macquarie
CLOUD SERVICES

BPFDoor Technical Analysis

An Evasive Linux Backdoor

An active Chinese global surveillance tool associated with Chinese threat actors turned out to be deployed on "thousands" of systems running Linux.

Debarghaya Dasgupta

Contents

Introduction

BPFdoor reminds me of my old project AmOgh(A rootkit simulator) that I published to help security researchers about 3years ago. In principle, these are both very similar to their trigger functions. I found out about BPFDoor from my colleague Branislav.

Google led me to an article by Security researcher Kevin Beaumont, who uncovered a new evasive backdoor targeting Linux associated with the Chinese Red Menshen threat actors. Investigators at PricewaterhouseCoopers have also noted the backdoor in their latest Cyber Threat Intelligence Retrospect Report (p. 36).

I obtained the source code for this backdoor from Pastebin by user JOHNGALT14, which enabled me to perform a deep analysis, followed by the ELF controller from a popular Malware Research forum.

BPFDoor performs the following functions at a high level:

- Goes memory resident and uses anti-forensics and evasion to hide.
- It loads a Berkeley Packet Filter (BPF) sniffer to watch traffic efficiently and work in front of local firewalls to see packets (hence BPFDoor).
- Upon receiving a particular packet, it modifies the local firewall to allow the attacker's IP address to access resources like spawned shells or bindshell connections.
- To avoid detection, operations are hidden using process masquerading.

While the malware takes steps to evade casual observation, it is easily seen if you know where and how to look. I assume my previous experience with AmOgh also helped me.

Source Build Size and Compatibility

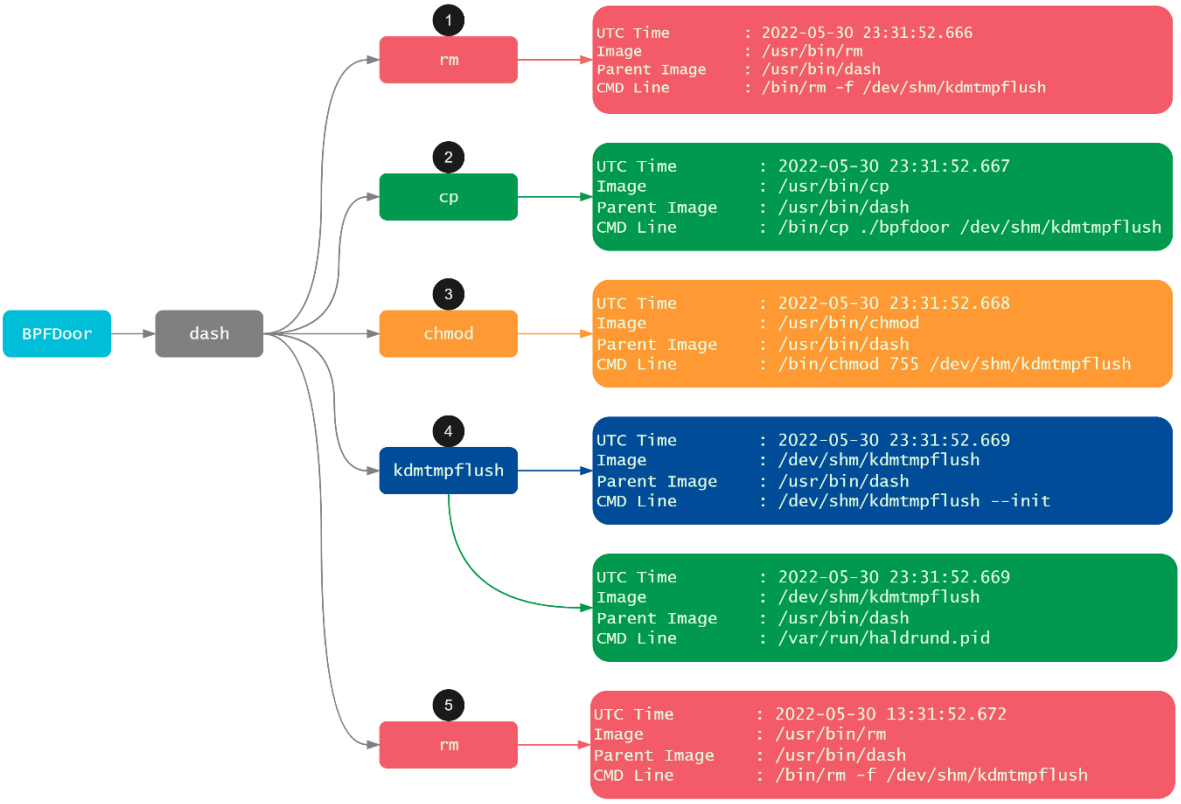
The BPFDoor source is small, focused, and well written. BPF is widely available across various operating systems, and its core-shell functions are likely to work across platforms without significant modifications. The dynamically linked binary is small at about 35K on Debian:

```
-rwxr-xr-x  1 attackdefenseadmin attackdefenseadmin   34952 May 29 11:28 bpfdoor
```

