Do Devices' Computing Resources and Network Bandwidth Influence Attacker Behavior?

Group 2I: Kenneth Fadojutimi, Paul Makarevich, Romero Schwarz, Solomon Ucko & Eric Wang

Advanced Cybersecurity Experience for Students, Honors College, University of Maryland

HACS 200: Applied Cybersecurity Foundations II

Instructor: Dr. Michel Cukier

December 2, 2024

Executive Summary

The goal of our project was to see if having different computing and network resources on a honeypot affects attacker behavior. We varied computing and network resources across four factors: CPU time, allocated RAM, outbound latency, and network bandwidth, for a total of sixteen configurations. We hypothesized that increasing the allotted computing and networking resources would cause attackers to spend more time, run more commands, and more frequently download malware on our honeypot. We believed that machines with above-average access to computing and network resources may be of higher value target to attackers–the added computational power could be useful in a distributed BitCoin mining operation, for instance.

We collected data on the types of malware that attackers deployed (if any), along with their time spent in the honeypot and number of commands run. Additionally, we tracked the amount of computing resources that the attackers used while on the honeypot, which could be an indicator of attacks that hijack computing and network resources.

Ultimately, we found no connection between the system's available resources and attacker behavior. Across all configurations, we observed no difference in the time that attackers spent in honeypots, nor the number of commands they run. Though we expected that attackers might deploy malware frequently, we found that very few attackers downloaded anything onto the honeypot. Across 28,206 attacks, we had only 20 downloaded files. Furthermore, none of the attackers downloaded any malware at all - all files that were downloaded were not malicious.

Background Research

Various studies of cyberattacker behavior have found that computing and network resources can make internet-connected devices valuable targets for cyberattackers. For instance, computing resources on compromised systems can be hijacked to perform computations for mining cryptocurrency in operations called “cryptojackings”. Additionally, a fast and reliable internet connection on a device makes it a good addition to a botnet. The findings of Chamotra et al. (2016) are consistent with these statements, finding that attackers frequently target IoT devices for the creation of a botnet. IoT devices are specifically targeted for two main reasons. Firstly, they are generally vulnerable; conventional antivirus/antimalware programs don’t apply for IoT devices, and many IoT users lack the experience to change security configurations or set up firewalls. Additionally, IoT devices have broadband internet access, making them reliable nodes in a botnet. Chamotra et al. (2016) deployed various honeypots, and found that in most instances attacks downloaded malware after successfully gaining access to the honeypot. Multiple malware samples were found to be botnet software; one was a self-propagating virus targeting Linux machines.

Huang et al. (2014) examines how attackers can hijack machines to perform distributed cryptocurrency mining operations. By installing a daemon on a machine to use unused computing resources, especially while a machine is idle, attackers can create a network of machines mining bitcoins, with low risk of detection. Cryptocurrency that is generated can be effectively laundered through a series of proxies, or can be sent directly to an attacker's account. Cryptojacking operations like these are not uncommon; mining malware was installed on industrial systems throughout Europe in 2018 (Newman, 2018). As such, given that cryptojacking attacks can be incredibly profitable, and that they already have prevalence, it is not unreasonable to expect that an excess of computing resources on a honeypot might draw the attention of cryptojackers. Though, even if we believe that cryptojackers may be more likely to target machines with more resources, there are many other possibilities — a different study found a wide variety of types of malware deployed on honeypots, even though they were targeting just self-propagating viruses (Zhuge et al., 2007)

While many studies can find that computing and network resources may be valuable targets for cyberattackers, few examine the relationship between these resources and attacker behavior, that is, finding if the presence of plentiful computing resources causes attackers to behave differently. Understanding the change in attacker behavior (if any) could allow organizations to prioritize protecting systems which attackers are more likely to attack, and in doing so better their security.

Experimental Design Changes

Our experimental design has deviated little from our original. We initially wanted to vary the available RAM, number of CPU cores, connection bandwidth, and latency on our honeypots, which we did, however, we had to make adjustments to accommodate the resources that were given to us. We were initially uncertain of what machines the Department of IT would let us host our honeypots on, and assumed that we might be able to get more resources. Unfortunately, this was not the case, and we had to limit the 'high' resource configurations. We were able to get more dedicated CPU cores (8) and slightly more RAM (24 gigabytes). While this was an improvement, we believe that this lack of resources may have influenced our experiment — to a cryptojacker, a machine with two CPU cores is likely only marginally more valuable than a machine with one, considering that most modern systems need at least 4 to run applications (Horowitz, 2024).

We did encounter some issues implementing our initial design, but we were ultimately able to resolve them. Firstly, limiting CPU usage can be done in a variety of ways. Our first method involved restricting any container to using one or two dedicated cores, for instance, container 1 could only use cores 1 and 2, and container 2 only 3 and 4. That way, a high processing load on one container would not impact the others. We later realized that since we only have 8 cores, if all containers had a high load, it would impact the processing power of the host, which may cause issues with the honeypot system overall. Ultimately, we decided to throttle CPU availability by CPU time, rather than cores. By restricting how many milliseconds per second any container could use a CPU for, we can spread the processing load across all cores and prevent the honeypot host from encountering any issues.

Our latency configuration also proved to be an issue. At first, we assumed we could apply a latency to both inbound and outbound packets, however, we quickly realized this was not possible. It is impossible to apply a latency to inbound packets as that would require either rerouting them through a proxy or the sender delaying them, which we could not control. We could not create a proxy because of our limited resources.

One thing that we did change is the amount of time that an attacker can idle for in the container before being disconnected, which was 5 minutes. However, very few attackers ever reached this threshold.

Research Question and Hypothesis

For this research, we want to find the effect of varying computing resources, bandwidth, and outbound connection latency on attacker behavior. Specifically, we want to vary Internet bandwidth, latency, CPU power, and RAM, and we want to find out if attackers are more interested in machines with higher computing resources, higher bandwidth, and lower latency. We define attackers as any person trying to access a virtual machine that is not within their jurisdiction. We define attacker behavior or interaction as the amount of time an attacker spends on a machine and the number of commands an attacker executes on a machine. Our independent variables for this experiment are the download and upload speeds of the Internet connection on the virtual machine, the relative CPU power assigned to the machine, the amount of RAM assigned to the machine, and the amount of additional latency added to outbound packets on the machine. Our dependent variables are the time an attacker spends on the machine, the number of commands an attacker executes, and the type of malware deployed (if any). We have four alternate hypotheses, and one null:

H_{a1}: Increasing the bandwidth on the network of a virtual machine will increase the amount of time the attacker spends on the system.

H_{a2}: Increasing the RAM on a virtual machine will increase the amount of time the attacker spends on the system.

H_{a3}: Increasing the CPU power on a virtual machine will increase the amount of time the attacker spends on the system.

H_{a4}: Adding latency to outbound connections on a virtual machine will decrease the amount of time the attacker spends on the system.

H_{a5}: Increasing the bandwidth on the network of a virtual machine will increase the number of commands an attacker runs on the system.

H_{a6}: Increasing the RAM on a virtual machine will increase the number of commands an attacker runs on the system.

H_{a7}: Increasing the CPU power on a virtual machine will increase the number of commands an attacker runs on the system.

H_{a8}: Adding latency to outbound connections on a virtual machine will decrease the number of commands an attacker runs on the system.

H_0: Changes to RAM, CPU power, outbound latency, and bandwidth on a virtual machine have no effect on attacker interaction.

Experimental Design

Since we were given 5 public IP addresses to host our honeypots on, we set up a script to create one honeypot for each IP address. The script creates an Ubuntu LXC container from a snapshot, limits the resources available to the container (as specified later), adds IP routing rules so that attackers connecting to the IP through SSH are routed to the container through a man-in-the-middle (MITM) server, waits for an attacker, starts a timer when the attacker is on the system, kicks the attacker off the system after 5 minutes of idle time or 20 minutes of connection time (if the attacker hasn't already left), extracts all relevant data, destroys the honeypot, deletes the IP routing rules, until finally restarting itself immediately after. When the honeypots are started, each container is given randomized configurations for four computing or network resources, with high and low configurations for each. They are: CPU time (10% vs 20%), available memory (1 GB or 4 GB), network bandwidth (256 Kbit/sec or 2048 Kbit/sec), and additional outbound latency (none or 1 second). In total, this forms sixteen possible configurations. Configurations are randomized and each configuration is equally likely and can be repeated.

