# Fun with the new bpfdoor (2023)

unfinished.bike/fun-with-the-new-bpfdoor-2023

May 14, 2023

## unfinished.bike

May 14, 2023

I was recently provided a sample of the recently announced stealthier variant of bpfdoor, malware targeting Linux that is almost certainly a state-funded Chinese threat actor (Red Menshen). The sample analyzed was a8a32ec29a31f152ba20a30eb483520fe50f2dce6c9aa9135d88f7c9c511d7, detectable by 11 of 62 detectors on VirusTotal.

I was particularly curious what the bpfdoor surface area looked like, and if it was easy it was to detect using existing open-source tools and common Linux command-line utilities.

To experiment, I used my favorite VM manager on macOS or Linux for this analysis: Lima, with the default Ubuntu 22.10 image.

## Running bpfdoor as a regular user

I first ran bpfdoor as an unprivileged user to see what system calls would be executed:

```
strace -o /tmp/st.user -f ./x.bin
```

I've removed the less interesting lines of output, but the program does astonishingly little:

```
2655  execve("./x.bin", ["./x.bin"], 0x7fff9dad6ff8 /* 23 vars */) = 0
2655  openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
2655  openat(AT_FDCWD, "/var/run/initd.lock", O_RDWR|O_CREAT, 0666) = -1 EACCES
(Permission denied)
2655  flock(-1, LOCK_EX|LOCK_NB)        = -1 EBADF (Bad file descriptor)
2655  clone(child_stack=NULL, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD,
child_tidptr=0x7ff8d1b39a10) = 2656
2655  +++ exited with 0 +++
2656  close(0)                          = 0
2656  close(1)                          = 0
2656  close(2)                          = 0
2656  setsid()                          = 2656
2656  getrandom("\xa4\xd5\x9d\x71\xb3\xe0\x98\xe1", 8, GRND_NONBLOCK) = 8
2656  socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ALL)) = -1 EPERM (Operation not
permitted)
2656  exit_group(0)                     = ?
2656  +++ exited with 0 +++
```

The only noteworthy things here are:

- It tries to create /var/run/initd.lock but fails because it requires root
- It tries to set up a raw socket to listen to all protocols but fails because it requires root.
- It forks into the background via `clone()` and `setsid()`.

It's not unusual to see a bug with the flock() call to fd=-1 because openat() returned an error rather than a file handle.

## Running as root

```
2669  openat(AT_FDCWD, "/var/run/initd.lock", O_RDWR|O_CREAT, 0666) = 3
2669  flock(3, LOCK_EX|LOCK_NB)         = 0
2669  clone(child_stack=NULL, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD,
child_tidptr=0x7fb6d948ba10) = 3319
2669  exit_group(0 <unfinished ...>
3319  close(0 <unfinished ...>
2669  +++ exited with 0 +++
3319  close(1)                          = 0
3319  close(2)                          = 0
3319  setsid()                          = 3319
3319  getrandom("\x6c\x07\x1c\x75\x6b\xae\xfe\xdf", 8, GRND_NONBLOCK) = 8
3319  socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ALL)) = 0
3319  setsockopt(0, SOL_SOCKET, SO_ATTACH_FILTER, {len=30, filter=0x7ffd2270fa90},
16) = 0
3319  recvfrom(0,
"RUU\341\314\22RU\300\250\5\2\10\0E\0\0Lp\220\0\0@\6~\272\300\250\5\2\300\250"...,
65536, 0, NULL, NULL) = 90
3319  recvfrom(0,
"RUU\341\314\22RU\300\250\5\2\10\0E\0\0(p\221\0\0@\6~\335\300\250\5\2\300\250"...,
65536, 0, NULL, NULL) = 54
3319  recvfrom(0,
"RUU\341\314\22RU\300\250\5\2\10\0E\0\0Lp\222\0\0@\6~\270\300\250\5\2\300\250"...,
65536, 0, NULL, NULL) = 90
3319  recvfrom(0,
"RUU\341\314\22RU\300\250\5\2\10\0E\0\0(p\223\0\0@\6~\333\300\250\5\2\300\250"...,
65536, 0, NULL, NULL) = 54
```

First, it opens a lock, which works this time:

```
-rw-r--r-- 1 root root 0 May 13 12:45 /run/initd.lock
```

As mentioned in the bpfdoor analysis by deep instinct, we can see that it sets a BPF filter via `setsockopt()`, and loops waiting for the magic byte sequence: `\x44\x30\xCD\x9F\x5E\x14\x27\x66`.