Implant Operation Steps

The binary itself 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. It does, however, require root permissions to run.

When run, the binary has an initialisation 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 timestamps the binary file /dev/shm/kdmtmpflush and initiates a packet capture loop.
5. The forked version creates a dropper at /var/run/haldrund.PID to mark it as a resident and prevent multiple starts.
6. The actual execution process deletes the timestamped /dev/shm/kdmtmpflush and exits.
7. The forked version remains resident and monitors traffic for a magic packet to initiate attacker operations such as a bindshell.

Persistence

The implant is focused on a single task, so it does not have persistence mechanisms. 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.

The implant will need to be somewhere else on the host to survive reboots or be inserted again remotely for persistence reasons. Incident response teams that find this implant operating should assume the actual 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 only in RAM and clears out every reboot. The exciting 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 an epoch which translates to **Thursday, October 30, 2008, 7:17:16 PM** (GMT)

We searched to see if this date was significant but didn't see anything obvious. It could have some importance to the author or could be random.

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

```
$ls -al /dev/shm
total 36
drwxrwxrwt   3  root root    60  May 29 23:33 .
drwxr-xr-x   18 root root  3960  May 29 23:33 ..
-rwxr-xr-x   1  root root  36616 Oct 30 2008  kdmtmpflush
```

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 the file may be left behind if there is a problem, such as a hard shutdown or crash. The implant will not start if this file is present as it is used to mark that it may already be running.

```
$strace -f ./bpfdoor
execve("./bpfdoor", ["./bpfdoor"], 0x7ffeb541c368 /* 30 vars */) = 0
brk(NULL)                               = 0x5590343b5000
arch_prctl(0x3001 /* ARCH_??? */, 0x7fff53f114e0) = -1 EINVAL (Invalid argument)
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=28424, ...}) = 0
mmap(NULL, 28424, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7fe76a403000
close(3)                                = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\300A\2\0\0\0\0\0"..., 832) = 832
pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784
pread64(3, "\4\0\0\0\20\0\0\0\5\0\0\0GNU\0\2\0\0\300\4\0\0\0\3\0\0\0\0\0\0", 32, 848) = 32
pread64(3, "\4\0\0\0\24\0\0\0\3\0\0\0GNU\0\30x\346\264ur\f|Q\226\236i\253-'o"..., 68, 880) = 68
fstat(3, {st_mode=S_IFREG|0755, st_size=2029592, ...}) = 0
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fe76a401000
pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784
pread64(3, "\4\0\0\0\20\0\0\0\5\0\0\0GNU\0\2\0\0\300\4\0\0\0\3\0\0\0\0\0\0", 32, 848) = 32
pread64(3, "\4\0\0\0\24\0\0\0\3\0\0\0GNU\0\30x\346\264ur\f|Q\226\236i\253-'o"..., 68, 880) = 68
mmap(NULL, 2037344, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7fe76a20f000
mmap(0x7fe76a231000, 1540096, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x22000) =
0x7fe76a231000
mmap(0x7fe76a3a9000, 319488, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x19a000) = 0x7fe76a3a9000
mmap(0x7fe76a3f7000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1e7000) =
0x7fe76a3f7000
mmap(0x7fe76a3fd000, 13920, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) =
0x7fe76a3fd000
close(3)                                = 0
arch_prctl(ARCH_SET_FS, 0x7fe76a402540) = 0
mprotect(0x7fe76a3f7000, 16384, PROT_READ) = 0
mprotect(0x5590342a2000, 4096, PROT_READ) = 0
mprotect(0x7fe76a437000, 4096, PROT_READ) = 0
munmap(0x7fe76a403000, 28424)           = 0
access("/var/run/haldrund.pid", R_OK)   = -1 ENOENT (No such file or directory)
getuid()                                = 1000
exit_group(0)                           = ?
+++ exited with 0 +++
```

