ARP SPOOFING DETECTION ON SWITCHED ETHERNET NETWORKS: A FEASIBILITY STUDY[1]

Marco Antônio Carnut (kiko@tempest.com.br)
Tempest Security Technologies
Centro de Estudos e Sistemas Avançados do Recife - CESAR
Universidade Federal de Pernambuco – CIn/UFPE

João J. C. Gondim (gondim@cic.unb.br)
Departamento de Ciência da Computação
Instituto de Exatas – IE
Universidade de Brasília – UnB

ABSTRACT

This paper describes a set of techniques to detect ARP spoofing attacks on switched Ethernet networks, both by suggesting implementations to be made directly to the switches' firmwares and alternative techniques that rely only on external elements, such as specialized sniffers and inference from SNMP data collection. These elements are combined in an architecture general enough for practical implementation in production networks. Results from laboratory and real-world detection experiments using several popular attack tools are also presented.

1 INTRODUCTION

Classical training in computer networking teaches that ethernet switches provide better security because they are supposed to forward traffic only to the appropriate port, without relaying it to unintended recipients where a passive sniffer may be in action [1, 2]. ARP spoofing is a powerful technique that circumvents this protection, being often used as a first step in more elaborate attacks. Although well known in the security practictioners communities [3, 4, 5], it is still surprisingly absent from the standard training and textbooks in networking. This, along with a lack of specialized tools to detect it, makes most real life attacks go entirely unnoticed.

Although a few tools and some newer IDSs can detect ARP anomalies, most of them do not specifically target ARP spoofing – an interesting attack to detect because it is highly intentional. No viruses or worms use it, so when such attack is in effect, there is certainly some human controlling it. Besides, most IDSs rely on sniffer-based sensors to detect those anomalies, greatly restricting their effectiveness in switched networks.

This paper proposes specific sniffer-based tools to detect ARP spoofing attacks and examines the feasibility of detecting them on swiched networks in a passive and efficient way, using the switches' SNMP MIB-II traffic statistics counters.

The rest of this paper is organized as follows. Section 2 presents a brief description of ARP and the strategies used by common ARP spoofing attack tools. The resulting characterization will provide the basis for several detection techniques: section 3 describes straightforward sniffer-based detection strategies, their applicability and limitations; section 4 builds on some principles learned from the previous section to devise detection and prevention techniques at the switch level; section 5 describes a detection strategy for switched environments based on inferring a possible attack by observing patterns on the packet counts that the switches' SNMP MIB-II [8] commonly provide. Finally, section 6 shows how to combine all these components into a detection framework that very closely resembles traditional IDS architectures, while section 7 wraps-up conclusions and future work directions.

2 ARP AND ARP SPOOFING

2.1 ARP: Address Resolution Protocol

ARP [6, 7] can be seen as an *adaptation layer* between the IP and MAC layers to dynamically translate between 32-bit IP addresses and 48-bit Ethernet addresses. It is a very simple protocol, comprised of only two messages:

- **Request ("who-has"):** specifies the IP address of the host whose MAC address we want to find out. It is almost always sent as a broadcast frame, so as to hopefully reach the host with the desired IP address when we don't know its MAC address.
- **Reply ("is-at"):** the answer a host should send, specifying the MAC address associated to that IP. It is almost always sent as a unicast frame directed to the MAC address of the machine that sent the request.

Whenever IP needs to send a packet to some host on the local network, a kernel-resident table called *ARP cache* is looked up to check whether the MAC address corresponding to the destination IP address is already known. If it is, IP can carry on: a frame is assembled with its destination address pointing to the MAC taken from the ARP cache and then sent on the wire. Otherwise, a broadcast ARP request frame is sent asking the host with the destination IP address to inform its MAC address in a unicast ARP reply frame. Figure 1 shows this process in greater detail.

There are other aspects to the ARP protocol, its implementation and practical usage that we should be aware of:

[1] This paper was presented at the 5th Simpósio Segurança em Informática (Symposium Security in Informatics) held at Instituto Tecnológico da Aeronáutica (Brazilian Air Force Technology Institute), São José dos Campos, São Paulo, Brazil, Nov 4-6, 2003.

