Different Data Types in Databases and their Effect on Cyber Attacker Behavior

Ezinwa Agbo, Michael McDonald, Hannah Sitther, Ansh Tyagi

Group 1H

HACS200

Executive Summary

The goal of this project was to gain knowledge on how attacker behaviors might change in relation to the type of data within a database that they attacked and entered. Our team sought to answer the question of "How does the type of data files in a database affect the time an attacker spends in the database, the amount of commands used, and whether or not they deploy malware?" For the purpose of the experiment malware will be defined as any intrusive software that is installed or deployed into a system by an attacker that serves the purpose of data manipulation or maintaining persistence and allowing entry in a different manner than our honeypot login We wanted to know if the existence of certain files types, such as mp3, mp4, text or even no files at all would have any bearing on what an attacker might do. Our team had two alternative hypotheses for this experiment with the first being that we believed databases with mp4 and mp3 files would have more time spent in them compared to databases with text or no files in them. Our second alternative hypothesis was that the mp3 and mp4 databases would have a higher use of <*curl*> and <*wget*> commands and therefore more malware deployed compared to text file and no file databases. Our null hypothesis was that there would be no significant difference in time spent, amount of commands used, or amount of occurrences of <*curl*> and <*wget*> for malware deployment. We collected data on the time spent by each attacker that entered our honeypots, along with the number of commands used per attacker that entered, the overall occurrence of the <*curl*> command in each honeypot configuration, and the overall occurrence of the <*wget*> command in each honeypot configuration. Our team found that the average time spent per attacker in each honeypot, average amount of commands used per attacker in each honeypot, and overall occurrences of the <*curl*> and <*wget*> commands per honeypot configuration are statistically insignificant.

Background Research

There are a few examples of research done that have some similarities to the experiment we will conduct. One study conducted back in 2007 sought to profile attacker behavior after an SSH compromise and used a similar experiment design to ours, utilizing four honeypots running SSH to collect data relating to attacker login and commands ran. Their experiments found that only 0.31% of attacks on their honeypots were successful, with a successful attack being defined as guessing the correct password and gaining entry into the honeypot (Ramsbrock et al., 2007). Of those successful attacks only 22.09% of the time did the attacker run any commands (Ramsbrock et al., 2007). However it is this study that gives us reason to believe that some amount of attackers will run commands within our honeypots. It is for this reason that we will collect data on the number of commands run for every attacker that enters our honeypots. While we expect to see commands run by at least a small number of attackers, even if we don't that in itself tells a lot about their behavior and provides interesting discussion points.

Another research team set up an experiment to study attacker behavior using a singular high interaction honeypot that monitored attacker keystrokes and commands. They found that out of 210 successful breaches of their honeypots, 96 of the attackers used the <*wget*> command following their breach to download malicious programs from the internet (Nicomette et al., 2011). This experiment gives us reason to believe that some amount of attackers who enter our honeypot will install some form of malware or at the very least use the <*wget*> command. 30% of attackers uploaded their malware through an alternate SSH connection through the use of the *sftp* command, which gives more reason to believe malware could be deployed in our honeypots. Based on this research our team would expect to see a moderate usage of the <*curl*> and <*wget*> commands within our honeypots which is why we will be counting the overall occurrences of

those two commands separately for each of our honeypot configurations. As we were informed that most attackers on UMD's network are bots, our team felt it would be wiser to track overall usage as opposed to usage per attacker since statistically most of those attackers wouldn't be using malware. Furthermore, our team recognized from this research that there were alternate ways attackers could deploy malware but with the existence of so many different commands, we decided to monitor for usage of <*curl*> and <*wget*> due to what this past study found

Another study utilized 102 medium-interaction honeypots and observed how differences in honeypot location, the difficulty of break-in, and the population of files would affect attacker behavior (Barron et al., 2017). They found that 80% of both human and bot attackers execute less than 10 commands after gaining entry into a honeypot and 74% of bot attackers attempt to download and execute malware. These two factors provide us with even more reason to believe that we need to monitor the amount of attacker commands, and the amount of times malware is installed. Since this research found most attackers use only a few commands, we decided to set our recycling time to every three minutes, since that would be sufficient time for the average attacker, but also leave time for possible outliers to spend more time in the honeypots.

Finally, a study conducted at a Swedish university in 2010 sought to use honeypots to categorize attacker skill level based on the exploited vulnerabilities in networks. Of the five attackers that managed to enter the honeypot there were a list of tools utilized by them: *wget.exe* to upload a file, *goonshell.php* which is a backdoor to get shell access through the uploading of the file, *install.bat* to control the distribution of the uploaded files, *scc.exe* which serves as a backdoor, *y.php* to control processes via the web, along with a few other tools used to maintain control of the honeypot (Aliyev, 2010). The use of these tools within this study again gives us reason to expect the use of malware and commands to upload malware and interact with the

honeypot. While there were a couple different commands used, <*wget*> was once again present, which was a trend within the other research which gave our team further reason to believe that monitoring the usage of <*wget*> as opposed to a different command would yield us more data.

Experimental Design Changes

First and foremost our experimental question was slightly altered before we began even writing the code for our honeypots. Our team was told numerous times by different faculty members that malware would likely be quite rare and that it shouldn't be something that we rely upon a lot for data, so the part of our experimental question aiming to see how the data type of files in a database would affect whether or not an attacker deploys malware was scrapped and we instead sought so see if the overall occurrences of the <*wget*> and <*curl*> commands would be higher within a certain honeypot configuration compared to another. This change would allow us to have more statistical significance within our results by looking at a broader scale. If we checked the deployment of malware per attacker the average would be zero with the occasional attacker having 1, and when conducting statistical analysis the tests would indicate that the result is statistically insignificant. However by comparing overall occurrences between the honeypot configurations the differences we're comparing would have a higher chance of being statistically significant since we would only be comparing four values to each other. And we chose <*wget*> and <*curl*> to monitor since the research we did prior to designing the experiment suggested that <*curl*> and <*wget*> would be the most common methods of deploying malware.

Our original design was to have 4 different honeypots with different configurations, each one holding a different data type. One would hold mp4, mp3, and text files, while a control honeypot would have no files in them. The files would each have 20 copies placed into a

directory with the name of the file type. The attackers would have root permissions so they could do whatever they possibly wanted within the honeypots. The files we would use would be security camera footage for mp4, lecture recording audio for mp3, and fake personal data for students and faculty for text files. The honeypots would also recycle every 5 minutes, kicking out the current attacker if they are still present and resetting the environment to its default state for the next attacker.