In general, any process deleting its binary after running will be malicious.

```
$ls -alR /proc/*/exe 2> /dev/null | grep deleted
lrwxrwxrwx 1 root root 0 May 29 23:34 /proc/2133/exe -> /dev/shm/kdmtmpflush (deleted)
```

Masquerading

This is a partial list of process names that the malware can use to masquerade himself.

```
/sbin/udevd -d
/sbin/mingetty /dev/tty
/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
/usr/lib/systemd/systemd-machined
/usr/libexec/postfix/master
qmgr -l -t fifo -u
```

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

```
$ps aux | grep "dbus-daemon --system"
message+     726  0.0  0.1   7640  4812 ?        Ss   May29   0:00 /usr/bin/dbus-daemon --system --address=systemd: --nofork --nopidfile --systemd-activation --syslog-only
attackd+   47175  0.0  0.0   8176  2548 pts/0    S+   03:27   0:00 grep --color=auto dbus-daemon --system
```

If you find a suspicious process ID (PID), you can quickly investigate what the actual name may be by going to /proc/<PID>. You will see the exe link, which will be pointing to the exact binary location, which can confirm at least what the binary was called when it started. Also, you'll see the timestamp on the file is when the binary was created, which can help bracket the time of intrusion.

Linux also helpfully labels the binary as "(deleted)".

```
$cd /proc/726
$sudo ls -al
total 0
dr-xr-xr-x   9 messagebus messagebus 0 May 29 15:24 .
dr-xr-xr-x 203 root       root       0 May 29 15:24 ..
-r--r--r--   1 root       root       0 May 30 03:24 arch_status
dr-xr-xr-x   2 messagebus messagebus 0 May 29 15:24 attr
-rw-r--r--   1 root       root       0 May 30 03:24 autogroup
-r--------   1 root       root       0 May 30 03:24 auxv
-r--r--r--   1 root       root       0 May 29 15:24 cgroup
--w-------   1 root       root       0 May 30 03:24 clear_refs
-r--r--r--   1 root       root       0 May 29 15:24 cmdline
-rw-r--r--   1 root       root       0 May 29 15:24 comm
-rw-r--r--   1 root       root       0 May 30 03:24 coredump_filter
-r--r--r--   1 root       root       0 May 30 03:24 cpu_resctrl_groups
-r--r--r--   1 root       root       0 May 30 03:24 cpuset
lrwxrwxrwx   1 root       root       0 May 30 03:24 cwd -> /
-r--------   1 root       root       0 May 30 03:24 environ
lrwxrwxrwx   1 root       root       0 May 29 15:24 exe -> '/dev/shm/kdmtmpflush (deleted)'
dr-x------   2 root       root       0 May 29 15:24 fd
dr-x------   2 root       root       0 May 30 03:24 fdinfo
-rw-r--r--   1 root       root       0 May 30 03:24 gid_map
-r--------   1 root       root       0 May 30 03:24 io
-r--r--r--   1 root       root       0 May 30 03:24 limits
-rw-r--r--   1 root       root       0 May 29 15:24 loginuid
```

Environment Wipe

Before going fully resident, the implant's last thing is to wipe out the process environment. When you start a process on Linux, it stores a lot of useful forensic information in /proc/<PID>/environ. SSH connections that started the process, usernames, and other helpful information can often be found here.