When an attacker connects to the honeypot, their traffic is first routed through the ACES MITM server, and then to the container. The ACES MITM server has detailed logs on the actions of the attacker inside the honeypot, allowing us to analyze what commands they execute, when they enter and exit the honeypot, what username or password they use, and what IP they use to connect, all of which we collect for analysis.

While an attacker is inside the honeypot, they may download files. In order to capture these files for analysis, we have poisoned the curl, wget, and scp commands to run strace on the original commands, capturing hexdumps of reads from stdin and writes to any file descriptor, into a log file that we can subsequently parse the file contents out of using a Python script. This method allows us to capture the file contents without affecting the network traffic at all, and with little-to-no effect on the behavior the attacker sees, in order to avoid the attacker noticing that our system is a honeypot. (See "Curl and Wget Poisoner" in Appendix C for the full bash script, which embeds the Python script.)

Additionally, in order to prevent multiple attackers from accessing the honeypot concurrently, an IP table rule is set up to block connections from any IP except for that of the attacker currently in the honeypot. This rule is removed when the attacker disconnects or is disconnected.

It should be noted that before opening our IPs to the public, we used a set of firewall rules provided by the Department of IT at the University of Maryland.

While an attacker is in the honeypot, they are under multiple time limitations. Firstly, the maximum time any attacker may spend in a container, regardless of idle time, is 20 minutes, after which the connection is severed and the container recycled. 20 minutes is a considerable time, however, we set this limit with consideration for large files that attackers might download, such as cryptocurrency miners. If a large file is downloaded on one of the low-bandwidth configurations, it would take a very long time and is likely to be cut short if the attacker is disconnected too soon. In addition to this, an attacker can spend no more than 5 minutes idle, as defined by the time since the last executed command. Idle time is not considered when measuring the time an attacker spends in the honeypot — we measure from the time the attacker enters the honeypot to the last command. Note that the 20 minute timer starts when the attacker enters the honeypot; honeypots waiting for attackers to connect will not time out and recycle.

Finally, when the attacker either closes the SSH connection or reaches a time limit, the container is recycled to prepare for the next attack. Before recycling, however, multiple scripts run to get the data from the honeypot. Firstly, the MITM server is closed and the log file saved to our data directory. Secondly, the files (if any) that were downloaded are tarred and zipped together and uploaded to VirusTotal, which gives us access to a summary of potentially malicious files in the archive. After uploading, we retrieve a base64 token and a hash which allows us to access the report at any time. This token is stored for later analysis. Our CSV file containing the resource usage data is run while the honeypot is active, so we do not need to copy anything over in that sense. After all of the relevant data is extracted, the container is shut down and deleted, the MITM server is stopped, and the IP routing rules relevant to the IP address are removed. We then restart the same process for each IP from the moment that the next Ubuntu LXC container is created from a snapshot.

Data Collection

We collected the amount of time the attacker spent in the honeypot, the commands run, the number of commands run, the username, password, and attacker IP used to login, the outbound connection usage, the inbound connection usage, the memory usage, the CPU usage, and a VirusTotal API summary of all downloaded files through the curl, wget, and scp commands.

We collected the time data (from the moment of attacker login to logout in milliseconds), username, password, and attacker IP using our provided HACS MITM server, which automatically logs the login and logout information of attackers. We collected the CPU, connection, and memory statistics using the lxc-info command every 5 seconds that the attacker was in the honeypot. For the downloaded files, we used poisoned curl, wget, and scp commands to intercept and duplicate downloaded files in another location without disrupting the download. When an attacker left the honeypot, we archived all files downloaded and uploaded them to VirusTotal, an online malware analyzer which also works with archives. After this, we saved the hash of the files submitted, so that we could access the report on the scan at any time in the future. While it would be preferable to keep the files for further analysis, we decided it would be best to discard the sample, as the files could be large and we could run out of storage — we are running five containers, which takes up a lot of storage space, leaving us with little room for any malware files, especially on a system with only 32 GB of storage.

As for data preprocessing, we had to do quite a bit. We used regular expressions to find all our username, password, IP, command, and time data in our MITM log files. We split lines with commands on the semicolon character, as the semicolon character splits commands in Unix systems, and counted every subdivision as its own command.

We excluded data points in only one case: when the attacker disconnected before authentication finished. We excluded a total of 7,675 out of 28,206 data points for this reason. This happened frequently on our high-latency honeypots, as authentication generally took more than three seconds. In many cases, attackers would disconnect before they actually entered the honeypot, making the data for that connection invalid. If we did not remove this data, the high-latency honeypot data would have larger clustering around three seconds, and would be incorrect, as we only want to measure when an attacker makes it inside the honeypot.

We grouped the resultant data into three categories and three separate CSV files. One CSV file had the username, password, time, attacker IP, and attacker IP data. Another CSV file had each command, the first word in the command, and the order number of the command (relative to others during the same attack). Finally, our last one had the CPU usage, RAM usage, upload connection usage, and download connection usage, as well as a timestamp for which period the snapshot of the data was taken. Each entry in the CSV files was labelled by the configuration number, the log number, and the honeypot IP, so that we could use joins to link data together if need be, as well as perform relative analysis.

All log files and hashes were kept on the system itself, and we have a master preprocessing file which creates all relevant CSV files from the logs. In this manner, we could easily change our preprocessing if we find that we need some other data, or that one of our methods needs to be tweaked. We created an option for a backup to a Google Drive, as well as an Azure MySQL server if we ran out of storage, but we never had to use it.

Data Analysis

Firstly, we will note that no attackers downloaded malicious files or malware, as defined by VirusTotal API. Next, note that our usage (CPU, RAM, bandwidth) statistics were gathered every 5 seconds, but most attacks were under 5 seconds, so our usage data is lacking. Therefore, we decided that our analysis would be strictly based on the amount of time the attacker spent in the honeypot, as well as the number of commands that the attacker ran. We see in the data that the number of commands and the amount of time each attacker spent in the honeypot are different distributions, if we split the data based on the latency configurations. Of course, we will quantify such assertions, but, to see them visually, consult Appendix A, Figures 1 and 2. Figure 3 displays each configuration side by side.

Secondly, given the drastic difference in distribution amongst configurations with and without latency, we must split our data by latency to prevent the effect of the latency from obscuring the effect of any other variables. It is visible that our data is not normally distributed; to confirm this, we used a Shapiro-Wilk test to see if the data was normal. For 1 second of additional latency, our p-value was $1.474*10^{-98}$. For no additional latency, our p-value was $3.652*10^{-118}$. According to the test, these p-values mean that there would be less than a $1.474*10^{-96}$ percent chance that if the data were normal, we would get the distributions we have. Thus, we conclude that our data is not normally distributed for both 1 second of additional latency, and no additional latency. As our data is not normally distributed, we cannot run any ANOVA tests, as an ANOVA test assumes normally distributed data; therefore, we must use the Kruskal-Wallis test, a non-parametric equivalent. Since our command data is counting data, which is discrete, the best test for us to use is also the Kruskal-Wallis test, as it works for discrete data as well as continuous data.

We performed Kruskal-Wallis tests on the CPU, memory, and bandwidth factors, on 1 second additional latency and no additional latency. The results showed that only the altered CPU time has an effect on attack time ($p < 0.05$) for both latency configuration types — all other differences in times were not statistically significant. However, for the number of commands run it is the *interaction* of altered CPU time and high latency that affect attacker behavior, as the difference based on CPU time is not statistically significant for no-latency configurations. See Appendix A, Tables 1 and 2 for the results of the Kruskal-Wallis test on both time spent in the honeypot and the number of commands run.

We hypothesize that the differences observed on the configurations with different allowed CPU times are due to how CPU time limits can stagger program execution. If a program has unlimited CPU time, then it may use as much time as the system scheduler permits before finishing. However, if it is limited to 20ms per 100ms of CPU time, then its execution is effectively staggered, that is, if it needs the full 20ms of CPU time every time, it will have to 'wait' at least 80ms in between. For instance, suppose an attacker runs a program that takes exactly 40ms of CPU time to run, and is limited to 20ms of CPU time per 100ms. The real execution time will then be between 120 and 200 ms — it will run for 20ms in the first tenth of a second and 20ms somewhere within the second. However, if they are limited to only 10ms of CPU time, the real execution time becomes 310 to 400 ms. In this way, by limiting the CPU time allotted to the honeypot, attackers stay in the container longer, as it takes longer to execute their commands.

