Universitat Politècnica de Catalunya

Escola Politècnica Superior d'Enginyeria de Manresa

HoneyIo4

The construction of a virtual, low-interaction IoT Honeypot

Treball Final de Grau

Alejandro Guerra Manzanares

Supervisor: Senior Researcher PhD. Hayretdin Bahsi - TTÜ
Remote supervisor: Dr. Francisco del Águila López - UPC

Index of Contents

Acknowledgements

I would like to thank Senior Investigator and Cybersecurity Specialist PhD. Mr. Hayretdin Bähsi for its invaluable and selfless support along this project without whom could have not be accomplished. Thanks for your hours working in my project selflessly, with your always kind and careful attitude and availability. It has been a pleasure to learn and carry out this project with one among the best cybersecurity specialists. Thanks for your time and for introducing me in the cybersecurity field.

I would like also to thank my remote supervisor, Dr. Francisco del Águila López for create the basis without whom it could not have been developed. It seems that many hours struggling with challenging and never ending practices have had an invaluable utility when developing this project. Thanks for introducing me in the passionate field of networking and internet.

The Internet (of things) Era

Human life has changed dramatically since the arrival of the new millennium. First was the single computer, which arrived to solve human problems and automatize complex processes (e.g.: the deciphering of *Enigma* code in II World War), next it evolved to the interconnection of computers and information sharing, also called computer networks or *networking*, thanks to Cold War and the development of the very first scratch and embryo of Internet, ArpaNet, by the U.S. Department of Defense. Although investigators like Gordon Corera State that "the computer was born to spy. That the first computer was created in secret to aid intelligence work, but all computers (and specially networked computers) are also uniquely useful for – and vulnerable to – espionage" [1], networking and internet is nowadays not exclusive to computers, but for all kind of daily use and traditional devices that have now interconnection capabilities with internet and other devices, the well-known IoT, Internet of Things. Cyber is now everywhere. Smart or connected devices are now not only computers but cellphones, TV's, coffee makers, washing machines and whatever you can imagine, including extreme examples like airplane engines or oil pipes. According to a report from Gartner, there will be 8.4 billion connected *things* in 2017, up 31 percent from 2016, and will reach 20.4 billion by 2020 [2]. Our life will be definitely digitally connected. Social relations have now evolved to the digital world with the boom of social media: Facebook, Instagram, Twitter, Linkedin, Whatsapp, etc…Creation of new concepts like e-governance, smart cities, smart cars, etc. We buy things from the other side of the world via credit card (cash is not fashionable any more). There is no doubt that internet is now everywhere at every time, thus we are increasingly more dependent of the cyber element.

The New Cyber (criminal) field

The creation and expansion of the digital field has brought also new challenges and threats. Real (or analogic) world bad guys continue doing their criminal acts in the analogic way. Some of these are not possible in the digital field, but some of them can be easily adopted and with more impunity for the criminal in the cyber or digital world.
Acts like scamming, stealing, fraud, damage, cheating...can be easily done in the computer world (no matter if you are a script kiddie or a tech expert), by combining the newness of the field and the general naïve of the people in the digital world. Threats in the digital world can come from thousand miles from the physical place and sometimes difficult to be aware of. Real world threats are easy to feel (e.g.: armed-assault robbery) but it is difficult and requires more skills and experience to perceive and recognize the threat from that unusual but not suspicious email that tells you to check your banking online account via that easy to click link, everything seems harmless. But the reality is just the opposite, as we are each time more and more dependent of technology, our life is becoming more digital (credit card passwords, bank accounts, social media accounts, email addresses, etc.) most of our personal information is on the cloud and we are losing the control of it, sometimes totally.

How cybercriminals attack. Defense mechanisms

Cyberattacks can be classified in a simple way as targeted (if the victim has been

explicitly chosen and everything has been done in order to attack that specific target) or untargeted (if the victim is not explicitly chosen, randomly attacked for meeting specific criteria or having done some specific thing, e.g.: clicking an advertisement). All these attacks, although have different characteristics has some common steps or phases (called cyber kill chain in the case of advanced persistent threats) in order to be performed successfully by the attacker: reconnaissance, scanning, gaining access, maintaining access and covering tracks. Reconnaissance is the phase where the information about the objective is gathered from open or other sources in order to find define the target; scanning phase involves directly looking for entry points in the target system, by using network scanners and gathering information about vulnerabilities; gaining access phase involves the exploitation of such vulnerabilities found while maintain access involves actions in order to ensure and get access to the broken system in the future and finally, covering tracks involves hide tracks and any other information suspicious that the system has been broken into. The main importance of these steps is that by knowing how the attackers attack, the defenders can defend themselves wisely and effectively.

Defense mechanisms against cyberattacks can be briefly categorized as Active Cyber Defense (ACD) or Passive Cyber Defense (PCD). Passive defenses are defined to include all kind of “measures taken to reduce the probability of and to minimize the effects of damage caused by hostile action without the intention of taking the initiative” [3]. Traditionally used mechanisms are IDS, IPS, Firewalls, AntiVirus, etc. which react to hacker activities, without any kind of proactivity. In the other hand, active defenses are proactive measures taken for “detecting or obtaining information as to a cyber intrusion, cyber-attack, or impending cyber operation or for determining the origin of an operation that involves launching a preemptive, preventive, or cyber counter-operation against the source” [4]. An optimal defensive approach should combine both perspectives, concretely, “passive defenses are a necessary 10 component of a well-designed cyber defense program, but they are no longer sufficient to address increasingly sophisticated threats” [5]. ACD are “a set of operating concepts that all involve taking the initiative and engaging the adversary in some way” [5]. A well-known deceptive example of this approach is honeypots, honeynets and honeytokens.

The honey world: honeypots, honeynets and honeytokens

Honeypots

Deception is an ACD technique employed by both offensive and defensive actors. “Attackers use deception to lure victims to visit fake websites or to enter Personally Identifiable Information (PII) into contrived emails. The defending teams also employed deception, with the intent of having attackers steal files which were intentionally fake or which pointed the attacker in the wrong direction. Deception could serve as a deterrent as the attacker wasted valuable time on purposefully misleading information, with potentially diminished returns seen as an end goal” [5]. In the defensive field, honeypots are used as the most well-known deception technique. A honeypot is basically a “security resource whose value lies in being probed, attacked, or compromised” [6]. A

system set up as a decoy to entice attackers. But not exclusively "to distract hackers, they're also great at tracking down all sorts of information" [7]. As a lure, their main purpose is to attract the hacker, to detect and study the behavior of the attacker in order to enhance and improve system and network cyber security defenses. Any connectivity attempt, activity or external interaction with the honeypot should be considered suspicious since a normal user will never interact with the honeypot nor have any awareness of its existence. Thus, helps to reduce false positives that are mainly common in traditional security defenses (passive defenses such as IPS, IDS, etc.). Honeypots can be classified basically in two ways: purpose and interaction level.

Regarding its purpose, honeypots can be categorized as production or research capabilities. Briefly, "production honeypots protect an organization, while research honeypots are used to learn" [6]. This distinction is merely conceptual and not absolute, the same honeypot can be used as production or research, depending on the use and the built-in functionalities.

Production honeypots main aim is to protect the environment of the organization that deployed it and help to mitigate the risk of attackers. They create a safer environment by detecting attacks. Production honeypots are "easier to build and deploy than research honeypots because they require less functionality" [6]. Paradoxically, the so called production "honeypots do not have any production value" [15] for itself for the company, as it is not providing any real productive value for it. "They are just a security resource whose value lies in being probed, attacked, or compromised, and any new activities or network traffic that comes from the honeypot indicates that it has been successfully compromised. As such, a compromise is very easy to detect on honeypots. False positives, as commonly found on traditional intrusion detection systems, do not exist on honeypots" [15]. Production honeypots usually fake services or systems in a limited way. Thus, they also provide less risk for the network if the honeypot is compromised but the information gathered about the attack or attackers is also very limited. They are easier to maintain and provide protection to the organizations network by providing prevention, detection and reaction mechanisms.

Research honeypots main aim is to learn about the hackers. They are not mainly used in organizations, but their "primary mission is to research the threats organizations may face, such as who the attackers are, how they are organized, what kind of tools they use to attack other systems, and where they obtained those tools" [6]. Research honeypots could be considered as a counterintelligence mechanism to enhance protection against evolving and skilled attackers. The gathered information helps to improve the security of the resources indirectly. They are often deployed by universities, security research companies, military, government agencies, etc. They are more complex than production honeypots, "just because to learn about attackers, you need to give them real operating systems and applications with which to interact. This gives us far more information than simply what applications are being probed." [6]. This also provide greater risk and also more management efforts. In fact, "research honeypots could potentially reduce the security of an organization, since they require extensive resources and maintenance" [6].

Regarding to the interaction level, a honeypot can be categorized as: low-interaction, medium-interaction and high-interaction. This categorization refers basically to the built-in capabilities and the interaction provided with the attacker. "Are you hoping to

catch the attackers in action and learn about their tools and tactics? If so, you need to build a complex honeypot that gives the attacker a complete operating system with which to interact. Are you primarily interested in detecting unauthorized activity, such as scanning? For this you can build a simple honeypot that merely emulates a variety of services in operation. If someone connects to these servers, then you know it is most likely unauthorized activity. Are you hoping to capture the latest worm for analysis? Then you need a customized honeypot with the intelligence to interact with the worm and capture the worm activity" [6]. As can be seen, for each goal the capabilities should be different and honeypots offer a huge variety of options and different functionalities. Furthermore, by increasing the interaction and capabilities of a honeypot, it provides more extensive and detailed information but also more potential damage can be done by the attacker.

Low-interaction honeypots are usually production honeypots (they are used to help protect organizations) that simulate, emulate or fake one or more services or functionalities. They are easy to install and, maintain and deploy and they do not provide actual real services or functionalities. Their interaction with the attacker is very limited (e.g.: allowing scanning and basic connection into a port/service, but not more) thus the information and risk are very limited, as the attackers' interaction capacities are also very limited. "There is also no operating system for the attacker to interact with, so the honeypot cannot be used to attack or monitor other system" [6]. Attackers can only interact with the limited amount of emulated services. The information provided by these honeypots is mainly limited to transactional information that is "the data collected about the circumstances of the attack but not about the attack itself" [6]. Examples of transactional information are time and date of attack, source IP address and source port of the attack and destination IP address and destination port of the attack. Additional information can be gathered, depending whether the emulated service allows any kind of interaction with the attacker or not. So, low-interaction honeypots main purpose is to capture well-known behavior. "The attacker acts a certain way, and the honeypot responds in a predetermined manner" [6]. They are not useful if the attacker vector is not known or against unexpected behavior or attacks.

Medium-interaction honeypots are a step up in the interaction scale. They "offer attackers more ability to interact than do low-interaction honeypots but less functionality than high-interaction solutions" [6]. They are designed to provide certain responses beyond the limited capabilities of a low-interaction honeypot would give and they expect and manage certain level of activity by the attacker. They are still emulating a service or functionality so the risk is still quite low, there is no full operating system for the attacker to interact. This kind of honeypots are usually more time consuming to install and configure than low-interaction honeypots, as they are more complex in nature. "Often these solutions are not prepackaged commercial products. Instead, they involve a high level of development and customization from an organization" [6]. As the attacker has more interaction with the honeypot, the deployment must be done with secure mechanisms to ensure that the hacker cannot use the honeypot to harm other systems in the network and the increased capabilities are not vulnerable to exploitation by an attacker. They can gather more information than low-interaction honeypots and learn how the attacker behave in the specific attack (e.g.: how the attacker compromised the system, how elevated privileges, etc.).

High-interaction honeypots are in the top level of the interaction scale. High-interaction

honeypots are real systems with completely operative operating systems and applications that are given to the attacker without any emulation or restriction. “They give us a vast amount of information about attackers, but they are extremely time consuming to build and maintain, and they come with the highest level of risk” [6]. High-interaction honeypots provide a huge amount of information and are mostly used in research environments to learn about attackers, attacks and its behavior. “To create such an environment, few to no modifications can be made to the actual honeypots. Most often standard builds are no different than production systems found in many organizations today. The only thing that defines these systems as honeypots is that they have no production value-their value lies in being probed, attacked, or compromised” [6]. As they are real operating systems, their risk level is really high, so their deployment has to be done in a very secure manner, making them hard to maintain and deploy. For that reason, they are usually placed in controlled environments, usually behind a firewall. The firewall allows the attacker to compromise the honeypot but it prevents the attacker to use the honeypot to launch attacks. So, a great deal of work it is needed to build a firewall with proper rulebases. Installation and configuration are difficult and time consuming, not only for the honeypot, as they involve other technologies such as firewall or Intrusion Detection Systems. Everything has to be perfectly fitted. Along with this complexity comes a high level of risk. “The more interaction we allow the attacker, the more that can go wrong. However, once implemented correctly, the high-interaction honeypot can give insight into attackers that no other honeypot can” [6].

Level of interaction	Work to install and configure	Work to deploy and maintain	Information gathering	Level of risk
Low	Easy	Easy	Limited	Low
Medium	Involved	Involved	Variable	Medium
High	Difficult	Difficult	Extensive	High

Figure 1. Honeypots interaction level comparison [6].

To illustrate a complete step by step differentiated scenario for previous points, a low-interaction honeypot emulating a web server could place a listener on port 80 or port 443 with no other interaction needed. To upgrade the honeypot to a medium-interaction would need implement a complete emulation of the webserver and all its capabilities. Finally, a high-level interaction honeypot would require the complete deployment of a full and real web server, without any restriction or limitations of interaction and additional security features (like a properly configured firewall) to avoid the use of the honeypots as an attack weapon for the real attacker into our local area network.

Additionally, honeypots can be categorized as physical or virtual. A physical honeypot is a physical machine with a corresponding IP address allocated on the network. “A virtual honeypot is hosted on another machine that responds to network traffic directed to the virtual honeypot” [11].

Last but not least, a new differentiation has been created recently, by differentiating traditional honeypots as server honeypots, such as the ones previously introduced that expose (by simulation or real deployment) server services and wait calmly to be attacked, as shown in figure 2a, and a “newer technology called client honeypots that

deals with a different attack vector" [15]. Client honeypots deal with an attack vector that is not detectable by server honeypots, the so-called client side attacks. Those are "assaults of clients that originate from malicious servers. This could be a seemingly harmless visit to a website with a browser. As part of a server's response to a client request, the malicious website might serve code that is targeted at exploiting a vulnerability of the browser as shown in figure 2b. As a result, a mere visit to the website might leave a machine exploited with malware. Client honeypots are designed to interact with servers and detect the attacks of servers" [15]. Client honeypots are few fish in the vast honeypots sea, but they are not least in importance. These honeypots deal with important client side attacks performed by malicious web servers which client honeypots interact with by driving a web browser on the honeypots system. Compromises are mainly detected by monitoring changes to a list of files, directories and system configuration after the interaction between the client honeypot and the suspicious malicious server.

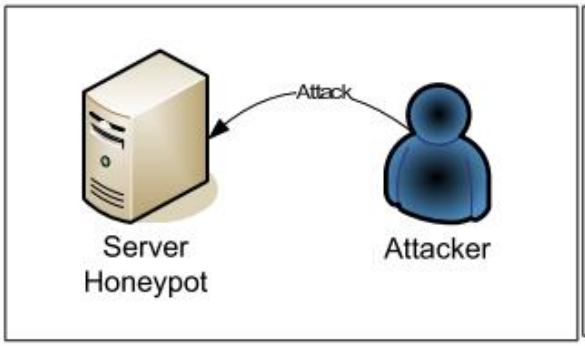

Figure 2a. Server honeypot architecture [14].

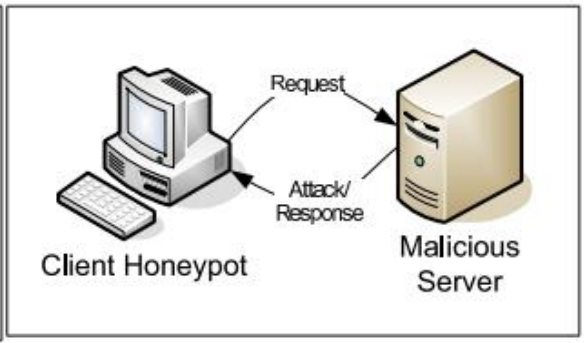

Figure 2b. Client honeypot architecture[14]

Honeynets

Honeynets can be considered as a special type of honeypot. Specifically, they are considered a high-interaction honeypot with the main purpose of capture extensive information of threats and attacks. In brief, a honeynet is a "network of real computers for attackers to interact with. These victim systems (honeypots within the honeynet) can be any type of system, service, or information you want to provide" [8]. The systems are high interaction, so they are composed by real systems, applications and other services waiting for attackers to interact with them. A honeynet can contain one or more honeypots that are not actual production systems, so any interaction with these systems implies malicious or unauthorized activity. "Any connections initiated inbound to your honeynet are most likely a probe, scan, or attack. Any unauthorized outbound connections from your honeynet imply someone has compromised a system and has initiated outbound activity" [8]. Thus analyzing is made simple with the use of a honeypot, there is no need to look through thousands of megabytes of firewall or IDS logs to know that something happened. Furthermore, false positives are almost eliminated from the field. As honeynets are nothing more than an architecture of honeypots, that has to be deployed correctly. The key element of the honeynet architecture is what is called as honeywall. "This is a gateway device that separates your honeypots from the rest of the world. Any traffic going to or from the honeypots must go through the honeywall. This gateway is traditionally a layer 2 bridging device, meaning the device should be invisible to anyone interacting with the honeypots" [8]. As an example, next diagram shows a typical honeynet architecture. The honeywall has

3 interfaces: eth0 and eth1 separate the honeypots from everything else (they are bridged interfaces that have no IP stack) while eth2 (that is optional) has an IP stack allowing for remote administration. "This network is reachable via the honeywall gateway, a stealth inline network bridge that closely monitors and controls the network data flow to and from the honeypots in the network. Data capture includes network traffic captured on the honeywall gateway, system event data captured in logs, and keylog data gathered by a stealth keylogger on the honeypot systems" [9].

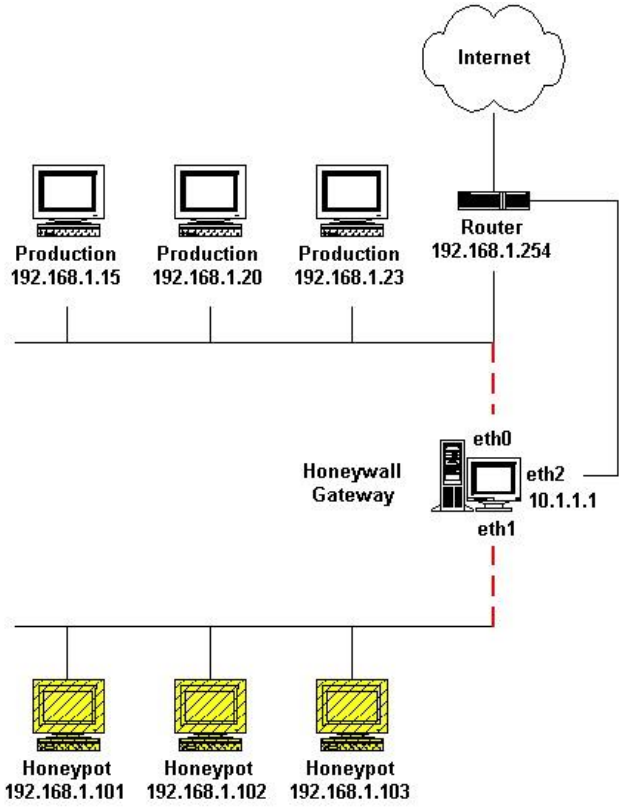

Figure 3. Example of a honeynet [8].

Honeynets create a highly controlled network that allows the complete control and monitoring of what is happening in every moment within it. There you can deploy whatever target system you want (there is no actual limit, e.g.: routers, printers, servers, etc.) and simply watch what is happening inside the network, how intruders interact with the honeypots systems and gather all the information. As a tradeoff, the risk is extremely high as has the same risks involved by a single high-interaction honeypot but incremented by a more complex architecture that has to be fitted perfectly on each of the possible failure points.

