Distributed Authentication Methods for ARP Cache Poisoning Mitigation

Omar Chedid

November 22, 2019

Abstract - **The address resolution protocol (ARP) is a network communication protocol used to map network layer addresses to link layer addresses. Under normal circumstances, the protocol has proven to function efficiently. However, the protocol is not designed to cope with malicious hosts that can take advantage of the lack of authentication to launch identity impersonation attacks. Multiple methods have been proposed to detect, mitigate, and prevent such attacks, but each has its own limitations. In this paper, we propose a distributed majority voting scheme in which peers on the local area network contribute to the detection of ARP cache poisoning by sharing caches. The new protocol effectively prevents cache poisoning at the cost of minimal overhead traffic and application layer blocking time.**

Keywords: **ARP Spoofing, ARP Cache Poisoning, MITM, DOS, Majority Voting, Distributed Systems**

I. Introduction

The address resolution protocol [10] (RFC 826) is a network communication protocol used to map network layer addresses (IP) to link layer addresses (MAC). The protocol relies on requests and replies communicated between hosts belonging to the same subnet and is never communicated across networks; hence making it a layer two protocol. The process of resolving the IP address mapping of a given host begins by broadcasting an ARP request on to the network. This request queries all other hosts for the MAC address of the host being resolved. If a host on that broadcast domain is assigned an IP address that matches the IP address being queried, then it will reply with its MAC address. Once a reply is received by any given host, the IP to MAC mapping is stored in that hosts local ARP cache. A major flaw with the ARP protocol is that it lacks any form of authentication. Combined with the fact that the protocol is stateless implies that hosts will accept any ARP reply even if they did not send out a request. Adversaries can take advantage of this vulnerability to poison a given targets ARP cache. As a result of simple cache manipulation, this vulnerability can lead to attacks such as identity impersonation, denial of service, and man-in-the-middle relays.

The problem of ARP cache poisoning roots back to the fundamental security principle of authentication. The absence of an authentication mechanism makes it extremely difficult to verify legitimate traffic and detect malicious behavior on the network. Several schemes have been proposed to mitigate cache poisoning but each has its own limitations. The trivial solution is to set static ARP entries manually for each cache. This approach eliminates the possibility of cache poisoning all together, however, it does not scale well for large networks that are mobile and dynamically changing over time. Other approaches introduce mechanisms on top of ARP which add a whole new level of complexity to an inherently fast, simple, and stateless protocol. These mechanisms include public key cryptography, centralized servers, and dynamic intrusion detection systems. While all these methods help mitigate cache poisoning to some extent, they are not practically deployable due to their computational complexity, centralized design, and administrative cost.

In this paper, we attack the problem at hand through a distributed computing approach in which we rely on a voting mechanism to detect the possibility of cache poisoning. Our proposed protocol adopts the traditional ARP protocol and combines it with an end host algorithm that statistically corrects malicious cache entries. Overall, we achieve authentication and improve the chances of detecting and correcting malicious cache entries. Since our approach relies on statistical analysis, detection accuracy is affected by the number of hosts active in the network. The rest of the paper is organized as follows. Section II provides the motivation behind this paper by providing examples of the attacks that can be carried out with cache poisoning. Section III highlights important contributions achieved in previous related works. Section IV goes in depth into the proposed algorithm and its components. Finally, section V evaluates the results collected, and section VI concludes the paper.

II. ARP Cache Poisoning Attacks

ARP cache poisoning is a vulnerability exploited by an attacker to modify the cache table of a target. The goal of the exploit is to place the attacker in a beneficial position on the network that allows him to sniff, drop, or forge data. The following are some of the attacks that take advantage of ARP cache poisoning. In all cases, the hosts with MAC addresses A and B are the targets, and the host with MAC address C is the attacker:

Identity Impersonation

In Figure 1, the attacker sends out a fabricated unicast ARP reply message to the target with MAC address A claiming to own the MAC address B. Upon updating its cache, the host with MAC address A thinks that it is communicating with the host with MAC address B, when in fact all communication is done with the adversary. In general, this type of attack is easily detected because once a session is hijacked, it is very difficult for the adversary to fabricate the network behavior of the host it is impersonating.