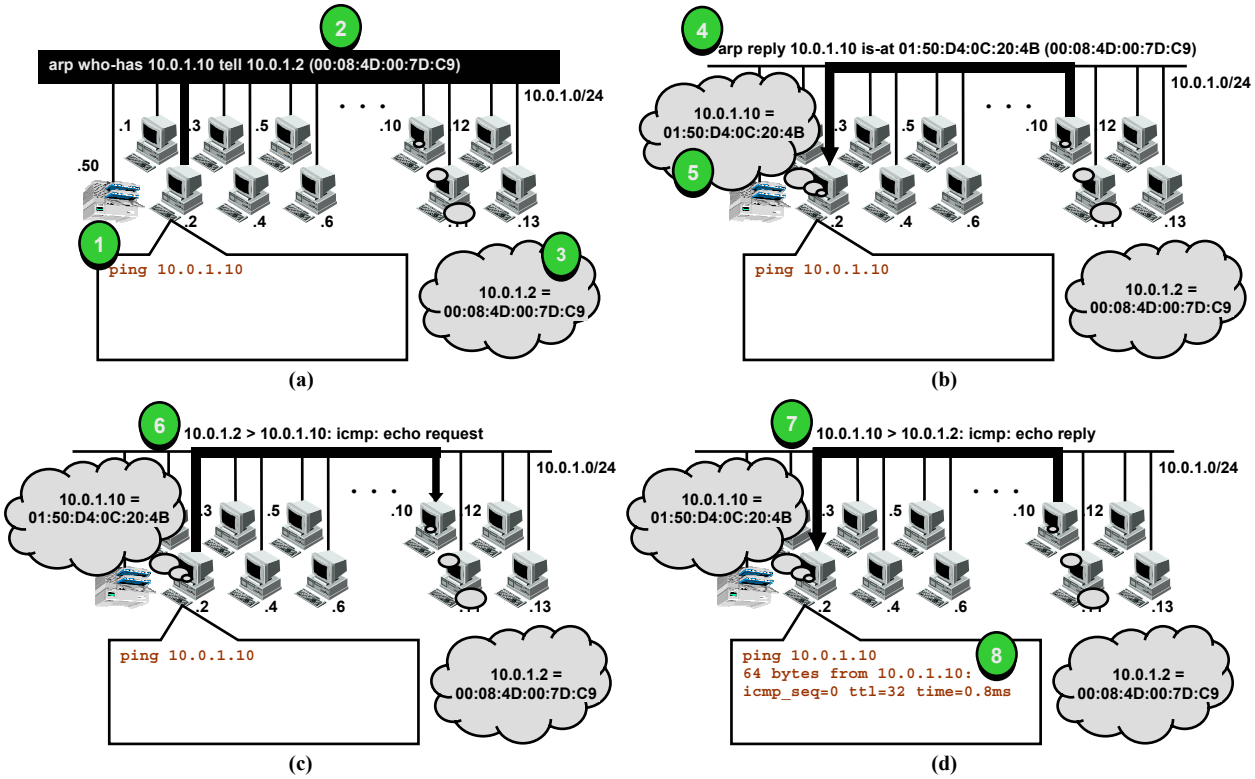

Figure 1: ARP protocol in action in a local network. In (a1), station 10.0.1.2 tries to ping 10.0.1.10 but, since it doesn't know which MAC address to send the frame to, it (a2) broadcasts an ARP request asking who has it. All machines on the network (assumed to initially be with their ARP caches empty) hear but ignore this request, except 10.0.1.10. In (a3), it first records in its ARP cache the IP and MAC addresses of the machine which asked. In (b4), it sends a unicast ("private") reply telling its MAC address back to 10.0.1.2, which stores it in its ARP cache in (b5). Now that both machines know their respective MAC addresses, the ICMP echo (the "ping request") can be sent (c6). When the other station replies (d7), the IP layer passes it along to the ping application program, which prints some of its contents (d8). Further traffic will require no ARP exchanges until entries expire in either ARP caches.

- **Static ARP mappings:** Most implementations provide means for the administrator to insert static entries in ARP caches. In most systems, those entries may not be overwritten by ARP exchanges. We will take advantage of this feature in section 6 to make a secure setup for our detection infrastructure.
- **Gratuitous ARP:** When an IP interface is brought up, the ARP subsystem first tries to detect potential IP address duplicates in the local network by issuing a *gratuitous ARP*: an ARP request looking for its own IP address. It tries a few times and, if there's no response, it assumes no one is using that IP address and brings the interface up. On the other hand, if some other machine on the local network responds, it reports to the administrator that a duplicate IP address has been detected and shuts down the interface.
- **Proxy ARP:** some network setups have machines legitimally responding ARP on behalf of other hosts. A common situation is a router that answers ARP with its own MAC address for hosts connected through another interface. It is very uncommon to have a host answer ARP for another host *in the same local network*, but this is not necessarily illegitimate.

The key point to observe is the fact that normal ARP behavior is always on the form of one or more broadcast requests followed by one unicast reply (it is exceedingly rare to have more than one reply in legitimate exchanges).

2.2 ARP Spoofing Attacks

ARP is supposed to be stateless – that is, it should not correlate requests with replies – and thus prone to accepting replies even if it hasn't issued a request. The ARP spoofing attack, detailed in figure 2, is greatly facilitated by this fact: the attacker sends a spoofed ARP reply to one of the victim hosts telling the IP address of the other victim is at the attacker's MAC address. This is the so-called *cache poisoning* component of the attack.

Since the victim's ARP subsystem has no recollection on whether it really asked that or not, it accepts the reply and updates its ARP cache accordingly. This causes packets destined to the victim's interlocutor to be diverted to the attacker, which can do whatever she wants with it: discard them, resulting in a denial-of-service; relay them to the legitimate destination while storing them for later analysis (the combination of a passive sniffer and an ARP spoofer is often called an *active sniffer*); or even modifying them in real time for use with more sophisticated attacks, such as TCP hijacking or the cryptographic man-in-the-middle attack during initial public key exchange. We call this the *relaying* component of the attack.

The procedure described above puts the attacker in the position of intercepting only one direction of the communication between the victims. In some attack