The first technical change we made was to take away root permissions for attackers in our honeypot. This was made on week seven after we were already late on deployment so we could fulfill the requirement that two user's could not enter and be inside a honeypot at the same time. We originally added a rule in the /etc/security/limits.conf file that would restrict max logins for all users to one, which in theory should've prevented all user's from being logged in to a honeypot at the same time as another user. However a surprising discovery was that this rule, along with all others we tried didn't apply to the root users. To circumvent this we took away root permissions for all attackers. This was a simple fix that prevented us from having to write more complex IP table rules when we were already late for deployment.

The second and final technical change made was on week eight which was lowering the recycle time from 5 minutes to 3 minutes. Our initial data showed that almost every attacker was in the honeypot for 15 seconds or less, so keeping the recycle time at 5 minutes would simply be wasting time. If we were to lower the time to three minutes we would still be giving ample time for outliers to spend time on the honeypot, while also giving ourselves more data through the increased frequency of recycling.

Hypothesis

We have two alternative hypotheses with the first being that we believe that attackers will spend more time on average in databases with mp4 and mp3 files as opposed to just text files. Our second alternative hypothesis is that we believe that databases with mp4 or mp3 files will have more will have a higher use of commands and more activity. We believe this because we assume that mp4 and mp3 files could garner interest from the attackers more than the text files simply due to the rarity of those raw file types being in a database. Also if attackers were to actually analyze the mp4 and mp3 files it would require the to actually watch and/or listen to them, requiring more time than it would to simply skim a text file. Our null hypothesis is that there is no significant difference in time spent in databases with different types of data files, there will be no significant difference in the amount or type of commands used in the different databases. We expect to answer the question of "How does the type of data files in a database affect the time an attacker spends in the database, the amount of commands used, and whether or not they deploy malware?"

Experimental design

For our experiment we used four different container types, each with a different configuration, which is characterized by a different file type representing our honey. The first container configuration contains a directory named "mp4" which contains twenty copies of an mp4 file of fake security camera footage. The copyright free fake camera footage was obtained from Youtube. The second container configuration contains a directory named "mp3" with twenty copies of an mp3 file of fake UMD classroom lecture recordings. The fake lecture recording was obtained from an educational youtube channel and is copyright free. The third

container configuration contains text files which contain fake student names and passwords to their University of Maryland student accounts. The fake student names and passwords were generated by Chatgpt. The last container contains a directory named “no files” with no data in it and represents a control container. As for the overall structure of what we have built, all of our main scripts lay in the home directory. These scripts include the main script, MITM script, DIT firewall rules, and data collection script. We have a “data” directory within our home directory with a subdirectory called “logs” which contain all of our MITM logs. Our python scripts were written in pycharm and kept on a group member's local machine since the graphs they generated were unable to be generated on our Honeypot Host VM.

Our team recycled each honeypot every three minutes with our recycling script destroying all ip table rules and killing all forever processes using the <*p-kill*> command along with destroying all containers we had up. Our crontab/cron job is responsible for calling the recycling script, which deletes all iptable rules, forever processes and containers, followed by the main script, which creates a new set of honeypots, randomly assigning external ips to each of the four configurations. These scripts were called in order every three minutes by our crontab. Our main script creates a template container for each configuration with open ssh installed and the correct honey directory copied into the container. The template containers are never deleted and are left running so that every subsequent container can be made by simply copying the correct template configuration. We also call our scheduling script within the main script so that it would be called once and run forever. Towards the bottom of the main script we specify that “maxlogins” into the container is one so only one attacker can come into a container at a time. The main script also uses the <*shuff*> command to shuffle the ip addresses to assign to each container and the port numbers that should be assigned to each one. It is also able to install ssh

into each container. It is overall responsible for starting the containers and specifying where the MITM logs are going to be stored. Those logs are stored in a directory called data which has a subdirectory called logs which contains every MITM log generated by attackers coming into our containers.

The MITM script is responsible for starting up the MITM server which is the main way we are collecting data and figuring out when attackers enter and exit, along with the commands they run within the container. The MITM server receives parameters from the main script which are the container name, external ip, date, and port number of the container. It sets up ip table rules so that every attacker of any ip address is let into the honeypot without blocking anyone. It also has a <*sleep*> command before the forever processes, this was implemented because we were having an issue during deployment which was not letting our forever processes start correctly. This MITM script is able to also start an MITM server for each container separately because it has the ip addresses and container names being passed into it as parameters from the main method so we did not have to hardcode and specify container names or ip addresses.

We also created a Python script that would be able to generate box plot data for commands used, time spent, and amount of times <*wget*> and <*curl*> were used within the specific container. This helped us create visualizations for our data and see if there were any outliers within our files. Lastly we also created a crontab that is able to run multiple of our scripts at reboot to make sure we would still be able to collect data if our system reboots. As for our health logs, we are using Net Data to track the health of our honeypot as that website gives us better visuals to see data points such as disk space usage, RAM usage, network traffic, etc. All relevant scripts such as the Main script, Recycling Script, MITM script, etc are provided below in Appendix C.

Data Collection

We were collecting four data points which were the amount of time spent in each container and the amount of commands run within each container when the attacker enters the honeypot. All this data is stored within the MITM logs within the logs directory on our system, however it would be difficult to go through every log to brute force calculate this data. So we created a parsing script that goes through every MITM log file and <greps> the type of file and the time spent by that specific attacker, and we put that data into a single file called "timeSpentData" within our home directory. We did the same for commands used where we automatically parsed all MITM logs generated to figure out the number of commands run within each container, while also specifying the file type of that container. We recorded a total of 36,914 breaches into our honeypots.

For the most part, attackers who logged into the containers either tried to get into the honeypot but were not successful or they got into the container, ran two commands or less, and then exited the container. Initially, we thought it would be best to just create an MITM server and let it collect all the data and have separate files for each attacker. We did not realize that there would be so many MITM logs generated daily to the point where we couldn't go into each file and manually calculate and process the data on time spent and amount of commands used in the container. So the parsing script that we created reduced the amount of time we had to spend monitoring the data. And it was important to categorize our data with the file type it came from, for example, a data point within our time spent file would look like "2.34 mp4". So first we would have the time spent and then the data type of the file within that specific container the attacker was trying to get into. This made it easier for our Python script to make boxplots and it

also helped our other Python script which helped us perform a Kruskal Wallis test upon this data to see which data type was getting more activity in general. So by categorizing our data by file type and parsing it for specific data points we needed we were able to process the data efficiently.

We did not eliminate any data except for the data we collected when we entered our own containers to test whether MITM logs were being generated or not. This is because we did not have any significant outliers or points at which our honeypots stopped recycling. The data also does not have any preprocessing as we were mainly collecting data on time spent and the amount of commands used within the container.

