Regulation of IoT Device Communications Using MUD Files and IPTables

Gabriel Brown

(gb2582@columbia.edu)

Fall 2019

Code: https://gitlab.com/columbia.irt/riot/tree/master/Fall2019/CombineRouter

Columbia Internet Real-Time Lab

Index

1. Abstract

Due to IoT devices having limited computational power, being intended for non-technical users, and usually being cheaply made products, they are difficult to secure against cyber-attack. As the number of IoT devices in the world continues to grow, IoT botnets are becoming a larger and larger issue, threatening internet infrastructure. A potential solution to this threat is to limit IoT communications only to approved endpoints. To implement this technique, one needs the set of valid endpoints for each IoT device, as well as a program to run on every home router that will implement a firewall for each connected IoT device. This paper introduces a high-level overview of system to do this, and details my work on the home router program.

2. Introduction

2.a Background

In the past decade, many of the devices being connected to the internet are not personal computers but devices embedded with internet connectivity. This class of devices has generally been described as *The Internet of Things* (IoT) and has brought with it new security and privacy risks. While the term IoT has a potentially broad scope, I will use the term here to refer to consumer-oriented devices.

IoT devices have different security risks than general purpose devices. IoT devices generally have limited computational power, and thus lack to resources to implement security features that would be standard on general purpose devices. Furthermore, IoT devices are usually intended for non-technical users; for this reason, they are built to operate with little to no configuration from the user. This means that devices often never change the default password. Finally, many IoT devices are cheaply made products, which often results in devices being shipped with vulnerable software that won't be patched, either because there is no way to update the software or because there isn't sufficient economic incentive to do so. For these three reasons, IoT devices are much more vulnerable to security threats than general purpose devices. Furthermore, for these reasons, techniques that are effective for securing general purpose devices are not effective for securing IoT devices.

In addition to compromising consumer privacy and safety, IoT devices infected with malware pose a threat to internet infrastructure. There are hundreds of millions of IoT devices in the world, and the vulnerability of these devices has allowed large botnets of infected devices to emerge. For example, the Mirai Botnet, a botnet consisting of infected Linux IoT devices, had sufficient power to launch successful DDoS attacks against a DNS service provider in 2016, leaving major websites inaccessible for hours on the east coast. The number of IoT devices in the world is only growing, thus it is a pressing issue that this class of devices be secured.

While traditional techniques are not effective for securing IoT devices, IoT devices have a defining quality that we can take advantage of to make them more secure. IoT devices all have a limited set of intended uses, and therefore have a limited set of endpoints that they need to communicate with over the network. Thus, we can limit the endpoints that an IoT device is allowed to communicate with over the network without limiting the functionality of the device.

Based on this idea, the Internet Engineering Task Force (IETF) released a new internet standard in March 2019, RFC 8520, standardizing the format of a Manufacturer Usage Description (MUD) file that could be created by IoT device manufacturers for each of their products. A MUD file for a given IoT device contains a set of endpoints that the IoT device should be able to receive communication from, as well as a set of endpoints that the IoT device should be able to send communication to. This set of network communication rules can then be automatically be turned into firewall rules on the local area network. In theory, a manufacturer would host MUD files for their devices on a server, and when one of their devices connected to the internet, the device would include a URL to its MUD file in option 161 of its DHCP discover packet. The router would then download the IoT device's MUD file and implement the communication rules as a firewall. This system, if implemented correctly, could reduce the attack surface of IoT devices by a huge margin.

Thus far, however, almost no IoT device manufacturers are generating or hosting MUD files in accordance with RFC 8520. Even if that were to change today, there would still be hundreds of millions of existing devices that do not have MUD files. This project aims to secure IoT devices both with and without MUD files by limiting their endpoints to ones necessary for device functionality.

