Detecting and Analyzing Zero-day Attacks using Honeypots

Emma Mirica
University "Politehnica" of Bucharest
Computer Science and Engineering Department
Bucharest, Romania
Email: emma.mirica@cti.pub.ro

Constantin Musca
University "Politehnica" of Bucharest
Computer Science and Engineering Department
Bucharest, Romania
Email: constantin.musca@cti.pub.ro

***Abstract*—Computer networks are overwhelmed by self propagating malware (worms, viruses, trojans). Although the number of security vulnerabilities grows every day, not the same thing can be said about the number of defense methods. But the most delicate problem in the information security domain remains detecting unknown attacks known as *zero-day attacks*. This paper presents methods for isolating the malicious traffic by using a honeypot system and analyzing it in order to automatically generate attack signatures for the Snort intrusion detection/prevention system. The honeypot is deployed as a virtual machine and its job is to log as much information as it can about the attacks. Then, using a protected machine, the logs are collected remotely, through a safe connection, for analysis. The challenge is to mitigate the risk we are exposed to and at the same time search for unknown attacks.**

***Index Terms*—Honeypot, Zero-day attacks, Intrusion detection/prevention system**

I. Introduction

There has been an exponential growth in the number of attacks and the current security mechanisms can't keep up. Every day hackers find new ways to break into systems, destroying, modifying and stealing private data. They bypass firewalls, security appliances, anti-virus applications and nothing seems to stop them.

The primary goal of this research paper is to reinvent the way information security research is done. We need to change the approach used in detecting unknown attacks. *Sun Tzu* said centuries ago that if you know the enemy and know yourself, you need not fear the result of a hundred battles.

Instead of building firewalls and writing intrusion detection and prevention systems, we are going to lure in attackers and study their penetration methods. The primary concept in our research is represented by the *honeypot* term, which signifies a system similar to a production one that is used to gather information about the attackers and their attack vectors.

We use an isolated environment (a virtual machine) to deploy the honeypot system, which consists of software components that constantly analyze what is happening to the system. By doing so we don't have the needle in the haystack problem, which is trying to figure out whether the traffic is malicious or not. The honeypot system will have only malicious activity because it will not be used as a production system.

Using a protected machine we will capture the collected data through an encrypted tunnel and then process it. The aim of this research is to build a system that automatically detects unknown attacks and generates signatures for the Snort intrusion detection/prevention system. In other words, we deploy an attack analysis framework on the protected machine and create IDS/IPS signatures by analyzing the incoming traffic on the honeypot.

In order to successfully deploy a honeypot, we need to satisfy the data control, data capture, data analysis and data collection requirements. Thus, we have to mitigate the risk of having the attacker circumvent the container's security mechanisms. The best way to do this is to use a combination of different security layers, which makes it hard for the attacker to penetrate the security container. Also, we have to capture as much information as we can without being detected by the attacker. In the end, the data is transformed into information using the zero-day attack detection framework.

The remainder of this paper is organized as follows: Section II presents an overview of related work. Section III discusses in detail the system's architecture. The implementation is presented in Section IV and some experimental results are described in Section V. In the end, we summarize the contributions and future work in Section VI.

II. Related Work

A similar work is presented by *Reshma R. Patel* in his *"Zero-Day Attack Signatures Detection using Honeypot"* article [1]. He describes *Honeycomb*, a *Honeyd* host-based detection system extension, which applies the *Longest Common Substring* (LCS) algorithm on per-service data captured by the honeypot.

Honeyd is a framework that simulates computer systems at the network level in order to fool the attackers. The strategy used is to put the honeypot close to the production servers and start common services like *Telnet*, *Hyper Text Transfer Protocol*, *File Transfer Protocol* in order to lure the attackers.

The main idea is that all the incoming traffic on the honeypot is considered malicious and is used to generate Snort and Bro signatures. However, the problem is that it suffers from false positives because it can overgeneralize per-service data and generate a non-attack signature.

