

Implementation of Telnet IoT Honeypot

Peter Elgar - 5396328

Adriaan de Vos - 4422643

Suzanne Maquelin - 5402840

Wouter Zonnenveld - 4582861

Abstract

The paper “IoTPOT: A Novel Honeypot for Revealing Current IoT Threats” by Pa et al. describes a design for a honeypot to analyse telnet malware targeting IoT devices. This report describes an implementation that was inspired by the design of the original paper. This report also provides the full source code and deployment documentation which, in contrast to the original paper, stimulates further research efforts. Not all parts of the original design are incorporated due to time limitations. The result shows a docker instance with multiple virtual machines which are reachable through our IoT Honeypot server that contains a custom telnet implementation, allows concurrent connections, is easily scalable, and ensures full logging capabilities for data analysis.

Keywords: IoT, honeypot, telnet, smarthome, malware

1 Introduction

In recent years there has been an exponential increase in the use of devices connected to the Internet, Lueth (2020). TVs, fire alarms, fridges are now often connected to the internet and have created a lot of new functionality. The catch-all term for these devices is called the Internet of Things or IoT.

Despite the fact that these devices have brought a lot of new functionality. A lot of them are not well secured and are easily accessible, Naik and Maral (2017). Many of these devices either have no password security, standard default passwords or passwords that can be easily brute-forced. Making these devices a perfect target for hackers. Where the hackers can in turn install malware on these devices. These devices are perfect for DDoS attacks or sending spam e-mails. Furthermore, many of these IoT devices are even more attractive than PC's as these devices are on 24/7. Moreover, they often have no antivirus and have easy access to powerful shells (such as BusyBox ¹).

One protocol many IoT devices use is Telnet. Telnet is a very old protocol that was never designed in the first place to be secure. Even though it may not be used by a lot of humans anymore, a lot of IoT devices still use the Telnet protocol because it is relatively easy to implement. Furthermore, Telnet attacks have skyrocketed and are often the favourite form of attack, Michael (2019).

Because of the before-mentioned cases, this motivated us to implement the paper by Pa et al. (2016). The paper was about the implementation of an IoT honeypot that ran multiple virtual machines which could emulate several device architectures. These devices were accessible from the outside through the Telnet protocol. However, once a virtual device was infected then the device could not connect to the outside world. In this research paper, they left the system running for 81 days and then decided to analyse the attacks and malware. However, as we only had a limited amount of time, the decision was made to only focus on the implementation of the IoT honeypot.

¹<https://busybox.net/>

2 Problem Requirements

The design of our IoT Honeypot would have to adhere to a number of requirements. First, we will discuss the high-level design from the original paper and use this as a basis for creating the requirements. Then we will go more in-depth about the different components and the corresponding requirements.

2.1 High-level Overview

First of all, we shall give a quick overview of what the roles of the components are before going into more detail. Our implementation is directly based on the original design which can be seen in figure 1. This design shows a profiler that scans the internet to store profiles. These profiles consist of Telnet banners, the method of authentication, and command interaction. The FrontEndResponder will emulate an unsecured IoT device based on the information provided from the profiles database and it is able to interact with rogue IoT clients which might want to send any commands to infect our Honeypot. These commands are executed in a safe virtual environment that support different architectures. The output of these commands are relayed back to the rogue IoT client so they are not aware of our FrontEndResponder playing the man in the middle. The target IoT devices can be on several different CPU architectures. The orchestration and interaction between the FrontEndRespodners and virtual machines is done by the Manager. In our implementation, we have chosen not to implement the downloader part, as this was also not fully implemented in Pa et al. (2016). In addition, we have used an existing dataset to generate the profiles instead of scanning the internet. Finally, the command interaction in the profiles is not supported because the dataset did not contain this information.

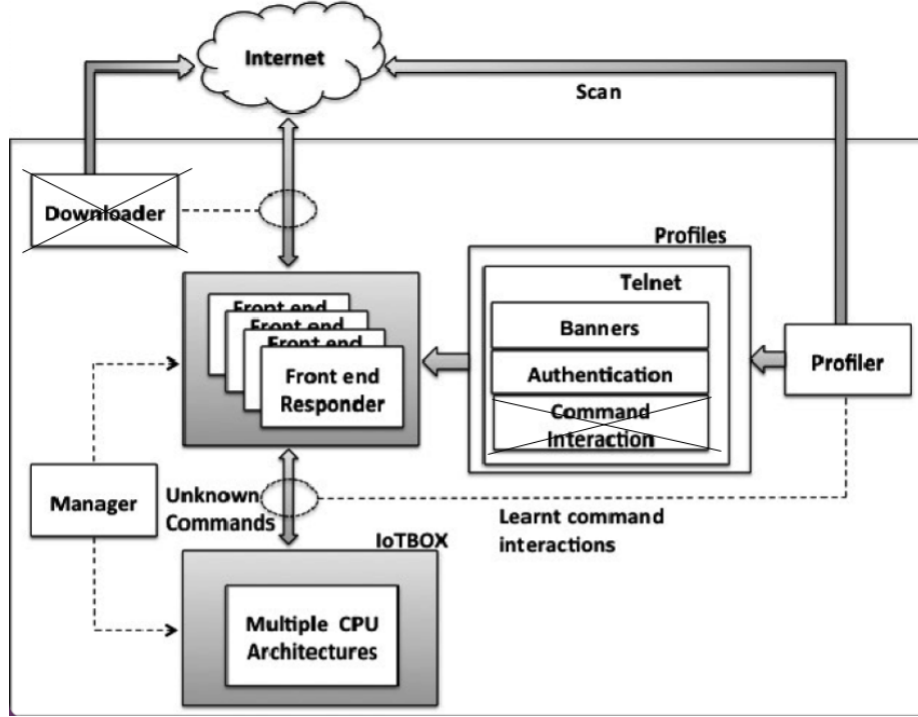


Figure 1. Overview of IoTPOT adapted from Pa et al. (2016)

2.2 Telnet

The decision was made to implement a custom telnet implementation to be able to log all low-level information available as current libraries do not provide this granularity. Firstly it is important that after doing the TCP three-way handshake, the client and the server can optionally exchange Telnet options and sub-negotiation commands. Either the Telnet server or client can initiate requests like "Do Echo" or "Do NAWs" to negotiate certain features of the used telnet session. The server needs to support these options and needs to be able to send valid responses back to the client because otherwise it might be detected as a honeypot instead of a legit device. After the exchange of Telnet options, the server should send a welcome message to the client, immediately followed by a login prompt. Then the client should send a pair of username/password to login to the server, also known as the authentication phase. If the credentials are valid the command interaction phase can begin.

