Detecting BPFDoor Backdoor Variants Abusing BPF Filters

trendmicro.com/en_us/research/23/g/detecting-bpfdoor-backdoor-variants-abusing-bpf-filters.html

July 13, 2023

Malware

An analysis of advanced persistent threat (APT) group Red Menshen's different variants of backdoor BPFDoor as it evolves since it was first documented in 2021.

By: Fernando Merces July 13, 2023 Read time: (words)

Advanced persistent threat (APT) groups have broadened their focus to include Linux and cloud servers in the past few years. Noticeable examples include ransomware groups targeting VMware ESXi servers, Mirai botnet variants, and groups targeting the cloud with stealers and cryptomining malware.

Similarly, APT groups have increased their presence on non-Windows targets. An example is Sandworm attacking routers shipped with Linux. While malware used by cybercrime usually has the broadest possible targets, malware used by APT groups are mostly about keeping and maintaining routines' stealth. Red Menshen (also known as DecisiveArchitect or Red Dev 18), an APT group targeting interests in the Middle East and Asian countries, has been constantly improving its BPFDoor backdoor over the years after it emerged in 2021. BPFDoor has since become more difficult to detect due to the improved usage of Berkeley Packet Filter (BPF), a technology that allows programs to attach network filters to an open socket that's being used by the threat actors behind BPFDoor to bypass firewalls' inbound traffic rules and similar network protection solutions in Linux and Solaris operating systems (OS). Trend Micro detects the Linux and Solaris BPFDoor versions as Backdoor.Linux.BPFDOOR and Backdoor.Solaris.BPFDOOR.ZAJE , respectively. Additional patterns related to the indicators have also been added for Trend products' monitoring and detection.

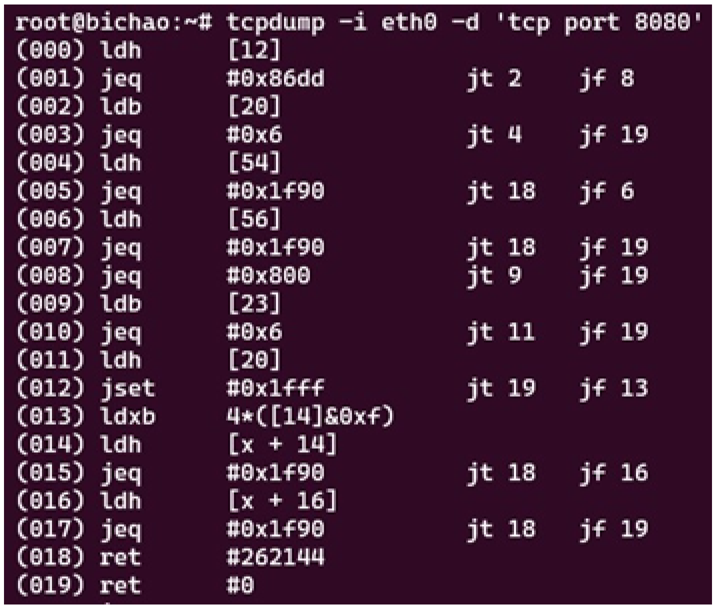
This entry shows how Red Menshen evolved their BPF filters with a six-fold increase in their BPF programs' instructions when compared to samples found in 2022. This is a clear sign that BPFDoor is under active development and that it has been proven successful enough for the attacks to merit a return on the malware developers' investment from this upgrade effort. In this entry, we also give security insights and suggest techniques for defenders to detect the presence of BPFDoor in infected systems.

What BPF is for

From a technical perspective, the most interesting feature of BPFDoor is its ability to load packet filters in the operating system's kernel. Although that mechanism is often called Berkeley Packet Filter (BPF), the Linux implementation equivalent is called Linux Socket Filtering (LSF). Nevertheless, under the Linux context, both terms refer to the same technology.

BPF/LSF is now considered a subset of eBPF (once called extended BPF, now it's not an acronym), a technology that supports programs to achieve a multitude of tasks and not only network packet filtering (which BPF does as its basic function). eBPF, in comparison, is akin to an abstract virtual machine (VM) that can run user-defined programs within a sandbox in the Linux kernel, much like running applications or software in a controlled environment. To differentiate BPF filters from eBPF programs, some literature refers to it as "Classic BPF" (cBPF).

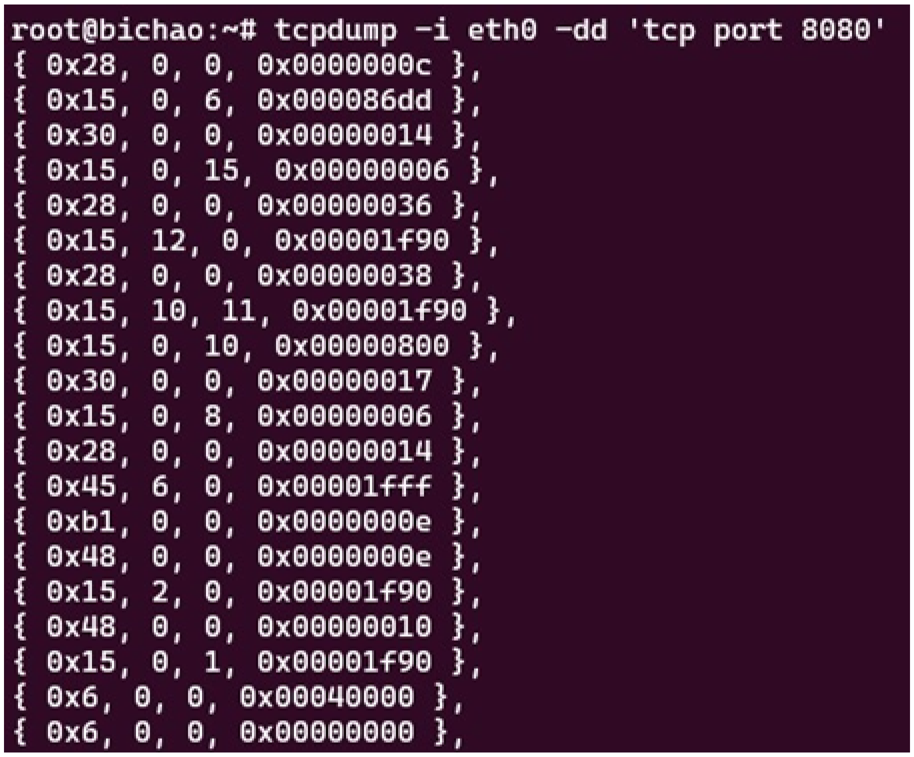
Classic BPF allows a running program to add a packet filtering rule to an open network socket. The program can then read from the socket and be informed when an arriving packet triggers the previously inserted rule. A library that heavily uses BPF in Linux is libpcap, which is used by programs such as tcpdump. In fact, you can even see the BPF filter generated by a libpcap filter expression with the *-d* option from tcpdump:



```
root@bichao:~# tcpdump -i eth0 -d 'tcp port 8080'
(000) ldh      [12]
(001) jeq      #0x86dd          jt 2    jf 8
(002) ldb      [20]
(003) jeq      #0x6             jt 4    jf 19
(004) ldh      [54]
(005) jeq      #0x1f90          jt 18   jf 6
(006) ldh      [56]
(007) jeq      #0x1f90          jt 18   jf 19
(008) jeq      #0x800           jt 9    jf 19
(009) ldb      [23]
(010) jeq      #0x6             jt 11   jf 19
(011) ldh      [20]
(012) jset     #0x1fff          jt 19   jf 13
(013) ldxb     4*([14]&0xf)
(014) ldh      [x + 14]
(015) jeq      #0x1f90          jt 18   jf 16
(016) ldh      [x + 16]
(017) jeq      #0x1f90          jt 18   jf 19
(018) ret      #262144
(019) ret      #0
```