Honeytokens

Honeytokens are a special kind of honeypots. They are honeypots that are not computers or physical resources. Until this point, all of the reviewed manifestations of honeypots have been computers, that is the traditional manifestation but not the only one. Actually, a honeypot it is only a resource that is created as a lure for attackers to interact with. In that sense, a honeytoken is exactly a honeypot which is not a computer. It can be any kind of digital entity or resource. For example, a credit card number, Word document, a database entry or even a bogus login. "Honeytokens come in many shapes

or sizes but they all share the same concept: a digital or information system resource whose value lies in the unauthorized use of that resource. Just as a honeypot computer has no authorized value, no honeytoken has any authorized use" [10]. As a usual honeypot, any interaction with the honeytoken most likely represents unauthorized or malicious activity. They can be used to test, lure and detect suspected violations and other behaviors against law or security policies.

HoneyD: A widely used low-interaction honeypot

HoneyD (http://www.honeyd.org) is an open source honeypot solution designed for Unix systems. It was first released in April 2002, created and maintained by Niels Provos of the University of Michigan. Begotten as a virtual low-interaction production honeypot, it simulates TCP and UDP services and is used to detect attacks or unauthorized activity by emulating a wide variety of services. It only expects interaction at network level, simulating only the network stack and not the rest of services of an operating system like file system manipulation or I/O operations. HoneyD is a Unix daemon that simulates the TCP/IP stack of many operating systems, supporting TCP, UDP and ICMP. "It listens to network requests destined for its configured virtual honeypots. Honeyd responds according to the services that run on the virtual honeypot. Before sending a response packet to the network, the packet is modified by Honeyd's personality engine to match the network behavior of the configured operating system personality" [11]. HoneyD powerful features include the creation of virtual realistic networks topologies (e.g.: adding routers with configurable link features such as latency and packet loss).

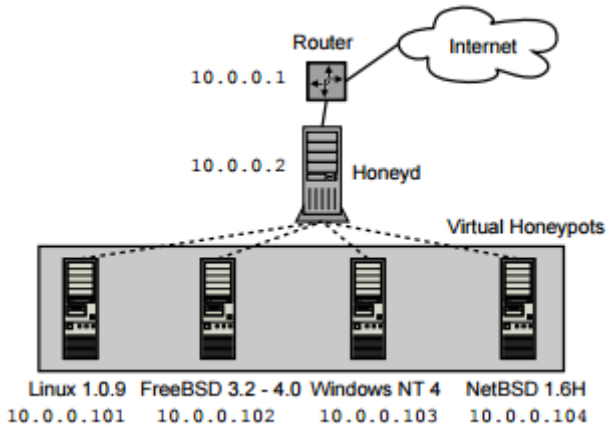

Figure 4. Example of virtual honeypot creation with HoneyD [11].

"Honeyd receives traffic for its virtual honeypots via a router or Proxy ARP. For each honeypot, Honeyd can simulate the network stack behavior of a different operating system" [11]. When a packet arrives to the daemon for one of the virtual honeypots, it is received and processed by a central packet dispatcher. The dispatcher checks the length of the IP packet and verifies its checksum. The daemon only accepts packets for ICMP, TCP or UDP. Other protocols are discarded. Then, "the dispatcher queries the configuration database for a honeypot configuration that corresponds to the destination IP address. If no such configuration exists, the default template is used. Then the dispatcher

calls the protocol specific handler with the received packet and the corresponding honeypot configuration" [11] to create a proper response.

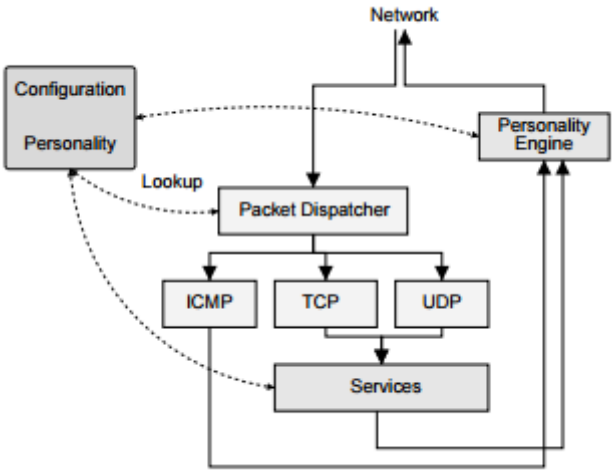

Figure 5. HoneyD architecture [11].

Before any packet is sent to the network, it is processed by the personality engine. The main purpose of the personality engine is to adjust the packet's content so that it seems to originate from the network stack of the configured operating system. The concept of personality refers to the network stack behavior of a virtual honeypot. The daemon uses the Nmap fingerprint list as a reference. Each fingerprint has a format similar to the following example:

```
Fingerprint IRIX 6.5.15m on SGI O2
TSeq(Class=TD%gcd=<104%SI=<1AE%IPID=I%TS=2HZ)
T1(DF=N%W=EF2A%ACK=S++%Flags=AS%Ops=MNWNNTNNM)
T2(Resp=Y%DF=N%W=0%ACK=S%Flags=AR%Ops=)
T3(Resp=Y%DF=N%W=EF2A%ACK=O%Flags=A%Ops=NNT)
T4(DF=N%W=0%ACK=O%Flags=R%Ops=)
T5(DF=N%W=0%ACK=S++%Flags=AR%Ops=)
T6(DF=N%W=0%ACK=O%Flags=R%Ops=)
T7(DF=N%W=0%ACK=S%Flags=AR%Ops=)
PU(Resp=N)
```

Figure 6. Fingerprint example [14].

"We use the string after the Fingerprint token as the personality name. The lines after the name describe test results for nine different tests" [11]. These tests will be covered in a deeper way in future chapters. HoneyD additional features include the creation of virtual routing topologies and a simple CLI to create in an easy manner highly customizable honeypots. Without any doubt, HoneyD is a good choice and starting point for deploy and maintain low-interaction honeypots and honeynets.

IoT devices attacks: the arise of botnets

Hacking is an always evolving and adapting art of exploitation of devices. The rise of IoT devices has opened a new interesting field for the bad guys to play in. "Since years,

it is known that many Internet of Things (IoT) devices are vulnerable to simple intrusion attempts, for example, using weak or even default passwords" [12]. These weaknesses (mostly permitted unconsciously by the human owners of the devices) are exploited easily by the hackers that compromise the device, install a backdoor or a rootkit, and take permanent control of it. Usually, the infected device, becomes part of a botnet (named as bot or zombie). A botnet is, by definition, "a collection of computers, connected to the internet, that interact to accomplish some distributed task" [13]. The botnet is controlled by a single person or a group of people (called the botmaster), under a command and control structure (C&C). A C&C refers to single (or multiple) servers (called C&C servers) that allow and simplify the control of all the specific hosts within the botnet by the botmaster, which could be hundreds or thousands. The botmaster sends commands through C&C server to the bot, that controls all the bots within the botnet. Botnets are often used for illegal purposes, often related with money issues (earn by the hacker or loss for the victims). Attacks like Distributed Denial of Service (DDoS), Spam/Phishing, Ad-ware, Click Fraud, etc. are often committed using botnets. In 2012, "Carna botnet revealed that there were more than 1.2 million open devices that allowed logins with empty or default credentials. In January 2014, an Internet-connected fridge was discovered as a part of a botnet sending over 750,000 spam e-mails. In December 2014, online DDoS services (i.e. booters) knocked down Sony and Microsoft's gaming networks, presumably powered by thousands of compromised IoT devices such as home routers" [12]. A special case in the botnet world is Mirai botnet. Mirai is an army of hundreds of thousands of IoT devices, most of them cameras (but also printers and routers) that is the cause of the latest worse DDoS attacks in 2016: ~600mbps against Brian Krebs website, ~1 terabit against OVH and ~1.2 terabit against DYn. Zombie devices (bots) are scattered all over the world, but specially in Vietnam, Brazil, United States, China and Mexico (which sum up over 50% devices of the botnet). [22-25]

Figure 7. Example of Mirai botnet infected device – bot or zombie device [14].

Mirai scans the net looking for interesting IP addresses. It looks for unsecured IoT devices that could be remotely accessed via easily guessable login credentials – usually factory default usernames and passwords (e.g.: admin/admin), and also brute forces guessing passwords (dictionary attacks based on simple login/password credentials). Once the device is broken into, Mirai malware execute scripts that eradicate other worms or Trojans in the device, was well as prohibiting remote connection attempts of the hijacked device (e.g.: closing SSH, Telnet and HTTP ports). The main use of Mirai, is, as seen before, "to launch HTTP floods and various network (OSI layer 3-4) DDoS

attacks". Mirai is capable of launching "GRE IP and GRE ETH floods, as well as SYN and ACD floods, STOMP (Simple Text Oriented Message Protocol) floods, DNS floods and UDP flood attacks" [14]. That's not all, Mirai is becoming each day more and more powerful. So, as a result, from an attacker point of view, IoT devices are attractive because they are usually 24/7 online (not like PC), have no antivirus protection installed, and tend to have weak login passwords that provide easy access to attackers to powerful shells (like BusyBox). This picture creates a new challenge for security specialists, and places IoT devices' security as an important new area of cybersecurity research.

The Need of an IoT HoneyPot

"DDoS attacks increase over 125 percent year over year" [16], "the size (bandwidth usage) of DDoS attacks went up 1233% over the past 5 years and more than 7900% compared to the attacks registered in 2005" [17], "they are simply taking advantage of the sheer number of unsecured IoT devices that are deployed today" [18]. Those are only 3 easy to find examples of the daily increasing IoT botnet landscape. Statistics are clear, DDoS attacks are increasing in size, duration and frequency, and mostly thanks to the increasing number of IoT devices that are being compromised and attached to a botnet. Mirai, the most well-known IoT botnet, has an army of hundreds of thousands devices, ready to attack. Botnets are easy to hire in the dark web and can cause hundreds or thousands of euros of losses in each attack. But not only the money, reputation is an active that companies can loss easily after an attack and that is really hard to restore. This landscape is only happening with 10 billion devices in the field, this value will double by 2020. The landscape can only get worse, much worse. For this reason, IoT device' security and awareness must be placed in an important place in the cybersecurity research picture. There is an actual and future need of security in such devices if we do not want to be spectators of catastrophic attacks. So, nowadays there is a lack of IoT honeypots that could help to investigate, learn and detect such kind of attacks and, of course, protect devices within and outside companies' internal networks.

The answer: HoneyIo4

My contribution to this need is HoneyIo4. HoneyIo4 is a IoT production low-interaction virtual honeypot that simulates 4 IoT devices in order to lure and detect unauthorized access to the networks and prevent the compromise of IoT devices in local area networks. As the research in IoT security is still not much developed, there is a need of new tools that should fill the gap that there is nowadays. HoneyIo4 pretends to be a contribution to the development of IoT security tools in this area.

About HoneyIo4

HoneyIo4 is a production low-interaction virtual honeypot that simulates 4 common IoT devices: camera, printer, video game console and cash registering machine. The user can choose, via an easy-to-use Graphical User Interface (GUI) that is mainly a web browser running on a Linux machine, which device wants to simulate and some additional options (services/ports open/closed, delay and running time).

As a production honeypot, its main purpose is active defense (protection). There is no expected interaction with it, so every interaction could be considered as unauthorized access, and thus, a potential threat for our network. As a virtual honeypot, it simulates using a Linux Virtual Machine a real world's device. As a low-interaction honeypot, it only simulates some part of the service/device, and not the whole capabilities (such as banner advertisement, login/connection simulation, etc.). More specifically, it tricks network scanners (like Nmap, etc.) by simulating some IoT Operating Systems, so the scanners detect the honeypot as some IoT device. Thus, the main idea behind HoneyIo4 is to provide 4 different options of IoT Operating System's fingerprint simulation to fake hackers attack's second phase: scanning. While defeating scanning, by providing wrong OS information, the rest of the attack cycle is redirected in a wrong way, that it turns the attack unsuccessful.

When attacking, hackers follow five main steps: reconnaissance, scanning and enumeration, gaining access, maintaining access and covering tracks. Reconnaissance is about information gathering of objectives and targets of attack, usually getting the information from open sources and not interacting with the target network or hosts directly (not testing packets are sent in this phase). Scanning and enumeration phase involves using the information gathered in reconnaissance phase with actively apply tools and techniques to gather more in-depth information about real targets on the network, it usually involves direct interaction with the target network or hosts by sending packets or similar interaction mechanisms. Gaining access or exploitation involves the exploitation of the vulnerabilities/attack vectors found in previous phase and entering/compromising the target machine. Maintaining access' main purpose is to ensure that the hacker has a way back to the compromised machine in the future. Finally, covering tracks is the way that the hacker ensures that neither users not IT professionals are aware about their activity. These 5 steps are a cyclic feedback process that never ends, as shows diagram below.

Figure 8. The five phases of hacking

As a result, each step provides valuable information to the next step, having special importance the very first ones. Although main idea of honeypot is more on collecting

information about the attackers and their methods, it can also have an impact in attack set up. Reconnaissance and specially scanning create the attack planification that is executed lately (exploitation and post-exploitation). So, if the planification is wrong, the attack will be sure unsuccessful, wasting the time and efforts of the attacker in an impossible mission. Applying this to HoneyIo4, if scanning provides wrong information, by simulating something that is not actually capable of being exploited, every step after will lead to defeat for the hacker. That is the aim of HoneyIo7, that is the purpose of their creation: lure and trick.

HoneyIo4 implementation

HoneyIo4 main idea is way simplistic of Honeyd. IoT Operating Systems (henceforth OS) are simulated not in every aspect of them but in their TCP/IP network stack. Network scanners are used by hackers to know about target operating systems and services running on the machine. Gathered this information, they just look for an attack vector to exploit any known vulnerability (if exists). Network scanners like the well-known and widely used NMAP provide OS detection information by using TCP/IP stack fingerprinting. NMAP main use is not black hat, as OS detection helps system administrators and penetration testers to:

- **Determine vulnerability of target hosts. "**The best way to verify that a vulnerability is real is to exploit it, but that risks crashing the service and can lead to wasted hours or even days of frustrating exploitation efforts if the service turns out to be patched. OS detection can help reduce these false positives. Scanning your whole network with OS detection to find machines which need patching before the bad guys do" [19].

- **Tailor exploits.** Even after you discover a vulnerability in a target system, OS detection can be helpful in exploiting it. "Exploit vulnerabilities often require custom-tailored shellcode with offsets and assembly payloads generated to match the target OS and hardware architecture. In some cases, you only get one try because the service crashes if you get the shellcode wrong" [19].

- **Network inventory and support**. An inventory (know what OS are running in each machine) can also "be useful for IT budgeting and ensuring that all company equipment is accounted for" [19].

- **Detection of unauthorized and dangerous devices.** Regular scanning can detect unauthorized devices (added without permission to the network) for investigation and containment.

- **Social engineering.** By knowing what technology is the machine running you can fake the support center and trick the user if the naïve administrator assumes that only an authorized engineer is calling and would know such information about his system.

NMAP Remote OS Detection and Fingerprinting

NMAP performs OS detection by sending a bunch of TCP and UDP packets to the remote hosts and analyzes (performing specific tests) the responses to those packets.

"After performing dozens of tests such as TCP ISN sampling, TCP options support and ordering, IP ID sampling, and the initial window size check, Nmap compares the results to its database of more than 2,600 known OS fingerprints and prints out the OS details if there is a match" [20]. OS fingerprints are created with the results of these tests (once analyzed the responses), and they are nothing more than the specific results of that tests and values that a particular OS provides to the Nmap proof packets. Nmap fingerprints are stored in a specific database, called nmap-os-db database, which is a publicly available resource [21]. The OS fingerprint format is a tradeoff between human comprehension and brevity. Brevity (compacted format) is needed to optimize and speed-up the Nmap comparison process. When Nmap is running, and OS detection has been enabled, it stores the target fingerprint in memory (tests' responses) and reads back and compares it to the ones (all of them) located in nmap-os-db database (called *reference fingerprints*). If there is a match, so the OS is known, it displays the results to the user in plain and in an easy to understand detailed readable format (figure 9). If there is not a specific match, Nmap displays after the scanning a *subject fingerprint*, which is a raw ASCII-encoded version fingerprint with the particular tests' results (it can also be forced to display it if there is a match by using Nmap debug mode, -d) of the unidentified machine (figure 10).

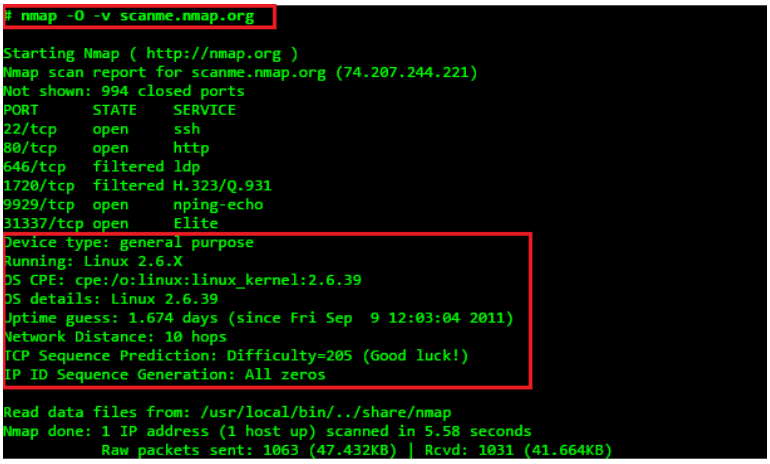

Figure 9. Nmap OS detection results – matched fingerprint [22].

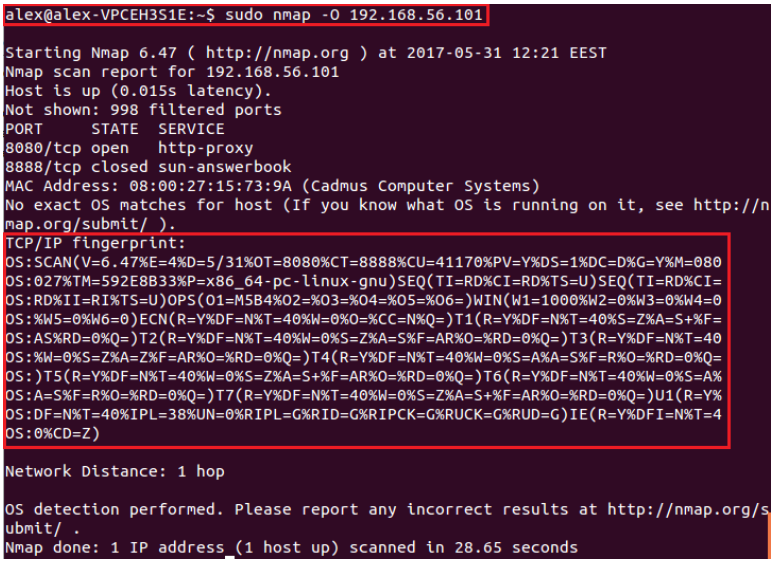

Figure 10. Nmap OS detection results – unknown fingerprint.

Nevertheless, the reality is quite different. An exact OS fingerprint is quite difficult nowadays, mostly because the huge amount of different flavors that exist from the same theoretical OS (e.g.: Ubuntu versions). So, the main and average result of the scanning process is a guess, with some percentage that indicates the amount that the system fits some specific fingerprint on Nmap database (that is likely to be the real OS running on the system). To show this, I performed a search in Shodan [26], which is a webpage where you can find Internet connected devices, querying about Ubuntu running devices. I found a specific IP that showed that it was running Ubuntu OS and Apache web service. Then, I performed from my machine an Nmap OS detection scanning, so the results are shown in Figure 11.

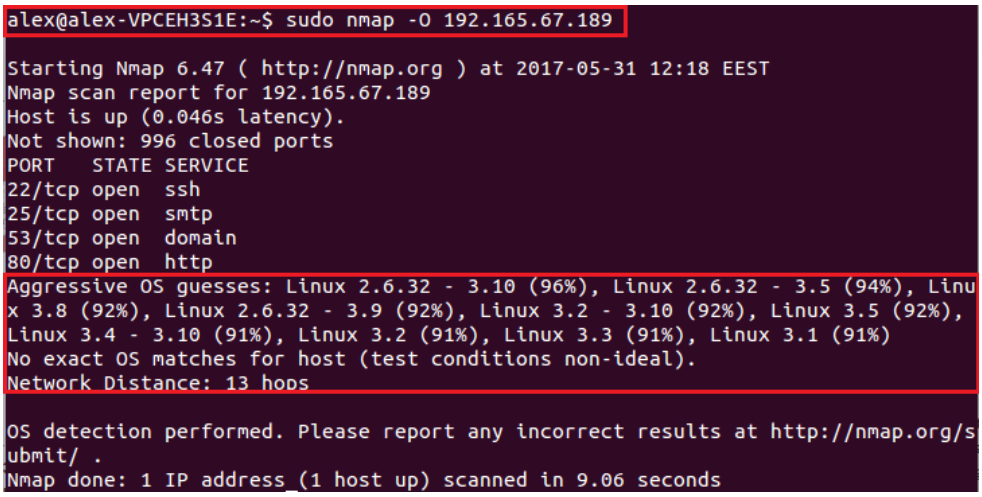

Figure 11. Nmap OS detection results – partial matching results.

