FrostyGoop’s Zoom-In: A Closer Look into the Malware Artifacts, Behaviors and Network Communications

Asher Davila, Chris Navarrete : : 11/19/2024

Executive Summary

In July 2024, the operational technology (OT)-centric malware FrostyGoop/BUSTLEBERM became publicly known, after attackers used it to disrupt critical infrastructure. The outage occurred after the Cyber Security Situation Center (CSSC), affiliated with the Security Service of Ukraine, disclosed details [PDF] of an attack on a municipal energy company in Ukraine in early 2024.

FrostyGoop is the ninth reported OT-centric malware, but the first that used Modbus TCP communications to impact the power supply to heating services for over 600 apartment buildings. FrostyGoop can be used both within a compromised perimeter and externally if the target device is accessible over the internet. FrostyGoop sends Modbus commands to read or modify data on industrial control systems (ICS) devices, causing damage to the environment where attackers installed it.

Based on this reporting, we conducted a deeper analysis and uncovered new samples of FrostyGoop and other related indicators. These new indicators include configuration files and libraries used by the malware, as well as artifacts associated with an infection. We also investigate network communications and provide new insights based on open-source intelligence (OSINT) data and our own telemetry.

OT malware is an increasing concern of security professionals across the globe, and FrostyGoop provides a notable case study of this growing threat.

Palo Alto Networks customers are better protected from the threats discussed in this article through our products and services such as the following:

- Industrial OT Security
- Information provided by Palo Alto Networks Next-Generation Firewall with Advanced Threat Prevention
- Advanced WildFire
- Cortex Xpanse
- Cortex XDR and Cortex XSIAM
- Prisma Cloud

If you think you might have been compromised or have an urgent matter, contact the Unit 42 Incident Response team.

Related Unit 42 Topics **JSON**, IoT Security, Russia

Technical Analysis of FrostyGoop

Attackers employed this malware associated with Russian actors in a cyberattack that caused a two-day heating system outage affecting over 600 apartment buildings in Ukraine, during sub-zero temperatures.

According to an open-source report, attackers made the initial compromise through a vulnerability in a MikroTik router. However, we have not confirmed this delivery method and bad actors might instead have delivered the malware via OT devices exposed to the internet.

FrostyGoop makes use of the Modbus TCP protocol to interact directly with ICS/OT devices, and therefore it is considered an ICS-centric malware. This is the ninth known ICS-centric malware.

In addition, Modbus is one of the most common protocols used in critical infrastructure. During this attack, the adversaries dispatched Modbus commands to ENCO control devices, leading to inaccurate measurements and system malfunctions. Remediating these issues took nearly two days.

Although bad actors used the malware to attack ENCO control devices, the malware can attack any other type of device that speaks Modbus TCP. Our telemetry indicates that 1,088,175 Modbus TCP devices were exposed to the internet from Sept. 2-Oct. 2, 2024, and 6,211,623 devices were exposed overall.

The details needed by FrostyGoop to establish a Modbus TCP connection and send Modbus commands to a targeted ICS device can be provided as command-line arguments or included in a separate JSON configuration file.

Malware Samples Analysis

FrostyGoop is compiled using the Go programming language, sometimes referred to as Golang. The malware uses a relatively obscure open-source Modbus implementation.

Further analysis of the Modbus library revealed this implementation does not natively support supplying arguments using a JSON file, making this a strong identifier for the malware. Moreover, the JSON object structure follows a specific format based on the commands this malware supports. FrostyGoop also contains capabilities for logging the output to a console or to a JSON file.

Attackers can supply two types of parameters to FrostyGoop:

- The first type of parameter consists of the possible operations an attacker can execute toward the registers of a Modbus device
- The second parameter consists of timing configurations.

Figure 1 shows an example of the first type of parameter for an operation using Tasks and Iplist under the register for main::main.TaskList___runtime.structtype_fields.

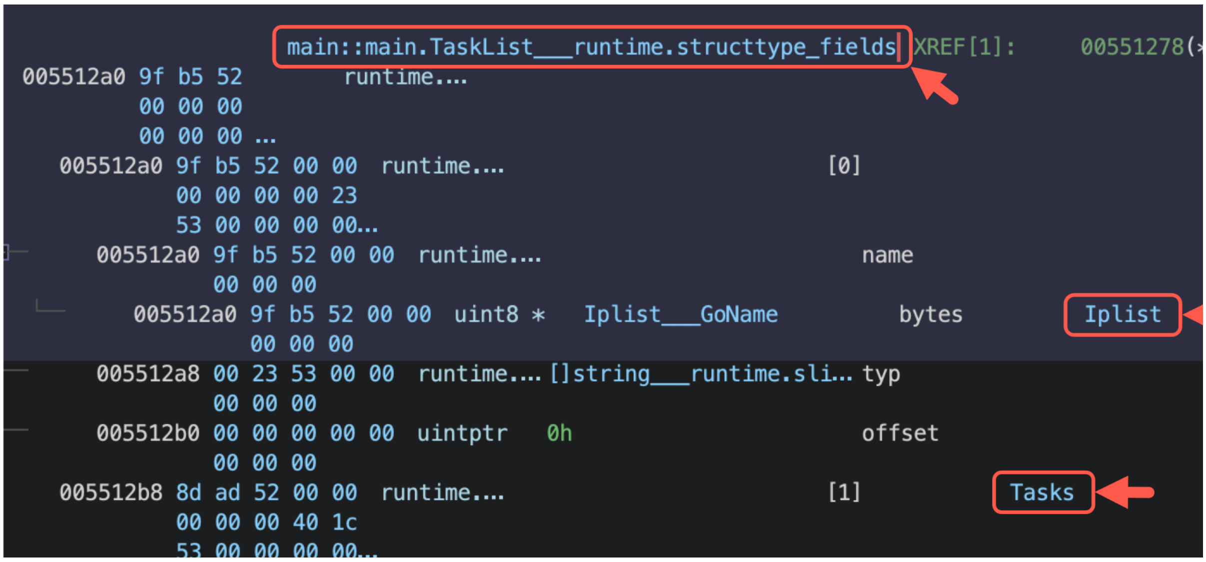


Figure 1. Binary Ninja showing FrostyGoop operations for Tasks and Iplist under the main::main.TaskList___runtime.structtype_fields register.

Figure 2 shows an example of an operation for Code, Address, Count, Value and State under the register for main::main.TaskList___runtime.structtype_fields.

```
    53 00 00 00 00...
  00553380 81 a0 52 00 00  runtime....                      name
           00 00 00
     00553380 81 a0 52 00 00  uint8 *   Code___GoName          bytes    Code
              00 00 00
  00553388 c0 3b 53 00 00  runtime....int___runtime._type   typ
           00 00 00
  00553390 00 00 00 00 00  uintptr   0h                     offset
           00 00 00
00553398 f7 bb 52 00 00  runtime....                      [1]
         00 00 00 c0 3b
         53 00 00 00 00...
  00553398 f7 bb 52 00 00  runtime....                      name
           00 00 00
     00553398 f7 bb 52 00 00  uint8 *   Address___GoName       bytes    Addres
              00 00 00
  005533a0 c0 3b 53 00 00  runtime....int___runtime._type   typ
           00 00 00
  005533a8 08 00 00 00 00  uintptr   8h                     offset
           00 00 00
005533b0 b1 ac 52 00 00  runtime....                      [2]
         00 00 00 c0 3b
         53 00 00 00 00...
  005533b0 b1 ac 52 00 00  runtime....                      name
           00 00 00
     005533b0 b1 ac 52 00 00  uint8 *   Count___GoName         bytes    Count
              00 00 00
  005533b8 c0 3b 53 00 00  runtime....int___runtime._type   typ
           00 00 00
  005533c0 10 00 00 00 00  uintptr   10h                    offset
           00 00 00
005533c8 a1 ad 52 00 00  runtime....                      [3]      Value
         00 00 00 c0 3b
         53 00 00 00 00...
005533e0 79 ad 52 00 00  runtime....                      [4]
         00 00 00 c0 3b
         53 00 00 00 00...
  005533e0 79 ad 52 00 00  runtime....                      name
           00 00 00
     005533e0 79 ad 52 00 00  uint8 *   State___GoName         bytes    State
              00 00 00
  005533e8 c0 3b 53 00 00  runtime....int___runtime._type   typ
           00 00 00
```

Figure 2. Binary Ninja showing FrostyGoop operations for Code, Address, Count, Value and State under the main::main.TaskList___runtime.structtype_fields register.

