ARP Cache Poisoning Detection and Prevention

A Project

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In partial Fulfillment

of the Requirements for the Degree

Master of Computer Science

By

Silky Manwani

Dec 2003

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

Dr. Mark Stamp

Dr. Chris Pollett

Dr. David Blockus

APPROVED FOR THE UNIVERSITY

ACKNOWLEDGEMENTS

I would like to thank Professor Mark Stamp for his guidance, patience and insights without which my project would not have been possible. I would also like to thank Manoj Dutta, a Sr. Engineer at IPInfusion for giving me numerous pointers and advice that helped me finish this project.

Table of Contents

Index of figures

1. Abstract

Address resolution refers to the process of dynamically finding the Media Access Control (MAC) address of a computer on a network. The Address Resolution Protocol (ARP) thus provides a dynamic mapping between the two different forms of addresses: the 32-bit Internet Protocol (IP) address and the 48-bit MAC address that the data link layer uses.

ARP cache poisoning is the act of introducing a specious IP-to-Ethernet address mapping in another host's ARP cache [1]. This results in diversion of traffic, either to a different host on the LAN or no host at all. ARP spoofing, also known as the "Man In The Middle" attack, can thus be used to compromise the subnet. Even though ARP spoofing is possible only on a LAN it is still a security breach.

This report gives a brief introduction to Address Resolution Protocol. There is also a brief description on the Linux implementation of ARP functions and related important networking structures. The report then goes on to discuss ARP Cache Poisoning and provides a solution to detect and prevent it on RedHat Linux.

2. Introduction

2.1 Address Resolution Protocol

ARP is a protocol used by the IP network layer to map IP addresses to hardware addresses that are used by the data link layer. ARP operates below the network layer as a part of the Open Systems Interconnection (OSI) link layer, and is used when IP is used over the Ethernet.

There are two types of addresses that are used to uniquely identify a host:

- MAC Address

  This address is known by various names: hardware address, LAN address, physical address, or Network Interface Card (NIC) address. Each computer's network interface card is assigned a globally unique six-byte address by the factory that manufactured the card. This is the source physical address used by the host's network interface. When a host sends out an IP packet, it uses this source address and it receives all packets that match its own hardware address or the broadcast address. This Ethernet address, typically a 48-bit address, is a link layer address and depends on the network interface card used.

- IP Address

  Internet Protocol operates at the network layer and is independent of the hardware address. The IP address of a host is a 32-bit address assigned to a host and is either static or dynamically assigned by Dynamic Host Configuration Protocol (DHCP).

When an Ethernet frame is broadcast from one machine on a LAN to another, the 48-bit MAC address is used to determine the interface for which the frame is destined. The device driver does not consult the destination IP address in the IP datagram for the resolution of the address.

Address resolution refers to the process of dynamically finding a MAC address of a computer on a network. The protocol thus provides a dynamic mapping between the two different forms of addresses: the 32-bit IP address and the 48-bit hardware address that the data link layer uses. The process is dynamic as it happens automatically and is normally not a concern of either the application user or the system administrator.

In a shared Ethernet where hosts use the TCP/IP suite for communication, IP packets need to be encapsulated in Ethernet frames before they can be transmitted on to the wire. There is a one-to-one mapping between the set of IP addresses and the set of Ethernet addresses. Before the packet can be encapsulated in an Ethernet frame, the host sending the packet needs the recipient's link/Ethernet address. Therefore, ARP is used to find the destination Ethernet address using the IP address.

2.2 ARP Example

The example below shows how ARP is used on a LAN:

Suppose a user types the following command

% ftp linux.com

The following steps are executed:

1. The FTP client calls *gethostbyname()* to resolve the hostname linux.com into its 32-bit IP address. Domain Name System (DNS) is used to do this conversion.
2. Transport Control Protocol (TCP) is then asked to establish a connection with this 32-bit IP address.
3. An IP datagram is sent to this IP address to request connection to the remote machine.
4. The IP datagram can be sent directly to the destination host if it is on a LAN. Otherwise, IP determines the next hop router to which this packet needs to be sent. An IP datagram is then sent to this locally attached host or router.
5. On Ethernet, for a host to send an IP packet to a destination, it must know not only its IP address but also the 48-bit Ethernet address. It is therefore necessary to map the 32-bit IP address to the 48-bit MAC address. This is the core function of ARP.
6. A broadcast Ethernet frame called as ARP request is sent out on the LAN. This ARP request contains the IP address of the destination host.
7. Every host on the Ethernet receives this ARP request and the destination host recognizes that its hardware address is being asked for and thus responds with an

ARP reply. The ARP reply contains the 32-bit IP address and its 48-bit MAC address.

8. The host receives the ARP reply and is now ready to send the IP datagram.
9. The IP packet is sent to the destination.

The flowchart below shows a pictorial depiction of the above steps [5].

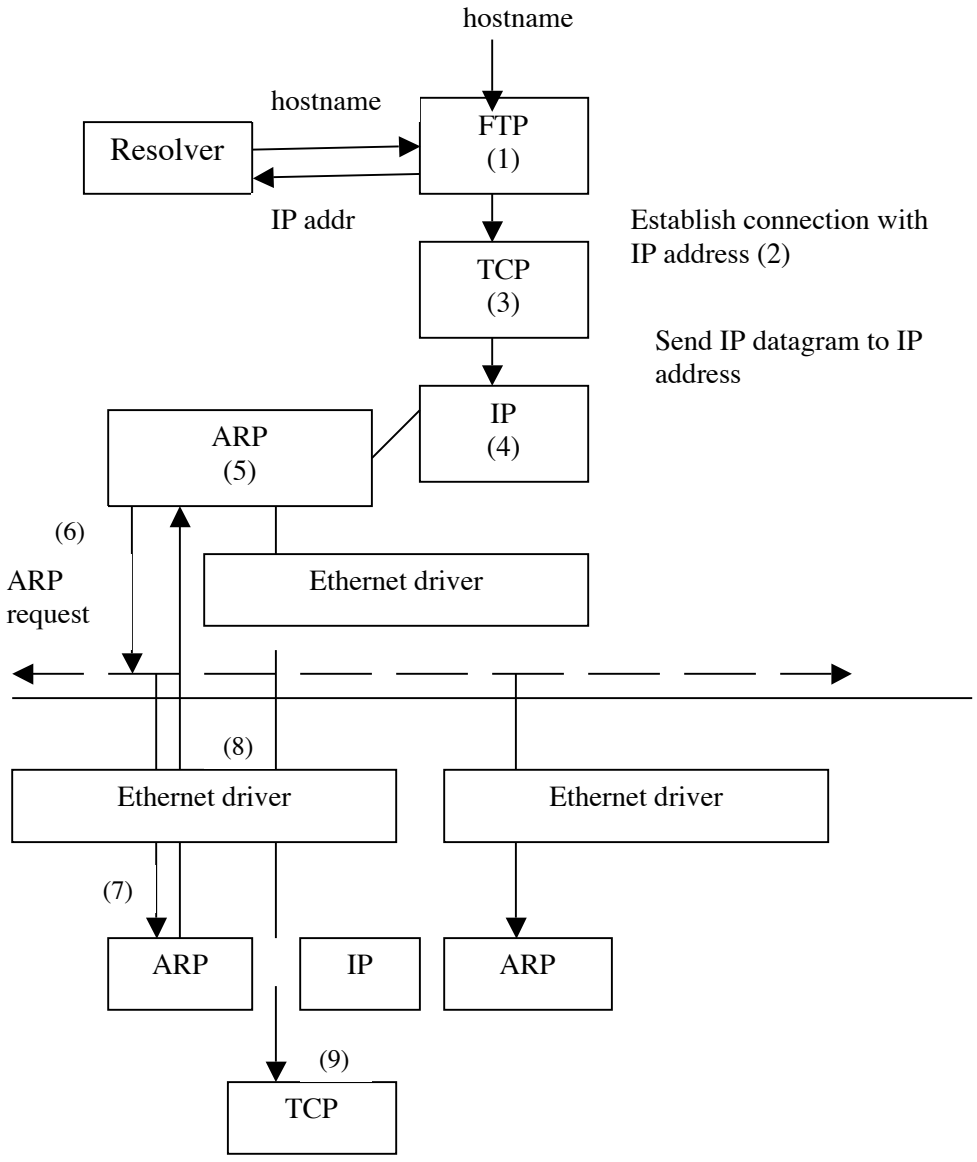

Figure 1. ARP operation after user types ftp hostname

2.3 ARP Structures

The figure below shows the format of an ARP frame when used on an Ethernet [5].