Data Analysis

Our team ran the Kruskal Wallis tests for all of our data since we could not assume there was a normal distribution of underlying data between the groups. The test is useful for determining differences between multiple groups when tested by an independent variable. We used these tests to compare the time spent, total commands used, <*curl*> commands used, and <*wget*> commands used within all four of our honeypot configurations. The Kruskal Wallis test allowed us to see if the differences in values observed are in fact statistically significant. The p-value based on the Kruskal Wallis test for time spent in the container was 0.6306713291121411. For the configuration with mp4 files, the mean, max, and total seconds spent was 4.192, 156.723, and 38689.269 seconds respectively. For the configuration with mp3 files, the mean, max, and total seconds spent was 4.135, 142.744, and 37510.236 seconds respectively. For the configuration with text files, the mean, max, and total seconds spent was 4.127, 123.270, and 38050.655 seconds respectively. For the configuration with no files, the mean, max, and total seconds spent was 4.170, 137.210, and 39164.202 seconds respectively.

The Kruskal Wallis P value for total commands executed within the container was 0.6575186930923467. For the configuration with mp4 files, the mean, max, and total occurrences were 0, 6, and 9230 respectively. For the configuration with mp3 files, the mean, max, and total occurrences were 0, 5, and 9072 respectively. For the configuration with only text files, the mean, max, and total occurrences were 0, 5, and 9220 respectively. Lastly, for the configuration with no files, the mean, max, and total occurrences were 0, 6, and 9392 respectively. The Kruskal Wallis P value for <*curl*> commands was 0.04421112859555253. For the configuration with mp4 files, the mean, max, and occurrences were 0, 2, and 9230 respectively. For the honeypot with mp3 files, the mean, max, and occurrences were 0, 2, and 9072 respectively. For the configuration with text files, the mean, max, and total occurrences were 0, 2, and 9220 respectively. For configurations with no files, the mean, max, and total occurrences were 0, 2, and 9392 occurrences respectively. We did not have enough data to run any statistical test for our <*wget*> commands used data set. Our mp4 and no file configurations had zero instances of the <*wget*> commands, our mp3 configuration had 3 total occurrences with the max usage in one session being once, and our no files configuration had 2 total occurrences with the max usage in one session being once.

A p-value less than 0.05 indicates that differences in data observed between groups are statistically significant. The results from the Kruskal Wallis test for time spent show that it is not possible to reject the null hypothesis. The p-value produced as a result of running the Kruskal Wallis test was 0.6, which is larger than 0.05. This indicates that the data collected has no statistical difference, therefore meaning our null hypothesis is upheld and our alternate hypothesis must be rejected. The p-value for commands executed was 0.657, which is also more than 0.05, meaning we are unable to reject the null hypothesis for commands use, and we can

conclude that the type of honey in the container does not have a significant impact on the commands executed within the container. However, the p-value produced by the Kruskal Wallis test for <*curl*> commands used in the container was 0.04, which is less than 0.05. In this instance, we are able to reject the null hypothesis and conclude that the data type within a database does in fact lead to a statistical difference in the amount of malware deployed through the <*curl*> command. Of course, being that there was not enough data for the use of the <*wget*> command to be able to run a statistical analysis test, we are unable to draw any conclusions on the differences in data between groups and we fail to reject the null hypothesis.

Conclusion

The results indicate that there is no difference in time spent in a container and commands used across all four of the honeypots. The p-value given by the Kruskal-Wallis test fails to reject the null hypothesis, which states that there would not be any difference amongst all four containers. Our alternative hypothesis is that attackers would on average spend more time in databases with mp3 and mp4 files. We also expected to see a higher command usage in these containers as well. However, the results indicate that for time spent in containers as well as commands, there is actually no difference between all honeypot configurations. We are able to reject our null hypothesis when comparing uses of the <*curl*> commands within containers. Based on the results, we can conclude that the type of honey in the container, whether text files, mp3 files, or even no files, does not make a difference in the attackers' length of stay in the container or the commands used once they are in the container. Based on the data analysis for time spent in containers, the median time spent was consistently around 2.1 seconds, which

shows that the type of honey placed in the container does not impact the attacker's behavior that much. When comparing these variables, we fail to reject the null hypothesis.

However, when comparing <*curl*> commands used in the containers in an effort to analyze possible malware used, we can see that the analysis actually rejects the null hypothesis as the recorded p-value is 0.044, which is less than 0.05. However, when closely examining the data collected, the total occurrences of the <*curl*> command in containers with mp4 files was 9230, while occurrences in containers with no files were slightly greater at 9392 total occurrences. In addition, configurations with mp3 files had the lowest <*curl*> command usage, totaling at 9072. This also goes against our hypothesis, as we expected containers with mp4 files to have more occurrences of attacker activity. This was also surprising as we did not expect to record so many instances of attackers using the <*curl*> command over the <*wget*>.

Overall, the tests ran mainly indicated that there is no significant impact on the type of files placed in a honeypot configuration on the attacker's behavior. If we were able to continue work on this project, we would spend more time developing our scripts to gain more accurate information about an attacker. For example, we hope to gain more information about an attacker's whereabouts, as well as have better scripts to specifically compare the commands the attackers are using within the honeypots.

Appendix A

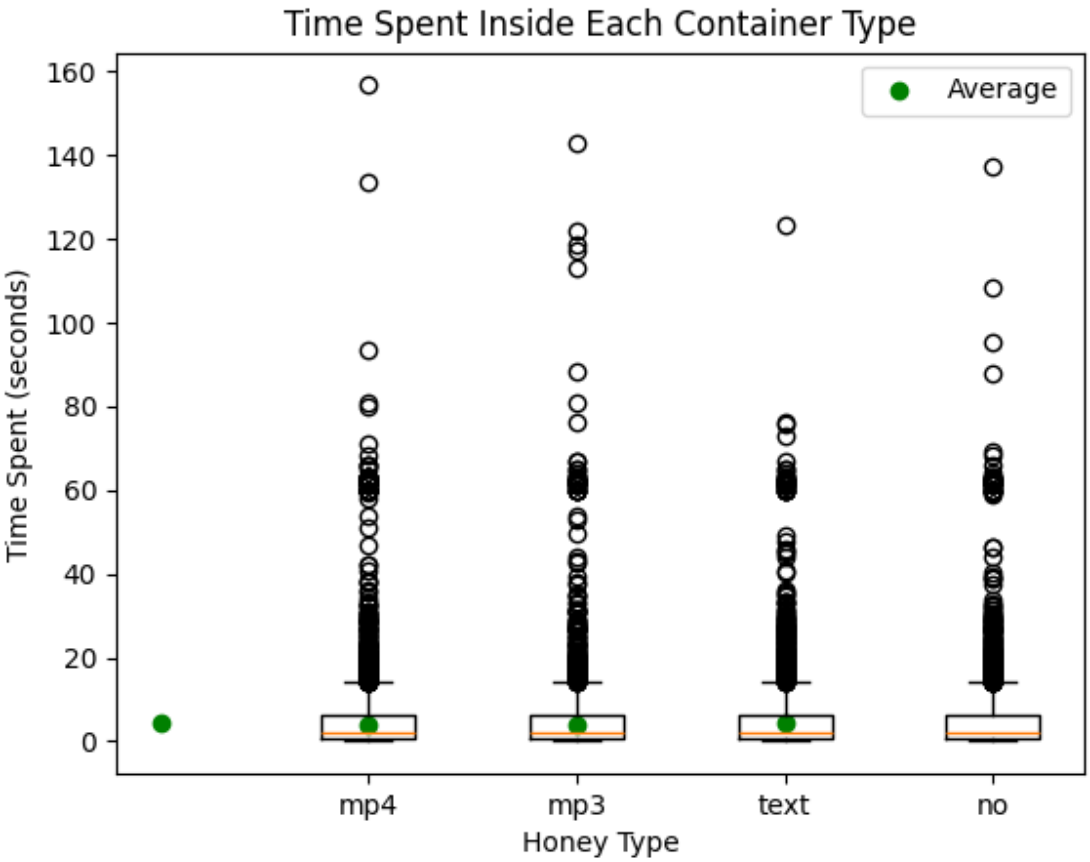
Completing this project has helped us learn numerous skills in relation to both technical skills as well as soft skills. Completing this project as a team allowed us to learn time management, as we often had to split up tasks into smaller tasks in order to meet the deadlines. The project also allowed us to have a good experience of how projects might work in the