Figure 3 shows the timing configuration for main.Cycle.getCycleConfig.

```
layout.len = 8;
layout.str = (uint8 *)"15:04:05";
tVar22 = time::time.Parse(layout,local_90->StartTime);
tVar14 = time::time.ParseDuration(local_90->WorkTime);
tVar15 = time::time.ParseDuration(local_90->PeriodTime);
tVar16 = time::time.ParseDuration(local_90->IntervalTime);
```

Figure 3. Binary Ninja showing FrostyGoop timing configuration in the registry entry under main.Cycle.getCycleConfig(main.Cycle x,main.Cmd cmd).

FrostyGoop also leverages Goccy's go-json library, a faster JSON encoder and decoder compatible with the Go programming language standard encoding/json package. In addition, it incorporates a specific open-source execution controller named queues. The relative obscurity of this code means it can serve as another possible indicator of FrostyGoop.

Figure 4 shows our analysis of a Windows executable file for FrostyGoop within the tool Binary Ninja. This analysis reveals URLs from open-source libraries for modbus, go-json and queues.

```
2506  0x0019ab28 0x0059b528 23   24   .rdata ascii   github.com/rolfl/modbus
2537  0x0019ae2f 0x0059b82f 24   25   .rdata ascii   github.com/goccy/go-json
2565  0x0019b107 0x0059bb07 25   26   .rdata ascii   github.com/hsblhsn/queue
```
Figure 4. Open-source libraries: Modbus, go-json and queues.

Although not all FrostyGoop samples contain the strings shown in Figure 4, other strings contained within those libraries can serve as part of the detection for this malware.

FrostyGoop also implements a debugger evasion technique by checking the BeingDebugged value in Windows' Process Environment Block (PEB). Figure 5 shows this method in the disassembled code from a FrostyGoop sample. This method provides an alternative way to check the PEB's BeingDebugged flag without calling IsDebuggerPresent(). Attackers use this technique to detect and avoid debuggers used by malware analysts.

```
void sub_7ffc3ef12410(int32_t arg1 @ rax) __noreturn

00007ffc3ef12410  void sub_7ffc3ef12410(int32_t arg1 @ rax) __noreturn

00007ffc3ef12410  4053                push   rbx {var_8}
00007ffc3ef12412  4881ec90050000      sub    rsp, 0x590
00007ffc3ef12419  488364242800        and    qword [rsp+0x28 {var_570}], 0x0
00007ffc3ef1241f  bb01000000          mov    ebx, 0x1
00007ffc3ef12424  8364243800          and    dword [rsp+0x38 {var_560}], 0x0
00007ffc3ef12429  895c2424            mov    dword [rsp+0x24 {var_574}], ebx  {0x1}
00007ffc3ef1242d  894c2420            mov    dword [rsp+0x20 {var_578}], ecx

00007ffc3ef12431  448ac3              mov    r8b, bl
00007ffc3ef12434  488d9424c0000000    lea    rdx, [rsp+0xc0 {var_4d8}]
00007ffc3ef1243c  488d4c2420          lea    rcx, [rsp+0x20]
00007ffc3ef12441  e82af3f9ff          call   sub_7ffc3eeb1770
00007ffc3ef12446  65488b0c25600000…   mov    rcx, qword [gs:0x60]
00007ffc3ef1244f  80790200            cmp    byte [rcx+0x2 {_PEB::BeingDebugged}], 0x0
00007ffc3ef12453  7505                jne    0x7ffc3ef1245a

00007ffc3ef12455  80c3ff              add    bl, 0xff
00007ffc3ef12458  74d7                je     0x7ffc3ef12431

00007ffc3ef1245a  8bc8                mov    ecx, eax
00007ffc3ef1245c  e8afffffff          call   sub_7ffc3ef12410
```
Figure 5. Disassembled code from a FrostyGoop sample showing a check for the PEB's BeingDebugged flag.

Go-encrypt.exe Sample Analysis

Our investigation revealed a Windows executable sample named go-encrypt.exe written in Go that was not FrostyGoop, but it originally appeared on the same approximate date that other indicators of FrostyGoop were reported. Command-line options for this software reveal the file is used to encrypt and decrypt JSON files as illustrated in Figure 6.

```
C:\Users\Asher\Downloads>go-encrypt.exe -h
Usage of go-encrypt.exe:
  -decrypt
        true/false
  -encrypt
        true/false
  -input string
        input=[FILE.json]
  -output string
        output=[FILE.bin] (default "result.bin")

C:\Users\Asher\Downloads>
```
Figure 6. Command-line options for go-encrypt.exe.

After executing go-encrypt.exe using the -encrypt argument, it creates two files:

- An encrypted JSON
- A 32-byte file containing a decryption key named key

Figure 7 shows the encryption, decryption and the generated key.

```
C:\Users\Asher\Downloads>go-encrypt.exe -input task_test.json -output x.bin -encrypt true
2024/10/08 01:31:39 [runtime.main:proc.go:250][INFO] Key: %x
 [71 76 90 67 120 104 86 98 85 97 88 54 88 50 75 77 71 78 116 89 74 66 51 50 75 103 70 11
56 100 117 88]

C:\Users\Asher\Downloads>go-encrypt.exe -input x.bin -output clean.json -decrypt true
2024/10/08 01:31:54 [runtime.main:proc.go:250][INFO] Key: %x
 [71 76 90 67 120 104 86 98 85 97 88 54 88 50 75 77 71 78 116 89 74 66 51 50 75 103 70 11
56 100 117 88]
Decrypted file was created with file permissions 0777

C:\Users\Asher\Downloads>dir
 Volume in drive C has no label.
 Volume Serial Number is 0934-ECDE

 Directory of C:\Users\Asher\Downloads

10/08/2024  01:31 AM    <DIR>          .
10/08/2024  01:07 AM    <DIR>          ..
10/08/2024  01:31 AM               379 clean.json
10/08/2024  12:57 AM         1,773,568 go-encrypt.exe
10/08/2024  01:31 AM                32 key
10/08/2024  12:58 AM               379 task_test.json
10/08/2024  01:31 AM               524 x.bin
               5 File(s)      1,774,882 bytes
```

Figure 7. Using go-encrypt.exe to encrypt and decrypt a JSON file.