One thing I find fascinating is how simple the initialization is: the previous iteration of bpfdoor did so much more in the name of “stealth”:

- copies itself to `/dev/shm`
- renaming itself in the process table via `prctl`
- deletes itself from disk
- timestomping

Red Menshen must have noticed that every method for achieving stealth is also a reliable detection method. So, the new bpfdoor keeps it simple by not trying to be stealthy. In fact, this binary does so little that it's suspicious. In 2023, most advanced evasion methods are not worth it on Linux: it is good enough to hide in plain sight.

## Detection

Using the `make detect` rule from osquery-detection-kit, I examined which existing rules would alert on the presence of the latest bpfdoor. 3 of them did:

- unexpected raw socket: unexpected packet sniffers, just like this one! Near-zero false-positive rate.
- recently created executables: programs executed within 45 seconds of when it likely landed on disk, based on ctime and btime. This catch-all has found every malware it's encountered, but it requires a comprehensive exception list.
- unexpected /var/run file: Inspired by reading the bpfdoor technical analysis, it's good to see this fired when faced with the real thing.

That said, I think we can do better. Let's see what the malware looks like from /proc.

## Exploring bpfdoor using /proc

To get an idea of what I can use for further detecting bpfdoor, I wanted to see how it was seen via /proc. First, what libraries does it link against? Based on the report, I'm not expecting anything other than libc:

```
% sudo cat /proc/3319/maps

00400000-00448000 r-xp 00000000 fc:01 3210                                /tmp/x.bin
00648000-00649000 r--p 00048000 fc:01 3210                                /tmp/x.bin
00649000-0064a000 rw-p 00049000 fc:01 3210                                /tmp/x.bin
0064a000-0066a000 rw-p 00000000 00:00 0
00c36000-00c57000 rw-p 00000000 00:00 0                                   [heap]
7fb6d9200000-7fb6d9222000 r--p 00000000 fc:01 3648
/usr/lib/x86_64-linux-gnu/libc.so.6
7fb6d9222000-7fb6d939b000 r-xp 00022000 fc:01 3648
/usr/lib/x86_64-linux-gnu/libc.so.6
7fb6d939b000-7fb6d93f2000 r--p 0019b000 fc:01 3648
/usr/lib/x86_64-linux-gnu/libc.so.6
7fb6d93f2000-7fb6d93f6000 r--p 001f1000 fc:01 3648
/usr/lib/x86_64-linux-gnu/libc.so.6
7fb6d93f6000-7fb6d93f8000 rw-p 001f5000 fc:01 3648
/usr/lib/x86_64-linux-gnu/libc.so.6
7fb6d93f8000-7fb6d9405000 rw-p 00000000 00:00 0
7fb6d948b000-7fb6d948e000 rw-p 00000000 00:00 0
7fb6d9495000-7fb6d9497000 rw-p 00000000 00:00 0
7fb6d9497000-7fb6d9498000 r--p 00000000 fc:01 3645
/usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7fb6d9498000-7fb6d94c1000 r-xp 00001000 fc:01 3645
/usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7fb6d94c1000-7fb6d94cb000 r--p 0002a000 fc:01 3645
/usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7fb6d94cb000-7fb6d94cd000 r--p 00034000 fc:01 3645
/usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7fb6d94cd000-7fb6d94cf000 rw-p 00036000 fc:01 3645
/usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7ffd226f0000-7ffd22711000 rw-p 00000000 00:00 0                           [stack]
7ffd22720000-7ffd22724000 r--p 00000000 00:00 0                           [vvar]
7ffd22724000-7ffd22726000 r-xp 00000000 00:00 0                           [vdso]
ffffffffff600000-ffffffffff601000 --xp 00000000 00:00 0                   [vsyscall]
```

What about open file handles?

```
% sudo lsof -p 3319

COMMAND  PID USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
x.bin   3319 root  cwd    DIR   0,52      200    9 /tmp/lima/osquery-defense-kit/out
x.bin   3319 root  rtd    DIR  252,1     4096    2 /
x.bin   3319 root  txt    REG  252,1   302576 3210 /tmp/x.bin
x.bin   3319 root  mem    REG  252,1  2072888 3648 /usr/lib/x86_64-linux-
gnu/libc.so.6
x.bin   3319 root  mem    REG  252,1   228720 3645 /usr/lib/x86_64-linux-gnu/ld-
linux-x86-64.so.2
x.bin   3319 root    0u  pack  33049      0t0  ALL type=SOCK_RAW
x.bin   3319 root    3u   REG   0,25        0 1322 /run/initd.lock
```

lsof is handy, but to see the raw socket from /proc, we need to do a little bit more digging:

```
# cat /proc/net/packet

sk               RefCnt Type Proto  Iface R Rmem   User   Inode
ffff92d346ba6800 3      3    88cc   2     1 0      100    19458
ffff92d34631d800 3      3    0003   0     1 241920 0      33089
```