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

This is a pretty good mechanism and ensures no environment variables are left in the running process. The result is that the implant goes the environment completely blank, which can happen under 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. An empty environment is unusual for standard processes.

```
$sudo strings /proc/726/environ
LANG=C.UTF-8
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/snap/bin
SGX_AESM_ADDR=1
LISTEN_PID=726
LISTEN_FDS=1
LISTEN_FDNAMES=dbus.socket
SYSTEMD_NSS_DYNAMIC_BYPASS=1
INVOCATION_ID=71d7c5aebb604a7495a39b2485515251
JOURNAL_STREAM=8:22997
```

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 analysed 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 regular expression syntax cannot represent.

We have reverse-engineered the bytecode below to show you what it is doing. It does 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 are here:

```
A10:    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 begins at offset 14 of ethernet packet; so this is offset 9 in the IPv4 packet)
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 datagram data
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 from l3; register contains IP protocol)
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 ICMP data
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 goto l29
l17:    jeq #0x6, l18, l29     // if Protocol==TCP goto l18, else goto l29 (jumped to from l3; register contains IP protocol)
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 X=20; packet offset 46 = TCP segment offset 12
l22:    and #0xf0              // A = A & 0xf0 (high nibble of TCP offset 12 = Data offset = TCP header size in 32-bit words)
l23:    rsh #2                 // A = A >> 2 (this has the effect of multiplying the high nibble by four, e.g A>>4 then A<<2). A now contains number of bytes in the TCP header
l24:    add x                  // A = A + X. Adding IPv4 header length + TCP header length.
l25:    tax                    // X = A
l26:    ldh [x+14]             // A = halfword from packet offset X+14 (14 is ethernet header, x is IPv4+TCP header, so this offset is first TCP payload byte)
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;
}
```

BPF packet filter that filters the following traffic

- The incoming packet is IPv4
- If the incoming packet is UDP, check that 0x7255 is in its payload
- If the incoming packet is ICMP, check that it is an ICMP Echo Request and that 0x7255 is in its payload
- If the incoming packet is TCP, check that 0x5293 is in its payload

Note:

- ✓ If the incoming packet matches the above conditions, then the code will check that it includes the correct password; there are two different passwords – the source code calls them 'pass' and 'pass2' – (both encrypted with RC4).
- ✓ If the provided password does not match any of the two passwords, it sends back a UDP packet with the payload '1' to the port included in the incoming packet. This is a strange behaviour because it can be used for remotely scanning compromised hosts (as this bpfdoor-scanner is doing), but we guess it is also a method for remotely detecting that the backdoor is still active.
- ✓ If one of the passwords is correct, then
    - If the password is the same one initialised in 'pass' - in the leaked source code, it is 'justforfun'- it will open a remote shell in a high TCP port.
    - If the password is the same one initialised in 'pass2' - in the leaked source code, it is 'socket' - it will start a reverse shell to the remote IP address and the remote port included in the incoming packet.

You will need to send the correct data in the packet to get past the filter, as shown above. The filter rejects most traffic from entering the main packet decoding loop so that the implant will run with minimal CPU signature. Packets that make it through the BPF check are quickly checked for a valid password before further operations occur.

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 a crucial point to understand.

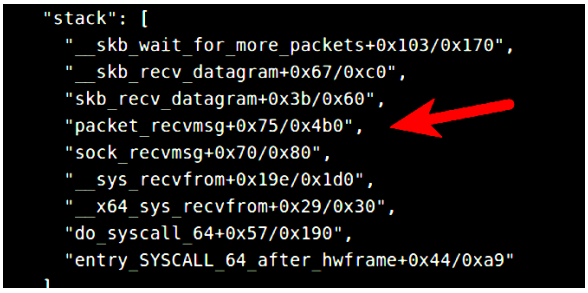
Further, if you have a network perimeter firewall that allows traffic to a host on ICMP, UDP, or TCP to any port, the implant can see it. Any inbound traffic to the server can activate the magic packet check and initiate a backdoor.

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 typical 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 reference to function calls in the Linux kernel, indicating that the process is likely grabbing packets.