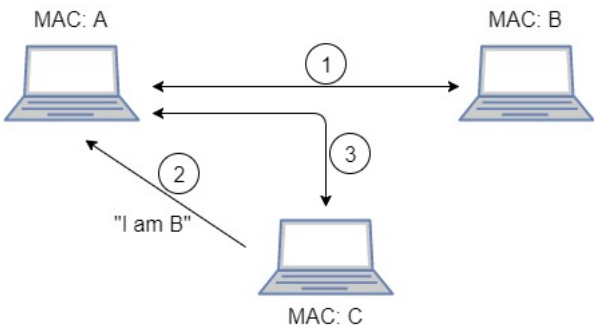

Figure 1: Identity impersonation attack

Man-In-The-Middle

A more sophisticated version of the identity impersonation attack gives the attacker access to a targets traffic without having to interact with it directly. In a man-in-the-middle attack, the adversary convinces his target that he is the gateway while convincing the gateway that he is the target. By enabling IP forwarding on his machine, all of the targets traffic flows through the attacker undetected. Assuming traffic is not encrypted, an adversary will have full access to the plain text passing through his network interface card (NIC). Figure 2 provides a detailed visual of this attack.

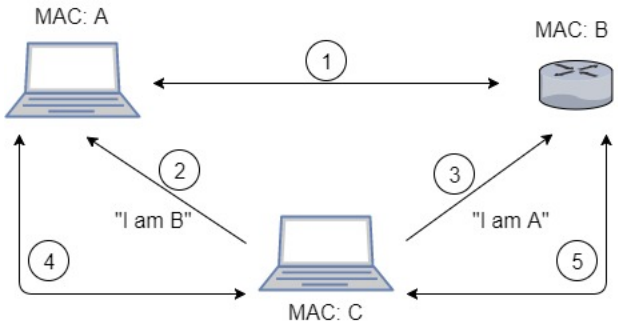

Figure 2: MITM attack

Denial of Service

In this attack, the adversary can prevent the target from leaving the network by continuously sending out ARP reply messages claiming to be the gateway. This will force all of the targets traffic to be redirected towards the attacker who then simply drops all packets. This process effectively denies the target from accessing the Internet as seen in Figure 3.

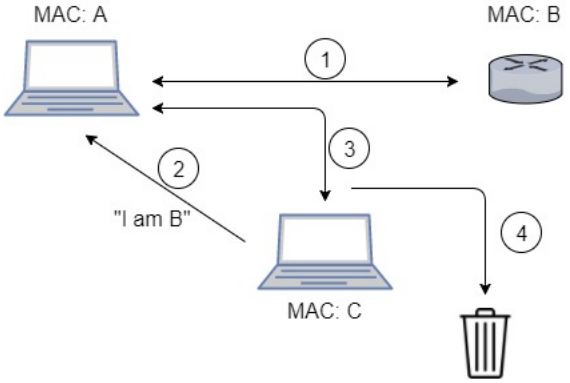

Figure 3: DOS attack

Remote Access

On a final note, ARP cache poisoning can be exploited even from outside of the local area network. All the attacker has to do is gain remote access to one of the hosts currently connected to the LAN by compromising some vulnerability in its operating system. From there on, everything can be automated with the numerous ARP poisoning scripts and tools open sourced online [11]. We have highlighted only a couple of the attacks that exploit ARP cache poisoning to emphasize the dangers of the exploit. We now turn to methods provided in related works used to detect and mitigate them.