The `Inode` field is misleading, but you can use it to find the associated process ID via:

```
$ sudo find /proc -type l -lname "socket:\[33089\]" 2>/dev/null

/proc/3319/task/3319/fd/0
/proc/3319/fd/0
```

Alternatively, you can use this to see all filehandles for the process ID:

```
$ ls -la /proc/3319/fd

total 0
dr-x------ 2 root root  0 May 13 13:03 .
dr-xr-xr-x 9 root root  0 May 13 13:03 ..
lrwx------ 1 root root 64 May 13 13:03 0 -> 'socket:[33089]'
lrwx------ 1 root root 64 May 13 13:03 3 -> /run/initd.lock
```

Once you have a process ID, you can resolve the path to the program:

```
sudo ls -lad /proc/3319/exe

lrwxrwxrwx 1 root root 0 May 14 00:48 /proc/3319/exe -> /tmp/x.bin
```

## Exploring bpfdoor using strings

Running `strings <path>` reveals some interesting messages:

```
[-] Execute command failed
/var/run/initd.lock
```

libtom/libtomcrypt has been bundled in, so we see lines such as:

```
LTC_ARGCHK '%s' failure on line %d of file %s
X.509v%i certificate
  Issued by: [%s]%s (%s)
  Issued to: [%s]%s (%s, %s)
  Subject: %s
  Validity: %s - %s
  OCSP: %s
  Serial number:
...
LibTomCrypt 1.17 (Tom St Denis, tomstdenis@gmail.com)
LibTomCrypt is public domain software.
Built on Oct  4 2022 at 16:09:32
```

That last string is important: this iteration of bpfdoor could have been wandering around Cyberspace since October 2022 (7 months ago) without detection. It also appears that the bad guys used Red Hat Enterprise Linux 7.0 (nearly 10 years old!) to build the binary:

```
GCC: (GNU) 4.8.5 20150623 (Red Hat 4.8.5-44)
```

## New detection possibilities

After looking at /proc, a couple of new detection ideas came up:

- Programs with /var/run lock files open
- Root processes with a socket and no shared libraries
- World-readable lock files in /var/run
- Minimalist socket users with few open files
- Processes where fd 0 is a non-UNIX socket

There are certainly more possibilities depending on how this backdoor is launched: for example, based on cwd or cgroup. I have not yet seen information published on how this backdoor is actually executed.

I implemented each of these detection ideas: once for osquery to use in production, and once in shell just for fun. The osquery queries have been tested across Ubuntu, Fedora, Arch Linux, and NixOS, and the shell scripts have only been tested on Ubuntu.

### Programs with /run lock files left open

It's unusual for a program to have an open file in /var/run, but I suspect this may eventually find a false positive. Here's an osquery and a shell script to find these:

```
SELECT p.* FROM processes p JOIN process_open_files pof ON p.pid = pof.pid AND
pof.path LIKE "/run/%.lock";
```

```
sudo find /proc -lname "/run/*.lock" 2>/dev/null
```

### Root processes with a socket and no shared libraries

Most programs that use a socket are either fully static, or import a library like OpenSSL. bpfdoor isn't either. Here is another osquery and shell pair:

```
SELECT p.*,
    COUNT(DISTINCT pmm.path) AS pmm_count
FROM processes p
    JOIN process_open_sockets pos ON p.pid = pos.pid
    LEFT JOIN process_memory_map pmm ON p.pid = pmm.pid
    AND pmm.path LIKE "%.so.%"
    -- Yes, this is a weird performance optimization
WHERE p.pid IN (
        SELECT pid
        FROM processes
        WHERE p.euid = 0
            AND p.path NOT IN (
                '/usr/bin/containerd',
                '/usr/bin/fusermount3',
                '/usr/sbin/acpid',
                '/usr/sbin/mcelog',
                '/usr/bin/docker-proxy'
            )
    )
GROUP BY pos.pid -- libc.so, ld-linux
HAVING pmm_count = 2;

cd /proc || exit

for pid in *; do
    [[ ! -f ${pid}/exe || ${pid} =~ "self" ]] && continue

    euid=$(grep Uid /proc/${pid}/status | awk '{ print $2 }')
    [[ "${euid}" != 0 ]] && continue

    sockets=$(sudo find /proc/${pid}/fd -lname "socket:*" | wc -l)
    [[ "${sockets}" == 0 ]] && continue

    libs=$(sudo find /proc/${pid}/map_files/ -type l -lname "*.so.*" -exec readlink
{} \; | sort -u | wc -l)
    [[ "${libs}" != 2 ]] && continue

    path=$(readlink /proc/$pid/exe)
    name=$(cat /proc/$pid/comm)
    echo "euid=0 process with sockets and no libs: ${name} [${pid}] at ${path}"
done
```