This can be summed up in figure 2.

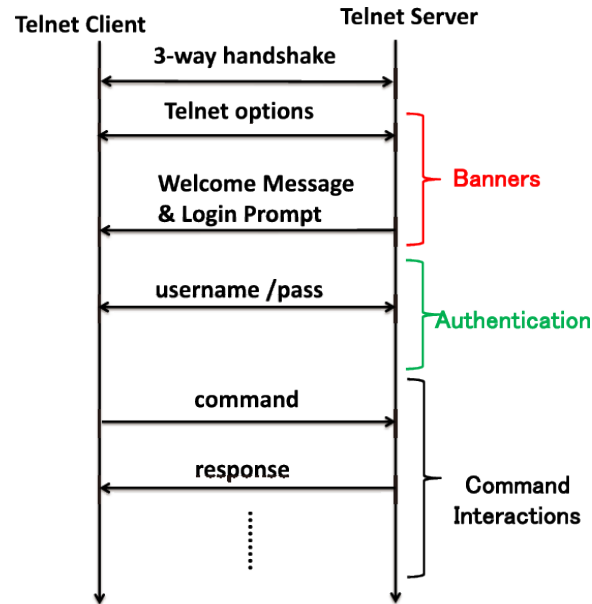


Figure 2. Telnet Protocol taken from Pa et al. (2016)

2.3 Profiler

The Profiler should be the component where the different profiles of IoT devices are stored. So that these can be accessed by the FrontEndResponder in order to impersonate an IoT device. Each profile should include the banners and the mode of authentication. The information that is stored in these profiles could be gathered by scanning the internet, but we have used an existing dataset.

2.4 FrontEndResponder

This component should act as an IoT device and simulates all the functionality that should normally be available within the real IoT Device. It should handle negotiating options, sub-negotiation, banner interaction, and relaying and responding to commands.

2.5 Logging

When a client connects to the server it must be the case that the dialogue and the client's information are logged and stored on a database. This is necessary otherwise there is no other way of analysis. This was not specifically mentioned in Pa et al. (2016), yet we saw the implementation as paramount. Without the logging we otherwise would not have the possibility to effectively analyze the results, debug our code and learn about the Telnet protocol.

2.6 Simulating different devices

The IoT honeypot must be able to support several CPU architectures which it can simulate. As the aim of this project is to simulate several IoT devices architectures and make it possible for clients to access them and install malware on them. As most IoT devices run on Linux, the decision was made to start with running embedded Linux OS on different CPU architectures.

2.7 Methods of Authentication

There should be different settings of how the client must authenticate. Either with always accepting the credentials, one of several default usernames/passwords e.g. root/root. Then, thirdly, there must be the possibility that the client must attempt x times to log in. Finally, there must be a way to rotate through these settings, either choosing the method of authentication by random, rotation or making it profile specific.

2.8 Manager

There should be a central component that is responsible for setting up connections to the database and the virtual machines. This component should also open a socket to accept incoming connections and have the ability to monitor virtual machines to see if they need to be refreshed. The manager should provide the link between the FrontEndResponder and the virtual machines.

3 Implementation

In this section, we will explain our implementation that is based on the IoT Honeypot paper. As the original paper merely provided an overview of the system components and omits many implementation details, we had to give our own interpretation of the system implementation. We decided to implement the system within a single Docker instance, so it can easily be deployed, and it will always be the same on every computer. We used the programming language Python, MongoDB as database, and QEMU to simulate the different system architectures.

First, we will describe in section 3.1 our initial entry point “main.py” that is responsible for linking all components together. Then we will describe all main components such as the FrontEndResponder in section 3.2, the Manager in section 3.3 and the database logic and objects in section 3.4. We continue our explanation of the implementation in section 3.5 with the Docker image and the configuration of the QEMU virtual machines. Instructions for running our implementation can be found in appendix A.

Finally, we decided to implement our own Telnet protocol functionality. The reason for this was that the current Python plugins did not hold the desired functionality or flexibility that we required in regards to low-level logging. Therefore, we decided to dive deeper into how the Telnet protocol works and what the different functions are.

3.1 Initial Entry Point / Startup

The initial entry point of our software is the file `main.py`. This file is responsible for initializing all necessary components and accepting incoming Telnet connections that will be passed to the FrontEndResponder. The order of initialization is as follows: First, the logging is initialized because it allows us to notice any warnings and errors that occur within the program. Second, we will initialize the database connection that allows us to retrieve profile information and store finished Telnet sessions. Third, we will initialize the manager that is responsible for creating a connection to all virtual machines. Fourth, we open a raw socket that is used to communicate with incoming Telnet connections from rogue IoT devices. Finally, we have an endless loop to listen for new Telnet connections and a rather complex shutdown function to cleanly exit all the separate components and threads. Below we will expand upon these functions.

3.1.1 Initialize Logging

As the owner that is hosting our IoT honeypot, you want to have insight into what is happening within the program. Therefore, we have made logging that is able to both log to `stdout` and the disk. Logging to the console is useful when developing, as the developer immediately sees what happens. In addition, this logging will appear when running the application as a system service. In addition we are logging to the disk in the folder `./logs/` which stores a history of all runs with the timestamp as the filename. For this, we have used the `logging` library of Python because we did not want to reinvent the wheel. For this, we only had to specify a `FileHandler` and `StreamHandler` and the format we wanted the loglines to be. An example of our log messages during startup can be seen in the image below.

```
[2021-04-04 11:32:18,929] {vm.py:27} INFO - Trying to set up a telnet connection to the x86 64 VM...
[2021-04-04 11:32:18,929] {vm.py:27} INFO - Trying to set up a telnet connection to the mips VM...
[2021-04-04 11:32:18,929] {vm.py:27} INFO - Trying to set up a telnet connection to the ARM VM...
[2021-04-04 11:32:18,929] {main.py:40} INFO - FrontEndResponder is listening on port 23.
[2021-04-04 11:32:28,219] {telnet_thread.py:41} INFO - Incoming connection received from: 127.0.0.1.
[2021-04-04 11:32:28,219] {telnet_thread.py:42} INFO - For 127.0.0.1: Using mips architecture.
[2021-04-04 11:32:29,498] {telnet_thread.py:55} INFO - For 127.0.0.1: Requesting random profile from database.
```

Figure 3. An example of logging during startup

3.1.2 Initialize Database Connection

For this functionality, we use functions that will be described more extensively in the DatabaseConnection section 3.4. For initialization, we create two database connections to the MongoDB server as we use two different collections to store the data. These database connections are then passed to classes that contain the logic for retrieving and storing the profiles and logging.