Ethernet Dest Address	Ethernet Source address	Frame type	Hard type	Prot type	Hard size	Prot size	op	Sender Ethernet address	Sender IP address	Target Ethernet address	Target IP address

Figure 2. ARP Frame on Ethernet

The Ethernet destination address, Ethernet source address and frame type define the 14-byte header called the *Ether_header*. The next five fields form the *arphdr*, which is common to ARP requests and replies irrespective of the type of media, such as Ethernet. The *ether_arp* structure constitutes the *arphdr*, the sender addresses and the target addresses when ARP is used on Ethernet. These addresses include the Ethernet address as well as the IP address.

The structure for *arphdr* and *ether_arp*, as given in [9]:

```
struct arphdr {
        u_short ar_hrd;          /*format of hardware address */
        u_short ar_pro;          /*format of protocol address  */
        u_short ar_hln;          /*length of hardware address */
        u_short ar_pln;          /*length of protocol address */
        u_short ar_op;           /* ARP/RARP operation*/
};
```

```
struct ether_arp {
        struct arp_hdr ea_hdr ;
        u_char arp_sha[6];              /* sender hardware address*/
        u_char arp_spa[6];              /* sender protocol address */
        u_char arp_tha[6];              /* target hardware address */
        u_char arp_tpa[6];              /* target protocol address */
};

struct llinfo_arp {
        struct llinfo_arp *la_next;
        struct llinfo_arp *la_prev;
        struct rtentry *la_rt;
        struct mbuf *la_hold;
        long la_asked;
};
```

A *llinfo_arp* structure exists for each ARP entry. In addition, one of these *llinfo_arp* structures is allocated as a global structure and this is used as the head of a doubly linked list. Since this structure is the only structure that has ARP entries for Ethernet to IP address correspondence, it is often referred as the ARP Cache in BSD systems.

In the *llinfo_arp* structure, *la_next* and *la_prev* form the doubly linked list and *la_rt* points to the corresponding routing table entry.

When ARP receives an IP packet to be sent to another host on the LAN and the destination host's MAC address is not in its ARP cache, an ARP request is broadcast. A reply needs to be received before the IP packet can be sent. The message is stored in an *mbuf* datagram in the BSD system (*sk_buff* in Linux) and its pointer is stored in the *la_hold* field. When the ARP reply is received, if the pointer pointed to by *la_hold* holds any packets, they are sent.

La_asked keeps a count of number of times an ARP request has been sent to the IP address and not received an ARP reply for the same. There is an upper limit to this value, and when this limit is reached, the destination host is considered down and another request will not be sent for a default time of 20 seconds. This time is defined by the value of the *arpt_down* variable.

ARP is a request-response protocol and does not have a state. An ARP request that contains the source IP address, source Ethernet address, and the target IP address is broadcast on the LAN. All hosts on the LAN receive this frame and check the target IP address against their own IP address. If the two addresses match, the respective host then sends an ARP response with its own Ethernet Address. This response is unicast, in that it is addressed only to the sender of the request.

2.4 ARP Cache Poisoning

"Address Resolution Protocol cache poisoning is the act, by a malicious host on the LAN, of introducing a spurious IP-to-Ethernet address mapping in another host's ARP cache" [1]. The result of ARP cache poisoning is that the IP traffic intended for one host is diverted to a different host.

There are various ways in which a host's ARP cache can be poisoned [1].

2.4.1 Scenario one: Broadcast Request

Explanation

Depending on the nature of the ARP requests received, ARP caches these entries. Thus, if host A sends out a broadcast request for host B, it is possible that a host C might cache host A's IP-to-Ethernet address mapping. Hence, an attacker can easily pretend to send a valid request causing ARP cache poisoning of various hosts.

The example below demonstrates this scenario using three Linux hosts.

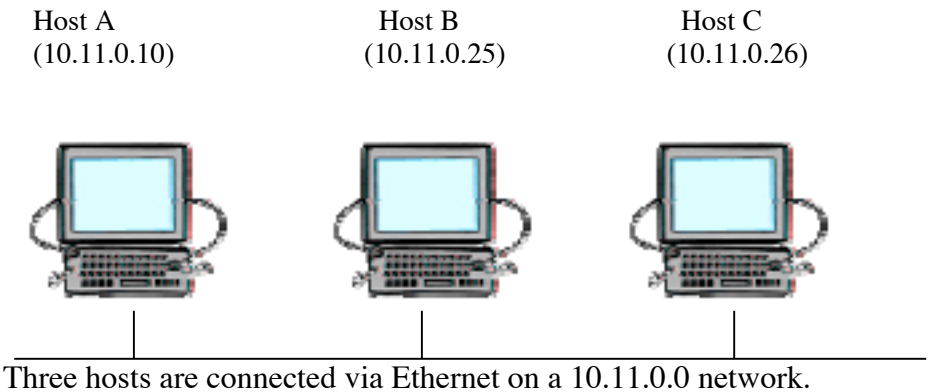

Figure 3. Broadcast Request scenario

The figure below shows host A's ARP cache assuming host A has communicated with host C.

```
[root@localhost /]# arp

Address     Hwtype      Hwaddress              Flags Mask        Iface
10.11.0.26 ether       00:03:93:5A:74:FC      C                 eth0
```

To cause ARP cache poisoning, host B's IP address is changed to 10.11.0.26 resulting in two hosts on the LAN with the same IP address. Host B then sends a broadcast message to all the hosts on the network. This IP-to-Ethernet address mapping is cached on host A and causes ARP cache poisoning. The figure below shows the new IP-to-Ethernet address mapping in host A's ARP cache.

```
[root@localhost /]# arp

Address     Hwtype      Hwaddress              Flags Mask        Iface
10.11.0.26 ether       00:30:65:D5:99:6E      C                 eth0
```

Result

When a malicious host uses another host's IP address and sends out a broadcast request, Linux caches the new IP-to-Ethernet address mapping, thus causing ARP Cache poisoning.

2.4.2 Scenario: Multiple Responses

Explanation

In this scenario, the malicious user waits for an ARP request and then sends out a specious response to that request. Even if another legitimate user responds to the request, there could be a race condition that the hacker might win.

The example below demonstrates this scenario using three Linux hosts.