Figure 8 shows the content of an encrypted JSON file generated by go-encrypt.exe.

```
x.bin

Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Decoded text

00000000  49 A7 87 25 3E 23 07 B8 A8 29 4E A0 40 70 D1 25  I§‡%>#.¸¨)N @pÑ%
00000010  22 D3 AC D7 95 36 B3 4B 85 BC 28 C6 34 9C CC 56  "Ó¬×•6³K…¼(Æ4œÌV
00000020  DA CB 48 06 03 F0 76 EB 32 B9 6B 8F D5 74 9A B1  ÚËH..ðvë2¹k.Õtš±
00000030  F6 36 92 AF A3 20 D9 BF 2A 79 2C AE 1C 17 B4 65  ö6’¯£ Ù¿*y,®..´e
00000040  79 C1 6C 3A EA B4 47 49 E4 84 AB C9 36 27 17 5B  yÁl:ê´GIä„«É6'.[
00000050  2C 7A A9 8B 4D 9B B1 77 70 57 31 EA 25 33 73 05  ,z©‹M›±wpW1ê%3s.
00000060  1B 7A C5 92 DB 87 95 FD 59 5A 4B 20 51 89 3D 76  .zÅ’Û‡•ýYZK Q‰=v
00000070  A1 A6 80 FA A2 71 26 CB 16 22 E2 7B 44 7C 70 20  ¡¦€ú¢q&Ë."â{D|p 
00000080  98 B2 9B A7 8C 69 C6 4A 95 9E 81 44 B4 0E 24 82  ˜²›§ŒiÆJ•ž.D´.$‚
00000090  AF 1C 3E E9 19 5D 66 70 4B FE 35 4F 48 C9 4B 9B  ¯.>é.]fpKþ5OHÉK›
000000A0  43 AB C3 10 4F 11 F7 7D 34 3F 32 71 43 38 4E 7D  C«Ã.O.÷}4?2qC8N}
000000B0  98 9C 60 A8 DE C5 C4 59 23 B4 EA 6D 01 B3 1A B0  ˜œ`¨ÞÅÄY#´êm.³.°
000000C0  C8 96 EA 09 CF 6E E6 C1 E6 D8 7E FD A9 2C 91 00  È–ê.ÏnæÁæØ~ý©,‘.
000000D0  9E C7 93 55 0F 82 C9 98 40 0B F5 7E 47 B9 5A 29  žÇ“U.‚É˜@.õ~G¹Z)
000000E0  FE E3 F1 D6 08 31 44 EA EE 70 AF D3 D2 43 E6 36  þãñÖ.1Dêîp¯ÓÒCæ6
000000F0  4F AE D6 AF 69 A8 7C 8C 2E 6D 9F EE C8 D2 3F 58  O®Ö¯i¨|Œ.mŸîÈÒ?X
00000100  D4 DD 0D 98 5F 34 E6 69 54 52 E6 A2 88 97 7A F3  ÔÝ.˜_4æiTRæ¢ˆ—zó
00000110  2E B0 32 7B 27 0C 2A 4B A7 36 29 28 75 3E 89 32  .°2{'.*K§6)(u>‰2
00000120  9C 79 23 36 C3 DF C0 4D 10 87 31 AF 0E 28 8C E8  œy#6ÃßÀM.‡1¯.(Œè
00000130  6F 49 EC 2B 47 F3 39 FF 94 B8 C6 6C B8 01 75 99  oIì+Gó9ÿ”¸Æl¸.u™
00000140  29 F8 01 B6 89 02 1F 8C B8 11 6F 91 CF 98 C9 86  )ø.¶‰..Œ¸.o‘Ï˜É†
00000150  1A D3 37 FA 39 E4 06 9E B2 D6 76 BD AA 30 88 A5  .Ó7ú9ä.ž²Öv½ª0ˆ¥
00000160  44 2A BC 17 36 C6 CE F7 B7 EF A1 01 4A 37 1E 02  D*¼.6ÆÎ÷·ï¡.J7..
00000170  F8 3E A6 53 78 F7 1A C0 E6 5D 4B DB 77 BB AE 22  ø>¦Sx÷.Àæ]KÛw»®"
00000180  B7 FE F8 A5 60 B8 C6 64 EC CF 6C E6 23 58 3F 3D  ·þø¥`¸ÆdìÏlæ#X?=
00000190  30 2B 1D 63 CA EC 81 12 31 C1 9A 94 26 D3 B5 91  0+.cÊì..1Áš”&Óµ‘
000001A0  B8 E7 71 2B 1C 0E D4 C0 BC DD 3A 5B 38 C8 36 85  ¸çq+..ÔÀ¼Ý:[8È6…
000001B0  64 26 70 B4 F2 0F 3E 0D 41 AD CA D1 E4 3C 78 53  d&p´ò.>.A.ÊÑä<xS
000001C0  7B 69 93 A1 3D 64 7C C9 F0 41 A7 19 93 1C FB 31  {i“¡=d|ÉðA§.“.û1
000001D0  3C B2 A8 B0 11 6D DA CE 74 6A 35 4B 6E 61 BF 05  <²¨°.mÚÎtj5Kna¿.
000001E0  68 5D 3F 14 FE 79 AD 40 1B 7E 7D 9B 06 2F D2 D2  h]?.þy.@.~}›./ÒÒ
000001F0  F6 F5 4A 1B C4 66 DD 50 23 0F 2B 47 FA F1 BF 9F  öõJ.ÄfÝP#.+Gúñ¿Ÿ
00000200  AB 66 A7 5C 38 7A E0 39 EE 86 95 E3              «f§\8zà9î†•ã
```

Figure 8. An encrypted JSON file viewed in a hex editor.

Figure 9 shows a filtered list of processes generated by go-encrypt.exe in Process Monitor. We have highlighted when go-encrypt.exe created the decryption file named key and the 32 character content of this key file.

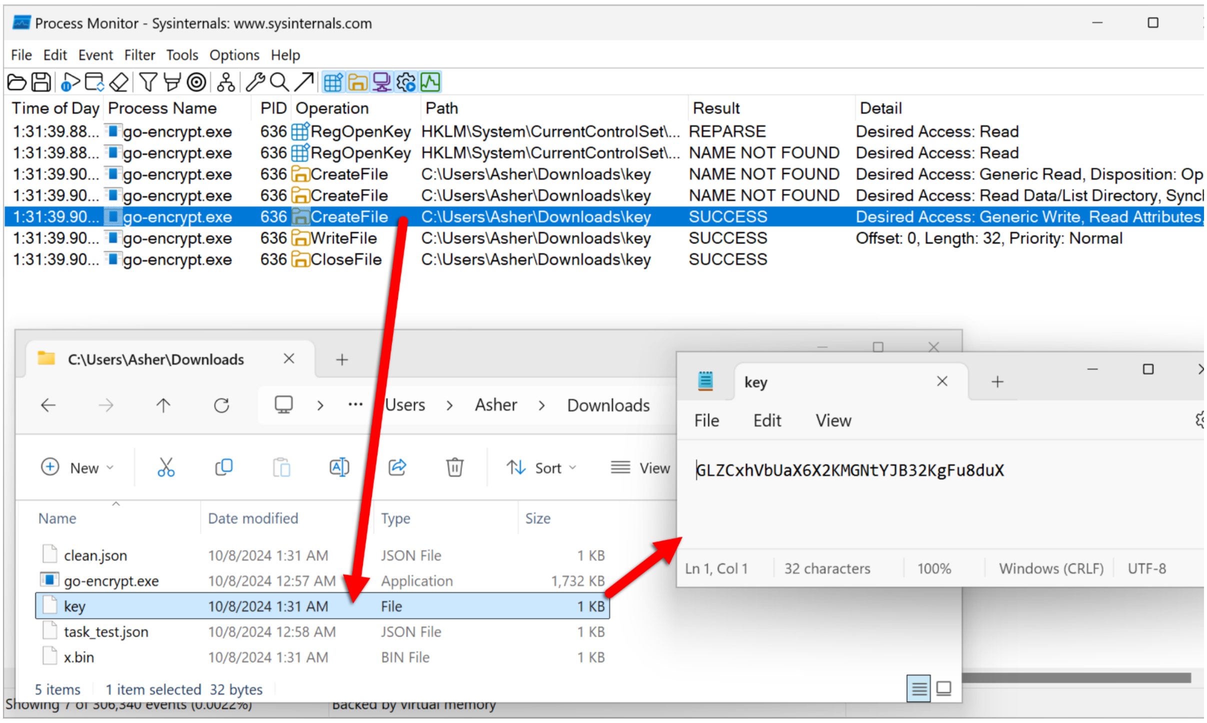


Figure 9. Process Monitor showing go-encrypt.exe generating the key file.

Decompiling go-encrypt.exe revealed it uses the Cipher Feedback (CFB) mode of the AES encryption algorithm to create the encryption/decryption key in the key file as shown in Figures 10 and 11.