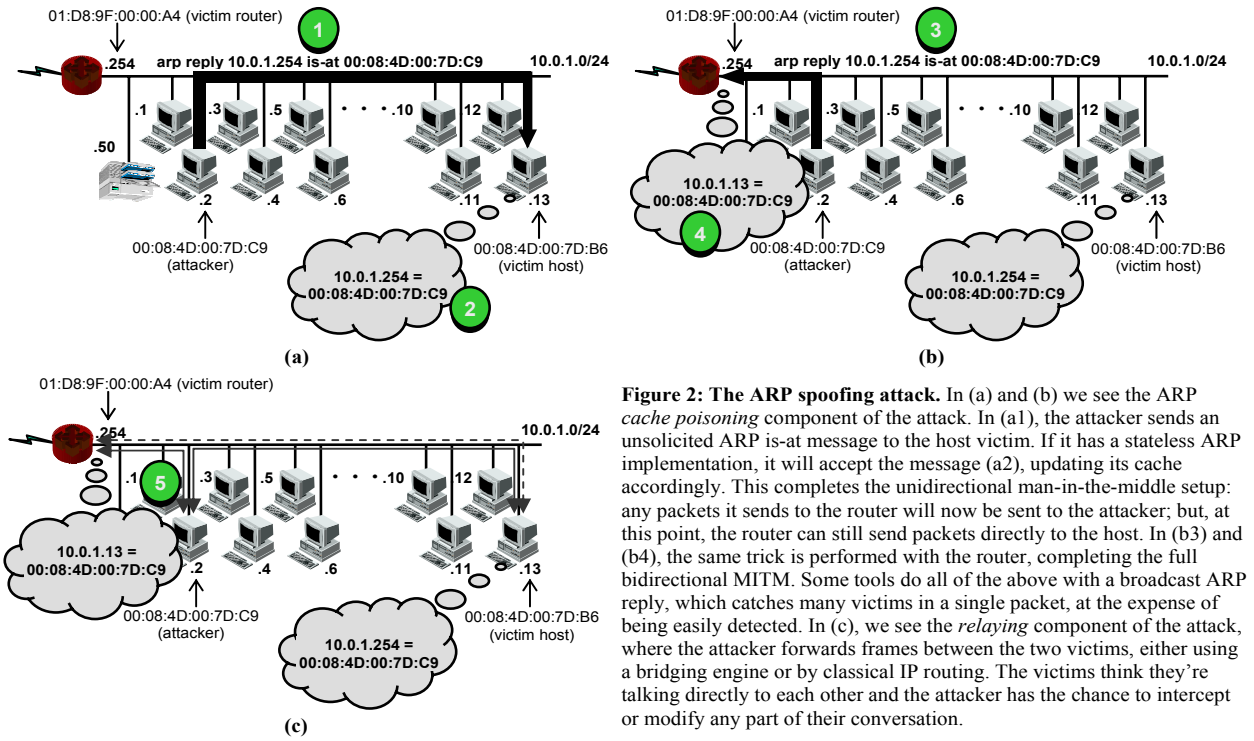

Figure 2: The ARP spoofing attack. In (a) and (b) we see the ARP *cache poisoning* component of the attack. In (a1), the attacker sends an unsolicited ARP is-at message to the host victim. If it has a stateless ARP implementation, it will accept the message (a2), updating its cache accordingly. This completes the unidirectional man-in-the-middle setup: any packets it sends to the router will now be sent to the attacker; but, at this point, the router can still send packets directly to the host. In (b3) and (b4), the same trick is performed with the router, completing the full bidirectional MITM. Some tools do all of the above with a broadcast ARP reply, which catches many victims in a single packet, at the expense of being easily detected. In (c), we see the *relaying* component of the attack, where the attacker forwards frames between the two victims, either using a bridging engine or by classical IP routing. The victims think they're talking directly to each other and the attacker has the chance to intercept or modify any part of their conversation.

scenarios, this may be just enough. If the attacker wishes, she can perform a similar poisoning on the other victim, making the interception fully bidirectional.

The primary motivation for performing this attack is because it works even on switched networks. Although it is often said that switches are immune to interception, this is only correct for *passive interception*, but not for an active attack like this. The "port security" feature some switches sport for limiting the amount of learned MAC addresses per port, while useful for resisting to flooding attacks or broadcast storms, is of no avail to stop this attack (we confirmed that in our lab experiments). The key point is that ARP spoofing is not about faking MAC addresses or fooling the switch; it is about faking *MAC ⇔ IP mappings* and fooling *the hosts* – the switches simply go along. Static MAC ⇔ port mappings in the switches only stops attack tools which rely on faked MAC addresses – as we shall see later on, some tools do that, while others don't; it is not really essential for the attack to work. Static ARP ⇔ IP mappings on every host are a functional countermeasure, although not practical for anything but the smallest networks.

Some ARP implementations are very credulous: for instance, Windows 95 and 98 update their ARP caches *even for static entries*. Windows NT and above, however, don't accept this, but are easy prey for the standard attack outlined above. They also gladly accept broadcast ARP replies, making it very easy to spoof dozens of machines with a single packet.

ARP cache entries are supposed to expire: after some time (typically between tens of seconds to a few hours, according to the implementation), they are deleted, so that new ARP request-replies are generated. This helps to reestablish communications when hosts *move* – that is, when their interfaces are reconfigured with the other IP or MAC addresses. Most implementations also handle host movements by updating their ARP caches when they receive gratuitous ARPs.

These timeouts force spoofers to keep sending fake ARP replies periodically. If the victims are credulous, it suffices to send spoofed ARP replies every ten seconds or so. As long as the spoofer keeps doing this, the victims will not issue ARP requests for that IP, since their ARP cache entries will always be within the timeout threshold. We call this the "quiet" cache poisoning. If the attacker can perform what she wants within a single cache expiration period, she could send just one or two ARP frames. We call this the "ultra-quiet" poisoning. The quieter the attacks, the harder they are to detect, primarily on switched networks.

Broadcast-based attacks, although often involving very few packets, are classified as extremely noisy, because broadcasts can be readily detected by sniffers even on switched networks.

Newer implementations, such as in recent Linux kernels, have made ARP stateful and thus a lot more skeptical: it remembers which requests it has made and only accepts replies for those. This makes the attack harder, but not impossible, since we can often force the victim to ask: suppose the attacker fabricates a fake ICMP echo request from one victim to the other, sending it *directly through the ethernet driver*, bypassing IP – so as to avoid the initial ARP request-reply exchange. Along with that, the attacker also sends a train of a few fake ARP replies. When the victim receives the ICMP echo request, it will try to discover the other victim's MAC address by issuing an ARP request that will be almost instantly answered by one of the replies in the train, before the legitimate host may have time to answer itself. By the time its answer arrives, the victim will no longer accept it.

Against skeptical victims, the attacker runs into a *race condition*: since they ignore all ARP replies except the ones matching previous requests, common approaches are either to send the replies as fast as possible, hoping that the fake reply arrives before the legitimate one; or combine it with the ICMP + train of ARPs technique previously described. These attacks are also rather noisy, in the sense that the attacker needs to send dozens to thousands of packets to succeed. We will see that this makes them more easily detectable.

The relaying strategy also varies for each attack tool, which also determines the choice of MAC addresses using in the poisoning portion of the attack:

- **Layer-2 bridging:** the tool receives frames at the ethernet interface (say, by means of the ethertap device in Unix-like OSs) and resends it immediately changing just the destination MAC address. When performing the poisoning, the attacker sends ARP replies mapping one of the victim's IP to a randomly invented MAC address so as to prevent interaction with its own IP stack.
- **Layer-3 IP routing:** if the attacker uses the very MAC address currently associated to some of its interfaces, it can just let the IP layer forward the packets itself – all it has to do is enable IP forwarding and disable ICMP redirects in its own IP stack.

3 DETECTION ON HUBBED NETWORKS

In shared-medium network segments, where a sniffer can listen to all traffic, ARP anomalies are easy to watch. An widely used program for this purpose is LBL's `arpwatch` [13]. Its main focus, however, is not detecting ARP spoofing attacks; instead, it aims to keep sysadmins informed (usually via email) about changes in the IP ⇔ MAC mappings. This catches many interesting events, such as IP addresses being changed (by authorized personnel or not), MAC addresses being changed (either by software reconfiguration or by phisically replacing ethernet cards), new machines being added to the network (because of gratuitous ARPs), common misconfigurations (like IP address conflicts), etc. However, `arpwatch` can't tell these non-malicious events apart from intentional ARP spoofing attacks. On large busy networks with overworked or lax sysadmins, where typically hundreds of ARP anomalies are reported daily, many real serious attacks may pass unchecked.