As we can see, although there is not a total matching, there is a really good match on Linux OS (more than 90%), especially in some specific versions. So, as can be seen, the most common real output of Nmap OS detection is an OS guess (or Aggressive OS guess) that provides some valuable information about the most likely OS running in target machine (with a high true positive ratio). If Nmap OS detection scanning doesn't show up any high percentage match, it is also possible to force Nmap to show interesting results (using the flag parameter --osscan-guess). Nmap-os-db is a big fingerprint database (that has not information of every networked OS, but the most common ones) made by Nmap users (who submit them) but is rarely changed by them. "Adding or modifying a fingerprint is a moderately complex process and there is usually no reason ever to remove one. The best way to get an updated version of the OS database is to get the latest release of Nmap" [27]. Finally, note that although Nmap can be used as a normal Linux user, remote OS detection capabilities require root privileges to run it (sudo).

Remote OS detection process

Nmap remote OS detection process involves direct interaction with the target machine. More specifically, Nmap sends probe packets to the target machine, analyzes some specific fields of the replies, performs some specific tests and process packet data, and stores that information in its memory (the so called subject fingerprint). Afterwards, it compares this stored fingerprint to the ones in nmap-os-db database and it provides the output (OS running information if there is a match or subject fingerprint if there is none). Nmap OS fingerprinting process involves the sending of up to 16 packets (TCP,

UDP and ICMP probes) to known open and closed ports of the target machine. “These probes are specially designed to exploit various ambiguities in the standard protocol RFCs. Then Nmap listens for responses. Dozens of attributes in those responses are analyzed and combined to generate a fingerprint. Every probe packet is tracked and resent at least once if there is no response. All of the packets are IPv4 with a random IP ID value. Probes to an open TCP port are skipped if no such port has been found. For closed TCP or UDP ports, Nmap will first check if such a port has been found. If not, Nmap will just pick a port at random and hope for the best” [28]. The fingerprinting process consists of up to 16 probes sent in a specific order:

1) First part of fingerprinting process is the sending of 6 TCP probes which responses are used to define the first 4 lines of the fingerprint. These probes are sent every 100 milliseconds, so it takes 500 milliseconds to send them all. Timing is important for some of the algorithms applied to the responses (initial sequence numbers, IP ID’s and TCP timestamps), as they are time dependent. “Each packet send in this first step is a TCP SYN packet to a detected open port in the target remote machine. The sequence and acknowledgment numbers are random (but saved so Nmap can differentiate responses)” [28]. The main difference between the packets is the TCP options they use and the TCP window field (see [28] for more detailed information).

   The responses and results of the answers to this packets define the first four fingerprint lines: SEQ (contains results based on sequence analysis of the probe packets: GCD, SP, ISR, TI, II, CI, TS and SS), OPS (contains the TCP options received in the replies, numbered from 01 to 06), WIN (contains window sizes received, numbered from 01 to 06) and T1 (contains various specific tests for packet #1: named R, DF, T, TG, W, S, A, F, O, RD and Q).

```
SEQ(SP=FB-107%GCD=1-6%ISR=100-110%TI=I%II=I%SS=S%TS=U)
OPS(O1=M5B4%O2=M5B4%O3=M5B4%O4=M5B4%O5=M5B4%O6=M5B4)
WIN(W1=2000%W2=2000%W3=2000%W4=2000%W5=2000%W6=2000)

T1(R=Y%DF=N%T=FA-104%TG=FF%S=O%A=S+%F=AS%RD=0%Q=)
```

Figure 12. Example of first lines of a Nmap OS fingerprint [27]

2) Following these 6 packets, Nmap sends two ICMP echo request (aka ping) packets to the target machine. The differences between both of them are minimum but enough to test ICMP reply behavior of the target machine. ICMP request ID and sequence numbers are incremented by one from the previous query values. The result of both probes form the last line of a fingerprint (IE) that contains up to five fields: R, DFI, T, TG and CD. T and CD are extracted from the first probe and the rest are done in combination with the data of the two replies.These ICMP probes follow immediately after the TCP sequence probes to ensure valid results of the shared IP ID sequence number test (named SS and specified in the SEQ line; see [28] for more detailed explanation).

```
IE(DFI=S%T=FA-104%TG=FF%CD=S)
```

Figure 13. Example of last line of a Nmap OS fingerprint [27]

3) Next single packet sent my Nmap tests for explicit congestion notification

(ECN) support in the target machine TCP stack. "ECN is a method for improving Internet performance by allowing routers to signal congestion problems before they start having to drop packets. It is documented in RFC 3168. Nmap tests this by sending a SYN packet which also has the ECN CWR and ECE congestion control flags set" [28].
The results of this reply include the fields: R, DF, T, TG, W, O, CC and Q. See [28] for more detailed explanation about the tests.

```
ECN(R=Y%DF=N%T=FA-104%TG=FF%W=2000%O=M5B4%CC=N%Q=)
```

Figure 14. Example of ECN line of a Nmap OS fingerprint [27]

4) Following ECN test, Nmap sends 6 TCP probe packets. Except one (last one changes the window value) all data sent in the six packets is the same, but the flags are different in all cases. First 3 packets (named as T2-T4) are sent to an open port and last 3 packets (named T5-T7) are sent to a closed port (if there is one). Every single response to these single packets is recorded and create a new line in the fingerprint (T2-T7), which fields named as: R, DF, T, TG, W, S, A, F, O, RD and Q (more information in reference [28]).

```
T2(R=N)
T3(R=Y%DF=N%T=100%TG=FF%W=1FC4%S=O%A=O%F=A%O=%RD=0%Q=)
T4(R=Y%DF=N%T=100%TG=FF%W=2000%S=A%A=Z%F=R%O=%RD=0%Q=)
T5(R=Y%DF=N%T=100%TG=FF%W=0%S=Z%A=S+%F=AR%O=%RD=0%Q=)
T6(R=Y%DF=N%T=100%TG=FF%W=0%S=A%A=Z%F=R%O=%RD=0%Q=)
T7(R=Y%DF=N%T=100%TG=FF%W=0%S=Z%A=S%F=AR%O=%RD=0%Q=)
```

Figure 15. Example of T2-T7 line of a Nmap OS fingerprint [27]

5) Finally, as a last test, Nmap sends a UDP packet to a closed port in the target machine. "If the port is truly closed and there is no firewall in place, Nmap expects to receive an ICMP port unreachable message in return" [28].
The response to this packet defines line U1 in Nmap fingerprints, with fields: R, DF, T, TG, IPL, UN, RIPL, RID, RIPCK, RUCK, and RUD (check out reference [28] for more detailed information).

```
U1(DF=N%T=100%TG=FF%IPL=38%UN=0%RIPL=G%RID=G%RIPCK=G%RUCK=0%RUD=G)
```

Figure 16. Example of U1 line of a Nmap OS fingerprint [27]

Nmap documentation defines how the test and algorithms are applied in order to get the results of each single field in each specific line in the fingerprint. A detailed explanation of each test is out of the scope of this project (see reference [28]) but they have to be well known in order to trick Nmap OS detection mechanism, as HoneyIo4 does.

Understanding a fingerprint: the whole picture

Reference fingerprints

Up to here, we have defined the lines of a fingerprint, which is basically what Nmap uses to compare and guess the OS running in the target machine. But a complete

fingerprint contains additional lines, as can be seen in next example.

```
# D-Link DGS-3024
Fingerprint D-Link DGS-3024 switch
Class D-Link | embedded || switch
CPE cpe:/h:dlink:dgs-3024 auto
SEQ(SP=8E-98%GCD=1-6%ISR=92-9C%TI=I%II=RI%SS=O|S%TS=1)
OPS(O1=M5B4NW0NNT11%O2=M5B4NW0NNT11%O3=M5B4NW0NNT11%O4=M5B4NW0NNT11%O5=M5B4NW0NNT11%O6=M5B4NNT11)
WIN(W1=2000%W2=2000%W3=2000%W4=2000%W5=2000%W6=2000)
ECN(R=Y%DF=N%T=19-23%TG=20%W=2000%O=M5B4NW0%CC=N%Q=)
T1(R=Y%DF=N%T=19-23%TG=20%S=O%A=S+%F=AS%RD=0%Q=)
T2(R=N)
T3(R=Y%DF=N%T=19-23%TG=20%W=2000%S=O%A=S+%F=AS%O=|M5B4NW0NNT11NNLLLLLLLLLL%RD=0%Q=)
T4(R=Y%DF=N%T=19-23%TG=20%W=0%S=A%A=Z%F=R%O=%RD=0%Q=)
T5(R=Y%DF=N%T=19-23%TG=20%W=0%S=Z%A=S+%F=AR%O=%RD=0%Q=)
T6(R=Y%DF=N%T=19-23%TG=20%W=0%S=A%A=Z%F=R%O=%RD=0%Q=)
T7(R=Y%DF=N%T=19-23%TG=20%W=0%S=Z%A=S%F=AR%O=%RD=0%Q=)
U1(DF=N%T=19-23%TG=20%IPL=38%UN=127|128%RIPL=G%RID=G%RIPCK=G%RUCK=G%RUD=G)
IE(DFI=S%T=19-23%TG=20%CD=S)
```

Figure 17. Complete Nmap *Reference Fingerprint* [27]

As can be shown in picture above, there are two differentiated areas. Bottom part (highlighted in green) is the results of the probe tests, explained before, and what Nmap uses to compare with the subject fingerprint. Nmap tries to match subject fingerprint test lines with reference fingerprint test lines. Top part (highlighted in red) is used to describe the OS they represent, and part of this information is the one provided by Nmap as an output when there is a match. First commented line is not used, and it is only a partial reference. "The Fingerprint line first serves as a token so Nmap knows to start loading a new fingerprint. Each fingerprint only has one such line. Immediately after the Fingerprint token (and a space) comes a textual description of the operating system(s) represented by this fingerprint. These are in free-form English text, designed for human interpretation rather than a machine parser. Nevertheless, Nmap tries to stick with a consistent format including the vendor, product name, and then version number" [28]. As Nmap developers state, "in an ideal world, every different OS would correspond to exactly one unique fingerprint. Unfortunately, OS vendors don't make life so easy for us. The same OS release may fingerprint differently based on what network drivers are in use, user-configurable options, patch levels, processor architecture, amount of RAM available, firewall settings, and more. Sometimes the fingerprints differ for no discernible reason. While the reference fingerprint format has an expression syntax for coping with slight variations, creating multiple fingerprints for the same OS is often preferable when major differences are discovered" [28]. The class lines (can be allocated more than one) specify four fields: vendor, OS family, OS generation and device type, all separated by the pipe symbol (|). Vendor is the company that makes an OS or device; OS family includes products such as Windows, Linux, IOS, OpenBSD, etc. (when is not clear, *embedded* keyword is used); OS generation defines more detailed information (e.g.: version 2.4.X in case of Linux or Vista in Windows case). If embedded was used or this was not known, this field is omitted in some fingerprints; device type is a broad classification of devices used by Nmap, that can be readen here [29], e.g: video game console, router, etc. Finally, CPE (Common Platform Enumeration) gives equivalents of class lines, referencing hardware and software (see more [30]).

Subject fingerprints

As stated in previous pages, *subject fingerprints* contain the information provided and results to Nmap tests for the specific target machine after the scanning process. This

fingerprint is then compared with thousands of reference fingerprints in order to match them. Test result expressions (lines) is what is being compared and are almost the same in both fingerprints (if there is a match) but, in order to match all instances (hardware can modify the response) of a particular OS, reference fingerprints are a little bit more generalized in order to match all of them, so they usually contain a value range that is accepted in each specific field instead of one exact value. Subject fingerprints, as they are used by Nmap, have a different format when printed (-d, debug mode) and in order to extract legible information they have to be cleaned a little bit.

```
OS:SCAN(V=5.05BETA1%D=8/23%OT=22%CT=1%CU=42341%PV=N%DS=0%DC=L%G=Y%TM=4A91CB
OS:90%P=i686-pc-linux-gnu)SEQ(SP=C9%GCD=1%ISR=CF%TI=Z%CI=Z%II=I%TS=A)OPS(O1
OS:=M400CST11NW5%O2=M400CST11NW5%O3=M400CNNT11NW5%O4=M400CST11NW5%O5=M400CS
OS:T11NW5%O6=M400CST11)WIN(W1=8000%W2=8000%W3=8000%W4=8000%W5=8000%W6=8000)
OS:ECN(R=Y%DF=Y%T=40%W=8018%O=M400CNNSNW5%CC=N%Q=)T1(R=Y%DF=Y%T=40%S=O%A=S+
OS:%F=AS%RD=0%Q=)T2(R=N)T3(R=Y%DF=Y%T=40%W=8000%S=O%A=S+%F=AS%O=M400CST11NW
OS:5%RD=0%Q=)T4(R=Y%DF=Y%T=40%W=0%S=A%A=Z%F=R%O=%RD=0%Q=)T5(R=Y%DF=Y%T=40%W
OS:=0%S=Z%A=S+%F=AR%O=%RD=0%Q=)T6(R=Y%DF=Y%T=40%W=0%S=A%A=Z%F=R%O=%RD=0%Q=)
OS:T7(R=Y%DF=Y%T=40%W=0%S=Z%A=S+%F=AR%O=%RD=0%Q=)U1(R=Y%DF=N%T=40%IPL=164%U
OS:N=0%RIPL=G%RID=G%RIPCK=G%RUCK=G%RUD=G)IE(R=Y%DFI=N%T=40%CD=S)
```

Figure 18. Raw subject fingerprint output in debug mode [29]

If we eliminate the "OS:" keywords and separate each line appropriately, we end up with the following picture.

```
SCAN(V=5.05BETA1%D=8/23%OT=22%CT=1%CU=42341%PV=N%DS=0%DC=L%G=Y%TM=4A91CB90%
     P=i686-pc-linux-gnu)
SEQ(SP=C9%GCD=1%ISR=CF%TI=Z%CI=Z%II=I%TS=A)
OPS(O1=M400CST11NW5%O2=M400CST11NW5%O3=M400CNNT11NW5%
    O4=M400CST11NW5%O5=M400CST11NW5%O6=M400CST11)
WIN(W1=8000%W2=8000%W3=8000%W4=8000%W5=8000%W6=8000)
ECN(R=Y%DF=Y%T=40%W=8018%O=M400CNNSNW5%CC=N%Q=)
T1(R=Y%DF=Y%T=40%S=O%A=S+%F=AS%RD=0%Q=)
T2(R=N)
T3(R=Y%DF=Y%T=40%W=8000%S=O%A=S+%F=AS%O=M400CST11NW5%RD=0%Q=)
T4(R=Y%DF=Y%T=40%W=0%S=A%A=Z%F=R%O=%RD=0%Q=)
T5(R=Y%DF=Y%T=40%W=0%S=Z%A=S+%F=AR%O=%RD=0%Q=)
T6(R=Y%DF=Y%T=40%W=0%S=A%A=Z%F=R%O=%RD=0%Q=)
T7(R=Y%DF=Y%T=40%W=0%S=Z%A=S+%F=AR%O=%RD=0%Q=)
U1(R=Y%DF=N%T=40%IPL=164%UN=0%RIPL=G%RID=G%RIPCK=G%RUCK=G%RUD=G)
IE(R=Y%DFI=N%T=40%CD=S)
```

Figure 19. Cleaned subject fingerprint [29]

As we can see, this format is really similar to the test part of a reference fingerprint except for the SCAN line, which is information stored by Nmap regarding the scanning process and machine (not about the target machine) and which explanation is out of the scope of this chapter. See [29] for more information about this information line.

Fooling Nmap OS Detection

In order to trick Nmap OS Detection, the process of how it scans and analyses the responses is critical (explained in previous pages). Nmap is just an example of a network scanner (the most used one) but all of them work in a similar way, analyzing specific OS implementations' responses to TCP/IP stack probes. Next sections explain

the tools and platforms used to achieve this goal.

Virtual Machine Platform: Linux

HoneyIo4 main platform is a Linux based virtual machine OS. More specifically is Ubuntu 15.04 version. This chose was done by affinity and easy usage by the developer, which is used to this Linux flavor more than others, but HoneyIo4 can be used in other Linux VM environments. As it is a VM, it is portable and can run in Windows machines as well with the appropriated configuration and hypervisor tool (like Oracle VirtualBox, VMWare, QEMU, etc). In my case, I used Oracle VirtualBox installed in an Ubuntu 14.04 machine (which acts as the host machine of the Hypervisor that runs the Ubuntu 15.04 VM machine, aka honeypot).

Programming HoneyIo4: Python

HoneyIo4 is programmed in Python programming language. Python is high level, interpreted, general purpose, open-source language that supports both functional and object-oriented programming paradigms. It is a powerful language, with a huge community behind and an extremely easy-to-understand syntax. As it is an interpreted language, it needs an interpreter in order to run, that can be easily installed from Linux repositories or https://www.python.org. To run a simple python script (named example.py) with no parameters, just issue: **python example.py** in Linux CLI and it will be executed properly. Once selected the platform, programming language and knowing in deep Nmap scanning process, I started coding planification. Discovering Nmap scanning process was the hardest part, making lots of dumb scans to real machines and operating systems in order to analyze and see the packets flow in Wireshark environment, as can be seen in pictures below.

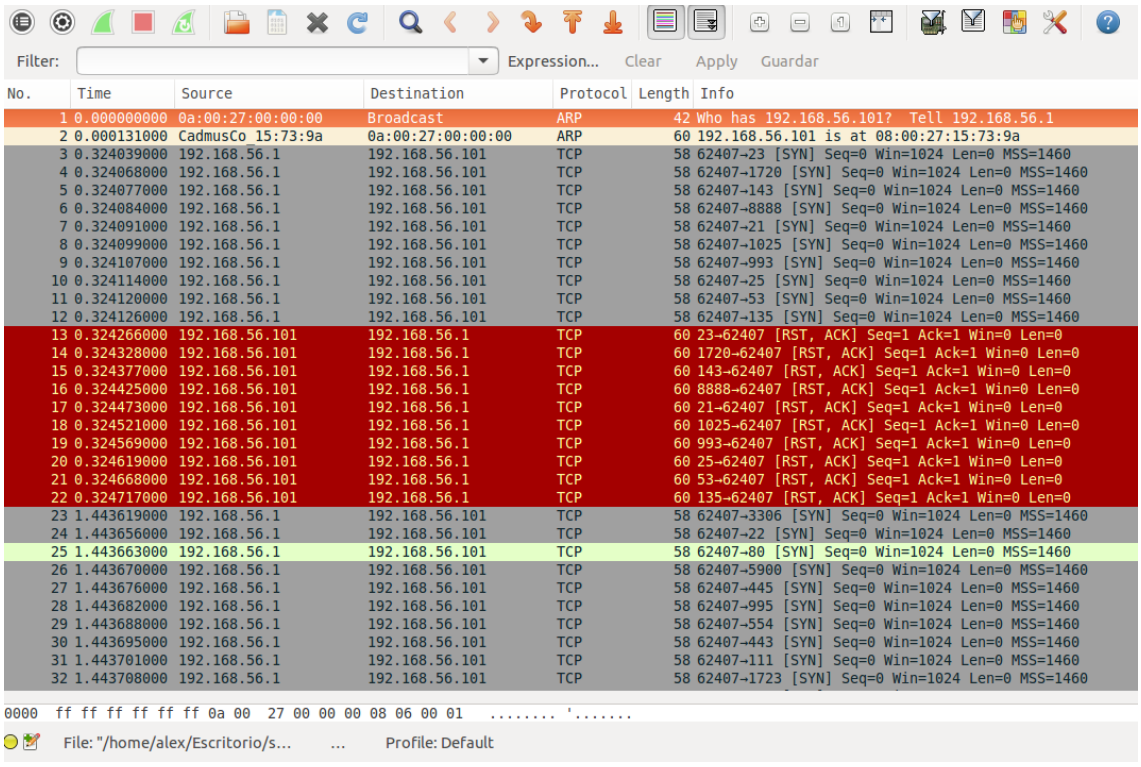

Figure 20. Port scanning by Nmap

The experimental setup was formed by two machines: a host machine (attacker) and a Virtual Machine connected to the same LAN (victim). The IP of the attacker was 192.168.56.1, using a Ubuntu 14.04 machine. Victim machine was created by an hypervisor (Virtualbox) running a Ubuntu 15.04 machine, whith IP 192.168.56.101. The only open port in the victim machine was 8080 and the rest of them were assumed to be closed. Attacker performed a simple OS scanning with Nmap tool. When the environment was set up, I started the experiment.