Only in one case did the bandwidth change cause any significant change — when there was no latency, the bandwidth affected the time attackers spent in the honeypot. One might assume that SSH traffic uses a minimal amount of bandwidth, as it is only sending ASCII text back and forth, but it is more than one might expect. Firstly, it is sent over TCP, which has a high overhead, and requires multiple handshakes to confirm a packet has been received. Additionally, SSH also requires a fair amount of data to be sent outside of what the user is typing and what they are receiving. SSH must perform key exchanges multiple times throughout the session, if it is long enough, along with sending keep alive packets. As such, the 256kbit traffic limit is a constraint for attackers, as all traffic will take longer because it is sent out at a slower rate. We can even somewhat quantify this difference, as on no latency configurations, 183 attackers left before connecting to the honeypot on 256 kbit bandwidth configurations, while 0 attackers left before connecting to the honeypot on 2048 kbit bandwidth configurations.

However, it should be noted that this difference due to bandwidth for time spent in the honeypot was not observed when there was a latency applied ($p = 0.62$). Here, it is likely that the effect of the high latency overshadowed the effect of the low bandwidth.

Conclusion

We interpret our p-value by saying that when our p-value is less than 0.05, it means that there is less than 5% chance that our data occurred given in the distribution it did given that our null hypothesis were true, and we also accept 0.05 as our threshold for statistical significance. Our findings show that we can reject the null hypothesis but that we only have support for the alternative hypothesis that having increased outbound latency would decrease attacker interaction (as we found a difference relating to commands). However, we can also reject our null hypothesis in relation to the time data for CPU time, as there was a significant difference ($p<0.05$ by Table 1) for both latency configurations. Nevertheless, we hypothesized that having increased CPU time would increase the amount of time spent on the honeypot, but in fact it actually decreased, by a factor of about 25%. We think that this may occur because attackers are more interested in waiting for an output of a command than actually using the resources on the machine. We also know that latency is clearly a factor that affects how long the attacker is in the honeypot ($p<0.05$ by Table 3), but we found that the median time for 1s latency configurations is actually marginally higher than that of no latency, so we cannot support our alternative hypothesis in this regard. However, for commands, we can say that having a 1s latency did decrease attacker interaction, as we have a median of 1 command for 1s latency configurations, and a median of 3 commands for no latency configurations, along with a staggering 0.0 p-value, which means that it would be practically impossible for the data to occur in such a distribution given that the latency caused no difference in the distribution. We also include an interpretation for the significance for bandwidth for time earlier in our Data Analysis section. Across our configurations, and accounting for the guaranteed difference between high and no latency configurations, the time attackers spent in a honeypot and the number of commands they executed did not vary significantly based on available RAM in any test.

However, it should be noted that the difference in our computing resources across configurations is marginal. For an attacker looking to execute a program that would incur a heavy computational load (e.g. running a cryptocurrency mining script), having a single extra core or three extra gigabytes of RAM is unlikely to make much of a difference. Further experimentation should investigate changes in attacker behavior depending on more dramatic differences of computational resources, though this would require multiple powerful computers to be available. Another possible addition to the experiment would be a GPU, as GPUs are valuable for parallel processing computations (and by extension cryptocurrency mining), which could make them valuable targets for attackers.

Appendix A: Reflections

Lessons learned

After completing our research, we learned much about attacker behavior and how computing resources are handled in a Linux environment. We learned that most attacks are purely exploratory and automated–very few actually attempted to take anything from the system or download any files. We had expected that attackers would regularly attempt to download malware, however, this was not the case, rendering our initial experiment plan untenable. Finally, in order to execute our experiment, we had to become familiar with resource handling in Linux, including setting CPU time limits, understanding multicore processing and swap space, IP routing, and applying network rules, especially for latency.

Were you surprised by the attacks? The attackers' behavior?

Yes—we expected more interactive attacks to occur, even though we knew a majority of the attackers were going to be automated–noninteractive and exploratory. We also expected more malware, but only received a handful of files, none of which were malicious. However, we were not surprised by some aspects of attacker behavior, especially how attackers reacted to high latency times.

Pitfalls and failures encountered in the experiment

Some of our initial plans fell through. We had originally hoped to have a GPU made available so that we could experiment with it, but that wasn't possible. We also hoped to have more CPU cores and RAM available, but this also wasn't possible. Additionally, we had some trouble with data collection early in our experiment; some data that we collected was completely invalid due to issues with our own measurement methods which we did not detect until after deploying. This reduced the amount of data we were able to collect.

Feedback on the project (negative feedback is welcomed and does not negatively impact your grade)

It would have helped us a lot if it were made clearer how much resources we would be getting from the Division of IT, even if just an estimate. If we knew that we wouldn't be able to get what we initially planned for, we would have been able to switch topics and still stay on track for deployment. In particular, it would've been helpful to have more CPU cores available; for example, with 16 cores, we could have run 5 honeypots with 1 or 3 cores each and 1 core left over for the host, thereby running more honeypots, varying one of the independent variables over a wider range of values, and avoiding resource conflicts between honeypots and the host.

We appreciate the time that the instructors and TAs spent to make sure our project was on the right track.

Appendix B: Tables and Figures

Table 1. Kruskal Wallis Test for Time Spent in Container, Split by Latency

Latency	**CPU**	**RAM**	**Bandwidth**
0s	1.610e-81	0.07276	0.03953
1s	8.627e-84	0.1103	0.6154

Table 2. Kruskal Wallis Test for Commands Run, Split by Latency

Latency	**CPU**	**RAM**	**Bandwidth**
0s	0.6213	0.4732	0.8160
1s	2.606e-06	0.6687	0.6452

Table 3. Kruskal Wallis Test on Latency for Commands Run and Time Spent

Test Type	**p-value**
Time	1.489e-29
Commands	0.0 (extremely small)

