2016 IEEE 41st Conference on Local Computer Networks

Securing ARP in Software Defined Networks

Talal Alharbi
School of ITEE
The University of Queensland
Brisbane, Australia
Email:t.alharbi@uq.edu.au

Dario Durando
School of ITEE
The University of Queensland
Brisbane, Australia
Email:durandod@eurecom.fr

Farzaneh Pakzad
School of ITEE
The University of Queensland
Brisbane, Australia
Email:farzaneh.pakzad@uq.net.au

Marius Portmann
School of ITEE
The University of Queensland
Brisbane, Australia
Email: marius@ieee.org

***Abstract*—The mapping of Layer 3 (IP) to Layer 2 (MAC) addresses is a key service in IP networks, and is achieved via the ARP protocol in IPv4, and the NDP protocol in IPv6. Due to their stateless nature and lack of authentication, both ARP and NDP are vulnerable to spoofing attacks, which can enable Denial of Service (DoS) or man-in-the-middle (MITM) attacks. In this paper, we discuss the problem of ARP spoofing in the context of Software Defined Networks (SDNs), and present a new mitigation approach which leverages the centralised network control of SDN.**

Keywords*—*SDN, Security, ARP

I. Introduction

The resolution of network layer addresses to link layer addresses is an essential function in packet switched networks. In IPv4 networks, this service is provided by the Address Resolution Protocol (ARP), and in IPv6 by the Neighbour Discovery Protocol (NDP), in particular via Neighbour Solicitation (NS) and Neighbour Advertisement (NA). Both ARP and NDP (NS/NA) are vulnerable to spoofing or poisoning attacks, where an attacker can create false entries in a host's ARP cache in IPv4 or Neighbour Cache in the case of IPv6 [1]. As a result of a successful attack, packets are sent to a malicious node instead of the intended destination. This can be used to launch DoS and MITM attacks. While some of the technical details vary, the basic mechanism of ARP and NDP (NS/NA) and their vulnerabilities to spoofing attacks are very similar. In this paper, we will focus our discussion on ARP, but our findings equally apply to NDP (NS/NA) and IPv6. Hence, our proposed SDN-based ARP spoofing mitigation mechanism can easily be adapted to NDP (NS/NA).

II. Background

A. ARP

Figure 1 shows the structure of ARP packets. The ARP payload is encapsulated in an Ethernet frame with an *Ether Type* field of *0x0806*. The payload includes *Sender Hardware Address* (SHA) and *Target Hardware Address* (THA), which are the MAC addresses of the sender and the intended receiver, i.e. target. The frame also contains the *Sender Protocol Address* (SPA) and *Target Protocol Addresses* (TPA) fields, which represent the IP addresses of the sender and the target. The *Operation* field indicates if the packet is a request or a reply.

To find the MAC address for a given IP address, a node broadcasts an ARP Request with the SHA field set to its own MAC address and the SPA set to its own IP address. The TPA field is set to the IP address of the target node, while the THA field is initialised with a dummy value of *00:00:00:00:00:00*,

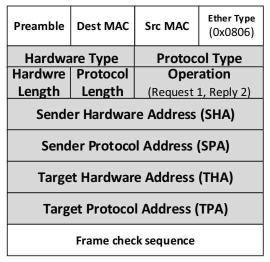
Preamble
Dest MAC
Src MAC
Ether Type (0x0806)
Hardware Type
Protocol Type
Hardwre Length
Protocol Length
Operation (Request 1, Reply 2)
Sender Hardware Address (SHA)
Sender Protocol Address (SPA)
Target Hardware Address (THA)
Target Protocol Address (TPA)
Frame check sequence

Fig. 1. ARP Frame Structure

representing the unknown target MAC address. Each node that receives the ARP Request will learn about the IP-MAC address mapping of the sending node (SHA and SPA), and will add the information to its ARP cache. Each recipient node will check if the TPA value in the request matches its own IP address, and if so, will respond with an ARP Reply message.