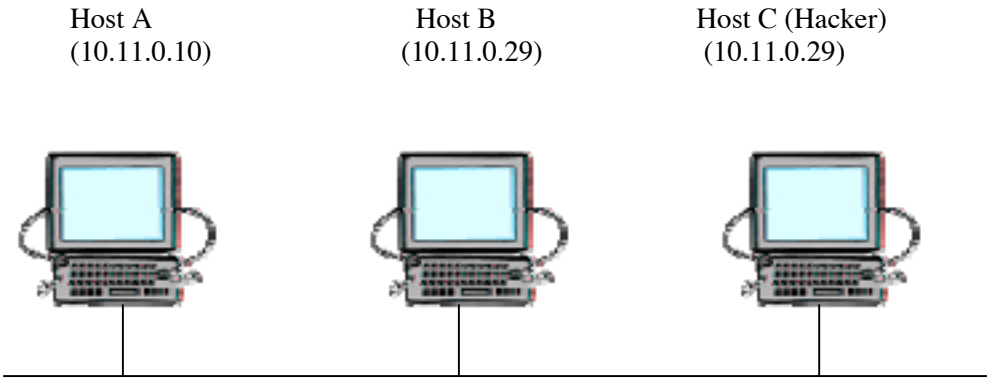

Figure 4. Multiple Responses scenario

As shown in the figure, two hosts on the network have the same IP address. If host A tries to communicate with host B and it sends out an ARP request for the IP address 10.11.0.29, both host B and host C will send out an ARP response and host C could win the race condition. The figure below shows host A's ARP cache with the malicious host C's IP-to-Ethernet address mapping.

```
[root@localhost /]# arp

Address     Hwtype     Hwaddress              Flags Mask        Iface
10.11.0.29 ether      00:03:93:5A:74:FC      C                 eth0
```

Result

ARP cache could be poisoned when multiple ARP responses are received, as there is a race condition that the hacker might win. In this case, hacker's IP-to-Ethernet address mapping is cached by the victim's host causing its ARP cache to be poisoned.

2.4.3 Scenario: Unsolicited Response

Explanation

Hosts do not keep track of the requests that they send out. Hence, a response that is not associated with any request sent out by a host will be accepted and processed. If a malicious host sends out an ARP response packet on the LAN with spurious mapping, it could poison the ARP cache of the victim. If this response is broadcast, it could poison the ARP cache of every host on the LAN.

Result

Since ARP is a stateless protocol, it does not keep track of outgoing requests and incoming responses. Hence, unsolicited responses are processed and can cause ARP cache poisoning.

3. Linux Kernel Specifics

This section discusses the various Linux Kernel specifics that are required for the implementation of this project.

3.1 The Routing Tables

There are three main tables that take part in the routing process in the Linux kernel:

The Neighbor table

This table contains information about hosts that are connected to this host physically on the local network. Initially, when a host comes up on the network, the neighbor table does not have any entries, as it has not passed any network traffic. Entries in this table are not persistent and depend on the host's communication with other computers on the network. Entries are added when needed, and deleted after the time expires for each entry. This time is defined by *retrans_time* and has a default value of 100 sec. It is possible to set up permanent entries in the neighbor table using the "*arp*" command.

The Linux operating system uses ARP for the maintenance of the Neighbor table. Hence the Neighbor table is the focus of this project.

Routing cache

This routing table is the most critical routing table in the Linux kernel. This table stores the most recently used routing entries and uses fast hash lookup. The kernel first consults this table with the source IP address, destination IP address, and Type of Service to find a

matching routing entry, and if successful, the IP packet is forwarded. This routing entry contains information like source IP address, destination IP address, device to be used to send the packet, a pointer to the neighbor table for the next link to this route and pointers to other relevant networking structures.

Forwarding Information Base (FIB)

This table contains routing information needed to reach any valid IP address on the network. When an IP packet needs to be sent to a host outside the local network, the routing cache is first checked for a matching entry with the source, destination and type of service. If an entry is found, it is used, otherwise the FIB is consulted for the route information. Even though the FIB is slower than the routing cache, it is complete. This new entry is then added in the Routing Cache.

3.1.1 Neighbor Table

As mentioned previously this table contains entries of a host's directly connected computers via the Ethernet. At any given time, this table could be empty or have entries of all the directly connected hosts, depending on its communication with the rest of the hosts. Figure 5 shows the neighbor table data structure and its relationships. **neigh_tables* is a global variable that can be used to access the list of neighbor tables. Each of these neighbor tables contains information like queue sizes, pointers to device functions, and device pointers.

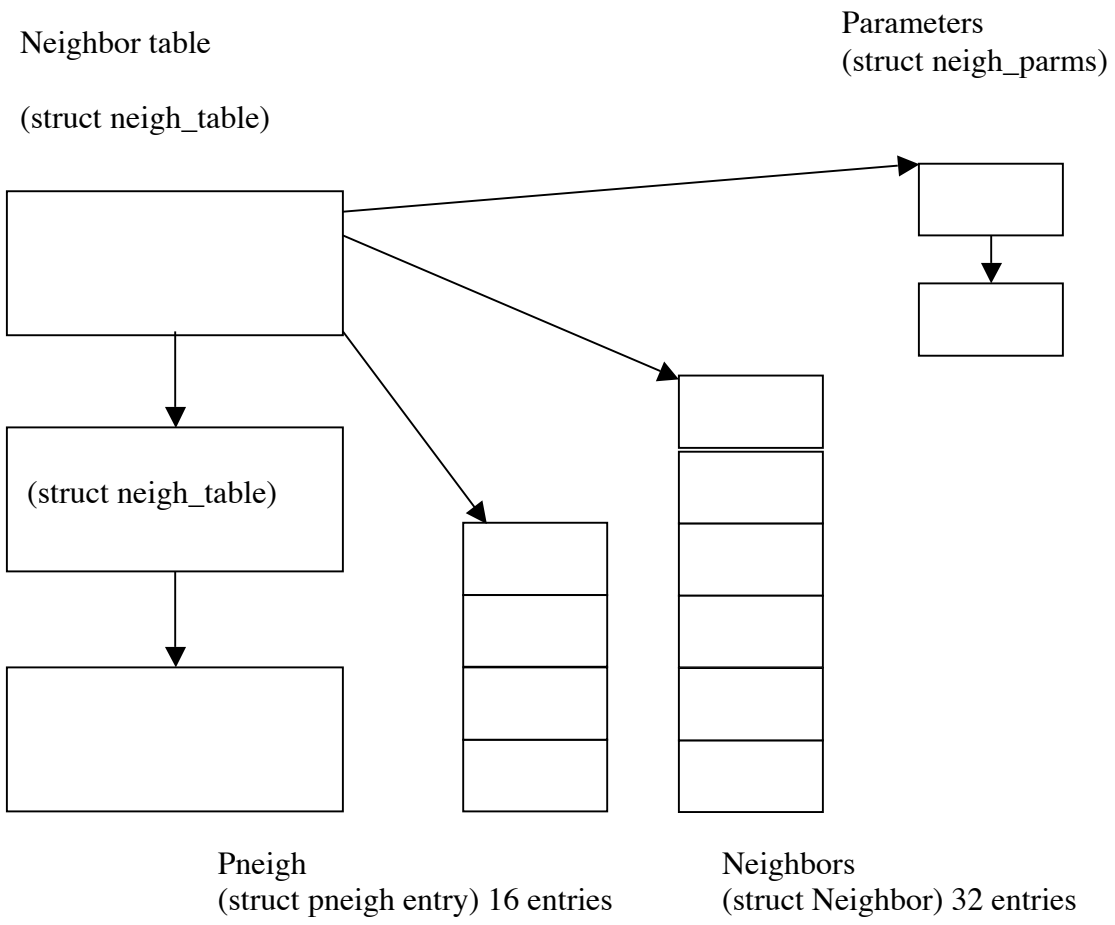

Figure 5. Neighbor Table structure

*struct neigh_table *neigh_tables*

This is a pointer to a list of the neighbor tables. Each of these tables contains specific information about a set of neighbors.

struct neigh_table

This is a low level detailed structure that contains information on device pointers, pointer to device functions, and various queues. All devices connected using the same interface will be in the same *neigh_table*. The following fields or structures are important for our implementation:

- Struct *neigh_table* *next: pointer to the next table in the list.
- Struct Neighbor *hash_buckets[]: hash table of various neighbors that are associated with this *neigh_table*.

Struct Neighbor

This data structure is used for each neighbor. It contains detailed information for each of a host's neighbors. The following fields are important for our implementation.