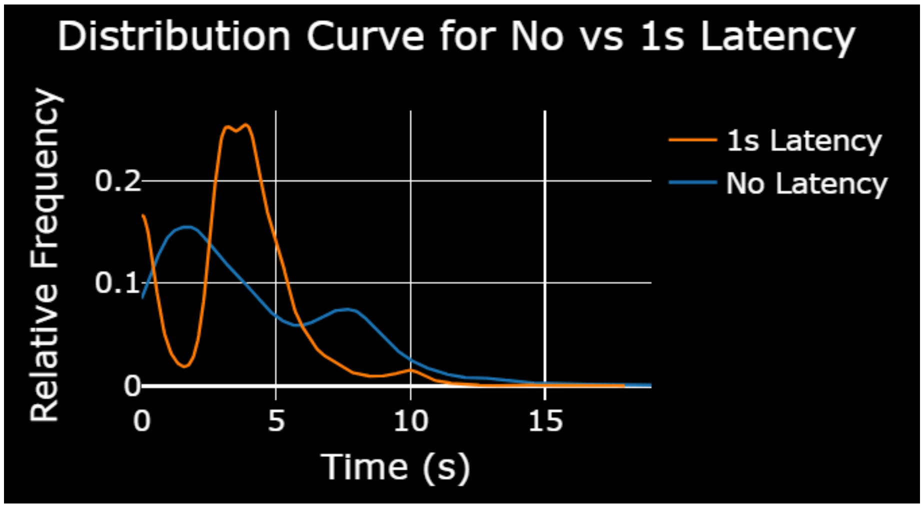


Figure 1. Distribution of Time Spent in Honeypot

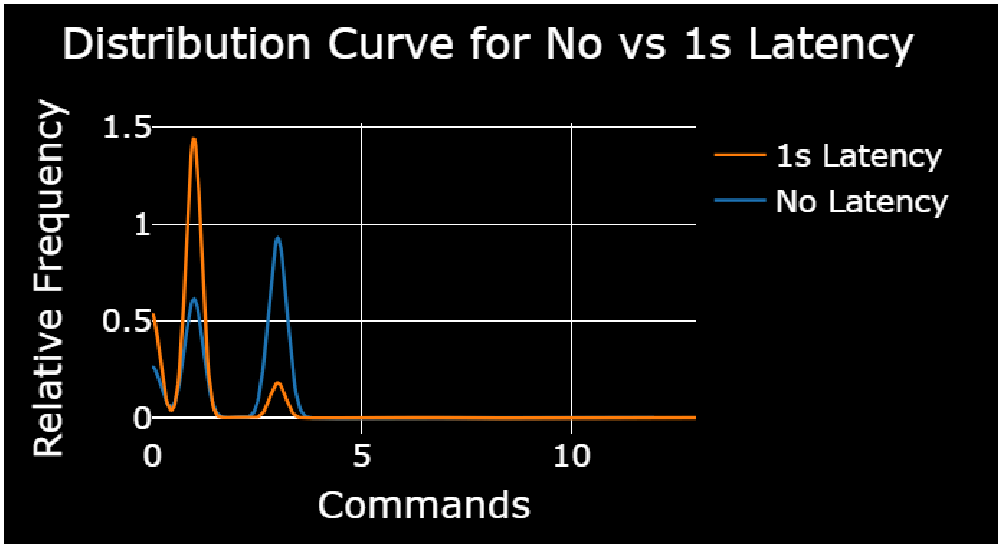


Figure 2. Distribution of Commands Run in Honeypot

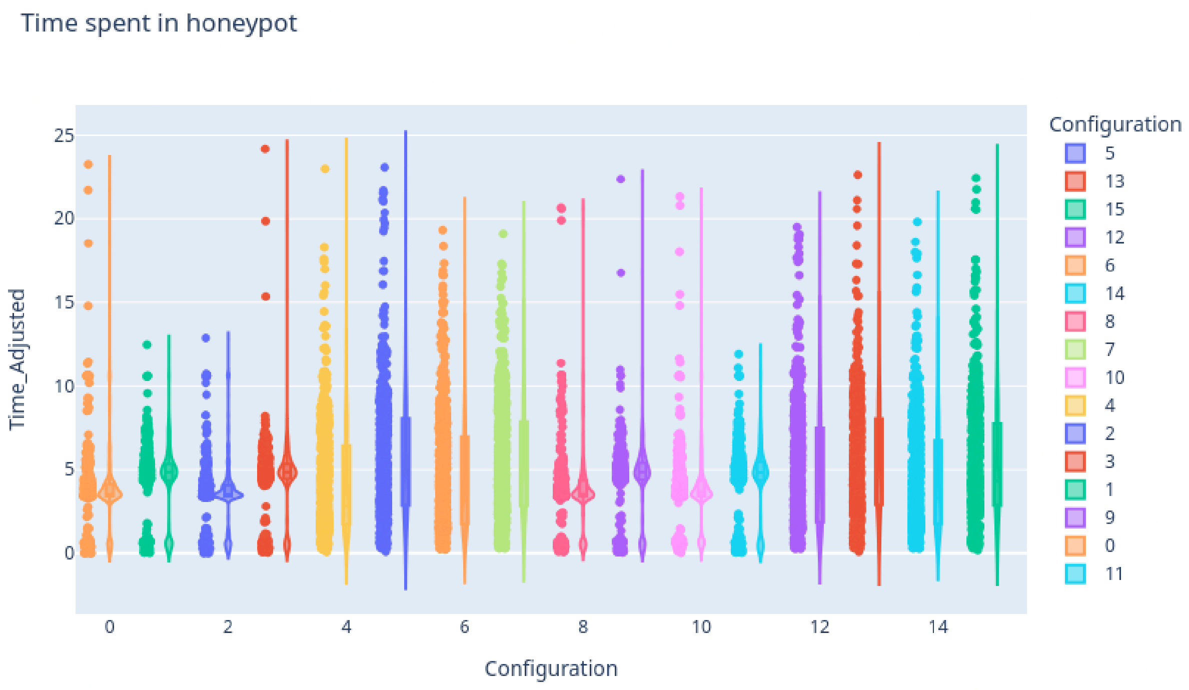


Figure 3. Time Spent in Honeypot by Configuration

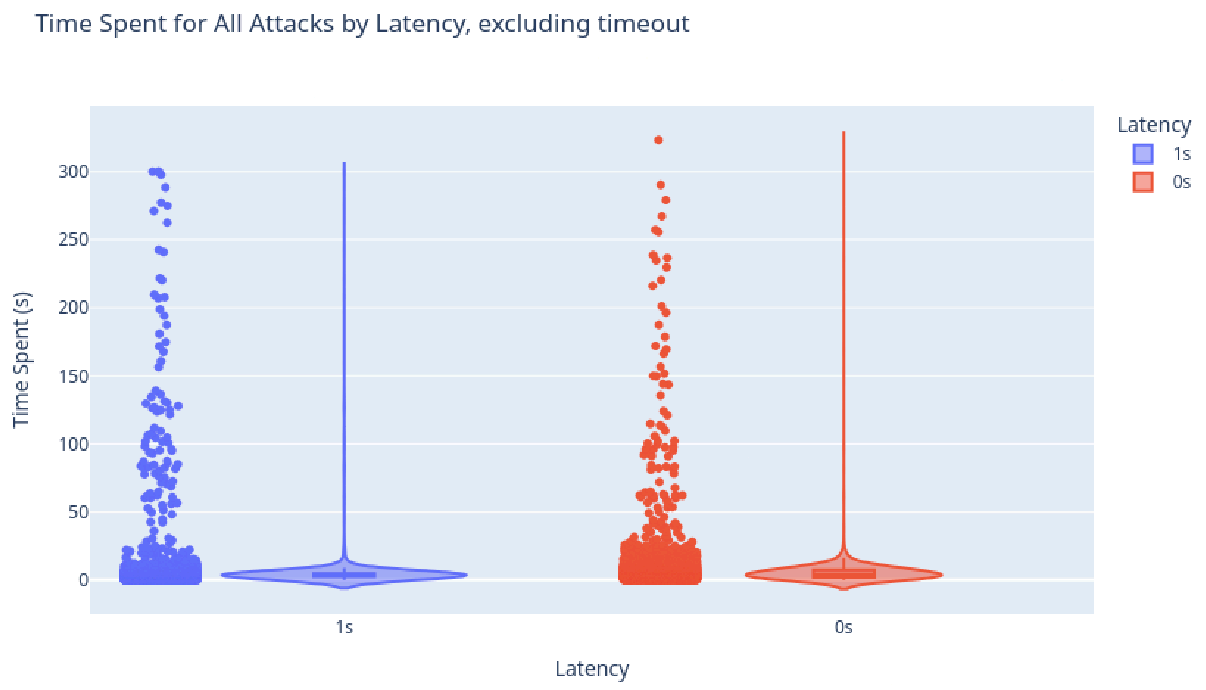
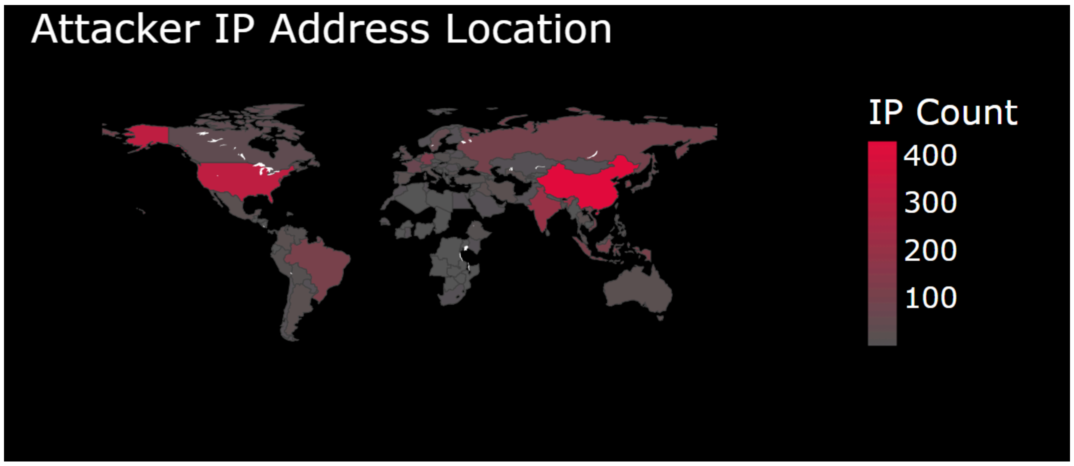