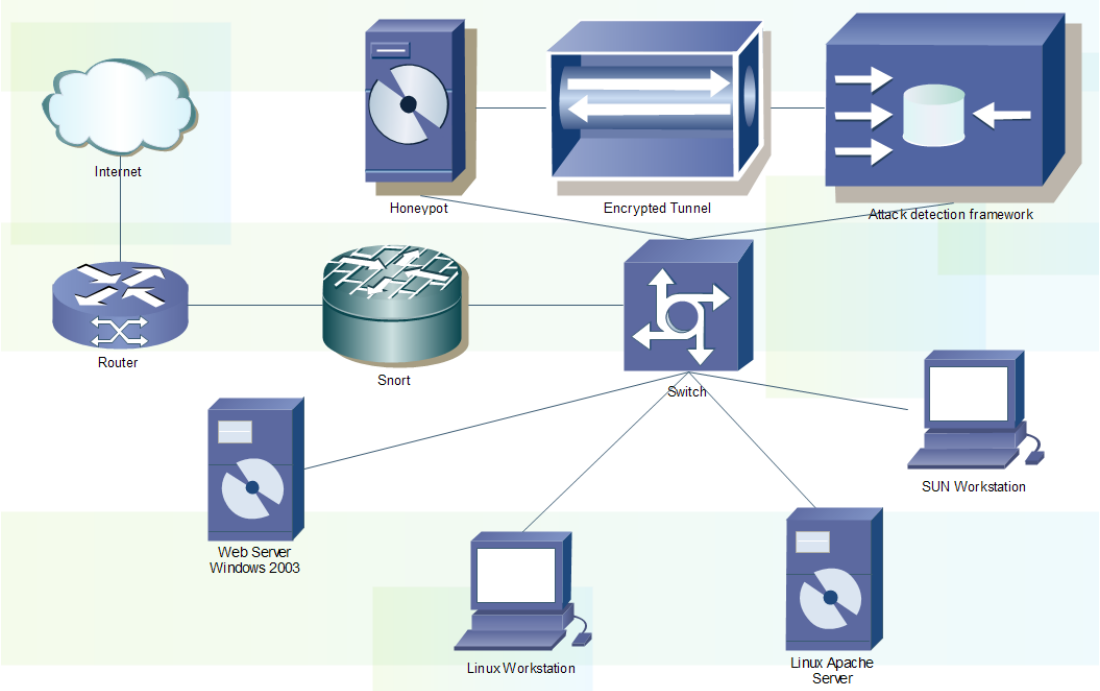

Fig. 1. Architecture

Honeycomb also uses the *Dynamic Taint Analysis* for tracking incoming data from the network throughout the processes. Tainting the data permits the system to send alerts when tainted data is used in sensitive operations like JMP. Reshma Patel's system was able to successfully detect the Slammer worm and create a signature for it.

The challenge with the *Dynamic Taint Analysis* approach is represented by the propagation of the taint marks. We taint the memory by associating a tag (a number) with every chunk of memory allocated. The method's primary purpose is to detect Illegal Memory Accesses (IMAs), but it can also be used to monitor how the data received through a network socket is used later in the program. The *"Effective Memory Protection Using Dynamic Tainting"* [2] article presents an extensive research regarding the propagation of taint marks in order to detect illegal memory accesses. Every time a chunk of memory is allocated, a tag is associated with both the memory and the corresponding pointer. When one accesses a memory address <m> using a pointer <p>, the dynamic tainting component stops the execution if the tags are different. By marking and tracking the data at runtime, one can verify if there are any memory faults.

Another interesting article is *"Detecting Targeted Attacks using Shadow Honeypots"* [3]. The proposed solution combines the following components: a filtering component, anomaly detection sensors and a shadow honeypot. A *shadow honeypot* is another instance of the protected software and shares the internal state with the regular application. The difference is that the application's original source code is patched with the shadow honeypot code. The shadow honeypot is instrumented to detect specific types of attacks. Based on the anomaly detection sensors' prediction, the system calls the shadow instance or the regular application.