Figure 1. BPF filter generated by tcpdump

The code shown in Figure 1 can be understood as a "BPF assembly." The kernel implements a virtual machine to understand this code. The bytecode that these instructions represent can also be seen with the *-dd* option:



```
root@bichao:~# tcpdump -i eth0 -dd 'tcp port 8080'
{ 0x28, 0, 0, 0x0000000c },
{ 0x15, 0, 6, 0x000086dd },
{ 0x30, 0, 0, 0x00000014 },
{ 0x15, 0, 15, 0x00000006 },
{ 0x28, 0, 0, 0x00000036 },
{ 0x15, 12, 0, 0x00001f90 },
{ 0x28, 0, 0, 0x00000038 },
{ 0x15, 10, 11, 0x00001f90 },
{ 0x15, 0, 10, 0x00000800 },
{ 0x30, 0, 0, 0x00000017 },
{ 0x15, 0, 8, 0x00000006 },
{ 0x28, 0, 0, 0x00000014 },
{ 0x45, 6, 0, 0x00001fff },
{ 0xb1, 0, 0, 0x0000000e },
{ 0x48, 0, 0, 0x0000000e },
{ 0x15, 2, 0, 0x00001f90 },
{ 0x48, 0, 0, 0x00000010 },
{ 0x15, 0, 1, 0x00001f90 },
{ 0x6, 0, 0, 0x00040000 },
{ 0x6, 0, 0, 0x00000000 },
```

Figure 2. Bytecode of BPF filter generated by tcpdump

As the second figure suggests, each classic BPF instruction is 8-bytes long. For further information on this, refer to the official Linux kernel documentation.

How BPFDoor works

The BPF filters used by BPFDoor allow the actors to activate the backdoor with a single network packet. Due to the way BPF is implemented in the targeted operating system, the magic packet triggers the backdoor even when the packet is blocked by a firewall. In fact, the packet reaches the kernel's BPF engine first, which is enough to activate the resident backdoor waiting for it. Similar features are common in rootkits but not easily found in backdoors.

BPFDoor samples load classic BPF filters into a running kernel. While the Linux samples load the compiled filters using the *SO_ATTACH_FILTER* option from *setsockopt()* syscall, the Solaris sample uses libpcap functions to compile and load the filter at runtime. The filters expect packets containing a magic number and, when it arrives, BPFDoor connects back to the source IP address of whoever sent the matching packet. In brief, magic numbers or magic constants are numeric literals with no explanation for their respective meanings used in the source code, or have a distinctive value that uniquely stand for specific identifiers.

The reverse connection is then used to send commands to the infected machine's shell via a pipe. In other words, BPFDoor opens a reverse shell that accepts pretty much any command remotely sent by the attacker. Because BPFDoor needs root privileges to work, the reverse shell it opens is also privileged.

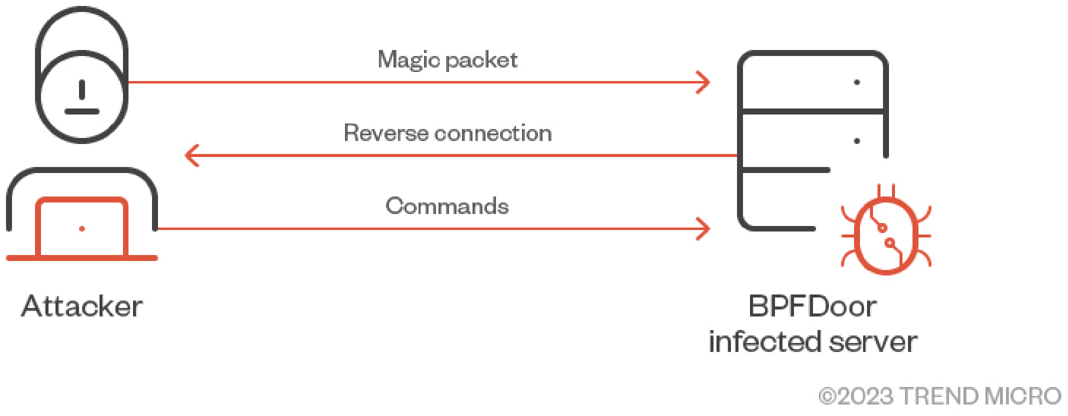
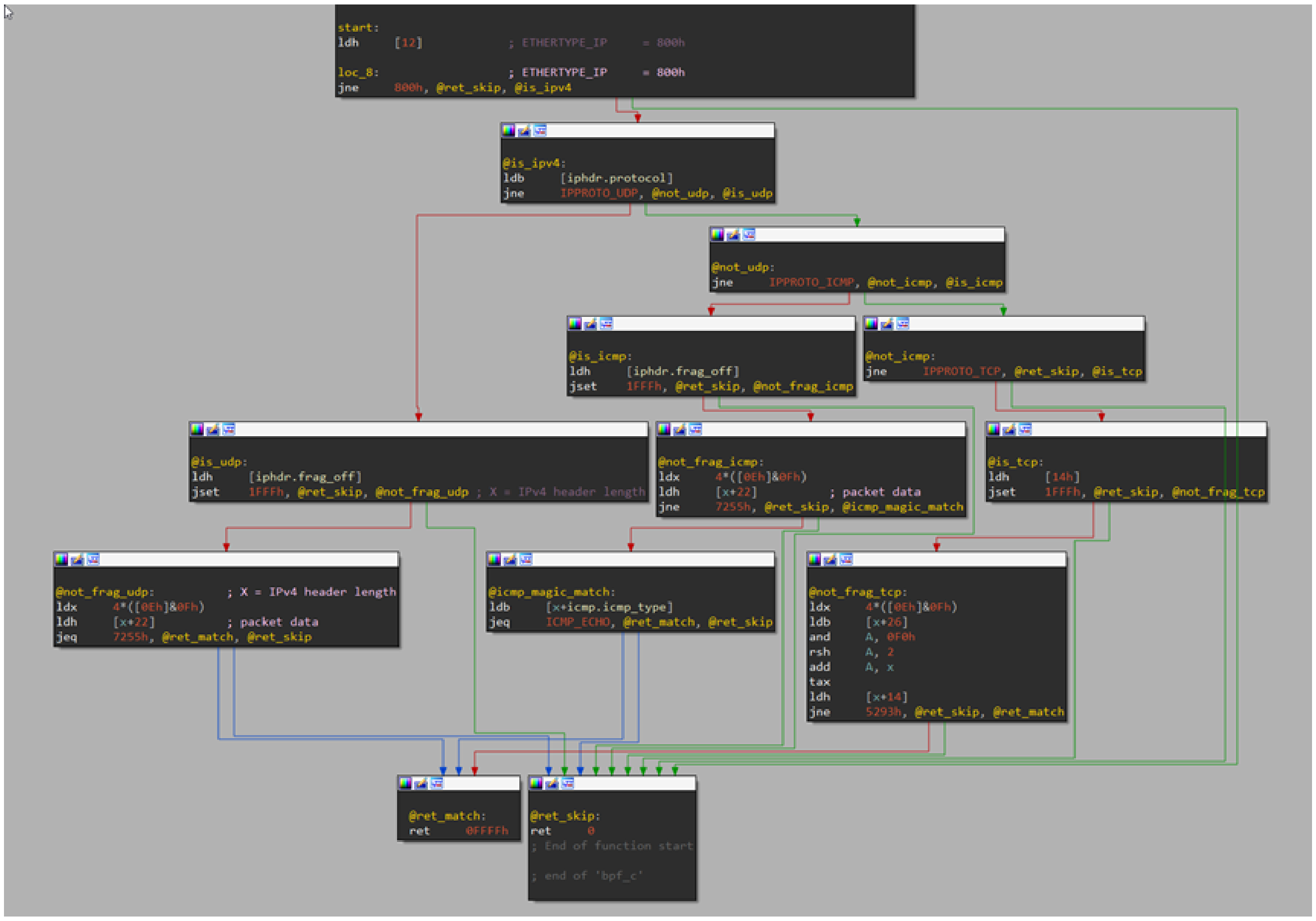