III. Related Works

The four main techniques used to tackle ARP cache poisoning are static, dynamic, centralized, and cryptographic techniques. Starting off with the static techniques; Cisco Systems offer an administratively static solution called switch port security [12]. This method requires network administrators to manually configure the number of different MAC addresses allowed per port on their network devices. Port security has proven to work well for small to medium sized networks, however, it does not scale well for larger enterprises. In addition, this method requires the installation of high cost routers and switches that might not be available for smaller businesses. Another costly but highly effective solution proposed by [5] suggests dividing a given network into a large number of subnets; each with a small number of hosts. This approach prevents exposing the entire network in case a host is compromised. Again, this technique; like all static techniques faces the issue of scalability and administrative costs placed on network administrators.

On the other hand, dynamic techniques follow a more automated approach to detecting ARP cache poisoning that requires minimal administrative intervention . These techniques use network monitoring tools such as XARP [9], ARPWatch [6], Anti Netcut [1], NoCut [2], and AntiARP [3] to log ARP traffic and take note of suspicious behavior. These tools monitor changes in local caches and report unusual patterns back to network administrators. In addition, most of these tools run in promiscuous mode which gives them access to all traffic on the network. This allows these tools to pick up and detect signature traffic patterns used in ARP MITM attacks. While all of these methods successfully detect ARP cache poisoning attacks, they heavily rely on network administrators to take further action towards mitigating them. Other dynamic techniques use intrusion detection systems (IDS) such as Snort [15] for detection. These systems conventionally use machine learning methods to classify malicious traffic patterns. Although most of these systems offer high accuracy rates, they tend to also have high false positive rates which create an inconvenience for network administrators.

[8] propose implementing an ARP central server (ACS) that manages and authenticates all ARP traffic on the network. The server also implements a secondary long term cache as a backup in case the server itself has been poisoned. Although they offer an elegant solution for cache poisoning correction, the authors disregard that fact that the server itself can be impersonated and thus compromised. This makes the approach flawed because it creates a single point of failure for gaining access to the link layer of the network. Similarly, TARP [14] sends out a ticket with every ARP message as a form of authentication. The approach relies on implementing a centralized Local Ticket Agent (LTA) which faces the same problems that the ACS does.

Moving on to the cryptographic approaches proposed to solve ARP cache poisoning, we start off with S-ARP [4]. This method introduced public key cryptography as a layer on top of ARP that implements authentication. It is important to note that with this approach, a key distribution center (KDC) is needed to distribute keys to the hosts in the network. While S-ARP solves the issue of authentication, it does so at the cost of time consuming computations and complex infrastructure implementation. On the contrary, [13] propose using one-way hash functions to generate a Merkle hash tree that is distributed to all hosts. This approach has a computational advantage over public key cryptography since hash functions are known to be faster. The drawback of the design is that it is based on a single trusted node in the network which is assigned the task of computing the Merkle tree.

IV. Proposed Solution

In the previous section we discussed some of the key weaknesses found in recently proposed solutions. In this section we demonstrate how our design avoids major pitfalls and still manages to detect and correct cache poisoning. Before we dive in to the proposed method, we emphasize that our system makes a strong but realistically feasible assumption about the environment that it is running in. The system assumes that the first two entities spawned in the network are trusted and secured entities that are not trying to poison each others caches. Even if these entities are compromised at a later point in time, the system will detect and correct any malicious entries found in the cache tables of these entities as new hosts join the network. We will consider one of the entities in our network to be the gateway, and the other to be the first host that connects to the subnet. From now on, any time the term “error” is mentioned in the text, it is referring to a malicious entry found in a cache.

We are proposing the implementation of a peer-to-peer system that distributes the ARP cache tables of hosts belonging to a subnet among each other. The purpose of sharing this information with peers is to provide an error detection and correction mechanism on the network. By following a distributed approach, we avoid the problem of creating a single point of failure which is observed in the centralized systems discussed previously. In addition, our

approach is autonomous in the sense that no administrative intervention is needed. Finally, the proposed methods is backwards compatible with conventional ARP and most of the methods mentioned in the related works section which encourages its use as an additional line of defense. Before we explain the details of the error detection and correction algorithm, we propose our cache distribution protocol first.