Figure 4. Difference between No Latency and 1s Latency Configurations

Appendix C: Scripts and Interesting Information

Heat Map of Global Attacker IP Locations



Recycling Script

```
#!/bin/bash
if [ $# -lt 4 ]
then
        /bin/echo "Incorrect number of arguments. Program takes in four arguments: the target container
name, the external IP to listen to, the number of the honeypot (1-4), and the port number for the MITM
server (diff for each honeypot)"
        exit 1
fi

sudo /bin/lxc-stop $1
sudo /bin/lxc-destroy $1
sudo /bin/lxc-snapshot -n HONEYPOT_SNAPSHOT -r snap0 -N $1
sudo /bin/lxc-start $1
sleep 3
sudo /bin/lxc-attach -n $1 -- bash -c "sudo systemctl restart sshd"

while [[ $(sudo /bin/lxc-info -iH $1) == "" ]]
do
       /bin/sleep 1
done
containerip=$(sudo /bin/lxc-info $1 -iH)
/bin/sleep 1
while [ $(sudo /sbin/iptables --table nat -C POSTROUTING --source $containerip --destination 0.0.0.0/0
--jump SNAT --to-source $2; /bin/echo $?) != "0" ]
do
```

```
        sudo /sbin/iptables --table nat --insert POSTROUTING --source $containerip --destination 0.0.0.0/0
--jump SNAT --to-source $2
        /bin/sleep 0.3
done
while [ $(sudo /sbin/iptables --table nat -C PREROUTING --source 0.0.0.0/0 --destination $2 --jump DNAT
--to-destination $containerip; /bin/echo $?) != "0" ]
do
        sudo /sbin/iptables --table nat --insert PREROUTING --source 0.0.0.0/0 --destination $2 --jump
DNAT --to-destination $containerip
        /bin/sleep 0.3
done
while [ $(sudo /sbin/iptables --table nat -C PREROUTING --source 0.0.0.0/0 --destination $2 --protocol tcp
--match multiport --dport 22 --jump DNAT --to-destination 127.0.0.1:$4; /bin/echo $?) != "0" ]
do
        sudo /sbin/iptables --table nat --insert PREROUTING --source 0.0.0.0/0 --destination $2 --protocol
tcp --match multiport --dport 22 --jump DNAT --to-destination 127.0.0.1:$4
        /bin/sleep 0.3
done
sudo /sbin/sysctl -w net.ipv4.conf.all.route_localnet=1

loginfile="/home/student/number_logins/number_login_$3.txt"
if [ -f $loginfile ]
then
        number=$(/bin/cat $loginfile)
        number=$((number+1))
else
        /bin/touch $loginfile
        number=1
        # Ports are controlled by the file ports.txt. We are still deciding which ports we will have open.

        #sudo ip addr add $2/16 brd + dev eth0
        /bin/echo "created log file, added prerouting rules"
fi
/bin/echo ${number} > $loginfile

random_mod=$(($RANDOM % 16))

log_name="/home/student/data/HONEYPOT$3/logs/log_${random_mod}_${number}"
csv_log="/home/student/data/HONEYPOT$3/csv/csv_${random_mod}_${number}"
base_64_log="/home/student/data/HONEYPOT$3/bases/base_${random_mod}_${number}"
time_log="/home/student/data/HONEYPOT$3/time/time_${random_mod}_${number}"
idle_log="/home/student/idle_$3"
sudo /bin/rm $idle_log
sudo /bin/touch $idle_log
sudo /bin/lxc-cgroup -n $1 cpu.cfs_period_us 100000

if [ $random_mod -eq 0 ]
then
        sudo /bin/lxc-attach $1 -- bash -c "sudo tc qdisc add dev eth0 handle ffff: ingress"
        sudo /bin/lxc-attach $1 -- bash -c "sudo tc filter add dev eth0 protocol ip parent ffff: prio 50
u32 match ip src 0.0.0.0/0 police rate 256kbit burst 15k drop flowid :1"
```

```
        sudo /bin/lxc-attach $1 -- bash -c "sudo tc qdisc add dev eth0 root netem delay 1000ms"
        sudo /bin/lxc-cgroup -n $1 memory.limit_in_bytes 4000000000
        sudo /bin/lxc-cgroup -n $1 cpu.cfs_quota_us 20000
fi


if [ $random_mod -eq 1 ]
then
        sudo /bin/lxc-attach $1 -- bash -c "sudo tc qdisc add dev eth0 handle ffff: ingress"
        sudo /bin/lxc-attach $1 -- bash -c "sudo tc filter add dev eth0 protocol ip parent ffff: prio 50
u32 match ip src 0.0.0.0/0 police rate 256kbit burst 15k drop flowid :1"
        sudo /bin/lxc-attach $1 -- bash -c "sudo tc qdisc add dev eth0 root netem delay 1000ms"
        sudo /bin/lxc-cgroup -n $1 memory.limit_in_bytes 4000000000
        sudo /bin/lxc-cgroup -n $1 cpu.cfs_quota_us 10000
fi

if [ $random_mod -eq 2 ]
then
        sudo /bin/lxc-attach $1 -- bash -c "sudo tc qdisc add dev eth0 handle ffff: ingress"
        sudo /bin/lxc-attach $1 -- bash -c "sudo tc filter add dev eth0 protocol ip parent ffff: prio 50
u32 match ip src 0.0.0.0/0 police rate 256kbit burst 15k drop flowid :1"
        sudo /bin/lxc-attach $1 -- bash -c "sudo tc qdisc add dev eth0 root netem delay 1000ms"
        sudo /bin/lxc-cgroup -n $1 memory.limit_in_bytes 1000000000
        sudo /bin/lxc-cgroup -n $1 cpu.cfs_quota_us 20000
fi

if [ $random_mod -eq 3 ]
then
        sudo /bin/lxc-attach $1 -- bash -c "sudo tc qdisc add dev eth0 handle ffff: ingress"
        sudo /bin/lxc-attach $1 -- bash -c "sudo tc filter add dev eth0 protocol ip parent ffff: prio 50
u32 match ip src 0.0.0.0/0 police rate 256kbit burst 15k drop flowid :1"
        sudo /bin/lxc-attach $1 -- bash -c "sudo tc qdisc add dev eth0 root netem delay 1000ms"
        sudo /bin/lxc-cgroup -n $1 memory.limit_in_bytes 1000000000
        sudo /bin/lxc-cgroup -n $1 cpu.cfs_quota_us 10000
fi

if [ $random_mod -eq 4 ]
then
        sudo /bin/lxc-attach $1 -- bash -c "sudo tc qdisc add dev eth0 handle ffff: ingress"
        sudo /bin/lxc-attach $1 -- bash -c "sudo tc filter add dev eth0 protocol ip parent ffff: prio 50
u32 match ip src 0.0.0.0/0 police rate 256kbit burst 15k drop flowid :1"
        sudo /bin/lxc-cgroup -n $1 memory.limit_in_bytes 4000000000
        sudo /bin/lxc-cgroup -n $1 cpu.cfs_quota_us 20000
fi

if [ $random_mod -eq 5 ]
then
        sudo /bin/lxc-attach $1 -- bash -c "sudo tc qdisc add dev eth0 handle ffff: ingress"
        sudo /bin/lxc-attach $1 -- bash -c "sudo tc filter add dev eth0 protocol ip parent ffff: prio 50
u32 match ip src 0.0.0.0/0 police rate 256kbit burst 15k drop flowid :1"
        sudo /bin/lxc-cgroup -n $1 memory.limit_in_bytes 4000000000
        sudo /bin/lxc-cgroup -n $1 cpu.cfs_quota_us 10000
fi

if [ $random_mod -eq 6 ]
then
        sudo /bin/lxc-attach $1 -- bash -c "sudo tc qdisc add dev eth0 handle ffff: ingress"
        sudo /bin/lxc-attach $1 -- bash -c "sudo tc filter add dev eth0 protocol ip parent ffff: prio 50
u32 match ip src 0.0.0.0/0 police rate 256kbit burst 15k drop flowid :1"
        sudo /bin/lxc-cgroup -n $1 memory.limit_in_bytes 1000000000
        sudo /bin/lxc-cgroup -n $1 cpu.cfs_quota_us 20000
fi

if [ $random_mod -eq 7 ]
then
        sudo /bin/lxc-attach $1 -- bash -c "sudo tc qdisc add dev eth0 handle ffff: ingress"
        sudo /bin/lxc-attach $1 -- bash -c "sudo tc filter add dev eth0 protocol ip parent ffff: prio 50
u32 match ip src 0.0.0.0/0 police rate 256kbit burst 15k drop flowid :1"
        sudo /bin/lxc-cgroup -n $1 memory.limit_in_bytes 1000000000
        sudo /bin/lxc-cgroup -n $1 cpu.cfs_quota_us 10000
fi

if [ $random_mod -eq 8 ]
then
        sudo /bin/lxc-attach $1 -- bash -c "sudo tc qdisc add dev eth0 handle ffff: ingress"
        sudo /bin/lxc-attach $1 -- bash -c "sudo tc filter add dev eth0 protocol ip parent ffff: prio 50
u32 match ip src 0.0.0.0/0 police rate 2048kbit burst 15k drop flowid :1"
        sudo /bin/lxc-attach $1 -- bash -c "sudo tc qdisc add dev eth0 root netem delay 1000ms"
        sudo /bin/lxc-cgroup -n $1 memory.limit_in_bytes 4000000000
        sudo /bin/lxc-cgroup -n $1 cpu.cfs_quota_us 20000
fi