2.a High Level Overview of System Architecture

2.a.i Router Script

At a high level, the project has two parts. The first part is a script that runs on a home router. It is this script's job to detect if a new IoT device has joined the network, and to limit the endpoints a new IoT device is allowed to connect to. The router script detects new devices by monitoring network traffic on the router. When the router script detects a DHCP Discover packet, it determines if the packet originated from an IoT device or a general-purpose device attempting to connect to the internet. If it is an IoT device, the script enters its main function flow outlined in figure 1.

First, the script checks if the IoT device has a MUD URL in option 161 of the DHCP packet. If so, it downloads the MUD file from the manufacturer server, and converts the allowed endpoint list in the MUD file to a set of IPTables rules. By using IPTables, the router can limit the set of endpoints that a given IoT device can communicate with.

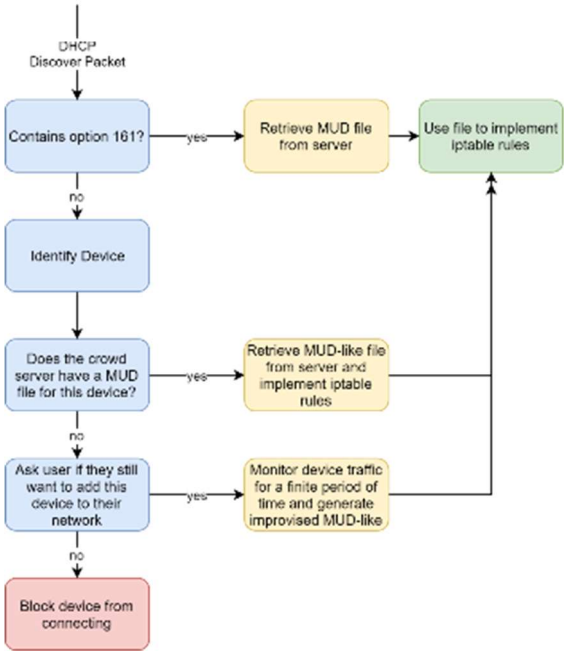

Figure 1: a high-level overview of the router script

If the IoT device's DHCP Discover packet does not have a MUD URL, then more work must be done to get a set of allowed endpoints for the device. This brings us to the second major part of the project.

2.a.ii MUD-like Crowd-Source Server

Generating the set of all legitimate endpoints for an IoT device without access to the devices source code requires a computer networking testbed and an extended period of time to monitor the device (usually 24 hours, though sometimes more). Thus, it is possible for computer networking labs to generate MUD-like files for IoT devices even if the manufacturer has not done so. If a lab were to generate a MUD-like file for every IoT device on the market, and host them on a server, then the router script, upon identifying the connecting IoT device, could download the MUD-like file for that device, even if it didn't have a MUD URL in its option 161. This would provide a way to secure IoT devices without a manufacturer-provided MUD.

However, this idea is naïve for a couple of reasons. First, there are thousands of models of IoT devices, each requiring its own MUD-like file. It is unrealistic to expect a single lab to invest the time and energy required to generate MUD-like files for every IoT device on the market. Second, this MUD-like server would, in theory, would be utilized every time an IoT device connected to a new network, and so it is unrealistic to expect a single university server to handle all of the traffic.

A less naïve version of this idea is to create a crowd-sourced version of this MUD-like server. Instead of asking a single lab or university to carry the entire burden of MUD-like file generation and hosting, a number of participating organizations would opt to host a node of a crowd-

source server. Each node acts as a server within the distributed MUD-like server, sharing the computational burden of hosting the MUD-like files. Furthermore, each organization could contribute new MUD-like files to the collective by adding them to their node. While there are many IoT device models, there are few enough that it is feasible that a cooperative effort between a number of labs and universities could yield MUD-like files for the majority of popular IoT device models. This would allow the majority of IoT devices currently in use to be secured against cyber-attacks.

In the remainder of this report, I will explain my personal contributions to this project in greater technical detail. My work has focused on the building and testing of the router script. Development of the MUD-like crowd-source server is occurring in parallel to this work.

3. Router Script

3.a Introduction

I inherited a router script from a previous semesters' students. I tested this iteration of the router script by writing a program for a raspberry pi that imitated an IoT device, connecting to the router and generating network traffic. A discussion of this test tool is in section 4a.