Hence the main idea of this project is to combine the advantages of a shadow honeypot and an anomaly detection system to prevent malicious requests to the protected application. This way, when a request is received the filtering component will block known attacks or will pass the request to the next stage to be analyzed by one or more anomaly detection sensors. The anomaly sensors will signal the system if the request is potentially dangerous. If it is, the request is processed by the shadow honeypot, which will notify the filtering process that the attack was real (this way further requests will be blocked by the filtering component). If the request doesn't present any danger, it will be processed by the application itself.

The shadow honeypot implementation was successfully tested against some known exploits such as Mozilla PNG exploit and other Apache-specific exploits. The results were promising despite the considerable cost of processing suspicious traffic on the shadow honeypot.

III. Architecture

In order to lure attackers, we must first set our trap. The honeypot (or eventually honeypots) will have to be placed in our network alongside other systems: several workstations that run different operating systems, servers and others. The network will be protected by an IDS/IPS system that will be improved by using our setup: a honeypot that communicates through an encrypted channel with a protected machine where our implementation of an attack detection framework is running.

The system's general architecture is illustrated in Figure 1. It is a simple and efficient approach of detecting unknown network-based attacks. Its major components are: the honeypot system, a framework that generates signatures and a filtering component.

The filtering component is actually an intrusion detection/prevention system (such as *Snort*). Its purpose is to block attacks based on the signatures it knows. This component analyzes the traffic and based on its signatures, the traffic is dropped or passed as it is malicious or not. The traffic passing through the filtering component is logged by the honeypot component. The honeypot doesn't do any processing. It only captures some information. The framework is implemented on another machine, a protected one. This machine will retrieve the information saved on the honeypot through a secure channel. This framework analyzes the logs and based on different methods it generates new signatures for the filtering component.

The idea is to build the honeypot system using a virtual machine or *Honeyd*. This honeypot will have many common services running (such as apache, postfix, etc.) in order to make it seem more valuable and lure in the attackers.

The filtering system can be implemented using *Snort*. *Snort* is an open source network intrusion detection and prevention system that can work both on Unix and Windows.

The framework will implement different methods (for example memory tainting) of detecting and analyzing attacks.

The processing logic of the system is: when traffic flows through the filtering component, it will be analyzed by the filtering component based on the signatures it knows. If the traffic turns out to be malicious the filtering component will not let it pass. On the other hand, if the traffic doesn't match any signature it will flow through the network, including the honeypot system, which will log information about it. Based on the logs it retrieves from the honeypot, the framework runs several algorithms and generates new signatures if it detects unusual activity. This way, the filtering component is improved and it can detect previously unknown attacks.

The first step of our project was to build the honeypot and to collect information. We used two types of honeypots: a virtual machine and a virtual honeypot created with *Honeyd*. Also, in order to obtain valuable and secure information about the honeypot's state, we process these logs on a protected machine. More details will be given in the next sections.

IV. Implementation

As was mentioned before, we only focused on collecting information from the honeypot. This raised some questions: what information is valuable, how to transfer this information securely to the protected machine, what statistics are relevant in defining and detecting new attacks.

A honeypot can also be classified according to the level of interaction the attacker has with it. And so we have *low-interaction honeypots* and *high-interaction honeypots*. The first one can be a port listener program that logs any connection without doing an actual task. On the other hand, the high-interaction honeypot can be a server that runs real services. As one can see, the second category is riskier as with a low-interaction honeypot the only thing the attacker can do is open and close some ports.

We decided to implement our solution for these two different types of honeypots. The next sections describe these two solutions and their specific implementation.

A. Implementing data collection on Honeyd

A **virtual honeypot** [4] is a software application designed to appear to be a real functioning network, but is actually used to be probed and attacked by malicious users. The difference between a *honeypot* and a *virtual honeypot* is that the first one is a hardware device that lures attackers into its trap, whereas the second one is a software that emulates a network. With the classification from the previous section, a virtual honeypot is a low-interaction honeypot.