Prior to OS detection scanning, Nmap has to detect open/closed ports as they are the target of probe packets and also it can guess what services are running (in case of well-known ports are used without change). This can be seen in Figure 20, as Nmap sends TCP SYN (using SYN port scanning, IP 192.168.56.1), the host replies with Reset-Ack packet to show closed ports (IP 192.168.56.101). Figure 21 (below) shows one step further in the detection process. In this case, port 8080 was detected as an open port (first three lines of the pcap file, numbered 35-85-86), as there is a SYN packet that has a reply SYN-ACK to establish connection but, then, the scanner aborts and sends a RST packet, so the three-way handshake is not finished and the connection is not established. Then, following packets are the probes sent by Nmap to perform remote OS detection. In Figure 21 we can see, in blocks of three (SYN packet, answer and RST), the first 6 probes (lines 2102 to 2119), ECN (lines 2126-2128) and 3 TCP packets to open port (lines 2129-2131, highlighted in white). The following additional lines are retransmitted packets for whose Nmap did not received an answer yet.

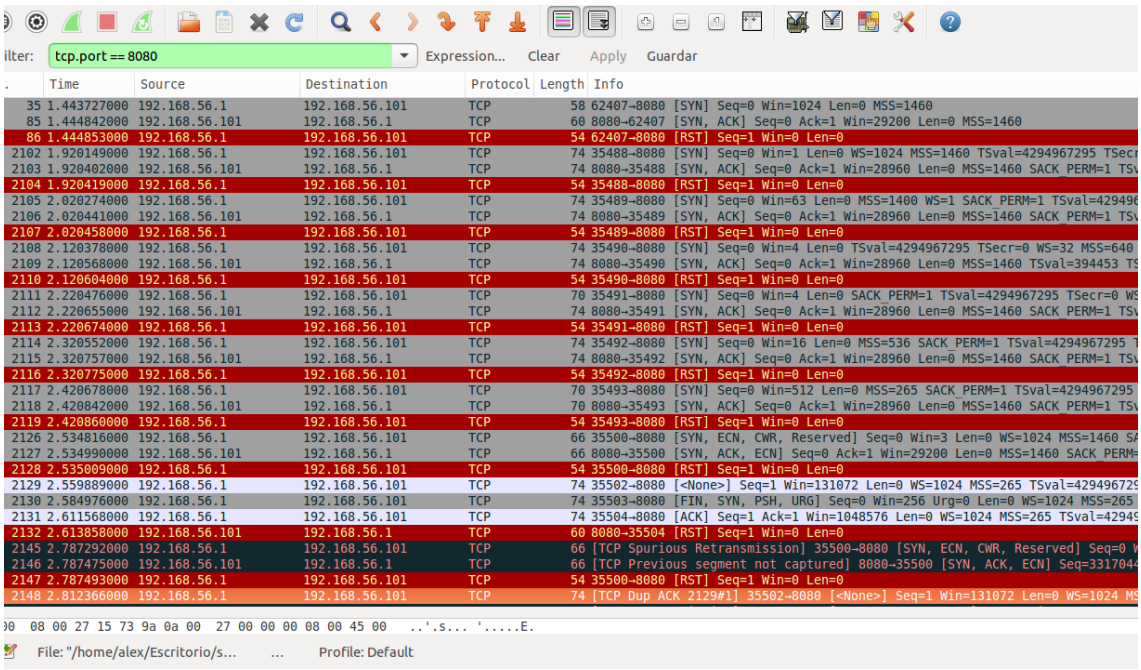

Figure 21. Port scanning and open port probe packets by Nmap

As a filter was applied to Wireshark in the screenshot above, the process seems to behave differently as it was explained. But reality is, as can be seen in number, that following TCP SYN probes, Nmap sent ICMP echo request packets, as can be seen in Figure 22. These are two slightly different ICMP packets that are answered by the target machine. As we can see, Nmap retransmits some of the probes to ensure its application. If we check the line numbers of first ICMP probes, we can see that are 2120-2125,

which are immediately following the numeration of TCP probes and before ECN packet. Figure 23 shows the packets sent to closed port (before packets 2129-2131 in previous screenshot). As Nmap detected in this case 999 closed ports, it chose randomly on closed port to send it. It chose port 1 as can be seen in Figure 23. Following port scanning, we can see the 3 packet probes sent by Nmap to the closed port and its following replies by the target machine (lines 2133 to 2138). Rest are retransmitted.

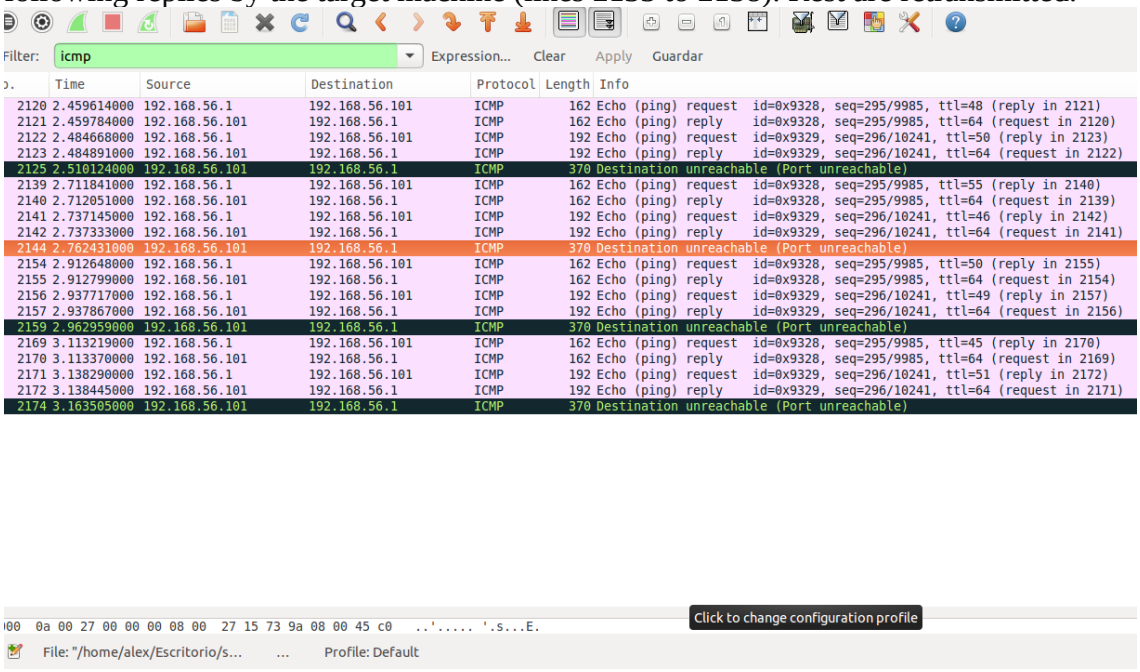

Figure 22. ICMP probe packets sent by Nmap

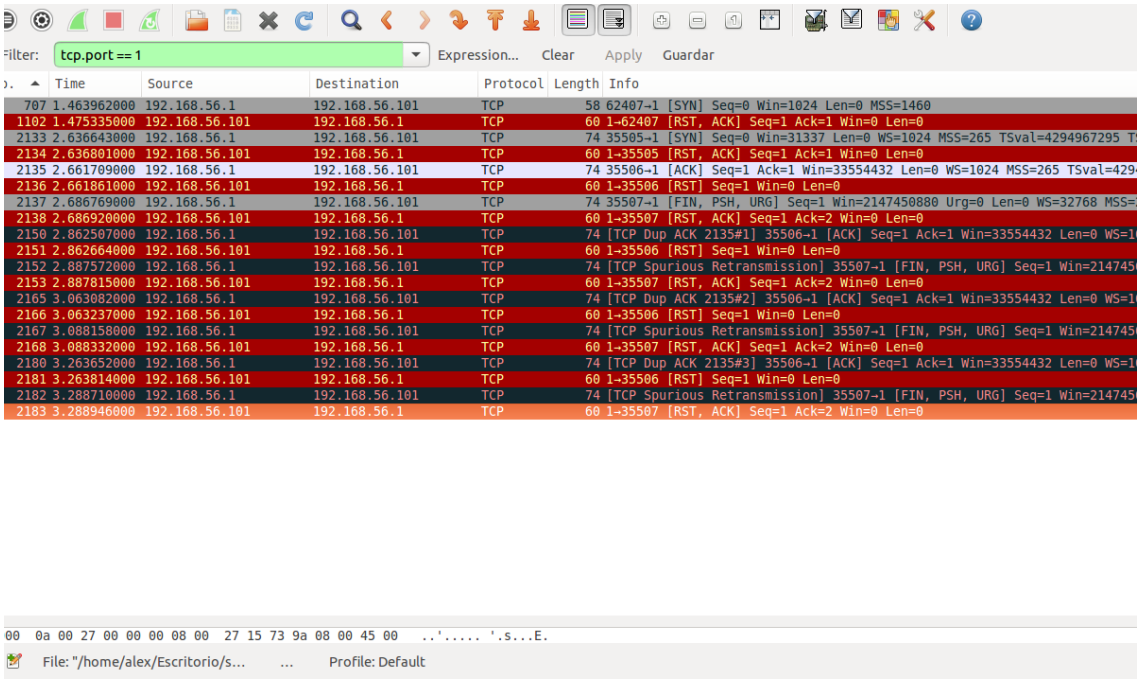

Figure 23. Closed port probe packets sent by Nmap

Finally, as can be guessed there is only missing one probe packet, the one regarding UDP probe. Figure 24 shows that packet with its reply. As we can see, again, Nmap

retransmits and resends more than one time the probe packet. As expected, for a non-firewall filtered port UDP message, the response is an ICMP message: Destination and Port unreachable (ICMP message type 3 code 3). Then, this was used to know the practical implementation of the scanning process, and not only the theoretical one (explained before). As we can see, there are a lot of retransmissions of packets and the process performance respects the theoretical explanation.

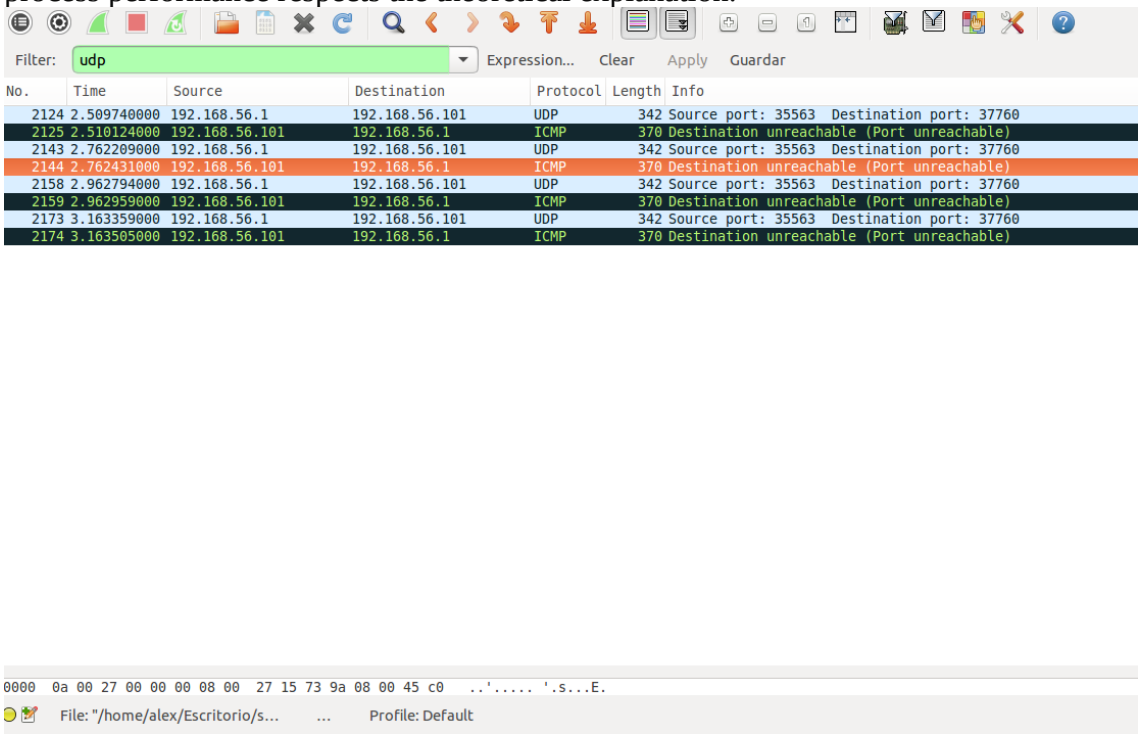

Figure 24. UDP probe by Nmap to a closed port

Once I identified all the packets (up to 16 in repeated waves) sent by Nmap I analyzed the responses done by real machines. In this case, I just coded a listening socket on port 8080 with Python in the target machine (IP 192.168.56.101), executed it, let the OS answer and captured the traffic in the scanner interface (IP 192.168.56.101). As can be seen below, I used socket library in Python to create, bind and use listening mode of the recently created socket.

```
import socket
import sys

# Create a TCP/IP socket
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Bind the socket to the port
server_address = ('192.168.56.101', 8080)
print >>sys.stderr, 'starting up on %s port %s' % server_address
sock.bind(server_address)

# Listen for incoming connections
sock.listen(1)
```

Figure 25. Socket creation with Python socket library

Previous to this step, I worked a with socket communication between the 2 machines, host and virtual machine, in order to workout with Scapy and sockets. I used Scapy for its easy to integrate capabilities within Python code (as it has been created itself in

Python) and at the same time can be run as an autonomous powerful program. Other alternatives are Pypacker, Libcrafter and Ostinato, but all of them lack of integration capabilities in Python code (Scapy allows to use the same commands in a similar way both in CLI and Python code, so its integration is easier). Then, I created a listening socket (aka server), that replied back to a client the information that this sent to the server. For this purpose I created a listening socket in VM machine (IP 192.168.56.101), coded as can be seen in Figure 26. In the other side, the client machine was the real host (IP 192.168.56.1) that was coded using Scapy library. I created a raw TCP packet and sent it to the listening server (VM). As can be seen in Figure 27, packet creation in Scapy is done by adding layers of information, and finally using the function send, which is used to send the datagrams created.

```
import socket
import sys

# Create a TCP/IP socket
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Bind the socket to the port
server_address = ('192.168.56.101', 8080)
print >>sys.stderr, 'starting up on %s port %s' % server_address
sock.bind(server_address)

# Listen for incoming connections
sock.listen(1)

while True:
    # Wait for a connection
    print >>sys.stderr, 'waiting for a connection'
    connection, client_address = sock.accept()
    try:
        print >>sys.stderr, 'connection from', client_address

        # Receive the data in small chunks and retransmit it
        while True:
            data = connection.recv(4096)
            print >>sys.stderr, 'received "%s"' % data
            if data:
                print >>sys.stderr, 'sending data back to the client'
                connection.sendall(data)
            else:
                print >>sys.stderr, 'no more data from', client_address
                break

    finally:
        # Clean up the connection
        connection.close()
```

Figure 26. Listening-echo socket in port

```
1  from scapy.all import *
2
3  send(IP(dst="192.168.56.101")/TCP(sport=80,dport=8080,flags="SA")/Raw(load="Message received"))
```

Figure 27. Client message, sent by Scapy function send( )

To test both, I first executed the listening socket script in the VM, that acted as a server, and it bound the port in listening mode, awaiting for connections. As can be seen in Figure 28.

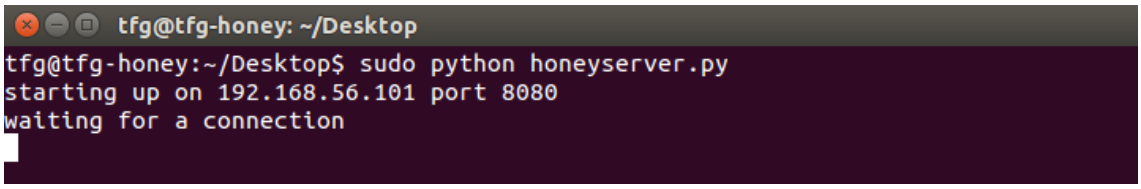

Figure 28. Listening socket waiting for incoming connections (server)

Then, in the other side, host machine, I executed the script acting as a client (connecting to the server). As seen in Figure 29, it was sent, printing out the message Sent 1 packets, that automatically is printed when send() function sends successfully a message.

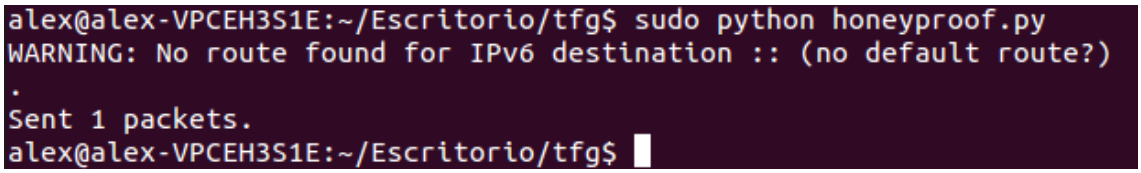

Figure 29. Client sent a message

This test was successful, but then I modified the sending script to also receive messages, using socket library (not Scapy). These tests main purpose was to test connectivity and socket performance. As can be seen in picture below, were successful.

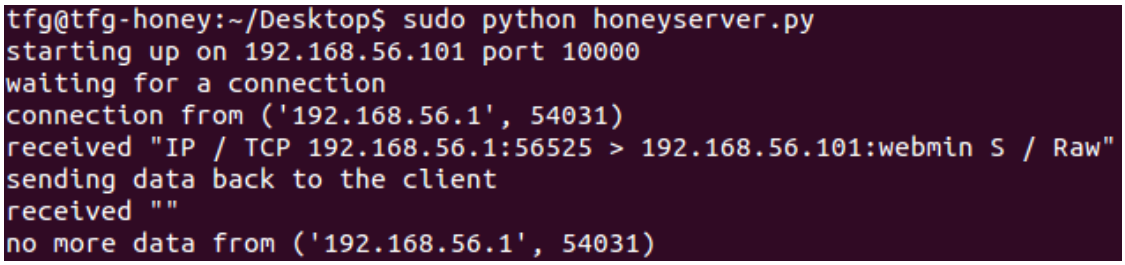

Figure 30. Message received successfully from client, and replied back.