The fields of the ARP Reply message are initialised as follows. SHA is set to the MAC address of the node that received the ARP Request, and represents the answer to the question posed in the request. SPA is set to the corresponding IP address, and is copied from the TPA field in the corresponding ARP Request. Finally, the THA and TPA fields of the ARP Reply are initialised as the SHA and SPA fields of the corresponding ARP Request. The ARP Reply is then unicast to the MAC address of the sender of the ARP Request. The recipient of the ARP Reply will update its ARP cache and add the newly learned SPA-SHA address mapping [2]. ARP also support *gratuitous* replies, which are unsolicited messages without a corresponding request. We refer to the ARP handling approach discussed above as *Regular ARP*, in order to contrast it with *Proxy ARP*. Proxy ARP is a mechanism where a node, e.g. a router, answers ARP Requests on behalf of the target host.

B. ARP Spoofing

The basic security problem with ARP (and NDP) is that it is a stateless protocol, i.e. it treats each request or reply independently from any previous communication. As a consequence, a host will readily accept information from gratuitous ARP Replies, without having sent a corresponding request. In addition, the ARP protocol has no mechanism to authenticate the sender of an ARP Request or Reply message, or to check the integrity and validity of the provided information. As a result, it is relatively easy for an attacker to poison a host's ARP cache with a false IP-MAC address mapping. All he

DOI 10.1109/LCN.2016.83

needs to do is to craft an ARP Request or Reply message with a false SPA field. A host receiving the message will simply trust the content and update its ARP cache accordingly.

C. *Traditional ARP Spoofing Countermeasures*

S-ARP [3] proposes a cryptographic protection for ARP. It uses public key cryptography and relies on each node having a public/private key pair. The ARP packet format is extended and adds a digital signature field, providing message authenticity and integrity. S-ARP only protects ARP Reply messages and therefore remains vulnerable to ARP Request based spoofing attacks. The other key limitation of this approach is its reliance on a Public Key Infrastructure , in particular a Certificate Authority, for its operation. This has proven to be impractical. A similar cryptographic solution exists for the protection of the NDP (NS/NA). The Secure Neighbor Discovery (SEND) protocol is a security extension of NDP [4]. This approach shares similar practical challenges with S-ARP imposed by the problem of bootstrapping and key management.

A number of non-cryptographic ARP spoofing detection and prevention have been proposed for traditional IP networks. A survey of these measures is provided in [5]. One of the most prominent is Dynamic ARP Inspection (DAI) [6]. DAI is implemented in Ethernet switches and checks ARP packets against a trusted database of IP-MAC address mappings, which is challenging to establish. Further drawbacks of DAI include its proprietary nature and relatively high cost [5].

D. *ARP Handling in SDN*

In this paper, we assume OpenFlow as the SDN southbound interface, and will make use of its *Packet-In* and *Packet-Out* messages, which allow encapsulated data packets to be sent between switches and the controller.

There are two basic approaches to handling ARP in SDN. The first, which we refer to as *Regular ARP*, is as discussed in Section II-A. Here, the SDN controller simply provides packet forwarding functionality. The second approach to handle ARP in SDN is via Proxy ARP, which is well suited to SDN, due to its centralised control plane. Here, ARP Requests are sent to the controller, which then answers them on behalf of the target hosts. This assumes that the controller has the required IP-MAC address mapping information.

III. ARP and NDP (NS/NA) Spoofing in SDN

A. *Experimental Platform*

We discuss and demonstrate the vulnerability of SDN to ARP and NDP (NS/NA) spoofing attacks via experiments, for both Regular and Proxy ARP. For this and our later experiments, we used Mininet and Open vSwitch, and POX as our SDN controller platforms. In order to craft ARP packets for our attacks, we used the Scapy packet manipulation library. All our experiments were run on a PC with a 3 GHz Intel Core 2 Duo CPU with 4 GB of RAM, running Ubuntu Linux 14.04. For now, we use the simple scenario with 3 switches and 3 hosts shown in Figure 2.

