ARPnat table of contents (probably need to rename arpnat)

Contents

1 Introduction

ARPNAT is a solution developed in Python using the SDN controller POX. The DAI is based on the fact that the information about the network is already known to the network administrator in some way, so that he/she can create a static mapping between IP addresses and MAC addresses. However, this assumption is not always true. In open networks, where everyone can access, techniques like DHCP snooping may not applicable or trustworthy. This is why this component was created. It aims to reduce the possibility of a poisoning to a minimum while not knowing anything about the network itself. The main idea behind this component is to avoid the possibility or having possibly malicious ARP packets travelling through the network.

2 Design

Whenever an ARP request is sent in the network, it is immediately sent to the controller, where the Source Protocol Address (SPA) and Source Hardware Adrress (SHA) values in the arp packet payload are modified to prevent any possible arp request attack. After this procedure, the new "sanitized" packet is sent back in the network, where it reaches the target host. From the host's point of view, this procedure is transparent, since it does not know that the packet has been previously modified, and so it will answer to the sanitized packet's source address with its addresses. The controller then catches the ARP reply, modifies the Target Protocol and Hardware Address fields(TPA/THA), putting back the initial informations that were used in the first request packet, and unicasts it to the initial sender. The ARPNAT component has two different tables: one used to keep track of the pending ARP requests that are waiting for an answer, and the other used to keep in memory the recently received replies. Both are designed to keep in memory the entries for a certain amount of time, after which they are deleted. As it can be seen in figure XXX, if H1 sends an ARP request to the network, the controller will take care of it, modify the SHA/SPA fields to match a predifined safe value that can be selected by the user (in our case we chose `10.0.0.100/00:11:22:33:44:55`) and then broadcast it to the network. H2 will receive the request and reply to `00:11:22:33:44:55` instead of `00:00:00:00:00:01`. The controller upon receiving the ARP reply from H2 will change THA/TPA to `10.0.0.1/00:00:00:00:00:01` and unicast the reply to H1, that will in this way receive the same packet it would have received in a traditional network.

3 Security evaluation(could put some terminal screenshots to demonstrate it works)

The component is called ARPNAT because it acts in some way like a masking device between any host and the rest of the network. This component is designed to deal with both kind of arp posoning attacks previously described

3.1 how does it deal with the attacks (request-replies)

1. **Request**: In case of an ARP request attack, the poisoning information is stored in the SHA/SPA fields of the ARP packet. So, whenever an ARP request is sent by a host towards the network, the first switch that receives automatically redirects it to the controller via a set of preinstalled rules. The controller handles the incoming request by modifying the aforementioned fields with a safe IP/MAC pair that can be decided at launch by the network administrator. After this modification of the ARP packet, the latter is sent again in the network. When the target host receives the packet, it will store in its ARP cache only the safe information that was inserted by the controller. In this way, the potentially malicious information that could have

been injected in the ARP request is not transmitted to the network and will not update the ARP tables of the other hosts. After this, the packet is sent to the safe address decided at the beginning, captured by the controller, that redirects the reply to original sender. [SCREENSHOT OF ARP TABLE BEFORE AND AFTER MALICIOUS PACKET IS SENT]

2. **Reply**: In case of an ARP reply attack, the procedure is harder. Infact, since we do not have a static table with all the informations about IP addresses and MAC addresses, there is no way to know if the information received is safe or not. Moreover, being ARP a stateless protocol, it is not possible to distinguish between requested and gratuitous replies coming from the network.
   In our implementation, we keep track of pending arp requests that are waiting for a reply. In the case we receive a reply from an address which is not in out waiting list (gratuitous case), we just drop the packet. Otherwise, if we are waiting for a reply from that address, the packet is redirected to the original sender. This behaviour can be exploited by an attacker, instead of just sending a poisoned ARP reply to the network by answering an ARP request with a poisoned packet. In case the malicious packet reaches the controller before the real reply, it will just be forwarded to the host that sent the original arp request, and the real reply will be dropped as a gratuitous reply. For this reason, the controller keeps track of the replies sent back to the network by the controller for a certain amount of time. If another reply for the same target IP but different MAC is received in that window of time, the controller sends an alert, telling the user that suspicious behaviour has been detected. Unfortunately there is no way for the controller to know if the first ARP reply received was malicious or not, so ARPNAT proceeds in sanitizing the entry in the ARP table, modifying the entry with a safe mac address in correspondance to the attacked IP address. The next time the host will try to reach that IP, it will fail, trying to connect to a non-existing MAC address, and try again to send an ARP request. In case multiple attacks happen, the traffic towards that IP could need to be blocked. A possible exploit of this defense mechanism could involve a DoS attack towards the original target of the ARP request, so that it will not answer the request, allowing only the attacker to reply, triggering no defense mechanisms. However multiple DoS defense mechanisms were discussed in [paper X] and [paper Y], so the simple integration of those components and ours can prevent this scenario from happening.