While the router script briefly functioned correctly when the router's IPTables were flushed, and the internal database for the script was reset, anything other than this carefully controlled scenario resulted in the malfunctioning of the program. Specifically, I encountered three major problems with the script:

a. After about three minutes, the router script would crash
b. In some scenarios, the router script would allow the IoT Traffic Generator to connect to forbidden endpoints
c. In other scenarios, the router script would forbid the IoT Traffic Generator from connecting to valid endpoints

Furthermore, stylistically the code was a mess – there was little to no separation of functionality in the code, and so modifying a feature usually required modifications throughout the program. For these reasons, I rewrote the entire router script, keeping the function flow of the program intact, while completely changing the internal structure of the code to something more robust and maintainable. This section will include a full technical discussion of all of the new features that I have written in the router script to resolve the three problems above.

3.b Program Crashing – Memory Leak

Relevant files: combinedsniffer.py (https://gitlab.com/columbia.irt/riot/blob/master/Fall2019/CombineRouter/combinedsniffer.py)

The solution for the router script crashing was simple. The script uses Scapy's sniff function to inspect network traffic on the router. By default, the sniff function saves every packet to program memory. After about three minutes in the old script, this would cause the program memory to exceed the available memory on the router, and the operating system would

terminate the program. There is no reason this program needs to save every packet to memory, so we can fix this problem by setting the store parameter to 0. This is what the old line of code looks like:

sniff(prn=packetHandler.sniff)

We simply add sniff=0 as a parameter. Here is the new line:

sniff(prn=packetHandler.sniff, sniff=0)

After this change, I mapped the memory footprint of the process over time and found that it converged – this suggests that there are no other memory leaks. This is important because this script should be able to run for upwards of a month without needing to be restarted.

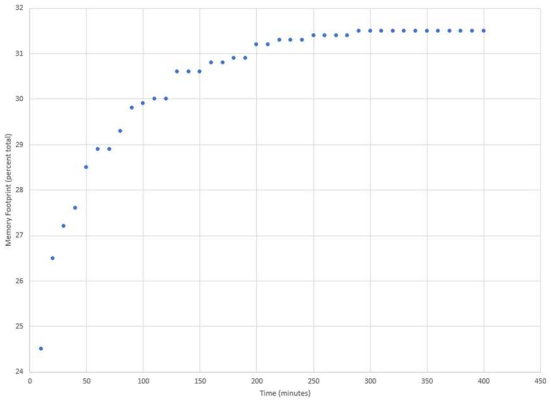

Figure 2: memory footprint of new router script over time

3.c Allowing Forbidden Endpoints

Relevant files: PacketHandler.py, FirewallManager.py, DatabaseManager.py, DeviceChain.py

The old router script allowed forbidden endpoints when a previously connected IoT device reconnected after the router was rebooted. The source of this problem was a discontinuity between the IPTables rules and the router script's database that recorded the MAC address of known devices. When an IoT device connected to the router for the first time, the script would download the device's MUD file and generate a set of IPTable rules. It would then add the MAC address of the IoT device to an SQLite database. This database was meant to keep track of known devices. When a known device (a device with a MAC address in the database) reconnected to the router, it was assumed that this device already had an existing set of IPTables rules to regulate traffic; thus, the script did nothing. The problem with this assumption was that after the router rebooted, the IPTables were automatically flushed. After rebooting the router, the IPTables rules for known devices should have been reloaded, but they were not.

I solved this problem with an object-oriented approach. I separated functionality into packet processing, firewall management, IPTables chain management, and database management. Packet processing controls high-level behavior, while firewall management, database

management and IPTables chain management handle the implementation details. I'll take a top-down approach to explaining these objects.

