip_addr = 1234567890" );
        char *mysqlQuery;
        char *ip;
        char *mac;
        unsigned int ipv4; // IP as unsigned int
        int x = 0; // counter
        int pass = 1; // this will keep track of which pass we are on,
mostly important for passes 2 and 3
        // For the report:  The booleans are for the mailed digest on
error types
        boolean mismatch = FALSE;
        boolean no_response = FALSE;
        boolean old_unbound = FALSE;
        boolean unbound_response = FALSE; // This one is reset at the
bottom of the loop through unbound_ips
        FILE *out;
```

```
        char *reportMsg;

        /* Scan IPs in pman_db.ips_and_macs.
           Return 0 if no mismatch, 1 if a mismatch.
         */

        if( open_report() == -1 ) {
                printf( "%s:  Report file could not be opened, scan
aborted.\n", myName );
                return( -1 );
        }
        // Mismatch report header
        reportMsg = ( char * ) malloc( MAX_SIZE_MSG );
        if( reportMsg == NULL ) {
                printf( "%s: Out of memory in pass_one().  Scan
aborting, contact an adminstrator.\n", myName );
                exit( -1 );
        }

        sprintf( reportMsg, "IP - MAC Address
mismatches\n**************************\n" );
        add_report( reportMsg );

      // Get list of IPs to scan.
        ip = ( char * ) malloc( MAX_SIZE_MSG );
        if( ip == NULL ) {
                printf( "%s: Out of memory in pass_one().  Scan
aborting, contact an adminstrator.\n", myName );
                free( reportMsg );
                exit( -1 );
        }

        mysqlQuery = ( char * ) malloc( MAX_SIZE_QUERY + 1 );
        if( mysqlQuery == NULL ) {
                printf( "%s: Out of memory in pass_one().\n", myName );
                free( ip );
                free( reportMsg );
                exit( -1 );
        }
        mysql_init( &conn );
        if( !mysql_real_connect( &conn, server, mysqlUser, mysqlPass,
mysqlDB, 0, NULL, 0 )) {
                printf( "%s\n", mysql_error( &conn ));
                free( mysqlQuery );
            free( ip );
            free( reportMsg );
                mysql_close( &conn );
                return( -1 );
        }
        sprintf( mysqlQuery, "SELECT ip_addr, mac_addr FROM
ips_and_macs WHERE scan = 1" );

      mysql_real_query( &conn, mysqlQuery, strlen( mysqlQuery ));
        if( !( res = mysql_store_result( &conn ))) {
                printf( "%s\n", mysql_error( &conn ));
                free( mysqlQuery );
            free( ip );
```

```
            free( reportMsg );
                mysql_close( &conn );
                return( -1 );
        }


        mac = ( char * ) malloc( MAX_SIZE_ETHADDR + 1 );
        if( mac == NULL ) {
                printf( "%s: Out of memory in pass_one().\n", myName );
                free( mysqlQuery );
                free( ip );
            free( reportMsg );
                exit( -1 );
        }

        while( row = mysql_fetch_row( res )) {
                ipv4 = strtoul( row[ 0 ], NULL, 10 );
                ip = int_to_dotted( ipv4 );
                sprintf( mac, "%s", row[ 1 ] );
                for( x = 0; x < 5; x++ )
                        sprintf( target_mac[ x ], "" );
                get_mac( ip );

            // first, deal with a non-responder.
                if( strlen( target_mac[ 0 ] ) == 0 &&
                    strlen( target_mac[ 1 ] ) == 0 &&
                    strlen( target_mac[ 2 ] ) == 0 &&
                    strlen( target_mac[ 3 ] ) == 0 &&
                    strlen( target_mac[ 4 ] ) == 0 ) {
                        if( !ip_is_bound( ip )) {
                                // add the IP address to non-responders
                                mysql_init( &conn_temp );
                                sprintf( mysqlQuery, "INSERT INTO
no_response VALUES(%u, 1, NOW())", ipv4 );
                                if( !mysql_real_connect( &conn_temp,
server, mysqlUser, mysqlPass, mysqlDB, 0, NULL, 0 )) {
                                        printf( "%s\n", mysql_error(
&conn_temp ));
                                        printf( "%s: IP address %s did
not respond, but there was an error adding it to the table
no_response.\n", myName, row[ 0 ] );
                                }
                                else {
                                         mysql_real_query( &conn_temp,
mysqlQuery, strlen( mysqlQuery ));
                                }
                                mysql_close( &conn_temp );
                        }
                }

            // otherwise, compare the stuff in target_mac with
mac...they should be the same.  Or, if the
            // packet got dropped, the array member for that packet
should be a zero-length string.
                else {
                        if(( strcmp( mac, target_mac[ 0 ] ) == 0 ||
strlen( target_mac[ 0 ] ) == 0 ) &&
```

```
                            ( strcmp( mac, target_mac[ 1 ] ) == 0 ||
strlen( target_mac[ 1 ] ) == 0 ) &&
                            ( strcmp( mac, target_mac[ 2 ] ) == 0 ||
strlen( target_mac[ 2 ] ) == 0 ) &&
                            ( strcmp( mac, target_mac[ 3 ] ) == 0 ||
strlen( target_mac[ 3 ] ) == 0 ) &&
                            ( strcmp( mac, target_mac[ 4 ] ) == 0 ||
strlen( target_mac[ 4 ] ) == 0 )) {

                                    // All is good
/*  We don't really need this stuff here, unless the decision is made
later to leave non-responders in the
    no_response table between scans.
                                    sprintf( mysqlQuery, "DELETE FROM
no_response WHERE ip_addr = %u", ipv4 );
                                    mysql_init( &conn_temp );
                                    if( !mysql_real_connect( &conn_temp,
server, mysqlUser, mysqlPass, mysqlDB, 0, NULL, 0 )) {
                                            printf( "%s\n", mysql_error(
&conn_temp ));
                                            printf( "%s:  Please remove
ip_addr %u manually from no_response.\n", myName, ipv4 );
                                    }
                                    else {
                                            mysql_real_query( &conn_temp,
mysqlQuery, strlen( mysqlQuery ) );
                                    }
                                    mysql_close( &conn_temp );
 */
                            }
                            else {
                                    // A stolen IP address! Report it!

                                    printf( "%s:  The IP address %s does
not report its assigned MAC address of %s. This will be reported
shortly.\n", myName, ip, row[ 1 ] );
                                    mismatch = TRUE;
                                    reportMsg = ( char * ) malloc(
MAX_SIZE_MSG );
                                    if( reportMsg == NULL ) {
                                            printf( "%s: Out of memory
pass_one().  Scan aborted, contact adminstrator.\n", myName );
                                            exit( -1 );
                                    }
                                    sprintf( reportMsg, "IP address %s
should have MAC %s, but reported:\n%s\n%s\n%s\n%s\n%s\n.",
                                            ip, row[ 1 ], target_mac[ 0 ],
target_mac[ 1 ], target_mac[ 2 ],
                                            target_mac[ 3 ], target_mac[ 4
] );
                                    add_report( reportMsg );
                                    free( reportMsg );

                            }
                    }
            } // while
```

```
      free( reportMsg );
      free( ip );
      free( mac );
      free( mysqlQuery );
      mysql_free_result( res );
      mysql_close( &conn );

      if( mismatch == FALSE )
            return( 0 );
      else
            return( 1 );
}

int pass_two_three( void ) {

      // Scan ipman_db.no_response, report mismatches
      // Return 0 if no mismatches, 1 if there is a mismatch
        MYSQL conn;
        MYSQL conn_temp; // the xxx_temp vars here may be reset
immediately after use
        MYSQL_RES *res;
        MYSQL_RES *res_temp;
        MYSQL_ROW row;
        MYSQL_ROW row_temp;
        char *server = "localhost";
        char *mysqlUser = "user";
        char *mysqlPass = "pass";
        char *mysqlDB = "ipman_db";
        int MAX_SIZE_QUERY = strlen( "SELECT no_response.ip_addr,
mac_addr FROM no_response, ips_and_macs WHERE no_response.ip_addr =
ips_and_macs.ip_addr" );
        char *mysqlQuery;
        char *ip;
        char *mac;
        unsigned int ipv4; // IP as unsigned int
        int x = 0; // counter
        int pass = 1; // this will keep track of which pass we are on,
mostly important for passes 2 and 3
        // For the report:  The booleans are for the mailed digest on
error types
        boolean mismatch = FALSE;
        boolean no_response = FALSE;
        boolean old_unbound = FALSE;
        boolean unbound_response = FALSE; // This one is reset at the
bottom of the loop through unbound_ips
        FILE *out;
        char *reportMsg;