- Struct device *dev: device that is used to connect to the neighbor.
- __u8 nud_state: This status flag can take values like reachable, stale, incomplete, and such, and is used to determine the status of the route. Thus, this flag can be used to purge a route by marking it as an invalid route.
- struct sk_buff_head arp_queue: pointer to ARP packets for this neighbor.

3.1.2 Routing Cache

The routing cache holds entries of every route that is either currently in use or has been used very recently. When an IP packet is to be sent, this table is first searched for the corresponding route. The IP packet is sent if a match is found in the routing cache, otherwise the FIB is searched and a new entry is created in the routing cache.

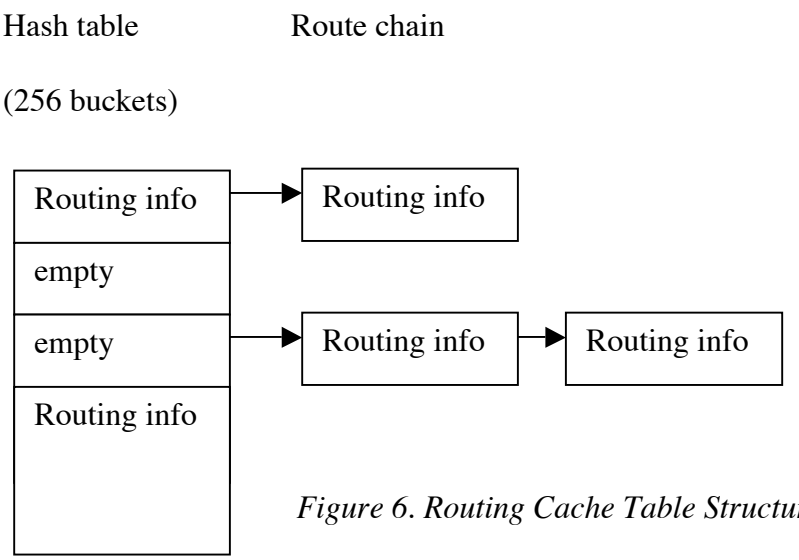

Figure 6. Routing Cache Table Structure

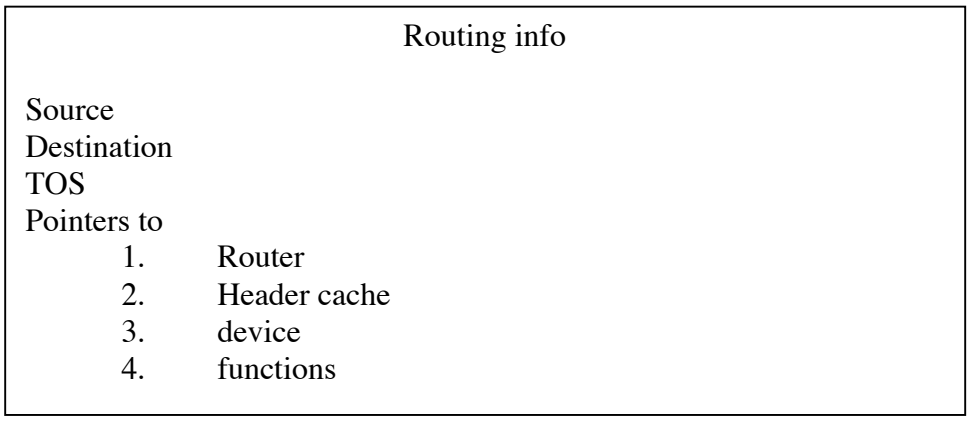

Figure 7. Routing Info Structure [10]

*struct rtable *rt_hash_table[RT_HASH_DIVISOR]*

This is a global variable and contains 256 buckets of pointers pointing to chains of routing cache entries. The kernel uses the source address, destination address, and type of service (TOS) to compute a hash that is used to get an entry point to the table.

Struct rtable

This structure contains cache entries for various destinations.

- __u32 rt_dst : destination address
- __u32 rt_src: source address
- rt_int iif : input interface
- __u32 rt_gateway : the address of the host to route through to reach the required destination.

Dst_entry

This structure contains the destination cache entry.

- struct neighbor *neighbor: a pointer to the next neighbor in this route.
- Pmtu: maximum size of the packet for this route.
- struct device *dev: the input/output device to be used for this route.
- Int (*input) (struct sk_buff*): a pointer to the input function to be called for this route. This is *tcp_rcv()* in most cases.
- Int (*output) (struct sk_buff *): a pointer to the output function to be used for this route. This is *dev_queue_xmit()* in most cases.

Traversal example using Routing cache

In this example, a host on network has an IP address of 178.18.1.2 and wants to communicate with 178.18.1.1.

- *ip_route_ouput()* is called to find the route to this host. This function then calls *rt_hash_code()* with source address, destination address, and type of service as its parameters.
- *rt_hash_code()* computes a hash function using the above three parameters and the result is used to find an index to the hash table.
- The result is used to index the routing cache hash table. The entry at this index is checked to see if it matches the required destination address, source address and type of service.
- If a match is found, then a pointer to this route is returned and statistics of this route are updated in *dst_cache* structure.
- If a match is not found, the next entry is compared (next entry is found using *u.rt_next).*
- If a match is not found in the routing cache, *ip_route_input()* returns and the calling function calls *ip_route_input_slow()* to get the entry from FIB.

3.1.3 Forwarding Information Base

The FIB table has routing information needed to reach any host with a valid IP address and a mask. Thus, it is also the most important routing table. When a host needs to communicate with another host, a search is first made in the routing cache for a route to the required destination. If an entry is not found, IP then searches in the FIB for a corresponding match. When a match is found in FIB, this route is copied in the routing cache and the packet is sent on its way.

Traversal example using Forwarding Information Base

IP tries to find a matching entry in the FIB only if a match has not been found in the routing cache.

- *ip_route_output_slow()* is called by IP as route for the required destination is not present in the routing cache. *rt_key* structure is formed with source IP address, destination IP address, and Type of Service with a value of 2.
- *ip_route_output_slow()* takes the *rt_key* structure and calls *fib_lookup()*.
- *fib_lookup()* makes the local table find the key.
- *fn_hash_lookup()* looks in the local table's hash, starting at the most specific zone, i.e zone 24.
- *fz_key()* builds a test key by performing an AND on the zone mask and destination address, giving a certain key value.
- *fz_chain()* is used next to perform a hash into the zone's hash table of various nodes. If this node is not empty, node's key and search key are compared. If there is a match, *fib_result* structure is filled with the needed route information.

- If a match is not found, process repeats by calling the *fz_key()* with a new zone, and *fz_chain()* to perform a hash until an exact match is found.
- *ip_route_output_slow()* takes this *fib_result* and creates a new routing cache entry.

3.2 SK_BUFF

SK_Buff is one of the most important packet data structures used in Linux. As the data is passed between various protocol layers, time is not wasted in copying parameters and payloads back and forth between the protocols. Data is copied only twice, once from the user space to the kernel space and then from the kernel space to an outbound medium.