if [ $random_mod -eq 9 ]
then
        sudo /bin/lxc-attach $1 -- bash -c "sudo tc qdisc add dev eth0 handle ffff: ingress"
        sudo /bin/lxc-attach $1 -- bash -c "sudo tc filter add dev eth0 protocol ip parent ffff: prio 50
u32 match ip src 0.0.0.0/0 police rate 2048kbit burst 15k drop flowid :1"
        sudo /bin/lxc-attach $1 -- bash -c "sudo tc qdisc add dev eth0 root netem delay 1000ms"
        sudo /bin/lxc-cgroup -n $1 memory.limit_in_bytes 4000000000
        sudo /bin/lxc-cgroup -n $1 cpu.cfs_quota_us 10000
fi

if [ $random_mod -eq 10 ]
then
        sudo /bin/lxc-attach $1 -- bash -c "sudo tc qdisc add dev eth0 handle ffff: ingress"
        sudo /bin/lxc-attach $1 -- bash -c "sudo tc filter add dev eth0 protocol ip parent ffff: prio 50
u32 match ip src 0.0.0.0/0 police rate 2048kbit burst 15k drop flowid :1"
        sudo /bin/lxc-attach $1 -- bash -c "sudo tc qdisc add dev eth0 root netem delay 1000ms"
        sudo /bin/lxc-cgroup -n $1 memory.limit_in_bytes 1000000000
        sudo /bin/lxc-cgroup -n $1 cpu.cfs_quota_us 20000
fi
```

```
if [ $random_mod -eq 11 ]
then
        sudo /bin/lxc-attach $1 -- bash -c "sudo tc qdisc add dev eth0 handle ffff: ingress"
        sudo /bin/lxc-attach $1 -- bash -c "sudo tc filter add dev eth0 protocol ip parent ffff: prio 50
u32 match ip src 0.0.0.0/0 police rate 2048kbit burst 15k drop flowid :1"
        sudo /bin/lxc-attach $1 -- bash -c "sudo tc qdisc add dev eth0 root netem delay 1000ms"
        sudo /bin/lxc-cgroup -n $1 memory.limit_in_bytes 1000000000
        sudo /bin/lxc-cgroup -n $1 cpu.cfs_quota_us 10000
fi

if [ $random_mod -eq 12 ]
then
        sudo /bin/lxc-attach $1 -- bash -c "sudo tc qdisc add dev eth0 handle ffff: ingress"
        sudo /bin/lxc-attach $1 -- bash -c "sudo tc filter add dev eth0 protocol ip parent ffff: prio 50
u32 match ip src 0.0.0.0/0 police rate 2048kbit burst 15k drop flowid :1"
        sudo /bin/lxc-cgroup -n $1 memory.limit_in_bytes 4000000000
        sudo /bin/lxc-cgroup -n $1 cpu.cfs_quota_us 20000
fi

if [ $random_mod -eq 13 ]
then
        sudo /bin/lxc-attach $1 -- bash -c "sudo tc qdisc add dev eth0 handle ffff: ingress"
        sudo /bin/lxc-attach $1 -- bash -c "sudo tc filter add dev eth0 protocol ip parent ffff: prio 50
u32 match ip src 0.0.0.0/0 police rate 2048kbit burst 15k drop flowid :1"
        sudo /bin/lxc-cgroup -n $1 memory.limit_in_bytes 4000000000
        sudo /bin/lxc-cgroup -n $1 cpu.cfs_quota_us 10000
fi

if [ $random_mod -eq 14 ]
then
        sudo /bin/lxc-attach $1 -- bash -c "sudo tc qdisc add dev eth0 handle ffff: ingress"
        sudo /bin/lxc-attach $1 -- bash -c "sudo tc filter add dev eth0 protocol ip parent ffff: prio 50
u32 match ip src 0.0.0.0/0 police rate 2048kbit burst 15k drop flowid :1"
        sudo /bin/lxc-cgroup -n $1 memory.limit_in_bytes 1000000000
        sudo /bin/lxc-cgroup -n $1 cpu.cfs_quota_us 20000
fi

if [ $random_mod -eq 15 ]
then
        sudo /bin/lxc-attach $1 -- bash -c "sudo tc qdisc add dev eth0 handle ffff: ingress"
        sudo /bin/lxc-attach $1 -- bash -c "sudo tc filter add dev eth0 protocol ip parent ffff: prio 50
u32 match ip src 0.0.0.0/0 police rate 2048kbit burst 15k drop flowid :1"
        sudo /bin/lxc-cgroup -n $1 memory.limit_in_bytes 1000000000
        sudo /bin/lxc-cgroup -n $1 cpu.cfs_quota_us 10000
fi

sudo /bin/forever -l $log_name.log start /home/student/MITM/mitm.js -n $1 -i $containerip -p $4
--auto-access --auto-access-fixed 1 --debug

/bin/echo "MADE IT PAST GETTING PID AND MITM"
while [ ! -f $log_name.log ]
```

```
do
        /bin/sleep 0.001
done
/bin/echo "made it past sleep"
if (/bin/tail -f -n0 $log_name.log | /bin/grep -q  "Auto-access is now disabled for the remainder"
--line-buffered)
then
        /bin/echo "auto access reached"
        attackerip=$(/bin/cat $log_name.log | /bin/grep "Attacker connected" | /bin/cut -d' ' -f8)
        /bin/echo "attacker ip grabbed"
        sudo /sbin/iptables --insert INPUT --source !$attackerip --destination 127.0.0.1 -p tcp --dport $4
-j DROP
        /bin/echo "new rule added"
        (/bin/sleep 30; /home/student/pot_notime_timeout $log_name.log) &
        if (/bin/tail -f -n0 $log_name.log | /bin/grep -q  "is inside container" --line-buffered)
        then
                current_time=$(/bin/date +%s%3N)
        fi
        /home/student/new_data_collection $1 $csv_log 0 $current_time "$3" "${random_mod}" "${number}" &
        collection_pid=$(/bin/echo $!)
        /bin/echo "ATTACKER IN"
        # 20 minute timer, appends to waiting log file if time elapses
        (/bin/sleep 1200; /home/student/pot_timeout $log_name.log) &
fi
/bin/echo "MADE IT PAST AUTHENTICATION"
newtime=$(/bin/date +%s%3N)
/home/student/idle_script $log_name.log $idle_log $newtime &
/home/student/our_closer $log_name.log &
if (/bin/tail -f -n0 $log_name.log | /bin/grep -q " closed connection" --line-buffered)
then
        end_t=$(/bin/date +%s%3N)
        if [ $(/bin/wc -l $idle_log | awk '{print $1}') -ne 0 ]
        then
                ending_time=$(($end_t - $current_time - 300000))
        else
                ending_time=$(($end_t - $current_time))
        fi

        /bin/echo "$3," "${random_mod}," "${number}," $ending_time | sudo /bin/tee -a $time_log
        sudo /bin/kill -9 $(sudo /bin/forever list -q | /bin/sed -r
"s/\x1B\[([0-9]{1,3}(;[0-9]{1,2};?)?)?[mGK]//g" | /bin/awk '{$1=$1};1' | /bin/cut -d' ' -f 4- | /bin/grep
$4 | /bin/rev | /bin/cut -d' ' -f 4 | /bin/rev)
        sudo /bin/kill -9 $(sudo /bin/forever list -q | /bin/sed -r
"s/\x1B\[([0-9]{1,3}(;[0-9]{1,2};?)?)?[mGK]//g" | /bin/awk '{$1=$1};1' | /bin/cut -d' ' -f 4- | /bin/grep
$4 | /bin/rev | /bin/cut -d' ' -f 3 | /bin/rev)
        sudo /bin/kill -9 $(sudo /bin/lsof -i :$4 | /bin/awk '{print $2}' | /bin/tail -1)
        sudo /bin/kill -9 $(sudo /bin/lsof -i :$4 | /bin/awk '{print $2}' | /bin/tail -1)

        #get rid of collection script
        sudo /bin/kill -9 $collection_pid
        sudo /sbin/iptables --delete INPUT --source !$attackerip --destination 127.0.0.1 -p tcp --dport $4
-j DROP
        # get rid of iptables rules
        while [ $(sudo /sbin/iptables --table nat --delete PREROUTING --source 0.0.0.0/0 --destination $2
--protocol tcp --match multiport --dport 22 --jump DNAT --to-destination 127.0.0.1:$4; /bin/echo $?) !=
"0" ]
        do
                /bin/sleep 0.3
        done
        /bin/sleep 1
        while [ $(sudo /sbin/iptables --table nat --delete PREROUTING --source 0.0.0.0/0 --destination $2
--jump DNAT --to-destination $containerip; /bin/echo $?) != "0" ]
        do
                /bin/sleep 0.3
        done
        while [ $(sudo /sbin/iptables --table nat --delete POSTROUTING --source $containerip --destination
0.0.0.0/0 --jump SNAT --to-source $2; /bin/echo $?) != "0" ]
        do
                /bin/sleep 0.3
        done

        #Runs collection script, queues job to fetch data once prepared.
        #Puts data in /home/student/data with filenames indicating honeypot name
        #and the filename that was analyzed
        #(./collect.sh $1 "/home/student/data/${1}/cfg_${cfg}_no_${number}") &
        /home/student/get_virus_results.sh $1 $base_64_log
        source /home/student/honeypot.sh $1 $2 $3 $4 $pid
fi
```