3.2 behaviour with normal sdn l2_learning switch and with arpresponder-host_tracker

1. **ARP Responder**: ARP responder updates its central arp table when a packetIn event is raised by an arp request. This behaviour does not work with our implementation, since requests, that raise packetIn events (POX events raised everytime

a packet is received at a switch, unless the packet was sent directly by the controller via a packetOut message), are not trustworthy. The only case in which we can trust information in our system is when the controller is sending a reply back to the original sender of the request, so we had to modify the source code of `arp_responder.py` in order to update only in case of packetOut event (triggered whenever the controller sends a single packet to the network) raised by an ARP reply packet. In case multiple replies are detected, ARP responder, that could have its ARP table poisoned, will delete entry created by the possibly malicious packets. In case a new packet needs to be sent to that host, ARP responder will be bypassed, since it contains an empty entry, and the ARP request will be sent in the network.

2. **Host Tracker**: Host tracker tables are updated when an ARP packet is received through a packetIn event . The Host tracker component however relies on a mapping between MAC addresses as Keys and IP addresses as values, while ARP table have IP addresses as keys and MAC addresses as values. This behaviour is kept to take into account the possiblity of virtual interfaces. However it makes the process of deleting an entry in the table a bit harder. Even though it was not implemented in this work, a possible solution would be to introduce a multiple reply event, that would notify the host tracker component on which entry to delete form its table. This feature was not implemented due to the lack of time (MAYBE I CAN DO THIS, BUT FOR NOW IT IS NOT DONE) but will be our next objective.

4 Implementation(i kept the rules in the file, but I think we can eliminate them)

The component was developed entirely using the POX [reference] platform, an intuitive and well documented project that was born as a fork of the first SDN controller, NOX [reference]. The ARPNAT features are all performed by one component, that runs together with a forwarding mechanism. In our experiment it was run with the forwarding.l2_learning switch, a component that acts as a standard L2 learning switch, that installs in the switches rules that are exact matches on as many fields as possible. Upon start, the component installs 4 rules in the switches. The rules are the following(obtained running the command `ovs-ofctl dump-flows s1`):

1.
   ```
   cookie=0x0, duration=211.220s, table=0, n_packets=0, n_bytes=0, idle_age=211,
   priority=28672, arp, dl_src=00:11:22:33:44:55, arp_spa=10.0.0.100,arp_op=1
   actions=FLOOD
   ```

   The first rule matches the ARP requests that have been sanitized by the controller and are free to travel through the network. So whenever they are received by a switch, they are simply flooded in the network (`actions=FLOOD`).

2. `cookie=0x0, duration=211.180s, table=0, n_packets=0, n_bytes=0, idle_age=211, priority=28672, arp, dl_dst=00:11:22:33:44:55, arp_tpa=10.0.0.100, arp_op=2 actions=CONTROLLER:65535`

   The second rule simply sends all the replies that are sent by the target host directly to the controller, that can now modify it and send it back to the original host (`action=CONTROLLER:65535`).

3. `cookie=0x0, duration=211.180s, table=0, n_packets=0, n_bytes=0, idle_age=211, priority=28672, arp, arp_op=1 actions=CONTROLLER:65535`

   The third rule sends all the ARP requests created by the hosts to the controller in order to sanitize them from possibly malicious content(`action=CONTROLLER:65535`).

4. `cookie=0x0, duration=211.180s, table=0, n_packets=0, n_bytes=0, idle_age=211, priority=28672, arp, arp_op=2 actions=ANY`

   This final rule drops all the replies that are not sent to the controller by the second rule. This means all the replies that have as target an address different from the safe one(`action=ANY`).

The reader has to keep in mind that the order or the rules is important, since the packets are matched to the rules according to their order in the flow table. This means that a packet that satisfies the first rule, will not try to match the second one, but will instead complete the action associated with the first rule.