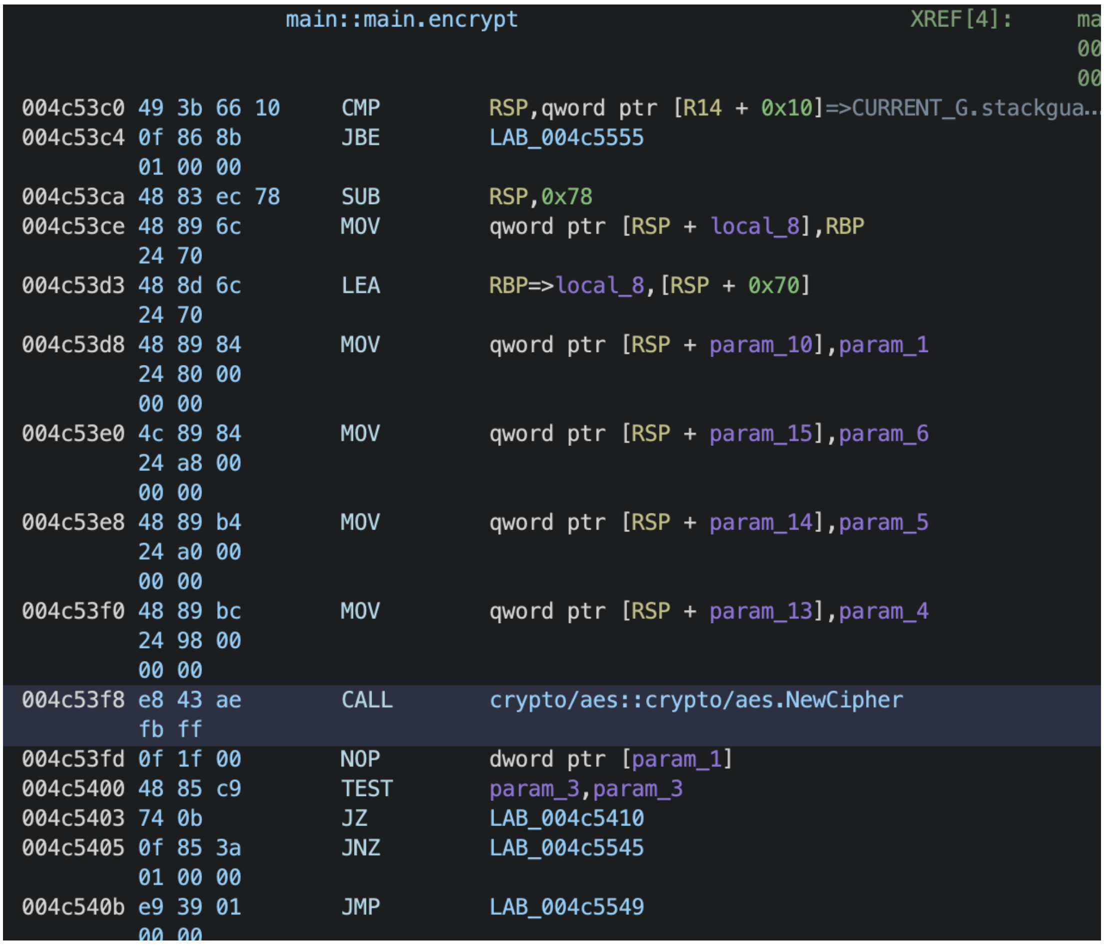


Figure 10. Decompiled code of go-encrypt.exe showing its AES main encryption routine.

Decompile: main.encrypt - (go-encrypt.exe)

```
  param_10 = param_1;
  param_14 = param_5;
  param_13 = param_4;
  param_15 = param_6;
  while (&stack0x00000000 <= CURRENT_G.stackguard0) {
    runtime::runtime.morestack_noctxt();
  }
  crypto/aes::crypto/aes.NewCipher();
  if (extraout_RCX != 0) {
    if (extraout_RCX != 0) {
      uVar3 = *(undefined8 *)(extraout_RCX + 8);
    }
    else {
      uVar3 = 0;
    }
                    /* WARNING: Subroutine does not return */
    runtime::runtime.gopanic(uVar3,extraout_RDI);
  }
  auVar5 = encoding/base64::encoding/base64.(*Encoding).EncodeToString(DAT_005ae5b0);
  runtime::runtime.stringtoslicebyte(0,auVar5._0_8_,auVar5._8_8_);
  uVar1 = extraout_RBX_00 + 0x10;
  [Var6 = runtime::runtime.makeslice();
  puVar2 = [Var6.array;
  if (uVar1 < 0x10) {
                    /* WARNING: Subroutine does not return */
    runtime::runtime.panicSliceAcap(puVar2,[Var6.len,0x10);
  }
  io::io.ReadAtLeast(DAT_005aeed0,DAT_005aeed8,puVar2,0x10,uVar1,0x10);
  if (extraout_RBX_01 == 0) {
    auVar5 = crypto/cipher::crypto/cipher.newCFB(extraout_RAX,extraout_RBX,puVar2,0x10,uVar1,0)
    (**(code **)(auVar5._0_8_ + 0x18))
              (auVar5._8_8_,puVar2 + ((dword)(-extraout_RBX_00 >> 0x3f) & 0x10),extraout_RBX_0
               extraout_RBX_00,extraout_RAX_00,extraout_RBX_00,extraout_RCX_00);
    return puVar2;
  }
  iVar4 = extraout_RBX_01;
  if (extraout_RBX_01 != 0) {
    iVar4 = *(int *)(extraout_RBX_01 + 8);
  }
                    /* WARNING: Subroutine does not return */
  runtime::runtime.gopanic(iVar4,extraout_RCX_01);
}
```

Figure 11. Decompiled code of go-encrypt.exe showing CFB mode.

As shown previously for the key generated in Figure 7, the key value is in decimal format. The decimal value of the key from Figure 7 is:

- 71 76 90 67 120 104 86 98 85 97 88 54 88 50 75 77 71 78 116 89 74 66 51 50 75 103 70 117 56 100 117 88

We can decode these decimal numbers into the 32-byte value of the key through a variety of methods, like the Python script shown in Figure 12. This script will convert the binary values to hexadecimal.

C: > temp > decoder.py > to_hex

```python
def to_hex(key_dec:list)->list:
    result = ' '.join('%02x'%i for i in key_dec)
    return result
```

Figure 12. Example of a Python script to convert the decimal value of the key to hexadecimal.

The 32-byte value of the key in hexadecimal is:

- 47 4c 5a 43 78 68 56 62 55 61 58 36 58 32 4b 4d 47 4e 74 59 4a 42 33 32 4b 67 46 75 38 64 75 58

According to Go's documentation for aes.NewCipher, a byte stream of 32 bytes corresponds to a 256-bit AES encryption in CFBmode. We confirmed the 32-byte hexadecimal value of the key from our example matches the ASCII value in the key file using CyberChef as shown below in Figure 13.

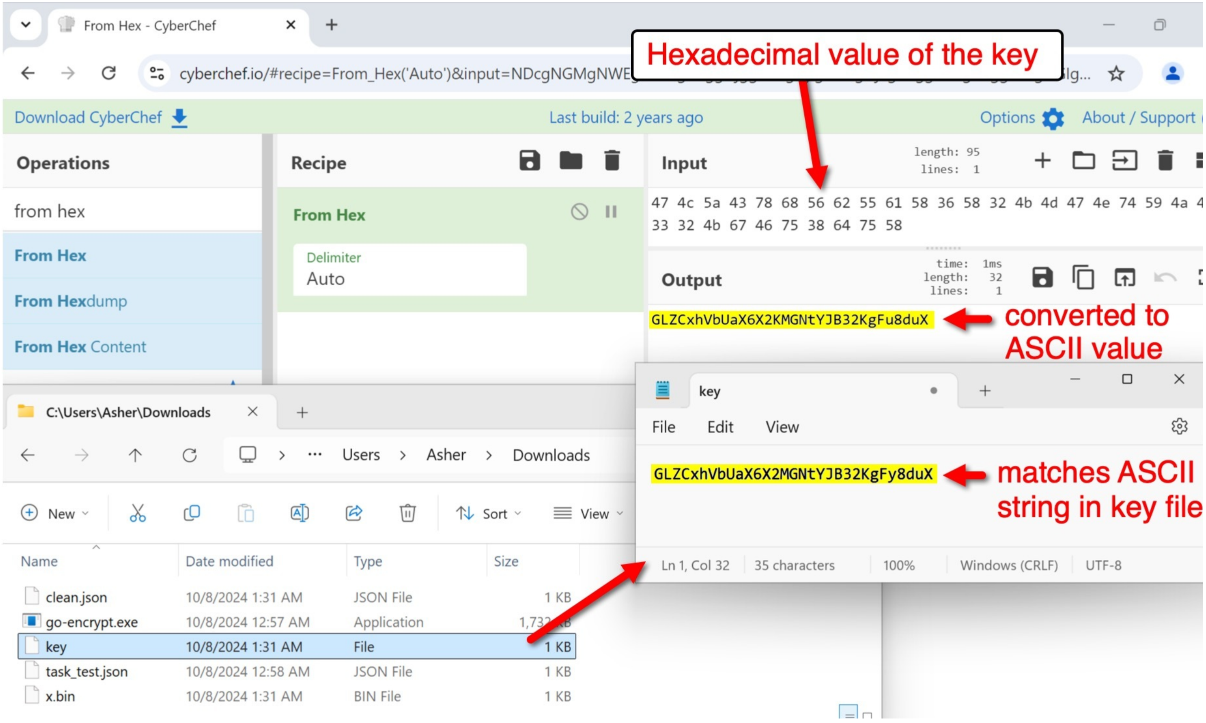


Figure 13. Using CyberChef to verify the hex value matches the ASCII value of the key file.