3.c.i PacketHandler

The PacketHandler object is responsible for processing every sniffed packet and making high-level decisions. PacketHandler's sniff function is called every time Scapy sniffs a packet on the router. When the PacketHandler detects a DHCP Discover packet, it first checks if the device is a known device by calling the FirewallManager function isMACRegistered. If the MAC has already been registered, then no further work is necessary, but if the MAC has not been registered, then the PacketHandler goes about registering the new device.

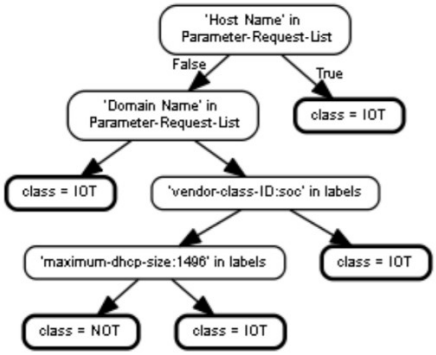

Figure 3: Decision tree visualization of DHCP -based classifier

The first step in device registration is to determine if the DHCP Discover packet came from an IoT device or a general-purpose device. To do this, I wrote an algorithm based on the technique laid out in "IoT or NoT: Identifying IoT Devices in a ShortTime Scale" (figure 3). Additionally, I add a check for a MUD URL in option 161, since the presence of this is an indicator that the DHCP packet originated from an IoT device.

If the DHCP Discover packet did not come from an IoT device, then the script does nothing further.

If the DHCP Discover packet did come from an IoT device, then the script goes about retrieving a MUD file or a MUD-like file for the device. Since MUD and MUD-like files are identically formatted, the origin of the file makes no difference for lower layers of the router script functionality.

A MUD file is simple to retrieve. The script retrieves the MUD URL from option 161 in the DHCP packet. It then downloads the file at this URL.

However, if there is no option 161 in the DHCP packet, then the router script must download a MUD-like file from a MUD Crowd-Source Server. First, the router script must identify the model

of IoT device that sent the DHCP packet. Often, this can be done using the Fingerbank API, which works by mapping blocks of MAC addresses to different manufacturers. Sometimes this doesn't work, and the device model can be inferred from the DNS requests that the device makes upon being given an IP address. Finally, using machine learning techniques on the device networking behavior may yield the device model, though this remains an open problem.

Once the router script identifies the device model, it will request a MUD-like file for it from the MUD-like Crowd-Source Server (MCS). An in-depth explanation of the MCS is beyond the scope of this report. The MCS will either return a MUD-like file or respond with a file-not-found message. Given a MUD-like file, the script can continue as though it had received a MUD file.

Henceforth, we will refer to all files in a MUD format as MUD files, since MUD and MUD-like files will both be treated the same way at this point in the program. Since both device model identification and the MCS remain in development, the MUD-like file retrieval system descripted above is not yet supported in the new router script.

Regardless of origin, the MUD file and MAC address of the connecting IoT device are then passed as parameters to the FirewallManager's MUDtoFirewall function.

3.c.ii FirewallManager

The FirewallManager is the object responsible for building and maintaining the firewall for IoT devices. It contains one DatabaseManager object, and a dictionary (map) of <MAC address, DeviceChain object> pairs.

When the script begins (which in production will be upon router startup), the firewall manager sets up an IPTables chain challed the "IoT chain". All packets being forwarded by the router pass through the IoT chain. Initially, the IoT chain is empty. The IoT chain's job is to forward packets originating from an IoT device to their respective IPTables device chain. If a packet did not originate from an IoT device, then it will pass through the IoT chain without being forwarded and continue on to other IPTables rules on the router. In this way, this system does not interfere with other system administration meant for general-purpose devices.

When the MUDtoFirewall function is called, the MUD file is parsed to a set of valid endpoints. Specifically, the MUDtoRules function in gmud_decode.py returns two lists, a set of rules regulating incoming communication, and a set of rules regulating outgoing communication. Each rule is a structure containing a protocol, a port number, and a domain name or IP address. This structure represents a valid endpoint. It is assumed that all endpoints other that the set of valid endpoints are forbidden.