The structure below shows the *sk_buff* packet structure:

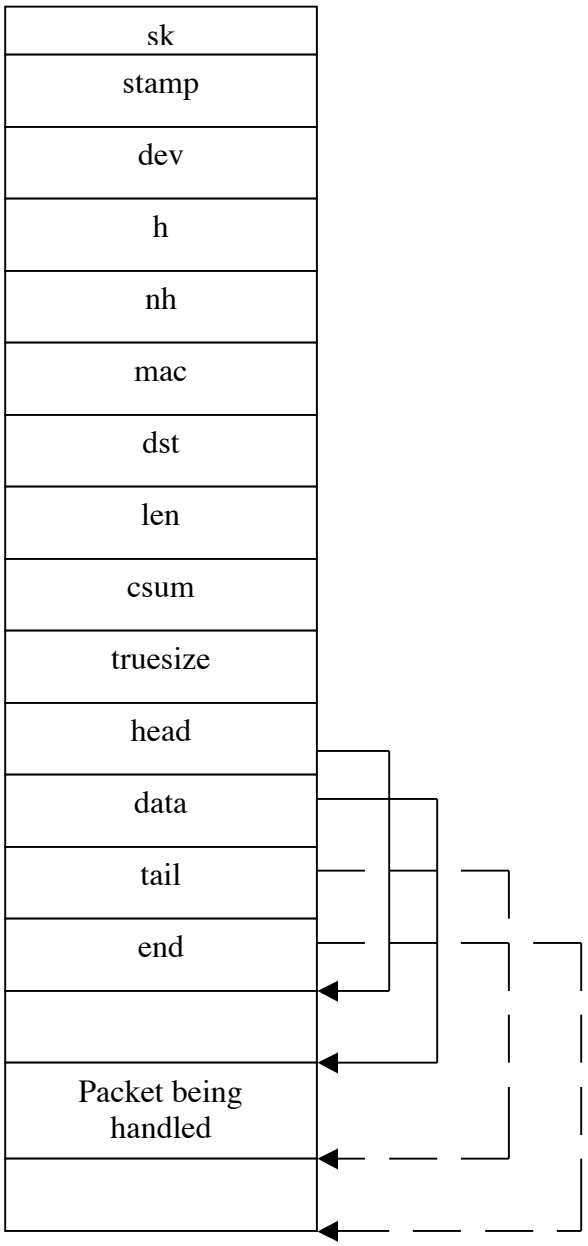

Figure 8. Sk_buff structure

Description of the various fields follows:

sk: pointer to the socket owning this packet

stamp: time this packet arrived

h: transport layer header pointer

nh: network layer header

mac: pointer to link layer header

dst: pointer to dst_entry

len: actual data length

csum: checksum of the packet

next: pointer to the next sk_buff{}

prev: pointer to the previous sk_buff{}

dev: dev currently being used

data: pointer to the start of data.

tail: pointer to end of protocol data.

end: pointer to end of the buffer holding this packet.

3.3 ARP Functions in Linux

The following are the functions in the Linux Source Code that are essential for ARP:

3.3.1 arp_rcv () [10,14]

Defined in: net/ipv4/arp.c

Parameters: Struct sk_buff *skb, Struct net_device * dev, struct packet_type *pk

The ARP handler calls this function when an ARP packet is received. This method is responsible for updating the ARP cache/neighbor table if it receives an ARP response with a valid IP and MAC address mapping. If an ARP request is received and the host's IP address matches the target IP address in the ARP packet, a response is sent back.

The following actions are taken by this function:

1. Performs error checking for non-ARP devices, and verifies if the packet is for this host.
2. Checks if the operation in the ARP packet is Reply/Request type.
3. Extracts the data from the skbuffer packet.
4. If the address is a loopback/multicast address, then the request is considered to be a bad request.
5. If the received message is a request and *ip_route_input()* is true, then
    a. If it is a local packet
        i. Calls *neigh_event_ns()* to look up and update the neighbor that sent the packet.

ii. Reply is sent with the device address (MAC address)

Else:

i. *Neigh_event_ns()* is called to look up and modify the neighbor that sent the packet..

ii. *Neigh_release()* is called.

iii. If required, *arp_send()* is called with the address

iv. Otherwise, *pneigh_enqueue()* is called and returns 0

6. If the received message is a reply:
   a. *__neigh_lookup()* is called.
   b. A check is done to see if multiple ARP replies have been received, if so, only the first one is kept
   c. *neigh_update()* is called and ARP entry is updated
7. skbuffer packet is freed.
8. return from the function.

3.3.2 arp_send () [10,14]

Defined in: net/ipv4/arp.c

Parameters: int type, int ptype, u32 dest_ip,

struct net_device *dev, u32 src_ip,

unsigned char *dest_hw, unsigned char *src_hw,

unsigned char *target_hw.

This method is responsible for creating a new skbuffer packet and filling it with all the ARP information it receives. It checks if the device that is to be used to send out the ARP

packet supports ARP. The function then calls *dev_queue_xmit()* with this skbuffer to send the ARP packet. The function executes the following steps:

1. The function first checks to see if the device, that is supposed to be used to send the ARP packet, supports ARP.
2. A new skbuffer is allocated.
3. All the buffer header information is filled.
4. All the ARP information, such as the source MAC address, source IP address, and message type like Request/Reply, is filled.
5. *dev_queue_xmit()* is called to send the ARP packet with the filled skbuffer packet.

3.3.3 arp_req_get () [10,14]

Defined in: net/ipv4/arp.c

Parameters: struct arpreq *r, struct net_device *dev

This function is used to look up in the ARP table for a match with the given IP address. The result of the function indicates whether or not a match is found. The function executes the following steps:

1. Extracts the IP Address from the *arpreq* structure.
2. Calls *__neigh_lookup()* to find an entry for the given IP address.
3. Copies data from neighbor entry to *arpreq* entry.
4. Returns 0 if an entry is found in the ARP table, or ENXIO if not.

3.3.4 ip_route_input() [10,14]

Defined in: net/ipv4/route.c

Parameters: struct sk_buff *skb, u32 daddr, u32 saddr, u8 tos, struct net_device *dev

This function is used by methods like *arp_rcv()* to lookup a routing entry in the routing hash table. It takes the source address, destination address, and type of service as its parameters and computes a hash value. A search is made in the hash table for a matching entry and a routing entry is returned if a match is found. The following are the steps executed by this function:

1. Hash value for the address is calculated.
2. Searches through the hash table to see if an entry is found with (source address, destination address, Type Of Service and IIF/OIF)
3. If a matching entry is found, routing entry is returned and the various stats are updated.
4. Calls *ip_route_input_slow()* if no match is found.

3.3.5 ip_route_input_slow() [10,14]

Defined in: net/ipv4/route.c

Parameters: struct sk_buff *skb, u32 daddr, u32 saddr,

u8 tos, struct net_device *dev

This function is called when a routing entry is not found in the fast routing cache. It is called with the same parameters as that of *ip_route_input()*. *Fib_lookup()* is called to look up in the slow, but complete, FIB table for the routing entry. This entry from the FIB is

used to create an entry in the routing cache. The following steps are executed by this function:

1. Creates a cache key for routing table.
2. Does error checking for addresses like broadcast and loopback.
3. Calls *fib_lookup()* to find the corresponding route.
4. Creates a new routing table entry and initializes it with information such as source address, destination address, TOS, and various flags.
5. Validates the packet source, and returns from the function if the source is bad.
6. Calls *rt_set_nexthop()* to find the next destination
7. Calls *rt_intern_hash()*.

3.3.6 neigh_event_ns() [10,14]

Defined in: net/core/neighbor.c

Parameters: struct neigh_table *tbl,

u8 *lladdr, void *saddr,

struct net_device *dev

Neigh_event_ns() is called to lookup an address in the neighbor table and return a pointer to this neighbor. Thus, it takes a pointer to the neighbor table as one of its parameters. A call is made to *__neigh_lookup()* to look up the needed address. If an entry is found, a pointer to this neighbor is returned and the entry is updated accordingly. The following are the steps executed by the function:

1. Calls *__neigh_lookup()* to look up the address in the neighbor table.

2. Calls *neigh_update()*, if a neighbor is found.
3. Returns a pointer to this neighbor.

3.3.7 neigh_update() [10,14]

Defined in: net/core/Neighbor.c

Parameters: struct Neighbor *neigh, const u8 *lladdr, u8 new, int override, int arp

The core function of this method is to update an entry in the neighbor table. Thus, it takes a pointer to the neighbor table as one of its parameters. A check is first made to see if the table can be modified. If the required entry is old, the neighbor's status is checked by calling *neigh_suspect()*. The address is overwritten or updated only if the override flag value is 1. This function is used to purge an entry in the neighbor table by marking it as an Invalid entry. The function executes the following steps:

1. Checks table permissions to see if it can be modified.
2. If this entry is old, neighbor status is checked by calling *neigh_suspect()*.
3. If the device needs an address and if the address has changed, the override flag is checked. If the override flag value is 1, the address is overwritten.
4. Calls *neigh_sync()* to verify that the neighbor is still up.
5. Updates the neighbor contact time.
6. Returns 0 if the old entry was valid, and new entry does not change the old address.
7. Replaces old address with new if they are different.
8. Returns 0 if the two states match.
9. Calls *neigh_suspect()* to verify the connection

a. If old state was invalid, it goes through the queued ARP packets, and calls the neighbor output function.
b. ARP queue is purged.