3.1.3 Initialize VM Manager

This ensures that a Telnet connection is created to all QEMU virtual machines and a thread is created that regularly checks if any of the virtual machines need to be restarted. The functionality will be described extensively in section 3.3.

3.1.4 Initialize Socket

To accept new incoming connections from rogue IoT devices we need to initialize a socket that we listen to. We used Python's socket library to initialise a raw Unix socket. We initialise the socket as standard procedure and make sure that the socket is reusable when the program has closed. Then we bind it to the telnet port (23) and enable it to retrieve new connections before logging the action in the debug log file.

3.1.5 Accept incoming connections and cleanly exit

Finally, there is an endless loop that automatically accepts the telnet connection and propagates the session to a TelnetThread in the FrontEndResponder. The loop will endlessly go on unless there is a keyboard interrupt. Then the method carefully logs that it is shutting down. Making sure that all current sessions cleanly exit and have time to store their information to the database, cleanly exiting all running threads and finally closing the database connection.

3.2 FrontEndResponder

The FrontEndResponder consists of a number of components, namely the files `telnet_thread.py` and `byte_parser.py`. For every incoming telnet connection, a new TelnetThread object is created to ensure our application allows concurrent users and is efficiently multithreading it.

3.2.1 Initialization of a TelnetThread

In the `TelnetThread` class we initialise it firstly by making a thread and then by initialising the variables of the client's socket object called `conn`, the profile database called `database_profile`, the logging database called `database_logging`, the client IP called `client_IP` and the vm connection called `vm_connection`. Then we initialise the variables `alive`, for if the thread is alive or not, and so we can close the thread in a "friendly" way. Continuing we initialise the `history` variable (which are the requests that the client has sent to the server) and the `profile` variable. Then two booleans for if a welcome message has been sent (`welcome_send`) and if the client has logged in (`logged_in`). Then the login username called `login_username` (an empty string) and the login counter called `login_counter` (an integer set to 0).

3.2.2 Data Input/Output & Profile Selection

Then in the `run` method, which comes after the initialisation, a random profile is retrieved from the profile database, and the `telnet_thread` method gets called which monitors the incoming and outgoing bytes. If the connection has been closed then a log will be sent to the logging database.

Before moving on to the other methods, it is important to first explain the logic and data structure that is required to sent information back to the client in the `send_to_client` method. The only parameter being the data as a bytearray that has to be sent. In order to send the data there is a try catch statement. It will try to send the data except for the case that there is a Broken Pipe Error, in which case the thread is closed. If there is no Broken Pipe Error then the method will send as much data as it can through the client's socket, returning how much it was able to send. If nothing has been sent then an error will be raised. This loop will continue until all the data has been sent.

In the `telnet_thread` method, the method keeps on looping as long as the variable `alive` is true. If the client has sent new data to the server then the whole buffer is retrieved

from the socket and firstly added to the `history` variable. After that the buffer gets parsed by the `parse_buffer` function in the `byte_parser.py` file. The `parse_buffer` function separately returns the `commands` and the `text`. If no welcome message has been sent yet, then one will be sent. Furthermore, if the client has suddenly disappeared then the `TelnetThread` will notice this and cleanly exit. Finally, if the client has not logged in yet, then the `login` method gets called with the text from the `parse_buffer`.

3.2.3 Login Functionality

In the `login` method the parameter `text` is given. It first checks whether a login name has already been given. If that is not the case then the parameter `text` will become the login name, and the method sends a message to the client asking for the password. However, if it is the case that the client has already given its username, then the method gets its current profile's authentications type. These can be four settings; always accept, accepts after x amount of tries, accept for specific username/password or never accept. For the first case it is rather straightforward, a welcome message gets sent to the client and the variable `logged_in` gets assigned to true. For the second case the `login_counter` variable gets updated by one, then gets checked if the current `login_counter` is greater than the x amount of tries needed. If it is greater than x, then like the previous setting, `logged_in` gets assigned to true and a welcome message gets sent. Otherwise, `login_username` gets set to None again and the client is requested to try again. For the third authentication type, if the password given is the same as the password that is required, then the client is accepted like in the previous cases. Otherwise like the previous case the `login_username` is set to None and the server sends a message to the client to try again. Finally, for the last case the server sets the `login_username` to None and sends a message that the client has to try again. This case is interesting as it means that we can look at all the possible login combinations that a client may try.

3.2.4 Processing Telnet Commands & Options

In the `process_commands` method the `commands` array gets given as a parameter. Here the method loops through all commands inside the `commands` array. Here we have an if else statement, if it is the case that the first byte is greater than 250 then the command gets given to the `respond_to_option` method, more on this method later. However, if that is not the case then the method loops through all the bytes in the command. If the command contains the number 244 that means that the command `Interrupt Process` has been called then the method `terminate_thread` gets called, which simply sets the `alive` variable to false and the message that the client has terminated the connection gets logged. If it is the case that the command contains the byte 250, then that means the sub negotiation has been requested, so the method `sub_negotiation` gets called. In this method it simply accepts the only request we came across when testing the `FrontEndResponder` with our own devices, namely *negotiate about window size* here the server does not need to respond.

For the method `respond_to_options` we have two parameters, the first is obligatory which is the `command`. The second parameter, which is optional and by default set to true, is whether to `accept` the command or not. The `command` given to this method is only (meant to be) made out of two bytes. If it is the case that `accept` is true, then it has four cases, namely if the first byte of the command is 251 meaning "I will", 252 meaning "I won't", 253 meaning "You do" or 254 meaning "You don't". For the first case (251) the response is usually to send to the client a byte array containing three bytes, 255 (IAC which should be at the start of every command, more on this in the `byte_parser` section), 253 (you do) and the second byte of the command given. This, however, is not the case for if the second byte is equal to 33 or 34. As 33 is the Remote flow control option, which means the client does not have to start a new line for the command to be accepted. Furthermore, 34 is linemode, we decided not to accept this option as implementing it was beyond the scope of this implementation and would have taken far too long to implement. Therefore, for the bytes 33 and 34 the server returns a byte array containing the bytes 255 (IAC), 252 (I wont) and the second byte of the command. For the other options (252, 253 and 254) the response is as follows: for 252 it is a byte array with 255, 254 (You don't) and the second command byte; for 253 it is a byte array with 255, 251 (I will) and the second command

byte; and finally for 254 it is 255, 252 (I won't) and the second command byte.