Although we cannot confirm go-encrypt.exe was used for the FrostyGoop attack, two circumstances indicate the attackers might have used it during this activity:

- First, it is used to encrypt and decrypt JSON files, and encrypted JSON files are an essential element of FrostyGoop functionality.
- Second, go-encrypt.exe first appeared in the wild around the same time as the FrostyGoop samples and the task_test.json file.

Therefore, attackers could have used this piece of software to conceal target information in JSON files for later use to perpetrate attacks.

Investigation of the Targeted Infrastructure

According to the Dragos report on FrostyGoop, they initially discovered this malware in April 2024. This report notes an example of a FrostyGoop configuration file named task_test.json.

Searching VirusTotal, we found one occurrence of task_test.json on Oct. 10, 2023. Pivoting on that file, we discovered Windows executable files that we subsequently identified as FrostyGoop and go-encrypt.exe.

Figure 14 shows the same first-seen date of Oct. 10, 2023, for task_test.json, go-encrypt.exe and the other Windows executable files.

FILES - 8

First seen desc | Sort by | Export | Tools

	Detections	Size	First seen	Last seen
5d2e4fd08f81e3b2eb2f3eaae16eb32ae02e760afc36fa17f4649322f6da53… 5d2e4fd08f81e3b2eb2f3eaae16eb32ae02e760afc36fa17f4649322f6da53fb.exe peexe idle 64bits	53 / 74	3.53 MB	2023-10-30 16:13:12	2024-09-06 10:38:41
2fd9cb69ef30c0d00a61851b2d96350a9be68c7f1f25a31f896082cfbf3955… c:\1002455031.exe peexe 64bits idle	48 / 75	3.20 MB	2023-10-30 16:09:31	2024-07-24 17:19:33
9cf30d82a86a9485f7bbd0786a5de207cf4902691a3efcfc966248cb1e87d5… go-encrypt.exe peexe 64bits	41 / 70	1.69 MB	2023-10-30 10:16:40	2023-10-30 10:16:40
c64b67c116044708e282d0d1a8caea2360270a7fc679befa5e28d1ca15f671… hello2.exe peexe 64bits idle	32 / 75	1.86 MB	2023-10-30 09:59:11	2023-10-30 09:59:11
a25f91b6133cb4eb3ecb3e0598bbab16b80baa40059e623e387a6b1082d6f5… modbus.exe peexe idle 64bits	28 / 72	2.40 MB	2023-10-30 09:33:50	2023-10-30 09:33:50
06919e6651820eb7f783cea8f5bc78184f3d437bc9c6cde9bfbe1e38e5c731… task_test.json json	0 / 61	379 B	2023-10-30 09:31:22	2023-10-30 09:31:22
a63ba88ad869085f1625729708ba65e87f5b37d7be9153b3db1a1b0e3fed30… c:\628700703.exe peexe 64bits idle	46 / 75	2.33 MB	2023-10-30 09:27:04	2024-09-05 13:32:44
91062ed8cc5d92a3235936fb93c1e9181b901ce6fb9d4100cc01167cdc0874… modbus.exe peexe 64bits idle	32 / 75	2.40 MB	2023-10-30 09:00:06	2023-10-30 09:45:52

Figure 14. Malware samples and task_test.json detection timestamps in VirusTotal.

The data structure of task_test.json and its key/values are the format we would expect to be used as a configuration file by a FrostyGoop executable file. Our analysis of FrostyGoop samples indicates the malware performs read, write and write-multiple Modbus operations. The content of the task_test.json sample depicted in Figure 15 only shows read operations (Code 3).

Unset
```
{
    "Iplist": ["86.[illegible]"
    ],
    "Tasks": [

        {
            "Code": 3,
            "Address": 53370,
            "Count": 1,
            "Value": 1
        },
        {
            "Code": 3,
            "Address": 53760,
            "Count": 123,
            "Value": 0
        },
        {
            "Code": 3,
            "Address": 53882,
            "Count": 1,
            "Value": 1
        },
        {
            "Code": 3,
            "Address": 54272,
            "Count": 123,
            "Value": 0
        }
    ]
}
```
Figure 15. The content of task_test.json used by a FrostyGoop malware sample.

The IP address contained in this JSON file corresponds to an ENCO control device located in Romania as noted.

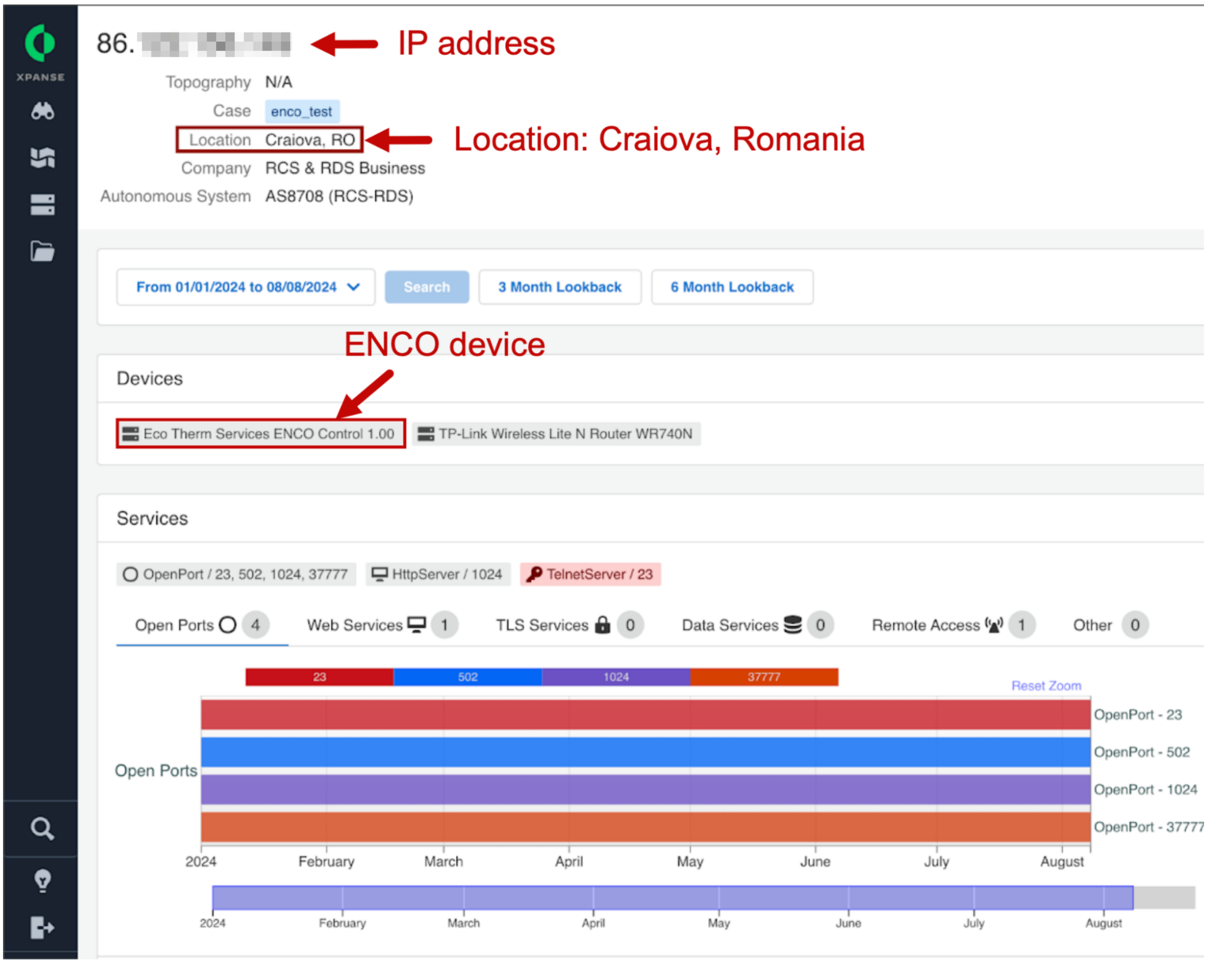


Figure 16. Xpanse query indicating it is an ENCO device in Romania.

Widening our search for exposed ENCO devices, our telemetry revealed 32 IP addresses, all located in either Romania or Ukraine as noted in Figure 17.