10. Return 0 from the function.

3.3.8 Dev_queue_xmit() [10,14]

Defined in: net/core/dev.c

Parameters: struct sk_buff *skb

This function is used to send the packet over the device. Thus, this function is used not only by ARP but also by IP to send the packets over any required device. The following steps are executed by this function:

1. Checks to see if device supports checksumming. If packet is not checksummed, completes checksumming of the packet.
2. Checks to see if the device has a queue:
   a. If it does, the packet is added to the queue
   b. Device is woken up.
3. Calls *hard_start_xmit()* if there is no queue.

3.4 ARP functions on BSD

This section gives a brief introduction to the related ARP functions in a BSD operating system. The implementation of ARP varies in different operating systems but the underlying concept is the same across various systems.

In BSD, there are nine ARP functions defined in the following files:

Net/if_arp.h: arphdr structure definition.

Netinet/if_ether.h: other structures and some constant definitions.

Netinet/if_ether.c: ARP functions.

A brief description of some of these functions is provided here.

3.4.1 arpwhohas function

This function broadcasts an ARP request. It is usually called by the ARP resolve function.

3.4.2 arprequest function

This function is called when an ARP request, which is usually broadcast, is to be sent. This function forms the request packet and then calls the interface's output function.

3.4.3 arpintr function

When the *ether_input* receives a packet that has the frame type set as *ETHERTYPE_ARP*, it schedules a software interrupt and the received frame is then appended to ARP's input queue. When the kernel receives an interrupt, *arpintr* is called to process it. This function

does the initial filtering of the packets and checks if the packet is valid and then passes it to *in_arpinput function*.

3.4.4 in_arpinput function

Each ARP request or ARP reply received is processed by the *in_arpinout* function. There are certain rules that this function follows:

1. If a request is received for an IP address for this host, a reply is sent. If the request is not broadcast, an entry is created in the ARP cache for the host.
2. If a response is received for a request sent by this host, the ARP entry is now complete. The other host's hardware address is now stored and any messages queued for this machine are sent.
3. If a host receives a request that contains the sender IP address being the same as the receiver host's IP address, then it is logged as an error.
4. If a host receives a request or a reply for which an ARP entry already exists, but the received packet contains a different hardware address, the entry is updated with a new hardware address.
5. A host can be configured as a proxy server. This means that the host would respond to requests intended for other hosts.

3.4.5 arpresolve function

This function is called to obtain the Ethernet address for an IP address.

3.4.6 arplookup function

This function calls the *rtalloc1* to look up an ARP entry in the Internet routing table. If this function succeeds, it returns a pointer to *llinfo_arp* structure that contains more information about this entry, otherwise a null pointer is returned.

1. This function is called to look up an entry with the source IP address of a received ARP packet. It also creates an entry for this IP address if an entry does not already exist.
2. It is also called to see if a proxy ARP entry exists for the destination IP address of an ARP packet that is received.
3. Called by *arpresolve* to look up or create an entry for the target IP address for a packet that is about to be sent.

4. Design Consideration

Mahesh Tripunitara and Partha Dutta [1] have proposed a solution to ARP Cache poisoning. These are the following design considerations for their solution.

- Asynchronous

  The solution does not involve polling the ARP cache every few units of time. The downside of polling the ARP cache is deciding the time interval between such checks. Polling too often burdens system performance, whereas increasing the time interval may result in missing relevant activity.

- Middleware

  The proposed solution does not access networking components of the Operating system [1]. Some modules are introduced into the existing system without any change in the existing ones.

- Compatible

  The module developed is added only to some hosts in a LAN, leading to protection of only some host's cache. All other hosts continue to run in the same way, unaware that their neighbors may be protecting their ARP cache.

These design considerations are intended for a Streams based operating system like Solaris. In a Streams based operating system it is possible to create a module and insert it into the networking system without any changes to the existing modules.

In Linux, modules can be easily created and inserted into the networking subsystem but problems arise because modules in Linux do not always have access to the kernel. Thus, the implementation for RedHat Linux is not a fully middleware approach as it involves changes to the existing kernel.

Changes implemented on a Linux host to protect its ARP cache are fully compatible with other hosts on the Ethernet as these changes are transparent to the neighbors.

4.1 Architecture

The architecture for solving the problem of ARP cache poisoning as proposed in [1] for a Streams based operating system is as shown below:

- *checker* is the module added to the operating system.

The *checker* module intercepts messages in both the downward as well the upward direction. Requests and responses each travel in both directions. Traffic moving downwards is logged when it is an ARP request. The checker module makes a decision, whether the traffic should be allowed to go upwards or not.

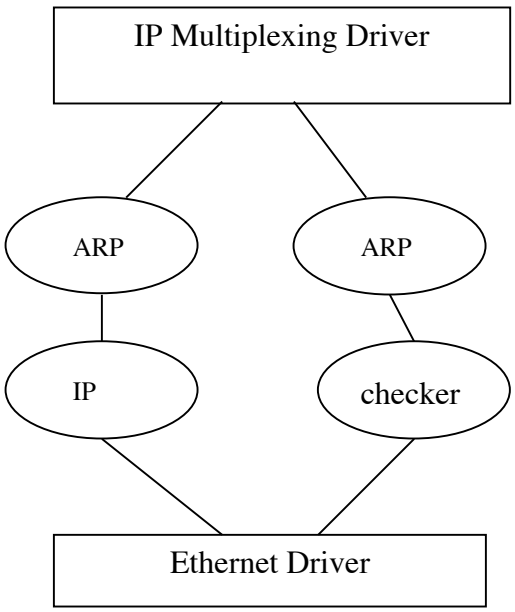

Figure 9. Architecture for solving ARP Cache poisoning [1]

The request messages moving downwards are recorded, so that when a response comes in, it can be matched with the corresponding request. These messages are stored in lists. There are two separate lists for downstream as well as upstream traffic.

This *checker* module can be loaded dynamically without having to rebuild the kernel in a Streams based operating system. In RedHat Linux, a few changes need to be made to the kernel in addition to the dynamic loading of the *checker* module.

The *checker* module implemented in Linux has two lists:

arp_checker_requested_list

This list stores all the ARP requests that are sent out from a host. Each entry is a structure that stores the IP address, MAC address (which will be null for requests), and a pointer to the next entry in the list. Thus, before an ARP request is sent out, the checker keeps a log of this entry in this list, if it does not already exist. This makes the host go in a *requested* state.

arp_checker_responded_list

When an ARP response is received, the checker module checks in the requested list for a match. If a match is found in the requested list for this IP address, an entry is added in the response list. Each entry in the response list has the same structure as the entry in the requested list. The only difference is that the MAC field in the entry holds a MAC address of the host, as received in the ARP response.

4.2 Algorithm

The algorithm that is to be executed in the Streams module as suggested in [1] is discussed below. The most significant challenge in my project deals with the modification and implementation of this algorithm for a different operating system, like Linux.