You can search the entire /proc/*/stack area for any process showing packet capture functions like the above. False alarms with this search are possible, but you can narrow down potential candidates like the one below. The red arrow is the PID in question running BPFDoor.

```
$sudo grep packet_recvmsg /proc/*/stack
/proc/48353/stack:[<0>] packet_recvmsg+0x75/0x4b0
/proc/750/stack:[<0>] unix_seqpacket_recvmsg+0x1a/0x30d
```

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 robust cipher for this application and is the only genuinely secure cipher you can write on a napkin. It's an excellent choice for a 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 an excellent 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 set up 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". Some operation check status might show the implant is still running. Relevant code below:

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

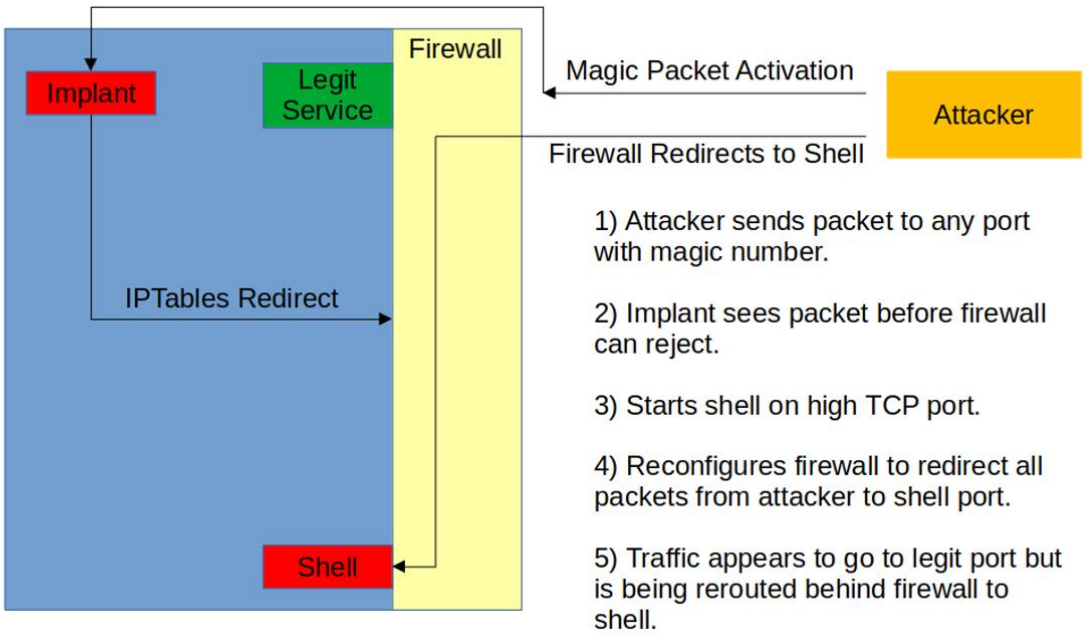
Note the above is not passed through the RC4 encryption. If you are investigating this on your network, it may be worthwhile looking for single UDP packets with just the data "1" in them and nothing else from many hosts over time to a single host IP (controller).

Firewall Bypass for Bindshell Backdoor

The implant has a unique feature to bypass the host firewall and make the traffic look legitimate. When the host receives the magic packet, 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. The traffic will look like TLS/SSL traffic, but the attacker interacts 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:

- ✓ Implant listens to all traffic coming to the host regardless of the firewall.
- ✓ Implant sees the magic packet.
- ✓ Magic packet can contain the attacker's IP address, port, and password. Otherwise, it uses the origin IP address.
- ✓ Depending on the password, the implant will open a local or connect-back backdoor.
- ✓ Implant selects a TCP port sequentially starting at 42391 up to 43391.
- ✓ Implant binds to the first unused port in this range.
- ✓ For the local shell, iptables is called to redirect all traffic from the attacker host IP from the requested port to the bound port range from the above steps.
- ✓ Attacker connects with TCP to the defined port they requested (e.g. TCP port 443).
- ✓ Traffic from that host is redirected from the port to the shell routing behind the firewall.
- ✓ Observed traffic still appears to be going to the host on a legitimate port but is being routed to the shell on the host.