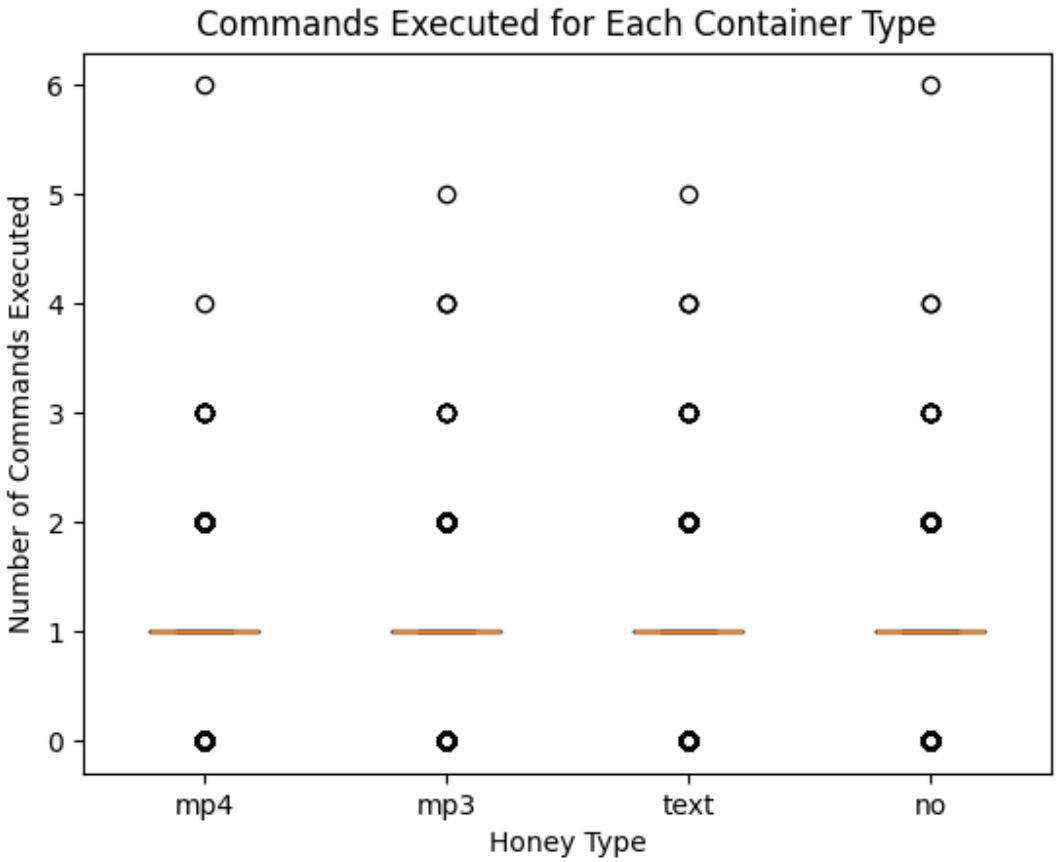
professional workspace. For example, delegating tasks and presenting progress before the rest of the teams through standups were beneficial in understanding how a professional space was run. Standups were also helpful to see how other teams were handling issues within their projects. From completing this project, we learned that the type of files or information placed in a database does not affect the attacker's behavior, especially the time spent in the container as well as the commands used by the attacker. This could be attributed to the fact that many of the attacks were made by bots. Our group was moderately surprised by the attacks and the attacker's behavior since the behavior drastically differed from what we originally expected the attackers to do. Another surprising point of data that we collected is that <*curl*> commands were actually used a lot more over <*wget*> commands, and we were