A new DeviceChain object is then created, passing the MAC address of the corresponding IoT device as a parameter. The MAC address, DeviceChain pair is then stored in the FirewallManager's dictionary of known devices. Then the buildChain function is called with the newly created DeviceChain, passing the list of outbound communication rules as a parameter.

3.c.iii DeviceChain

The device chain is responsible for creating and maintaining a firewall for a given IoT device.

When the buildChain function is called on the DeviceChain object, a new IPTables chain is created. This chain is uniquely identified by the MAC address of the IoT device it was created for. Then, a new rule is added to the IoT chain, which forwards all packets with a source MAC address equal to the device's to the IPTables device chain.

Finally, the buildChain function populates the IPTables device chain with rules based on the list of outgoing communication rules passed to it. The DeviceChain object creates an IoT chain and populates that chain with rules, but it also keeps an identical list of rules internally in program memory. This is useful for IPTables chain maintenance, which I will discuss in the next section.

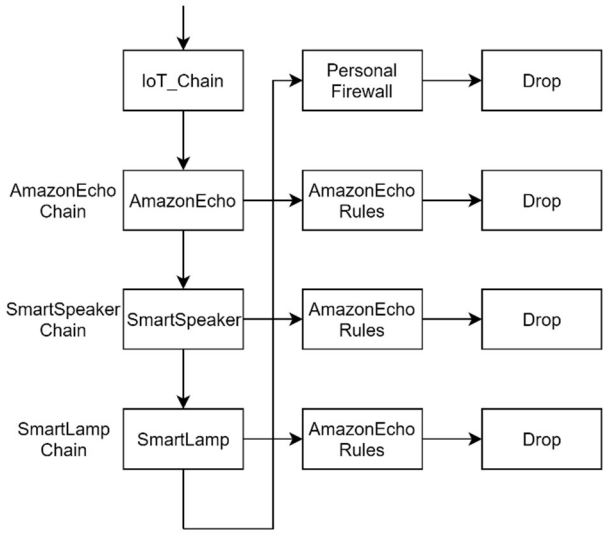

Figure 4: a visualization of the IPTables structure

All packets are routed through the IoT chain. If the packet originated from an IoT device, then it is forwarded to the corresponding IPTables device chain. If the packet's protocol, destination port, and destination address all match a valid endpoint specified by the IoT device's MUD file, then the packet will be allowed through. If not, the packet will be dropped.

One might notice that we are only creating IPTables rules for outbound traffic from IoT devices. However, a MUD file contains a set of rules for incoming and outgoing packets. The reason we do not generate a set of IPTables rules for packets destined for an IoT device is that while IPTables supports a '--mac-source' flag, there is no corresponding '--mac-destination' flag. Does this mean it is impossible to regulate incoming traffic to an IoT device with IPTables? Of course not. One could, for example, inspect DHCP Response packets and record the IP address assigned to an IoT device when it connects to the router. We could then create traffic rules based on a destination IP, updating this IP when it changes.

If we were to regulate incoming communication to IoT devices, malicious actors could still spoof the IP address of their packets to look as though they came from a valid endpoint. They, of course, would not be able to establish any kind of two-way communication this way, since they wouldn't own the IP address associated with the valid endpoint, but they could still send packets that would reach the IoT device. Thus, by regulating both incoming and outgoing transmission, hackers can still send packets to IoT devices, but can't get a response.

However, by regulating *only* outgoing packets from IoT devices, we get the exact same result. Attackers can send packets to IoT devices, but cannot establish two-way communication, due to regulated outgoing communications. For this reason, I do not think that the added complexity needed to regulate incoming communication is justified by an increase in security.

Now that we have an efficient, maintainable IPTables structure for regulating IoT traffic, we need to find a way to preserve the IPTables state between reboots. We cannot simply use the iptables-save command line function after each change to the IPTables because DD-WRT does not support iptables-save. Furthermore, this solution would cause bugs if the IPTables rules were to be significantly modified by a user or other programs. Since IPTables are a commonly used userspace utility, this would not be a good solution.