        /* Scan IPs in pman_db.ips_and_macs.
           Return 0 if no mismatch, 1 if a mismatch.
         */

        if( open_report() == -1 ) {
                printf( "%s:  Report file could not be opened, scan
aborted.\n", myName );
                return( -1 );
        }
```

```
        // Mismatch report header
        reportMsg = ( char * ) malloc( MAX_SIZE_MSG );
        if( reportMsg == NULL ) {
                printf( "%s: Out of memory in pass_one().  Scan
aborting, contact an adminstrator.\n", myName );
                exit( -1 );
        }

        sprintf( reportMsg, "IP - MAC Address
mismatches\n**************************\n" );
        add_report( reportMsg );

      // Get list of IPs to scan.
        ip = ( char * ) malloc( MAX_SIZE_MSG );
        if( ip == NULL ) {
                printf( "%s: Out of memory in pass_one().  Scan
aborting, contact an adminstrator.\n", myName );
                free( reportMsg );
                exit( -1 );
        }

        mysqlQuery = ( char * ) malloc( MAX_SIZE_QUERY + 1 );
        if( mysqlQuery == NULL ) {
                printf( "%s: Out of memory in pass_one().\n", myName );
                free( ip );
                free( reportMsg );
                exit( -1 );
        }
        mysql_init( &conn );
        if( !mysql_real_connect( &conn, server, mysqlUser, mysqlPass,
mysqlDB, 0, NULL, 0 )) {
                printf( "%s\n", mysql_error( &conn ));
                free( mysqlQuery );
                free( ip );
                free( reportMsg );
                mysql_close( &conn );
                return( -1 );
        }
        sprintf( mysqlQuery,
            "SELECT no_response.ip_addr, mac_addr FROM no_response,
ips_and_macs WHERE no_response.ip_addr = ips_and_macs.ip_addr" );

      mysql_real_query( &conn, mysqlQuery, strlen( mysqlQuery ));
      if( mysql_errno( &conn )) {
            printf( "%s\n", mysql_error( &conn ));
            free( mysqlQuery );
            free( ip );
            free( mac );
            free( reportMsg );
            mysql_close( &conn );
            return( -1 );
      }
        if( !( res = mysql_store_result( &conn ))) {
                printf( "%s\n", mysql_error( &conn ));
                free( mysqlQuery );
                free( ip );
                free( reportMsg );
```

```
                mysql_close( &conn );
                return( -1 );
        }

      mac = ( char * ) malloc( MAX_SIZE_ETHADDR + 1 );
        if( mac == NULL ) {
                printf( "%s: Out of memory in pass_one().\n", myName );
                free( mysqlQuery );
                free( ip );
                free( reportMsg );
                exit( -1 );
        }

        while( row = mysql_fetch_row( res )) {
                ipv4 = strtoul( row[ 0 ], NULL, 10 );
                ip = int_to_dotted( ipv4 );
                sprintf( mac, "%s", row[ 1 ] );
                for( x = 0; x < 5; x++ )
                        sprintf( target_mac[ x ], "" );
                get_mac( ip );

            // first, deal with a non-responder.
                if( strlen( target_mac[ 0 ] ) == 0 &&
                    strlen( target_mac[ 1 ] ) == 0 &&
                    strlen( target_mac[ 2 ] ) == 0 &&
                    strlen( target_mac[ 3 ] ) == 0 &&
                    strlen( target_mac[ 4 ] ) == 0 ) {
                        if( !ip_is_bound( ip )) {
                                // add the IP address to non-responders
                                mysql_init( &conn_temp );
                                sprintf( mysqlQuery, "INSERT INTO
no_response VALUES(%u, 1, NOW())", ipv4 );
                                if( !mysql_real_connect( &conn_temp,
server, mysqlUser, mysqlPass, mysqlDB, 0, NULL, 0 )) {
                                        printf( "%s\n", mysql_error(
&conn_temp ));
                                        printf( "%s: IP address %s did
not respond, but there was an error adding it to the table
no_response.\n", myName, row[ 0 ] );
                                }
                                else {
                                         mysql_real_query( &conn_temp,
mysqlQuery, strlen( mysqlQuery ));
                                }
                                mysql_close( &conn_temp );
                        }
                }
// otherwise, compare the stuff in target_mac with ip...they should be
the same.  Or, if the
// packet got dropped, the array member for that packet should be a
zero-length string.
                else {
                        if(( strcmp( mac, target_mac[ 0 ] ) == 0 ||
strlen( target_mac[ 0 ] ) == 0 ) &&
                           ( strcmp( mac, target_mac[ 1 ] ) == 0 ||
strlen( target_mac[ 1 ] ) == 0 ) &&
```

```
                            ( strcmp( mac, target_mac[ 2 ] ) == 0 ||
strlen( target_mac[ 2 ] ) == 0 ) &&
                            ( strcmp( mac, target_mac[ 3 ] ) == 0 ||
strlen( target_mac[ 3 ] ) == 0 ) &&
                            ( strcmp( mac, target_mac[ 4 ] ) == 0 ||
strlen( target_mac[ 4 ] ) == 0 )) {

                                    // All is good
/*  We don't really need this stuff here, unless the decision is made
later to leave non-responders in the
    no_response table between scans.
                                    sprintf( mysqlQuery, "DELETE FROM
no_response WHERE ip_addr = %u", ipv4 );
                                    mysql_init( &conn_temp );
                                    if( !mysql_real_connect( &conn_temp,
server, mysqlUser, mysqlPass, mysqlDB, 0, NULL, 0 )) {
                                            printf( "%s\n", mysql_error(
&conn_temp ));
                                            printf( "%s:  Please remove
ip_addr %u manually from no_response.\n", myName, ipv4 );
                                    }
                                    else {
                                            mysql_real_query( &conn_temp,
mysqlQuery, strlen( mysqlQuery ) );
                                    }
                                    mysql_close( &conn_temp );
 */
                            }
                            else {
                                    // A stolen IP address! Report it!

                                    printf( "%s:  The IP address %s does
not report its assigned MAC address of %s. This will be reported
shortly.\n", myName, ip, row[ 1 ] );
                                    mismatch = TRUE;
                                    reportMsg = ( char * ) malloc(
MAX_SIZE_MSG );
                                    if( reportMsg == NULL ) {
                                            printf( "%s: Out of memory
pass_one().  Scan aborted, contact adminstrator.\n", myName );
                                            exit( -1 );
                                    }
                                    sprintf( reportMsg, "IP address %s
should have MAC %s, but reported:\n%s\n%s\n%s\n%s\n%s\n.",
                                            ip, row[ 1 ], target_mac[ 0 ],
target_mac[ 1 ], target_mac[ 2 ],
                                            target_mac[ 3 ], target_mac[ 4
] );
                                    add_report( reportMsg );
                                    free( reportMsg );

                            }
                    }
            } // while

            free( reportMsg );
            free( ip );
```

```
        free( mac );
        free( mysqlQuery );
        mysql_free_result( res );
        mysql_close( &conn );

        if( mismatch == FALSE )
                return( 0 );
        else
                return( 1 );

}

void pcap_callback_fct( u_char *what, struct pcap_pkthdr *pkt_header,
u_char *packet ) {

        struct ether_header *eth_header;  //  net/ethernet.h
        struct ether_arp *arp_header;   // linux/if_arp.h
        u_char src_ha[ 19 ]; //source hardware address
        u_char src_pa[ 17 ]; //source protocol address

        eth_header = ( struct ether_header * ) packet;

        // If it's an ARP packet, get some ARP info from it:
        if( ntohs( eth_header->ether_type ) == ETHERTYPE_ARP ) {
                arp_header = ( struct ether_arp * ) ( packet + sizeof(
struct ether_header ) );
                if( arp_header->ea_hdr.ar_op == ntohs( ARPOP_REPLY ) )
{
                        snprintf( src_ha, sizeof( src_ha ) - 1,
"%02x:%02x:%02x:%02x:%02x:%02x:",
                                arp_header->arp_sha[ 0 ], arp_header-
>arp_sha[ 1 ],
                                arp_header->arp_sha[ 2 ], arp_header-
>arp_sha[ 3 ],
                                arp_header->arp_sha[ 4 ], arp_header-
>arp_sha[ 5 ] );

                        snprintf( src_pa, sizeof( src_pa ) - 1,
"%d.%d.%d.%d",
                                arp_header->arp_spa[ 0 ], arp_header-
>arp_spa[ 1 ],
                                arp_header->arp_spa[ 2 ], arp_header-
>arp_spa[ 3 ] );
                        memcpy( temp_mac, src_ha, 19 );
                }
        }
}


void print_usage( void ) {

      printf( "Usage: ipman [OPTIONS]\n" );
      printf( "Manage IP addresses by comparing IP address to
authorized Ethernet address.\n\n" );
      printf( "--bind,\t\t-b\tBind an IP address or an entire class C
to the local interface.\n" );
```