3.2.5 Processing System Commands

After the IoT client has successfully authenticated itself. It is allowed to directly send any commands to one of the virtual machines. This is done by taking the `text` that results from the `parse_buffer` and sending it to the VM object called `vm_connection`. The response from the virtual machine is directly returned to the client. For more information see section 3.3.2.

3.2.6 Byte Parser

In this file we have two dictionaries and a function. These dictionaries contain all the existing telnet commands (RFC854 ²) and all the existing command options (RFC855 ³) that are defined in an Request for Comments (RFC) by the Internet Society (ISOC).

The function we have called `parse_buffer` which (as the name suggests) parses the `buffer` that has been sent by the client. At the start of the function we initialise an empty array for the commands called `commands`, an empty string for the text called `text` and an `index`. Then the function loops over the buffer, taking the first byte. If the byte corresponds to 255 meaning IAC, then this indicates that a command has started. All Telnet commands must start with 255. If this is not the case, then the byte gets converted to a character and added to the `text` string to store all bytes that are not part of a telnet command and are thus part of the login text or system command that needs to be relayed to the virtual machine.

However, if it is the case that the byte is equal to 255 then the `index` gets incremented. Before analysing the buffer, there first needs to be a check whether the buffer is as long as the current `index`. If that is the case then the byte in that `index` of the buffer can be checked if it is a Telnet command. If that is not the case, then the error gets logged and the

²<https://tools.ietf.org/html/rfc854>

³<https://tools.ietf.org/html/rfc855>

`index` gets incremented. If it does correspond to a Telnet command, then there is another if else statement to see if the Telnet command is greater than 250 or not. If greater than 250 then it is either one of 4 commands (will, won't, do, don't) with a command option. After incrementing the `index` again and checking if the `index` is not out of bounds, the Telnet option gets looked up in the corresponding dictionary. Again if this is a valid command option, then the command and the command option get put into an array and appended to the `commands` array. Please note, for every if/else statement we make debug logs. For the case that the option given is not a recognised option, this gets logged as well. For the case that the command is not above 250, there are two cases. Either it is a command on its own e.g. 244 Interrupt process, which gets put in an array and appended to the `commands` array, or it is the start of a sub negotiation i.e. 250, which means the command array will be longer. For the sub negotiation the function then loops over the buffer appending all the commands till `buffer[index]` is equal to 255 (IAC), and `buffer[index + 1]` is equal to 240 (end of sub negotiation). Once this loop has been finished the command array get appended to the array containing all the commands. Finally when the loop has finished the `commands` array and the `text` get returned.

We understand that the logic from this function is rather dense and we suggest checking out the code as it contains more context and comments that describe the parsing of these telnet commands and options.

3.3 Manager

This section describes the component that is responsible for managing the virtual machines (VMs) and the ability to relay commands from the client to a specific architecture.

3.3.1 Managing Virtual Machines

An essential part of this project is to have virtual machines with different architectures available that can be used to analyse how different IoT clients behave in different environments. In `manager.py`, when the Manager object is initialized it will create and store VM objects for each virtual machine that is available. More about this in the next paragraph.

The most important part of this component is to have a recurring check that fires every x minutes to clean and refresh the virtual machines, as they will be reused for multiple incoming Telnet sessions. This check can be configured for a certain period of time, and it makes sure to only refresh the VM if it is currently not in use. Refreshing the virtual machine is done by closing the Telnet connection to this machine, running a bash script called `qemu-restart-a-vm.sh`, and creating a new connection to the freshly configured virtual machine.

3.3.2 Virtual Machine Connection

For connecting to the virtual machines with different architectures we initialize a single Telnet session to the QEMU manager of the specific virtual machine that can be reused for multiple (concurrent) IoT clients. When starting this persistent session we first need to flush the messages that are currently in the buffer because they will contain the startup log and boot message from the virtual machine. In addition, we will keep track of the number of concurrent users to allow only re-configuring this virtual machine when it's not in use. This component allows relaying Telnet commands that are retrieved within the `FrontEndResponder` from the IoT client to the virtual machines. This is done by copying the IoT client's command to the persistent Telnet session and making sure that the response, minus the echo of the command is retrieved and relayed back to the `FrontEndResponder`. Finally, the VM object contains a function to request a string of the architecture that is used. This is used for log messages and log entries in the database.

3.4 Data storage

In the original paper, no information was provided on the implementation of data storage. At the beginning of the project, we were not yet sure about everything we wanted to store. Therefore, a document-based database was chosen to store profiles and logging records. Moreover, the need for extensive querying capabilities of an SQL database was not deemed necessary. As the team would be working on the project from different locations and on different computers, a cloud-based database would easily allow the different group members to use the same database with the same data. MongoDB Atlas is an online document-based

(NoSQL) database that has a free plan for students. As MongoDB Atlas fulfils all the aforementioned requirements, MongoDB Atlas was chosen as the storage medium. In this section, we shall briefly talk about how we implemented the database connection and how the logging was done.

3.4.1 Database Connection

To connect to the database, a Python library, called `pymongo`, is available. The file `DatabaseConnection.py` imports this library and contains basic functionality. This includes functions such as adding and deleting profiles. Also queries based on a profile ID, or entire profiles (matching each field) are implemented in this file. In the class, the client database is defined by calling `pymongo.MongoClient` with the database's URI. In the initialisation method, the database and the collection are defined. Thanks to the two parameters given (two strings) one with the name of the database and the other with the name of the collection/table.

The first method in the class after the initialisation is the `add_entry` which takes a JSON as a parameter. First, an ID gets generated and put in the JSON, then the JSON is added as a new entry in the collection, then the ID gets returned.