Curl and Wget Poisoner

```
#!/bin/bash

# REQUIRED SETUP: Run `apt install strace` inside the container; everything else needed (bash,
 grep, cut, xxd) should already be installed
# NOTE: The numbers after `--write=` might vary depending on the curl/wget version and the
 command-line options.
#       However, if the numbers are wrong, most of the file contents might not be captured,
 but the attacker should not notice anything.
# COMMENT: This approach avoids having any significant effect on network traffic or command
 behavior regardless of options
#          This works by using strace to log all of the syscalls made by curl/wget/scp,
 including hexdumping the full contents
```

```
#          of any data written to the file descriptors that I observed it uses for its output
 file/stream.


if [[ $# -ne 1 ]]
then
  echo "usage: $0 <container>"
  exit 1
fi

# https://stackoverflow.com/a/1168084/5445670
python_script_base64=$(
base64 -w0 <<'EOF'
import re
import string
import sys


_, prefix, command, *argv = sys.argv  # usage: cat ...-strace.txt | python3 ....py $prefix
 $command "$@" (command is wget, curl, or scp)

stdin_buffer = []
stdin_closed = False
stdout = {'name': '<STDOUT>', 'data_written': []}
stderr = {'name': '<STDERR>', 'data_written': []}
interesting_open_files = {1: stdout, 2: stderr}
interesting_closed_files = []
current_buffer = None

for line in sys.stdin:
    if current_buffer is not None:
        if line.startswith(' | '):
            current_buffer.append(bytes.fromhex(line[10:59]))
            continue
        else:
            current_buffer = None
    if line.startswith('openat('):
        match = re.fullmatch(r'openat\(AT_FDCWD, (".*"), [A-Z_|]*O_CREAT[A-Z_|]*, \d+\) =
 (\d+)[\r\n]*', line)
        if match is not None:
            name, fd = match.groups()
            interesting_open_files[int(fd)] = {'name': name, 'data_written': []}
    elif line.startswith('write('):
        fd = line[line.find('(')+1:line.find(',')]
        try:
            file = interesting_open_files[int(fd)]
        except KeyError:
            pass
        else:
            current_buffer = file['data_written']
    elif line.startswith('read(0,') and not stdin_closed:
        current_buffer = stdin_buffer
    elif line.startswith('close('):
        fd = line[line.find('(')+1:line.find(')')]
        if fd == '0' and not stdin_closed:
            stdin_closed = True
        else:
            try:
                file = interesting_open_files[int(fd)]
            except KeyError:
                pass
            else:
                interesting_closed_files.append({'name': file['name'], 'data_written':
 b''.join(file['data_written'])})
                del interesting_open_files[int(fd)]

for file in interesting_open_files.values():
    processed_file = {'name': file['name'], 'data_written': b''.join(file['data_written'])}
    if file['name'] == '<STDERR>':
        processed_stderr = processed_file
    else:
        interesting_closed_files.append(processed_file)

with open(f'{prefix}--stderr.txt', 'xb') as f:
    # noinspection PyUnboundLocalVariable
    f.write(processed_stderr['data_written'])

stdin_data = b''.join(stdin_buffer)
with open(f'{prefix}--stdin.txt', 'xb') as f:
    # noinspection PyUnboundLocalVariable
    f.write(stdin_data)
```

```
OPEN_QUOTE = '‘'.encode('UTF-8')
CLOSE_QUOTE = '’'.encode('UTF-8')

sources = []
if command == 'curl' or (command == 'wget' and not processed_stderr['data_written']):
    saw_dash_dash = False
    for arg in argv:
        if arg == '--':
            saw_dash_dash = True
        elif saw_dash_dash or not arg.startswith('-'):
            sources.append({'source_name': arg})
elif command == 'wget':
    url = length = filename = None
    for line in processed_stderr['data_written'].splitlines(False):
        if line.startswith(b'--'):
            _, url_bytes = line.split(b'--  ')
            url = url_bytes.decode('ASCII')
        elif line.startswith(b'Saving to:'):
            _, filename_and_close_quote = line.split(OPEN_QUOTE)
            if not filename_and_close_quote.endswith(CLOSE_QUOTE):
                raise ValueError(f'{filename_and_close_quote=} should end in {CLOSE_QUOTE}')
            filename_bytes = filename_and_close_quote[:-len(CLOSE_QUOTE)]
            filename = filename_bytes.decode('ASCII')
        elif b' saved [' in line:
            if OPEN_QUOTE + filename.encode('ASCII') + CLOSE_QUOTE not in line:
                raise ValueError(f'{line=} does not contain quoted {filename=}')
            _, length_info_with_closing_bracket = line.rsplit(b'[')
            if not length_info_with_closing_bracket.endswith(b']'):
                raise ValueError(f'{length_info_with_closing_bracket=}')
            length_info = length_info_with_closing_bracket[:-1]
            # using `set` and unpacking ensures that all the slash-separated parts are the
 same
            length_extracted, = set(length_info.split(b'/'))
            length = int(length_extracted)
            sources.append({'source_name': url, 'data_length': length, 'destination_name':
 filename})
            url = length = filename = None
elif command == 'scp':
    if set(stdin_data) <= set(b'\x00\x01\x02'):
        pass  # this is sending, not receiving, a file
    else:
        i = 0
        if stdin_data[i] in set(b'\x01\x02'):
            # error message
            i = stdin_data.find(b'\n', i) + 1
        elif stdin_data[i] == b'C'[0]:
            mode = stdin_data[i+1:i+5]
            if not (set(mode) <= set(b'01234567')):
                raise ValueError(f'mode is not octal: {mode}')
            space = stdin_data[i+5]
            if space != b' '[0]:
                raise ValueError(f'space is not space: {space}')
            end_of_size = stdin_data.find(b' ', i + 6)
            size_str = stdin_data[i + 5:end_of_size]
            size = int(size_str)
            end_of_filename = stdin_data.find(b'\n', end_of_size)
            filename = stdin_data[end_of_size+1:end_of_filename].decode('ASCII')
            data = stdin_data[end_of_filename+1:end_of_filename+1+size]
            i = end_of_filename + 1 + size
            sources.append({'source_name': filename, 'data': data, 'data_length': len(data)})
        elif stdin_data[i] == b'D'[0]:
            i = stdin_data.find(b'\n', i) + 1
        elif stdin_data[i] == b'E'[0]:
            if i+1 != len(stdin_data):
                raise ValueError(f'reached exit with more data after: {i=},
 {len(stdin_data)=}')
        elif stdin_data[i] == b'T'[0]:
            i = stdin_data.find(b'\n', i) + 1
else:
    raise ValueError(f'unknown command: {command}')

with open(f'{prefix}--filename-info.txt', 'x') as f:
    # noinspection PyUnboundLocalVariable
    f.write(repr(sources))

num_interesting_files = len(interesting_closed_files)
num_digits = len(str(num_interesting_files))

def sanitize_filename_char(c):
    if c in string.ascii_letters + string.digits:
        return c
    elif c == '/':
        return '-'
    elif c in '"<>':
        return ''
    else:
        return '_'

def sanitize_filename(filename):
    while '//' in filename: # normalize slashes
        filename = filename.replace('//', '/')
    return ''.join(map(sanitize_filename_char, filename))



def try_resolve_source_name(dest_name, data):
    data_length = len(data)
    matching_source_name = None
    for source in sources:
        if (source.get('dest_name', None) in {dest_name, None}
                and source.get('data_length', None) in {data_length, None}
                and source.get('data', None) in {data, None}):
            if matching_source_name is None:
                matching_source_name = source['source_name']
            else:
                # multiple matches found
                return None
    return matching_source_name



for i, file in enumerate(interesting_closed_files):
    source_name = try_resolve_source_name(file['name'], file['data_written'])
    encoded_filename = sanitize_filename(file['name'])
    if source_name is None:
        encoded_source_name = ""
    else:
        encoded_source_name = "--" + sanitize_filename(source_name)
    full_file_path = f'{prefix}-{"" if file["data_written"] else
 "empty"}-{str(i).zfill(num_digits)}--{encoded_filename}{encoded_source_name}.bin'
    with open(full_file_path, 'xb') as f:
       f.write(file['data_written'])
EOF
)
```

```
sudo lxc-attach -n "$1" -- mkdir /var/log/.downloads
sudo lxc-attach -n "$1" -- chmod a+rwx /var/log/.downloads
sudo lxc-attach -n "$1" -- mv -n /usr/bin/curl{,-real}
sudo lxc-attach -n "$1" -- mv -n /usr/bin/wget{,-real}
sudo lxc-attach -n "$1" -- mv -n /usr/bin/scp{,-real}
sudo lxc-attach -n "$1" -- sh -c 'echo "#!/bin/bash\nprefix=/var/log/.downloads/\$(date -u
 +%Y%m%d_%H%M%S_%s_%N)\necho curl \"\$@\" > \$prefix-command.txt\nenv >
 \$prefix-env.txt\n(strace --read=0 --write=all -o \$prefix-strace.txt /usr/bin/curl-real
 \"\$@\" 3>&1 >&2 2>&3 3>&- | sed \"s/\<curl-real\>/curl/g\") 3>&1 >&2 2>&3 3>&-\ncat
 \$prefix-strace.txt | python3 -c \"\$(echo '"$python_script_base64"' | base64 -d)\"
 \"\$prefix\" curl \"\$@\"" > /usr/bin/curl'
sudo lxc-attach -n "$1" -- sh -c 'echo "#!/bin/bash\nprefix=/var/log/.downloads/\$(date -u
 +%Y%m%d_%H%M%S_%s_%N)\necho wget \"\$@\" > \$prefix-command.txt\nenv >
 \$prefix-env.txt\n(strace --read=0 --write=all -o \$prefix-strace.txt /usr/bin/wget-real
 \"\$@\" 3>&1 >&2 2>&3 3>&- | sed \"s/\<wget-real\>/wget/g\") 3>&1 >&2 2>&3 3>&-\ncat
 \$prefix-strace.txt | python3 -c \"\$(echo '"$python_script_base64"' | base64 -d)\"
 \"\$prefix\" wget \"\$@\"" > /usr/bin/wget'
sudo lxc-attach -n "$1" -- sh -c 'echo "#!/bin/bash\nprefix=/var/log/.downloads/\$(date -u
 +%Y%m%d_%H%M%S_%s_%N)\necho scp \"\$@\" > \$prefix-command.txt\nenv >
 \$prefix-env.txt\n(strace --read=0 --write=all -o \$prefix-strace.txt /usr/bin/scp-real
 \"\$@\" 3>&1 >&2 2>&3 3>&- | sed \"s/\<scp-real\>/scp/g\") 3>&1 >&2 2>&3 3>&-\ncat
 \$prefix-strace.txt | python3 -c \"\$(echo '"$python_script_base64"' | base64 -d)\"
 \"\$prefix\" scp \"\$@\"" > /usr/bin/scp'
sudo lxc-attach -n "$1" -- chmod a+x /usr/bin/{curl,wget,scp}

# This is what the generated file looks like for curl; wget and scp are similar, just with
 `curl` replaced with `wget` or `scp`:
# ```
# #!/bin/bash
# prefix=/var/log/.downloads/$(date -u +%Y%m%d_%H%M%S_%s_%N)
# echo curl "$@" > $prefix-command.txt
# env > $prefix-env.txt
# (strace --read=0 --write=all -o $prefix-strace.txt /usr/bin/wget-real "$@" 3>&1 >&2 2>&3
 3>&- | sed 's/\<curl-real\>/curl/g') 3>&1 >&2 2>&3 3>&-
# cat $prefix-strace.txt | python3 -c "$(echo ... base64-encoded Python script ... | base64
 -d)" "$prefix" curl "$@"
# ```
# Notes about the strace line:
# - The I/O redirection stuff swaps stdout and stderr back and forth; see
 https://stackoverflow.com/a/52575087/5445670,
```

```
 https://www.baeldung.com/linux/filter-standard-error,
 https://www.gnu.org/software/bash/manual/bash.html#Basic-Shell-Features
# - The regex things are word boundaries; see https://stackoverflow.com/a/2458693/5445670
```