4.1 arpnat implementation

The arpnat controller handles all packetIn events before the forwarding.l2_learning switch due to an imposed priority. So whenever a packet raises a PacketIn event, it is sent to the controller. In these cases the arpnat component starts a procedure to modify the content of the packet. Once the packet is modified in the right way, it is sent to the network through a PacketOut message, either with the FLOOD option is case of a request or unicast in case of a reply. If the packet that raises the event is not an ARP packet, it will pass through the component unmodified and it will be sent right after to the l2_learning switch, that will deal with it like it normally would. The two tables used by the component to keep track of the pending requests and arrived replies are Python dictionaries designed to automatically delete entries after a timeout.

4.2 modification of arp responder

The arp responder component had to be modified to be compatible with the arpnat implementation. A PacketOut handler was introduced to be able to update the ARP table only in case of a reply. Infact trusted replies are sent only by the controller through a packetOut message, that does not raise a packetIn event. Whenever a multiple reply attack is detected, the entry corresponding to the attacked IP is deleted from the table.

5 Experimental evaluation

The experiments to evaluate the efficiency and security of the component were performed using the mininet platform. The modelled topology was the one used also for the previous solution, in order to be able to compare the results.

5.1 cpu

[NEED TO LIST COMPUTER SPECS] The analisys of the cpu consumption of the component was performed by generating a high load of traffic in the network and measure the processing power needed by the controller. To define what is a high load of traffic for a controller in case of ARP traffic, the university network was analyzed in order to have meaningful data. From this analysis it resulted that the average number of ARP packets in the network was around 60 a second, with the highest peak of traffic beeing around 200 ARP packets. We decided to run our experiments with 500 ARP requests generated every second and sent to random hosts in the network using python scripting and scapy [scapy ref.]. The results can be seen in figure [XXX]. The 4 scenarios analyzed were the following: l2_learning switch alone, l2_learning switch with ARPNAT component, l2_learning switch adn arp_responder, l2_learning switch with arp_responder and arpnat. All of these were analyzed in both topologies. From the results it is clear that l2_learning switch alone is the most cpu consuming. The result is most evident in the linear case, due to the high number of switches involved. The reason behind this high value is that l2_learning switches as default behaviour send always multicast packets to the controller, and ARP requests are multicast, so whenever a request reaches a new switch, it will inevitably be handled bu the controller. Using ARPNAT on the other hand eliminates this problem, thanks to the rules installed at launch. Each request and reply is sent just once to the controller, then it is either dropped or redirected through rules. The two cases involving arp_responder are very similar and cheap in term of cpu, and this is expected. Once the switches create their arp table, no additional arp packet is required in the network, hence creating needing very few interactions with the controller. Apart from l2_learning switch, all the other component compositions are consistently low in both topologies. The results show that the design of the component allows good scalability, maintaining the same performances from a optimized tree topology to a linear worst case scenario, where each switch is connected to only one host and the number of switches that need to be crossed is maximum.

5.2 time

To analyze the possible time overhead that our additional feature could add, we decided to use the ping application [ping ref. ?]. In our experiment, a ICMP request is sent from H1 to every other host in the network for 30 times to collect reliable data, taking care of deleting entries in ARP tables and flows in the switches. Figure [XXX] reports the average time for a ping request to receive an ICMP reply from the target host. The results shows that the average time for H1 to ping any other host is actually smaller when arpnat is used. This result could seem counterintuitive at first, but there is a reason behind these performances. In case of simple l2_learning switch, the ARP requests are sent to the controller everytime they raise a PacketIn event, which means every time they are received by a switch. So every switch will send the packet to the controller, that will tell the switch to flood the message to all its ports. In our case, every arp packet is sent to the controller only once, and then the flow rules installed in the switches will take care of the rest, making the whole process faster. This is very clear in the linear topology case, where the l2_learning switch is by far the slowest component. All the other component have very similar delays, as they require a lot less interaction with the controller. In the tree topology case, the l2_learning switch is again the slowest, but the delays are very similar due to the design of the network, optimized to reduce to a minimum the number of travelled switches to reach any host, which is the main slowing factor in the linear case.
Overall the arpnat component scores better performances that the plain l2_learning switch in both cpu consumption and time delay, and the same performances in case also arp_responder is used. This is essentially caused by the unoptimized implementation of the component for protocols that make use of broadcast traffic, like ARP.