That was our motivation to implement a specialized sniffer to evaluate strategies geared specifically towards detecting ARP spoofing attacks. We assembled a test network where all machines where interconnected through a hub. The two victim machines ran Windows NT (easy prey), which had their ARP caches manually cleared before each test. The spoofer machine ran Linux 2.4, using Hunt 1.4 [10] and angst-0.4b [11] as attack tools to perform both cache poisoning and packet relay. On later tests, we used a Linux 2.4 victim. Its skepitcal ARP proved immune to those tools, although it was easily fooled by denver 1.0 [12], which uses the ICMP+ARP strategy.

The resulting detection tool will be used as a building block in the final architecture described in section 6. The specific algorithms we developed are described in the following subsections.

3.1 The Request-Reply Mismatch Algorithm

The sniffer listens for ARP packets, keeping a table of pending requests keyed by MAC address. Entries are removed from the table when the matching reply arrives after a timeout period (we used two seconds). If a reply is seen without a matching request being present in the table, we notify the administrator that we're under attack. Basically, this is the same strategy used by the skeptic kernels, but applied in our sniffing context, so it's not vulnerable to the ARP+ICMP interaction trick – in this mode, our tool ignores everything but ARP frames.

We made the program flag as attacks some very unusual occurences, such as broadcast ARP replies (generated by the `arpspoof` program of the `dsniff` [9] package) and unicast requests.

In our laboratory, this algorithm performed flawlessly: we never had even a single false positive, and it never missed any attacks, correctly identifying the MAC and IP addresses used by the offender. Further testing is needed, however, to see how it performs in unusual environments – for instance, some high avaliablity systems based on IP takeover work by generating odd ARP events that our algorithm may incorrectly consider an attack. Hopefully, these cases are rare enough to be treated as exceptions.

3.2 The Duplicate Packet Detection Algorithm

In this algorithm, the sniffer listens for IP packets. We hash the entire packet content, modulo the ethernet addresses, the IP TTL field and the IP header checksum, using the hash as index to a state table marking the time we've last seen this packet. Entries are removed when they've been lingering on the table for more than two seconds. If we receive a packet whose hash is already on the table (assuming we have a decent collision free hash algorithm), we alert the administrator that we're seeing the layer-2 or the layer-3 relaying component of the attack.

This algorithm achieves nearly the same detection accuracy of the previous one, except that it might detect legitimate routers – it didn't happen in our test networks, but it might happen elsewhere. (Having a layer 3 router forwarding packets for the same segment is unusual, but sometimes it does happen – for instance, two IP networks sharing the same physical medium or a routing misconfiguration).

It may seem surprising that upper-layer (e.g., TCP) retransmissions don't trigger false positives, but it is easily explained by the fact that the IP ID field gets incremented, so retransmissions do not generate exact packet duplicates.

The algorithm's main drawback is that it uses a lot of memory – although the hashes are smaller than the original the packets, the table grows to a few megabytes during periods of intense network traffic.

3.3 Switch Packet Counters Simulation

The tool implementing those techniques was called Switch Statistics Simulator Agent – SSSA, for short – because it originally started life with an entirely different purpose: we needed MIB-II's `ifTable`-like statistics even when there were no switches. Only later we added the aforementioned dedicated detection algorithms.

The switch statistics simulation works as a simple "frequency/load count" on the packet's source and destination MAC addresses: when we receive a packet, we allocate a "virtual switch port number" for each MAC address we don't have already. We then count how many bytes and packets went from each virtual port to another, in the exact same way the embedded SNMP agents on dedicated switches calculate them. In fact, all this output is made available via the SNMP protocol, so that the detection tool we will develop on section 5 can't even distinguish whether it is talking to a real switch or to our simulated one.

4 DETECTION ON SWITCHED NETWORKS

For efficiency, most LANs are structured hierarchically where one or more high speed core switches provide connectivity to lower-end (called "distribution" and "border") switches and/or hubs. Switches act as "layer 2 routers", keeping a table (called "MAC address table") of which MACs are on which ports so that most of the time they only forward unicast frames to the specific port the destination host is connected to. Here's the reason why many attack tools use unicast ARP packets – on switched environments, if our detection tool is located on a different port than the attacker or the victims, the attack packets will only rarely reach our sniffer, thus greatly minimizing the chance of detection.

We said "only rarely" because sometimes switches do send unicast frames to all ports: when their MAC address tables don't contain which port the destination MAC is at, they have no choice but to momentarily act as hub, sending the packet to all ports. In switch tech parlance, this is not said to be a broadcast; it is called an *overload*. When the target machine later responds, the switch records the port it is connected to in its MAC address table.

Overloads are infrequent, being caused primarily by expiration policies in the MAC address table entries. But they do allow a sniffer to get occasional glimpses at the traffic from other ports. This makes an ARP spoofing attack possible, though hard, to be detected by using a sniffer. In our test network, modified so that all participants were on different switch ports, both `arpwatch` and SSSA managed to detect a few attacks – but only when they were very noisy (like one ARP reply per second or faster in the poisoning phase) and lasted for several minutes. By making the attack as quiet as possible, our lab tests showed it was very easy to evade detection at all. This confirms what the underground communities already knew: the attack is quite practical and effective.

Many commercial switches provide facilities for *port mirroring*: configuring the switch for replicating traffic from one or more ports to another port, called the *mirroring port*, where we could connect our sniffers. However, since there is usually a substantial performance impact when port mirroring is in effect, this strategy makes ARP spoofing detection based on sniffing not quite viable on switched LANs.

4.1 Detection on Reprogrammable Switches

The next logical step would be to implement our detection strategies directly in the switch – it is the one device that sees all traffic. In principle, our algorithms should be easy to implement in commercial switches directly in their firmwares. However, only their manufacturers have the tools and documentation to actually do this.