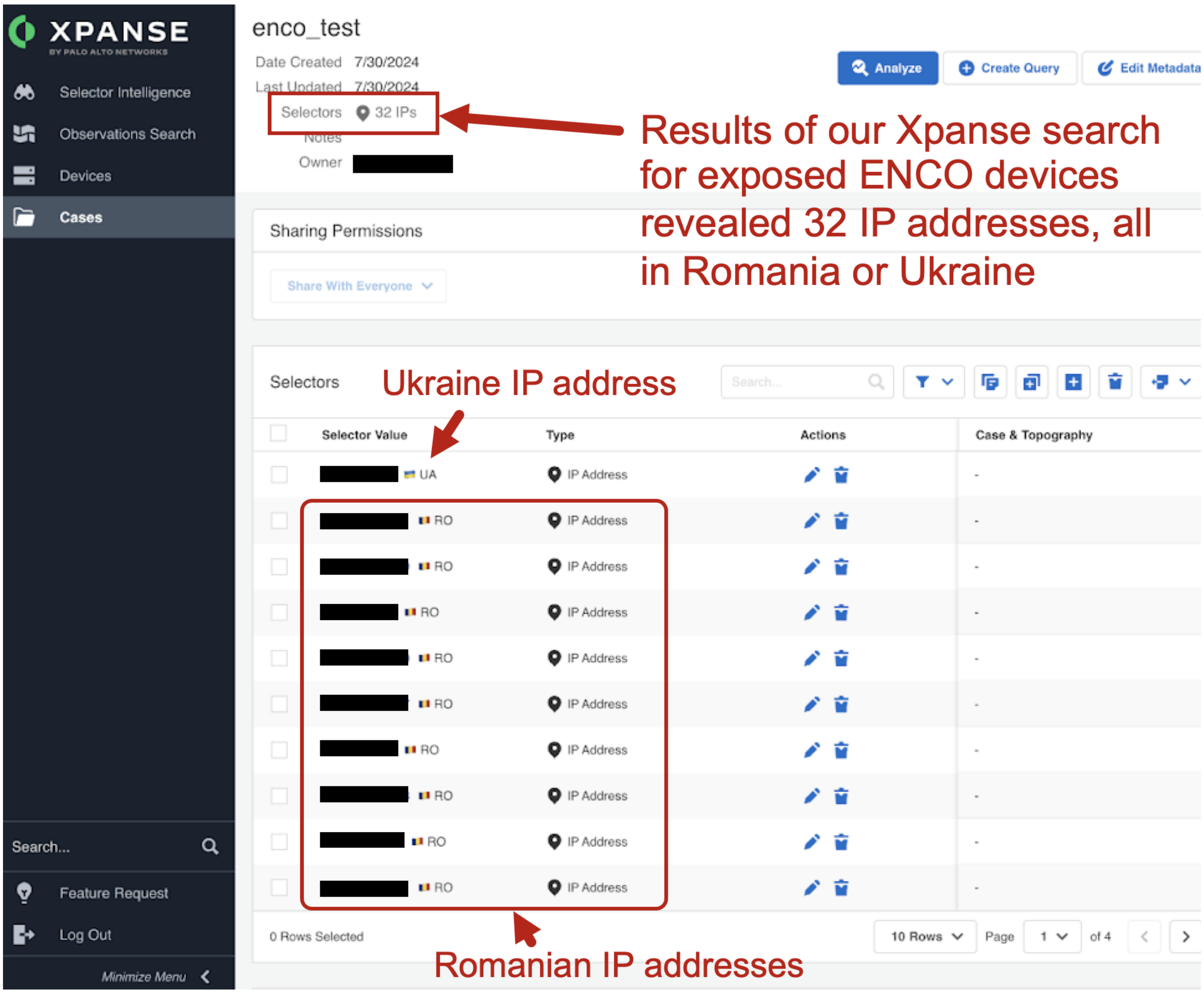
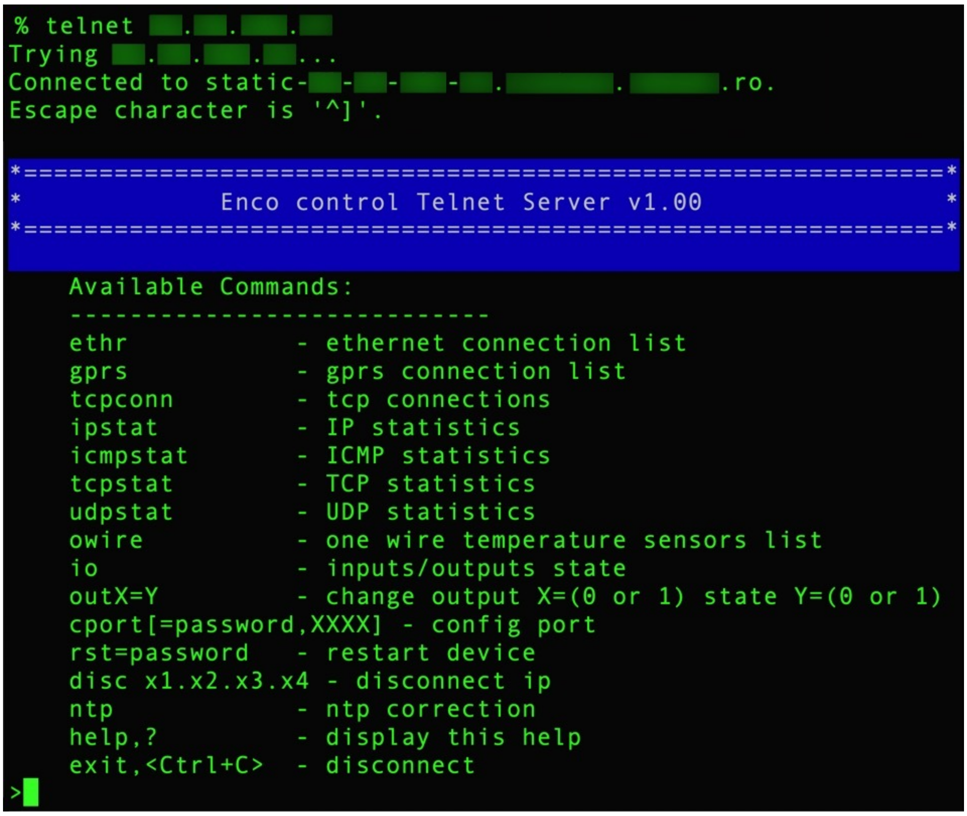


Figure 17. Xpanse query of IP addresses with exposed ENCO devices.

The ENCO devices we discovered all have TCP port 23 exposed for Telnet. Telnet provides a communications and management interface that is considered obsolete because it has no built-in encryption.

Simply connecting to an exposed ENCO device over Telnet reveals an ENCO banner with a list of available commands as shown below in Figure 18. This provides a reportedly easy method to probe for and identify ENCO programmable logic controller (PLC) devices on the internet.