Honeyd [5] is defined as "a small daemon that creates virtual hosts on a network". It is an open source software released under GNU General Public License. The *Honeyd* framework is very flexible as it offers users the option to configure different arbitrary services, which appear to be running on different operating systems. Using *Honeyd* it is very easy to configure a virtual honeypot.

When talking about virtual honeypots, Honeyd is the best choice as:

- It is free and easy to configure and deploy.
- It can emulate different operating systems: Windows NT, Windows 2000, Linux, Solaris, Cisco etc.
- It emulates services, not only at application level, but also at the TCP/IP stack level.
- It offers logging capabilities. It can log TCP, UDP, ICMP activity, but one can also add logging capabilities to the scripts that emulate services.
- It can be easily installed on a Windows or Linux machine.

We installed *Honeyd* on a customized Linux machine used mostly for penetration testing, called *Backtrack* [6].

To create a new honeypot machine we must write a configuration file. Through this configuration file we tell honeyd what operating system the honeypot should have, what ports to open, what services to run and so on. A simple example of a configuration file can be seen in Listing 1.

```
create default
set default default tcp action block
set default default udp action block
set default default icmp action block

create windows
set windows personality "Microsoft Windows XP
    Professional SP1"
set windows default tcp action reset
add windows tcp port 135 open
add windows tcp port 139 open
add windows tcp port 445 open

set windows ethernet "00:00:24:ab:8c:12"
dhcp windows on eth0
```

Listing 1. honeyd.conf

The honeypots are created using the **honeyd** command (see Listing 2).

```
# honeyd  -d  -f  honeyd.conf
```

Listing 2. The honeyd command

The "-d" option tells honeyd not to run in background and so to allow for more verbose output [7].

Using "create" within the configuration file we create a new template for a honeypot. This way we can create as many honeypots as we want within the same file. For a honeypot we set the personality, meaning when another device connects to this honeypot it will appear to have the personality set (Windows XP Pro SP1 in our case). We can also set some common open ports, and the action if the traffic is not directed to the open ports defined in this configuration file. We can also set the MAC address and tell the template to acquire an IP address from dhcp. The challenge is to simulate a real operating system. One must know what are the properties that define an operating system. In our example, we created a Windows machine and we also opened three common ports (135, 139, 445) for a Windows system.

With a simple setup like this, it is obvious why we chose Honeyd for our project. Another reason is the logging capabilities of Honeyd. After deploying a virtual machine using Honeyd, an output similar to the one listed in Listing 3 is shown. These logs are written in the */var/log/syslog* file.

```
Honeyd V1.5c Copyright (c) 2002-2007 Niels Provos
honeyd[1870]: started with -d -f honeyd.conf
Warning: Impossible SI range in Class fingerprint "
    IBM OS/400 V4R2M0"
Warning: Impossible SI range in Class fingerprint "
    Microsoft Windows NT 4.0 SP3"
honeyd[1870]: listening promiscuously on eth0: (arp
    or ip proto 47 or (udp and src ...
honeyd[1870]: [eth0] trying DHCP
honeyd[1870]: Demoting process privileges to uid
    65534, gid 65534
honeyd[1870]: [eth0] got DHCP offer: 192.168.99.135
honeyd[1870]: Updating ARP binding: 00:00:24:c8:e3
    :34 -&gt; 192.168.99.135
```

Listing 3. Honeyd log output

With this kind of input we can easily obtain important information about the connections on the honeypot. For now we are interested in information regarding connection requests, established connections, connection resets, ARP replies and ICMP replies. For this purpose we used the dictionaries shown in Listing 4.

```
# ip statistics table
ip_stats = {'<entry_type>' : {'<source_ip>' : {'<
    dst_ip>' : [('<protocol>','<dst_port>')]}}}

# arp statistics table
arp_stats = {'<ip_address>' : ['<mac_address>']}

# icmp statistics table
icmp_stats = {'<dst_ip>' : ['<src_ip>']}
```

Listing 4. Honeyd processing dictionaries

To create the *ip_stats* dictionary we had to parse the entries that concerned the connections. Honeyd logs information about connection requests, when a connection is established and when the connection is dropped by reset. The *entry_type* key in this dictionary represents exactly this type of information. It identifies which event occurred: a connection request, a connection was established or a connection was reset. These statistics can indicate different types of attacks: ping sweep, flood, scanning or Denial of Service.