Then we have the `delete_entry`, `update_entry` and `find_entries` these take a JSON as entry (and `update_entry` takes a second parameter being the new value). All the names of these methods should be self-explanatory.

Then we have the two methods that are quite similar, `find_entry` and `find_entry_on_id`. `find_entry` searches for one entry using a JSON and returns the entry. `find_entry_on_id` finds one entry using a unique ID and returns that entry.

Finally, we have the `get_random_entry`, where a random entry gets returned from the collection. This is mainly used to get a random profile when starting a new thread.

3.4.2 Profiles

Profiles, which are stored in the database, consist of five different fields: an ID, Telnet options, banner information, an authentication profile, and command interactions. The functionality for storage of profiles is more complex than the functionality in `DatabaseConnection.py`. To improve modularity the new functionality is defined in `ProfileLogic.py`. The new functionality includes checking whether or not a profile already exists in the database before adding another profile, querying all profiles which match some banner information, and updating the command interactions field.

To collect banner information for the profile database, two different approaches were taken. Firstly the masscan tool Graham (2021) was used. However, since a lot of the information retrieved from this tool was useless for banner information, another method was chosen instead. A Github page Rapid7 (2021) was found (thanks to our supervisor) containing over 100 banner profiles. These profiles however were originally used the other way around. More specifically, incoming banners were parsed against regex expressions to see which device profile fit the incoming banner. Fortunately for each regex expression, an example banner that would fit the regex was included as a comment. To still be able to extract the banners, first a function was created to extract all comments from the `telnet_banners.xml` file. This function is implemented in `collect_banners.py` and uses a regular expression. The results are written to a file to temporarily hold these banners. Now each line in this file represents a banner. Using `push_banners_to_database.py`, the banners are then added to the database with an authentication profile to always accept incoming connections and with an empty command interaction field.

3.4.3 Logging Logic

In the `LoggingLogic` class we have two methods. The first one being the initialisation method which essentially initialises the database connection and the specific collection. The second method is the method `insert_log` which takes the parameters profile ID, client IP, architecture, and history buffer (i.e. all commands sent to the server). These get

put into JSON form along with a timestamp. Then adds the entry using the `add_entry` method from the `database_connection` class.

3.5 Docker & QEMU

QEMU is an open-source processor emulator. In this project, QEMU is used to simulate different CPU architectures which represent different IoT devices. When trying to set up QEMU, it was quickly realised that commands differ significantly based on which operating system someone is using. To improve the portability of the application, the decision was made to run QEMU in a docker container which could easily be used on all operating systems.

The docker image is built using a Dockerfile which provides the container with starting instructions. First, the Dockerfile states from which image the container should be built. As starting image, a Debian image (Debian 10.9, with image id `463adba1ec3f`) was chosen. Debian was chosen as it allows for easy installation of software packages, and is mainly centred around interaction via command line. Secondly, the Dockerfile instructs the container to update and then pull the specified packages. Many of the packages are for QEMU, but also packages such as Telnet and wget are installed for later use. In the next section, three QEMU architectures are downloaded from OpenWrt (more on this later). Hereafter, a setup script is copied into the container and makes it executable by setting the correct permissions. The setup script runs the three different QEMU architectures, each creating a Telnet endpoint on a different port. Lastly, the IoT Honeypot project is copied into the container and the project requirements, listed in `requirements.txt`, are installed.

3.5.1 QEMU architectures

In Pa et al. (2016) nine different architectures were implemented in the IoTBOX. However, due to time constraints, our implementation has three different architectures. The implemented architectures are x86, Mips, and ARM which are all OpenWrt based. Although OpenWrt ARM was not one of the architectures in Pa et al. (2016), it is a commonly used architecture, which is why we made the decision to add this particular architecture nonethe-

less. Several other Debian based architectures were attempted, namely Mipsel, PPC, and Debian ARM. However, with each of them, the team ran into issues that could not be resolved in time. For Mipsel, the example on the OpenWrt site ⁴ was followed, however, the team found it to no longer be working. An error was found having to do with the wrong endianness of the image used, even though the same image as in the example was used. For Debian ARM the Debian image requires an ssh key to log in. Adding an ssh key to an image is described in ⁵, however, the required actions were not recognised in the docker container. Unfortunately, no solution to this problem was found. The install information for PPC Debian Qemu ⁶ was very limited and following the given instructions resulted in a failure during booting. A different image did not solve this problem.

In consultation with our supervisor, we decided that showing 3 different processor architectures was enough to show the multi-architecture feasibility of this project. Our current IoT Honeypot supports easy addition of different virtual machines when they are successfully configured.

3.5.2 Network

In the description of the IoTBOX in Pa et al. (2016), networking to the QEMU instances is achieved via a network bridge in combination with tap interfaces. In order to do this, first, the docker container needs the correct permissions. Using the `--cap-add=NET_ADMIN` argument, the docker gets permission to perform various network-related operations. In order to use tap interfaces, a device flag needs to be set as such `--device=/dev/net/tun`. Now, with a package called `bridge-utils`, a bridge can be created. However, the bridge itself does not yet have permissions to change the network. To fix this, a config file needs to be added to the container which grants the bridge this permission. The config file contains one simple instruction `allow br0` (br0 will be the name of the bridge) and is located in the `/etc/qemu` folder. Now using the command `brctl addbr br0`, we can create a working bridge. From here we can add tap interfaces to the bridge. This can easily be done by passing an

⁴https://openwrt.org/docs/guide-user/virtualization/qemu#openwrt_in_qemu_mips

⁵<https://wiki.debian.org/Arm64Qemu>

⁶<https://wiki.debian.org/DebianInstaller/PowerPC/qemu>

additional flag when starting a QEMU instance ⁷.

However, the bridge is not yet used for incoming internet traffic. The paper described to using NAT rules to connect the bridge to the internet. However, no details on the rules were given. We have chosen to work with PREROUTING rules. A PREROUTING rule redirects incoming traffic right after it has entered the network interface. The team has experimented extensively with these NAT rules, however, no successful rule with the desired effect has been found. Due to this problem, the team opted for a different approach for communication with the QEMU instances.

When launching a new QEMU instance, the flag `-serial telnet:localhost:1234,server,nowait` can be passed. What this does, is that a telnet endpoint is created for the QEMU instance on which it can be reached. In this example, the port is 1234 and by giving each QEMU instance a unique port, the instances can be contacted individually. With this new approach, both the network bridge and the network interfaces are no longer needed. However, this comes at the cost of not being able to use the IoTBOX as a stand-alone system for malware binary analysis. However, due to time constraints, this decision had to be made in order to still have a working system within the given time window.

