ARP Spoofing Detection And Network Trust Monitor

Mohamed Thowfiq A[1], Jawahar P S[2], NasreynAbinash A[3], Dr.N. SundaraRajulu[4]
[1, 2, 3, 4] Dept of Cybers ecurity
[1, 2, 3, 4] Dhanalakshmi Srinivasan University

Abstract- *ARP (Address Resolution Protocol) Spoofing is a very common and dangerous network attack. In this attack, a hacker sends fake ARP messages over a local area network to link their own MAC address with the IP address of a legitimate device, usually the default gateway or router. Once this is done, all the data meant for the internet goes through the attacker's computer first, allowing them to steal passwords and read private messages. For our mini-project, we developed a tool called ARPTM (ARP Spoofing Detection and Network Trust Monitor) using Python. Our main goal was to create a system that not only detects the attack but also stops it automatically. We designed a unique 'Trust Score' system out of 100 for every device on the network. The tool uses the Scapy library to read network packets in real-time. If a device acts suspiciously, like changing its MAC address or sending too many packets at once, its score drops. If the score goes below 60, our Python script automatically runs firewall commands to block the attacker and sends out healing packets to fix the network. Testing showed that our project detects and stops attacks in less than a second, making it very effective for securing local networks.*

Keywords: ARP Spoofing, Network Security, Trust Score, Python, Scapy, Man-in-the-Middle, Mini Project, Firewall Blocking.

I. INTRODUCTION

In today's digital world, Local Area Networks (LANs) are used everywhere, from our college labs to our homes and large companies. People spend a lot of money securing the outside of their networks with strong firewalls and antivirus software. However, the inside of the network is often left completely unprotected. If an attacker gets access to the local Wi-Fi, they can easily launch attacks against other users on the same network. One of the easiest and most dangerous internal attacks is ARP Spoofing.

To understand the problem, we first need to look at how computers talk to each other. When a computer wants to send data to another computer, it knows the IP address, but physical network switches need the physical MAC address to deliver the data. To find this MAC address, the computer sends an Address Resolution Protocol (ARP) request asking, 'Who has this IP address?' The device with that IP replies with its MAC address. This answer is saved in an 'ARP cache' to save time later.

The big problem with the ARP protocol, which was created way back in 1982, is that it has zero security. It operates on total trust. It does not verify if the reply is actually coming from the right person. An attacker can simply broadcast a fake ARP reply saying 'I am the router, here is my MAC address.' Every computer on the network will blindly believe this and update their cache. From that moment on, all traffic meant for the internet is sent directly to the attacker. This is known as a Man-in-the-Middle (MitM) attack.

While there are tools like Wireshark that can show you ARP packets, you have to sit there and watch the screen manually. If an attack happens while you are not looking, you won't know. Our team noticed that there was a lack of simple, free, and automatic tools that can run in the background to stop this. Therefore, for our mini-project, we decided to build ARPTM. We wanted to create a Python application that constantly watches the network, gives every device a 'Trust Score', and automatically blocks anyone trying to spoof the network, all without needing a human to click any buttons.

II. LITERATURE REVIEW

Before building our project, we researched what other people have done to stop ARP spoofing. We found that since this is an old vulnerability, many solutions have been proposed over the years, but most of them are not practical for normal users or students.

First, we looked at existing software tools. Tools like ARPwatch and XArp are widely used. ARPwatch is good at detecting changes, but it only sends an email alert. By the time the administrator reads the email, the attacker might have already stolen important passwords. XArp has a graphical interface, but it is a paid commercial software and does not automatically block the attacker on Windows. Wireshark is great for capturing packets, but it is a passive tool and cannot take action.

Next, we read research papers on cryptographic solutions. Some researchers proposed S-ARP (Secure ARP), which uses digital signatures to verify every ARP packet.

While this sounds perfect on paper, it requires updating the operating system and network settings on every single device in the network. This is impossible to implement in a college or public Wi-Fi network where anyone can bring their own laptop or phone.

We also explored recent academic papers. A very interesting paper we studied from ScienceDirect by Alsaaidah et al. [1] discussed using Deep Neural Networks (DNN) to detect ARP spoofing in Software-Defined Networks (SDNs). They achieved excellent accuracy using AI and dynamic cache management. However, we realized that running complex machine learning models requires powerful computers and central controllers, which standard home and college networks do not have.

Based on this literature survey, we identified a clear gap. Normal networks need a tool that is as smart as the modern research but as lightweight and easy to run as a simple Python script. Instead of using heavy AI, we decided to develop a deterministic, rule-based 'Trust Score' system. This would allow our project to be very fast, use almost zero CPU power, and still provide automated blocking capabilities.

III. SYSTEM REQUIREMENTS

Since the goal of this mini-project was to create a tool that anyone can run on a normal laptop, we made sure the system requirements were very basic.