Cache Distribution Protocol

To begin with, each host keeps track of the other hosts that it has shared its cache with along with time stamps that represent the last time the cache was shared and modified. By running hosts in promiscuous mode; they are given access to all ARP traffic on the network. Whenever an ARP reply packet is sent out on the network, all hosts will share their ARP cache table with the host receiving the reply. With the goal of saving bandwidth, a host can choose to not share it's cache table if it has already shared it before and has not modified it since. Once any host receives an ARP reply packet, it will update its current cache immediately (as proposed in conventional ARP) then will remain idle for up to α seconds until its peers caches have been received. Also, all peers will not accept any additional ARP traffic for at least $\alpha + \beta$ seconds until the verification algorithm has been executed on the receiving host were β represents the run time of the verification algorithm. The purpose of this timeout period is to prevent any error from propagating back on to the network. The cache distribution protocol is described below.

Algorithm 1 Cache Distribution Protocol

Require: NIC running in promiscuous mode
1: $LocalIP \leftarrow Host.IP$
2: **if** ARP Traffic Detected **then**
3: **if** $Packet.DestinationIP! = LocalIP$ **then**
4: Send Cache to $Packet.DestinationIP$
5: Reject all ARP traffic for $\alpha + \beta$ seconds
6: **else**
7: Execute ARP protocol
8: Accept caches for α seconds
9: Execute cache correction algorithm
10: **end if**
11: **end if**

It is clear that for the given protocol to function properly, two distinct parameters needs to be established. Both α and β are two non-deterministic parameters that are network and host dependent respectively. In section V of the paper, we provide a detailed analysis on the impact of the blocking time of these parameters on the performance of application layer applications.

Verification Algorithm

We will now explain how our error detection and correction algorithm works by considering the scenario of a network consisting of $N \geq 2$ entities. The cache distribution protocol will provide each entity with up to *N-1* cache tables that the verification algorithm will use to verify entries inside the local cache. Assume a host wants to verify an entry *X* found in its local cache. That entry will contain two components; an IP address IP_X and a MAC address MAC_X. This host also has access to a set of *N-1* ARP cache tables at most; that may or may not contain an entry for the IP address IP_X. We now assume that there exists $k \leq$ *N-1* cache tables that do have an entry for the IP address IP_X. For those *k* cache tables, we perform a majority vote on the value of the corresponding MAC address entry found in each of those tables. This means that our host will look over *k* MAC address values and find the one that appears the most. The host will then compare the entry found in its cache table with the most occurring MAC address. If the two are the same then the host agrees with the majority of his peers and is not under attack. If the two entries are different, then our algorithm will swap out the current value with the most occurring value found in the remote caches. In the case when the algorithm returns equally probable MAC address mappings, the host will not update his cache and will keep the current value. Algorithm 2 below provides a detailed description of how a cache is verified at each end host.

Algorithm 2 Cache Correction Algorithm

Require: $LocalCache, RemoteCaches$
1: **for each** $localEntry \in LocalCache$ **do**
2: $ipToVerify \leftarrow localEntry.IP$
3: $macToVerify \leftarrow localEntry.MAC$
4: $counter < MAC, count > \leftarrow null, 0$
5: **for each** $cache \in RemoteCaches$ **do**
6: **for each** $entry \in cache$ **do**
7: **if** $entry.IP = ipToVerify$ **then**
8: $counter \leftarrow entry.MAC, count++$
9: **end if**
10: **end for**
11: **end for**
12: $mostOccuringMac = Max(counter)$
13: **if** $macToVerify! = mostOccuringMac$ **then**
14: $localEntry.MAC \leftarrow mostOccuringMac$
15: **end if**
16: **end for**

V. Experimental Setup and Evaluation

To test the proposed protocol and verification algorithm, we have created a real network environment consisting of 3 virtual hosts on VMWare Workstation. Each host is running Ubuntu Linux with 4 GB of RAM and all hosts are interconnected using a NAT adapter configuration. Both the cache distribution protocol and the verification algorithm are implemented using Java and the TCP socket library. Finally, the environment is tested on a 16 GB 2.8 GHZ machine running Windows 10.