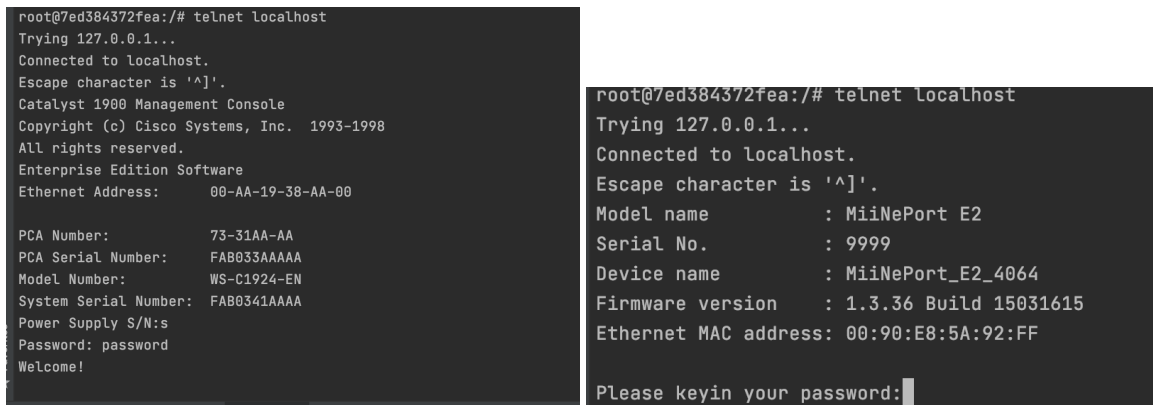
4 Results

In this section of the paper we shall show and discuss the results of our implementation. What we have achieved and what it looks like from the client's point of view and how the server reacts to a client connecting it.

4.1 Client's Point of View

Here are some examples of what the different profiles look like. In figure 4 it is obvious to the client that it is dealing with a different system and looks realistic that this is an IoT device not a server.

⁷<https://gist.github.com/extremecoders-re/e8fd8a67a515fee0c873dcafc81d811c>



```

root@7ed384372fea:/# telnet localhost
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Catalyst 1900 Management Console
Copyright (c) Cisco Systems, Inc. 1993-1998
All rights reserved.
Enterprise Edition Software
Ethernet Address: 00-AA-19-38-AA-00

PCA Number: 73-31AA-AA
PCA Serial Number: FAB033AAAAA
Model Number: WS-C1924-EN
System Serial Number: FAB0341AAAA
Power Supply S/N:s
Password: password
Welcome!

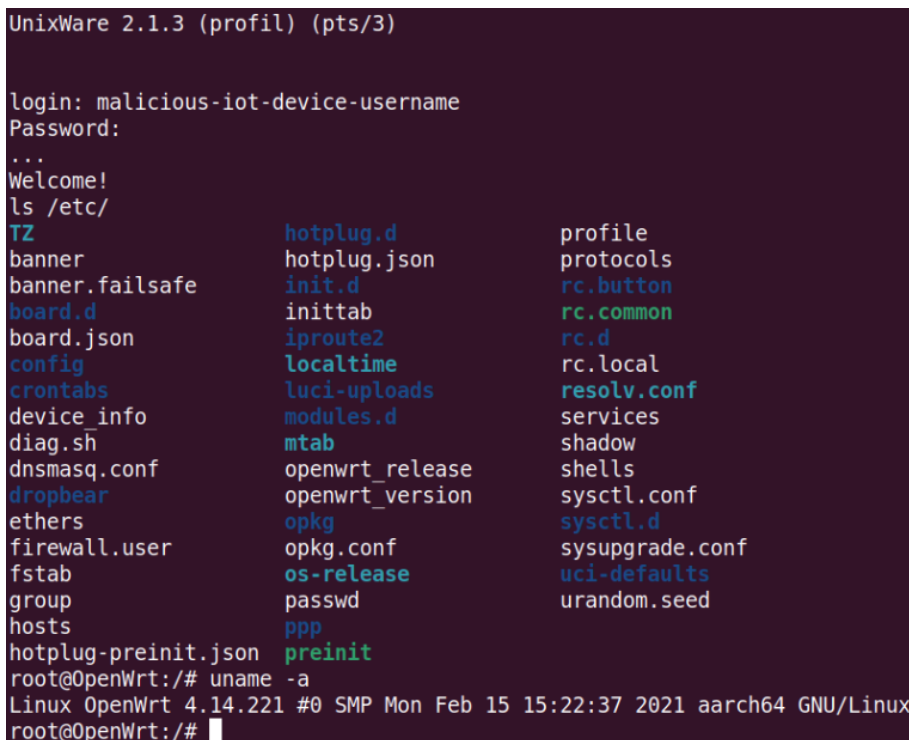
root@7ed384372fea:/# telnet localhost
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Model name : MiNePort E2
Serial No. : 9999
Device name : MiNePort_E2_4064
Firmware version : 1.3.36 Build 15031615
Ethernet MAC address: 00:90:E8:5A:92:FF

Please keyin your password:

```

Figure 4. Examples of different profiles

The following figure shows an example of a client connected to a virtual machine through our FrontEndResponder that is executing different system commands.



```

UnixWare 2.1.3 (profil) (pts/3)

login: malicious-iot-device-username
Password:
...
Welcome!
ls /etc/
TZ                hotplug.d         profile
banner            hotplug.json      protocols
banner.fail-safe  init.d            rc.button
board.d           inittab           rc.common
board.json        iproute2          rc.d
config            localtime         rc.local
crontabs          luci-uploads      resolv.conf
device_info       modules.d         services
diag.sh           mtab              shadow
dnsmasq.conf      openwrt_release  shells
dropbear          openwrt_version  sysctl.conf
ethers            opkg              sysctl.d
firewall.user     opkg.conf         sysupgrade.conf
fstab             os-release        uci-defaults
group             passwd            urandom.seed
hosts             ppp
hotplug-preinit.json preinit
root@OpenWrt:/# uname -a
Linux OpenWrt 4.14.221 #0 SMP Mon Feb 15 15:22:37 2021 aarch64 GNU/Linux
root@OpenWrt:/#