A. Hardware Requirements

The hardware needed to run the ARPTM tool is standard. It can run on almost any modern computer:

• Processor: An Intel Core i3 or any basic processor is more than enough to read network packets.
• RAM: 4 GB of RAM is required. During our testing, the Python script used less than 100 MB of RAM, so it will not slow down the computer.
• Network Card: The most important hardware requirement is a Wi-Fi or Ethernet adapter that supports 'Promiscuous Mode'. This mode allows the network card to hear all the traffic floating around the local network, not just the traffic meant for our specific laptop.

B. Software Requirements

We built the entire project using open-source tools so it would be completely free to use.

• Python 3: The core programming language used for the logic, the GUI, and the packet processing.
• Scapy: This is a powerful Python library used for sniffing network packets and creating custom packets.
• Npcap: This is a Windows driver. By default, Windows does not let Python read raw network packets from the kernel. Installing Npcap allows Scapy to intercept the packets directly from the network interface.
• Tkinter and Colorama: Tkinter is used to create the visual dashboard, and Colorama is used to print colored text (like red for alerts and green for safe) in the command prompt.

To run the project properly, the user must run the command prompt or VS Code terminal as an Administrator, because blocking IP addresses requires root permissions.

IV. PROPOSED SYSTEM ARCHITECTURE

We divided our Python project into four main parts so the code would be easy to manage. These modules run at the same time using Python threading.

A. The Packet Sniffing Module

This is the ears of our project. It runs constantly in the background. When we start the script, it first sends out an ARP request to every possible IP on the local subnet (like 192.168.1.1 to 254). Whoever replies gets added to our 'Known Good' list. This gives us a safe baseline.

After the scan, the module uses Scapy's 'sniff()' function. We applied a filter so it only captures 'arp' packets. This is a very important optimization we learned during the project: if we capture all internet traffic (like YouTube videos and web browsing), Python will crash from overload. By filtering only ARP packets at the kernel level, the script stays extremely fast and lightweight.

B. The Trust Score Engine

This is the main brain of our mini-project. We wanted a way to judge devices fairly, so we came up with a score out of 100. Instead of just saying 'Yes it is an attack' or 'No it is not', the score changes dynamically based on how the device behaves.

The logic we programmed works like this: when a new device joins the Wi-Fi, we give it a full score of 100. Every time it sends a normal, safe ARP packet, we give it +1 point (up to a maximum of 100). This rewards good behavior.

If a device starts misbehaving, we apply penalties. Sometimes, attackers use tools that blast hundreds of ARP requests per second to overload switches. We wrote a function with a sliding 1-second window. If a device sends more than 20 packets in one second, it is flagged for an 'ARP Flood', and we deduct 10 points. It serves as a warning.

The biggest penalty is for MAC address changes. If our database knows that the router's IP (192.168.1.1) has the MAC address AA:BB:CC:11:22:33, and suddenly a packet arrives claiming that 192.168.1.1 is now at DD:EE:FF:44:55:66, this is a clear spoofing attempt. Our script immediately deducts 50 points from the attacker's score.

We set a hard threshold at a score of 60. If any device's score drops below 60, it is officially marked as 'Dangerous' and sent to the Mitigation Engine.

Trust Score Logic Example:

```
If New_MAC is different from Saved_MAC:
   Trust_Score = Trust_Score - 50
   Print Alert
If Packets_Per_Second > 20:
   Trust_Score = Trust_Score - 10
If Trust_Score < 60:
   Start Auto-Block function
```

C. Mitigation and Network Healing

Detecting the attack is only half the battle. To complete our project objectives, we wrote an automated defense mechanism. When a score drops below 60, our Python script uses the 'subprocess' module to talk directly to the Windows operating system. It generates a command using 'netsh advfirewall' to create a firewall rule that permanently drops all traffic coming from the attacker's MAC address.

However, we realized during testing that even after we blocked the attacker, our computer's ARP cache was still poisoned, meaning our internet was still broken. To fix this, we added a 'Healing' feature. After blocking the attacker, our script looks up the correct MAC address of the router from our safe baseline. It then uses Scapy to craft a new, correct ARP reply and broadcasts it to the entire network three times. This forces our computer and all other computers to update their cache with the true MAC address, fixing the internet connection immediately.

D. Graphical Dashboard

To make the project look professional, we built a GUI using Tkinter. Since the packet sniffing uses a lot of processing, we didn't want the GUI to freeze. We solved this by having the background script write the current scores to a JSON file. The Tkinter GUI reads this file every 2 seconds and updates the screen. It shows a table of all connected devices, a live graph of their trust scores, and a colorful alert log. We also added an 'Export to CSV' button so users can save the attack logs for their records.

V. TESTING AND EXPERIMENTAL SETUP

To prove that our mini-project works in real life, we set up a controlled test environment. We did not want to test this on the college network to avoid causing issues, so we used a private mobile hotspot and two laptops.