Figure 3. BPFDoor backdoor activation

Our analysis shows that different BPFDoor samples insert different filters expecting different magic numbers to initiate the reverse connection. The following sections show what we found in disassembling and analyzing BPF bytecode sets found in different BPFDoor samples.

Pre-2023 samples

Most BPFDoor samples from 2018 to 2022 contain the same BPF program that accepts specific magic numbers for TCP (Transmission Control Protocol), UDP (User Datagram Protocol), and ICMP (Internet Control Message Protocol) protocols. The following is the disassembled and analyzed BPF bytecode from these first samples, which we call Variant A:



```
start:
ldh     [12]             ; ETHERTYPE_IP     = 800h

loc_8:                   ; ETHERTYPE_IP     = 800h
jne     800h, @ret_skip, @is_ipv4
```

```
@is_ipv4:
ldb     [iphdr.protocol]
jne     IPPROTO_UDP, @not_udp, @is_udp
```

```
@not_udp:
jne     IPPROTO_ICMP, @not_icmp, @is_icmp
```

```
@is_icmp:
ldh     [iphdr.frag_off]
jset    1FFFh, @ret_skip, @not_frag_icmp
```

```
@not_icmp:
jne     IPPROTO_TCP, @ret_skip, @is_tcp
```

```
@is_udp:
ldh     [iphdr.frag_off]
jset    1FFFh, @ret_skip, @not_frag_udp ; X = IPv4 header length
```

```
@not_frag_icmp:
ldx     4*([0Eh]&0Fh)
ldh     [x+22]              ; packet data
jne     7255h, @ret_skip, @icmp_magic_match
```

```
@is_tcp:
ldh     [14h]
jset    1FFFh, @ret_skip, @not_frag_tcp
```

```
@not_frag_udp:           ; X = IPv4 header length
ldx     4*([0Eh]&0Fh)
ldh     [x+22]           ; packet data
jeq     7255h, @ret_match, @ret_skip
```

```
@icmp_magic_match:
ldb     [x+icmp.icmp_type]
jeq     ICMP_ECHO, @ret_match, @ret_skip
```

```
@not_frag_tcp:
ldx     4*([0Eh]&0Fh)
ldb     [x+26]
and     A, 0F0h
rsh     A, 2
add     A, x
tax
ldh     [x+14]
jne     5293h, @ret_skip, @ret_match
```

```
@ret_match:
ret     0FFFFh
```

```
@ret_skip:
ret     0
; End of function start

; end of 'bpf_c'
```

Figure 4. The 30-instruction BPF program

In total, the BPF program used by these samples has 30 BPF instructions. From here on, we'll use the number of BPF instructions to measure the filter complexity. The following is an equivalent filter using libpcap syntax:

> udp[8:2]=0x7255 or (icmp[8:2]=0x7255 and icmp[icmptype] == icmp-echo) or tcp[((tcp[12]&0xf0)>>2):2]=0x5293

To access the data field, the filter expression uses offsets. For UDP and ICMP, the data starts at octet 8. For TCP, the calculation is a bit more complex: the data offset is stored as a 4-bit value in offset 12. That's why analysts see an "AND" operation with 0xf0. By shifting this value four places to the right, we get the first 4-bit nibble at offset 12. However, this field contains the data offset as the number of 32-bit words, not bytes, so it needs to be multiplied by four to get the actual value in bytes. As multiplying by four is the same as shifting the bits two places to the left, both operations can be reduced to a single shift operation two places to the right. In other words,