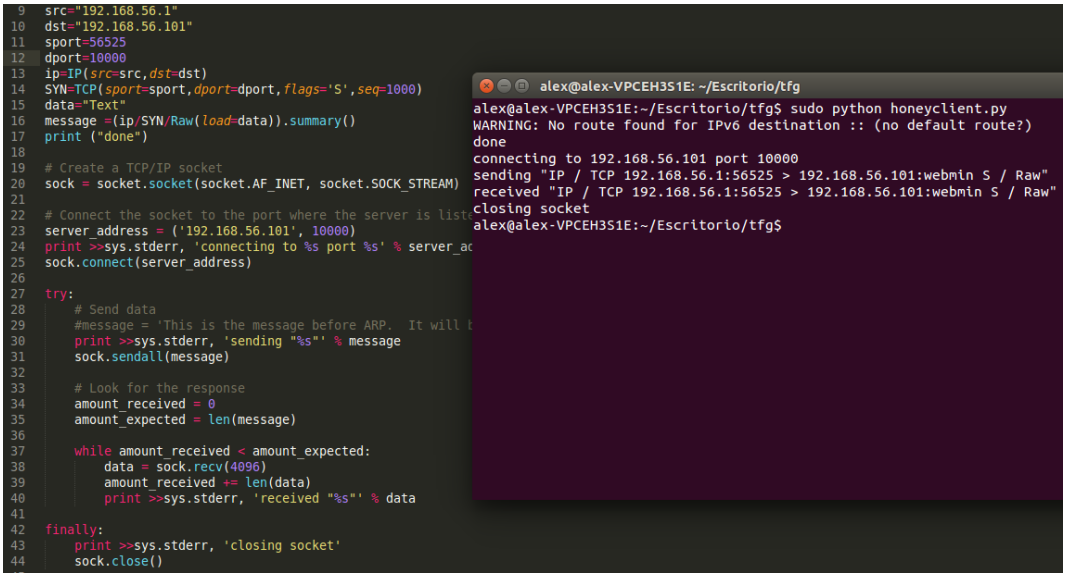

Figure 31. Message received by client sucessfully

Once working out with socket and Scapy, next goal was to fake a real and well-known machine: Ubuntu 15.04 (Linux flavor OS). In order to do that I used the pcap file of the capture that has been explained before (the whole real process), and just copied the packets information, bit by bit. I used this as a reference of working and tunning of the coding process but this was not the real aim of the project, which was to fake 4 IoT OS. First main issue when starting to proof the code against Nmap was that if I used socket creation, the OS answered itself to Nmap, so it was always detected as the real Ubuntu, but because of the real OS answering, and not the faking script. So, as a result, I had to fake it and do not let the OS to answer. For that reason, I used a Python library called python-iptables to modify iptables, which is "the tool that is used to manage netfilter, the standard packet filtering and manipulation framework under Linux. Iptables is used to set up, maintain, and inspect the tables of IPv4 packet filter rules in the Linux kernel. Several different tables may be defined. Each table contains a number of built-in chains and may also contain user- defined chains. Each chain is a list of rules which can match a set of packets. Each rule specifies what to do with a packet that matches. This is called a target, which may be a jump to a user-defined chain in the same table" [34]. Using iptables I could drop every incoming packet and do not let the OS answer. If this is not performed, the OS will reply to all as a closed port, Reset packet. The code for drop the incoming packets is as follows:

```
import iptc

chain = iptc.Chain(iptc.Table(iptc.Table.FILTER), "INPUT")
rule = iptc.Rule()
rule.in_interface = "eth1"
rule.src = "192.168.56.0/255.255.255.0"
target=iptc.Target(rule,"DROP")
rule.target=target
chain.insert_rule(rule)
#chain.delete_rule(rule)
```

Figure 32. Drop packets Python code

As can be seen in Figure 32, a rule is created and inserted into a chain. The incoming interface was eth1 (connected to same subnet of host machine, 192.168.56.0/24), the main rule for drop is every packet coming from source 192.168.56.0/24, that is, every packet sent by the hosts of the LAN and the rule applied: drop the packet. Last two lines insert or eliminate rules inside the chain. As can we see, if we execute the script as it is in Figure 32, the output (adding a few lines of code to print IpTables) is:

```
IP TABLES!
======================
Chain  INPUT
Rule proto: ip src: 192.168.56.0/255.255.255.0 dst: 0.0.0.0/0.0.0.0 in: eth1 out
: None
Matches: Target: DROP
======================
Chain  FORWARD
======================
Chain  OUTPUT
======================
```

Figure 33. Output of executing python-iptables code

This created a new rule that theoretically dropped all packets coming to interface eth1 with packet source 192.168.56.0/24. As can be seen in Figure 34, eth1 is the interface

connected to the same subnet as the host machine. Then I used Wireshark and Nmap again to test the performance of the new applied rule. I performed again a remote OS detection with Nmap and captured all packets in host exiting interface (which was vboxnet0), as can be seen in Figure 35. The results were successful, no answer was found in the Wireshark capture (Figure 36), and Nmap outputs was clear (Figure 37), all ports in the target machine were filtered (no response from the target).

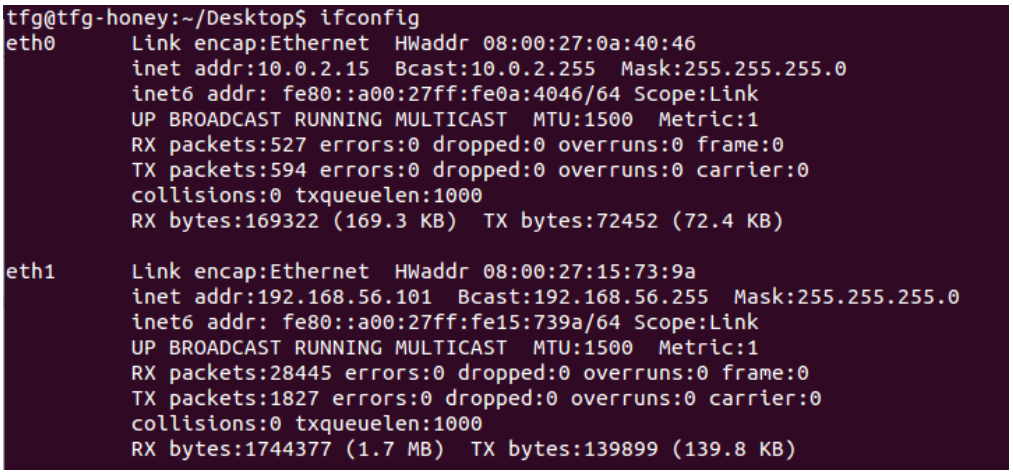

Figure 34. Interfaces in VM, loopback omitted in output

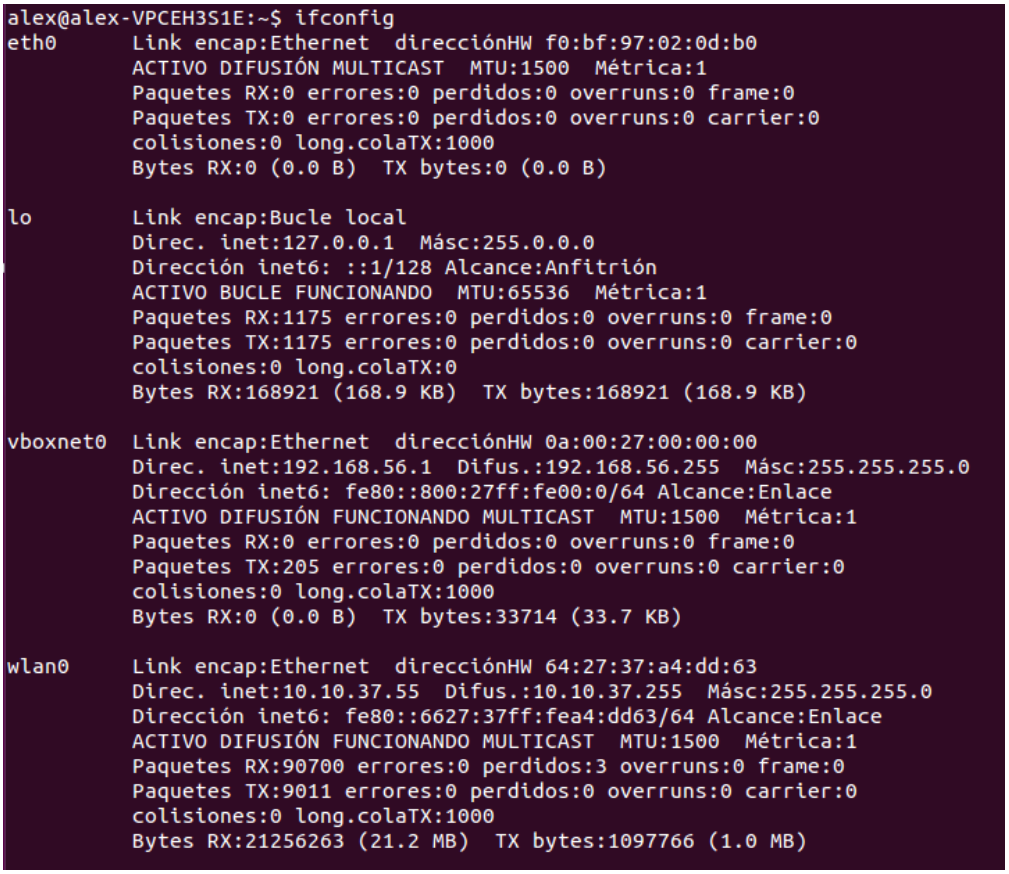

Figure 35. Interfaces in host machine

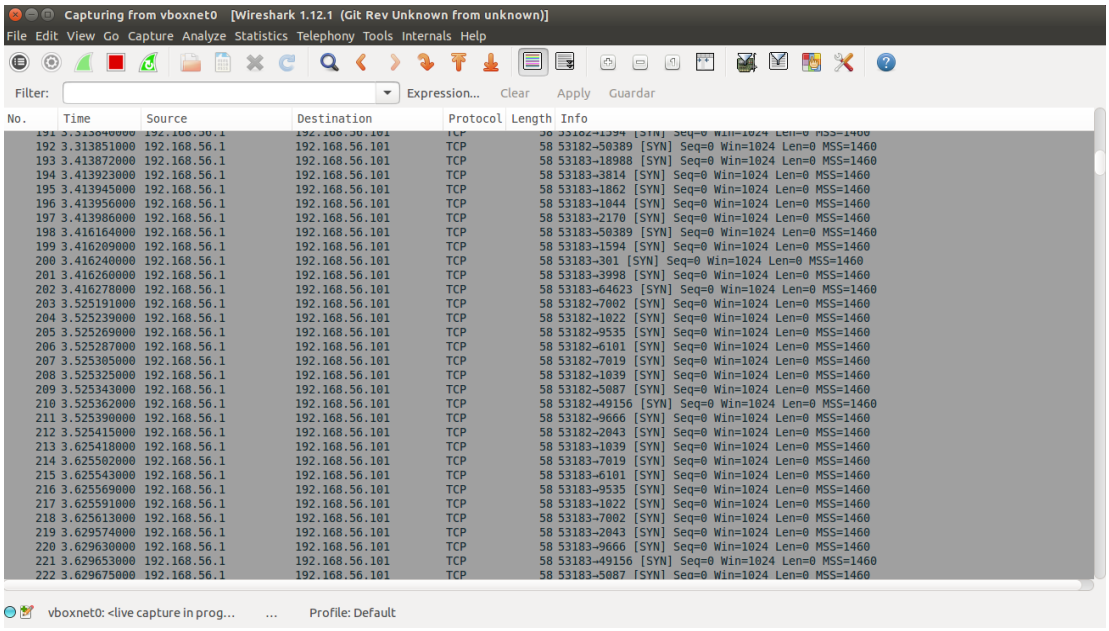

Figure 36. No answer to port scanning, all packets dropped

```
alex@alex-VPCEH3S1E:~$ sudo nmap -O 192.168.56.101

Starting Nmap 6.47 ( http://nmap.org ) at 2017-06-05 16:46 EEST
Nmap scan report for 192.168.56.101
Host is up (0.00028s latency).
All 1000 scanned ports on 192.168.56.101 are filtered
MAC Address: 08:00:27:15:73:9A (Cadmus Computer Systems)
Too many fingerprints match this host to give specific OS details
Network Distance: 1 hop

OS detection performed. Please report any incorrect results at http://nmap.org/s
ubmit/ .
Nmap done: 1 IP address (1 host up) scanned in 25.75 seconds
alex@alex-VPCEH3S1E:~$
```

Figure 37. All ports detected as filtered, as no answer was detected from alive host

Once ensured that all packets were dropped and no answer was received (so it fakes a power off machine), I used *Scapy* library to sniff all packets incoming to the specific interface with IP 192.168.56.101, the one connected to the same LAN as the target machine. Note that this entire project has been done using two machines connected to the same LAN, but Nmap can perform also remote OS detection to public IP addresses, as was shown before in the Shodan case. Scapy is a "powerful interactive packet manipulation program. It allows you to forge or decode packets of a wide number of protocols, send them on the wire, capture them, match requests and replies, and much more" [31]. It can be used as a program itself or integrated in Python code, with the same functionalities available for the developer. Scapy documentation [33] shows the powerful functionalities that it has through examples and guided tutorials. In my case the needs were simple: as my intention was to fake a specific OS, I had a four step process: capture incoming packets (sniff), know which probe packet was (packet interpretation), create an appropriate response (packet creation) and send it back to the scanner machine (reply). As Nmap never stops until it gets the whole fingerprint, I had to perform all this steps in a continuous cycle, as shown in the next flow chart.

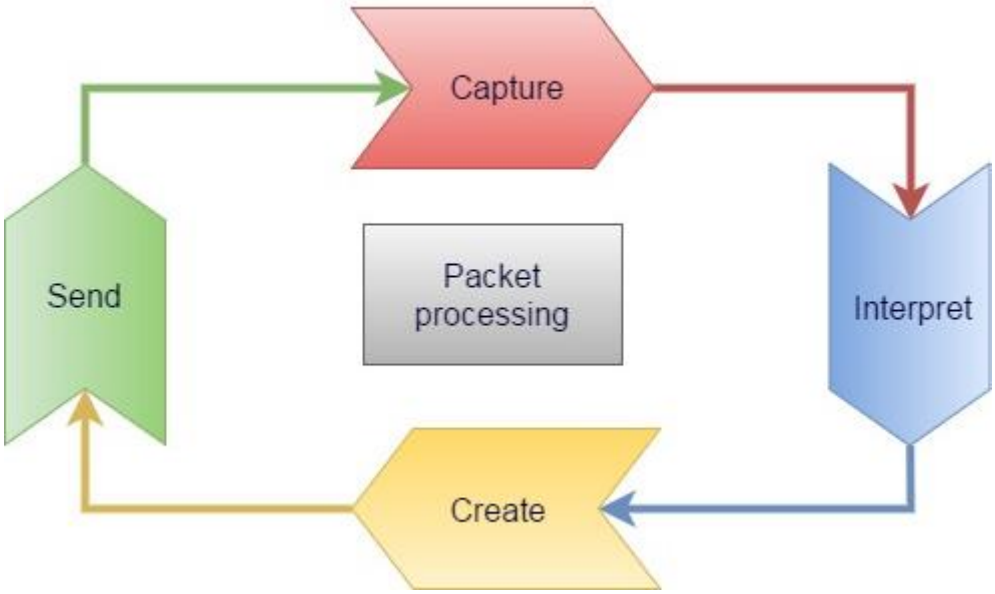

Figure 38. Packet processing cycle of HoneyIo4

In order to capture the traffic incoming to the specific interface (IP 192.168.56.101) I used Scapy library function called sniff. Sniff( ) is a blocking function that allows, when called, sniff packets from a specific interface, apply filters (to capture only specific traffic via keywords or regular expressions) and choose the number of packets sniffed at a time (it stores them into an indexed array that is the returning value for the function). As is a blocking function, it doesn't let the program code continue until it has finished. In my case I was interested in TCP, UDP and ICMP packets, coming to the specific LAN connected interface (vboxnet0, IP 192.165.56.101) and one packet at a time. So I coded:

```
a=sniff(filter="icmp or (port 8080 and tcp) or (port 8888 and tcp) or udp", count=1)
```

Figure 39. Sniff ( ) scapy blocking function coded

As we can see, it filters for icmp packets, tcp packets with destination 8080 (open port), tcp packets with destination 8888 (closed port) and udp packets. Rest of packets are discarded and not sniffed. I chose two random ports as open/closed for these tests. Counts establish that the function only blocks until one packet is captured, then it executes next code. The packet captured goes to an array (where can only be found one packet) and can be accessed with index 0, so **a[0]** references the captured packet. Once the packet is captured, it comes the interpretation and information extraction. In order to interpret the packets and identify them uniquely, I used simple Python **if** conditional statements to compare the incoming packet data with the known packet data of each specific probe (all probe packets are different), features that make them unique (usually the options field). So, once the packet was captured, a battery of tests was applied to it in order to identify what packet was. As can be seen in the example screenshot following this lines.

Figure 41. Packet identification performed via nested IF statements

I created three levels of packet identification filtering. From more general filter: protocol identification (ICMP, TCP or UDP; highlighted in red in Figure 41), then to port destination and other relevant data (closed, open port, etc.; highlighted in orange in Figure 41) to a final more detailed filtering of the packet, regarding to specific details that made them unique (such as TCP flags; highlighted in green in Figure 41). This helped to identify the packet in a systematic and efficient way, via nested IF statements. To access to the packet information, Scapy provides an easy to apply syntax. Once we have the packet a[0], each layer is accessed as an associative index; so to access TCP, there is only need to attach [TCP] to the packet expression and we will access to all the information regarding the TCP data of the packet. Once inside this layer, to access each specific field, we have to use the dot (.) and then the name of the field (it is accessed as object-oriented programming attributes). So, as an example, to access TCP sequence number field, we had to use: **a[0][TCP].seq**.

Once identified properly, comes the packet creation. By analyzing the Nmap fingerprint, we can know what Nmap expects from each specific probe in each specific Operating System. Thus, I created the appropriate response for each known arriving packet. As can be seen in Figure 42, Scapy packets are created layer by layer, allowing us to create a full customized packet to meet our specific needs (in this case, what Nmap expects as a response for each specific probe packet).

```
if a[0][TCP].dport == 8888:
    if a[0][TCP].flags==2:
        #T5 - answer - closed port
        packet = IP(dst="192.168.56.1",flags=2,ttl=0x40,id=ss)/TCP(dport=a[0][TCP].sport,
                sport=8888,flags="AR",seq=0,ack=a[0][TCP].seq+1,window=0)
        ss+=1
        send(packet)
        continue

    if a[0][TCP].flags==16:
        #T6 - answer - closed port
        packet = IP(dst="192.168.56.1",flags=2,ttl=0x40,id=ss)/TCP(dport=a[0][TCP].sport,
                sport=8888,flags="R",seq=a[0][TCP].ack,ack=0,window=0)
        ss+=1
        send(packet)
        continue

    if a[0][TCP].flags==41:
        #T7 - answer - closed port
        packet = IP(dst="192.168.56.1",flags=2,ttl=0x40,id=ss)/TCP(dport=a[0][TCP].sport,
                sport=8888,flags="AR",seq=0,ack=a[0][TCP].seq,window=0)
        ss+=1
        send(packet)
        continue
```

Figure 42. Packet creation with Scapy syntax

As can be seen in Figure 42 (above), after filtered and the probe packet has been identified (the comments specify an easy reading to know what packet is related), the answer packet is created by meeting the answer requirements. Some information is hard-written (e.g.: TCP flags field) and other is extracted from the captured packet, as can be seen in dport field definition. In order to meet some special requirements, for example IP header ID, as Nmap has specific tests for it that analyses the randomness of the value, sometimes is needed some special variable to define it and then modify its value, as shown in Figure 42: added 1 to the value after the packet is created, so the next packet, if arrives, will have next x+1 as ID and not the same as the previous one sent.
Finally, the packet has to be put into the wire and sent back to the scanning machine. For that purpose I used Scapy send( ) function that allows us to send Layer 3 packets.

```
if a[0][TCP].flags==16:
    #T6 - answer - closed port
    packet = IP(dst="192.168.56.1",flags=2,ttl=0x40,id=ss)/TCP(dport=a[0][TCP].sport,
            sport=8888,flags="R",seq=a[0][TCP].ack,ack=0,window=0)
    ss+=1
    send(packet)
    continue
```

Figure 43. Send function

Scapy has two different functions to send packets, depending on the layer. Send ( ) function will send packets at layer 3 (so it will handle routing and layer 2 for the user), while sendp ( ) will work at layer 2 and requires more specific configuration (such as the configuration of the Ethernet layer that, as can be seen, is not defined in none of the packets) because send ( ) handles with it.

Finally, as this process was performed for every single packet, once created there was no need to continue the checking process and a new packet could be sniffed. For that purpose I used Python **continue** keyword, that allows us to jump to the next iteration of the loop statement. Finally, as can be guessed, the whole process was a never ending process until Nmap finished its scan. For that reason, without the knowledge of how many iterations could be done, I used a **while True** statement at the beginning that

creates an infinite, never ending loop.

```
ss = 0x1234

while True:

    a=sniff(filter="icmp or (port 8080 and tcp) or (port 8888 and tcp) or udp", count=1)

    #tests for ICMP packet
    if ICMP in a[0]:
        packet = IP(dst="192.168.56.1", id=ss,flags=0,ttl=0x40)/ICMP(type=0,code=a[0][ICMP].code,id=a[0][ICMP].id,
                    seq=a[0][ICMP].seq)/(a[0][Raw])
        ss+=1
        send(packet)
        continue
```

Figure 44. Infinite loop as there is no known iterations

As can be seen in Figure 44, all the code is within the never ending loop and the usage of the continue keywords is for performance purposes just because if the packet has been detected and sent, there is no need to check for others and the program can skip all the others IF statements and loop again, that is sniffing a new packet and starting again the process. Note that all variables referenced before are defined before the loop, so they start with a fixed value that is modified within the loop.