We then opted to build a homebrew switch using a standard PC with a quad-port ethernet card using the bridging features in Linux 2.4's kernel along with `ebtables` [15], which provide frame-level filtering capabilites just like `iptables` [16] does for IP packets. Using `ebtables` it is trivial to implement the ARP Request-Reply Mismatch algorithm. In our lab setup, it performed with the same flawless performance. In fact, we could do more than just detect the attack: we could actually stop it – all we had to do was to tell ebtables to drop unmatched ARP reply frames. Protecting against `denver`'s ARP+ICMP trick required further measures to correlate ARPs with ICMPs.

When implemented on a dedicated commercial switch, however, this algorithm may suffer from a limitation: since memory is scarce in this environment, the state table should be limited not to grow too much. In this case, it becomes easy to make ARP spoofing undetectable by hiding it amidst a flood of fake ARP requests, filling the state table up to the point that it can hold no more pending ARP exchanges and causing an avalanche of false positives – due to the legitimate requests not being able to get to the table and their legitimate replies being incorrectly being flagged as unmatched.

4.2 The Request-Reply Imbalance Algorithm

Because of that, we went in pursuit of an algorithm that used only constant-sized tables. We arrived at a solution inspired in the MIB-II packet counters available on SNMP-capable switches – for each port, the switch maintains four 32-bit counters:

- `WhoHasIn`: gets incremented every time an ARP who-has requests is received at that port;
- `WhoHasOut:` likewise, for requests being transmitted through that port;
- `IsAtIn:` counts ARP is-at replies entering that port
- `IsAtOut:` same thing, for replies going out.

Maintaining those counters can be very efficiently implemented in the switch with a couple of comparisons on protocol number field and ARP message type.

The switch makes those counters available over SNMP (we create a private arc for that). An external tool then polls the counters for all ports at regular intervals (we used five or ten seconds), calculating the deltas between the current sample and the previous. Finally, for each port *p*, we calculate the `ai[p]` ("arp imbalance") variable as follows:

```
ai[p] = ΔWhoHasOut[p] - ΔIsAtIn[p]
if ai[p] > 0 then
    ai[p] = ai[p] - ΔWhoHasIn[p]
```

If the ARP imbalance of a given port is greater than zero, it means that this port transmitted more requests than received replies within the last sampling interval, which is perfectly normal. On the other hand, if this quantity is negative, it means that the port has received more replies than sent requests – the telltale sign of an ARP spoofing attack –, so we alert the administrator.

The additional subtraction on the conditional is to prevent the attacker from fooling our detection scheme by simply issuing bogus ARP requests to balance the amount of replies. If she does that, the ARP imbalance will become even more negative.

This scheme detected nearly all attacks in our test suite, correctly pinpointing the port which the attacker was (this is different from SSSA, which reports the attackers' MAC and IP).

There was just one little glitch: some false positives were displayed when, due to the discrete sampling, we were unlucky enough to have a legitimate ARP request in one sample and its associated ARP reply in the next, making the ARP imbalance equal to -1 in the first sample and equal to one in the next. We fixed this by "integration": when `ai[p]=-1`, we wait until the next sample to decide if it is an attack or not – we add the current delta measurements with the previous (which is equivalent to doubling the sampling interval). If the `ai[p]<0` imbalance criterion still holds, we raise the alarm.

5 DETECTION ON SWITCHES VIA SNMP

Since it was not feasible for us to reprogram the switches firmwares to implement our detection system, the next-best was to try to detect ARP imbalance using other source of information. Our approach is to infer ARP imbalance based on packet and data flows through switch ports.

Counters for packets in/out and bytes in/out flowing through each switch port are provided by the SNMP management framework in the ubiquitous MIB-II [8]. Specifically, there is a table called `ifTable` providing the following counters:

- `IfInOctets`: bytes entering the port
- `IfOutOctets`: bytes leaving the port
- `IfInUcastPkts`: unicast packets entering the port
- `IfOutUcastPkts`: unicast packets leaving the port
- `IfInNUcastPkts`/`IfOutNUcastPkts`: the same as above, for non-unicast packets.

These counters grow monotonically until they wrap up when they hit the integer representation limit. We are actually interested in the amount of data exchanged during regular sampling intervals. So we take a snapshot at time $t(i)$ and subtract each corresponding variable from its previous value at time $t(i\text{-}1)$. This is the same collection strategy used by MRTG [17] and most SNMP-based NMSs (Network Management Systems, such as HP OpenView, etc), but much faster – our typical *sampling rate* is 5 or 10 seconds, while most collection tools use 5 minutes.

For sake of conciseness, we will represent these values as follows (*p* is the port number, the delta means the aforementioned subtraction):

- `io[p]` = Δ`IfInOctets[p]`
- `iu[p]` = Δ`IfInUCastPkt[p]`
- `in[p]` = Δ`IfInNUCastPkt[p]`
- `oo[p]` = Δ`IfOutOctets[p]`
- `ou[p]` = Δ`IfOutUCastPkt[p]`
- `on[p]` = Δ`IfOutNUCastPkt[p]`

These variables compute traffic for all protocols, not just ARP or even IP – so, we will be implicitly assuming our network runs no other protocol suites except IP. Our proposition is that we can infer a possible ARP imbalance simply by looking at these deltas in a particular way. For simplicity, the following discussion assumes that we have at most one host connected to each switch port (that is, we have no cascading). We will also assume that little else happens in the network but the attack. We will later remove both restrictions.

First, we can get rid of the relaying component of the attack by computing the *packet imbalance*:

```
pi[p] = iu[p] - ou[p]
```

This is because the attacker resends nearly the same amount of packets through the very port it received, so they nearly cancel out. They possibly don't cancel out exactly because of the discreeteness of the sampling.

The packet imbalance is obviously not the same as the ARP imbalance we defined in section 4.2. But the unsolicited ARP is-at packets the attacker issues during the poisoning component of the attack make this number positive: from the point of view of the switch, these packets are coming in the port; and they don't get cancelled because they are unreplied, as we saw earlier. This turns out to be one of the key points of our strategy: approximate the ARP imbalance by the *imbalance on small packets*. Recall that ARP packets are small: they are only 48 bytes long so they end up being padded to the minimum ethernet length. Adding the FCS, their final size on the wire is 64 bytes.