> (x >> 4) * 4 == (x >> 4) << 2 == x >> 2

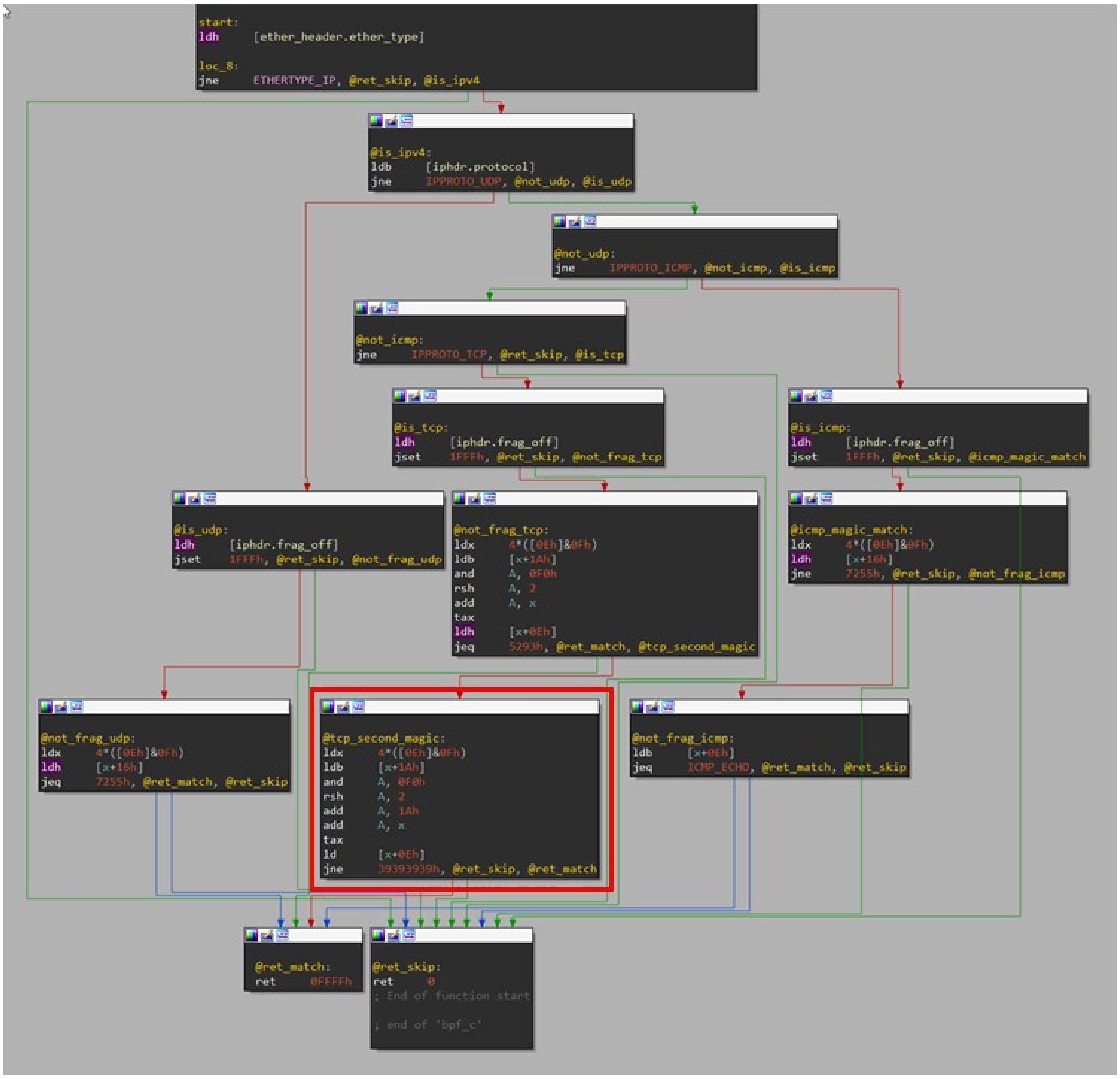
As the filter suggests, there are three different packets that activate the backdoor on an impacted system:

- UDP packet containing the magic number 0x7255 at the data field

- ICMP ECHO (ping) packet containing the same 0x7255 magic number at the data field
- TCP packet containing the magic number 0x5293 at the data field

BPFDoor filters in 2023

Using telfhash, we were able to source four samples that support an additional 4-byte magic number for TCP packets. The new BPF program contains 39 instructions. We're calling this program Variant B, and its disassembly is as follows:



```
start:
ldh     [ether_header.ether_type]

loc_8:
jne     ETHERTYPE_IP, @ret_skip, @is_ipv4

@is_ipv4:
ldb     [iphdr.protocol]
jne     IPPROTO_UDP, @not_udp, @is_udp

@not_udp:
jne     IPPROTO_ICMP, @not_icmp, @is_icmp

@not_icmp:
jne     IPPROTO_TCP, @ret_skip, @is_tcp

@is_tcp:
ldh     [iphdr.frag_off]
jset    1FFFh, @ret_skip, @not_frag_tcp

@is_icmp:
ldh     [iphdr.frag_off]
jset    1FFFh, @ret_skip, @icmp_magic_match

@is_udp:
ldh     [iphdr.frag_off]
jset    1FFFh, @ret_skip, @not_frag_udp

@not_frag_tcp:
ldx     4*([0Eh]&0Fh)
ldb     [x+1Ah]
and     A, 0F0h
rsh     A, 2
add     A, x
tax
ldh     [x+0Eh]
jeq     5293h, @ret_match, @tcp_second_magic

@icmp_magic_match:
ldx     4*([0Eh]&0Fh)
ldh     [x+16h]
jne     7255h, @ret_skip, @not_frag_icmp

@not_frag_udp:
ldx     4*([0Eh]&0Fh)
ldh     [x+16h]
jeq     7255h, @ret_match, @ret_skip

@tcp_second_magic:
ldx     4*([0Eh]&0Fh)
ldb     [x+1Ah]
and     A, 0F0h
rsh     A, 2
add     A, 1Ah
add     A, x
tax
ld      [x+0Eh]
jne     39393939h, @ret_skip, @ret_match

@not_frag_icmp:
ldb     [x+0Eh]
jeq     ICMP_ECHO, @ret_match, @ret_skip

@ret_match:
ret     0FFFFh

@ret_skip:
ret     0
; End of function start

; end of 'bpf_c'
```

Figure 5. 39-instruction BPF program

The nine additional instructions highlighted in the diagram add the abilities to activate the backdoor by a TCP packet containing the magic number 0x39393939 at a specific offset. This set of instructions might mean that the developers of BPFDoor wanted to have an additional way of activating the backdoor after its inner workings were detailed in a previously published article. An equivalent filter is as follows:

```
udp[8:2]=0x7255 or (icmp[8:2]=0x7255 and icmp[icmptype] == icmp-echo) or
tcp[((tcp[12]&0xf0)>>2):2]=0x5293 or tcp[((tcp[12]&0xf0)>>2)+26:4]=0x39393939
```

As the filter suggests, this new magic number should start at byte 26 in the TCP data. This means the first 26 bytes can be anything, which might be an attempt to make detection harder. It is also interesting to note that the previous magic numbers still work in this variant, which ensures compatibility with older versions of BPFDoor.

Doubtful feature: MAC address check

One BPFDoor sample uploaded to a public repository contained a BPF program containing 205 instructions, and we've called this sample Variant C. This is almost six times bigger than the previous BPF programs used by BPFDoor.