As a result, the mix and deep development of all of these features shapes the whole program of each specific OS simulation. This process was performed for all of the 4 IoT OS simulations performed, attending and respecting the expected responses to the probe packets by Nmap (analyzing its fingerprints). Some of the test deserved special attendance as they are unique algorithms performed by Nmap that have to be discovered in order to fake the OS properly (see for example TI, CI and II tests [28], while some of them are gathered directly from the packet as fingerprint values (like window or flags). Figure 45 shows an extract (first lines) of the complete code of one of the OS simulations. More specifically the one that fakes GoPro Hero3 adventure camera OS. There we can see all the features shown in previous pages in a more extended way with ~100 lines for this specific OS TCP/IP stack simulation.

```
from scapy.all import *
import iptc

chain = iptc.Chain(iptc.Table(iptc.Table.FILTER), "INPUT")
rule = iptc.Rule()
rule.in_interface = "eth1"
rule.src = "192.168.56.0/255.255.255.0"
target=iptc.Target(rule,"DROP")
rule.target=target
chain.insert_rule(rule)
#chain.delete_rule(rule)

ii = 0x1234
ci = 0xABCD

while True:

    a=sniff(filter="icmp or (port 8080 and tcp) or (port 8888 and tcp) or udp", count=1)

    if ICMP in a[0]:
        packet = IP(dst="192.168.56.1",id=ii,flags=0)/ICMP(type=0,code=0,id=a[0][ICMP].id,seq=a[0][ICMP].seq)/(a[0][Raw])
        ii+=1
        send(packet)
        continue

    if TCP in a[0]:
        if a[0][TCP].dport == 8888 and a[0][TCP].options == [("MSS",1460)]:
            packet = IP(dst="192.168.56.1",id=0x0000)/TCP(dport=a[0][TCP].sport,sport=8888,flags="RA",seq=1,ack=1,window=0)
            send(packet)
            continue

        if a[0][TCP].dport == 8888:
            if a[0][TCP].flags==2:
                #T5 - answer - closed port
                packet = IP(dst="192.168.56.1",flags=0,id=ci)/TCP(dport=a[0][TCP].sport,sport=8888,flags="AR",seq=0,
                            ack=a[0][TCP].seq+1,window=0)
                ci+=1
                send(packet)
                continue
```

Figure 45. Extract of GoPro OS Simulation code

In order to test HoneyIo4 properly, I created a typical scanning situation that can be seen in Figure 46. In this case, both computers are in the same private subnet, but the same example could be extended with combination of public IP's (when Network Address Translation (NAT) is in place).

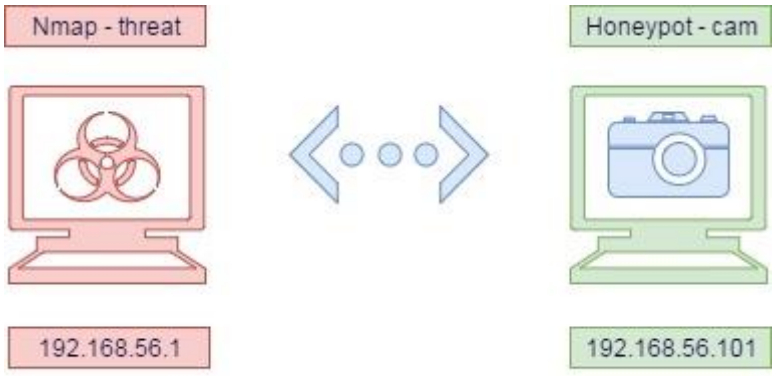

Figure 46. Testing scenario

In this case-scenario, attacker interacts with the victim by sending packets via Nmap. The attacker issues Nmap remote OS detection to the specific address that he knows is an alive host in the same LAN. Honeypot machine has GoPro Hero3 OS simulation in place and awaiting for incoming connections. Figure 47 shows the execution of the two activation commands on each of the machines (left is nmap/scanner and right is honeypot). As we can see, as long as Nmap starts to send packets, the Honeypots answers back (Sent 1 packets message indicates that Send ( ) function in Scapy has been called successfully).

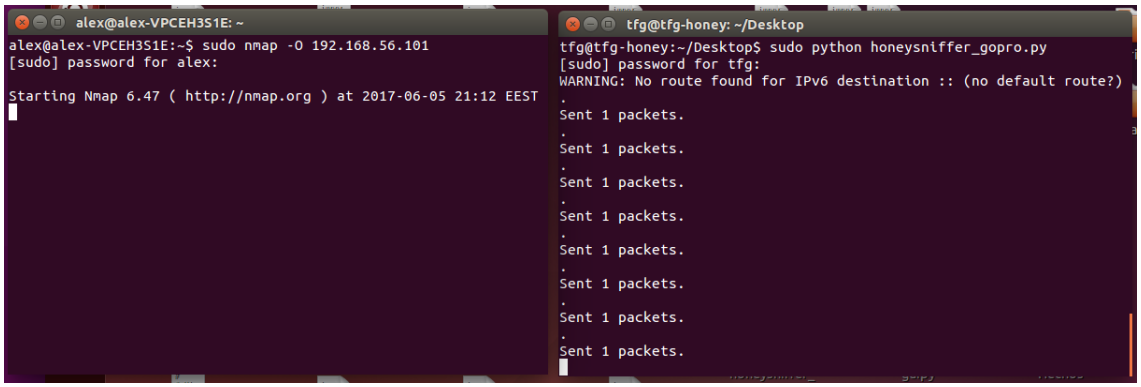

Figure 47. Testing start – left is scanning machine, right is HoneyIo4

After a while (in this case 35.95 seconds), the scanning process ends and Nmap shows the output. In this case, the machine running Ubuntu has been detected as a GoPro Hero3 camera (categorized by Nmap as webcam). It has detected also one open port and one closed port. Attacker trust this Nmap output and is deceived by HoneyIo4.

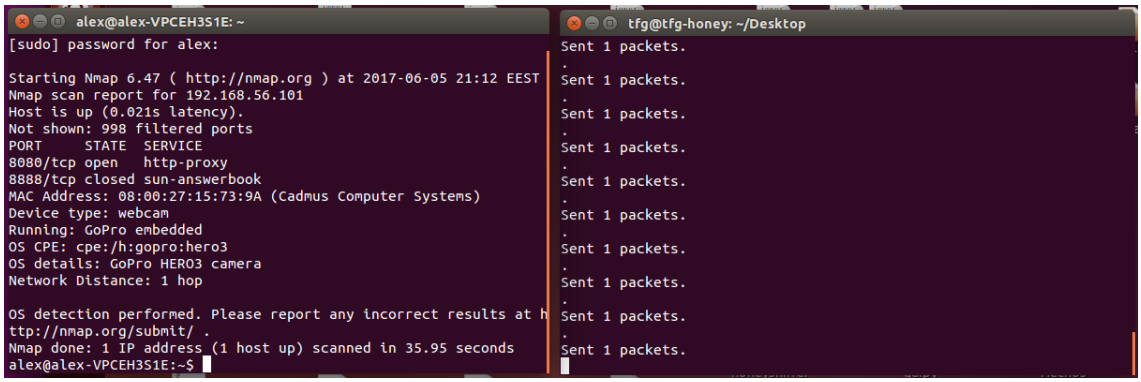

Figure 48. Testing ends – Nmap detects HoneyIo4 machine as GoPro Hero3 camera

At the same time, I captured with Wireshark all traffic flowing through vboxnet0 (interface connected to VLAN in attacker machine, IP 192.168.56.1). Figure 49 shows the traffic involving port 8080 (open port) on Honeypot. As we can see, an according to the fingerprint of GoPro Hero3, there is no response to most of TCP probe packets, so as there is no response in most of them (Figure 50), Nmap resends them awaiting response. After a couple retransmissions without response, it gives up. Figure 51 shows a different case, those are the TCP packets which destination is the closed port. In this capture, port scanning has been omitted and it is not shown. This screenshot shows a completely different behavior as the fingerprint states. There is answer to all closed port TCP packets from OS running GoPro Hero3 (Figure 52).

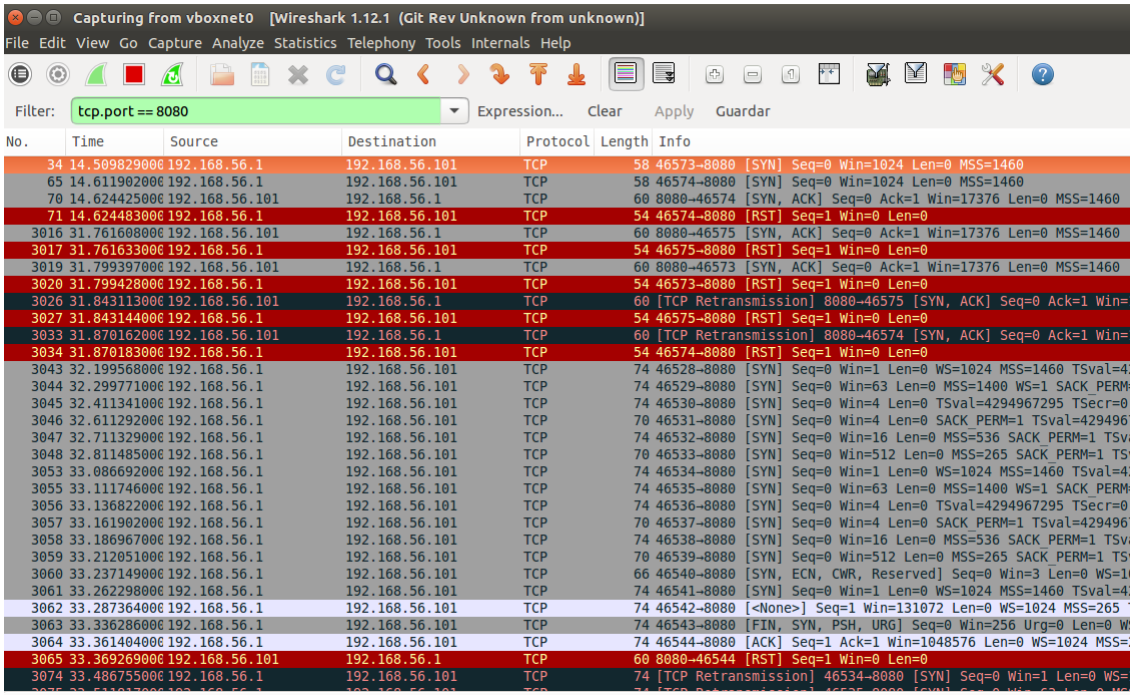

Figure 49. Traffic to port 8080, TCP probes to open port

```
# GoPro3 Camera
Fingerprint GoPro HERO3 camera
Class GoPro | embedded || webcam
CPE cpe:/h:gopro:hero3
SEQ(CI=I%II=I)
OPS(R=N)
WIN(R=N)
ECN(R=N)
T1(R=N)
T2(R=N)
T3(R=N)
T4(R=Y%DF=N%T=3B-45%TG=40%W=0%S=A%A=Z%F=R%O=%RD=0%Q=)
```

Figure 50. TCP probes to open port – reference fingerprint

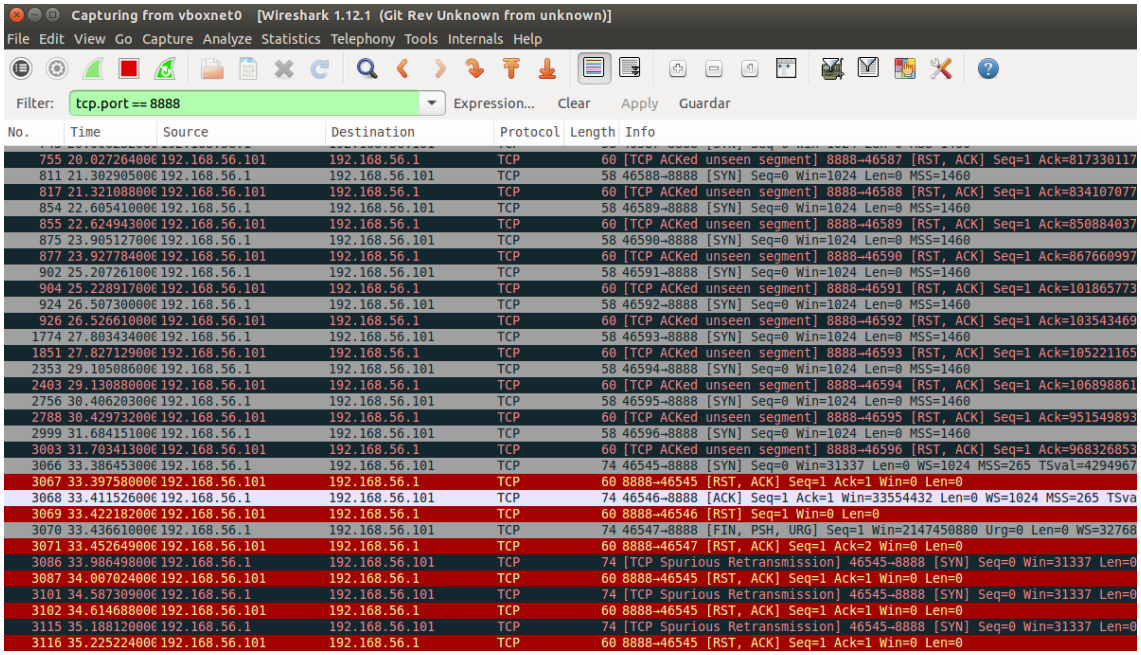

Figure 51. Traffic to port 8888, TCP probes to closed port

```
T5(R=Y%DF=N%T=3B-45%TG=40%W=0%S=Z%A=S+%F=AR%O=%RD=0%Q=)
T6(R=Y%DF=N%T=3B-45%TG=40%W=0%S=A%A=Z%F=R%O=%RD=0%Q=)
T7(R=Y%DF=N%T=3B-45%TG=40%W=0%S=Z%A=S+%F=AR%O=%RD=0%Q=)
```

Figure 52. TCP probes to closed port – reference fingerprint

Last two lines of the fingerprint references ICMP and UDP packets. Figure 53 and 54 show the Wireshark traffic capture for both cases, all congruent with Figure 55, which is fingerprint reference for those two test lines.

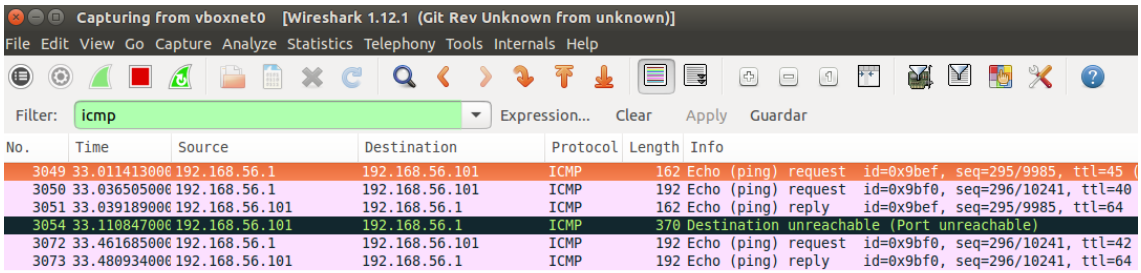

Figure 53. ICMP packet probes

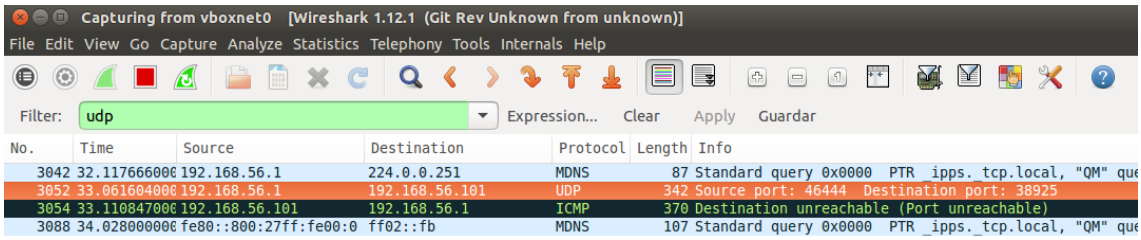

Figure 54. UDP packet probes

```
U1(DF=N%T=3B-45%TG=40%IPL=164%UN=0%RIPL=G%RID=G%RIPCK=G%RUCK=G%RUD=G)
IE(DFI=N%T=3B-45%TG=40%CD=Z)
```

Figure 55. UDP and ICMP test lines – reference fingerprint

As a result, HoneyIo4 deceived the scanning process successfully by simulating an IoT Operating System. This process was repeated for the other 3 OS that HoneyIo4 can simulate (Oki B4545 – printer, Nintendo Wii – video game console and Casio QT6600 – cash registering machine).

Next screenshots show the Nmap output in the remaining three IoT OS that HoneyIo4 can fake. Figure 55 shows Oki B4545 output when analyzed with Nmap, Figure 56 the output of Nintendo Wii scanning and Figure 57 the output of Casio QT6600 registering machine.

```
alex@alex-VPCEH3S1E: ~
alex@alex-VPCEH3S1E:~$ sudo nmap -O 192.168.56.101

Starting Nmap 6.47 ( http://nmap.org ) at 2017-06-07 19:22 EEST
Nmap scan report for 192.168.56.101
Host is up (0.011s latency).
Not shown: 998 filtered ports
PORT     STATE  SERVICE
8080/tcp open   http-proxy
8888/tcp closed sun-answerbook
MAC Address: 08:00:27:15:73:9A (Cadmus Computer Systems)
Device type: printer
Running: Oki embedded
OS CPE: cpe:/h:oki:b4545
OS details: Oki B4545 printer
Network Distance: 1 hop

OS detection performed. Please report any incorrect results at http://nmap.org/s
ubmit/ .
Nmap done: 1 IP address (1 host up) scanned in 31.57 seconds
```

Figure 56. Printer's Nmap scan output

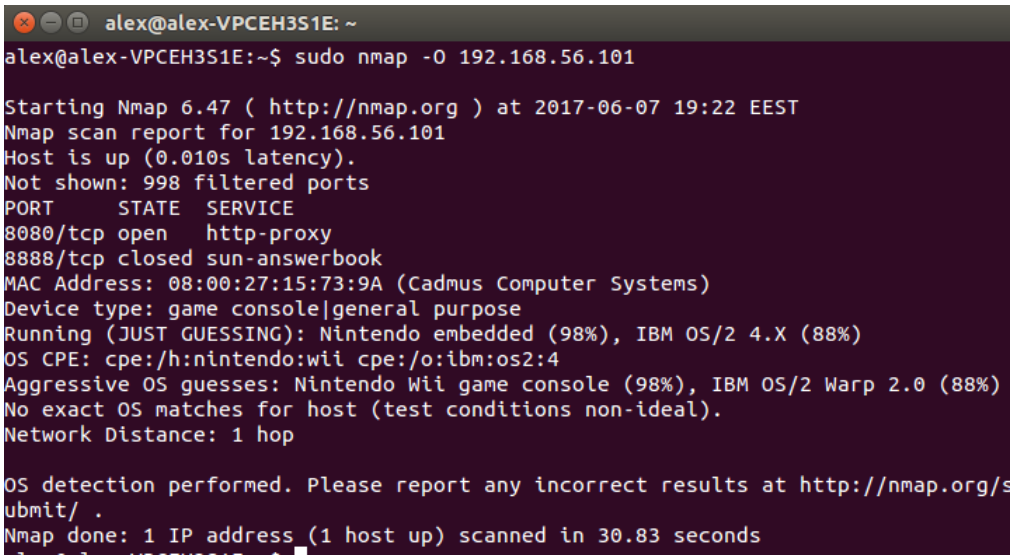

Figure 57. Wii's Nmap scan output

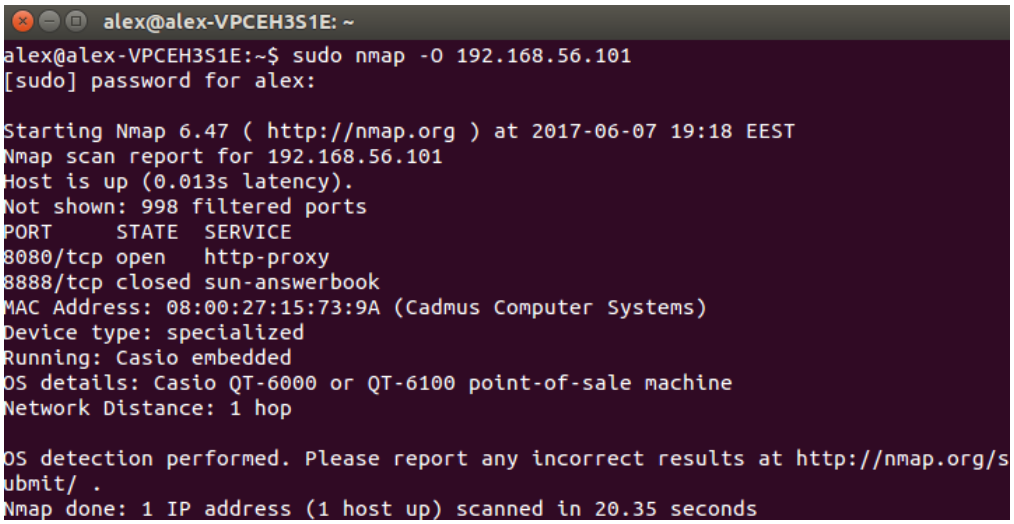

Figure 57. Wii's Nmap scan output

Graphical User Interface for HoneyIo4

HoneyIo4 can be easily executed in Ubuntu Command Line Interface (CLI) by issuing python and the chosen parameter (OS to simulate). As we have seen, it runs iptables and all the simulation process without further assistance. Nevertheless, some users that are not used to CLI may need further assistance or may have troubles running it. For that purpose, HoneyIo4 has a Graphical User Interface that runs in the web browser. It looks like a webpage where the user can easily select which OS simulate, activate and deactivate the honeypot at will.