The *arp_stats* dictionary is created based on the information received when an ARP request was made. This is useful when we want to check if a machine has changed its MAC address, indicating a possible spoofing attack.

The *icmp_stats* are very useful to know how many times an IP address was pinged by other IPs, possibly indicating a distributed attack when different IPs try to ping the same honeypot, or if we have more virtual honeypots in our network we can identify when an attacker does a ping sweep.

Relevant output is obtained based on these dictionaries. The script that generates useful output is written in Python and offers arguments to obtain statistics about connection requests, established connections, connection resets and ARP and ICMP stats. More about the output obtained and how to use this script in Section V-A.

B. Implementing data collection on Metasploitable

To implement the data collection on a honeypot built as a virtual machine, we chose the **Metasploitable2** solution [8]. This is an intentionally vulnerable Linux machine used to conduct security training, test security tools and practice some common penetration techniques. It can be downloaded from [9]. It is compatible with *VMware*, *VirtualBox* and others. By default, Metasploitable's network interfaces are bound to the NAT and Host-only network adapters, and the image should never be exposed to a hostile network.

Using *nmap* command we can easily find out what services run on the Metasploitable machine (Listing 5).

```
root@ubuntu:~# nmap -p0-65535 192.168.149.132
Starting Nmap 5.61TEST4 ( http://nmap.org ) at
    2012-05-31 21:14 PDT
Nmap scan report for 192.168.149.132
Host is up (0.00028s latency).
Not shown: 65506 closed ports
PORT      STATE SERVICE
21/tcp    open  ftp
22/tcp    open  ssh
23/tcp    open  telnet
25/tcp    open  smtp
53/tcp    open  domain
80/tcp    open  http
111/tcp   open  rpcbind
139/tcp   open  netbios-ssn
445/tcp   open  microsoft-ds
512/tcp   open  exec
513/tcp   open  login
514/tcp   open  shell
1099/tcp  open  rmiregistry
1524/tcp  open  ingreslock
2049/tcp  open  nfs
2121/tcp  open  ccproxy-ftp
3306/tcp  open  mysql
3632/tcp  open  distccd
5432/tcp  open  postgresql
5900/tcp  open  vnc
```

```
6000/tcp  open  X11
6667/tcp  open  irc
6697/tcp  open  unknown
8009/tcp  open  ajp13
8180/tcp  open  unknown
8787/tcp  open  unknown
39292/tcp open  unknown
43729/tcp open  unknown
44813/tcp open  unknown
55852/tcp open  unknown
MAC Address: 00:0C:29:9A:52:C1 (VMware)
```

Listing 5. Check for open ports on Metasploitable2 machine

Using the information shown in Listing 5, we can easily get a root console, as almost every one of these services provides a remote entry point into the system.

The easiest way is by using TCP ports 512, 513 and 514 that are known as "r" services. These have been configured to allow remote access from any host (one can check the *.rhosts* file on the Metasploitable). An attacker only needs to install *rsh-client* on his machine and by running a command similar to the one in Listing 6 he can obtain a root console on the honeypot.

```
# rlogin -l root 192.168.149.132
```

