Fixing the IoT Inspector for MDNS, SSDP, and UPNP

Leo Leaipailote '24

1 INTRODUCTION

The IoT Inspector is an open-source application developed by a team of researchers from various universities that aims to detect IoT (Internet of Things) devices connected to a local network for cybersecurity purposes. However, in the IoT Inspector's current state, there exists a major flaw in its functionality: MDNS SSDP and UPNP devices, which make up a large portion of IoT devices, are not being properly identified. After analyzing the source code of the IoT Inspector, the root cause of this error was found in a python file labeled 'netdiscowrapper.py' that is supposed to be responsible for extracting naming information about devices on a network. The reason behind this is that the 'netdisco' library, which 'netdiscowrapper.py' utilizes is deprecated. To address this problem, we first identified the underlying issue through experimentation with a raspberry pi and a local router. We then modified the source code in netdisco accordingly in a way that made UPNP, MDNS, and SSDP devices discoverable to the inspector. In addition to this, we collected important characteristics of each device such as the IP address, which was outputted in a csv file that contains all relevant information, ready to be processed for future use.

2 METHODOLOGY

In the following section, we first summarize the functionality of NetDisco to provide a more comprehensive understanding of how it behaves and identify the underlying issue. We then describe our solution for tackling MDNS and SSDP/UPNP devices.

2.1 Summary of Functionality and Underlying Issue

Once we realized that the netdisco library was deprecated, our next task was to identify the parts of netdisco that malfunctioned with hopes of repairing these issues to restore functionality. There are two notable calls to the netdisco library in netdiscowrapper.py. Both are located in the '_run_netdisco' function and are called 'netdis.scan()' and 'netdis.discover()'. 'Netdis' itself is a NetworkDiscovery object which is instantiated in the '_start_thread' function. The NetworkDiscovery class is defined in the netdisco library and located in a file named "discovery.py". The problem here is that while netdis.discover() should return a list of all discovered devices, it continues to return an empty array despite there being devices connected to the network that should be detected. We experimented with this by setting up a raspberry pi which was configured to be categorized as both a UPNP and MDNS device and connected to a private router that we set up.

In the constructor for a NetworkDiscovery object, we can see that it creates an empty MDNS and SSDP object. When the scan() function in discovery.py is called, it instantiates both an MDNS and SSDP object, where it then calls mdns.start() and ssdp.scan(). The MDNS and SSDP classes are defined in mdns.py and ssdp.py respectively, both of which are found in the netdisco library. The mdns.start() and ssdp.scan() both handle device detection for their respective types, and through the use of print statements we found that these functions are implemented correctly. It should be noted that in addition to calling ssdp.scan()

and mdns.start() in the scan() function, it also calls _load_device_support() located in discovery.py. This function populates a dictionary called self.discoverables (a dictionary that is initially empty and created in the NetworkDiscovery constructor) with a set of devices that is used further down in the pipeline; this is important for understanding why netdisco fails to meet our expectations which will be discussed in the next section.

While the scanning functions defined in netdisco are fully functional, the crux of the issue is located in the discovery() function. This function contains a for loop that iterates through key-value pairs of the self.discoverables dictionary with the device name acting as the key and a checker object acting as the value. Within the for loop there is an if statement that checks whether checker.is_discovered() returns true; if it does, then the checker is marked as found, and it will be appended to an array. Once the for loop finishes executing, this array containing all the found devices is returned. However, the reason that an empty array is always returned in our tests despite having an existing UPNP and MDNS device connected to the network is that the checker is never marked as found. Through the use of print statements, we found that the checker object is an instance of one of the classes defined in the discoverables directory located in netdisco's source code. The discoverables folder contains device templates for 5 MDNS devices (bluesound, bose soundtouch, enigma2 servers, LG smart devices, and SABnzbd servers), and 3 SSDP devices (frontier silicon, openhome, and Yamaha receivers). The 5 MDNS device templates are child classes of the MdnsDiscoverable class and all 3 SSDP device templates are child classes of the SSDPDiscoverable class, both of which themselves inherit functionality from the BaseDiscoverable parent class. BaseDiscoverable, SSDPDiscoverable and MDNSDiscoverable are defined in '__init__.py' in the discoverables folder.

At the point where the discover() function is called, the self.discoverables dictionary is populated with the device names of each of the 8 respective device templates contained in the discoverables directory (such as 'bluesound' or 'frontier_silicon', while the checker is an instance of the discoverable class associated with that particular device (the corresponding value pair for the key labeled 'frontier_silicon' would be a checker object that is an instance of the SSDPDiscoverable class for frontier silicon defined in 'frontier_silicon.py'). BaseDiscoverable contains the is_discovered() function that is being called on the checker in discover(). This function itself simply returns whether the array returned by another function called get_entries() is non-empty. The get_entries() function is only truly implemented individually in each of the 8 device template child classes, however, their functionality across all 8 are almost identical. In SSDP devices such as frontier silicon, the get_entries() function loops through an array of entries that is found by running SSDP.scan() and searches for an identification tag (in frontier silicon's case this tag is 'urn:schemas-frontier-silicon-com'), if the tag is found, get_entries will append the entry into an array, which will be returned on completion of the for loop. This is essentially what occurs in MDNS devices as well, with the only modification being that the entries set it loops over is found through MDNS.start() rather than SSDP.scan().