A similar idea allows us to compute and approximation of the average size of packets counted by `pi` (we call it "imbalance residual size" or simply "residue"):

```
rs[p] = abs((io[p]-oo[p])/pi[p])
```

Notice that this is not the same as the average packet length. Another subtle point is that `io[p]` and `oo[p]` include the amount of bytes transferred in *broadcast*

ESSAY		Samples		C3	
Tool	Appl	Total	Attack	Hits	%Hits
angst	ftp	21	5	4	80,0%
angst	telnet	31	18	13	72,2%
angst	tftp	20	5	5	100,0%
hunt	ftp	36	24	5	20,8%
hunt	telnet	39	27	22	81,5%
hunt	tftp	200	188	155	82,4%
denver	ftp	21	8	6	75,0%
denver	telnet	26	17	7	41,2%
denver	tftp	24	7	4	57,1%
none	ftp	21	0	0	0,0%
none	telnet	35	0	0	0,0%
none	tftp	29	0	0	0,0%
TOTALS		**503**	**299**	**221**	**73,9%**

Table 1: Results from our repeatable laboratory tests. "None" means that no attack was in progress – we measured this to see if any false positives showed up. "Total samples" means the amount of 5-second measurements performed; "Attack" counts in how many of them we had an attack in progress, as counted by the SSSA tool, which provides perfect detection. "Hits" counts how many samples the SNMP-based detection tool detected an attack using the C3 algorithm. Overall, in this controlled setting it displayed no false positives and 74% detection ratio, indicating the viability of using packet imbalance as a reasonable estimator for the ARP imbalance.

packets, and we are dividing just by the amount of residual *unicast* packets. We take the absolute value because signs will not matter for us later. If `pi[p]` is zero, we forcibly set `rs[p]` to zero.

Armed with this intution, we can now present our first detection algorithm (it was actually our third, so we call it "C3"; it was just the first to give good results):

1. **Compute imbalances:** for each operationally up ethernet port, compute `pi[p]` and `rs[p]` as discussed above; along with:
   `pf[p]=iu[p]+ou[p]`
   (This computes the "port flux", i.e., the total amount of unicast packets crossing the port either in and out.)
2. **Determine candidate attacker:** Find port *n* simultaneously satisfying:
   a) `pi[n] > 1`
      (Positive imbalance = issuing unreplied packets)
   b) `rs[n] < 65`
      (Residual packets are small)
   c) `pi[n]` is maximum
      (This is the most imbalanced emitter)
3. **Determine candidate victim:** find port `m` such that all conditions below are met:
   d) `pi[m] < -1`
      (Negative imbalance = receiving unreplied packets)
   e) `pi[m]` is minimum
      (This is the most imbalanced receiver)
4. **Verify if there is an attacker-victim coupling:** If there are such `n` and `m`, then `n` is the attacker port and `m` the victim's if and only if:
   f) `pi[n] >= -pi[m]`
      (The imbalance of the attacker should equal to or greater than this victim because there may be other victims.)
   g) `pf[n] > pf[m]`
      (The flux in the attacker port has to be necessarily bigger than the victim because it is both generating traffic and relaying the victim's traffic.)

We first tested this algorithm in a controlled laboratory environment: we built a network consisting of two victims (one Windows 2000, acting as "client", and a FreeBSD 4.5 acting as "server"), a Linux attacker and the detection station (also a Linux), each in different switch ports. We performed several test suites: one without an attack going on, as a "control group" and for measuring false positives (normal traffic incorrectly flagged as an attack); and others with the attacks going on, to see how many instances the attack would pass undetected in – that is, false negatives. In each essay, we tested a few applications representative of common traffic patterns: TELNET for interactive traffic, TFTP for stop-and-wait bulk transfer and FTP for pipe-filling bulk transfer. The test was run from an automated script, to ensure consistency and repeatability. Before each specific test, the script cleared each victim's ARP caches. The poisoning rate was 1 packet per second.

Table 1 summarizes the results we obtained for several popular attack tools. To allow an objective comparison with what would be the ideal detection ratio, we had all switch ports mirrored to another port where yet another machine was running the sniffer-based SSSA from section 3, running the Request-Reply Mismatch detection algorithm. We could detect all the attacks, correctly identifying the attacker port; in all instances, we had multiple detections in each test. The reported victim port sometimes jumped between one victim and the other, as expected.

Although not shown in Table 1, we also tested slower poisoning rates – 2, 5 and 10 seconds per spoofed ARP

packet. As we expected, the detection ratio fell as the attacks were quieter. What we didn't anticipate was that we could in many instances still detect the attacks even when the poisoning rate was slower than our sampling rate.

The low false positive rate – zero, in this laboratory, but it came as high as 5% in our production networks – is perhaps the most intuitively puzzling aspect: there seems to be no fundamental reason why ordinary traffic only rarely triggers our detection criterion. A rigorous analysis is beyond the scope of this paper, but we offer a few intuitive arguments:

- Most upper-layer network protocols are bidirectional, consisting mostly of request/reply unicast pairs that roughly balance themselves. Interactive traffic is a good example of this.
- Many protocols do send unreplied packets, but they are almost always broadcast. Exceptions, like some ICMP error messages are sporadic.
- Bulk transfers do generate sizable imbalances – especially protocols that try to fill the pipe, like TCP. However, this results in large residues – far larger than our threshold of 65 bytes.

In our experiments, minor changes to the algorithm, such as omitting step (f) or changing the imbalance thresholds from ±1 to zero in (a) and (d), promptly increased the false positive rate to unacceptable levels. As the reader may have guessed, we arrived at that particular formulation by an iterative process of gathering intuition from previous experiments, devising a new algorithm, trying it on the test network and comparing it against other competitors.