```

Figure 5. Example of a client executing commands on a virtual machine

4.2 Server's Point of View

In figure 6, per line we can observe the date and time, the file where the log gets called and then what is happening. We can see the new that a new connection is received and a VM and a profile get selected. Then commands get received and processed.

```
[2021-04-08 07:21:38,176] {telnet_thread.py:41} INFO - Incoming connection received from: 127.0.0.1.
[2021-04-08 07:21:38,176] {telnet_thread.py:42} INFO - For 127.0.0.1: Using mips architecture.
[2021-04-08 07:21:39,295] {telnet_thread.py:55} INFO - For 127.0.0.1: Requesting random profile from database. Got:
6059c39f29859e732d8e8277
[2021-04-08 07:21:39,295] {byte_parser.py:96} INFO - FROM 127.0.0.1 IAC DO : Suppress Go Ahead
[2021-04-08 07:21:39,296] {byte_parser.py:96} INFO - FROM 127.0.0.1 IAC WILL : End of Record
[2021-04-08 07:21:39,296] {byte_parser.py:96} INFO - FROM 127.0.0.1 IAC WILL : Negotiate About Window Size
[2021-04-08 07:21:39,296] {byte_parser.py:96} INFO - FROM 127.0.0.1 IAC WILL : Terminal Speed
[2021-04-08 07:21:39,296] {byte_parser.py:96} INFO - FROM 127.0.0.1 IAC WILL : Remote Flow Control
[2021-04-08 07:21:39,296] {byte_parser.py:96} INFO - FROM 127.0.0.1 IAC WILL : Linemode
[2021-04-08 07:21:39,296] {byte_parser.py:96} INFO - FROM 127.0.0.1 IAC WILL : New Environment Option
[2021-04-08 07:21:39,296] {byte_parser.py:96} INFO - FROM 127.0.0.1 IAC DO : Status
[2021-04-08 07:21:39,296] {byte_parser.py:103} INFO - FROM 127.0.0.1 IAC SB Negotiate About Window Size
[2021-04-08 07:21:39,297] {byte_parser.py:110} INFO - FROM 127.0.0.1 IAC SE
[2021-04-08 07:21:57,285] {telnet_thread.py:64} INFO - For 127.0.0.1: Tried to login with malicious-iot-device-use
rname:...
[2021-04-08 07:22:02,550] {telnet_thread.py:110} INFO - For 127.0.0.1: Run command "ls"
[2021-04-08 07:22:08,882] {manager.py:43} INFO - Checking if any VM needs to be refreshed...
[2021-04-08 07:22:10,401] {telnet_thread.py:110} INFO - For 127.0.0.1: Run command "uname -a"
[2021-04-08 07:22:14,557] {byte_parser.py:103} INFO - FROM 127.0.0.1 IAC SB Negotiate About Window Size
[2021-04-08 07:22:14,561] {byte_parser.py:110} INFO - FROM 127.0.0.1 IAC SE
```

Figure 6. Server's point of view when client connects to Server

The following figure shows a snippet that shows the shutdown process including closing the connections and writing the information to the MongoDB database.

```
[2021-04-08 07:31:08,889] {manager.py:43} INFO - Checking if any VM needs to be refreshed...
^C[2021-04-08 07:31:17,731] {main.py:58} INFO - Closing FrontEndResponder...
[2021-04-08 07:31:21,816] {telnet_thread.py:47} INFO - Connection closed from: 127.0.0.1
[2021-04-08 07:31:22,287] {telnet_thread.py:49} INFO - Session from 127.0.0.1 is logged to MongoDB with id: 606eb14
900174f5fa3949e2d
[2021-04-08 07:31:27,051] {telnet_thread.py:47} INFO - Connection closed from: 127.0.0.1
[2021-04-08 07:31:27,085] {telnet_thread.py:49} INFO - Session from 127.0.0.1 is logged to MongoDB with id: 606eb14
f00174f5fa3949e2e
```

Figure 7. Example of the server shutting down cleanly

4.3 Virtual Machines

In the figure 9 we can see the cases where the different virtual machines (VM's) get set up, checking if any need refreshing and then refreshing the VM's before reconnecting to these devices.

```

[2021-04-06 12:04:38,253] {vm.py:29} INFO - Trying to set up a telnet connection to the x86_64 VM...
[2021-04-06 12:04:38,255] {vm.py:29} INFO - Trying to set up a telnet connection to the mips VM...
[2021-04-06 12:04:38,258] {vm.py:29} INFO - Trying to set up a telnet connection to the ARM VM...
[2021-04-06 12:04:38,267] {main.py:40} INFO - FrontEndResponder is listening on port 23.
[2021-04-06 12:05:38,271] {manager.py:43} INFO - Checking if any VM needs to be refreshed...
[2021-04-06 12:06:38,284] {manager.py:43} INFO - Checking if any VM needs to be refreshed...
[2021-04-06 12:07:38,287] {manager.py:43} INFO - Checking if any VM needs to be refreshed...
[2021-04-06 12:08:38,290] {manager.py:43} INFO - Checking if any VM needs to be refreshed...
[2021-04-06 12:09:38,292] {manager.py:43} INFO - Checking if any VM needs to be refreshed...
[2021-04-06 12:10:38,295] {manager.py:43} INFO - Checking if any VM needs to be refreshed...
[2021-04-06 12:11:38,298] {manager.py:43} INFO - Checking if any VM needs to be refreshed...
[2021-04-06 12:12:38,303] {manager.py:43} INFO - Checking if any VM needs to be refreshed...
[2021-04-06 12:13:38,305] {manager.py:43} INFO - Checking if any VM needs to be refreshed...
[2021-04-06 12:14:38,315] {manager.py:43} INFO - Checking if any VM needs to be refreshed...
[2021-04-06 12:15:38,331] {manager.py:43} INFO - Checking if any VM needs to be refreshed...
[2021-04-06 12:16:38,341] {manager.py:43} INFO - Checking if any VM needs to be refreshed...
[2021-04-06 12:17:38,595] {manager.py:43} INFO - Checking if any VM needs to be refreshed...
[2021-04-06 12:18:38,598] {manager.py:43} INFO - Checking if any VM needs to be refreshed...
[2021-04-06 12:19:38,601] {manager.py:43} INFO - Checking if any VM needs to be refreshed...
[2021-04-06 12:20:38,606] {manager.py:43} INFO - Checking if any VM needs to be refreshed...
[2021-04-06 12:20:38,612] {manager.py:31} INFO - Refreshing the x86_64 architecture QEMU instance because it's old and unused.
[2021-04-06 12:20:41,729] {vm.py:29} INFO - Trying to set up a telnet connection to the x86_64 VM...
[2021-04-06 12:20:41,729] {manager.py:31} INFO - Refreshing the ARM architecture QEMU instance because it's old and unused.
[2021-04-06 12:20:42,032] {vm.py:29} INFO - Trying to set up a telnet connection to the ARM VM...
[2021-04-06 12:21:38,610] {manager.py:43} INFO - Checking if any VM needs to be refreshed...
[2021-04-06 12:21:38,621] {manager.py:31} INFO - Refreshing the mips architecture QEMU instance because it's old and unused.
[2021-04-06 12:21:38,992] {vm.py:29} INFO - Trying to set up a telnet connection to the mips VM...