NOTE: We disabled the RC4 encryption in the implant, for example, using Netcat. The actual implant would require the correct password and RC4 encryption layer to see these results. We connected to a host on SSH port 22. We get back a regular 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 redirects the traffic. When we connect again to port 22, we get a shell with root access instead of SSH. Here again, is the crucial 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!

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, it 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 172.16.3.1 --dport 22 -j REDIRECT --to-ports 42392
/sbin/iptables -D INPUT -p tcp -s 172.16.3.1 -j ACCEPT
```

Connect-Back Bindshell

The implant also can connect back to a host as defined in the magic packet. The operation here is essentially the same, but this may not be as stealthy as 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

If you do not supply any password or an incorrect password in a magic packet, the implant will send out "1" on UDP. We believe this is some status check to allow keeping tabs on many systems. **Organisations may 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. The shell also sets up some anti-forensics for good measure. The environment is carefully selected only to have the following. We'll use *strings /proc/<PID>/environ* to look at what it is doing below:

```
$sudo strings /proc/1239/environ
HOME=/tmp
PS1=[\u@\h \W]\\$
HISTFILE=/dev/null
MYSQL_HISTFILE=/dev/null
PATH=/bin:/usr/kerberos/sbin:/usr/kerberos/bin:/sbin:/usr/bin:/usr/sbin:/usr/local/bin:/usr/local/sbin:/usr/X11R6/bin:./bin
vt100
```

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

Implant Termination

The implant has mechanisms to terminate itself if there is an error or 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.

Hunting Tactics

The Sandbox is built on Azure and runs Sysmon for Linux, based on SysinternalsEBPF. I have also used the Sysmon Config from the MSTIC Team to monitor events. Post capture, I have used Azure Sentinel for SIEM & SOAR.

Sample Azure Sentinel Query for Early Detection