HoneyIo4 Graphical User Interface runs in the web browser (by issuing localhost in the navigation bar). In order to run it properly, it is needed to create a complete LAMP environment (Linux-Apache-MySQL-PHP). Once installed, it is really easy to run HoneyIo4 from its GUI.

HoneyIo4 GUI was coded using HTML as a markup language, CSS as a HTML styling language, JavaScript programming language to run specific client side functionalities and PHP to run the HoneyIo4 python script. I created a simple HTML, CSS interface that can be seen in Figure 56 that can be accessed by typing localhost/index.html. This works easily as there is a configured LAMP environment that creates a local web service environment accessible as a public website from the navbar.

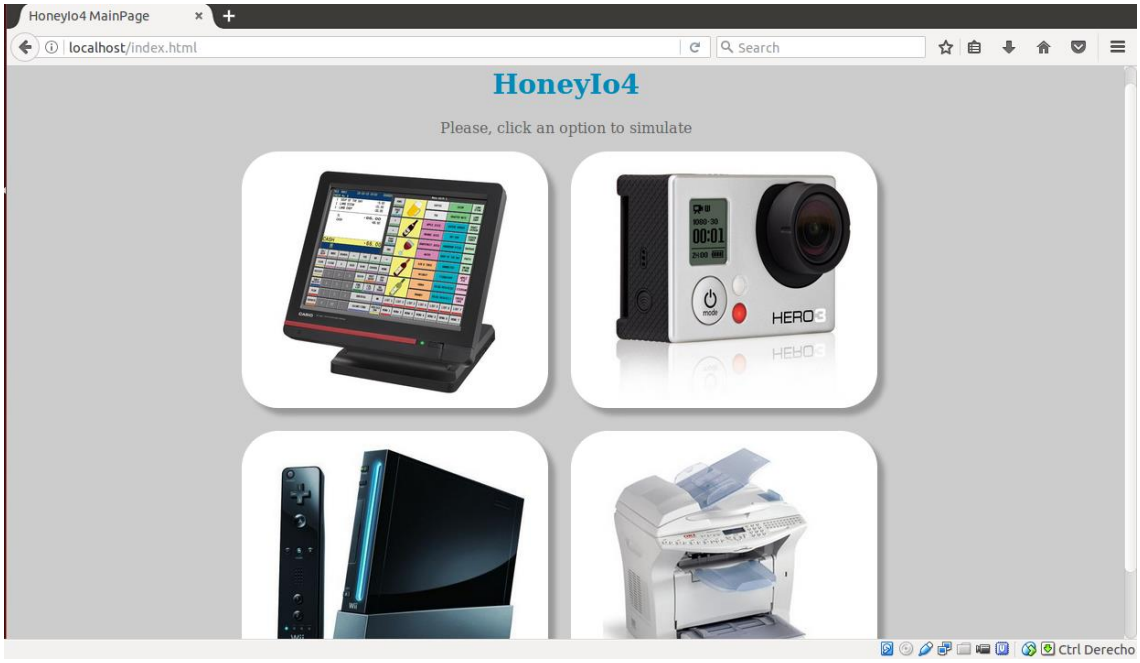

Figure 56. GUI of HoneyIo4 – accessible at localhost/index.html

This interface has an easy navigation, as almost everything is visual/graphical, allowing non experienced users to run HoneyIo4 easily. To activate one of the OS simulations, the user has to press over the chosen OS of the 4 provided by HoneyIo4 (more information is provided when the user is positioned over the image, as can be seen in Figure 57). When the user presses the selected OS, a new layer appears over the interface with two main options: Run Simulation or Stop Simulation, as can be seen in Figure 58. Everything done through steps is coded with HTML and CSS hardcoded. The user has then two options: Run or Stop. The next logical step is Run the simulation, when pressing the Run button, an html link calls a PHP coded webpage that executes the Python code regarding to that OS simulation. Figure 59 shows as an example the link that executes GoPro Hero3 OS (gopro.php) and Figure 60 the HTML link call.

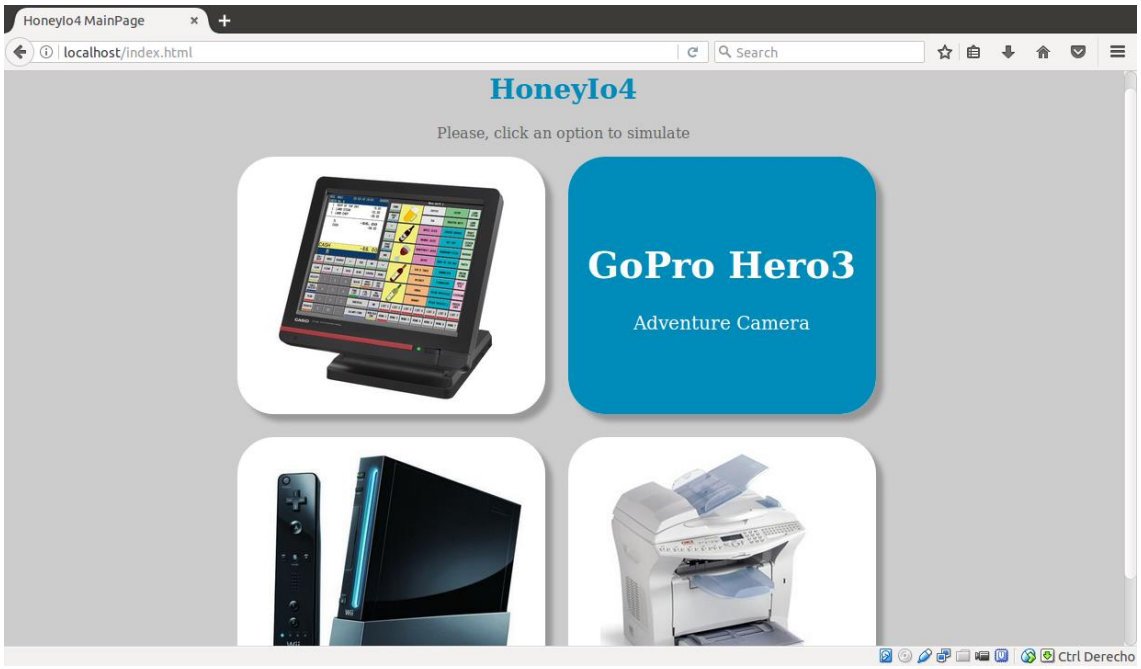

Figure 57. Mouse is over the image of GoPro Hero3 – information is displayed

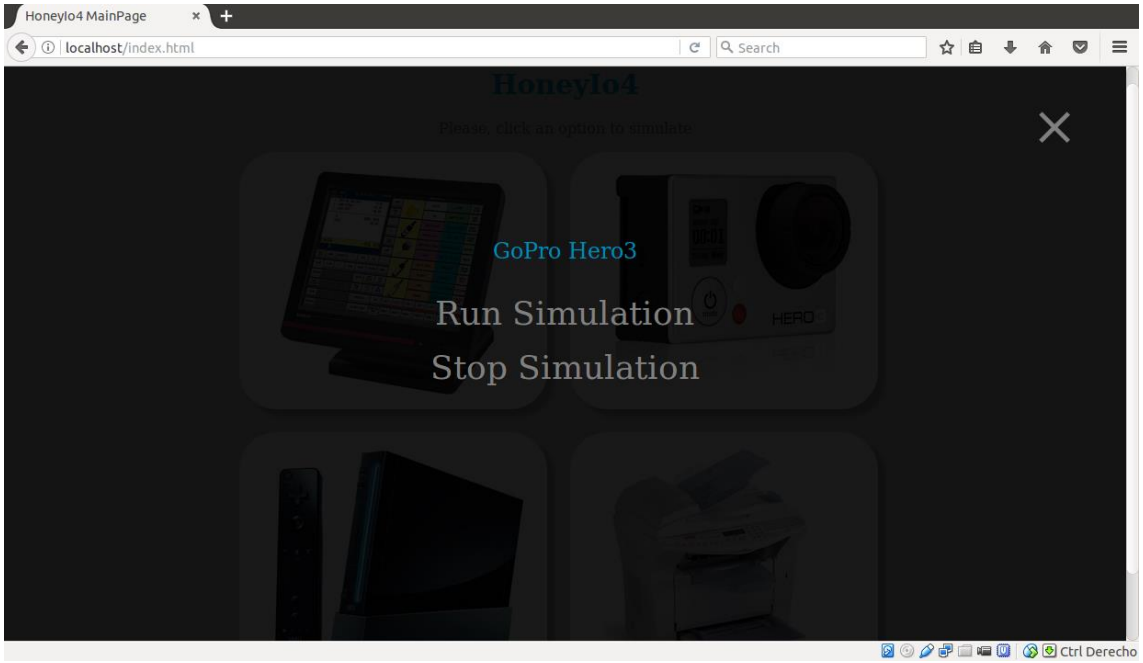

Figure 58. When clicked, a new over layer appears and shows the options available

```
gopro.php
<?php
exec('sudo python honeysniffer_gopro.py');
?>
```

Figure 59. gopro.php content

```
<!-- Overlay content -->
<div class="overlay2-content">
  <p class="header"> GoPro Hero3 </p>
  <a href="gopro.php" onclick="runOS('gopro')" class="act">Run Simulation</a>
  <a href="stop.php" onclick="stopOS('gopro')" class="desact">Stop Simulation</a>
</div>
```

Figure 60. Code in index.html that executes Python script (call to gopro.php)

As can be seen in Figure 59, the code is executed by using exec PHP built-in function that allows the execution of an external program inside PHP code. As the code is running socket management, it requires superuser privileges (sudo is needed), so some modification was done in order to allow Apache to execute this. As Apache normal user is www-data and has user privileges, it cannot run directly sudo commands with exec function. In order to allow www-data to run sudo commands, it has to be added to sudoers file (/etc/sudoers). Using the program visudo (only this one) this file can be modified securely. The most secure way is to enable www-data to have root permissions in some specific path and not in all the system, so I added www-data in sudoers and added the needed paths to execute the command, so, as a result, Apache can run successfully the python code. When the user presses Run simulation, as can be seen in Figure 60, it goes to the link gopro.php, that executes the py script and also performs onclick event (Javascript), calling runOS function. This function, as shown in Figure 61, handle with layers and some styling appearances, as can be show in Figure 62, it closes the over layer, and shows a green over layer in the image of the selected OS running.

```
function runOS(value) {

    if (value=="cash"){
        document.getElementById("active1").style.opacity = 100;
        document.getElementById("opt1").style.width = "0%";
  } else if (value=="gopro") {
     document.getElementById("active2").style.opacity = 100;
     document.getElementById("opt2").style.width = "0%";
  } else if (value=="wii") {
    document.getElementById("active3").style.opacity = 100;
    document.getElementById("opt3").style.width = "0%";
  } else {
    document.getElementById("active4").style.opacity = 100;
    document.getElementById("opt4").style.width = "0%";
  }
}

function stopOS(value) {

    if (value=="cash"){
        document.getElementById("active1").style.opacity = 0;
  } else if (value=="gopro") {
     document.getElementById("active2").style.opacity = 0;
  } else if (value=="wii") {
    document.getElementById("active3").style.opacity = 0;
  } else {
    document.getElementById("active4").style.opacity = 0;
  }
}
```

Figure 61. Javascript functions that handle with layers

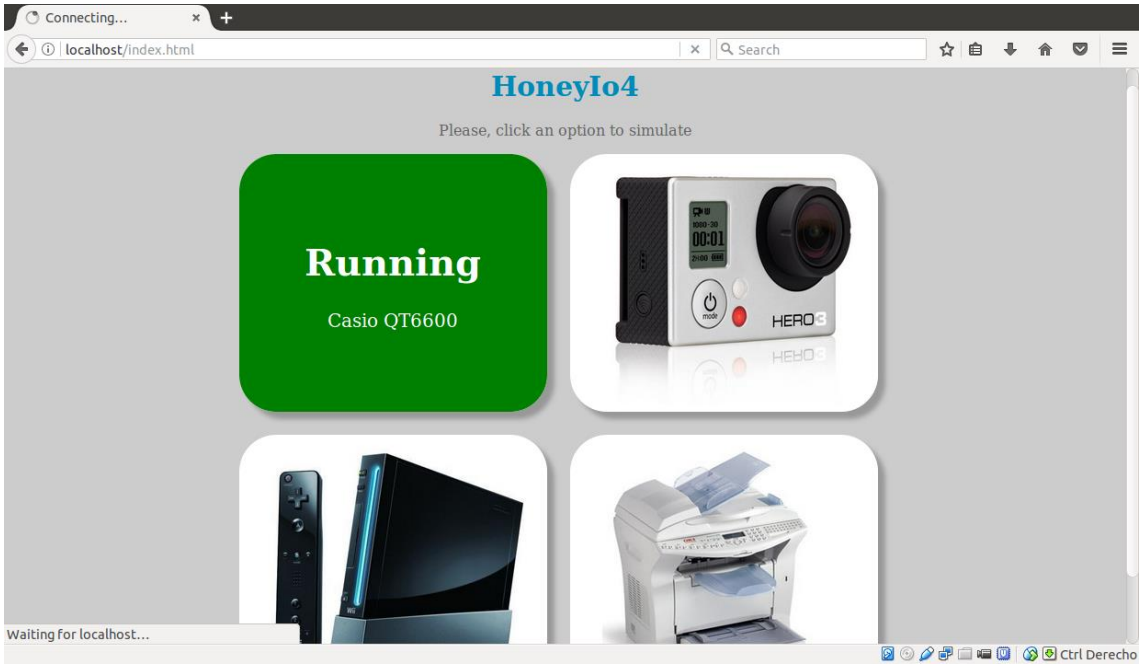

Figure 62. Script activated (over layer indicator in green)

This functionality serves as an indicator of which OS is running if any (displays the information for the user). When the user wants to Stop the simulation process, he/she has to press again the running image and the over layer appears again and by clicking Stop Simulation, everything comes to the initial state (Figure 63). Please note that, as all the simulations are running in the same "simulated" ports, only one simulation can be run at a time. So in order to active another one, the running one has to be stopped.

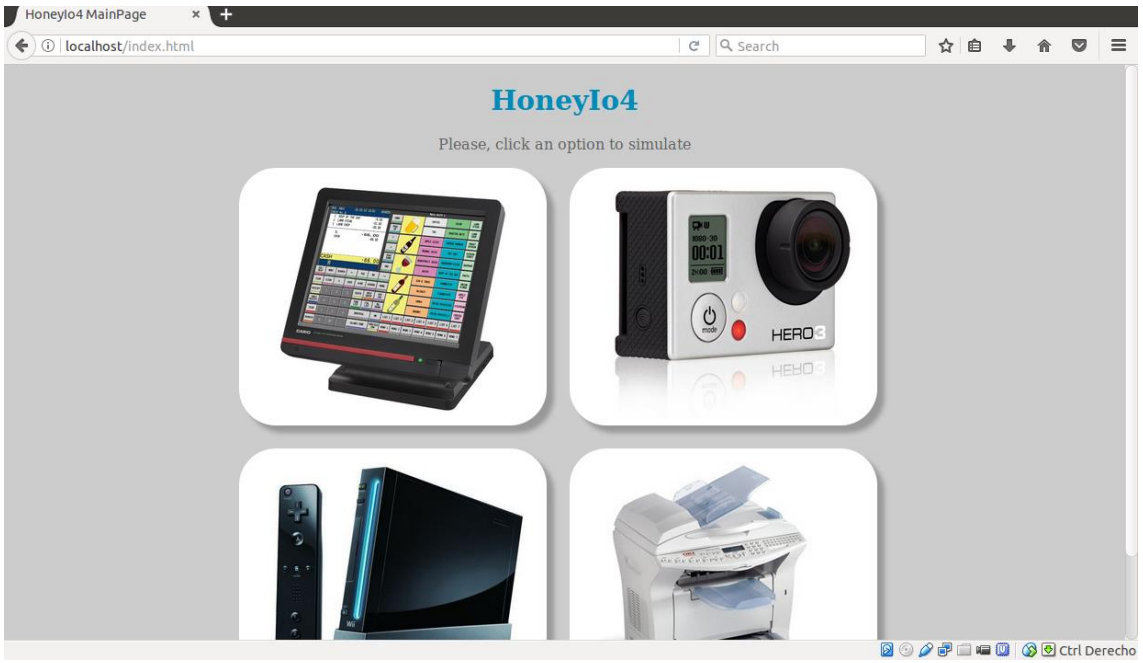

Figure 63. Simulation stopped, initial state is recovered

Now, if the user wants to run again the same OS simulation or another one, it has to simply do the same steps and the simulation will Run allowing him to stop at will. In

order to stop the Simulation, a new php webpage is called, as can be seen in previous Figure 60, called stop.php. This page executes the following code:

```php
<?php
exec('sudo touch /var/www/html/exit.doc');
exec('sudo python stopper.py');
exec('sudo rm /var/www/html/exit.doc');
header('Location: index.html');
?>
```

Figure 64. Code executed by stop.php

The code executed by stop.php are three calls to exec( ) function and one PHP redirection to the mainpage (using header( ) function). The first exec( ) creates a new file in Apache html folder (which is the one accessible and executed when navigate to localhost). This file main purpose is to stop the py code that is executing the simulation. As the OS simulation code is an infinite loop, there is no way to finish it without using Ctrl+C in CLI. Another option is to use another strategy to avoid that dramatic end of script. I added a new line to the beginning of the loop (Figure 65), that tests each time for the existence of a file in the same folder, called exit.doc. It this folder exists, the loop is finished by a break statement. If not, the loop continues.

```
ii = 0x1234
ci = 0xABCD
while True:

    if os.path.isfile('./exit.doc'):
    break

    a=sniff(filter="icmp or (port 8080 and tcp) or (port 8888 and tcp) or udp", count=1)
```

Figure 65. New added content that checks for file existence

As we don't know when to finish (whether Nmap has finished the scanning or not), if we press the Stop button when Nmap has finished, there are no packets incoming so the code is stuck in sniff ( ) function. To avoid this, and recheck again for new created file, I created a new python code that sends a new packet to unblock the sniffer (Figure 66) and perform a one more loop. It also drops the rule in IPTables in order to clear everything for a new use.

```
from scapy.all import *
import iptc

packet = IP(dst="192.168.56.101",id=0x10,flags=0)/UDP(sport=1000, dport=8080)
send(packet)

chain = iptc.Chain(iptc.Table(iptc.Table.FILTER), "INPUT")
rule = iptc.Rule()
rule.in_interface = "eth1"
rule.src = "192.168.56.0/255.255.255.0"
target=iptc.Target(rule,"DROP")
rule.target=target
chain.delete_rule(rule)
```

Figure 66. Stopper.py sends a new packet to unblock and drops rule in IpTables

So when the file is created, the extra packet is sent, a new loop is performed, the file is detected and the script is end successfully and correctly. Also, the rule disappears from IPTables, so everything turns to normal state. Finally, in order to clear everything to next simulation, the next exec removes the exit.doc file and next Header redirection redirects to the mainpage of HoneyIo4, everything is ready for a new simulation.

In order to test these I followed the steps that a user had to perform in order to run one simulation successfully. First I performed an Nmap scan without running any of the simulations (Figure 67). As can be confirmed, the VM is detected as a Linux environrment that is running a web-browser in port 80, everything normal in a web service running in a Linux based VM environment.