In the coming subsections, we will go back to the ARP cache poisoning attacks mentioned in section II and show how the proposed method mitigates them. It is important to note there exists a few special cases that our method does not address. In addition, the proposed protocol is implemented at the cost of network traffic volume Δ, cache distribution time α, and verification algorithm run time β.

Successful Mitigation Cases

Man-In-The-Middle

We will first start off with the man-in-the-middle scenario described in section II. The attackers main goal is to poison two or more entities on the network at the same time. The moment the attacker sends out the first ARP reply message to his first target, all other hosts on the network will detect this message and enter a blocking period for $\alpha + \beta$ seconds. Now, when the attacker sends out the second ARP reply message to his second target, that target will be in a blocking state and will simply drop the packet. Although it is costly, the blocking period proposed in the protocol makes sure that a single host at most can be poisoned on the network. Consequently, any MITM attack is reduced to a less sophisticated attack that involves the poisoning of a single entity.

Single Host Poisoning

Assuming that a single host has been poisoned by an attacker; we just showed that no other cache can possibly be poisoned on the network as a result of the cache distribution protocol. The poisoned host can be assured that all possible $N - 1$ caches that he receives carry valid network data. Consequently, once the cache verification algorithm is executed $\alpha + \beta$ seconds later; all poisoned entries in the cache are remediated. Since this algorithm is executed after every ARP reply message, it ensures that cache poisoning is always detected and corrected on the spot. Also, since all hosts enter a timeout period after detecting the ARP reply message, this implies that malicious ARP traffic is not propagate further on the network through the sharing of caches.

Limitations and Special Cases

IP Harvesting Attacks

In the previous subsection we showed how the proposed method resolved two common attack scenarios: MITM and identity impersonation. However, there exists a special case where an attacker is capable of getting his malicious IP-MAC binding registered in the cache of a target when that specific IP address has not yet been assigned to a node on the network. This attack is known as an IP harvesting attack. As mentioned in [8], if an attacker manages to register his IP-MAC binding on any node, it will be honored because no other legitimate cache will have a conflicting entry for that specific IP.

Legitimate IP Address Changes

The final scenario we will address is when a legitimate entity changes its IP address. In short, we are trying to differentiate between an actual IP address change and an IP address spoof. To solve this problem, we adopt the Antidote approach proposed in [7] which checks if the previous MAC address is still alive. Upon receiving a new IP-MAC mapping that contradicts with an entry found in the current local cache, a host will send out 50 probe packets to the original MAC address before running the verification procedure. If any of the 50 packets gets a reply then the entire procedure is aborted and the ARP reply is ignored. Otherwise, the host attempts to verify the entry he just received by using the information received by his peers and the verification algorithm. It is important to note that this scenario gives an attacker a small window for successfully spoofing an IP address. The attacker starts off by sending out a fabricated ARP reply message to one of his targets claiming to own a certain MAC address. He then immediately runs a denial of service attack on the MAC address he is claiming to own by flooding it with traffic that prevents it from replying to one of the 50 probe packets. With enough traffic generated and aimed at his target, this attack will slip by undetected.

Integrating the Antidote [7] Method

By combining the Antidote approach with algorithm 1, we add an extra layer of protection against cache poisoning; but also introduce new costs. For each ARP reply message; not only are we going to have to wait for $\alpha + \beta$ seconds, we now need to introduce a new timeout period γ which represents the time required to send out 50 probe

packets and wait for a reply. Similarly, the network traffic volume has increased from Δ to $\Delta + 50 * \lambda$ where λ represents the size of the probe packet. Algorithm 3 below shows the integration of the Antidote approach with algorithm 1.