B. *ARP Spoofing Experiments*

For the Regular ARP case, we use POX as the controller, with its L2 learning switch component (*l2_learning*) to implement packet forwarding functionality. In our example, we assume host *h1* has been compromised and wants to poison host *h2*'s ARP cache by creating an entry that maps IP address 10.0.0.3 to MAC address *00:00:00:00:00:01*. As a result, all packets from *h2* addressed to *h3* will be sent to the attacker *h1* instead. We implemented the attack by crafting an ARP Request with TPA=10.0.0.2, SPA=10.0.0.3 and SHA=00:00:00:00:00:01, and inject it from *h1* to switch *S1* via port *P1*. As a result, host *h2* updates its ARP cache and adds an entry mapping the IP address 10.0.0.3 to the MAC address 00:00:00:00:00:01, and hence the attack was successful. We further replicated these experiments for IPv6 and NDP (NS/NA), with identical results.

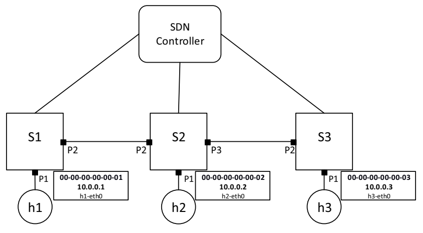

Fig. 2. Basic Example Scenario

We have replicated the above ARP spoofing attacks for Proxy ARP handling in SDN, using POX's *arp_responder* component. Instead of poisoning a host's ARP cache, the attack resulted in a poisoned ARP table of *arp_responder*, which in turn resulted in poisoned ARP Replies provided to hosts.

IV. Countermeasure: SARP_NAT

As mentioned, ARP spoofing attacks occur either via ARP Request or Reply messages. In both cases, an attacker will spoof the SPA and SHA fields in the ARP message in order to create an invalid address mapping. The key idea of our mechanism is to prevent the potentially spoofed information in the SHA and SPA fields to come into contact with any hosts and thereby prevent the poisoning of their ARP cache, in the case of Regular ARP. For Proxy ARP, our mechanism prevents the poisoning of the ARP table of the controller's ARP handler. Our goal is to implement a controller component which does not handle ARP itself, but secures the existing ARP handling mechanism. Our proposed solution is loosely inspired by Network Address Translation (NAT), and we therefore call it SARP_NAT. Algorithm 1 shows the packet processing of SARP_NAT at the controller, explained in the following.

A. *ARP Request based Attack*

To explain the basic operation of SARP_NAT, we consider the scenario shown in Figure 2. We will refer to the relevant lines in Algorithm 1. We assume Regular ARP handling for now. Host *h1* attempts to launch the same ARP Request based spoofing attack described in Section III-B, by injecting a poisoned ARP Request with an invalid SPA field to switch *S1*. Due to a pre-installed rule at the switch, the ARP Request is sent to the controller, where it is first handled by the SARP_NAT component. The SARP_NAT component stores each ARP Request in a list of pending ARP Requests ($pend\_req$), consisting of the following 6-tuple for each entry:

Algorithm 1 SARP_NAT

```
 1: for all received $pkt$ do
 2:     if $pkt$.type = ARP.REQUEST then
 3:         add entry in $pend\_req$ list
 4:         $pkt.arp.spa = spa_{safe}$
 5:         $pkt.arp.sha = sha_{safe}$
 6:     end if
 7:     if $pkt$.Type = ARP.REPLY then
 8:         if $pkt \notin pend\_req$ then
 9:             Gratuitous ARP Reply, drop
10:         else
11:             $pkt.arp.tpa = lookupSPA(pend\_req, pkt.arp.spa)$
12:             $pkt.arp.tha = lookupSHA(pend\_req, pkt.arp.spa)$
13:             $deleteEntry(pend\_req, TPA)$
14:             $addEntry(handled\_req, pkt, t_{handled})$
15:             send ARP Reply to $pkt.arp.tha$ via Packet-Out
16:             if $pkt \in handled\_req$ then
17:                 $expSHA = lookupSHA(handled\_req, pkt.arp.spa)$
18:                 if $pkt.arp.sha \neq expSHA$ then
19:                     Duplicate ARP Reply attack detected, drop
20:                     Delete potentially poisoned host ARP cache
21:                 else
22:                     Genuine duplicate ARP Reply, drop
23:                 end if
24:             else
25:                 Gratuitous ARP Reply, Drop Packet
26:             end if
27:         end if
28:     end if
29: end for
```