Listing 6. Obtaining root console on Metasploitable

As opposed to Honeyd, Metasploitable is a virtual machine and so it is a high-interaction honeypot. Honeyd has many features, one of them being the logging capabilities. Metasploitable doesn't offer this feature and so we decided to collect important logs, transfer them to the protected machine, and then process them. To protect the collected logs from being tampered with, we decided that the protected machine should retrieve these logs remotely, by running the following scripts: *log_fetcher.sh* and *log_archiver.sh*.

```
#!/bin/bash

LOG_DIR=/tmp/log
SYSLOG_DIR=/var/log
shred_folders=(${LOG_DIR} ${SYSLOG_DIR})

# create LOG_DIR if necessary
[ -d ${LOG_DIR} ] || mkdir -p ${LOG_DIR}

# remove all .gz and rotated files
find /var/log -type f -regex ".*\.gz$" -delete
find /var/log -type f -regex ".*\.[0-9]+$" -delete

# get process log
ps -A -o pid,ppid,comm,%cpu,time,%mem,eip,esp,ni,
    euser,ruser --sort ni > ${LOG_DIR}/ps.log

# get TCP/UDP general log
netstat -ltn | tr -s " " | cut -d" " -f4,7 > ${
    LOG_DIR}/netstat.log

# get list of installed packages
dpkg --get-selections > ${LOG_DIR}/installed-pkgs.
    log

tar -cz -C ${SYSLOG_DIR} . -C ${LOG_DIR} .

# shred log files
for folder in "${shred_folders[@]}"
do
        find ${folder} -type f -exec shred -z {} \;
done
```

Listing 7. log_archiver.sh

Basically the script does the following:

- Identifies important logs: system logs, daemon logs, open ports stats, kernel logs, processes stats, installed packages and so on. These are found in the */var/log* folder. Also some information is generated using netstat and ps.
- Shreds the file as we don't want to analyze the same information more than once. We also had to be careful not to lose valuable information between two collecting actions.

This script is periodically called on the protected machine by the *log-fetcher* script. Because we wanted to generate as little traffic as possible, we archive the logs and then transfer them to the protected machine.

```
#!/bin/bash

HONEYPOT_ADDR="192.168.149.132"
LOGS_DIR=logs

rm -rf ${LOGS_DIR}
mkdir -p ${LOGS_DIR}
ssh root@${HONEYPOT_ADDR} 'bash -s' < log_archiver.
    sh | tar -C ${LOGS_DIR} -xvz
```

Listing 8. log_fetcher.sh

We used the *ssh* protocol to retrieve the logging information remotely. To avoid being asked for the password every time the scripts were executed, we generated a new public key on our protected machine using *ssh-keygen* and then copied it on the Metasploitable2 machine using *ssh-copy-id*. Now we can run the *log_fetcher* script without being prompted for a password.

Next, on the protected machine, we must analyze the state of the honeypot. This is done by processing the periodically collected logs. Every time the log is fetched we check to see what happened on the honeypot since the last time we verified. Basically, we look for new root processes, installed packages or listening ports. A new root process tells us that an attacker managed to obtain administrator privileges, which allowed him to open a backdoor in the system. Our processing script captures important events like these in order to study the attacker's behavior. We parse the *ps* output and create a list with the logged processes, which is compared to the previous list built. Similarly, we generate lists for installed packages and listening ports using *netstat* and *dpkg* outputs. In addition, we check to see if there are new established TCP connections and log valuable pieces of information about them.

Because a process represents an important entity in an operating system we collect a lot of metadata about it: PID, PPID, CPU utilisation, EIP/ESP registers and effective/real user id. We use this information to infer which executables are used on the ongoing attack and to determine the attacker's target.

Furthermore, we retrieve and process all the logs from the daemons installed on the honeypot. In case the attacker obtains access to the SMTP server and he sends spam our script detects

this immediately. Thus, we analyze all the important events logged by the common services and try to correlate them.

Another important source of information is the *dmesg* log. We use it to detect if somebody inserted a kernel module, which can act as a rootkit. This malicious software hides it's existence from the other processes and it's critical to recognize it from the beginning.

In contrast to the honeyd solution, the real honeypot approach allows us to obtain more information about the attacker. Every service on the honeypot system is continuously monitored by the protected machine in order to capture all the steps of an attack.