Algorithm 3 Cache Distribution Protocol With Antidote

Require: NIC running in promiscuous mode
1: $LocalIP \leftarrow Host.IP$
2: **if** ARP Traffic Detected **then**
3: **if** $Packet.DestinationIP! = LocalIP$ **then**
4: Send Cache to $Packet.DestinationIP$
5: Reject all ARP traffic for $\alpha + \beta$ seconds
6: **else**
7: **if** mapping in packet contradicts cache **then**
8: Send 50 probe packets to original IP
9: **if** any reply returns **then**
10: Terminate procedure
11: **else**
12: Execute ARP protocol
13: Accept caches for α seconds
14: Execute cache correction algorithm
15: **end if**
16: **else**
17: Execute ARP protocol
18: Accept caches for α seconds
19: Execute cache correction algorithm
20: **end if**
21: **end if**
22: **end if**

Overhead and Run Time Complexity

In this subsection, we will be computing the overhead associated with our proposed method. We break down the overhead into two categories; traffic volume and blocking time. Cisco Systems recommend a maximum of 500 hosts per subnet as a result of broadcast overhead. We use this number to evaluate the worst case cost of our protocol and algorithm. We will first define the costs associated with traffic volume.

$$\begin{cases} \Delta & \text{Total Cache Distribution Traffic Volume} \\ \Lambda & \text{Cache packet size} \\ \lambda & \text{Probe packet size} \end{cases}$$

$$\begin{cases} \Lambda = N * (32 + 48) & \text{N} \geq 0 \\ \Delta = \Lambda * (N - 1) + 50 * \lambda & \text{N} \geq 1 \end{cases}$$

Assuming N = 500 hosts in our subnet; whenever an ARP reply message is sent out, N-1 caches are transmitted after it. If we also assume the worst case scenario of each cache having N entries in it, this implies an average cache size of 5 KB (500 entries * (32 bit IP address + 48 bit MAC address)). The worst case cost of cache verification is then roughly 2.5 MB (5 KB * 499) which can be broken up into 499 connections in parallel each transmitting 5 KB only.

We now turn our attention towards analyzing the blocking time of the proposed method. In addition to bandwidth costs, the protocol introduces an ARP timeout period of $\gamma + (\alpha + \beta)$.

$$\begin{cases} \alpha & \text{Cache Distribution Time} \\ \beta & \text{Verification Algorithm Run Time} \\ \gamma & \text{Probe Packet Distribution Time} \end{cases}$$

Both α and γ are dependant on the NIC installed on each host. β on the other hand depends on the computational specifications of the end host. The run time complexity of algorithm 2 is $O(n) + O(n^2)$ which has the worst case scenario of $O(n^2)$. With the goal of minimizing the overall blocking time and maintaining a minimum run time safety margin, we decided to give α and γ the value of 0.5 seconds and β the value of 1 second. We also added an extra safety margin blocking time of 0.5 seconds to make sure that all distributed tasks have been executed which gives a total blocking time of 2.5 seconds per request.

Application Layer Performance

We will now be modeling the worst case performance of our proposed protocol by visualizing the impact of the blocking time on the performance of the application layer. We assume that there exists N hosts on our subnet and all of these hosts have empty ARP caches. All N hosts are trying to resolve some IP-MAC binding at the same time. Since our protocol only allows the processing of a single request at a time, this turns conventional ARP into a contention protocol where hosts are competing for a slot to resolve an IP address. In the scenario where every single host is fighting for a slot, the worst case blocking time will grow with the number of hosts on the network.

The graph below models the contention time required to gain access to the ARP channel by the most unsuccessful host on the subnet. Clearly, the proposed protocol has a heavy impact on the performance of the application layer due to the build up of multiple timeout periods. As the number of hosts increases on the network, the probability of two or more hosts resolving some IP address at the same time will increase as well. The timeout period then becomes a function of the number of hosts.