6 GENERALIZED ARCHITECTURE

Outfitted with the tools described in sections 3 and 5, we can propose a general architecture, represented schematically on figure 3:

- **Protected Detection Station and Dedicated Network:** Our detection machine lies in a protected management network, connected to the management VLAN (usually VLAN 1) of each switch. The MAC addresses of the switches' management IPs are statically set and it runs an operating system that doesn't allow static ARPs to be overridden (we used Linux). Each management port is also statically configured to allow only the detection machine's MAC address; and, if the switch supports static ARP, we also set the IP ⇔ MAC binding of the detection station statically. These precautions, although tedious to set up and maintain, should be viable because this network is much smaller then the main data network; and are imperative to prevent an attacker from using ARP spoofing – the very attack we want to detect – to DoS or confuse our detection system.
- **Detection tool and switch polling:** The SNMP-based detection tool runs on the master detection station, polling the switches for their counters through an specific interface connected to the protected network. The configuration file of the tool specifies which ports on the master switch are cascaded on smaller switches, so it can construct a large "flat" table pretending that all ports from all switches are just like a single switch with a huge number of ports. (This eliminates the aforementioned restriction about only one machine per switch port). The tool uses a notation port@swich_ip_address for port names, so as to keep track of the original port locations when reporting possible attackers.

  Alarms generated by this tool are "tentative"; they will be further screened by the false positive elimination scheme described below.
- **Dedicated Port Mirorring Network:** Each switch has a spare port connected to another network used for occasional port mirroring – used both for automated false positive mitigation (described below) and for packet captures performed manually by the administrators for troubleshooting or other purposes.
- **False positive elimination:** The port mirroring dedicated network is also connected to another port on the master detection station, where an instance of the SSSA tool listens continuously. Most of the time, it hears nothing, since port mirrorings are seldom enabled. When the SNMP-based tool detects a suspicious event, it reconfigures (with no manual intervention or notification) the relevant switch to enable port mirroring for that port for some time (we used 80 seconds). The dedicated SSSA then receives everything in this port and, if it detects an attack, it alerts the administrator.

  On our production networks, our typical raw false positive rate per day was less than 0,5% and our peak rates were around 5%. This may seem small, but it means nearly one false positive per hour, which is far too annoying. Filtering those events through SSSA for a "second opinion" solved the problem: we never heard of the false positives anymore, since SSSA provides near-perfect detection. An important detail is that we configured this SSSA to run the duplicate packet matching algorithm – it is too easy to miss a careful, silent attack if we pay attention only to the poisoning part; and, since it is enabled only rarely, its high memory consumption becomes a non-issue.
- **Sniffers on each hub:** Each hub has an extra port connected to a dedicated machine running an instance of SSSA (nicknamed "sentinels"), allowing it to detect any attack performed by any machine connected, either directly or indirectly through further cascading, to this hub. Alarms generated by those sentinels are "authoritative", in the sense that they get immediately reported to the administrators. The other interface of the machine is connected directly to the management network, relying on the same precaution of having static ARP addresses.

  Besides sending alarms when they detect attacks, those SSSAs are also polled by the main detector via SNMP for their MIB-like "switch simulation" traffic statistics – this is needed because for the C3

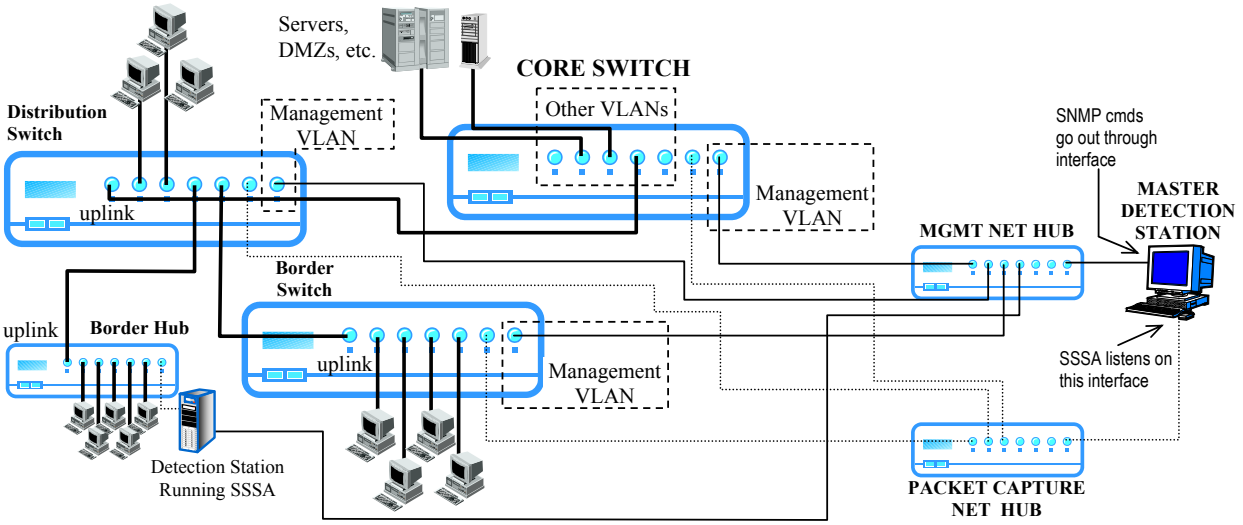

Figure 3: Example detection architecture with a core switch, two distribution/border switches and a border hub creating three logical networks. The main data network is shown with links represented by thick lines. This is where most of the servers, workstations, routers, etc., are., possibily organized in several VLANs and/or IP networks. The management network, with its links represented by solid thin lines, conencts the master detection station to the management port of all switches and relevant active elements for management tasks, mostly over SNMP. Whenever possible, devices in this network are configured with static ARP. On the detection station, our SNMP-based detector collects the traffic counters it needs through this network. It is also used to enable port mirroring on a specific switch when the detector suspects an attack is in progress. The packet capture network, represented by dotted lines, is used to collect traffic for further inspection by SSSA when the SNMP-based detector has enabled port mirroring. When this SSSA sucessfully flags an attack, administrators are notified through standard alarm mechanisms. On each hubbed segment, an instance of SSSA "sentinel" also runs to take advantage of the fact that is sees all packets through that part of the network, sending an alarm back to the master station through the management network when it sees an attack.

detection algorithm to work, it requires not only the victim or the attacker's traffic counters, but both, during the imbalance coupling test. As we said earlier, that was, in fact, the original motivation for writing the SSSA and the reason of its name; its specialized detection algorithms were introduced later.

Thus, our detection infrastructure, rather unsurprisingly, has to mirror the overall network architecture. Although it all seems complicated and perhaps a lot of work, the infrastructure is similar to what serious network management infrastructures usually look like. The dedicated packet capture network is also well suited to accommodate other IDS and diagnostic tools.