The first laptop was the 'Victim'. This was a Windows 11 machine running our ARPTM Python script. The second laptop was the 'Attacker'. We installed Kali Linux on a virtual machine and wrote a separate Python attack script (arp_attacker_demo.py) using Scapy to generate fake packets. The gateway was the mobile hotspot router.

We performed three main tests to check our logic.

Test 1: Single Spoof Attack. We fired one single fake packet claiming to be the router. Our script instantly caught the MAC address mismatch, dropped the score by 50 (from 100 to 50), and executed the Windows firewall block. The whole process took less than a second.

Test 2: ARP Flood. We used a 'for' loop in our attacker script to blast 50 random ARP requests in one second. The Trust Engine correctly calculated the rate, issued a warning, and dropped the score by 10 points. As the flood continued, the score kept dropping by 10 until it hit the 60 threshold, at which point the attacker was blocked.

Test 3: The Whitelist. We added a known MAC address to our whitelist.txt file. When that device sent fake packets, the system logged the event but did not block the device. This proves our whitelist feature works perfectly for trusted administrators.

VI. RESULTS AND DISCUSSION

The results of our testing were highly successful. The main goal of this mini-project was to create an automatic defense system, and the ARPTM tool proved that it can detect and block attackers much faster than a human ever could.

During our measurements, we found that it took an average of only 210 milliseconds to detect a spoofing attack and apply the firewall rule. This is incredibly fast. Most hacking tools like Ettercap need to send several packets over a few seconds to fully hijack a session. Because our script reacts in milliseconds, the attacker is blocked before they can steal any useful data.

We were also very happy with the system performance. One of our concerns was that sniffing packets constantly would make the laptop slow or drain the battery. However, by using the kernel-level ARP filter, our Python script only used about 0.5% of the CPU during normal monitoring. Even when an attack was happening and the GUI was flashing red alerts, the CPU usage only went up to about 8.7%, and RAM usage stayed under 80 MB. This means users can leave this script running in the background all day while they do their normal college work or browsing without noticing any slowdown.

The Trust Score system was the best part of the project. It completely stopped 'false positives'. Sometimes, regular network glitches happen. In older tools, a glitch might cause an annoying popup alert. In our system, a small glitch might drop the score from 100 to 90, but as soon as the device sends normal packets again, it earns its points back and goes back to 100. It only blocks devices that are truly acting malicious and crossing the 60 threshold.

VII. CONCLUSION AND FUTURE SCOPE

Through this mini-project, we successfully learned about network vulnerabilities and how to program real-world security tools. We proved that you do not need expensive enterprise hardware to secure a local network. Using basic Python libraries like Scapy, we built a complete Intrusion Detection and Prevention System (IDPS) that works automatically.

The ARPTM tool successfully bridges the gap between passive monitoring and active defense. By implementing the out-of-100 Trust Score logic, we created a fair and dynamic way to judge network devices. The addition of the automated firewall rules and the gratuitous healing packets means that the network can defend and repair itself without any human intervention.

For future work, there are several ways this project could be expanded. First, we could deploy the Python script on a small, cheap computer like a Raspberry Pi. This Pi could be plugged into a home router to protect the entire house 24/7. Second, as the internet moves away from IPv4, we need to update the tool to monitor IPv6 traffic. In IPv6, ARP is replaced by the Neighbor Discovery Protocol (NDP), which has the exact same security flaws. Adding NDP spoofing detection would make the tool future-proof. Overall, this project was a great success and provided us with deep practical knowledge of cybersecurity and Python programming.

REFERENCES

[1] A. Alsaaidah et al., "Enhancing security in Software-Defined Networks: An approach to efficient ARP spoofing attacks detection and mitigation," Telematics and Informatics Reports, vol. 14, p. 100129, 2024. [Online]. Available: ScienceDirect.

[2] D. Plummer, "RFC 826: An Ethernet Address Resolution Protocol," Internet Engineering Task Force (IETF), November 1982.

[3] S. Whalen, "An Introduction to ARP Spoofing," Node99 Research Group, April 2001.

[4] V. Ramachandran and S. Nandi, "Detecting ARP Spoofing: An Active Technique," Proceedings of the International Conference on Information Systems Security (ICISS), pp. 239-250, 2005.

[5] D. Bruschi, A. Ornaghi and E. Rosti, "S-ARP: A Secure Address Resolution Protocol," Proceedings of the 19th Annual Computer Security Applications Conference (ACSAC), pp. 66-74, 2003.

[6] W. Lootah, W. Enck and P. McDaniel, "TARP: Ticket-based Address Resolution Protocol," Computer Networks, Elsevier, vol. 51, no. 15, pp. 4322-4337, 2007.

[7] M. Yaibuates and S. Chaisiri, "ARP Poisoning Attack Detection with DHCP Server," Proceedings of the 2nd International Conference on Information Technology (ICIT), pp. 182-186, 2013.

[8] Scapy Documentation, Packet Manipulation in Python, Version 2.5, 2023. [Online]. Available: https://scapy.readthedocs.io