## World readable lock files in /var/run

Typically lock files are readable only by the root user. Malware often uses very relaxed file permissions.

```
SELECT * FROM file WHERE path LIKE "/tmp/%.lock" AND mode = "0644";

find /run/*.lock -perm 644
```

## Minimalist socket users with few open files

This creative query reveals minimalist programs that behave like a backdoor might:

- have 0-1 open files
- have 1-2 sockets open

It's an uncommon situation, but it is bound to have false positives in software that is designed in a way that each process has a specific role:

```
SELECT p.pid,
    p.path,
    p.name,
    p.start_time,
    GROUP_CONCAT(DISTINCT pos.protocol) AS protocols,
    pof.path AS pof_path,
    COUNT(DISTINCT pos.fd) AS scount,
    COUNT(DISTINCT pof.path) AS fcount,
    GROUP_CONCAT(DISTINCT pof.path) AS open_files,
    p.cgroup_path
FROM processes p
    JOIN process_open_sockets pos ON p.pid = pos.pid
    AND pos.protocol > 0
    LEFT JOIN process_open_files pof ON p.pid = pof.pid
WHERE p.start_time < (strftime('%s', 'now') -60)
AND p.path NOT IN (
    '/bin/registry',
    '/usr/bin/docker-proxy',
    '/usr/sbin/chronyd',
    '/usr/sbin/cups-browsed',
    '/usr/sbin/cupsd',
    '/usr/sbin/sshd'
)
AND p.path NOT LIKE '/nix/store/%-openssh-%/bin/sshd'
GROUP BY p.pid
HAVING scount <= 2
    AND fcount <= 1;
```

```
cd /proc || exit

for pid in *; do
    [[ ! -f ${pid}/exe || ${pid} =~ "self" ]] && continue

    fds=$(find /proc/${pid}/fd -lname "/*" | wc -l)
    [[ "${fds}" == 0 ]] && continue
    [[ "${fds}" -gt 1 ]] && continue

    # WARNING: ss -xp will print two fds on the same line if connected. Use grep -o
instead of -c
    #ss -xp | grep -v "^u_" | grep -o pid=${pid},"

    all_sockets=$(find /proc/${pid}/fd -lname "socket:*" | wc -l)
    [[ "${all_sockets}" -gt 2 ]] && continue

    # this isn't exactly what we want - ss doesn't show TYPE=sock of protocol=UNIX :(
    unix_sockets=$(ss -ap | grep "^u_" | grep -o "pid=${pid}," | wc -l)
    sockets=$(($all_sockets - $unix_sockets))

    [[ "${sockets}" == 0 ]] && continue
    [[ "${sockets}" -gt 2 ]] && continue

    path=$(readlink /proc/$pid/exe)
    [[ "${path}" == "/usr/sbin/sshd" ]] && continue

    name=$(cat /proc/$pid/comm)
    echo "minimalist socket user (${sockets} sockets and ${fds} files): ${name}
[${pid}] at ${path}"
done
```

## fd0 is a socket

I've saved my favorite for last. File descriptor 0 is usually stdin, but in bpfdoors case, it is actually the socket it uses to listen to traffic on. I've never seen this behavior before outside of bpfdoor:

```
SELECT * FROM process_open_sockets WHERE fd=0 AND family != 1;
```

```
cd /proc || exit

for pid in *; do
    [[ ! -f ${pid}/exe || ${pid} =~ "self" ]] && continue

    ino=$(readlink /proc/$pid/fd/0 | grep -o 'socket:.*' | cut -d"[" -f2 | cut -d"]"
-f1)
    grep -q " ${ino}" /proc/$pid/net/unix && continue

    path=$(readlink /proc/$pid/exe)
    name=$(cat /proc/$pid/comm)
    echo "fd0 is a socket: ${name} [${pid}] at ${path}"
done
```

## Final Thoughts

Ultimately, I was happy to see that this variant was detectable using osquery-defense-kit, and even happier that I could add additional rules to find future similar malware. Two philosophical viewpoints are critical to success in detection:

- Knowing what is considered normal in your environment
- Evasion is a means of detection

If you are interested in open-source queries that can find bpfdoor and other unusual programs, check out:

- https://github.com/chainguard-dev/osquery-defense-kit/
- https://github.com/tstromberg/sunlight

Thanks to Kevin Beaumont for providing the bpfdoor sample for analysis.