$(tpa, spa, sha, s, in\_port, t_{rec})$ (line 3 in Algorithm 1). The elements tpa, spa and sha represent the corresponding fields in the ARP Request, s and in_port represent the switch ID and ingress port via which the request was received, and t_{rec} is the time at which the request was received.

The SARP_NAT component then 'sanitises' the ARP Request message by overwriting the potentially poisoned fields SPA and SHA with safe dummy values, spa_{safe} and sha_{safe} (lines 4,5). These safe values are constant and can be any IP and MAC address pair that is not used on the local network. In our implementation, we used $spa_{safe} = 11.11.11.11$ and $sha_{safe} = 00:11:22:33:44:55$. The Target Protocol Address (TPA) remains unchanged. The SARP_NAT component then passes the sanitised ARP packet to the next controller component or event handler for processing. If we are using Regular ARP handling and a corresponding forwarding component, this will result in the ARP Request packet being broadcast in the network, where it will eventually be received by *h2*, the target node (TPA).

According to the pre-installed ARP rule, each ARP packet is sent to the controller for checking. However, this is not necessary for packets that have been sanitised by the SARP_NAT component and contain safe SPA and SHA values of spa_{safe} and sha_{safe}. We therefore modify the default ARP forwarding rule to specify that ARP packets with safe values are exempted, and are being forwarded as the normal forwarding rules.

If we assume Regular ARP handling, the ARP Request is forwarded to the target host with IP address TPA, host *h2* in our example. Host *h2* then parses the ARP Request packet and adds the specified SPA-SHA mapping in its ARP cache. This is the point where normally the ARP spoofing attack would have succeeded. However, in this case, host *h2* simply adds the safe dummy values to its ARP cache, which has no impact.

When *h2* creates an ARP Reply message, it assumes it is responding to the (non-existent) host with IP address spa_{safe} and MAC address sha_{safe}, and initialises the ARP Reply accordingly, i.e. with TPA=spa_{safe} and THA=sha_{safe}. According to the standard ARP behaviour, the sender of the ARP Reply (*h2*) initialises the fields SPA and SHA with its own IP and MAC address, in our example SPA=10.0.0.2 and SHA=00:00:00:00:00:02. The ARP Reply is then sent as a unicast frame via switch *S2*. Since the SPA and SHA fields are not the safe values spa_{safe} and sha_{safe}, the ARP Reply is forwarded to the controller by switch *S2*, where it is processed by the SARP_NAT component. For an ARP Reply, SARP_NAT first checks if it is a reply to a genuine request, i.e. if it is in $pend\_req$ (line 8). If that is not the case, we have a gratuitous ARP Reply, which we cannot trust and therefore drop. If, however, the ARR Reply matches an entry in $pend\_req$, the component performs the reverse address translation, by replacing the dummy values in the TPA and THA fields with the original values stored in $pend\_req$, consisting of the set of $(tpa, spa, sha, s, in\_port, t_{rec})$ 6-tuples (lines 11,12). In our example, the values are set as follows: TPA=10.0.0.3 and THA=00:00:00:00:00:01. The corresponding entry is now deleted from $pend\_req$ (line 13) and a new entry is created in a list of handled ARP Requests $handled\_req$ (line 14). In addition to the information in $pend\_req$, the list $handled\_req$ also contains the resolved MAC address obtained from the ARP Reply, as well as the time when the reply was received.