As discussed previously, the ARP handler calls the *arp_rcv()* function whenever an ARP packet is received. A hook is added in this function that calls the *arp_rcv_checker()* function in the *checker* module. The *checker* module is loaded dynamically. Thus, before the packet is processed by the *arp_rcv()* function, it first checks if the module is loaded. If so, the checker function processes the ARP packet and decides if the packet should flow up to be processed or be dropped.

Similarly, a hook is also placed in the *arp_send()* function that calls the *arp_send_checker()* function in the *checker* module. Before an ARP packet is sent onto the network, the checker function processes the skbuffer and logs information like the destination IP address and the MAC address if known.

When an ARP frame is received:

If this ARP frame is a response:

If a corresponding entry exists in the requested list:

Move the entry to the responded list and let the packet flow up

to be processed by the host's ARP implementation.

Else, a corresponding entry is not found in the requested list, and:

Check if there exists a corresponding entry in the

responded list, If yes then multiple responses have been received,

so:

Check whether the entry in the responded list

corresponding to the IP address is same as that in the

response.

If yes:

Refresh the entry in the ARP cache.

Else, the ARP cache entry is

not consistent with the received packet:

Log the incident. Drop the frame. Mark

the entry as Invalid.

Else, this is an unsolicited response.

Drop the frame and log the incident.

Else, this is a request, and:

Let the frame flow up and let the host's ARP implementation process it.

Else, a frame is being sent, and:

If the ARP frame is a response:

Let the frame flow down.

Else, this is a request, and:

Add a corresponding entry in the requested list.

Let the frame flow down.

4.3 Placement of the Code

The code discussed above could be placed in the following ways:

1. *Kernel*

   The code could be placed directly into the kernel. This has its advantages and disadvantages. The code is simpler to write, as new modules do not need to be loaded. However, the main disadvantages are that the code needs to be recompiled, reinstalled, and the system needs to be re-booted. It is also difficult to debug code in the kernel. Changes are permanent (until code is deleted and system is compiled, installed and booted again). Thus, the entire process can be very tedious and error-prone.

2. *Module*

   Modules are relatively simple to code. The process of loading modules and debugging them is less painful than making changes to the kernel. The problem is that the module needs access to the kernel that is not always available.

3. *Coding in Kernel and Module programming.*

   In this method, the user makes changes to the kernel once, compiles, installs and boots the system. Modules are then loaded making the process of testing and debugging very simple.

Considering these factors, we have used a method that involves module programming to implement the *checker* module and also some coding in the kernel to give the module access to the kernel.

5. Implementation

This section describes the implementation details of the new module and the various changes made to the kernel in order to detect and prevent ARP Cache poisoning.

5.1 Operating System Used

Linux operating system version 2.4.7 was used for the implementation of this project. Since Linux does not have Streams programming, the implementation is different on this platform as compared to Solaris.

5.2 Open Sources Utilized

Linux open source version 2.4.7 has been utilized for the implementation of the algorithm. The ARP subsystem in this kernel was slightly modified to accommodate the new kernel module.

5.3 Code placement

As discussed earlier, we decided to place the code in a module that will be loaded dynamically. However, since the module does not have access to the kernel, a few modifications have been made to the kernel. Certain symbols are exported in the kernel. The kernel is then compiled, installed and system is re-booted for changes to take effect.

5.4 Minimal Hardware requirements

To successfully design, implement, and test the algorithm for ARP Cache poisoning detection and prevention it is essential to have the following:

- One personal computer running Linux kernel 2.4.7 with an Ethernet card. This computer should have the source code installed.
- Two more personal computers with Ethernet cards.
- A hub that is used to connect the above three machines together.

5.5 Implementation Details in Linux

This section covers the implementation details of the algorithm in Linux using kernel 2.4.7.

5.5.1 Kernel Module

The kernel module that is loaded dynamically has two critical functions and a list of helper functions.

The two critical functions, described below are the *checker* functions called when an ARP packet is received or sent.

5.5.1.1 arp_send_checker() function

This function is called by the *arp_send()* function just before it sends out an ARP packet onto the network.

The function takes the following parameters:

- Type of ARP operation (request/reply)
- Destination IP address
- Device to be used for transmission
- Source IP address
- Destination MAC address
- Source MAC address

The function checks the type of ARP operation and if the operation is an ARP reply, the function returns. If the operation is an ARP request, a check is made to see if a request for this IP address exists in the requested list. An entry is added in the requested list if a match is not found.

5.5.1.2 arp_rcv_checker() function

This function is called by the *arp_rcv()* function after an ARP packet is received on a device.

The function takes the following parameters:

- Sk_buff packet, containing the ARP packet with the data as well as the header information.
- Device information on which the packet arrived.
- Packet type

The function returns an int value that is used by *arp_rcv()*, that is the calling function in the kernel to determine whether to continue processing the ARP packet or to drop it.

The function retrieves the required ARP information like the destination IP address, source IP address, destination MAC address, source MAC address, and type of ARP packet (Request/Reply). If the packet type is a request, the message is allowed to flow up to let the Linux kernel handle the request.

If the packet is an ARP reply, the method checks to see if it sent out a request for this IP address. It also decides if the response would cause ARP cache poisoning, and drops the packet if required.

5.5.2 Kernel Changes

The module does not have access to the kernel unless certain symbols get exported. Thus, the following changes need to be made in the kernel.

The following lines of code are added in */net/netsyms.c*

*extern int (*arp_rcv_checker)(struct sk_buff *, struct net_device *,struct packet_type *); ..(New)*

*extern int (*arp_send_checker)(int,int,u32, struct net_device *,u32,unsigned char *, unsigned char *, unsigned char *); ..(New)*

EXPORT_SYMBOL_NOVERS(arp_rcv_checker); …(New)

EXPORT_SYMBOL_NOVERS(arp_send_checker); ..(New)

EXPORT_SYMBOL(register_gifconf); ..(Old)

The following lines of code are added in the */core/ipv4/arp.c*

*int (*arp_rcv_checker)(struct sk_buff *, struct net_device *,struct packet_type *);*

*int (*arp_send_checker)(int,int,u32, struct net_device *,u32,unsigned char *, unsigned char *, unsigned char *);*

6. Test Cases and Results

This section provides various tests that were carried out to poison the ARP cache and the results obtained. The scenarios have been discussed before in Section 2.3.

6.1 Scenario one: Broadcast Request

Explanation

In this scenario, the hacker can forge an IP address and send a valid broadcast request that is cached by hosts on the LAN. This cached entry holds the hacker's mapping of IP and Ethernet address resulting in victim's ARP cache poisoning.

The example below demonstrates this scenario using three Linux hosts.