Therefore, netdisco only searches for certain IoT devices which it contains a template for such as Bluesound or Frontier Silicon, resulting in it being unable to identify other IoT devices, and making it less than ideal for the purposes of the IoT Inspector. While we were unable to verify if netdisco can even truly detect these devices due to us lacking possession of them, we were able to implement a general solution for all IoT devices which would render this point irrelevant. Our general solution differed for SSDP and MDNS devices, and the details for our implementation will be described in the next section.

2.2 Solutions

In the following subsection we discuss our implementation for a general solution which involves two separate components: a general solution for SSDP devices and another general solution for MDNS devices.

2.2.1 SSDP/UPNP Solution

In order to create a general solution for SSDP devices we realized that we had to make a departure from the specific identification method found in the get_entries() function. We discovered that the ssdp.scan() function produced a large amount of information expressed as strings. From here we identified a common substring that would be found in all UPNP devices, this being 'urn:schemas-upnp-org:device'. While this string may slightly differ for SSDP devices, identifying such a tag would most likely create a general solution for SSDP devices as well. Due to not having access to a SSDP device we could not confirm this, however, seeing as how UPNP and SSDP are very closely related and how both classes are defined in ssdp.py, we can assume with reasonable certainty that this is the case. The implementation of this particular aspect of the solution can be left for future work.

From here, we created a child class that inherited functionality from SSDPDiscoverable which was defined in a python file we created called general_device.py and redefined the get_entries() function by changing the identification tag to 'urn:schemas-upnp-org:device'. After thorough testing, we found that the raspberry pi was now being appended to the results array returned by discover() in the form of a general_device object. In addition to this, we added code in the _run_netdisco function located in the netdiscowrapper.py file which called the get_info function in discover() to obtain relevant information such as the IP address, model name, model number, and manufacturer. We wrote this information to a csv file which could then be used for clustering in the future.

2.2.2 MDNS Solution

The MDNS solution proved to be slightly more challenging. This is because there is no easily identifiable substring that is common to all MDNS devices, unlike for SSDP/UPNP. Additionally, inserting code at any point in the netdisco pipeline proved to be an arduous task that left room for many errors due to its complexity. As a result, we decided to create a function in the netdiscowrapper.py file that would handle MDNS device identification directly in order to circumvent the tedious nature of this work. We recognized that MDNS.start(), which handled the scanning process of MDNS devices, used a library called ZeroConf. By importing this library into netdiscowrapper.py, we were able to create a function called discover_mdns_devices() which used the service browser and listener objects defined in ZeroConf to obtain the device type and IP address of MDNS devices. More specifically, we iterated over an array we created that contained the most common MDNS device types such as '_http._tcp.local.', and used the listener and service browser to search for devices on the network that matched the given types. One key aspect of this solution is that execution of the program must be suspended for a short time in order for the listener to finish detecting devices; we accomplished this by calling ‘time.sleep(1)’.

Similar to the SSDP solution, we then wrote the information obtained from the listener including the device type, device name, and IP address to a csv file for future analysis. Ideally, the device type array we iterated over should contain an exhaustive list of all possible MDNS device types, but due to time constraints we were unable to find a simple way to do so. However, using the ZeroConf library, this should be possible and is unlikely to be difficult to implement; therefore, it is an area that should be explored in future work.

3 FUTURE WORK

Aside from finding a general identity tag for SSDP devices, and finding an exhaustive list of MDNS device types that will complete the work on device detection previously mentioned, future work should be conducted on the data processing aspect in order to further develop the IoT Inspector. Firstly, developing a clustering algorithm using unsupervised machine learning on the data we obtained from device detection can be useful in identifying commonalities across devices. For example, if the clustering algorithm determines that the data can be clustered based on type, this implies that the device type, such as Google devices or Amazon Echos, will be significant for grouping and categorizing new devices in the future. However, if the algorithm creates clusters based on some other measure, this would suggest that our efforts should be directed towards exploring those areas instead. Once this has been established, the next step would be to train a classification model using machine learning that could group future devices by the appropriate fields, which would aid in the analysis process.

4 DISCUSSION

From working closely with NetDisco over an extended period of time, I have concluded that NetDisco uses an inefficient process for identifying devices. The pipeline is too convoluted, making it difficult to debug and the process can be simplified greatly as proven through our MDNS solution. Looking holistically at our work, it may have been simpler to create our own implementation from scratch rather than trying to modify NetDisco. As a result, it is likely that for future considerations, we may return to our SSDP solution and extract the key components into a separate file or function in order to minimize the debugging complexities and flexibility issues of the NetDisco source code.

5 CONCLUSION

We were able to create a working solution for IoT device detection after identifying the issue which was related to a deprecated library called NetDisco: a key component of the IoT Inspector app’s functionality. However, additional testing and modification should be conducted to ensure that the solution is working for a larger number and variety of devices. Furthermore, clustering and classification algorithms should be developed to fully utilize the dataset obtained from our work.

References

Huang, Danny et al. “IoT Inspector: Crowdsourcing Labeled Network Traffic from Smart Home Devices at Scale”
https://iotinspector.org/papers/ubicomp-20.pdf