3.c.iv DatabaseManager

Instead, I created a database to store the MAC address and traffic rules of each device. This is all the information one needs to regenerate the IPTables rules after they have been flushed. Database functionality is handled by the DatabaseManager object.

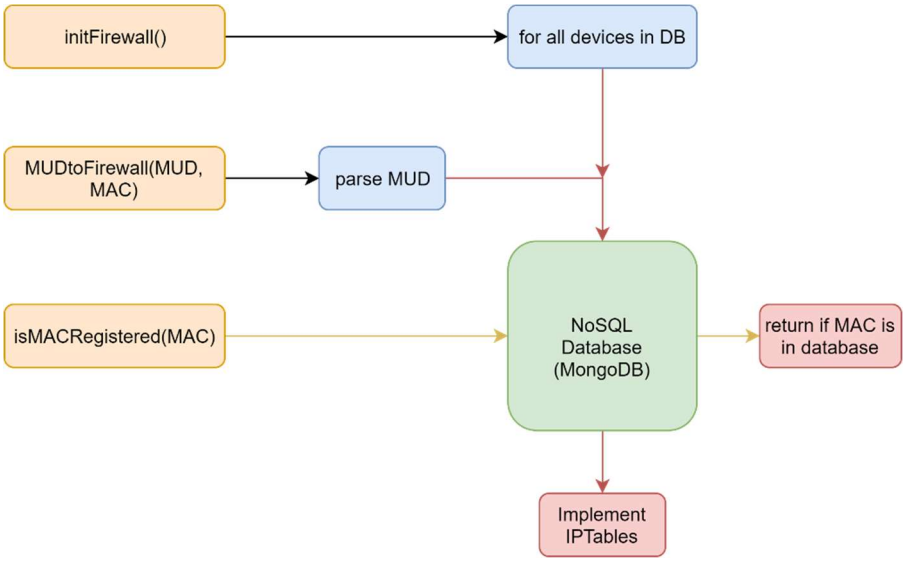

Figure 5: interaction between FirewallManager and DatabaseManager

When the DatabaseManager is initialized at the beginning of the program, it connects to a MongoDB database at a URL specified in the combinedsniffer.py file. I set up an online

MongoDB database with Atlas for the sake of ease; however, future development should switch to a local database.

Every time the MUDtoFirewall function in the FirewallManager is called, the FirewallManager calls the addIoT function in the DatabaseManager. This function creates a mongoengine document and adds it to the database.

When the router script restarts, the FirewallManager uses the database to retrieve the list of known devices and their respective traffic rules so that it can rebuild the IPTables rules. It does this by calling getIoTList, which returns a list of registered MAC addresses. Then, for each registered MAC address, it retrieves the traffic rules associated with that MAC from the database and implements them as IPTables rules using DeviceChain exactly the same way as explained before.

With this these new objects, I was able to fix the problems that caused IoT devices to be able to contact forbidden endpoints. In addition, I was able to structure the code in a logical way that will make it easier to modify and maintain in the future.

3.d Blocking Allowed Endpoints

The issue of valid endpoints being blocked by the old router script stemmed from two separate problems in DNS resolution. Most of the time, an endpoint in a MUD file specifies a domain name rather than an IP address. This means that an IoT device should be able to communicate with whatever IP addresses that domain name resolves to at a given point in time. The first issue with the old script was that it only generated an IPTables rule for the first IP address returned by the DNS resolver. This meant that if a domain name resolved to multiple IP addresses, the IoT device might try to contact a valid domain with a different IP than the one that the router script allowed traffic through. There is a second issue to consider as well; some domains resolve to different IP addresses at different points in time. If the router script were to generate IPTables rules when the IoT device connected, and then never update those rules, the router script would inadvertently block valid traffic when the IP associated with a domain name changed. The old router script had functionality to update IPTables rules when the DNS changed; however, there were too many other bugs on the script to test this functionality.

The issue of a domain name resolving to multiple IP addresses was easy to solve. When the DeviceChain generates IPTables rules, it resolves the domain name and implements an IPTables rule for each IP in the list of returned IP addresses.