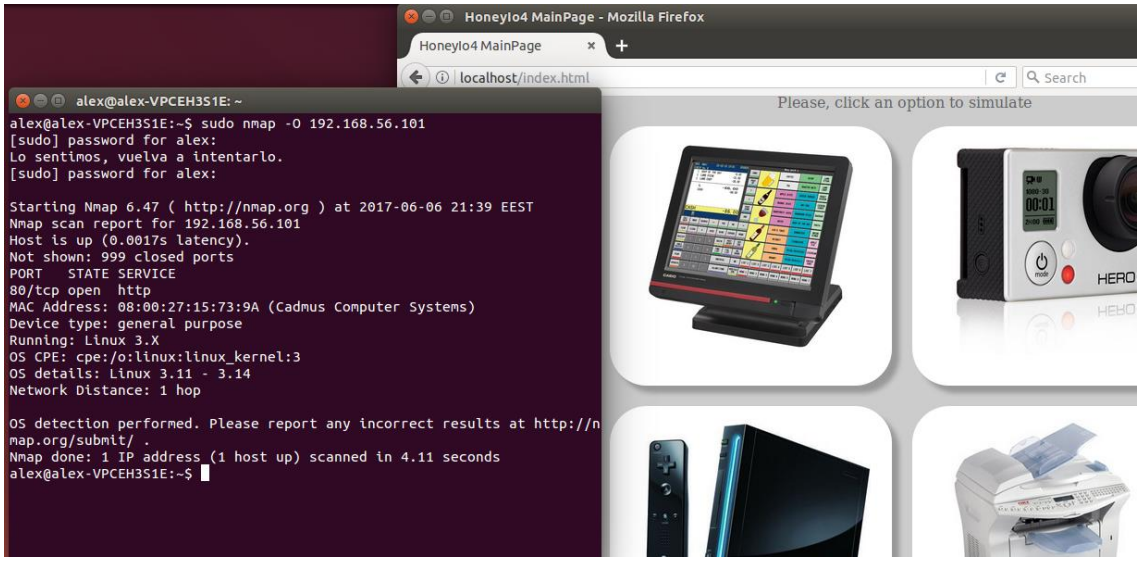

Figure 67. Stopper.py sends a new packet to unblock and drops rule in IpTables

Then I clicked and run one of the OS simulations, as can be seen in Figure 68. Nmap detected the host as IoT device running and IoT OS (cash registering machine).

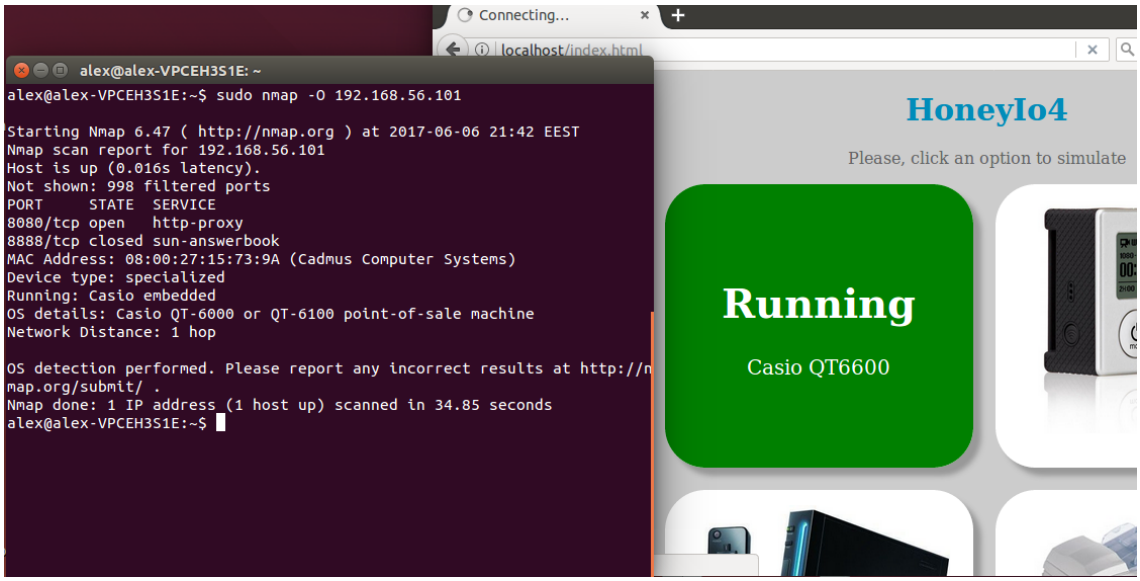

Figure 68. Nmap is tricked and IoT device running is detected

As a result, HoneyIo4 tricked Nmap and it detected an IoT OS running. HoneyIo4 working and tricking the network scanner. Then I wanted to stop the simulation (Figure 69) and clicked Stop Simulation, previous explained stop.php has to be executed and everything restored if working.

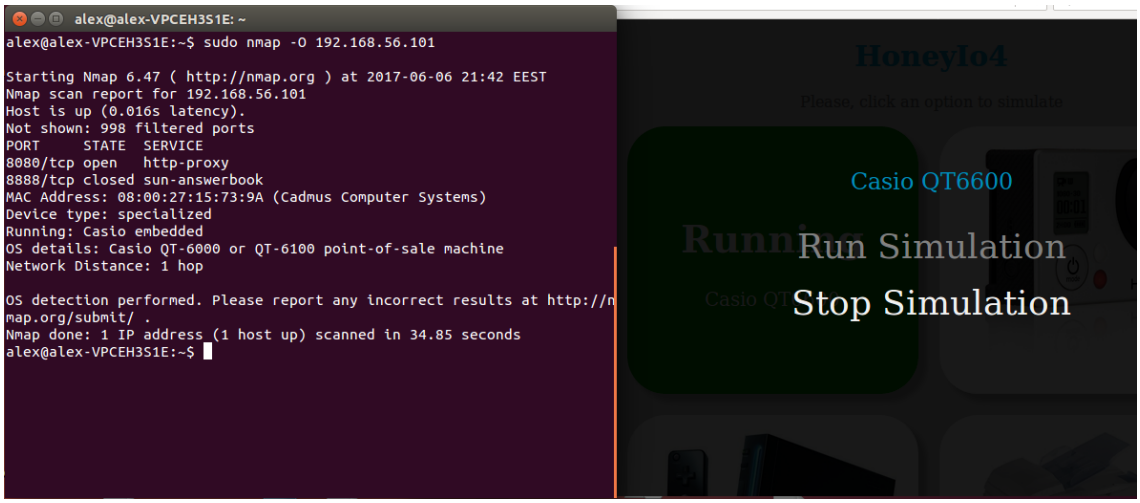

Figure 69. Before clicking Stop Simulation link.

I performed a new Nmap OS detection to check if everything was restored to normal state (no OS IoT simulated). Figure 70 confirms that all has been successfully return to previous state. No simulated IoT, just a normal Linux based VM ready for the next simulation.

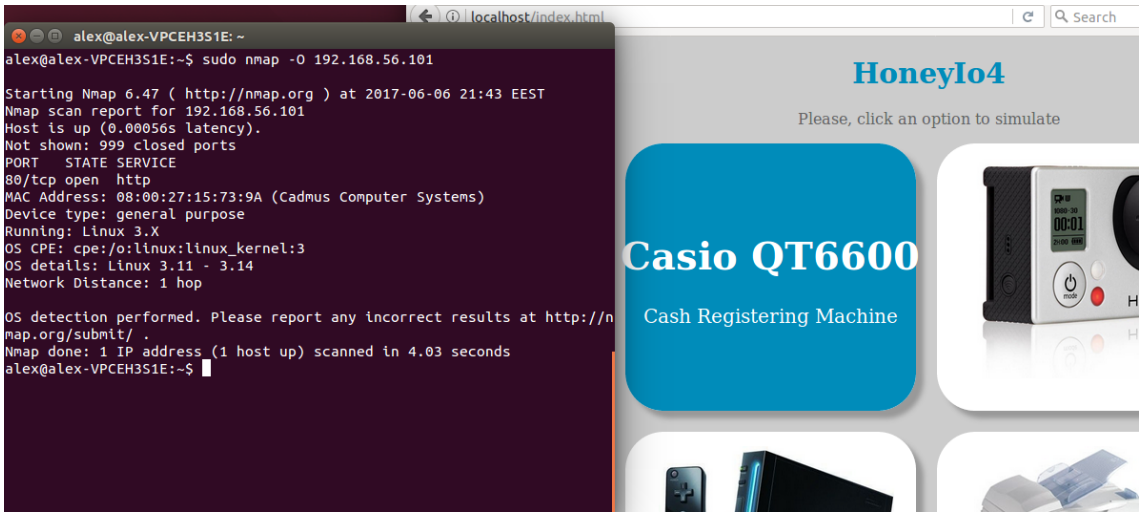

Figure 70. Before clicking Stop Simulation link.

I chose to drop the rule, so the VM was detected as it is. But if it is preferred, the IPTables rule could have been kept and the system had been detected as filtered, no ports open and unknown OS running (only that the system was alive could have been detected by Nmap). So the proof-of-concept of HoneyIo4 seemed to work successfully and trick Nmap by detecting an IoT OS running instead of the Ubuntu OS running. HoneyIo4 works both in CLI and GUI mode.

Present and Future work

HoneyIo4 simulates 4 IoT OS. It is a low-interaction honeypot with minimal capabilities that are reduced to OS simulation/detection by network scanners. Further work is needed to create a complete low-interaction Honeypot as HoneyD. HoneyD allows the user to "create more realistic networks, Honeyd supports the creation of virtual network topologies. The networks can be configured to contain routers with configurable link characteristics like latency and packet loss. When using tools like traceroute, the network traffic appears to follow the configured topology" [11]. While HoneyD allows the user to perform more than one simulation at a time and create a network virtual topology running of running PC based OS hosts, HoneyIo4 allows only running one simulation at a time but for IoT OS that HoneyD is not able to fake.

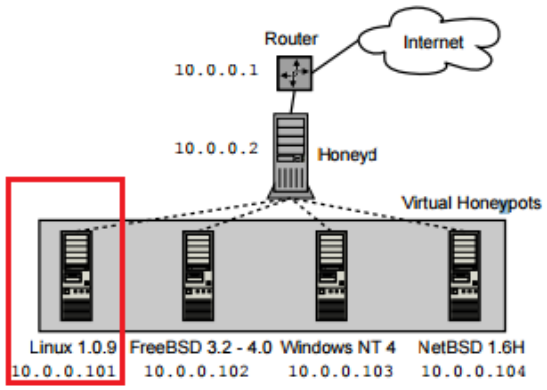

Figure 71. HoneyD and compared HoneyIo4 (highlighted in red).

In order to enhance HoneyIo4 to HoneyD level additional capabilities have to be added to HoneyIo4, although the most important one is covered (OS deception). HoneyIo4 should implement an external and differentiated engine, as HoneyD does (in Figure 71 the machine with IP 10.0.0.2), that allows to create routing and more instances of IoT running devices. Main OS personality engine is performed by HoneyIo4 but further capabilities require more implementation as HoneyD does like: basic services TCP handshakes and UDP banners, so more than one instance can be analyzed by the attacker in the network and the main engine have to track the connection and create a simple TCP state machine where the connection is created with the service and then dropped (HoneyD has not implemented fully receiver and congestion window management, so there is only three-handshake establishment). HoneyD has also redirection policies, that can be applied in HoneyIo4 by using IPTables rules, as I did before with the drop policy applied to incoming packets. Finally, *Honeyd keeps state for each honeypot. This includes information about ISN generation, the boot time of the honeypot and the current IP identification number. Before a packet is sent to the network, it passes through the personality engine* [11]. So, in order to create multiple instances of HoneyIo4, there is also the same need of a main core that keep track of all the information running and so it should be able to act appropriately to the incoming events for every instance. HoneyIo4 is a small approach to HoneyD, more technical requirements should be met in order to enhance HoneyIo4 to HoneyD but the main issue, OS simulation is already performed successfully by HoneyIo4 with the advantage of being capable of act as a IoT device, luring, tricking and deceiving IoT hunters.

Conclusion

HoneyIo4 is an IoT low-interaction honeypot that pretends to fill the gap in IoT active active defense mechanisms, lacking nowadays and with a great and exponential impact in the future. IoT devices are growing exponentially and are in the spotlight of the hackers to perform its misdeeds. HoneyIo4 can help to learn about attackers mechanisms. It allows the user to simulate 4 different and common IoT Operating System devices, lure and deceive the attackers' scanning process. Nmap outputs do not allow the attacker to hesitate if the machine is real or not, it is simply something that it is. Nmap detects the HoneyPot as an IoT device and the attacker it is informed in detail about it. HoneyIo4 can be run both from CLI or GUI, both for experienced and non-experienced users. Although its original capabilities are limited, it performs, as could have been shown across this report, successfully IoT OS detection deception. More functionalities can be added to this basic core to enhance HoneyIo4 capabilities: multiple instances, routing, TCP connection, banner advertisement, etc. are only a few of the improvements that can be added to the HoneyIo4 fingerprinting engine. HoneyIo4 like Honeyd work with fingerprints (TCP/IP stack responses) in order to trick network scanners, but in the field of IoT devices, a lacking point in the Honeyd implementation. Further implementation and work has to be done in HoneyIo4 to enhance its capabilities and being able to create a full topology of IoT devices, which will be common in the near future. HoneyIo4 is a good starting point in the defense of the massively growing number of common devices that are acquiring networking and communication capabilities, defenseless against the massive exploitation of its new capabilities by black hat hackers. Cars, washing machines, fridges, etc. are a new field in exploitation and can be used for multiple purposes by skilled hackers. In a global and interconnected world, HoneyIo4 helps to learn about those bad guys that emerge from the shadows to exploit, most of times without our awareness, our appreciated devices.

References

[1] G. Corera. Intercept, “The secret history of Computers and Spies”, 2015.

[2] R. van der Meulen, “Gartner Says 8.4 Billion Connected "Things" Will Be in Use in 2017, Up 31 Percent From 2016”, Press Release. [Online] Available: http://www.gartner.com/newsroom/id/3598917 [Accessed: 7-May-2017].

[3] R.M. Lee, “The Sliding Scale of Cyber Security”. [Online]. Available: https://www.sans.org/reading-room/whitepapers/analyst/sliding-scale-cyber-security-36240 [Accessed: 7-May-2017].

[4] CCDCOE, “Terms and Definition”. [Online]. Available: https://ccdcoe.org/cyber-definitions.html [Accessed: 8-May-2017].

[5] B. Frumess, “A definitive guide to active cyber defense: Modularizing cybersecurity”. [Online]. Available: https://www.researchgat.net/profile/Barnaby_Frumess/publication/290997565_A_DEFINITIVE_GUIDE_TO_ACTIVE_CYBER_DEFENSE_MODULARIZING_CYBERSECURITY/links/569d66bb08aed27a702faf32/A-DEFINITIVE-GUIDE-TO-ACTIVE-CYBER-DEFENSE-MODULARIZING-CYBERSECURITY.pdf [Accessed: 1-May-2017].

[6] L. Spitzner, “Honeypots: Tracking hackers”, 2002.

[7] M. Walker, “CEH Certified Ethical Hacker”, 2012.

[8] The Honeynet Project, “Know your enemy: honeynets”. [Online]. Available: http://old.honeynet.org/papers/honeynet/. [Accessed: 1-May-2017].

[9] C. Seifert, I. Welch, P. Komisarczuk, “Taxonomy of Honeypots”. [Online]. Available: http://www.mcs.vuw.ac.nz/comp/Publications/archive/CS-TR-06/CS-TR-06-12.pdf [Accessed: 27-Apr-2017].

[10] F. Pouget, M. Dacier, H. Debar, “White Paper: Honeypot, Honeynet, Honeytoken: Terminological issues”. [Online]. Available: http://www.eurecom.fr/en/publication/1275/download/ce-pougfa-030914b.pdf [Accessed: 2-May-2017].

[11] N. Provos, “Honeyd: A Virtual Honeypot Daemon”. [Online]. Available: http://metro.citi.umich.edu/u/provos/papers/honeyd-eabstract.pdf. [Accessed: 20-Apr-2017].

[12] Y.M. Pa Pa, S. Suzuki, K. Yoshioka, T. Matsumoto, T. Kasama, C. Rossow, “IoTPOT: Analysing the Rise of IoT Compromises”. [Online]. Available: https://www.usenix.org/system/files/conference/woot15/woot15-paper-pa.pdf [Accessed: 20-May-2017].

[13] T. Hudak, “Botnets”. [Online]. Available: https://www.korelogic.com/Resources/Presentations/botnets_issa.pdf. [Accessed: 1-Jun-2017]

[14] R. Graham, “Mirai and IoT Botnet Analysis”. [Online]. Available: https://www.rsaconference.com/writable/presentations/file_upload/hta-w10-mirai-and-iot-botnet-analysis.pdf [Accessed:1-Jun-2017].

[15] C. Seifert, I. Welch, P. Komisarczuk, “HoneyC - The Low-Interaction Client Honeypot”. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.61.6882&rep=rep1&type=pdf [Accessed: 20-May-2017].

[16] S. J. Vaughan-Nichols, “DDoS attacks increase over 125 percent year over year”. [Online]. Available: http://www.zdnet.com/article/ddos-attacks-increase-over-125-percent-year-over-year/ [Accessed: 1-Jun-2017].

[17] I. Arghire, “IoT Botnets Fuel DDoS Attacks Growth: Report”. [Online]. Available: http://www.securityweek.com/iot-botnets-fuel-ddos-attacks-growth-report. [Accessed: 1-Jun-2017].

[18] Arbor Networks, “IoT BOTNETS: Massive Attacks with No Turbocharging”. [Online]. Available: https://pages.arbornetworks.com/rs/082-KNA-087/images/WISR_Infographic_NoEndInSight_FINAL.pdf [Accessed: 1-Jun-2017].

[19] Nmap, “Chapter 8: Remote OS Detection”. [Online]: Available: https://nmap.org/book/osdetect.html [Accessed: 2-Jun-2017].

[20] Nmap, “OS Detection”. [Online]: Available: https://nmap.org/book/man-os-detection.html [Accessed: 2-Jun-2017].

[21] Nmap, “Nmap-os-db: latest version”. [Online]: Available: https://svn.nmap.org/nmap/nmap-os-db [Accessed: 2-Jun-2017].

[22] C. Cimpanu, “New Mirai Botnet Slams US College with 54-Hour DDoS Attack”. [Online]. Available: https://www.bleepingcomputer.com/news/security/new-mirai-botnet-slams-us-college-with-54-hour-ddos-attack/ [Accessed: 1-Jun-2017].

[23] M. Heller, “Modified Mirai botnet could infect five million routers”. [Online]. Available: http://searchsecurity.techtarget.com/news/450403881/Modified-Mirai-botnet-could-infect-five-million-routers [Accessed: 1-Jun-2017].

[24] N. Woolf, “DDoS attack that disrupted internet was largest of its kind in history, experts say” [Online]. Available: https://www.theguardian.com/technology/2016/oct/26/ddos-attack-dyn-mirai-botnet [Accessed: 1-Jun-2017].

[25] Malware Tech, “Mapping Mirai: A Botnet Case Study”. [Online]. Available: https://www.malwaretech.com/2016/10/mapping-mirai-a-botnet-case-study.html [Accessed: 1-Jun-2017].

[26] Shodan, “Search Query”. [Online]. Available: https://www.shodan.io/search?query=ubuntu [Accessed: 20-May-2017]

[27] Nmap, “Nmap OS Detection Database”. [Online]: Available:

https://nmap.org/book/nmap-os-db.html [Accessed: 20-May-2017].

[28] Nmap, “TCP/IP Fingerprinting Methods Supported by Nmap”. [Online]: Available: https://nmap.org/book/osdetect-methods.html [Accessed: 20-May-2017].

[29] Nmap, “Device Types”. [Online]: Available: https://nmap.org/book/osdetect-device-types.html [Accessed: 20-May-2017].

[30] Nmap, “Common Platform Enumeration (CPE)”. [Online]. Available: https://nmap.org/book/output-formats-cpe.html [Accessed: 20-May-2017].

[31] Nmap, “Understanding an Nmap Fingerprint”. [Online]. Available: https://nmap.org/book/osdetect-fingerprint-format.html [Accessed: 20-May-2017].

[32] Scapy, “Scapy”. [Online]. Available: http://www.secdev.org/projects/scapy/ [Accessed: 10-Mar-2017].

[33] Scapy, “Scapy Documentation”. [Online]. Available: https://scapy.readthedocs.io/en/latest/index.html [Accessed: 10-Mar-2017].

[34] Python IPTables, “ Python IPTables Documentation”. [Online]. Available: http://python-iptables.readthedocs.io/en/latest/intro.html [Accessed: 10-Mar-2017].