```
      printf( "--check=IP,\t-c\tCheck a single IP address (but don't
log a mismatch).\n" );
      printf( "--help,\t\t-h\tPrint this help and exit.\n" );
      printf( "--lookup=MAC,\t-l\tLook up authorized IP addresses for a
given MAC address.\n" );
      printf( "--noalert,\t-n\tPrint alerts to stdout rather than
sending alert\n" );
      printf( "--scan,\t\t-s\tScan IPs in local database and send alert
if there is a mismatch.\n" );
      printf( "--unbind,\t-u\tUnbind an IP address and mark the IP as
assigned in ipmanage.net.\n" );
      printf( "\nNote that only one option can be chosen with the
exception of --noalert (-n) with --scan (-s).\n" );
}

int query_ipmanage( char *ip, query action ) {

      // enum  queryType { INSERT, DELETE };
      // all the variables for getting info from user:
      unsigned int ipv4 = 0;
        int MAX_SIZE_DB_FIELD = 80;
        char *label;
        char *location;
        char *server_type;
        int charsRead;
        char verify;
        char *tempChar; // for replacing newline with null terminator

        // variables for providing info to user:
      int monitor; // for holding the value of the row count
        // Variables for the query
        MYSQL conn;
        MYSQL_RES *res;
        MYSQL_ROW row;

        int base = get_base( ip ); // baseindex from
ipmanage_ipplan.base
      if( base == -1 ) { //  No baseindex found!
            printf( "%s: No base index found for IP address %s.
Operation cancelled.\n", myName, ip );
            return( -1 );
      }
        char *server = "localhost";
        char *mysqlUser = "user";
        char *mysqlPass = "pass";
        char *mysqlDB = "ipmanage_ipplan";
        char *mysqlQuery;
        int MAX_SIZE_QUERY = strlen( "INSERT INTO ipaddr VALUES
(1234567890, '', '', '555-555-5555', '', 123456, NOW(),
'xxxxxxxxxxxxxxx'" ) + MAX_SIZE_DB_FIELD * 3;
                // Each of the blank fields here could be 80 characters
        int count;
      // for logging
        char *logMsg;
      char *old_server_label;

        label = ( char * ) malloc( MAX_SIZE_DB_FIELD );
```

```
        if( label == NULL ) {
                fprintf( stderr, "%s: Out of memory in
query_ipmanage().\n", myName );
                exit( -1 );
        }

        location = ( char * ) malloc( MAX_SIZE_DB_FIELD );
        if( location == NULL ) {
                fprintf( stderr, "%s: Out of memory in
query_ipmanage().\n", myName );
                free( label );
                exit( -1 );
        }

      server_type = ( char * ) malloc( MAX_SIZE_DB_FIELD );
        if( server_type == NULL ) {
                fprintf( stderr, "%s: Out of memory in
query_ipmanage().\n", myName );
                free( label );
                free( location );
                exit( -1 );
        }

        ipv4 = dotted_to_int( ip );

        if( action == INSERT ) {
                // Get server label, location, server type
                printf( "Enter the server label: " );
                charsRead = getline( &label, &MAX_SIZE_DB_FIELD, stdin
);
                if( charsRead == 1 ) {
                        fprintf( stderr, "%s: Server label cannot be
blank.\n", myName );
                        return( 1 );
                }
                /* Replace \n with \0 */
                if(( tempChar = strchr( label, '\n' )) != NULL )
                        *tempChar = '\0';

                printf( "Enter the server DC and cabinet: " );
                charsRead = getline( &location, &MAX_SIZE_DB_FIELD,
stdin );
                if( charsRead == 1 ) {
                        fprintf( stderr, "%s: Server location cannot be
blank.\n", myName );
                        return( 1 );
                }
                /* Replace \n with \0 */
                if(( tempChar = strchr( location, '\n' )) != NULL )
                        *tempChar = '\0';
                printf( "Enter the server type (dedicated, colo, etc.):
" );
                charsRead = getline( &server_type, &MAX_SIZE_DB_FIELD,
stdin );
                if( charsRead == 1 ) {
                        fprintf( stderr, "%s: Server type cannot be
blank.\n", myName );
```

```
                        return( 1 );
                }
                /* Replace \n with \0 */
                if(( tempChar = strchr( server_type, '\n' )) != NULL )
                        *tempChar = '\0';

                //  Echo and verify
                printf( "Adding a record for IP address %s for the
following server:\n", ip );
                printf( "Server label: %s\nLocation: %s\nType: %s\n",
label, location, server_type );
                printf( "Add this record [y/n]: " );
                verify = getchar(); getchar(); // swallow the carriage
return

                if( verify == 'y' ) {
                        // Then run the query
                        mysqlQuery = (char * ) malloc( MAX_SIZE_QUERY +
1 );
                        if( mysqlQuery == NULL ) {
                                fprintf( stderr, "%s: Out of memory in
query_ipmanage.\n", myName );
                                free( label );
                                free( location );
                                free( server_type );
                                exit( -1 );
                        }
                        // Check to see if there is record for the
address
                        sprintf( mysqlQuery, "SELECT COUNT(*) AS
num_rows FROM ipaddr WHERE ipaddr = %u", ipv4 );

                        mysql_init( &conn );
                        if( !mysql_real_connect( &conn, server,
mysqlUser, mysqlPass, mysqlDB, 0, NULL, 0 )) {
                                printf( "%s\n", mysql_error( &conn ));
                                free( label );
                                free( location );
                                free( server_type );
                                free( mysqlQuery );
                                mysql_close( &conn );
                                exit( -1 );
                        }

                        mysql_real_query( &conn, mysqlQuery, strlen(
mysqlQuery ));

                        if( !( res = mysql_use_result( &conn ))) {
                                printf( "%s\n", mysql_error( &conn ));
                                free( label );
                                free( location );
                                free( server_type );
                                free( mysqlQuery );
                                fprintf( stderr, "Invalid results from
mysql_use_results in query_ipmanage.\n" );
                        }
```

```
                        row = mysql_fetch_row( res );
                    sscanf( row[ 0 ], "%d", &count );
                        if( count > 0 ) {
                                printf( "IP address %s alread in table
ipmanage_ipplan.ipaddr", ip );
                                mysql_close( &conn );
                                return( 1 );
                        }
                        mysql_free_result( res );

                    // No record for the IP exists, go ahead and add it.
                        sprintf( mysqlQuery, "INSERT INTO ipaddr VALUES
(%u, '%s', '%s', '555-555-5555', \
                                '%s', %d, NOW(), '%s')",
                                ipv4, label, location, server_type,
base, username );

                        mysql_real_query( &conn, mysqlQuery, strlen(
mysqlQuery ));

                        if( mysql_affected_rows( &conn ) < 1 ) { // The
record was not added.  Log it.
                                printf( "The record was not added.  The
query was %s.  Contact an administrator.\n", mysqlQuery );
                                free( label );
                                free( location );
                                free( server_type );
                                free( mysqlQuery );

                                logMsg = ( char * ) malloc(
MAX_SIZE_MSG );
                                if( logMsg == NULL ) {
                                        printf( "%s: Out of memory in
query_ipmanage.\n" );
                                        mysql_close( &conn );
                                        exit( -1 );
                                }
                                sprintf( logMsg, "Failed to add record
for address %s to ipmanage_ipplan.  Query: %s.\n", ip, mysqlQuery );
                                logEntry( logMsg,
"/var/log/ipman/ipman.err" );
                        printf( "%s: %s.  Contact an administrator.\n",
myName, logMsg );
                                free( logMsg );
                                mysql_close( &conn );
                                return( 1 );
                        }

                        logMsg = ( char * ) malloc( MAX_SIZE_MSG );
                        if( logMsg == NULL ) {
                                printf( "%s: Out of memory in
query_ipmanage.\n" );
                                free( label );
                                free( location );
                                free( server_type );
                                free( mysqlQuery );
                                mysql_close( &conn );
```

```
                                exit( -1 );
                        }
                        sprintf( logMsg, "IP address %s assigned to
server %s by user %s.\n", ip, label, username );
                        logEntry( logMsg, "/var/log/ipman/ipman.unbind"
);
                  printf( "%s: %s", myName, logMsg );
                  free( logMsg );
                        return( 0 );
                }
                else { // user did not verify the entry
                  printf( "%s:  Operation cancelled.\n", myName );
                        free( label );
                        free( location );
                        free( server_type );
                        mysql_close( &conn );
                        return( 1 );
                }