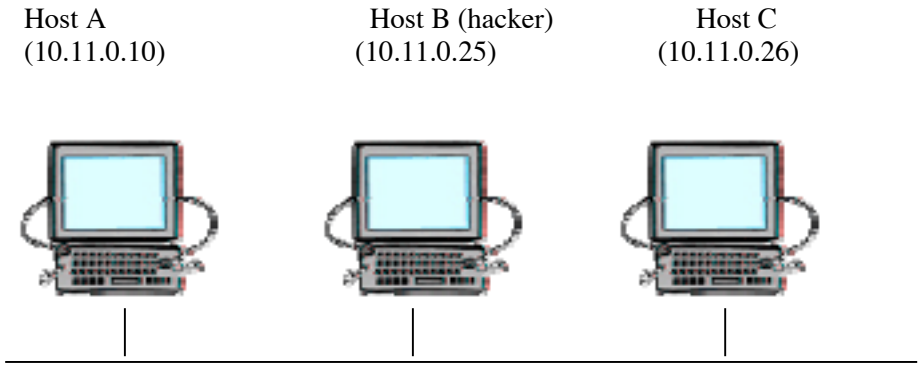

Figure 10. Broadcast Request scenario

Three hosts are connected via Ethernet on a 10.11.0.0 network. The figure below shows host A's ARP cache assuming host A has communicated with host C.

```
[root@localhost /]# arp

Address     Hwtype     Hwaddress             Flags Mask        Iface
10.11.0.26 ether      00:03:93:5A:74:FC     C                 eth0
```

To cause ARP poisoning, host B's IP address is changed to 10.11.0.26 resulting in two hosts with the same IP address. Host B then sends a broadcast message to all the hosts on the network. This new IP-to-Ethernet address mapping is cached on host A and causes ARP cache poisoning.

```
[root@localhost /]# arp

Address     Hwtype     Hwaddress             Flags Mask        Iface
10.11.0.26 ether      00:30:65:D5:99:6E     C                 eth0
```

Result

When a malicious host sends out a broadcast request, Linux caches this entry thus causing ARP Cache poisoning. This entry is not deleted from the ARP cache and is logged in */var/log/messages*. The reason for this is that this could be a genuine change in the network. Detecting the change and logging in the messages leaves it to the administrator to decide between a genuine change and an ARP cache poisoning.

6.2 Scenario: Multiple Responses

Explanation

In this scenario, the malicious user polls the Ethernet for an ARP request and then sends out a specious response to that request. Even if another legitimate response is received, there could be a race condition that the hacker might win.

The example below demonstrates this scenario using three Linux hosts.

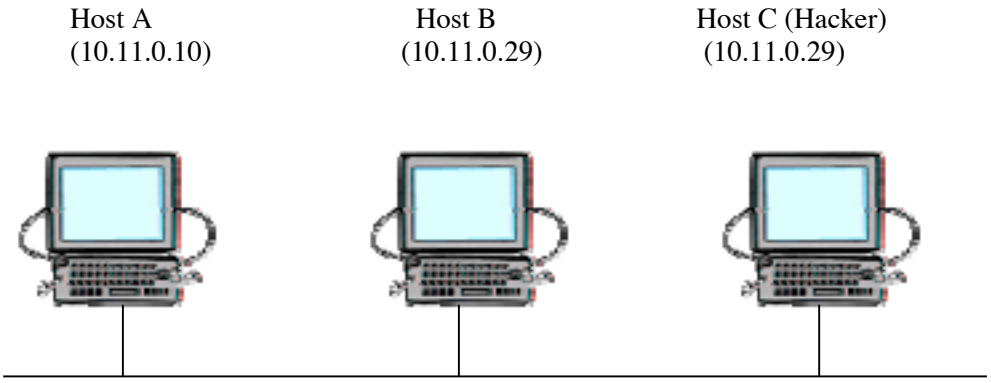

Figure 11. Multiple responses scenario

As shown in the figure, two hosts on the network have the same IP address. If host A tries to communicate with host B and it sends out an ARP request for the IP address 10.11.0.29, both host B and host C will send out an ARP response and host C could win the race condition.

The *checker* module maintains two lists: *arp_checker_requested_list* and *arp_checker_responded_list* to keep track of outgoing ARP requests and incoming ARP responses. The *arp_rcv_checker()* function discards the skbuffer when multiple responses are received and the ARP entry is marked as a *Failed* entry in the neighbor table. This avoids ARP cache poisoning on the host.

Thus, after the *checker* module is loaded, host A has an entry for host C that is marked *Failed*.

Result

ARP cache could be poisoned when multiple ARP responses are received, as there is a race condition that the hacker might win.

The algorithm maintains lists for ARP requests and responses. Thus, if there are multiple responses to a request, it is logged as a warning and entry is marked as *Failed* in the neighbor table.

6.3 Scenario: Unsolicited Response

Explanation

ARP is a stateless protocol. Thus, the protocol does not keep track of the requests it sends out. Hence, an unsolicited response with spurious mapping sent out by a malicious host could poison the ARP cache of the victim.

The *checker* module maintains a requested list for all the ARP requests that it has sent out. Thus, whenever an ARP response is received, the requested list is consulted to see if the ARP response should be processed. If an entry is not found in the requested list, it is an unsolicited response and is discarded.

Result

The *checker* function uses the requested list to find a corresponding entry for each response before it can flow up to be processed by *arp_rcv()*. If a match is not found, the packet is discarded.

Thus, all unsolicited responses are discarded and this avoids ARP cache poisoning on a host.

7. Conclusion

This section briefly discusses the challenges involved with this project, a few limitations and future work.

7.1 Innovations and Challenges

This project involved many complications and challenges. A few of them are highlighted here.

- An existing algorithm designed for use in a Streams based system was modified and implemented for ARP cache poisoning detection and prevention in Linux.
- Implementation of the modules at the system level is very complex.
- Implementation requires a thorough understanding of the networking system in Linux since the solution is very operating system dependent.

7.2 Limitations

There were a series of restrictions that came about in this project. A few of them are highlighted below.

- As discussed previously, the implementation of ARP varies on different operating systems. Since the module in this design is specific to the kernel, it is not portable, thus restricting its use.
- The algorithm is generic but not the implementation.

- Since the code is written in a module, it opens a security hole in the operating system. A user with root access can modify the module causing unpredictable results.

7.3 Future Work

This project deals with the changes to an existing algorithm for ARP Cache poisoning prevention and detection for a host running Linux. This technique can be used in LANs to protect important hosts.

The project could be extended in the following ways in the future:

- The solution provided for a host running Linux does not have a way of purging old entries from the two lists. Since memory is important at the kernel level a timer could be used for the entries in the list to delete them in a timely fashion.

- As mentioned previously, even though the algorithm requires very few changes to be ported onto a different operating system, the implementation varies. Thus, porting it to various operating systems could be done in the future.

- The solution does not provide any protection for ARP cache entries. Cryptography could be used for storing these entries in the ARP cache.

8. References

[1] T. Mahesh. Middleware Approach to Asynchronous and Backward Compatible Detection and Prevention of ARP Cache Poisoning. *http://www.acsac.org/1999/papers/fri-b-0830-dutta.pdf* , August 1999.

[2] J. Postel. Internet protocol. *RFC 791*, September 1981.

[3] J. Postel. Transmission datagram protocol. *RFC 793*, September 1981.

[4] S. A. Rago. *UNIX System V ProgrammingGuide*. Addison–Wesley Professional Computing Series, July 1993.

[5] R. W. Stevens. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison–Wesley Professional Computing Series, January 1994.

[6] SunMicrosystems. *STREAMS Programming Guide*. Solaris 2.6 AnswerBook Library.

[7] SunMicrosystems. *Manual Pages for Solaris 2.6*. 1994.

[8] Y. Volobuev. Playing redir games with ARP and ICMP.*The BUGTRAQ mailing list, http://www.goth.net/ iceburg/tcp/arp.games.html*, September 1997.

[9] R. W. Stevens. *TCP/IP Illustrated, Volume 2: The Implementation.* Addison–Wesley Professional Computing Series, January 1994.

[10] Glenn Herrin. Linux IP Networking. A guide to the Implementation and modification of the Linux Protocol Stack. *http://kernelnewbies.org/documents/ipnetworking/linuxipnetworking.html, May 2000*

[11] Wei Ye. Compile Linux Kernel. *http://www.isi.edu/~weiye/system/guide/kernel.html Jan 2002*

[12] Brown Martin, Guide to IP Layer Network Administration with Linux.

http://linux-ip.net/html/

[13] Fairhurst Gorry, Address Resolution Protocol

http://www.erg.abdn.ac.uk/users/gorry/course/inet-pages/arp.html Jan 2001

[14] Linux Source code

http://www.kernel.org/pub/linux/kernel/v2.4/ Jul 2001