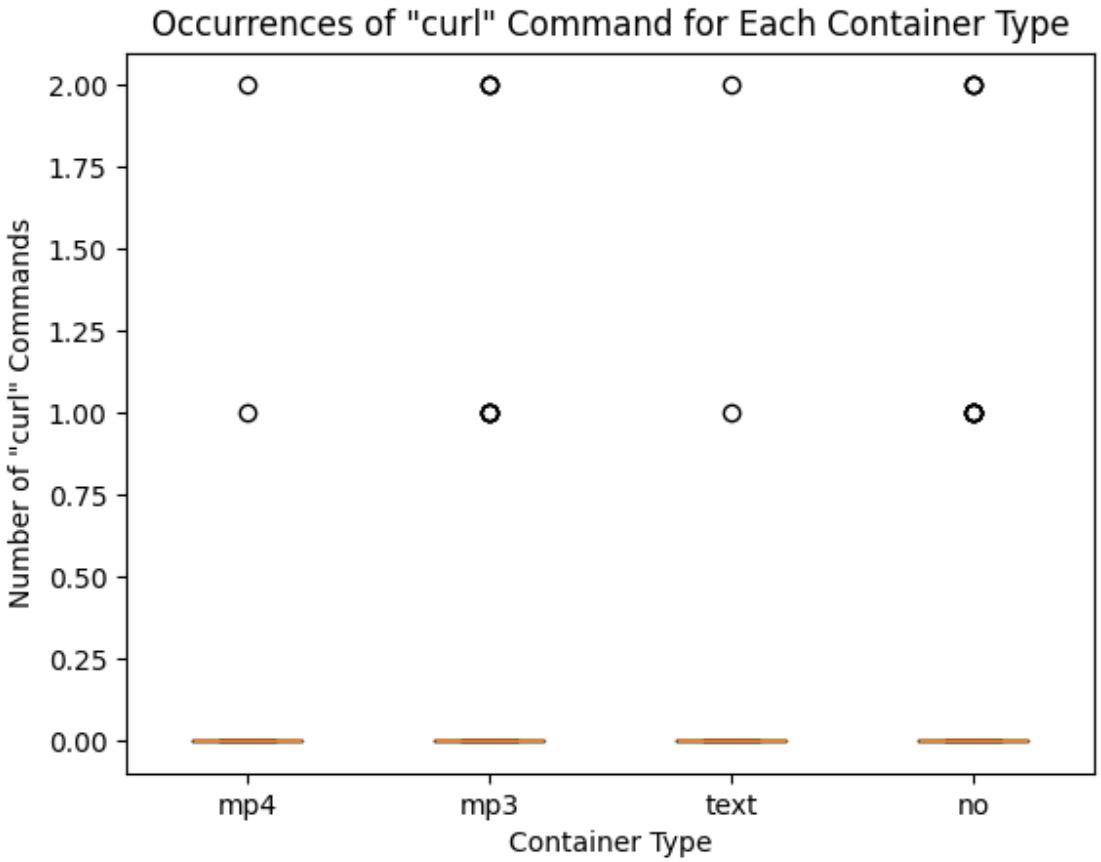
Our group encountered numerous issues during the deployment of our honeypots, which prevented us from deploying on time. Some of these issues included being unable to ssh into one of our containers, being unable to copy honey into one of the containers, as well as failing to block numerous attackers from entering the honeypot at once.

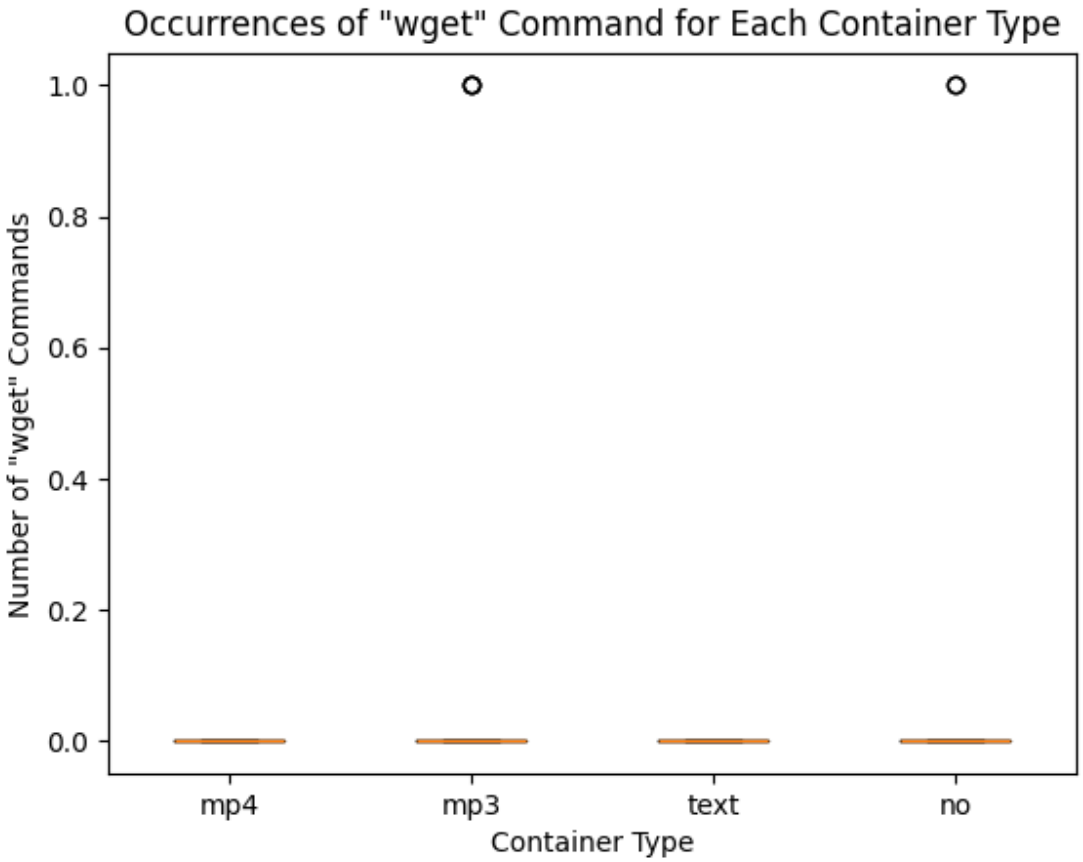
This class was structured well with good milestones and progress trackers in order to ensure teams were on track with the honeypot project. However, more time near the start of the semester to plan out the project and work on the design proposal would be useful. Furthermore, there also was a last minute rule added to the honeypot deployment checklist that threw our scheduling a little off. For the future, in advance communication of requirements for the honeypot would be appreciated.

Appendix B


Time Spent Inside Each Container Type
Average
Time Spent (seconds)
160
140
120
100
80
60
40
20
0
mp4
mp3
text
no
Honey Type


Commands Executed for Each Container Type
Number of Commands Executed
0
1
2
3
4
5
6
mp4
mp3
text
no
Honey Type


Occurrences of "curl" Command for Each Container Type
Number of "curl" Commands
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
mp4
mp3
text
no
Container Type