The issue of changing mappings between domain names and IP addresses is a more difficult issue to solve. Ideally, I would write code to sniff DNS response packets from IoT devices' DNS requests. If a valid domain name was being resolved, the program would add those endpoints to the corresponding IPTables device chain. Unfortunately, the Scapy sniff function intercepts DNS responses unreliably, and so this approach is not acceptable given our current tools. The Scapy sniff function, however, reliably intercepts the DNS requests. When the PacketHandler sees a DNS request from an IoT device, it parses the domain name being resolved calls the

FirewallManager function updateDNSMapping. updateDNSMapping then identifies the DeviceChain that needs to be updated and calls refreshDomain on the DeviceChain object corresponding to the IoT device that sent the DNS request.

Each DeviceChain object holds an internal list of IPTables rules in the corresponding IPTables device chain. When refreshDomain is called, DeviceChain traverses its internal list of chain rules and deletes all rules associated with the domain name being refreshed. It deletes these rules both in the internal list of chain rules and, using the index associated with its internal list, in IPTables. It then uses Python's DNS resolver to get a new list of IP addresses associated with that domain name, and adds the new rules both to the IPTables device chain and the internal list of chain rules.

This technique is not 100% effective. Sometimes the IoT device's DNS request returns a different set of IP addresses than the router script's. When this happens, the router script blocks a valid endpoint. Further work should be done to see if Scapy can be modified to reliably intercept DNS responses or if a different tool can be used. Intercepting the IoT device's DNS response is the only way to ensure the router script never blocks a valid endpoint.

This new router script solves all three problems found in the old script. It doesn't crash. It never allows an IoT device to access a forbidden endpoint, and it doesn't block allowed endpoints except in special cases (discussed in section 4.b). I proved the first claim by monitoring the router script's memory footprint for seven hours, and I proved the second two claims using test tools that I wrote. A discussion of the test tools follows.

4. Test Tools

I wrote two test tools for this project. The first is a configurable script that runs on a Raspberry Pi, masquerading as an IoT device and generating network traffic to test endpoint regulation. The second tool is for testing the router script's resilience to a domain name changing IP addresses.

4.a IoT Traffic Generator

Relevant Files: traffic_generator.py (https://gitlab.com/columbia.irt/riot/blob/master/Fall2019/TestTools/traffic_generator.py)

The IoT Traffic Generator can masquerade as any IoT device with a MUD file. Features of the Traffic Generator can easily be modified by changing values in the header of test.py. Unless specified otherwise in the configuration, the Traffic Generator first broadcasts a DHCP Discover packet with a MUD URL. If the router script is working correctly, it should download the MUD file from this URL and implement IPTables rules to regulate the Traffic Generator's network communication. The Traffic Generator then generates traffic to test if its communications are being properly regulated.

First, the Traffic Generator generates legitimate traffic to ensure that valid endpoints in the MUD file are not being blocked. It parses the MUD file into a set of valid endpoints, sends a packet to each endpoint, and waits for a response. If the Traffic Generator receives a response from an endpoint, then it knows that endpoint is not being blocked by the router script. If the

Traffic Generator does not receive a response, it could either be because the endpoint is being blocked or because the endpoint is unresponsive. The user can tell if an endpoint is unresponsive by attempting to contact it from a general-purpose device.

Next, the Traffic Generator generates illegitimate traffic to ensure that invalid endpoints are being blocked by the router script. Specific illegitimate endpoints to be tested can be specified by the user in the configuration header. The Traffic Generator iterates through the specified list of illegitimate endpoints, sending a packet to each one. If the Traffic Generator receives a response, then it knows that not all invalid endpoints are being blocked.

Finally, the Traffic Generator outputs a summary of the tests. Telling the user which valid endpoints were incorrectly blocked and which invalid endpoints it was able to access.