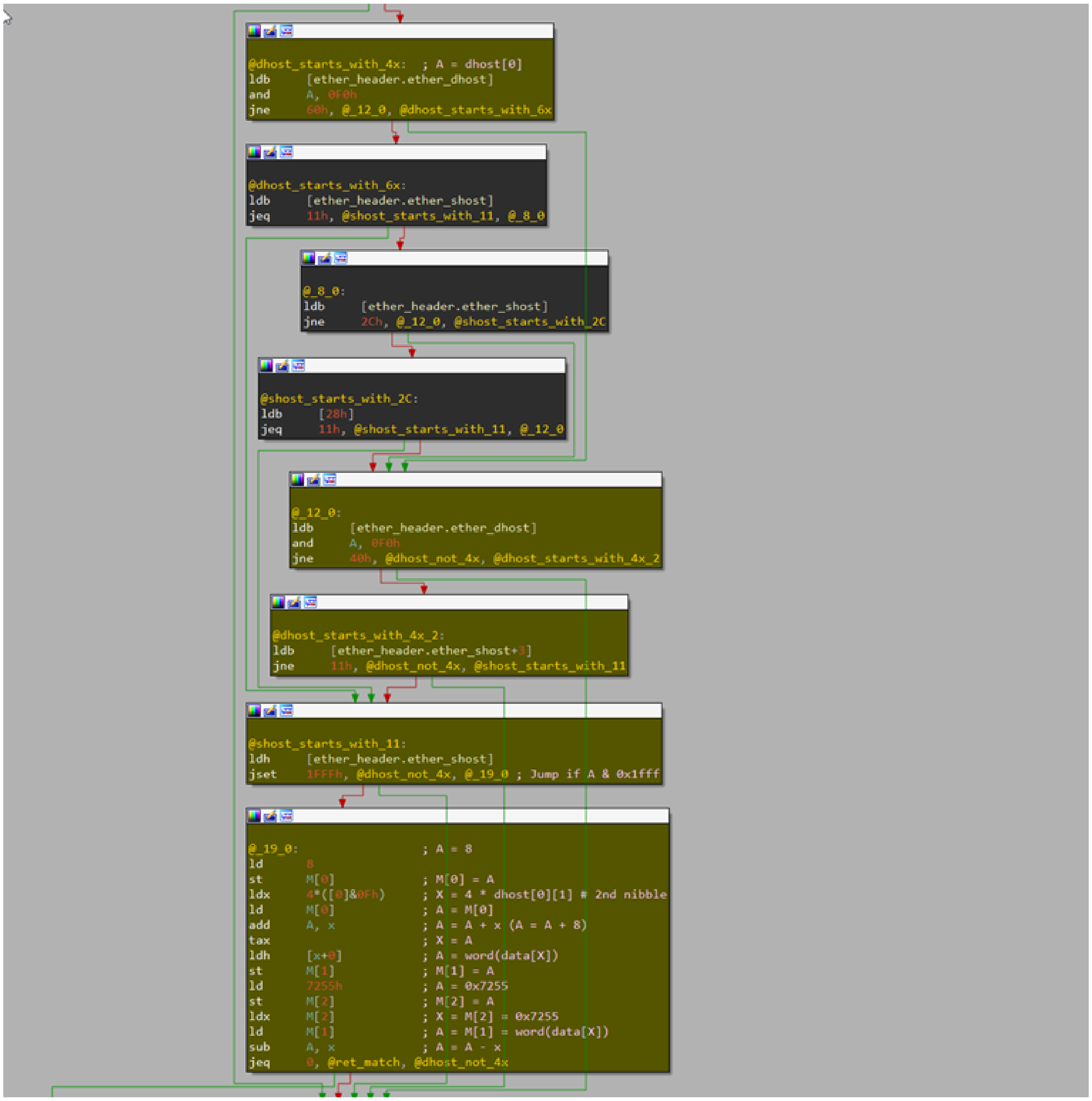
In this BFP program, the first 4-bit nibble of the packet's 48-bit (6 bytes) destination MAC addresses is checked. The backdoor seems to be activated only if this nibble is 0x4. In other words, if the destination MAC address starts with 0x4. This is achieved using the following BPF code:

```
l0:     ldb [0]                 # load the first byte of the packet to A register.
l1:     and A, #0xf0            # A = A & 0xf0 (bitwise AND).
l2:     jeq #0x40, l3, l33      # if the result is equals to 0x40, jump to location 3 (l3) and
continue execution.
```

The result will be 0x40 if the byte's highest nibble is 0x4. The program then uses the lowest nibble as an offset to locate the magic number. In our tests, the most feasible value for the lowest nibble is 0x3, which sets the program to look at the sixth byte from the data field.

However, we don't know whether this usage of the lowest nibble is intentional or not. One possibility is that the threat actors tried to come up with a BPF code to check for IPv4 and IPv6 packets, but ended up checking the destination MAC address by mistake. Another possibility is that the attackers tried to target machines whose network cards start with 0x4. As the first three bytes of MAC addresses serve as the Organizationally Unique Identifier (OUI), this would make the variant target victims with a particular NIC (network interface card) manufactured by a wide range of companies.

The possible magic numbers did not change compared to the 2022 samples, though. UDP and ICMP both accept 0x7255, while TCP accepts either 0x5293 or 0x39393939. There's also a code path for destination MAC addresses starting with 0x6 (or tentative to check for IPv6), but seems to be unreachable when checked. The following disassembly highlights the code path for a valid packet:



```
@dhost_starts_with_4x:  ; A = dhost[0]
ldb      [ether_header.ether_dhost]
and      A, 0F0h
jne      60h, @_12_0, @dhost_starts_with_6x
```

```
@dhost_starts_with_6x:
ldb      [ether_header.ether_shost]
jeq      11h, @shost_starts_with_11, @_8_0
```

```
@_8_0:
ldb      [ether_header.ether_shost]
jne      2Ch, @_12_0, @shost_starts_with_2C
```

```
@shost_starts_with_2C:
ldb      [28h]
jeq      11h, @shost_starts_with_11, @_12_0
```

```
@_12_0:
ldb      [ether_header.ether_dhost]
and      A, 0F0h
jne      40h, @dhost_not_4x, @dhost_starts_with_4x_2
```

```
@dhost_starts_with_4x_2:
ldb      [ether_header.ether_shost+3]
jne      11h, @dhost_not_4x, @shost_starts_with_11
```

```
@shost_starts_with_11:
ldh      [ether_header.ether_shost]
jset     1FFFh, @dhost_not_4x, @_19_0 ; Jump if A & 0x1fff
```

```
@_19_0:                       ; A = 8
ld       8
st       M[0]                 ; M[0] = A
ldx      4*([0]&0Fh)          ; X = 4 * dhost[0][1] # 2nd nibble
ld       M[0]                 ; A = M[0]
add      A, x                 ; A = A + x (A = A + 8)
tax                           ; X = A
ldh      [x+0]                ; A = word(data[X])
st       M[1]                 ; M[1] = A
ld       7255h                ; A = 0x7255
st       M[2]                 ; M[2] = A
ldx      M[2]                 ; X = M[2] = 0x7255
ld       M[1]                 ; A = M[1] = word(data[X])
sub      A, x                 ; A = A - x
jeq      0, @ret_match, @dhost_not_4x
```

Figure 6. First packet byte check in a 205-instruction BPF program

Three other samples from 2023 used an improved version of the above BPF program containing 229 instructions. This improved version also ensures the ICMP packet is from an ICMP ECHO request. We call it Variant D. Deep Instinct also revealed a 2023 sample that uses “44 30 CD 9F 5E 14 27 66” as the magic number, which we call Variant E.