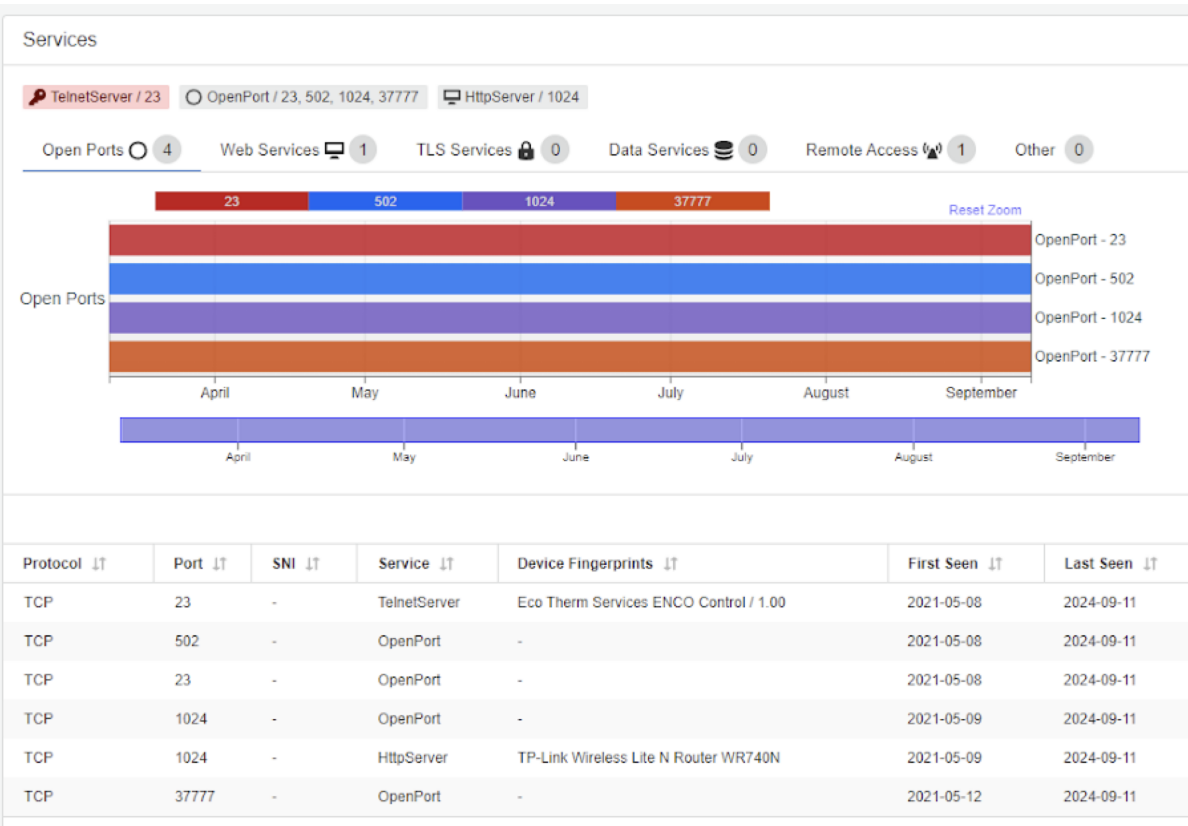


```
% telnet [redacted]
Trying [redacted]...
Connected to static-[redacted].ro.
Escape character is '^]'.

*==============================================================*
*                Enco control Telnet Server v1.00              *
*==============================================================*

    Available Commands:
    ----------------------------
    ethr              - ethernet connection list
    gprs              - gprs connection list
    tcpconn           - tcp connections
    ipstat            - IP statistics
    icmpstat          - ICMP statistics
    tcpstat           - TCP statistics
    udpstat           - UDP statistics
    owire             - one wire temperature sensors list
    io                - inputs/outputs state
    outX=Y            - change output X=(0 or 1) state Y=(0 or 1)
    cport[=password,XXXX] - config port
    rst=password      - restart device
    disc x1.x2.x3.x4 - disconnect ip
    ntp               - ntp correction
    help,?            - display this help
    exit,<Ctrl+C>     - disconnect
>
```

Figure 18. Telnet banner from an exposed ENCO device.

Figure 19 shows a portion of our Xpanse report covering details of the network services running on the server listed in task_test.json. This matches the exposed ports among the other ENCO exposed devices we discovered:

- TCP ports 23 (Telnet)
- 502 (Modbus)
- 1024 (Router WebUI)
- 37777 (ENCO connect port)



Protocol	Port	SNI	Service	Device Fingerprints	First Seen	Last Seen
TCP	23	-	TelnetServer	Eco Therm Services ENCO Control / 1.00	2021-05-08	2024-09-11
TCP	502	-	OpenPort	-	2021-05-08	2024-09-11
TCP	23	-	OpenPort	-	2021-05-08	2024-09-11
TCP	1024	-	OpenPort	-	2021-05-09	2024-09-11
TCP	1024	-	HttpServer	TP-Link Wireless Lite N Router WR740N	2021-05-09	2024-09-11
TCP	37777	-	OpenPort	-	2021-05-12	2024-09-11

Figure 19. Xpanse report for open ports on.

We can glean further information on the ENCO device by accessing it using a web browser and recording the traffic. Figure 20 shows a login screen shown when accessing the ENCO device from a web browser. By viewing the web traffic in Wireshark and examining the HTTP response headers, we find the router is being used as a web server and the name of the router is TP-LINK Wireless Lite N Router WR740N.

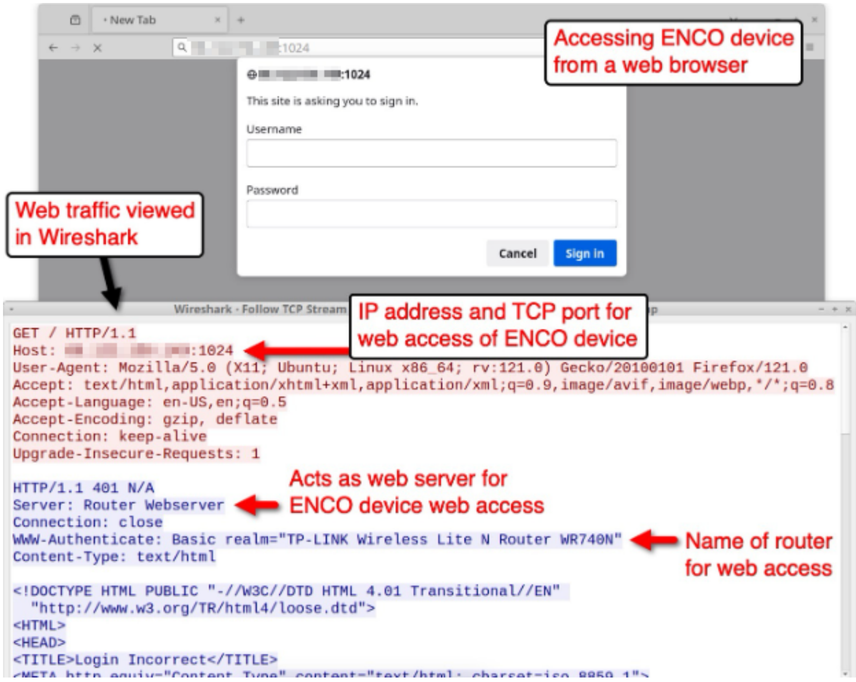


Figure 20. Information gleaned from accessing an ENCO device over a web browser.

According to the NIST website, versions 1 and 2 of the WR740N router's firmware are susceptible to a command injection vulnerability. However, there is no hard evidence to indicate that the attackers exploited this vulnerability in the July 2024 FrostyGoop attack.

Network Traffic Analysis

To analyze FrostyGoop traffic, we tested two samples using task_test.json as the configuration file. The two FrostyGoop samples have the following SHA256 hashes:

- 5d2e4fd08f81e3b2eb2f3eaae16eb32ae02e760afc36fa17f4649322f6da53fb
- a63ba88ad869085f1625729708ba65e87f5b37d7be9153b3db1a1b0e3fed309c

The task_test.json configuration file only has a function code value of 3, which represents a Modbus command to read the holding registers. Accordingly, the FrostyGoop samples only generated commands to read the holding registers of the targeted device at over TCP port 502.

Figure 21 shows an example of the Modbus traffic generated during our test of the FrostyGoop samples, filtered in Wireshark with a customized column display. It reveals Modbus traffic to over TCP port 502, as well as the four register values specified in the task_test.json configuration file:

- 53370
- 53882
- 53760
- 54272

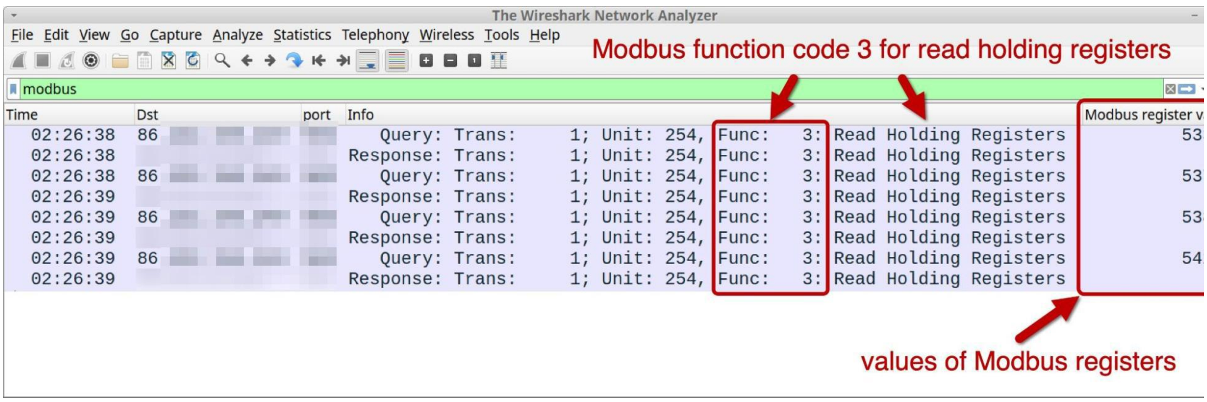


Figure 21. Example of Modbus traffic from our FrostyGoop sample test filtered in Wireshark.