```

Figure 8. Example of restarting VM's

Finally, in the following figure we show our IoT Honeypot in action with three different connected IoT clients. This is an overview that we often used during development to check if everything worked. In practice the clients will be outside the docker instance.

Overall we are satisfied with what we have achieved in this research. However, going forward we would like to mention the parts which we believe would be interesting to expand on or acknowledge our shortfalls.

Case 1: $\alpha = 1$. In this case, $\beta = 1$ and $\gamma = 1$. Then, $\alpha + \beta + \gamma = 3$ and $\alpha\beta\gamma = 1$. The inequality becomes

Secondly, as having mentioned this in the FrontEndResponder section of the imple-

mentation, we believe it may be interesting to implement even more telnet functions and options to provide a better telnet simulation to the FrontEndResponder. Implementing functionality such as linemode would have made this honeypot more accessible to other devices.

Thirdly, it would have been interesting to implement more CPU architectures on the QEMU. As the IoT landscape is large and diverse it would therefore have been interesting to implement many more CPU architectures to make the honeypot more realistic in being able to represent the current range of IoT devices. Furthermore, it would be interesting as a way of analysing the different kinds of malware and attacks. To see which devices get targeted by which attacks.

Another limitation of our work is that the standalone capabilities of the IoTBOX has not been implemented. This was due to difficulties in setting up NAT traffic rules which allow the QEMU instances to access the internet. Although this does not limit the system of IoTPOD as a whole, being able to use the IoTBOX as a stand-alone system would have been a useful additional feature to have. This is because it would allow us to generate an analysis report consisting different packets of data, such as total number of packets received.

Furthermore, it would have been interesting if when the client connected to the server that the server also scraped the client. As then, we could get more data, like in Pa et al. (2016), about which IoT devices get infected by which malware.

Finally, it would have been a good feature that certain profile specific command responses to the client would be saved in the Profiles. Possibly even some more general automatic responses in the FrontEndResponder. Instead of having to constantly ask for a response from the Virtual Machine. This would have made the server more scalable, which is of importance as the server is meant to run 24/7 for a couple of months in order to do effective research on IoT infections and attacks.

6 Conclusion

In this project, the implementation described in Pa et al. (2016) has been reproduced. Furthermore, the steps taken in the development process have been clearly outlined and different implementation choices have been motivated. In addition to the system components, an implementation of Telnet has been made for more fine grained control over the Telnet interaction. Three different QEMU architectures have been implemented in the IoTBOX which are able to simulate different types IoT devices. All communication between the FrontEndResponder and the IoTBOX is logged and stored for future analysis. However, some concessions have been made due to the limited time window. In a future iteration of this work, more Telnet options and QEMU architectures could be implemented to serve more Telnet clients. Also the stand-alone capabilities of the IoTBOX can be implemented to provide more analytic capabilities.

References

- Graham, R. D. (2021). masscan. <https://github.com/robertdavidgraham/masscan>.
- Lueth, K. L. (2020). State of the iot 2020: 12 billion iot connections, surpassing non-iot for the first time. <https://iot-analytics.com/state-of-the-iot-2020-12-billion-iot-connections-surpassing-non-iot-for-the-first-time/>.
- Michael, M. (2019). Attack landscape h2 2018: Attack traffic increases fourfold. <https://blog.f-secure.com/attack-landscape-h2-2018/>.
- Naik, S. and Maral, V. (2017). Cyber security — iot. In *2017 2nd IEEE International Conference on Recent Trends in Electronics, Information Communication Technology (RTEICT)*, pages 764–767.
- Pa, Y., Suzuki, S., Yoshioka, K., Matsumoto, T., Kasama, T., and Rossow, C. (2016). Iotpot: A novel honeypot for revealing current iot threats. *Journal of Information Processing*, 24:522–533.

Rapid7 (2021). Pattern recognition for hosts, services, and content. <https://github.com/rapid7/recog>.

This appendix entry contains the `readme.md` that is in the root of the repository:

A IoT HoneyPot

A.1 Supervisor:

Misha Glazunov M.Glazunov@tudelft.nl

A.2 Group members:

Adriaan de Vos - 4422643 - adriaan.devos@gmail.com

Peter Elgar - 5396328 - p.w.j.elgar@student.tudelft.nl

Suzanne Maquelin - 5402840 - s.l.maquelin@student.tudelft.nl

Wouter Zonneveld - 4582861 - w.r.zonneveld@student.tudelft.nl

A.3 Requirements

- Docker \geq 19

A.4 Repository Structure

`./iotpot/` folder contains the Python code for our IoT Honeypot.

`./database/` folder contains Python scripts that we used to populate our MongoDB database with banners.

`./qemu/` folder contains bash scripts that are related to qemu managing.

`Dockerfile` is used to build and run the IoT Honeypot.

A.4.1 IoTHoneypot Setup

On your host-pc:

- For building: `docker build -t iotbox .`
- For running: `docker run -it iotbox`

In the IoTHoneypot Docker Instance:

- To boot up the qemu instances, run `./qemu/qemu-setup.sh`.
- To run our IoTHoneypot server, move to `/iotpot/` and run `python3 main.py`.

A.5 Preparations for Development workspace

Requirements:

- Python3.8
- Python3 pip
- Python3 venv

Creation of virtual environment and installing dependencies:

```
python3 -m venv env
source env/bin/activate
python3 -m pip install -r requirements.txt
```