Victims and detection

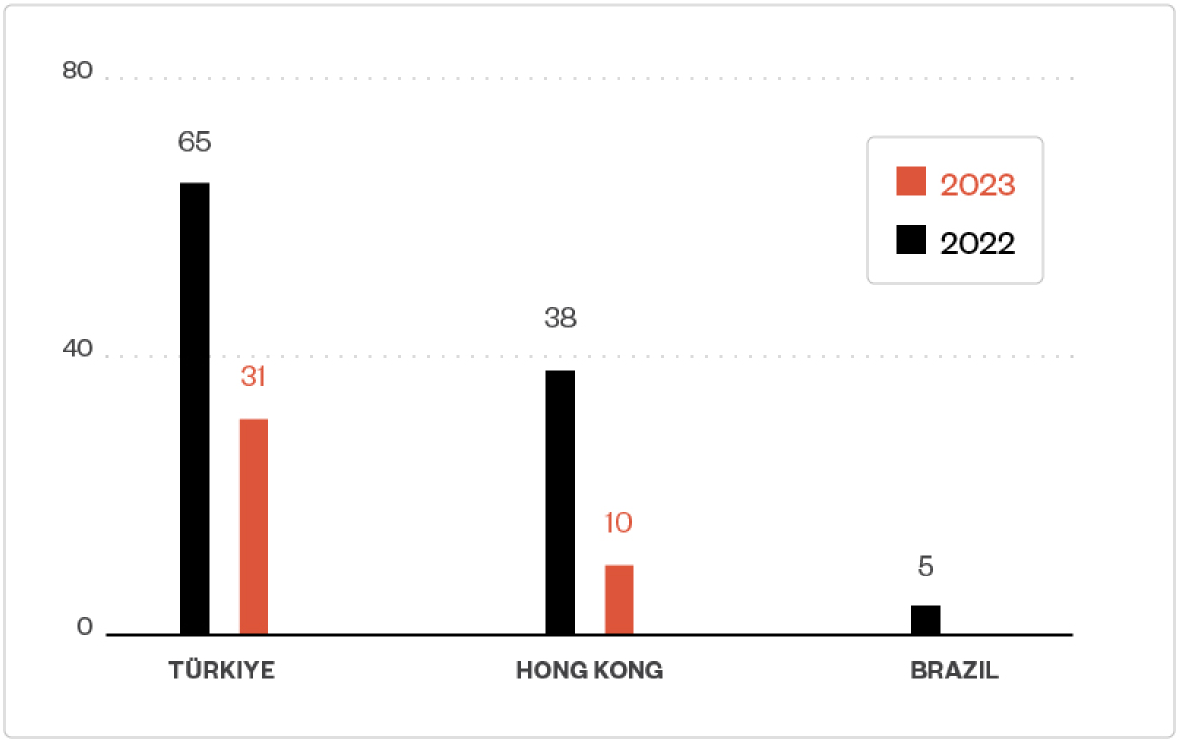


Figure 7. Countries targeted using BPFDoor

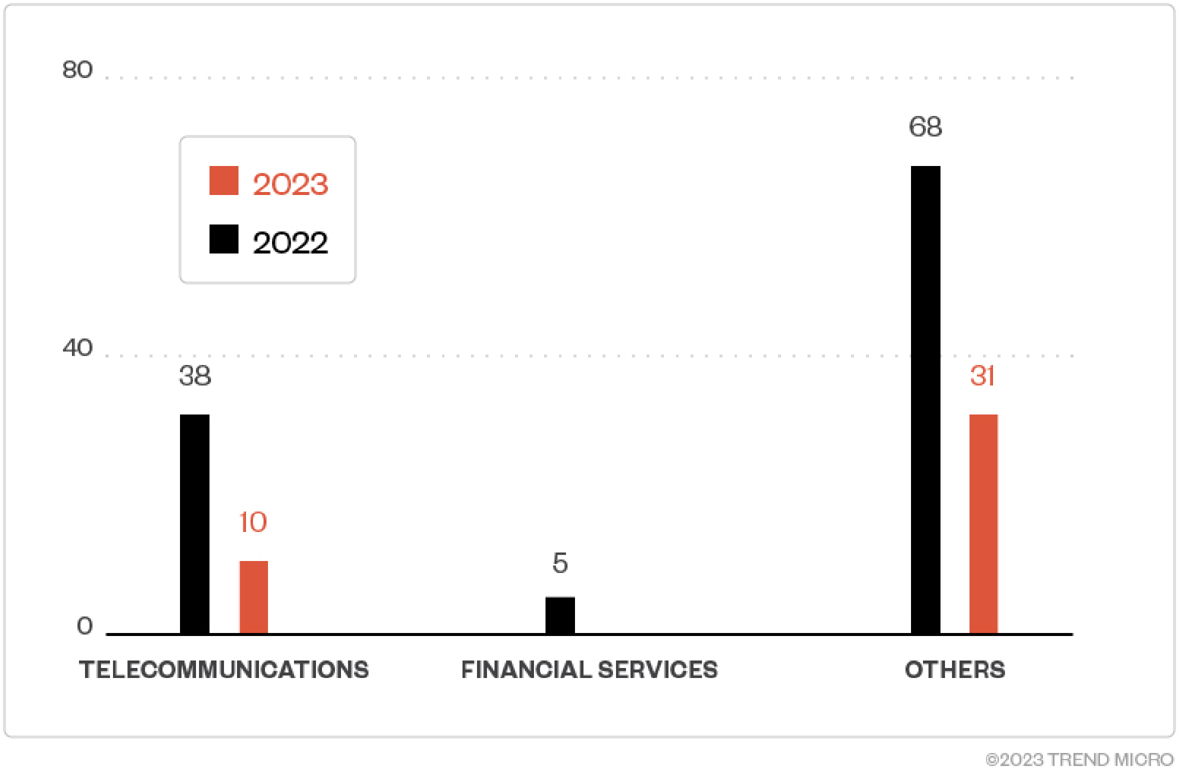
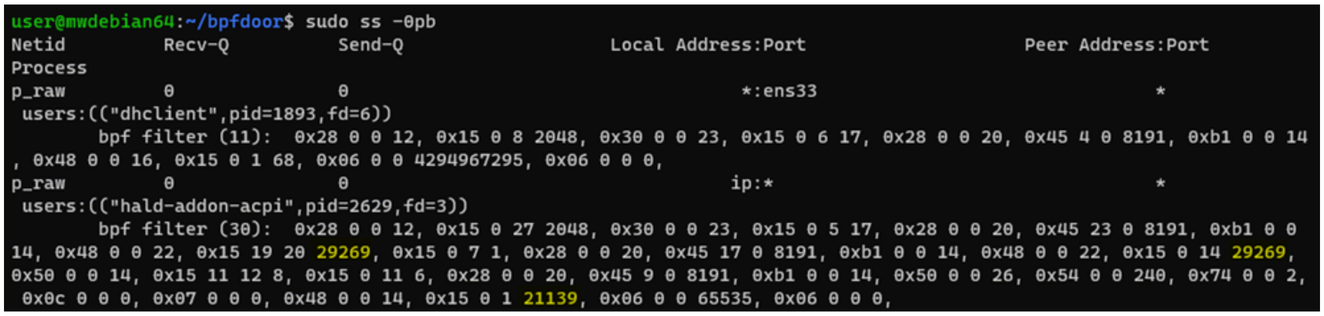


Figure 8. Industries targeted using BPFDoor

According to Trend Micro telemetry data, companies in the telecommunications sector in Türkiye and Hong Kong are being targeted by this threat actor. While we’re only halfway into 2023, there is a noticeable focus in unique detections for the said countries and industry compared to 2022. The data matches the initial report from PwC in 2022 pertaining to sectors, aside from “government, education, and logistics.” We recommend that defenders in these sectors check their servers carefully. One simple way of doing this is listing the running processes that inserted BPF filters in your Linux server with the **ss** command-line tool with the following parameters:

- -0 or --packet to display PACKET sockets
- -p or --processes to show the processes using such sockets
- -b or --bpf to dump the BPF programs attached to the sockets

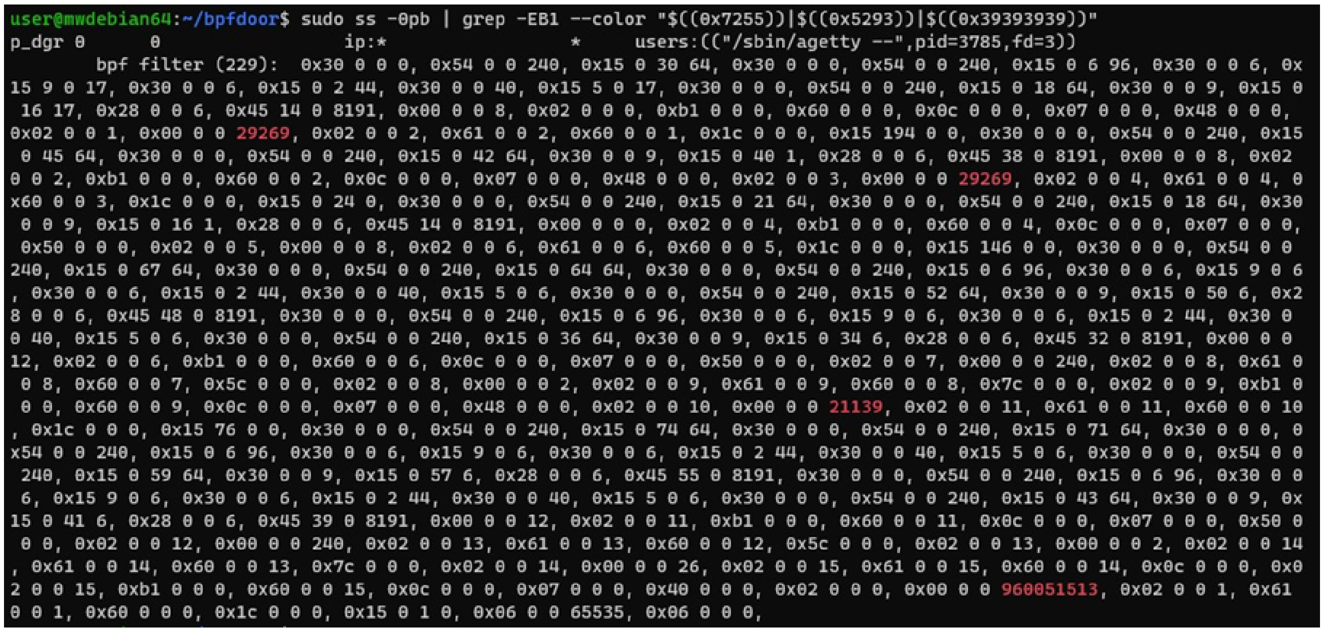
The following is an example of a Linux-based machine compromised by Variant A:



```
user@mwdebian64:~/bpfdoor$ sudo ss -0pb
Netid         Recv-Q          Send-Q                    Local Address:Port                    Peer Address:Port
Process
p_raw         0               0                                *:ens33                              *
 users:(("dhclient",pid=1893,fd=6))
        bpf filter (11):  0x28 0 0 12, 0x15 0 8 2048, 0x30 0 0 23, 0x15 0 6 17, 0x28 0 0 20, 0x45 4 0 8191, 0xb1 0 0 14
, 0x48 0 0 16, 0x15 0 1 68, 0x06 0 0 4294967295, 0x06 0 0 0,
p_raw         0               0                               ip:*                                  *
 users:(("hald-addon-acpi",pid=2629,fd=3))
        bpf filter (30):  0x28 0 0 12, 0x15 0 27 2048, 0x30 0 0 23, 0x15 0 5 17, 0x28 0 0 20, 0x45 23 0 8191, 0xb1 0 0
14, 0x48 0 0 22, 0x15 19 20 29269, 0x15 0 7 1, 0x28 0 0 20, 0x45 17 0 8191, 0xb1 0 0 14, 0x48 0 0 22, 0x15 0 14 29269,
0x50 0 0 14, 0x15 11 12 8, 0x15 0 11 6, 0x28 0 0 20, 0x45 9 0 8191, 0xb1 0 0 14, 0x50 0 0 26, 0x54 0 0 240, 0x74 0 0 2,
 0x0c 0 0 0, 0x07 0 0 0, 0x48 0 0 14, 0x15 0 1 21139, 0x06 0 0 65535, 0x06 0 0 0,
```