        }
        else { // action == DELETE

                mysqlQuery = (char * ) malloc( MAX_SIZE_QUERY + 1 );
                if( mysqlQuery == NULL ) {
                        printf( "%s: Out of memory in
query_ipmanage.\n", myName );
                        free( label );
                        free( location );
                        free( server_type );
                        exit( -1 );
                }
                // Check to see if there is record for the address
                sprintf( mysqlQuery, "SELECT userinf FROM ipaddr WHERE
ipaddr = %u", ipv4 );

                mysql_init( &conn );
                if( !mysql_real_connect( &conn, server, mysqlUser,
mysqlPass, mysqlDB, 0, NULL, 0 )) {
                        printf( "%s\n", mysql_error( &conn ));
                        free( label );
                        free( location );
                        free( server_type );
                        free( mysqlQuery );
                        exit( -1 );
                }

                mysql_real_query( &conn, mysqlQuery, strlen( mysqlQuery
));
                res = mysql_store_result( &conn );
            monitor = mysql_affected_rows( &conn );
                if( monitor < 1 ) {
                        printf( "There is no record to delete for IP
address %s.\n", ip );
                  free( label );
                  free( location );
                        free( server_type );
                        free( mysqlQuery );
```

```
                  mysql_free_result( res );
                  mysql_close( &conn );
                        return( 1 );
                }

            row = mysql_fetch_row( res );
                printf( "Delete record for IP address %s, assigned to
%s [y/n]: ", ip, row[ 0 ] );
                verify = getchar();

            if( verify == 'y' ) {
                  // free old result before new query, but first save
old server label
                  old_server_label = ( char * ) malloc(
MAX_SIZE_DB_FIELD );
                  if( old_server_label == NULL ) {
                        printf( "%s: Out of memory in
query_ipmanage().\n", myName );
                        free( label );
                                free( location );
                                free( server_type );
                                free( mysqlQuery );
                        mysql_close( &conn );
                        exit( -1 );
                  }

                  sprintf( old_server_label, "%s", row[ 0 ] );
                  mysql_free_result( res );

                        sprintf( mysqlQuery, "DELETE FROM ipaddr WHERE
ipaddr = %u", ipv4 );
                        mysql_real_query( &conn, mysqlQuery, strlen(
mysqlQuery ));
                        if( mysql_affected_rows( &conn ) < 1 ) {
                                printf( "Record for IP address %s was
not deleted.", ip );
                                logMsg = (char * ) malloc( MAX_SIZE_MSG
);
                                if( logMsg == NULL ) {
                                        printf( "%s:  Out of memory in
query_ipmanage().\n" );
                                        free( label );
                                        free( location );
                                        free( server_type );
                                        free( mysqlQuery );
                                   free( old_server_label );
                                        exit( -1 );
                                  }
                                  sprintf( logMsg, "Attempt to delete
ipmanage_ipplan record for %s failed (query: \
                                        %s)\n", mysqlQuery );
                                  logEntry( logMsg,
"/var/log/ipman/ipman.err" );
                        printf( "%s: %s Contact an administrator.\n",
myName, logMsg );
                                return( 1 );
                        }
```

```
                        else { // record was deleted, log it in
ipman.bind
                               // (since this would be part of a
binding operation)
                                printf( "Record for IP address %s was
deleted.\n", ip );
                                logMsg = (char * ) malloc( MAX_SIZE_MSG
);
                                if( logMsg == NULL ) {
                                        printf( "%s:  Out of memory in
query_ipmanage().\n" );
                                        free( label );
                                        free( location );
                                        free( server_type );
                                        free( mysqlQuery );
                                        exit( -1 );
                                }
                                sprintf( logMsg, "Record for IP %s
deleted by user %s. Record formerly allocated to server %s.\n", ip,
username, old_server_label );
                                logEntry( logMsg,
"/var/log/ipman/ipman.bind" );
                        printf( "%s: %s", myName, logMsg );
                                return( 0 );
                        }
                }
                printf( "Delete operation cancelled by user.\n" );
                return( 1 );
        } // action = DELETE

} // query_ipmanage()


int reload_ipaliases( void ) {

      /* fork a child, exec ipaliases, and exit.
           parent sleeps until child exits and returns proper exit
code.*/

      pid_t childpid;
      int status;

      if(( childpid = fork()) < 0 ) {
            printf( "%s: Error forking child process in
reload_ipaliases() ln 380.\n", myName );
            return( -1 );
      }
      else {
            if( childpid == 0 ) {
                  system( "/sbin/ifdown eth0" );
                  system( "/sbin/ifup eth0" );
                  system( "/scripts/ipaliases reload" );
                  exit( 0 );
            }
            else {
                  /* this is the parent, wait( for childpid )*/
                  while( childpid != wait( &status ))
```

```
                        ; /* Just waiting */
                  if( status ) {
                        printf( "%s: Error in child:
reload_ipaliases().\ n", myName );
                        return( -1 );
                  }
            }
      }

      return( 0 );
}

int report_non_responders( void ) {

      MYSQL conn;
        MYSQL_RES *res;
        MYSQL_ROW row;
        char *server = "localhost";
        char *mysqlUser = "user";
        char *mysqlPass = "pass";
        char *mysqlDB = "ipman_db";
        int MAX_SIZE_QUERY = strlen( "SELECT ip_addr, released_to FROM
unbound_ips WHERE ip_addr = 1234567890" );
        char *mysqlQuery;
        char *ip;
        char *mac;
      char *reportMsg;
        unsigned int ipv4; // IP as unsigned int
        int x = 0; // counter
      boolean no_response = FALSE;

      ip = ( char * ) malloc( MAX_SIZE_IPADDR );
      if( ip = NULL ) {
            printf( "%s:  Out of memory in report_non_reponders().\n",
myName );
            exit( -1 );
      }

      mac  = ( char * ) malloc( MAX_SIZE_ETHADDR );
        if( mac = NULL ) {
                printf( "%s:  Out of memory in
report_non_reponders().\n", myName );
            free( ip );
                exit( -1 );
        }

      mysqlQuery  = ( char * ) malloc( MAX_SIZE_QUERY );
        if( mac = NULL ) {
                printf( "%s:  Out of memory in
report_non_reponders().\n", myName );
                free( ip );
            free( mac );
                exit( -1 );
        }

      mysql_init( &conn );
```

```
        if( !mysql_real_connect( &conn, server, mysqlUser, mysqlPass,
mysqlDB, 0, NULL, 0 )) {
                printf( "mysql_real_connect():  %s\n", mysql_error(
&conn ));
                printf( "%s:  Third pass completed, mismatches
reported, but remaining IPs in no_response never responded.\n",
myName);
                printf( "%s:  Report on non-responders not completed.
Contact an administrator.\n", myName );
                free( mysqlQuery );
                free( ip );
                free( mac );
                mysql_close( &conn );
                return( -1 );
        }
        sprintf( mysqlQuery, "SELECT ip_addr FROM no_response" );

      mysql_real_query( &conn, mysqlQuery, strlen( mysqlQuery ));
      if( mysql_errno( &conn )) {
            printf( "mysql_real_query:  %s\n", mysql_error( &conn ));
      }
      if( !( res = mysql_store_result( &conn ))) {
                printf( "mysql_store_result():  %s\n", mysql_error(
&conn ));
                printf( "%s:  Third pass completed, mismatches
reported, but remaining IPs in no_response never responded.\n",
 myName);
                printf( "%s:  Report on non-responders not completed.
Contact an administrator.\n", myName );
                free( mysqlQuery );
                free( ip );
                free( mac );
                mysql_close( &conn );
                return( -1 );
        }

      if( mysql_affected_rows( &conn ) > 0 ) {
                no_response = TRUE;

            // Add no_response report header
                reportMsg = ( char * ) malloc( MAX_SIZE_MSG + 1 );
                if( reportMsg == NULL ) {
                        printf( "%s: Out of memory in
report_non_response().\n", myName );
                  printf( "Mismatches reported, non-responders in table
no_response have not responded.\n" );
                        printf( "Unbound IPs have not been queried.
Contact an administrator.\n" );
                        exit( -1 );
                }
                sprintf( reportMsg,
                      "\nIP addresses not responding to ARP
requests\n**************************************\n" );
                add_report( reportMsg );

                while( row = mysql_fetch_row( res )) {
                        ipv4 = strtoul( row[ 0 ], NULL, 10 );
```

```
                        ip = int_to_dotted( ipv4 );
                        sprintf( reportMsg, "%s\n", ip );
                        add_report ( reportMsg );
                }
                sprintf( mysqlQuery, "DELETE FROM no_response" );
                mysql_real_query( &conn, mysqlQuery, strlen( mysqlQuery
));
            free( reportMsg );
      }
      mysql_free_result( res );
      mysql_close( &conn );
      free( ip );
      free( mac );
      free( mysqlQuery );

      if( no_response == FALSE )
            return( 0 );
      else
            return( 1 );