By default, the IoT Traffic Generator masquerades as an Amazon Alexa; however, the user can specify which device the test tool should imitate. To output a list of devices that the Traffic Generator can imitate, include the '-ls' flag as an argument. For example, "sudo python traffic_generator.py -ls" prints a list of devices that the tool can imitate. By passing the name of the device's MUD to the Traffic Generator, the user can specify which device the Traffic Generator will imitate. For example, the command "sudo python traffic_generator.py HueBulb.json" causes the Traffic Generator to send a HueBulb MUD URL to the router script, and to test the endpoints specified in the HueBulb MUD file.

To add additional devices for the Traffic Generator to imitate, one needs a MUD file and a URL from which the router script can download that same MUD file. If the MUD file is not already hosted on the internet, this means one needs to host it on a server. One must then copy the MUD file to the Traffic Generator's MUDfiles directory, and update the test_devices.py file. test_devices.py maps the MUD file in the MUDfiles directory to the MUD URL that the Traffic Generator will include in its DHCP Discover. One must add an entry ("MUDfileName.json":"MUDURL") to the python dictionary that handles this mapping. This is all that is required for the Traffic Generator to imitate the behavior of a new device.

4.b DNS Resolution Test

While working to resolve the problem of the router script blocking valid endpoints, I came across a special endpoint with the domain name "device-fingerprintdb-v1.s3.amazonaws.com", that was especially difficult to prevent from being blocked. After some investigation, I found that this domain resolved to a different IP address every few seconds. This made it a perfect tool to test DNS updating functionality.

The DNS Resolution Test simply sends a DHCP Discover packet with an Amazon Alexa MUD URL, and then attempts to contact device-fingerprintdb-v1.s3.amazonaws.com every few seconds. Running this test while monitoring the router script shows that the router script updates relevant IPTables rules each time it intercepts a DNS request from an IoT device. However, the

router script sometimes blocks the test tool from contacting this valid endpoint due to the speed at which the domain-IP mapping changes.

5. Conclusion

Testing confirms that I have made substantial improvement in the functionality of the router script this semester. Hopefully this will provide solid ground upon which future progress can be made in implementing reliable device identification and creating and populating a MUD-like Crowd-Source Server. Future work for the router script includes moving the MongoDB database, currently hosted on Atlas, to the router, generating IPTables rules to regulate incoming traffic to IoT devices (if it is decided that this is important), and finding a way to inspect DNS response packets to improve the reliability of DNS updating. Once these tasks are completed, I believe that the router script will be ready to be packaged and distributed as a v0 prototype.

Appendix – Quick Start Guide (for future students)

These are instructions to run my code exactly as I have. Obviously, setting up a database in the cloud for every router in America is not a sustainable solution. This cloud database should be replaced with a local database as soon as possible. This is a trivial task if you can install MongoDB on the DD-WRT router (I found DD-WRT's package system too horrendous to complete this task).

You begin by setting up a MongoDB database. I recommend launching a free cluster with Atlas. You then follow some configuration instruction (whitelist the IP that the router will be connecting to the database with, set up a user for the database, etc.) to set things up. Then you want to connect your cluster to the router script (navigate: connect > connect your application). This will give you a connection string that looks something like this:

```
mongodb://<dbUser>:<password>@riotcluster-shard-00-00-
i0ezx.mongodb.net:27017,riotcluster-shard-00-01-
i0ezx.mongodb.net:27017,riotcluster-shard-00-02-
i0ezx.mongodb.net:27017/test?retryWrites=true&w=majority
```

Replace username and password, and then add `ssl=true&ssl_cert_reqs=CERT_NONE&replicaSet=RiotCluster-shard-0&authSource=admin&` directly after `test?` to ensure that there aren't any ssl certificate issues.

This is your value for dbURL. Replace the old value of dbURL to this in CombinedSniffer.py so that the router script communicates to this new database that you own.

You run the router script by running "python CombinedSniffer.py". Now, load the test tools onto a raspberry pi (or any Linux machine), connect to the router, and run "sudo python traffic_generator.py HueBulb.json". All tests should pass.