```
//Stage 1
vimFileEventLinuxSysmonFileCreated
| where Process == "/dev/shm/kdmtmpflush"
| where FilePath == "/run/haldrund.pid"
```

Possible binary left behind if the implant fails to load:

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

Dropper if implant active or did not clean up:

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

A deleted process called *kdmtmpflush* or similar.

Processes missing environment variables.

Any process running from */dev/shm*

Any process is operating with a packet socket open that you don't recognise 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 unknown 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.

Indicators of Compromise

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.

Filenames

These filenames can be used as IOCs because, while looking legitimate, they are used only by this malware family.

- /dev/shm/kdumpflush
- /dev/shm/kdumpdb
- /var/run/xinetd.lock
- /var/run/kdevrund.pid
- /var/run/haldrund.pid
- /var/run/syslogd.reboot

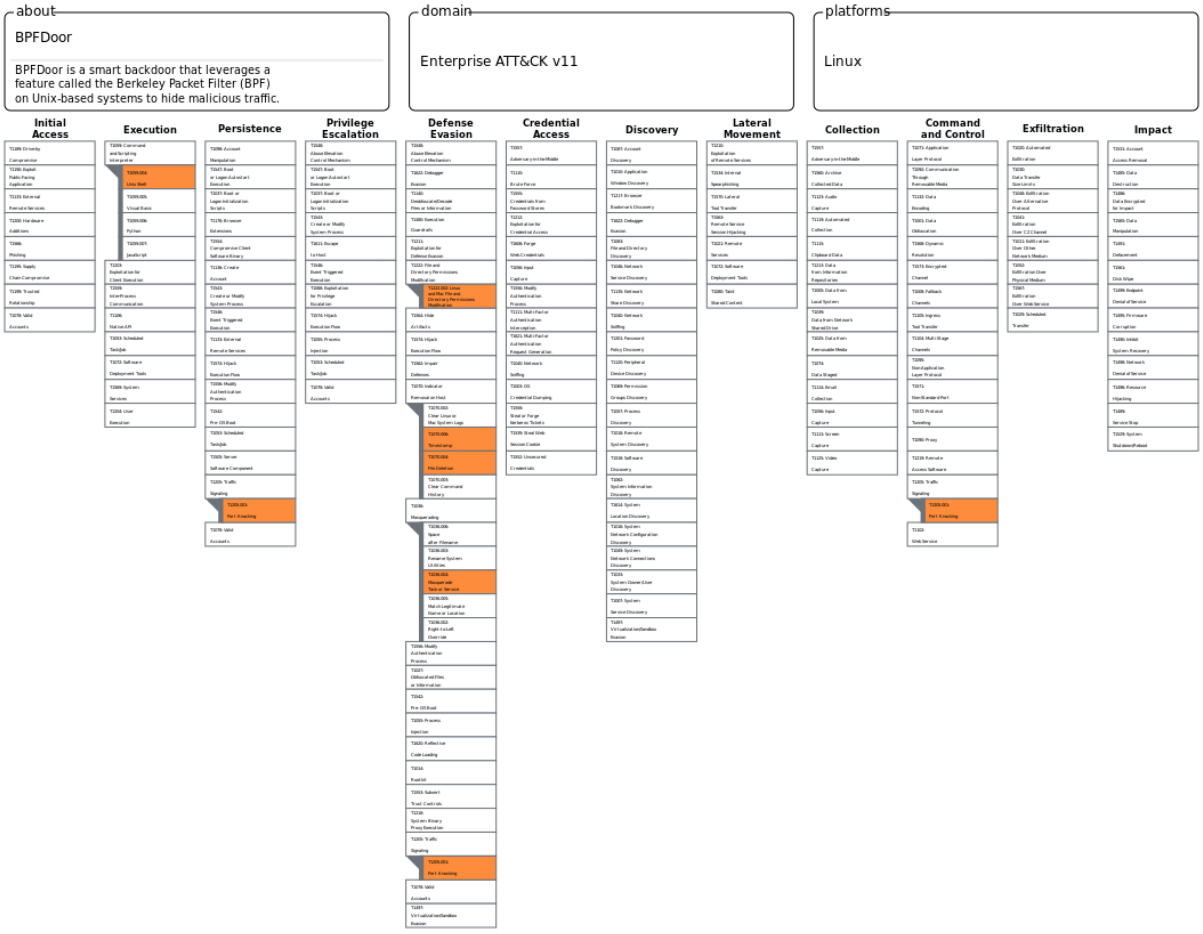
Hashes

(Kevin Beaumont maintains a good collection of BPFDoor hashes in VirusTotal)

07ecb1f2d9ffbd20a46cd36cd06b022db3cc8e45b1ecab62cd11f9ca7a26ab6d
1925e3cd8a1b0bba0d297830636cdb9ebf002698c8fa71e0063581204f4e8345
4c5cf8f977fc7c368a8e095700a44be36c8332462c0b1e41bff03238b2bf2a2d
591198c234416c6ccbcea6967963ca2ca0f17050be7eed1602198308d9127c78
599ae527f10ddb4625687748b7d3734ee51673b664f2e5d0346e64f85e185683
5b2a079690efb5f4e0944353dd883303ffd6bab4aad1f0c88b49a76ddcb28ee9
5faab159397964e630c4156f8852bcc6ee46df1cdd8be2a8d3f3d8e5980f3bb3
76bf736b25d5c9aaf6a84edd4e615796fffc338a893b49c120c0b4941ce37925
93f4262fce8c6b4f8e239c35a0679fbbbb722141b95a5f2af53a2bcafe4edd1c
96e906128095dead57fdc9ce8688bb889166b67c9a1b8fdb93d7cff7f3836bb9
97a546c7d08ad34dfab74c9c8a96986c54768c592a8dae521ddcf612a84fb8cc
c796fc66b655f6107eacbe78a37f0e8a2926f01fecebd9e68a66f0e261f91276
c80bd1c4a796b4d3944a097e96f384c85687daeedcdcf05cc885c8c9b279b09c
f47de978da1dbfc5e0f195745e3368d3ceef034e964817c66ba01396a1953d72
f8a5e735d6e79eb587954a371515a82a15883cf2eda9d7ddb8938b86e714ea27
fa0defdabd9fd43fe2ef1ec33574ea1af1290bd3d763fdb2bed443f2bd996d73
fd1b20ee5bd429046d3c04e9c675c41e9095bea70e0329bd32d7edd17ebaf68a
144526d30ae747982079d5d340d1ff116a7963aba2e3ed589e7ebc297ba0c1b3
fa0defdabd9fd43fe2ef1ec33574ea1af1290bd3d763fdb2bed443f2bd996d73
76bf736b25d5c9aaf6a84edd4e615796fffc338a893b49c120c0b4941ce37925
96e906128095dead57fdc9ce8688bb889166b67c9a1b8fdb93d7cff7f3836bb9