} // report_non_reponders()

int report_old_unbound( void ) {

      MYSQL conn;
      MYSQL_RES *res;
        MYSQL_ROW row;
        char *server = "localhost";
        char *mysqlUser = "user";
        char *mysqlPass = "pass";
        char *mysqlDB = "ipman_db";
        int MAX_SIZE_QUERY = strlen( "SELECT * FROM unbound_ips WHERE
DATE_SUB(CURDATE(),INTERVAL 2 DAY) > time_unbound" );
        char *mysqlQuery;
        char *ip;
        char *reportMsg;
        unsigned int ipv4; // IP as unsigned int
        int x = 0; // counter
        boolean old_unbound = FALSE;

      ip = ( char * ) malloc( MAX_SIZE_IPADDR );
      if( ip == NULL ) {
            printf( "%s:  Out of memory in report_old_unbound().  IP
address completed, mismatches\n", myName );
            printf( "non-responders, and first responses from recently
released addresses reported.\n" );
            printf( "Report will not be sent; contact an
administrator.\n" );
            exit( -1 );
      }

      mysqlQuery = ( char * ) malloc( MAX_SIZE_QUERY );
        if( mysqlQuery == NULL ) {
                printf( "%s:  Out of memory in report_old_unbound().
IP address completed, mismatches\n", myName );
                printf( "non-responders, and first responses from
recently released addresses reported.\n" );
```

```
                printf( "Report will not be sent; contact an
administrator.\n" );
                exit( -1 );
        }

      sprintf( mysqlQuery, "SELECT * FROM unbound_ips WHERE
DATE_SUB(CURDATE(),INTERVAL 2 DAY) > time_unbound" );
        mysql_init( &conn );
        if( !mysql_real_connect( &conn, server, mysqlUser, mysqlPass,
mysqlDB, 0, NULL, 0 )) {
                printf( "%s\n", mysql_error( &conn ));
                free( mysqlQuery );
                free( ip );
                free( reportMsg );
                mysql_close( &conn );
                return( -1 );
        }

      mysql_real_query( &conn, mysqlQuery, strlen( mysqlQuery ));

      if( !( res = mysql_store_result( &conn ))) {
                printf( "%s\n", mysql_error( &conn ));
                printf( "Third pass completed, mismatches and non-
responders reported, but IPs in unbound_ips\n" );
            printf( "not scanned to check for extensive period of being
unbound. Contact an administrator.\n" );
                free( mysqlQuery );
                free( ip );
                mysql_close( &conn );
                return( -1 );
        }

      if( mysql_affected_rows( &conn ) > 0 ) {
                old_unbound = TRUE;
             reportMsg = ( char * ) malloc( MAX_SIZE_MSG );
             if( reportMsg == NULL ) {
                   printf( "%s:  Out of memory in report_old_unbound().
IP address completed, mismatches\n", myName );
                   printf( "non-responders, and first responses from
recently released addresses reported.\n" );
                   printf( "Report will not be sent; contact an
administrator.\n" );
                   exit( -1 );
             }

                sprintf( reportMsg,
                        "IP addresses unbound for longer than two
days\n*******************************************\n" );
                add_report( reportMsg );
                sprintf( reportMsg,
"ip_addr\treleased_to\ttime_unbound\n" );
                add_report( reportMsg );
                while( row = mysql_fetch_row( res )) {
                        ipv4 = strtoul( row[ 0 ], NULL, 10 );
                        sprintf( ip, "%s", int_to_dotted( ipv4 ));
                        sprintf( reportMsg, "%s\t%s\t%s\n", ip, row[ 1
], row[ 2 ] );
```

```
                        add_report( reportMsg );
                    }
                free( reportMsg );
          }

        free( ip );
        free( mysqlQuery );
        mysql_close( &conn );
        if( old_unbound == FALSE )
              return( 0 );
        else
              return( 1 );

} //  report_old_unbound()

int report_unbound_response( void ) {

        MYSQL conn;
        MYSQL conn_temp;
        MYSQL_RES *res;
        MYSQL_RES *res_temp;
        MYSQL_ROW row;
        MYSQL_ROW row_temp;
        char *server = "localhost";
        char *mysqlUser = "user";
        char *mysqlPass = "pass";
        char *mysqlDB = "ipman_db";
        int MAX_SIZE_QUERY = strlen( "SELECT ip_addr, mac_addr FROM
ips_and_macs WHERE ip_addr = 1234567890" );
        char *mysqlQuery;
        char *ip;
        char *mac;
        char *reportMsg;
        unsigned int ipv4; // IP as unsigned int
        int x = 0; // counter
        boolean old_unbound = FALSE;
        boolean unbound_response = FALSE;
        mysqlQuery = ( char * ) malloc( MAX_SIZE_QUERY );

        if( mysqlQuery == NULL ) {
              printf( "%s: Out of memory in report_unbound_response().\n"
);
              exit( -1 );
        }

        ip = ( char * ) malloc( MAX_SIZE_QUERY );
          if( ip == NULL ) {
                    printf( "%s: Out of memory in
report_unbound_response().\n" );
                    free( mysqlQuery );
              exit( -1 );
          }

        mac = ( char * ) malloc( MAX_SIZE_QUERY );
          if( mac == NULL ) {
                    printf( "%s: Out of memory in
report_unbound_response().\n" );
```

```
                free( mysqlQuery );
            free( ip );
                exit( -1 );
        }

      reportMsg = ( char * ) malloc( MAX_SIZE_QUERY );
        if( reportMsg == NULL ) {
                printf( "%s: Out of memory in
report_unbound_response().\n" );
                free( mysqlQuery );
            free( ip );
            free( mac );
                exit( -1 );
        }

      sprintf( reportMsg, "\nNewly Allocated IP Addresses Responding to
Requests\n" );
      add_report( reportMsg );
      sprintf( reportMsg,
"************************************************\n" );
      add_report( reportMsg );
      sprintf( mysqlQuery, "SELECT ip_addr FROM unbound_ips" );
      mysql_init( &conn );
      if( mysql_errno( &conn )) {
            printf( "report_unbound_response(): %s\n", mysql_error(
&conn ));
            printf( "%s:  Third pass completed, mismatches and non-
responders reported, but scan of\n", myName );
                printf( "old unbound IPs aborted.  Contact an
administrator.\n" );
                free( mysqlQuery );
                free( ip );
                free( mac );
                mysql_close( &conn );
                return( -1 );
        }
      mysql_real_connect( &conn, server, mysqlUser, mysqlPass, mysqlDB,
0, NULL, 0 );
      if( mysql_errno( &conn )) {
                printf( "mysql_real_connect(): %s\n", mysql_error(
&conn ));
                printf( "%s:  Third pass completed, mismatches and non-
responders reported, but scan of\n", myName );
                printf( "old unbound IPs aborted.  Contact an
administrator.\n" );
                free( mysqlQuery );
                free( ip );
                free( mac );
                mysql_close( &conn );
                return( -1 );
        }
      mysql_real_query( &conn, mysqlQuery, strlen( mysqlQuery ));
      if( mysql_errno( &conn )) {
            printf( "%s\n", mysql_error( &conn ));
                printf( "mysql_real_query():  Third pass completed,
mismatches and non-responders reported,\n" );
```

```
                printf( "but scan of old unbound IPs aborted.  Contact
an administrator.\n" );
                free( mysqlQuery );
                free( ip );
                free( mac );
            mysql_close( &conn );
            return( -1 );
      }
      if( !( res = mysql_store_result( &conn ))) {
            printf( "%s\n", mysql_error( &conn ));
            printf( "mysql_store_result():  Third pass completed,
mismatches and non-responders reported,\n" );
            printf( "but scan of old unbound IPs aborted.  Contact an
administrator.\n" );
            free( mysqlQuery );
            free( ip );
            free( mac );
            mysql_close( &conn );
            return( -1 );
      }

      while( row = mysql_fetch_row( res )) {
            ipv4 = strtoul( row[ 0 ], NULL, 0 );
            sprintf( mysqlQuery, "SELECT ip_addr, mac_addr FROM
ips_and_macs WHERE ip_addr = %u", ipv4 );
            mysql_init( &conn_temp );
            mysql_real_connect( &conn_temp, server, mysqlUser,
mysqlPass, mysqlDB, 0, NULL, 0 );
            mysql_real_query( &conn_temp, mysqlQuery, strlen(
mysqlQuery ));
            if( !( res_temp = mysql_store_result( &conn_temp ))) {
                  printf( "%s\n", mysql_error( &conn_temp ));
                  printf( "%s:  Third pass completed, mismatches and
non-responders reported, but scan of\n", myName );
                  printf( "old unbound IPs aborted.  Contact an
administrator.\n" );
                        free( mysqlQuery );
                        free( ip );
                        free( mac );
                        mysql_free_result( res );
                        mysql_free_result( res_temp );
                        mysql_close( &conn_temp );
                        mysql_close( &conn );
                        return( -1 );
            }
            row_temp = mysql_fetch_row( res_temp );
            ipv4 = strtoul( row[ 0 ], NULL, 0 );
            ip = int_to_dotted( ipv4 );
            sprintf( mac, "%s", row_temp[ 1 ] );
            mysql_free_result( res_temp );
            mysql_close( &conn_temp );
            //Get rid of any junk in temp_mac[] from previous runs