Appendix C

Main script:

```
#!/bin/bash
#sudo ./DIT_firewall_rules
ips=( "128.8.238.199" "128.8.238.31" "128.8.238.50" "128.8.238.180")
ips=( $(shuf -e "${ips[@]}"))
scenarios=( "mp4_files" "mp3_files" "text_files" "no_files" )
```

```
ports=( "5000" "5184" "5660" "5713" )
ports=( $(shuf -e "${ports[@]}"))

sudo sysctl -w net.ipv4.conf.all.route_localnet=1
sudo sysctl -w net.ipv4.ip_forward=1
#  creates a base template
# checks if the template container isn't created
# if not it gets created and everything relevant
# is created inside it
if  ! sudo lxc-ls | grep -q "template" ;
then
  sudo lxc-create -n template -t download -- -d ubuntu -r focal -a amd64
  sudo lxc-start -n template
  # Installs ssh in container
  sudo lxc-attach -n template -- bash -c "sudo apt-get update"
  sleep 10
  sudo lxc-attach -n template -- bash -c "sudo apt-get install openssh-server -y"
  sleep 10
  sudo lxc-attach -n template -- bash -c "sudo systemctl enable ssh --now"
  # allow root ssh
  #sudo lxc-attach -n template -- bash -c "echo 'PermitRootLogin yes' >> /etc/ssh/sshd_config"
  #sudo lxc-attach -n template -- bash -c "systemctl restart sshd"
  sudo lxc-stop -n template
  # Create configuration templates #
  LENGTH=4
  for ((j = 0 ; j < $LENGTH; j++));
  do
                scenario=${scenarios[$j]}
                n="template_${scenario}"
                # Creates a copy of the base tempate
                sudo lxc-copy -n template -N $n
                sleep 10;
    #Grab the correct directory based on the scenario and
    #copy it into the template
    correct_directory=$(echo "$scenario" | cut -d '_' -f1)
    sudo cp -r /home/student/$correct_directory /var/lib/lxc/$n/rootfs/home

                sudo lxc-stop -n $n
  done
  # Calls scheduling script within this if
  # statement so it will only be called once
  # and just run forever in theor
fi
LENGTH=4
# Creates the actual honeypots #
for ((j = 0 ; j < $LENGTH; j++));
do
  #shuffles the scenarios everytime so it's not guranteed which configuration it gets
  scenarios=( $(shuf -e "${scenarios[@]}") )
              ext_ip=${ips[$j]}
              scenario=${scenarios[$j]}
              template="template_${scenario}"
              n="${scenario}_${ext_ip}" # name of the honeypot being deployed
              mask=24
              date=$(date "+%F-%H-%M-%S")
  # Picks one of the 4 port numbers
  port_num=${ports[$j]}
```

```
 echo "_____"
 echo "this is your external ip for this honeypot: $ext_ip"
 echo "--------------------------"
            # Copie over the template and  starts it
            sudo lxc-copy -n $template -N $n
            sudo lxc-start -n $n

            sudo sleep 10

            container_ip=$(sudo lxc-info -n $n -iH)
            echo "container: $n, container_ip: $container_ip, external_ip: $ext_ip"

 # calls the MITM script to set up the server and the iptables rules
 /home/student/MITM_script "$n" "$ext_ip" "$date" "$port_num"
 #changes honeypot configs so only one attacker can come in
 echo "root       hard    maxsyslogins          1" >> /var/lib/lxc/$n/rootfs/etc/security/limits.conf
 echo "*       hard    maxsyslogins          1" >> /var/lib/lxc/$n/rootfs/etc/security/limits.conf
 # create the file with the port number in it
 sudo touch /home/student/data/ports/${ext_ip}_port.txt
 sudo echo "$port_num" | sudo tee /home/student/data/ports/${ext_ip}_port.txt
done
exit 0
```

MITM Script:

```
#!/bin/bash
if [ "$#" -ne 4 ]
then
echo "Usage: container name & external IP"
exit 1
fi
# alter this so that the port is parameter
containername=$1
externalip=$2
date=$3
port_num=$4
containerip=`sudo lxc-info -n $containername -iH`
#sudo ip netns exec ${containername} sysctl -w net.ipv4.ip_forward=1
#ext_if=eth0
#sudo ip netns exec ${containername} ip addr add ${externalip}/32 dev ${ext_if}
sudo sysctl -w net.ipv4.conf.all.route_localnet=1
sudo ip addr add $2/24 brd + dev eth1
sudo iptables --table nat --insert PREROUTING --source 0.0.0.0/0 --destination $externalip --jump DNAT --to-destination "$containerip"
#sudo iptables -t nat -A PREROUTING -p tcp --dport 22 -d ${externalip} -j DNAT --to-destination ${containerip}:22
sudo iptables --table nat --insert POSTROUTING --source "$containerip" --destination 0.0.0.0/0 --jump SNAT --to-source $externalip
sudo iptables --table nat --insert PREROUTING --source 0.0.0.0/0 --destination $externalip --protocol tcp --dport 22 --jump DNAT
--to-destination 10.0.3.1:$port_num
sleep 10
sudo forever -a -l /home/student/data/logs/{$date}_{$containername} start /home/student/MITM/mitm.js -n $containername -i
"$containerip" -p $port_num --auto-access --auto-access-fixed 1 --debug --mitm-ip 10.0.3.1
```

Recycling Script:

```
#!/bin/bash
sudo /home/student/DIT_firewall_rules
# Get a list of containers with the specified format
containers=$(sudo lxc-ls)
valid_prefixes=("mp3_files" "mp4_files" "no_files" "text_files")
echo "RECYCLE SCRIPT STARTED"
echo "**************************************************"
for container in $containers; do
  # Extract the container name prefix (e.g., mp3_files, mp4_files, etc.)
  container_prefix=$(echo "$container" | cut -d '_' -f1,2)
  # Check if the prefix is in the list of valid prefixes
  if [[ " ${valid_prefixes[*]} " == *" $container_prefix "* ]]; then
    # Do whatever you need to do with these containers
    echo "Found a matching container: $container"
    scenario=$(echo $container | cut -d '_' -f1,2)
    ext_ip=$(echo $container | cut -d '_' -f3)
    cont_ip=$(sudo lxc-info -n $container -iH)
    net_mask=24
    port=$(sudo cat /home/student/data/ports/${ext_ip}_port.txt)
    echo "Processing container: $container"
    # Delete the iptables rules
    sudo iptables -w --table nat --delete PREROUTING --source 0.0.0.0/0 --destination $ext_ip --jump DNAT --to-destination $cont_ip
    sudo iptables -w --table nat --delete POSTROUTING --source $cont_ip --destination 0.0.0.0/0 --jump SNAT --to-source $ext_ip
    sudo iptables -w --table nat --delete PREROUTING --source 0.0.0.0/0 --destination $ext_ip --protocol tcp --dport 22 --jump DNAT
--to-destination 10.0.3.1:$port
    sudo ip addr delete $ext_ip/$net_mask brd + dev
    # Destroy the container
    sudo lxc-stop -n $container --kill
    sudo lxc-destroy -n $container
    # Deletes file with port number in it
    sudo rm /home/student/data/ports/${ext_ip}_port.txt
    echo "y"
    echo "Finished processing container: $container"
    echo "--------------------------------------------------------"
  else
    echo "Not the desired container"
    echo "--------------------------------------------------------"
  fi
done
sudo pkill -f node
echo "All forever processes killed"
```