Finally, the ARP Reply is directly forwarded to host *h1*, via an OpenFlow Packet-Out message to $s = S1$, with instructions to send it out via port in_port, i.e. port *P1* in this case (line 15). The processing by SARP_NAT is completely transparent to hosts.

The above discussion considered the case of Regular ARP handling. The proposed SARP_NAT mechanism can also be used for Proxy ARP handling. This required minor modifications to *arp_responder* in our implementation. Normally, *arp_responder* implicitly trusts the information in any ARP Request it receives from switches, and the (potentially poisoned) information is used to update its local ARP table. We modified *arp_responder* to only trust IP-MAC address mappings in ARP Reply messages which are sent in response to a genuine ARP Request. This prevents spoofing attacks using ARP Request messages against Proxy ARP handling.

B. *ARP Reply based Attack*

ARP spoofing can also be done via gratuitous ARP Replies. To prevent such attacks, SARP_NAT will only accept ARP Replies for which it has seen a corresponding request, i.e. for which there is an entry in the list of pending ARP Requests $pend\_req$, otherwise the ARP Reply is dropped (line 9). To avoid this check, an attacker can respond to a genuine ARP Request with a spoofed reply. In this case, the SARP_NAT component receives two ARP Replies with different values in the SHA field, one from the genuine target host, and one from the attacker. Unfortunately, it is impossible for the SARP_NAT component to determine which one is the genuine reply and which one is the spoofed one. SARP_NAT simply accepts the first ARP Reply it sees for a pending request, processes and forwards it. It further enters information of the reply into its $handled\_req$ list (line 14), in particular the corresponding SHA value and the time it was received.

If during a given time window t_{dup} another ARP Reply for the same request is received, but with a different IP to

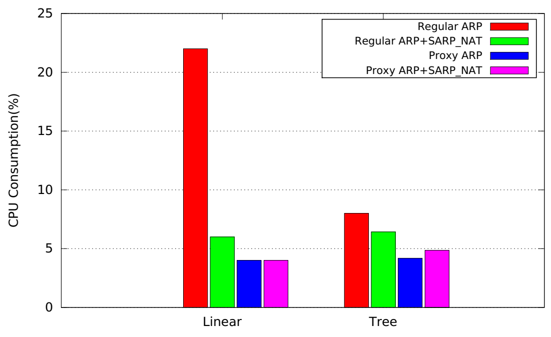

Fig. 3. Controller CPU Load (SARP_NAT)

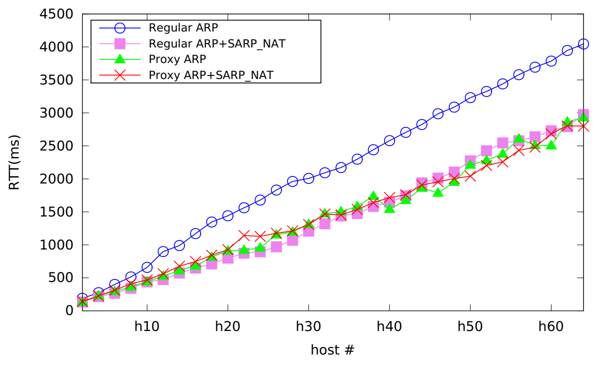

Fig. 4. RTT Linear Topology (SARP_NAT)

MAC address mapping (lines 17-19), a *duplicate ARP Reply* attack is detected. Since it is not possible to determine which of the ARP Replies was spoofed, we make the conservative assumption that it was the first one that was forwarded to the host. To mitigate the impact, the SARP_NAT component will overwrite the potentially poisoned entry in the host's ARP cache by sending a new ARP Reply with the same SPA but the safe value sha_{safe} (line 20). This stops any packets from being forwarded to the potentially wrong MAC address, thereby preventing a man-in-the-middle attack. As a downside, it also prevents the host from communicating to the target host, until the next valid ARP Reply is received. The only way for this kind of attack to go undetected is if the attacker can prevent the genuine host from sending its valid ARP Reply. While not impossible, this presents a significant challenge and greatly raises the bar for a ARP spoofing attacks. SARP_NAT's protection against ARP Reply based spoofing attacks works for both Regular ARP handling as well as Proxy ARP handling.