            for( x= 0; x <= 4; x++ )
                  sprintf( target_mac[ x ], "" );

            get_mac( ip );
```

```
      // if it's not a non-responder, get it out of the table
        // We also want to assign one responder's MAC to mac for ease
of comparison next
            if( strlen( target_mac[ 0 ] ) != 0 ) {
                  sprintf( mac, "%s", target_mac[ 0 ] );
                        unbound_response = TRUE;
            }
            else if( strlen( target_mac[ 0 ] ) != 0 ) {
                  sprintf( mac, "%s", target_mac[ 0 ] );
                        unbound_response = TRUE;
            }
            else if( strlen( target_mac[ 0 ] ) != 0 ) {
                  sprintf( mac, "%s", target_mac[ 0 ] );
                  unbound_response = TRUE;
            }
            else if( strlen( target_mac[ 0 ] ) != 0 ) {
                  sprintf( mac, "%s", target_mac[ 0 ] );
                  unbound_response = TRUE;
            }
            else if( strlen( target_mac[ 0 ] ) != 0 ) {
                  sprintf( mac, "%s", target_mac[ 0 ] );
                  unbound_response = TRUE;
            }
            else if( strlen( target_mac[ 0 ] ) != 0 ) {
                  sprintf( mac, "%s", target_mac[ 0 ] );
                  unbound_response = TRUE;
            }

            if( unbound_response == TRUE ) { // Then remove it from
unbound_ips
                  mysql_init( &conn_temp );
                  mysql_real_connect( &conn_temp, server, mysqlUser,
mysqlPass, mysqlDB, 0, NULL, 0 );
                  sprintf( mysqlQuery, "DELETE FROM unbound_ips WHERE
ip_addr = %u", ipv4 );
                  mysql_real_query( &conn_temp, mysqlQuery, strlen(
mysqlQuery ));
                  // If all target_mac[]s agree, register the new mac
in ips_and_macs
                        if(( strcmp( mac, target_mac[ 0 ] ) == 0 ||
strlen( target_mac[ 0 ] ) == 0 ) &&
                     ( strcmp( mac, target_mac[ 1 ] ) == 0 || strlen(
target_mac[ 1 ] ) == 0 ) &&
                        ( strcmp( mac, target_mac[ 2 ] ) == 0 || strlen(
target_mac[ 2 ] ) == 0 ) &&
                        ( strcmp( mac, target_mac[ 3 ] ) == 0 || strlen(
target_mac[ 3 ] ) == 0 ) &&
                        ( strcmp( mac, target_mac[ 4 ] ) == 0 || strlen(
target_mac[ 4 ] ) == 0 )) {
                        sprintf( mysqlQuery,
                           "UPDATE ips_and_macs SET mac_addr = '%s',
scan = 1 WHERE ip_addr = %u",
                           mac, ipv4 );

                        mysql_real_query( &conn_temp, mysqlQuery,
strlen( mysqlQuery ));
```

```
                        if( mysql_errno( &conn_temp ))
                              printf( "%s\n", mysql_error( &conn_temp
));
                  }
                  else {
                        // if it responds but is not in agreement with
itself, put in
                        // ips_and_macs, but also report it as a stolen
IP.  Change the
                              // value for mac, obviously, so we never
get a match as long as
                        // this one is in the table
                        sprintf( mac, "FF:FF:FF:FF:FF:FF" );
                        sprintf( mysqlQuery,
                              "INSERT INTO ips_and_macs VALUES(%u,
'255.255.255.0', '%s', 1)",
                              ipv4, mac );
                        mysql_real_query( &conn_temp, mysqlQuery,
strlen( mysqlQuery ));
                        sprintf( ip, "%s", int_to_dotted( ipv4 ));
                        sprintf( reportMsg, "Previously unbound IP
address %s has an address conflict.\n" );
                        add_report( reportMsg );
                        logEntry( reportMsg, "/var/log/ipman/ipman.err"
);
                  }
            } // if( unbound_response == TRUE )

      }// while( row = mysql_fetch_row( res ))
      free( mysqlQuery );
        free( ip );
        free( mac );
        free( reportMsg );
      mysql_free_result( res );
      mysql_close( &conn );

      if( unbound_response == TRUE )
            return( 1 );
      else
            return( 0 );
}// report_unbound_response()

int scan( int report ) {

      boolean mismatches = FALSE; // any mistmatches reported
      boolean old_unbound = FALSE; // any IPs in unbound_ips for more
than 2 days
      boolean unbound_response = FALSE; // any previously unbound IPs
that now respond
      boolean no_response = FALSE;// unbound IPs that have now
responded
      int scan_result = 0; // scan functions return -1 on error, 0 on
no mismatches, 1 on a mismatch
      char *reportMsg;
      int x = 0; // all-purpose counter

      /* Scan IPs in the local database.
```

```
           Return 0 on any kind of error.
           If report = 0, supress alert, otherwise alert as needed.
         */

        /* First pass -- mismatches will be reported immediately.
           Non-responders will be added to ipman_db.no_response
         */
        if( open_report() == -1 ) {
                printf( "%s:  Report file could not be opened, scan
aborted.\n", myName );
                return( -1 );
        }
        // Mismatch report header
        reportMsg = ( char * ) malloc( MAX_SIZE_MSG );
        if( reportMsg == NULL ) {
                printf( "%s: Out of memory in scan().  Scan aborting,
contact an adminstrator.\n", myName );
                exit( -1 );
        }

        sprintf( reportMsg, "IP - MAC Address
mismatches\n**************************\n" );
        add_report( reportMsg );

      scan_result = pass_one();
      if( scan_result == -1) {
            printf( "%s:  Scan failed.  Contact an administrator.\n" );
      }
      else {
            mismatches = ( scan_result == 0 )? FALSE : TRUE;
      }

      // Second pass -- mismatches will be reported immediately.

      scan_result = pass_two_three();
      if( scan_result == -1 ) {
            printf( "%s:  Second pass of the scan failed.  Contact an
administrator.\n", myName );
      }
      else {
            if( mismatches == FALSE ) {
                  mismatches = ( scan_result == 0 )? FALSE : TRUE;
            }
      }

      printf( "%s: Sleeping for 600 seconds.\n", myName );
        printf( "There will be a countdown for entertainment
purposes.\n" );
        for( x = 600; x > 0; x-- ) {
            if( x % 10 == 0 )
                  printf( "%d", x );
            else
                  printf( "." );
            fflush( stdout );
            sleep( 1 );
      }
      printf( "\n" );
```

```
      // Third pass -- mismatches will be reported immediately.
      scan_result = pass_two_three();
        if( scan_result == -1 ) {
                printf( "%s:  Third pass of the scan failed.  Contact
an administrator.\n", myName );
        }
        else {
                if( mismatches == FALSE ) {
                        mismatches = ( scan_result == 0 )? FALSE :
TRUE;
                }
        }

      scan_result = report_non_responders();
      if( scan_result == -1 ) {
            printf( "%s:  Report on non-responding IPs failed.  Contact
an administrator.\n", myName );
      }
      else {
            no_response = ( scan_result == 0 )? FALSE : TRUE;
      }

      scan_result = report_unbound_response();
      if( scan_result == -1 ) {
                printf( "%s:  Report on newly responding IPs recently
allocated failed.  Contact an administrator.\n", myName );
        }
        else {
                unbound_response = ( scan_result == 0 )? FALSE : TRUE;
        }

      scan_result = report_old_unbound();
      if( scan_result == -1 ) {
            printf( "%s:  Report on IPs unbound for more than 2 days
failed.  Contact an administrator.\n", myName );

      }
      else {
            old_unbound = ( scan_result == 0 )? FALSE : TRUE;
      }

      // Report digest -- In this report: Mismatches - yes; Non-
responders - no; Never responded - yes.

      reportMsg = ( char * ) malloc( MAX_SIZE_MSG );
      if( reportMsg == NULL ) {
            printf( "%s: Out of memory in scan().  Report available at
/var/log/ipman/ipman.report but not sent.\n", myName );
            exit( -1 );
      }

        sprintf( reportMsg, "IPMan report available at
/var/log/ipman/ipman.report.\nIn this report:\n" );

        if( mismatches )  {
                strcat( reportMsg, "IP address mismatches\n" );
```