After assembling this infrastrcuture in our production networks, we invited some of our sysadmin colleagues to perform test attacks to see what we could accomplish. They were intentionally not made aware of the inner workings and capabilities of the detection system – we only gave them the draft of this paper after the main tests. For the record, we all had institutional permission to conduct the attacks and conditions were carefully controlled so that, although most tests were conducted on peak usage hours, our users never noticed anything unusual. Our results can be summarized as follows:

- All attacks which happened to be performed on a hubbed segment were promply detected by its SSSA "sentinel", even the most silent and careful ones. This was hardly unexpected, given what we discussed in section 3.1.
- Small DoSs went undetected: careful attacks where just one or two ARP frames were sent to perform only the poisoning part of the attack went undetected when they were performed on the switched parts of the network.
- Most attacks that tried to accomplish something useful from the point of view of an intruder, like capturing NetBIOS hashes, hijacking TELNET connections or performing the cryptographic SSH or SSL man-in-the-middle attack (naturally, using ARP spoofing as a first step) were detected even when performed on the switched part of the network. Only very fast attacks, lasting less than 30 seconds, went consistently undetected.

Admittedly, this is more of anectodal evidence than a rigorous conclusion. Although we did regard those results as successful, further fine tuning and field testing is certainly needed to declare it as generally viable in most network setups. Since at the time this paper was written we had this system working for just a few weeks, we didn't have any real malicious attack incidents to report.

We did find a rather easy way to remain undetected for reasonably long periods on the switched part of the network: hiding behind volume traffic – by purposefully generating large file transfers from the attacker station to other machines and adjusting the poisoning rate to one packet each twenty seconds or so, we could stay below the detection threshold. However, we couldn't manage to stay like this forever – after a few dozen minutes, overloads cause some mismatched ARP request to get caught by some sentinel SSSA and trigger the alarm.

7 CONCLUSIONS AND FUTURE WORK

We presented a review of the ARP spoofing attack strategies implemented by common attack tools and used this understanding to develop detection algorithms based on packet captures with sniffers. The resulting

tool ends up being the cornerstone point in our overall strategy, both for direct detection and false positive mitigation.

Since it is impractical to continuously capture packets directly from the switches, we developed an algorithm to infer potential attacks from traffic flow statistics provided by SNMP's MIB-II standard counters. We showed how ARP imbalance can be approximated by packet imbalance under quite general conditions, with reasonable false positive/false negative rates.

The ARP imbalance concept, in turn, was developed when trying to devise a detection algorithm with constant-sized storage requirements for implementation in memory-constrained switches. We explored an implementation with a Linux PC acting like a bridge (not that memory constrained) that could detect and even prevent the attacks. Given the ease of the implementation, its negligible impact on performance and the effectiveness of the protection, it seems surprising that no commercial dedicated switch we know of offer those features. Even if it didn't prevent the attack, the detection could be extremely facilitated if the switch made available in their MIBs the ARP-specific counters we mentioned.

Finally, we proposed an architecture combining both detection strategies – the sniffer-based and the SNMP-based to make a full-fledged detection system for the entire local network quite similar to traditional IDS architectures. We discussed how our initial experiments indicate that, for attempts at serious network compromise that use ARP spoofing as part of the attack, the intruder is often forced to make a severe enough "noise" as to be detected by our system.

Work on algorithms besides "C3" is already being undertaken, aiming to further lower its false positive/negative ratios. It remains to be seen whether they can be made low enough to dismiss the false positive mitigation strategy. We are also engaged in making a Snort [18] plugin version of SSSA.

This article, tools, test suites and follow ups can be found in our implementation site: http://www.freeicp.org/twiki/bin/view/Arpspoof.

8 ACKNOWLEDGEMENTS

We are indebted to Tempest Security Technologies and Recife Center for Advanced Systems and Studies for allowing us to conduct the "real world" tests in their networks. We also would like to thank João Rocha for lending his spare switch to perform the repeatable lab tests and Kenny Moreira, Finatec and Cisco Academy for allowing us to use their labs.

9 REFERENCES

1. Andrew S. Tanembaum, *Computer Networks, 4th Edition*, 2002, Prentice-Hall, ISBN 0130661023
2. James Antonakos, *Ethernet Technologies: Part 3 – Network Building Blocks*, 2002, http://www.circuitcellar.com/library/ccofeature/antonakos0502/c0502ts.pdf
3. Graham, R., *Sniffing (network wiretap, sniffer) FAQ*, 2000, http://www.robertgraham.com/pubs/sniffing-faq.html
4. Douglas C. Hewes, *Overcoming the Difficulties of Packet Capturing on a Switched Network*, 2003, SANS Institute, http://www.giac.org/practical/gsec/Douglas_Hewes_GSEC.pdf
5. Volubuev, Y., *ARP and ICMP redirection games*, 1997, http://www.insecure.org/sploits/arp.games.html
6. David C. Plummer, *RFC 826: An Ethernet Address Resolution Protocol*, http://www.rfc-editor.org/rfc/rfc826.txt
7. W. Richard Stevens, *TCP/IP Illustrated, Volume 1 – The Protocols*, 1994, Addison-Wesley, ISBN 0-201-63346-9
8. William Stallings, *SNMP, SNMPv2, SNMPv3, and RMON 1 and 2*, 3rd Edition, 1998, Addison-Wesley, 0201485346
9. Dug Song, *Dsniff package*, http://naughty.monkey.org/~dugsong/dsniff
10. Paul Krauz's, *Hunt Project*, http://lin.fsid.cvut.cz/~kra
11. Patroklos G. Argyroudis, *Angst-0.4b Active Sniffer*, http://angst.sourceforge.net
12. *Denver Project*, http://www.ilionsecurity.ch/denver
13. Arpwatch, ftp://ftp.ee.lbl.gov/arpwatch.tar.gz
14. Ettercap Homepage, http://ettercap.sourceforge.net
15. Ethernet Bridge Tables, http://ebtables.sourceforge.net
16. Netfilter/IPtables – Packet Filtering Subsystem for Linux 2.4, http://www.netfilter.org
17. Tobias Oetiker, *MRTG – Multi Router Traffic Grapher*, http://mrtg.hdl.com/mrtg.html
18. *Snort Open Source Network Intrusion Detection System*, http://www.snort.org