C. SARP_NAT Overhead

To evaluate the overhead imposed by SARP_NAT, we performed experiments using two topologies, a linear topology with 64 switches and hosts, and a tree topology with depth 2 and fanout 8, and also 64 hosts. We measured the average controller CPU load for different ARP handling scenarios (Figure 3). For this experiment, we used POX with its L2 learning switch component. We generated ARP Requests at a constant rate of 500/s at $h1$, with target hosts set to $h2, h3, ..., h64, h2, h3, ...$ consecutively and continuously.

We consider the results for the linear topology first. Surprisingly, we see that for Regular ARP, running SARP_NAT results in a significant reduction in CPU load. This is due to the fact that we install a rule which immediately forwards broadcast ARP Request packets that have been 'sanitised' by the controller, instead of sending them to the controller at every switch, as is the behaviour of POX's *l2_learning* component. In this case, adding security has the benefit of a significant controller overhead reduction, without sacrificing any generality of the solution. For Proxy ARP, SARP_NAT does not provide any reduction in CPU load, since the packet forwarding is the same in both cases. Here, SARP_NAT minimally increases the CPU load by well below 1%.

For the tree topology and Regular ARP, we also see a reduction in CPU load due to SARP_NAT. However, the reduction is smaller compared to the linear topology, since the average path length is smaller, and hence the number of Packet-In messages processes by the controller is smaller too. For Proxy ARP, we see a small increase of less than 1%.

We also measured the impact of SARP_NAT on the ARP 'round trip time' (RTT), i.e. the time to handle an ARP Request. Due to lack of space, we only show the results for the linear topology, shown in Figure 4. The figure shows the ARP RTT for ARP Requests sent by $h1$, for different target hosts $h2, h3, ..., h64$, as indicated on the x-axis. We see that Regular ARP without SARP_NAT has the highest overall RTT values, due to the forwarding behaviour of the *l2_learning* component. The advantage of SARP_NAT is reflected here, with a reduction of 32% on average compared to the basic Regular ARP case. For Proxy ARP, SARP_NAT imposes a minimal overhead of 2% on average. Overall, we can say that SARP_NAT imposes a minimal overhead, and in the case of Regular ARP even achieves a significant overhead reduction.

V. Conclusions

In this paper, we have presented SARP_NAT, a novel approach to mitigate against ARP spoofing attacks in SDN. SARP_NAT does not rely on a trusted IP-MAC database, and can defend against both ARP Request and Reply based spoofing attacks. SARP_NAT's active address translation approach provides a novel solution to the problem of ARP spoofing in SDN, and can operate in conjunction with current SDN-based ARP handling approaches.

Acknowledgment

This work is supported by Majmaah University through the Saudi Arabian Culture Mission in Australia.

References

[1] F. A. Barbhuiya *et al.*, "Detection of neighbor solicitation and advertisement spoofing in ipv6 neighbor discovery protocol," in *International conference on Security of information and networks.*

[2] K. Kwon *et al.*, "Network security management using arp spoofing," in *ICCSA 2004.*

[3] D. Bruschi *et al.*, "S-arp: a secure address resolution protocol," in *Computer Security Applications Conference, 2003.*

[4] J. Arkko, J. Kempf, B. Zill, and P. Nikander, "Secure neighbor discovery (send)," RFC 3971, March, Tech. Rep., 2005.

[5] C. L. Abad *et al.*, "An analysis on the schemes for detecting and preventing arp cache poisoning attacks," in *ICDCSW'07.*

[6] *Dynamic ARP Inspection.* [Online]. Available: http://www.cisco.com/c/en/us/td/docs/switches/lan/catalyst6500/ios/12-2SX/configuration/guide/book/dynarp.html