```
        }
        if( no_response ) {
                strcat( reportMsg, "IP addresses not responding to ARP
requests\n" );
        }
        if( old_unbound ) {
                strcat( reportMsg, "IP addresses not responding,
released to customer more than 2 days ago\n" );
        }
      if( unbound_response ) {
            strcat( reportMsg, "IP addresses released to customer and
now responding\n" );
      }

        strcat( reportMsg, "The report will be overwritten at the next
scan.\n" );
        send_report( reportMsg );
        free( reportMsg );
        printf( "%s: Report sent.\nIPMan scan complete.\n", myName );
        return( 0 );

} // scan()

int send_report( char *sendMsg ) {

        char *sysBuff;
        char *subjectMsg = "IPMan scan report";
        char *rcpt_to = "admin@servercompany.com";

        sysBuff = ( char * ) malloc( MAX_SIZE_MSG  );
      if( sysBuff == NULL ) {
            printf( "%s: Out of memory in send_report().\n" );
            exit( -1 );
      }
        sprintf ( sysBuff, "echo \"%s\" | mail -s '%s' %s", sendMsg,
subjectMsg, rcpt_to );
        system ( sysBuff );
        return( 0 );
}

int set_scannable( char *ip, boolean scan ) {

      MYSQL conn;
        char *server = "localhost";
        char *mysqlUser = "user";
        char *mysqlPass = "pass";
        char *mysqlDB = "ipman_db";
        int MAX_SIZE_QUERY = strlen( "UPDATE ips_and_macs SET scan = 0
WHERE ip_addr = 1234567890" );
        int x = 0;  // just a counter
        char *mysqlQuery;
      unsigned int ipv4 = dotted_to_int( ip );
        char *logMsg;

      mysql_init( &conn );
      mysqlQuery = ( char * ) malloc ( MAX_SIZE_QUERY + 1 );
        if( mysqlQuery == NULL ) {
```

```
                printf( "%s: Out of memory in set_scannable() ln
181.\n", myName );
                exit( -1 );
        }
      sprintf( mysqlQuery, "UPDATE ips_and_macs SET scan = %d WHERE
ip_addr = %u", scan, ipv4 );
      mysql_real_connect( &conn, server, mysqlUser, mysqlPass, mysqlDB,
0, NULL, 0 );
      mysql_real_query( &conn, mysqlQuery, strlen( mysqlQuery ) );
      free( mysqlQuery );
      if( mysql_errno( &conn ) ) {
            printf( "%s: Error on UPDATE: %s.\n", myName, mysql_error(
&conn ) );
            free( mysqlQuery );
            logMsg = ( char * ) malloc( MAX_SIZE_MSG );
            if( logMsg == NULL ) {
                  printf( "%s: Out of memory in set_scannable. Error
updating ipman_db.ip_addr.scan for IP %s.\n", myName, ip );
                  exit( -1 );
            }
            sprintf( logMsg, "Error updating ipman_db.ip_addr.scan for
IP %s.  Error is %s. set_scannable scan = %d.\n", ip, mysql_error(
&conn ), scan );
            logEntry( logMsg, "/var/log/ipman/ipman.err" );
            printf( "%s: %sContact an administrator.\n", myName, logMsg
);
            free( logMsg );
            return( 1 );
      }
      return( 0 );
} // set_scannable()

void to_upper( char *the_string ) {

        int x;
        for( x = 0; x < strlen( the_string ); x++ )
                the_string[ x ] = toupper( the_string[ x ] );
        return;
}

int unbindip( const char *addr ) {

      /* open /etc/ips, copy lines that do NOT match addr
         to /etc/ips.tmp, delete /etc/ips, rename .tmp to ips.tmp

         Additionally, call query_ipmanage( char *ip, int *query_type
);
         Then call ipaliases reload, return 0 on success or -1 on error
       */
      FILE *in;
      FILE *out;
      char *ipstr;
      char *maskstr;
      char *tempstr;
      char *tokens = ":\n";
      int ipfound = 0;
      char *logMsg;
```

```
      query queryType = INSERT;
      boolean scan = FALSE; // value to set for
ipman_db.ips_and_macs.scan
      if(( in = fopen( "/etc/ips", "r" )) == NULL ) {
            printf( "%s: Could not open /etc/ips.\n" );
            return( -1 );
      }

      if(( out = fopen( "/etc/ips.tmp", "w" )) ==  NULL ) {
            printf( "%s: Could not create temp file /etc/ips.tmp.\n" );
            return( -1 );
      }

      tempstr = ( char * ) malloc( MAX_SIZE_IPADDR + 1 +
MAX_SIZE_NETMASK + 1 );
      do {
            fscanf( in, "%s", tempstr );
            if( feof( in ) || ferror( in ) ) break;
            ipstr = strtok( tempstr, tokens );
            maskstr = strtok( NULL, tokens );

            if( strcmp( ipstr, addr ) != 0 )
                  fprintf( out, "%s:%s%s", ipstr, maskstr, "\n" );
            else
                  ipfound = 1;

      } while( !feof( in ) );

      sprintf( tempstr, "%s", addr );

      if( !ipfound )
            printf( "%s: IP address %s not bound to this server.\n",
myName, addr );
      else {
            logMsg = ( char * ) malloc( MAX_SIZE_MSG );
            if( logMsg == NULL ) {
                  fprintf( stderr, "%s: Out of memory in unbindip() ln
608.\n", myName );
                  free( tempstr );
                  exit( -1 );
            }
            // add record to ipmanage_ipplan.ipaddr

            if( query_ipmanage( tempstr, queryType ) != 0 ) {
                  sprintf( logMsg, "IP address %s unbound from IP host
server by user %s, but record in ipmanage_ipplan.ipaddr was not
successfully added.\n", addr, username );
                  printf( "%s: %sContact an administrator.\n", myName,
logMsg );
                  logEntry( logMsg, "/var/log/ipman/ipman.err" );
                  free( logMsg );
                  free( tempstr );
                  return( -1 );
            }
            if( set_scannable( tempstr, scan ) != 0 ) {
                  sprintf( logMsg, "ipman_db.ips_and_macs.scan not set
to 0 for IP address %s.\n", addr );
```

```
                  printf( "%s: %sFix it or contact an
administrator.\n", myName, logMsg );
                  logEntry( logMsg, "/var/log/ipman/ipman.err" );
            }
            update_unbound( tempstr, queryType );
            sprintf( logMsg, "IP address %s unbound from IP host server
by user %s.\n", addr, username );
            printf( "%s: %s", myName, logMsg );
            logEntry( logMsg, "/var/log/ipman/ipman.unbind" );
            free( logMsg );
      }

      free( tempstr );

      fclose( out );
      fclose( in );

      remove( "/etc/ips" );
      rename( "/etc/ips.tmp", "/etc/ips" );

      if( reload_ipaliases() ) {
            printf( "%s:  Error reloading WHM ipaliases, contact an
administrator.\n", myName );
            return( -1 );
      }

      return( 0 );
}

int update_unbound( char *ip, query queryType ) {


      MYSQL conn;
        char *server = "localhost";
        char *mysqlUser = "user";
        char *mysqlPass = "pass";
        char *mysqlDB = "ipman_db";
      int MAX_SIZE_DB_FIELD = 80;
        int MAX_SIZE_QUERY = strlen( "INSERT INTO unbound_ips
VALUES(1234567890, '', NOW(), 12345)" ) + MAX_SIZE_DB_FIELD;
        int x = 0;  // just a counter
        char *mysqlQuery;
        unsigned int ipv4 = dotted_to_int( ip );
        char *logMsg;
      char *released_to;
      int charsRead;
      char *tmpchr;
      int user_id;

        mysqlQuery = ( char * ) malloc ( MAX_SIZE_QUERY + 1 );
        if( mysqlQuery == NULL ) {
                printf( "%s: Out of memory in update_unbound() ln
181.\n", myName );
                exit( -1 );
        }
      if( queryType == INSERT ) {
            released_to = ( char * ) malloc( MAX_SIZE_DB_FIELD + 1 );
```

