Level 2.1 - Detect a network scan

Attackers can use network scans to find devices in the network.
A specific kind of these scans is an **ARP scan** - send ARPs trying to resolve all network; existing hosts will respond

Password

`wp9s5ku3hg`

Instructions

> **Note:** If you are using the starter template provided, remember to update the filter on the `sniff()` function in `main()` to `filter="ip or arp"`

1. Create a new module `thresholds.py` with a class `Threshold`.
    - Its purpose is to detect whenever there are more than a certain number of packets (that match a certain filter) in a time window
    - We'll also keep this class as generic as possible, so it can be used to detect other scans

2. First, it needs a constructor (`__init__`), which takes the follow arguments:
    - `count`: Amount of packets that would trigger an alert
    - `window`: Window size in seconds
    - `group_key`: A function that receives a packet and returns a `group identifier`, used to associate each packet to a group.
        - Simple example: we have 20 red apples, 2 oranges, and 3 strawberries.
            - If we use `colour` as a group identifier, the results will be - red: 23, orange: 2
        - In an ARP scan, we want to group packets by source MAC address - if multiple machines are sending ARP requests, we want to maintain 1 tracking window for each machine
        - In this case, the groupk*ey function could be `lambda p: p[Ether].src`. This just simply returns each packet's source MAC address*
    - *`unique_key`: A function that receives a packet and returns a "unique value".*
        - *This unique value determines if the packet enters the tracking window - to enter the window, this value must not match any other "unique keys" of packets in the window.*
        - *For example, for detecting ARP scan, multiple ARP requests resolving the same IP should only count as one packet, because it isn't an ARP scan if the attacker only tries to discover one IP.*
        - *Therefore, we only want packets to enter the window if they're resolving an IP address that we haven't seen in another packet within the window*
        - *In this case, the unique*key function could be `lambda p:p[ARP].pdst`. This returns the destination of each ARP packet

3. In the `init` function, we will also need some way of tracking each source MAC address, and its associated tracking window
    - A dict will work here, using each source MAC address as a key
    - The value for each key should be initialised to an empty list. We will use this list as our 'tracking window' for each MAC, and add packets to it as they arrive
        - e.g., `{<group identifier> : []}`
        - Bonus: since we're adding packets sequentially (i.e., the first packet of the list will be the oldest, and the last packet the newest), there is another data structure that's more appropriate than a list
4. We'll also need to think about how to store each packet in the list
    - We don't need the full contents of each packet to detect a scan, just certain information
    - We'll need the time, so that we can check if the packet is older than our time window (and remove it if it has expired)
    - We'll also need the destination IP (aka, unique key) so that we can count the number of distinct ARP destination IPs
    - Each packet can be stored as a dict: `{'time': datetime.now(), 'key': self.unique_key(packet)}`
5. Next, we only want to maintain packets are in the window, so we need a function `remove_outdated`
    - This should remove any packets that are older than the threshold's window
    - Use the `time` library to check for packets that have expired
    - Also, we should remove any tracked source MAC addresses that haven't recieved any new ARP packets within the window time (tracking list is empty)

6. Last, add a `is_exceeded(self, packet)`
    - This method should handle check if each packet passes the uniqueness test (see `unique_key`). Add it to the tracking window if it does
        - Before adding a packet, remember to remove outdated packets (e.g. if the `window` is 3 seconds, all packets that arrived more than 3 seconds ago should be discarded)
    - It should also check if the number of packets in the tracking window has exceeded the count. Return `True` if it has.
7. In the main module, create one global `Threshold` object that will track ARP requests.
    - Every time an ARP request is captured, pass it to `is_exceeded`, and if it returns True, print an alert; for example: `*ALERT* ARP scan from 00:0c:29:8f:21:14`

Notes

GUIDING QUESTIONS

Use the guiding questions below to identify the malicious trigger and structure your code's detection:

- What do I need to extract from the ARP packet?
- How can I implement the sliding window algorithm?
    - What do I need to keep track of and log inside my window?
    - What data structures (e.g. lists, dictionaries, classes) can I use to implement my window so that it is easy to read and use?
    - When should I check if the window has exceeded?
    - When should I check if the logged packets are outdated?
- What should the values of my `count` and `window` be?

To submit

Submit a ZIP `nids.zip` containing all `.py` files.

Sentinel