Total Commands Used Boxplot generator:

```
import re
import matplotlib.pyplot as plt
import pandas as pd


# Function to parse the log file and extract relevant information
def parse_log_file(file_path):
    data = {"mp4": {"commands": []},
            "mp3": {"commands": []},
            "text": {"commands": []},
            "no": {"commands": []}}

    with open(file_path, 'r') as file:
        for line in file:
            match_command = re.match(r'(\d+) (\w+)', line)
            if match_command:
                num_commands = int(match_command.group(1))
                data_type = match_command.group(2)
                data[data_type]["commands"].append(num_commands)

    return data


# Function to create a boxplot from the parsed data
def create_boxplot(parsed_data):
    labels = list(parsed_data.keys())
    data_values = [parsed_data[label]["commands"] for label in labels]

    # Calculate statistics using pandas describe function
```

```
    statistics = {label: pd.Series(commands).describe() for label, commands in
zip(labels, data_values)}


    for label, stat in statistics.items():
        print(f"Statistics for {label}:")
        print(f"  Max: {int(stat['max'])}")
        print(f"  75%: {int(stat['75%'])}")
        print(f"  Median: {int(stat['50%'])}")
        print(f"  25%: {int(stat['25%'])}")
        print(f"  Min: {int(stat['min'])}")
        print(f"  Mean: {int(stat['mean'])}")
        print()


    # Plot the boxplot
    fig, ax = plt.subplots()
    boxplot = ax.boxplot(data_values, labels=labels)
    plt.title('Commands Executed for Each Container Type')
    plt.xlabel('Honey Type')
    plt.ylabel('Number of Commands Executed')
    plt.show()


# Example usage
log_file_path = 'C:\\Users\\Michael\\Desktop\\HACS200\\commandsUsedData'
parsed_data = parse_log_file(log_file_path)
create_boxplot(parsed_data)
```

Total Commands Used Kruskal-Wallace test:

```
import re
```

```
from scipy.stats import kruskal


# Function to parse the log file and extract relevant information
def parse_log_file(file_path):
    data = {"mp4": [], "mp3": [], "text": [], "no": []}

    with open(file_path, 'r') as file:
        for line in file:
            match_data = re.match(r'(\d+) (\w+)', line)
            if match_data:
                value = int(match_data.group(1))
                data_type = match_data.group(2)
                data[data_type].append(value)

    return data


# Function to perform Kruskal-Wallis test
def perform_kruskal_wallis(data):
    result_statistic, result_pvalue = kruskal(data["mp4"], data["mp3"],
data["text"], data["no"])
    return result_statistic, result_pvalue


# Example usage
log_file_path = 'C:\\Users\\Michael\\Desktop\\HACS200\\commandsUsedData'
parsed_data = parse_log_file(log_file_path)


# Check if there is enough data to perform the test
if all(len(values) >= 3 for values in parsed_data.values()):
    kruskal_statistic, kruskal_pvalue = perform_kruskal_wallis(parsed_data)
```

```
    print("Kruskal-Wallis Test Result:")
    print(f"Statistic: {kruskal_statistic}")
    print(f"P-value: {kruskal_pvalue}")
else:
    print("Not enough data to perform the Kruskal-Wallis test.")
```

Time Spent Data Boxplot generator:

```
import re
import matplotlib.pyplot as plt
import pandas as pd



# Function to parse the log file and extract relevant information
def parse_log_file(file_path):
    data = {"mp4": {"time_spent": [], "total_lines": 0},
            "mp3": {"time_spent": [], "total_lines": 0},
            "text": {"time_spent": [], "total_lines": 0},
            "no": {"time_spent": [], "total_lines": 0}}

    with open(file_path, 'r') as file:
        for line in file:
            match_time = re.match(r'([0-9.]+) (\w+)', line)
            if match_time:
                time_spent = float(match_time.group(1))
                data_type = match_time.group(2)
                data[data_type]["time_spent"].append(time_spent)
            data[data_type]["total_lines"] += 1
```

```
    return data



# Function to create a boxplot from the parsed data
def create_boxplot(parsed_data):
    labels = list(parsed_data.keys())
    data_values = [parsed_data[label]["time_spent"] for label in labels]

    fig, ax = plt.subplots()

    # Plot the boxplot
    boxplot = ax.boxplot(data_values, labels=labels)

    # Plot the average line
    averages = [sum(times) / len(times) for times in data_values]
    ax.scatter(labels, averages, color='green', marker='o', label='Average')

    # Annotate the total number of lines for each type
    for label in labels:
        total_line = parsed_data[label]["total_lines"]
        ax.text(labels.index(label) + 1, total_line + 1, str(total_line),
ha='center', va='bottom', color='blue')

    plt.title('Time Spent Inside Each Container Type')
    plt.xlabel('Honey Type')
    plt.ylabel('Time Spent (seconds)')
    plt.legend()
```

```
    # Print total time spent, total zero time spent, and overall total time
spent for each data type
    for label in labels:
        times = parsed_data[label]["time_spent"]
        total_time_spent = sum(times)
        total_zero_time_spent = times.count(0)

        print(f"Statistics for {label}:")
        print(f"  Total seconds spent: {total_time_spent:.3f} seconds")
        print(f"  Total times with 0.000 seconds spent:
{total_zero_time_spent}")

        if times:  # Avoid division by zero for empty lists
            statistics = pd.Series(times).describe()
            print(f"  Min: {statistics['min']:.3f}")
            print(f"  25%: {statistics['25%']:.3f}")
            print(f"  Median: {statistics['50%']:.3f}")
            print(f"  75%: {statistics['75%']:.3f}")
            print(f"  Max: {statistics['max']:.3f}")
            print(f"  Mean: {statistics['mean']:.3f}")

        print()

    overall_total_time_spent = sum(sum(times) for times in data_values)
    print(f"Overall total time spent: {overall_total_time_spent:.3f} seconds")
    plt.show()

    # Flush the output
    import sys
```

```
    sys.stdout.flush()



# Example usage
log_file_path = 'C:\\Users\\Michael\\Desktop\\HACS200\\timeSpentData'
parsed_data = parse_log_file(log_file_path)
create_boxplot(parsed_data)
```

Time Spent Kruskal Wallace test:

```
import re
from scipy.stats import kruskal


# Function to parse the log file and extract relevant information
def parse_time_log(file_path):
    data = {"mp4": [], "mp3": [], "text": [], "no": []}

    with open(file_path, 'r') as file:
        for line in file:
            match_data = re.match(r'([0-9.]+) (\w+)', line)
            if match_data:
                time = float(match_data.group(1))
                data_type = match_data.group(2)
                data[data_type].append(time)

    return data


# Function to perform Kruskal-Wallis test
def perform_kruskal_wallis(data):
```

```
    result_statistic, result_pvalue = kruskal(data["mp4"], data["mp3"],
data["text"], data["no"])
    return result_statistic, result_pvalue


# Example usage
log_file_path = 'C:\\Users\\Michael\\Desktop\\HACS200\\timeSpentData'
parsed_data = parse_time_log(log_file_path)


# Check if there is enough data to perform the test
if all(len(values) >= 3 for values in parsed_data.values()):
    kruskal_statistic, kruskal_pvalue = perform_kruskal_wallis(parsed_data)
    print("Kruskal-Wallis Test Result:")
    print(f"Statistic: {kruskal_statistic}")
    print(f"P-value: {kruskal_pvalue}")
else:
    print("Not enough data to perform the Kruskal-Wallis test.")
```