```
            if( released_to == NULL ) {
                  printf( "%s: Out of memory in update_unbound() ln
181.\n", myName );
                  exit( -1 );
            }
            printf( "Adding %s to unbound IPs table.  Enter label of
server released to: ", ip );
            charsRead = getline( &released_to, &MAX_SIZE_DB_FIELD + 1,
stdin );

            if ((tmpchr = strchr( released_to ,'\n')) != NULL) {
                        *tmpchr = '\0';
            }
            user_id = get_user_id();
            sprintf( mysqlQuery, "INSERT INTO unbound_ips VALUES( %u,
'%s', NOW(), %d)", ipv4, released_to, user_id );

            mysql_init( &conn );
            mysql_real_connect( &conn, server, mysqlUser, mysqlPass,
mysqlDB, 0, NULL, 0 );
            mysql_real_query( &conn, mysqlQuery, strlen( mysqlQuery )
);
            if( mysql_errno( &conn ) ) {
                  printf( "%s: Error on INSERT: %s.\n", myName,
mysql_error( &conn ) );
                  logMsg = ( char * ) malloc( MAX_SIZE_MSG + 1 );
                  if( logMsg == NULL ) {
                        printf( "%s: Out of memory in
update_unbound().\n", myName );
                              free( mysqlQuery );
                        free( released_to );
                        exit( -1 );
                  }
                  sprintf( logMsg, "Error adding to
ipman_db.unbound_ips for IP address %s: MySQL error was %s\n", ip,
mysql_error( &conn ));
                  printf( "%s: %sContact an administrator.\n", myName,
logMsg );
                  logEntry( logMsg, "/var/log/ipman/ipman.err" );
                  free( logMsg );
                  return( -1 );
            }
            logMsg = ( char * ) malloc( MAX_SIZE_MSG + 1 );
            if( logMsg == NULL ) {
                  printf( "%s: Out of memory in update_unbound().\n",
myName );
                  free( mysqlQuery );
                  free( released_to );
                  exit( -1 );
            }
            sprintf( logMsg, "IP address %s released to server %s and
entered in ipman_db.unbound_ips by user %s.\n", ip, released_to,
username );
            printf( "%s: %s", myName, logMsg );
            logEntry( logMsg, "/var/log/ipman/ipman.unbind" );
            mysql_close( &conn );
            free( mysqlQuery );
```

```
            free( released_to );
            return( 0 );
      }
      else {
            sprintf( mysqlQuery, "DELETE FROM unbound_ips WHERE ip_addr
= %u", ipv4 );
            mysql_init( &conn );
                mysql_real_connect( &conn, server, mysqlUser,
mysqlPass, mysqlDB, 0, NULL, 0 );
                mysql_real_query( &conn, mysqlQuery, strlen( mysqlQuery
) );
                if( mysql_errno( &conn ) ) {
                        printf( "%s: Error on DELETE: %s.\n", myName,
mysql_error( &conn ) );
                        logMsg = ( char * ) malloc( MAX_SIZE_MSG + 1 );
                        if( logMsg == NULL ) {
                                printf( "%s: Out of memory in
update_unbound().\n", myName );
                                free( mysqlQuery );
                                exit( -1 );
                        }
                        sprintf( logMsg, "Error deleting from
ipman_db.unbound_ips for IP address %s: MySQL error was %s\n", ip,
mysql_error( &conn ));
                        printf( "%s: %sContact an administrator.\n",
myName, logMsg );
                        logEntry( logMsg, "/var/log/ipman/ipman.err" );
                        free( logMsg );
                        return( -1 );
                }
                logMsg = ( char * ) malloc( MAX_SIZE_MSG + 1 );
                if( logMsg == NULL ) {
                        printf( "%s: Out of memory in
update_unbound().\n", myName );
                        free( mysqlQuery );
                        exit( -1 );
                }
                sprintf( logMsg, "IP address %s deleted from
ipman_db.unbound_ips by user %s.\n", ip, username );
                printf( "%s: %s", myName, logMsg );
                logEntry( logMsg, "/var/log/ipman/ipman.bind" );
                mysql_close( &conn );
                free( mysqlQuery );
                return( 0 );
        }

} // update_unbound()

int validate_addr( const char *addr, int *lbound, int *ubound ) {

      int octet1 = 0, octet2 = 0, octet3 = 0, octet4 = 0; /* Numeric
octets */
      char *strOctet1, *strOctet2, *strOctet3, *strOctet4; /* Octets as
strings */
      char tmpAddr[ MAX_SIZE_IPADDR + 1 ]; /* temp string for const
char */
      char *strLower;
```

```
      char *strUpper;
      int MAX_SIZE_OCTET = 4;
      enum inputType { INVALID, SINGLE_IP, IP_RANGE, C_BLOCK };

      enum inputType input;

      strOctet1 = ( char * ) malloc( MAX_SIZE_OCTET );
      strOctet2 = ( char * ) malloc( MAX_SIZE_OCTET );
      strOctet3 = ( char * ) malloc( MAX_SIZE_OCTET );
      strOctet4 = ( char * ) malloc( MAX_SIZE_OCTET );

      strcpy( tmpAddr, addr );
      strOctet1 = strtok( tmpAddr, "." );
      strOctet2 = strtok( NULL, "." );
      strOctet3 = strtok( NULL, "." );
      strOctet4 = strtok( NULL, "\n" );

      if( is_numeric( strOctet1 ) &&
          is_numeric( strOctet2 ) &&
          is_numeric( strOctet3 ) ) {
            sscanf( strOctet1, "%d", &octet1 );
            sscanf( strOctet2, "%d", &octet2 );
            sscanf( strOctet3, "%d", &octet3 );

      }
      else {
            input = INVALID;
            return( input );
      }

      sscanf( tmpAddr, "%d.%d.%d.%d", &octet1, &octet2, &octet3,
&octet4 );
      if( is_numeric( strOctet4 ) ) { /* We have a single IP address or
C block */
            sscanf( strOctet4, "%d", &octet4 );
            if( !  ((  octet1 >= 1 && octet1 <= 223 ) &&
                  (  octet2 >= 0 && octet2 <= 255 ) &&
                  (  octet3 >= 0 && octet3 <= 255 ) &&
                  (( octet4 >= 1 && octet4 <= 254 ) || octet4 == 0 ))
) {
                        input = INVALID;
                        return( input );
            }
            else if( octet4 == 0 ) { /* We have a C block */
                  *lbound = 2;
                  *ubound = 254;
                  input = C_BLOCK;
                  return( input );
            }
            else { /* A single IP */
                  *lbound = 0;
                  *ubound = 0;
                  input = SINGLE_IP;
                  return( input );
            }
      }
```

```
      else { /* We potentially have a range, verify it and return
INVALID or IP_RANGE */
            strLower = strtok( strOctet4, "-" );
            strUpper = strtok( NULL, "\n" );
            if( is_numeric( strLower ) ) {
                  sscanf( strLower, "%d", lbound );
            }
            else {
                  input = INVALID;
                  return( input );
            }

            if( is_numeric( strUpper ) ) {
                  sscanf( strUpper, "%d", ubound );
            }
            else {
                  input = INVALID;
                  return( input );
            }

            if( *lbound <= 1 || *ubound >= 255 || *lbound >= *ubound )
{
                  input = INVALID;
                  return( input );
            }
            else {
                  input = IP_RANGE;
                  return( input );
            }
      }
}
```

Appendix B

IPMAN_DB DATA DEFINITIONAL LANGUAGE

```
--
-- Table structure for table `ips_and_macs`
--

CREATE TABLE ips_and_macs (
  ip_addr int(11) unsigned NOT NULL default '0',
  netmask varchar(15) NOT NULL default '',
  mac_addr varchar(17) default NULL,
  scan tinyint(4) NOT NULL default '0',
  PRIMARY KEY  (ip_addr)
) TYPE=InnoDB;

--
-- Table structure for table `no_response`
--

CREATE TABLE no_response (
  ip_addr int(11) unsigned NOT NULL default '0',
  pass_num tinyint(4) NOT NULL default '0',
  pass_time timestamp(14) NOT NULL,
  PRIMARY KEY  (ip_addr),
  KEY IDX_no_response_FK (ip_addr),
  CONSTRAINT `no_response_ibfk_1` FOREIGN KEY (`ip_addr`) REFERENCES
`ips_and_macs` (`ip_addr`)
) TYPE=InnoDB;

--
-- Table structure for table `staff_members`
--

CREATE TABLE staff_members (
  staff_id int(11) NOT NULL default '0',
  staff_name varchar(16) default NULL,
  staff_password varchar(32) default NULL,
  userlevel int(11) NOT NULL default 1,
  PRIMARY KEY  (staff_id),
  KEY IDX_unbound_ips_FK (staff_id)
) TYPE=InnoDB;

--
-- Table structure for table `unbound_ips`
--

CREATE TABLE unbound_ips (
  ip_addr int(11) unsigned NOT NULL default '0',
  released_to varchar(80) NOT NULL default '',
  time_unbound datetime NOT NULL default '0000-00-00 00:00:00',
  staff_id int(11) NOT NULL default '0',
  PRIMARY KEY  (ip_addr),
  KEY IDX_unbound_ips_FK (ip_addr),
  KEY IDX_staff_id_FK (staff_id),
  CONSTRAINT `unbound_ips_ibfk_1` FOREIGN KEY (`ip_addr`) REFERENCES
`ips_and_macs` (`ip_addr`),
  CONSTRAINT `unbound_ips_ibfk_2` FOREIGN KEY (`staff_id`) REFERENCES
`staff_members` (`staff_id`)
) TYPE=InnoDB;
```