V. Experimental Results

As shown in the previous section, is far easier to implement the collecting phase on the Honeyd. Because of its logging feature we only had to worry about the processing script. The Honeyd implementation has several advantages:

1) The effort estimation based on lines of code is less than the effort of the Metasploitable implementation.
2) It can simulate several honeypots with little effort and low overhead.
3) We tried to reduce the traffic generated with the Metasploitable solution, but the Honeyd solution doesn't generate traffic at all. The Honeyd already runs on the protected machine, and so we don't have to transmit the logs over the network.

The disadvantage is that the output obtained with Honeyd isn't as verbose as the information we can get from Metasploitable.

The next two sections summarize the results we obtained with our implementations.

A. Results obtained with Honeyd

The script developed for parsing honeyd logs and for generating reports is presented in the Listing 9 below:

```
usage: honeyd_log_parser [-h] [-f FILE] [-v]
    [-req [REQUEST]] [-rst [RESET]]
    [-est [ESTABLISHED]] [-i [ICMP]] [-a [ARP]]

Honeyd log parser

optional arguments:
  -h, --help
        show this help message and exit

  -f FILE, --file FILE
        specify the output file

  -v, --verbose
        verbose output

  -req [REQUEST], --request [REQUEST]
        list request information grouped by source
        IP address

  -rst [RESET], --reset [RESET]
        list reset information grouped by source
        IP address

  -est [ESTABLISHED], --established [ESTABLISHED]
        list established information grouped by
        source IP address

  -i [ICMP], --icmp [ICMP]
        list icmp reply information grouped by
        destination IP address

  -a [ARP], --arp [ARP]
        list arp information grouped by IP address
```

Listing 9. Honeyd log parser

One can specify the report type by using the *honeyd_log_parser* arguments listed in the script's usage message.

In order to demonstrate the script's functionality we generated a *–icmp* report for a flood attack.

TABLE I
ICMP REPORT

Destination IP	Source IP	Pings
192.168.149.151	192.168.149.254	1
	192.168.149.135	123
192.168.149.132	192.168.149.254	1
	192.168.149.135	10

This report presented in Table I shows that the machine with the *192.168.149.135* IP has flooded the *192.168.149.151* machine using the ICMP protocol. Also, the *192.168.149.132* and *192.168.149.151* honeypot machines were pinged by the same IP, possibly indicating a ping sweep done by *192.168.149.135*.

Another interesting report is obtained running the *honeyd_log_parser* with *-req* option. The result is shown in Table II.

TABLE II
TCP REQUEST REPORT

Source IP	Destination IP	Protocol	Port	Num of conns
192.168.149.135	192.168.149.151	tcp	445	3
		tcp	139	3
		tcp	135	3
	192.168.149.132	tcp	445	2
		tcp	139	2
		tcp	135	1
192.168.149.148	192.168.149.132	tcp	445	2
		tcp	139	2
		tcp	135	41

It can be seen that the *192.168.149.135* machine scanned the standard windows ports of the *192.168.149.151* and *192.168.149.132* machines for potential vulnerabilities. This setup uses two virtual honeypots created similar to the one presented in Section IV-A. Also, another machine scanned the *192.168.149.132* standard ports, possibly indicating a distributed attack.

One can also check the MAC address and IP association using *-a* option of the script. This will show if a machine had several MAC addresses indicating a possible spoofing attack. The report obtained is similar to the one shown in Table III. When a spoofing attack occurs, the line for the IP address that changed its MAC address will contain more than one MAC entry.

TABLE III
ARP REPORT

IP Address	MAC Address
192.168.149.254	00:50:56:fd:ff:6a
192.168.149.135	00:0c:29:6e:d0:c2

B. Results obtained with Metasploitable

The implementation for Metasploitable can analyze different types of information. This is because Metasploitable is an actual system that offers us the opportunity to collect different statistics compared to the ones obtained from Honeyd.