c80bd1c4a796b4d3944a097e96f384c85687daeedcdcf05cc885c8c9b279b09c
f47de978da1dbfc5e0f195745e3368d3ceef034e964817c66ba01396a1953d72
07ecb1f2d9ffbd20a46cd36cd06b022db3cc8e45b1ecab62cd11f9ca7a26ab6d
4c5cf8f977fc7c368a8e095700a44be36c8332462c0b1e41bff03238b2bf2a2d
599ae527f10ddb4625687748b7d3734ee51673b664f2e5d0346e64f85e185683
5b2a079690efb5f4e0944353dd883303ffd6bab4aad1f0c88b49a76ddcb28ee9
5faab159397964e630c4156f8852bcc6ee46df1cdd8be2a8d3f3d8e5980f3bb3
93f4262fce8c6b4f8e239c35a0679fbbbb722141b95a5f2af53a2bcafe4edd1c
97a546c7d08ad34dfab74c9c8a96986c54768c592a8dae521ddcf612a84fb8cc
c796fc66b655f6107eacbe78a37f0e8a2926f01fecebd9e68a66f0e261f91276

f8a5e735d6e79eb587954a371515a82a15883cf2eda9d7ddb8938b86e714ea27
fd1b20ee5bd429046d3c04e9c675c41e9095bea70e0329bd32d7edd17ebaf68a

MITRE ATT&CK Mapping



- T1059.004: Command and Scripting Interpreter: Unix Shell
- T1070.004: Indicator Removal on Host: File Deletion
- T1036/004: Masquerading: Masquerade Task or Service
- T1070/006: Indicator Removal on Host: Timestomp
- T1222.002: Linux and Mac File and Directory Permissions Modification
- T1205.001: Traffic Signaling

Conclusion

This implant is well-executed, and layers knew tactics such as environment anti-forensics, timestomping, and process masquerading effectively. 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 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 was done by someone that knows what they are doing with an intent to remain undetected.

References

- cd00r https://packetstormsecurity.com/files/22121/cd00r.c.html
- The Bvp47 - a Top-tier Backdoor of US NSA Equation Group
- https://www.pangulab.cn/en/post/the_bvp47_a_top-tier_backdoor_of_us_nsa_equation_group/
- https://vms.drweb.com/virus/?i=21004786
- Knock Knock! Who's There? - An NSA VM https://reverse.put.as/2021/12/17/knockknock-whos-there/
- Hive BPF https://twitter.com/vx_herm1t/status/1445773123668807680
- BPF OpCodes https://elixir.bootlin.com/linux/latest/source/include/uapi/linux/bpf_common.h
- BPF filters documentation https://lwn.net/Articles/593476/
- Ghidra https://ghidra-sre.org/
- The wrong way to filter sockets with BPF https://natanyellin.com/posts/ebpf-filteringdone-right/
- Tweet on Red Menshen https://twitter.com/GossiTheDog/status/1522000023092965376/
- Tinker Telco Soldier Spy https://troopers.de/troopers22/talks/7cv8pz/
- Hey Uroburos! What's up? https://exatrack.com/public/Uroburos_EN.pdf
- Title inspiration / Reverse OST https://www.discogs.com/release/11125255-Kronos-TheHellenic-Terro

1800 004 943
macquariecloudservices.com