Works Cited

Chamotra, S., Sehgal, R. K., Ror, S., & singh, B. (2016). Honeypot Deployment in Broadband Networks. In Ray, I., Gaur, M. S., Conti, M., Sanghi, D., & Kamakoti, V. (Eds.), *Lecture Notes in Computer Science: Vol. 10063. Information Systems Security: 12th International Conference, ICISS 2016, Jaipur, India, December 16–20, 2016, Proceedings* (pp. 479–488). Springer. https://link.springer.com/chapter/10.1007/978-3-319-49806-5_27

Horowitz, D. (2024, July 31). CPU Cores Explained: How Many Do You Need? *HP® Tech Takes*. https://www.hp.com/us-en/shop/tech-takes/cpu-cores-how-many-do-i-need

Huang, D. Y., Dharmdasani, H., Meiklejohn, S., Dave, V., Grier, C., McCoy, D., Savage, S., Weaver, N., Snoeren, A. C., & Levchenko, K. (2014). *Botcoin: Monetizing Stolen Cycles*. Internet Society. Presented at 21st Network and Distributed System Security Symposium, NDSS '14, 23–26 February 2014, San Diego, CA, USA. https://cseweb.ucsd.edu//~snoeren/papers/botcoin-ndss14.pdf, https://nyuscholars.nyu.edu/en/publications/botcoin-monetizing-stolen-cycles, https://doi.org/10.14722/ndss/2014.23044

Newman, L. H. (2018, February 12). Now Cryptojacking Threatens Critical Infrastructure, Too. *WIRED*. Condé Nast. https://www.wired.com/story/cryptojacking-critical-infrastructure/

Zhuge, J., Holz, T., Han, X., Song, C., & Zou, W. (2007). Collecting Autonomous Spreading Malware Using High-Interaction Honeypots. In Qing, S., Imai, H., & Wang, G. (Eds.) *Lecture Notes in Computer Science: Vol. 4861. Information and Communications Security: 9th International Conference, ICICS 2007, Zhengzhou, China, December 2007, Proceedings* (pp. 438–451). Springer. https://link.springer.com/chapter/10.1007/978-3-540-77048-0_34