The arguments of the Metasploitable script that is used to process logs and generate reports are presented in Listing 10:

```
usage: honeypot_log_parser [-h] [-f FILE] [-ps]
                           [-ns] [-pkgs]

Honeypot log parser

optional arguments:
  -h, --help
      show this help message and exit
  -f FILE, --file FILE
      specify the output file
  -ps, --process
      list process relevant information
  -ns, --netstat
      list netstat relevant information
  -pkgs, --packages
      list package-related information
  -mods, --modules
      list kernel modules information
```

Listing 10. Honeypot log parser

The user can configure the script to listen for process, port, installed package or kernel module events. He can also choose to run the script with all of the options.

The script can be run as a regular user, that has ssh access to the honeypot. By running this script with the *-ps*, *-ns*, *-pkgs* and *-mods* arguments we get an output similar to the one presented in Listing 11.

```
New root process (PID:3451)
New installed package (nc)
New listening port (TCP:555)
New kernel module (syscontrol)
```

Listing 11. Honeypot log events

In this example we can see that a new root process (with pid 3451) is running. This could indicate that an attacker has root permissions and is now running a possible malicious process with full permissions. Also, a new package has been installed (nc). This clearly indicates that somebody else has root access to our honeypot. By running this script every few seconds we can capture information about new listening ports, new installed kernel modules and packages and new root processes.

As one can see with the Metasploitable implementation we can obtain useful information about processes as opposed to Honeyd implementation, where we only obtain information about the connections on the honeypot.

VI. CONCLUSION

Honeypots are a powerful tool for detecting unknown attacks. Because it only has malicious traffic, it is easier to identify an attack. They can be classified as low-level interaction and high-level interaction honeypots.

We were able to implement our processing logic for both types of honeypots: a high-interaction honeypot (using Metasploitable) and a low-interaction honeypot (using Honeyd). Both solutions are promising and give relevant output. As this paper shows, it is easier to use and implement a detecting method for Honeyd as it offers logging capabilities. On the other hand, we can get more valuable information by using a high-interaction honeypot. We can understand the attacker's method by analyzing his steps through our network.

A. Future Work

In this paper we developed a tool that analyzes logs from the honeypot. We were able to identify some simple attacks by analyzing our scripts' output. We plan to improve these scripts in order to identify other attacks. Also, based on these logs, we want to implement different types of algorithms to detect attacks. These algorithms will be incorporated in the detection framework that will generate new signatures for the Snort system.

ACKNOWLEDGMENT

We want to thank Laura Gheorghe for her constant support and useful advice. Her comments were always very helpful and motivating. We also thank Traian Popeea for his thorough review of the paper and Razvan Deaconescu for his help with our project's wiki and repository.

REFERENCES

[1] R. R. Patel and C. S. Thaker, "Zero-day attack signatures detection using honey-pot," *International Conference on Computer Communication and Networks CSI-COMNET-2011*, vol. 1, no. 1, pp. 4–27, 2011.

[2] J. Clause, I. Doudalis, A. Orso, and M. Prvulovic, "Effective memory protection using dynamic tainting," *Effective Memory Protection Using Dynamic Tainting*, vol. 1, no. 1, pp. 4–27, 2007.

[3] K. G. Anagnostakis, S. Sidiroglou, P. Akritidis, K. Xinidis, E. Markatos, and A. D. Keromytis, "Detecting targeted attacks using shadow honeypots," *Proceedings of the 14th USENIX Security Symposium*, vol. 1, no. 1, 2005.

[4] N. Provos and T. Holz, *Virtual Honeypots: From Botnet Tracking to Intrusion Detection*, 1st ed., 2007.

[5] "Honeyd development," http://www.honeyd.org/, [Online; accessed 12-10-2012].

[6] "Backtrack - linux machine," http://www.backtrack-linux.org/, [Online; accessed 11-09-2012].

[7] "Honeyd tutorial," http://travisaltman.com/honeypot-honeyd-tutorial-part-1-getting-started/, [Online; accessed 12-10-2012].

[8] "Metasploitable2 - linux vulnerable machine," https://community.rapid7.com/docs/DOC-1875, [Online; accessed 11-01-2012].

[9] "Metasploitable2 - download link," http://sourceforge.net/projects/metasploitable/files/Metasploitable2/, [Online; accessed 11-01-2012].