Curl Commands Used Boxplot Generator:

```
import re
import matplotlib.pyplot as plt
import pandas as pd


# Function to parse the log file and extract relevant information
def parse_log_file(file_path):
    data = {"mp4": {"commands": []},
            "mp3": {"commands": []},
            "text": {"commands": []},
            "no": {"commands": []}}
```

```
    with open(file_path, 'r') as file:
        for line in file:
            match_command = re.match(r'(\d+) (\w+)', line)
            if match_command:
                num_commands = int(match_command.group(1))
                data_type = match_command.group(2)
                data[data_type]["commands"].append(num_commands)

    return data

# Function to create a boxplot from the parsed data
def create_boxplot(parsed_data, command_name):
    labels = list(parsed_data.keys())
    data_values = [parsed_data[label]["commands"] for label in labels]

    # Calculate statistics using pandas describe function
    statistics = {label: pd.Series(commands).describe() for label, commands in
zip(labels, data_values)}

    for label, stat in statistics.items():
        print(f"Statistics for {label}:")
        print(f"  Max: {int(stat['max'])}")
        print(f"  75%: {int(stat['75%'])}")
        print(f"  Median: {int(stat['50%'])}")
        print(f"  25%: {int(stat['25%'])}")
        print(f"  Min: {int(stat['min'])}")
        print(f"  Mean: {int(stat['mean'])}")
        print(f"  Total Occurrences: {int(stat['count'])}")
```

```
        print()


    # Plot the boxplot
    fig, ax = plt.subplots()
    boxplot = ax.boxplot(data_values, labels=labels)
    plt.title(f'Occurrences of "{command_name}" Command for Each Container
Type')
    plt.xlabel('Container Type')
    plt.ylabel(f'Number of "{command_name}" Commands')
    plt.show()


# Example usage for "curl" commands
log_file_path_curl = 'C:\\Users\\Michael\\Desktop\\HACS200\\curlCommandsData
Dec 2'
parsed_data_curl = parse_log_file(log_file_path_curl)
create_boxplot(parsed_data_curl, "curl")
```

Curl Commands Used Kruskal Wallis test:

```
import re
from scipy.stats import kruskal


# Function to parse the log file and extract relevant information
def parse_log_file(file_path):
    data = {"mp4": [], "mp3": [], "text": [], "no": []}


    with open(file_path, 'r') as file:
        for line in file:
            match_data = re.match(r'(\d+) (\w+)', line)
```

```
            if match_data:
                number = int(match_data.group(1))
                config_name = match_data.group(2)
                data[config_name].append(number)

    return data

# Function to perform Kruskal-Wallis test
def perform_kruskal_wallis(data):
    result_statistic, result_pvalue = kruskal(data["mp4"], data["mp3"],
data["text"], data["no"])
    return result_statistic, result_pvalue

# Example usage
log_file_path = 'C:\\Users\\Michael\\Desktop\\HACS200\\curlCommandsData Dec 2'
parsed_data = parse_log_file(log_file_path)

# Check if there is enough data to perform the test
if all(len(values) >= 3 for values in parsed_data.values()):
    kruskal_statistic, kruskal_pvalue = perform_kruskal_wallis(parsed_data)
    print("Kruskal-Wallis Test Result:")
    print(f"Statistic: {kruskal_statistic}")
    print(f"P-value: {kruskal_pvalue}")
else:
    print("Not enough data to perform the Kruskal-Wallis test.")
```

Wget Commands Used Boxplot generator:

```
import re
import matplotlib.pyplot as plt
import pandas as pd


# Function to parse the log file and extract relevant information
def parse_log_file(file_path):
    data = {"mp4": {"commands": [], "total": 0},
            "mp3": {"commands": [], "total": 0},
            "text": {"commands": [], "total": 0},
            "no": {"commands": [], "total": 0}}

    with open(file_path, 'r') as file:
        for line in file:
            match_command = re.match(r'(\d+) (\w+)', line)
            if match_command:
                num_commands = int(match_command.group(1))
                data_type = match_command.group(2)
                data[data_type]["commands"].append(num_commands)
                data[data_type]["total"] += num_commands

    return data


# Function to create a boxplot from the parsed data
def create_boxplot(parsed_data):
    labels = list(parsed_data.keys())
    data_values = [parsed_data[label]["commands"] for label in labels]

    # Calculate statistics using pandas describe function
```

```
    statistics = {label: pd.Series(commands).describe() for label, commands in
zip(labels, data_values)}


    for label, stat in statistics.items():
        print(f"Statistics for {label}:")
        print(f"  Max: {int(stat['max'])}")
        print(f"  75%: {int(stat['75%'])}")
        print(f"  Median: {int(stat['50%'])}")
        print(f"  25%: {int(stat['25%'])}")
        print(f"  Min: {int(stat['min'])}")
        print(f"  Mean: {int(stat['mean'])}")
        print(f"  Total Occurrences: {parsed_data[label]['total']}")
        print()


    # Plot the boxplot
    fig, ax = plt.subplots()
    boxplot = ax.boxplot(data_values, labels=labels)
    plt.title('Occurrences of "wget" Command for Each Container Type')
    plt.xlabel('Container Type')
    plt.ylabel('Number of "wget" Commands')
    plt.show()


# Example usage
log_file_path = 'C:\\Users\\Michael\\Desktop\\HACS200\\wgetCommandsData Dec 2'
parsed_data = parse_log_file(log_file_path)
create_boxplot(parsed_data)
```

Bibliography

Ramsbrock, D., Berthier, R., & Cukier, M. (2007). Profiling Attacker Behavior Following SSH Compromises. In 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07) (pp. 119-124). Edinburgh, UK. doi:10.1109/DSN.2007.76.

Nicomette, V., Kaâniche, M., Alata, E., & others. (2011). Set-up and deployment of a high-interaction honeypot: Experiment and lessons learned. Journal of Computer Virology, 7, 143-157. https://doi.org/10.1007/s11416-010-0144-2.

Timothy Barron and Nick Nikiforakis. 2017. Picky Attackers: Quantifying the Role of System Properties on Intruder Behavior. In Proceedings of the 33rd Annual Computer Security Applications Conference (ACSAC '17). Association for Computing Machinery, New York, NY, USA, 387–398. https://doi.org/10.1145/3134600.3134614

Aliyev, V. (2010). Using honeypots to study skill level of attackers based on the exploited vulnerabilities
in the network. https://odr.chalmers.se/items/1f28e9cd-a6bd-4bf5-a246-9f8286f02850/full