Figure 9. Listing processes that loaded BPF filters

Figure 9 shows two processes using BPF filters: *dhclient* (PID 1893) has a legitimate 11-instruction BPF program attached to it as indicated by the number between parenthesis. Meanwhile, *hald-addon-acpi* (PID 2629) has a suspicious 30-instruction BPF filter. We highlighted the magic numbers checked by the filter to trigger the backdoor (*29269* == *0x7255* and *21139* == *0x5293*). We note that security teams should not rely on the process name as this might change for every infection. Instead, analysts can also look specifically for the magic numbers. The following is an output from a machine infected with Variant D, which loads a BPF filter containing 229 instructions:



```
user@mwdebian64:~/bpfdoor$ sudo ss -0pb | grep -EB1 --color "$((0x7255))|$((0x5293))|$((0x39393939))"
p_dgr 0      0                  ip:*                    *     users:(("/sbin/agetty --",pid=3785,fd=3))
        bpf filter (229):  0x30 0 0 0, 0x54 0 0 240, 0x15 0 30 64, 0x30 0 0 0, 0x54 0 0 240, 0x15 0 6 96, 0x30 0 0 6, 0x
15 9 0 17, 0x30 0 0 6, 0x15 0 2 44, 0x30 0 0 40, 0x15 5 0 17, 0x30 0 0 0, 0x54 0 0 240, 0x15 0 18 64, 0x30 0 0 9, 0x15 0
 16 17, 0x28 0 0 6, 0x45 14 0 8191, 0x00 0 0 8, 0x02 0 0 0, 0xb1 0 0 0, 0x60 0 0 0, 0x0c 0 0 0, 0x07 0 0 0, 0x48 0 0 0,
0x02 0 0 1, 0x00 0 0 29269, 0x02 0 0 2, 0x61 0 0 2, 0x60 0 0 1, 0x1c 0 0 0, 0x15 194 0 0, 0x30 0 0 0, 0x54 0 0 240, 0x15
 0 45 64, 0x30 0 0 0, 0x54 0 0 240, 0x15 0 42 64, 0x30 0 0 9, 0x15 0 40 1, 0x28 0 0 6, 0x45 38 0 8191, 0x00 0 0 8, 0x02
0 0 2, 0xb1 0 0 0, 0x60 0 0 2, 0x0c 0 0 0, 0x07 0 0 0, 0x48 0 0 0, 0x02 0 0 3, 0x00 0 0 29269, 0x02 0 0 4, 0x61 0 0 4, 0
x60 0 0 3, 0x1c 0 0 0, 0x15 0 24 0, 0x30 0 0 0, 0x54 0 0 240, 0x15 0 21 64, 0x30 0 0 0, 0x54 0 0 240, 0x15 0 18 64, 0x30
 0 0 9, 0x15 0 16 1, 0x28 0 0 6, 0x45 14 0 8191, 0x00 0 0 0, 0x02 0 0 4, 0xb1 0 0 0, 0x60 0 0 4, 0x0c 0 0 0, 0x07 0 0 0,
 0x50 0 0 0, 0x02 0 0 5, 0x00 0 0 8, 0x02 0 0 6, 0x61 0 0 6, 0x60 0 0 5, 0x1c 0 0 0, 0x15 146 0 0, 0x30 0 0 0, 0x54 0 0
240, 0x15 0 67 64, 0x30 0 0 0, 0x54 0 0 240, 0x15 0 64 64, 0x30 0 0 0, 0x54 0 0 240, 0x15 0 6 96, 0x30 0 0 6, 0x15 9 0 6
, 0x30 0 0 6, 0x15 0 2 44, 0x30 0 0 40, 0x15 5 0 6, 0x30 0 0 0, 0x54 0 0 240, 0x15 0 52 64, 0x30 0 0 9, 0x15 0 50 6, 0x2
8 0 0 6, 0x45 48 0 8191, 0x30 0 0 0, 0x54 0 0 240, 0x15 0 6 96, 0x30 0 0 6, 0x15 9 0 6, 0x30 0 0 6, 0x15 0 2 44, 0x30 0
0 40, 0x15 5 0 6, 0x30 0 0 0, 0x54 0 0 240, 0x15 0 36 64, 0x30 0 0 9, 0x15 0 34 6, 0x28 0 0 6, 0x45 32 0 8191, 0x00 0 0
12, 0x02 0 0 6, 0xb1 0 0 0, 0x60 0 0 6, 0x0c 0 0 0, 0x07 0 0 0, 0x50 0 0 0, 0x02 0 0 7, 0x00 0 0 240, 0x02 0 0 8, 0x61 0
 0 8, 0x60 0 0 7, 0x5c 0 0 0, 0x02 0 0 8, 0x00 0 0 2, 0x02 0 0 9, 0x61 0 0 9, 0x60 0 0 8, 0x7c 0 0 0, 0x02 0 0 9, 0xb1 0
 0 0, 0x60 0 0 9, 0x0c 0 0 0, 0x07 0 0 0, 0x48 0 0 0, 0x02 0 0 10, 0x00 0 0 21139, 0x02 0 0 11, 0x61 0 0 11, 0x60 0 0 10
, 0x1c 0 0 0, 0x15 76 0 0, 0x30 0 0 0, 0x54 0 0 240, 0x15 0 74 64, 0x30 0 0 0, 0x54 0 0 240, 0x15 0 71 64, 0x30 0 0 0, 0
x54 0 0 240, 0x15 0 6 96, 0x30 0 0 6, 0x15 9 0 6, 0x30 0 0 6, 0x15 0 2 44, 0x30 0 0 40, 0x15 5 0 6, 0x30 0 0 0, 0x54 0 0
 240, 0x15 0 59 64, 0x30 0 0 9, 0x15 0 57 6, 0x28 0 0 6, 0x45 55 0 8191, 0x30 0 0 0, 0x54 0 0 240, 0x15 0 6 96, 0x30 0 0
 6, 0x15 9 0 6, 0x30 0 0 6, 0x15 0 2 44, 0x30 0 0 40, 0x15 5 0 6, 0x30 0 0 0, 0x54 0 0 240, 0x15 0 43 64, 0x30 0 0 9, 0x
15 0 41 6, 0x28 0 0 6, 0x45 39 0 8191, 0x00 0 0 12, 0x02 0 0 11, 0xb1 0 0 0, 0x60 0 0 11, 0x0c 0 0 0, 0x07 0 0 0, 0x50 0
 0 0, 0x02 0 0 12, 0x00 0 0 240, 0x02 0 0 13, 0x61 0 0 13, 0x60 0 0 12, 0x5c 0 0 0, 0x02 0 0 13, 0x00 0 0 2, 0x02 0 0 14
, 0x61 0 0 14, 0x60 0 0 13, 0x7c 0 0 0, 0x02 0 0 14, 0x00 0 0 26, 0x02 0 0 15, 0x61 0 0 15, 0x60 0 0 14, 0x0c 0 0 0, 0x0
2 0 0 15, 0xb1 0 0 0, 0x60 0 0 15, 0x0c 0 0 0, 0x07 0 0 0, 0x40 0 0 0, 0x02 0 0 0, 0x00 0 0 960051513, 0x02 0 0 1, 0x61
0 0 1, 0x60 0 0 0, 0x1c 0 0 0, 0x15 0 1 0, 0x06 0 0 65535, 0x06 0 0 0,
```

Figure 10. Highlighting BPFDoor magic numbers in ss command output

Conclusion

Embedding BPF bytecode in malware samples represents a new challenge for security teams, particularly for malware analysts and network defenders. Inspecting these programs is key to create accurate file and network traffic rules. But analysts' and security teams' work might not be trivial and have their work cut out for them in going through the samples, especially due to the lack of tools to analyze and debug BPF bytecode. The usage of BPF filters adds a certain layer of sophistication rarely seen in cybercrime, although it exists in addition to APT attacks.

One malware example where the abuse of BPF filters has been observed is Symbiote malware. Although BPF filters are not new, they are not heavily used by malware. Tools such IDA Pro and most disassemblers are not natively ready, but they can be extended with plugins. An exception is Radare2, an open-source framework that functions as a reverse engineering toolkit. The Linux kernel team also provides a few command line tools to deal with BPF filters, but can be further improved for analysts' use. Additionally, we are not aware of any malware training currently covering this. For that reason, we can consider it a "new" thing on the malware analysis side, and might be a consideration for malware and threat trainers to add in their respective programs.

The evolution of BPF filters used by BPFDoor also shows that threat actors are working to improve their methods to cover their tracks and keep the backdoor stealthy. We recommend network defenders to update their existing rules to reflect these changes, and for malware analysts to jump into BPF filter analysis as soon as possible.

Attacks involving BPFDoor can give the attackers complete access to the infected machine. Defenders should really pay attention to the BPF programs running in their environments.

Mitre ATT&CK techniques

As we focused on the BPF programs BPFDoor variants load, the only relevant ATT&CK framework technique is **Traffic Signaling (T1205)** and its sub-technique **Traffic Signaling: Socket Filters (T1205-002)**.

Indicators of compromise (IOCs)

Download the indicators here.

Magic numbers

The following table contains the magic numbers supported by different versions of the BPF filters analyzed:

Protocol	**Magic Number**	
	Hexadecimal	**Decimal**
UDP	0x7255	29 269
ICMP	0x7255	29 269
TCP	0x5293	21 139
TCP	0x39393939	960 051 513

The following Linux command can help defenders and security teams investigate suspicious BPF programs from checking the previously mentioned magic numbers:

```
ss -0pb | grep -EB1 --color "$((0x7255))|$((0x5293))|$((0x39393939))"
```