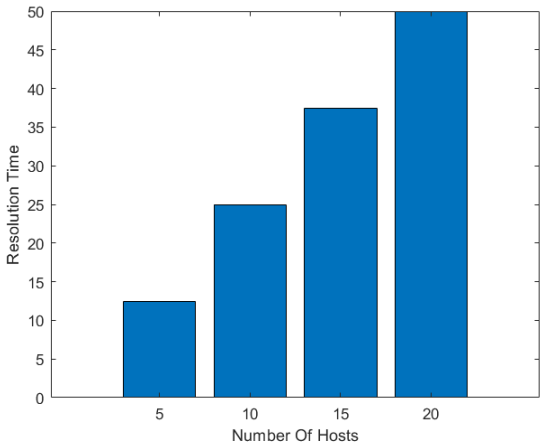

Figure 1: Worst Case Resolution Time (seconds) as a Function of the Number of Hosts on the Subnet

VI. Conclusion and Future Work

In this paper, we have explained how authentication can be exploited in the ARP protocol to launch multiple attacks on LANs. We have also mentioned approaches that solve this problem through cryptographic, centralized, dynamic, and static solutions. After highlighting the weaknesses of each approach, we proposed our distributed cache sharing algorithm which is combined with a majority voting scheme for error correction. The overhead of the approach is mentioned in the literature along with some special cases that our method does not solve. Clearly, there is a strong trade off between security and performance.

Future research in this area can be done to improve the algorithm proposed by making it completely distributed. One way of doing this is to broadcast the ARP reply to all peers and have them run the verification algorithm at their end. This will reduce the run time complexity of the algorithm to $O(n)$ at each host since a simple linear look-up is done. After running the algorithm, a small binary reply message can be sent out from each peer back to the original host indicating if the mapping in the ARP reply agrees with an entry found in that peers cache. Not only will this approach reduce run time, it will also reduced the bandwidth overhead because caches are not transmitted over the network anymore. In addition, the minimization of the parameters α, β, and γ will improve the performance of the protocol further.

References

[1] Anti netcut version 2.0.

[2] A. Ali. Nocut 1.001a.

[3] ColorSoft. Antiarp.

[4] A. Omaghi D. Bruschi and E. Rosti. S-arp: A secure address resolution protocol. In *Annual Computer Security Applications Conference*, December 2003.

[5] T. Demuth and A. Leitner. Arp spoofing and poisoning: Traffic tricks. In *Linux Magazine*, page 56:2631, July 2005.

[6] L. N. R. Group. Arpwatch, the ethemet monitor program; for keeping track of ethemet/ip address pairings.

[7] I.Teterin. Antidote.

[8] S. Kumar and S. Tapaswi. A centralized detection and prevention technique against arp poisoning. In *Proceedings Title: 2012 International Conference on Cyber Security, Cyber Warfare and Digital Forensic (CyberSec)*, pages pp. 259–264., 2012.

[9] C. Mayer. Xarp advanced arp spoofing detection.

[10] D. C. Plummer. An ethernet address resolution protocol. In *IETF RFC 826*, November 1982.

[11] SANS Institute InfoSec Reading Room. An ettercap prime. Technical report, SANS Institute, 2004.

[12] Cisco Systems. Configuring dynamic arp inspection. In *Catalyst 6500 Series Switch Cisco IOS Sofware Configuration Guide*, pages Chapter 39, Pages 39:1–39, 2006.

[13] V. Kumar V. Goyal and M. Singh. A new architecture for address resolution. 2005.

[14] W. Enck W. Lootah and P. McDaniel. Tarp: Ticket-based address resolution protocol. In *Annual Computer Security Applications Conference*, December 2005.

[15] Z. Jiang X. Hou and X. Tian. The detection and prevention for arp spoofing based on snort. In *2010 International Conference on Computer Application and System Modeling vol. 5 IEEE*, pages V4–137, 2010.