Figure 22 shows an example of a Modbus function code 3 request to read values from the holding registers of the ENCO device, starting with the register number 53760 for the next 123 registries. The device responded with values from registry 53760-53882. These registry entries hold UINT16 values for unsigned integers that can range from 0-65535.

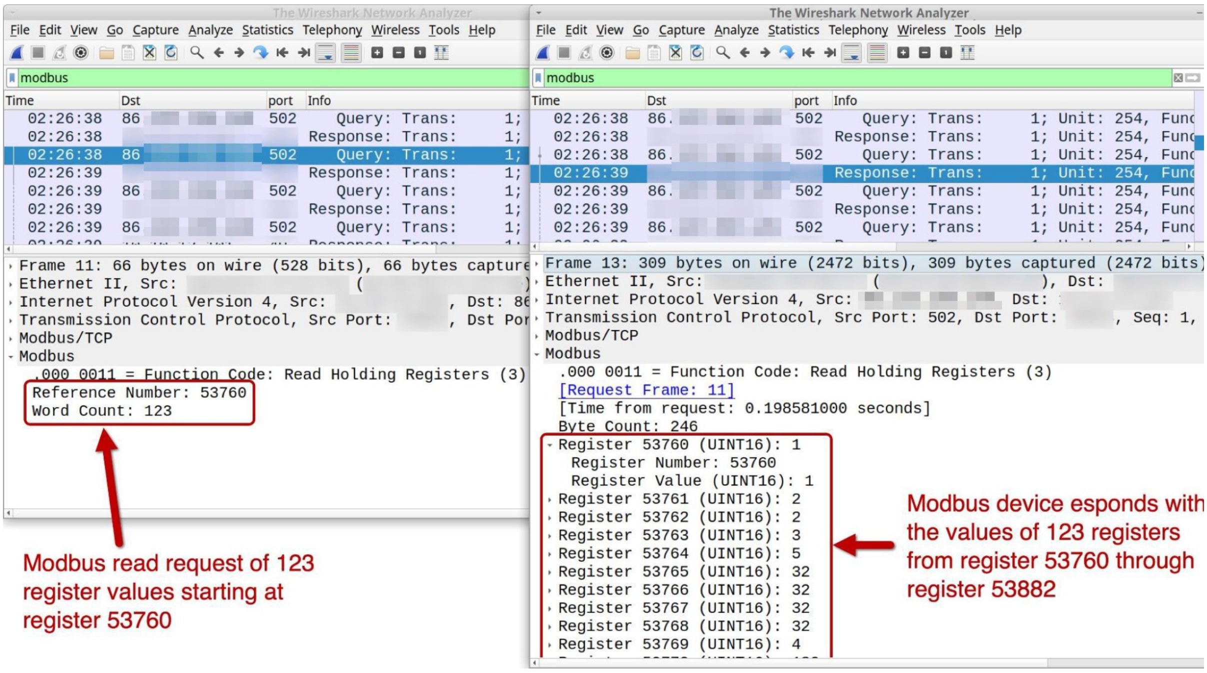


Figure 22. Modbus interaction with the hardware device.

We reverse engineered the samples to track down their functions. Our analysis revealed that the taskWorker function selects actions performed through the following function parameters:

- read holding registers (3)
- write (6)
- writeMultiple (6)

If the JSON configuration file contains the number 1 as a word count value, only one register is returned. If it does not contain the number 1 as a word count value, more than one register is returned.

Figure 23 shows a code snippet from a FrostyGoop sample with the logic to select Modbus operations depending on the value provided:

- 3 for read holding registers
- 6 for write single holding register
- 6 for write multiple holding registers operation

```
int64_t main.Task.taskWorker(int64_t arg1, int64_t arg2, int64_t arg3, int64_t arg4, int64_t arg5 @ ra
0051f806    if (arg5 == 3)
0051f806    {
0051f8ca        rdi = arg1;
0051f8e7        uint64_t rax_8;
0051f8e7        uint64_t rcx_6;
0051f8e7        rax_8 = main.MbConfig.read(arg_30, arg3, arg4, arg8, rdi, arg9);
0051f8ec        rdx_1 = rcx_6;
0051f8ef        rsi_1 = rax_8;
0051f8f2        r8 = 0x1a;
0051f8fa        rcx = var_e0;
0051f902        arg6 = var_140;
0051f806    }
0051f806    else if (arg5 == 6)
0051f810    {
0051f880        rdi = arg7;
0051f8a0        uint64_t rax_5;
0051f8a0        uint64_t rcx_4;
0051f8a0        rax_5 = main.MbConfig.write(arg_30, arg3, arg4, arg8, rdi, arg9);
0051f8a5        rdx_1 = rcx_4;
0051f8a8        rsi_1 = rax_5;
0051f8ab        r8 = 0x16;
0051f8b3        rcx = var_e0;
0051f8bb        arg6 = var_140;
0051f810    }
0051f810    else if (arg5 == 0x10)
0051f816    {
0051f82e        rdi = arg1;
0051f853        uint64_t rax_2;
0051f853        uint64_t rcx_2;
0051f853        rax_2 = main.MbConfig.writeMultiple(arg_30, arg5, arg8, arg3, arg4, rdi, arg9, zmm1
0051f858        rdx_1 = rcx_2;
0051f85b        rsi_1 = rax_2;
0051f85e        r8 = 0x21;
0051f866        rcx = var_e0;
0051f86e        arg6 = var_140;
0051f816    }
0051f816    else
0051f816    {
0051f818        rdx_1 = 0;
0051f81a        rdi = 0;
0051f81c        rsi_1 = 0;
0051f81e        r8 = 0;
0051f816    }
0051f816
```

Figure 23. Code snippet from a FrostyGoop sample showing how it implements Modbus operations.

Conclusion

With cyberattacks against ICS/OT devices and critical infrastructure increasing in recent years, the cybersecurity landscape in these types of environments has become increasingly dangerous. Countries like Ukraine, Romania, Israel, China, Russia and the United States have all been affected by attacks targeting their critical infrastructure. Prior to these incidents, cybersecurity in OT was not considered an essential part of their defensive operations.

The past decade has seen an increase in CS-centric malware, with FrostyGoop being the most recent prominent example. During this time frame, the number of OT and internet of things (IoT) devices exposed to the internet has drastically increased.

An increasing number of OT networks have been connected with IT networks to facilitate facilities management. This has unleashed new ways to perform cyberattacks that can not only damage the cyberspace realm, but also the physical world. Malicious actors can send control commands to field devices easily disguised as regular operations within network traffic, making the activity more difficult to detect and prevent.

For these reasons, we must implement security measures to prevent and mitigate these attacks. Palo Alto Networks customers are better protected from the threats discussed in this blog through the following products:

- Industrial OT Security is designed to:
  - Use machine learning techniques to detect abnormal network traffic and abnormal behavior in engineering workstations and field devices
  - Raise alerts in the event of a compromised environment, based on anomalous command access
  - Generate alerts based on Modbus operations
  - Implement analytics rules for detection of suspicious traffic including anonymous telnet login, brute-force login attempts, default credentials usage

- Cover and identify Common Vulnerabilities and Exposures (CVEs) in MikroTik and other common routers
    - Leverage upstream Advanced WildFire and Advanced Threat Prevention detections, along with IoT device detection capabilities, to detect malware command and control communication
    - Detect devices running vulnerable versions of firmware
- Next-Generation Firewall (NGFW) and Advanced Threat Prevention are designed to:
    - Provide complete visibility and control of the applications in use across all users and devices in all locations all the time
    - Automatically reprogram your firewall with the latest intelligence using inline machine learning as well as the application and threat signatures
    - Implement rules TID 31667 (Modbus read coils) and TID 31668 (Modbus write coils), which allows administrators to identify abnormal devices performing Modbus operations
    - Implement MikroTik CVEs related to prevent remote code execution and command injection vulnerabilities from being exploited within the network
- Advanced WildFire is designed to:
    - Identify malicious binaries and make verdict determinations when analyzing executing processes
    - Implement detection rules to identify, block and prevent deployment of FrostyGoop/BUSTLEBERM and its variants, as well as other ICS-centric ransomware and malware
- Cortex Xpanse is designed to:
    - Provide a complete, accurate and continuously updated inventory of all global internet-facing assets, including exposed OT services and devices
    - Enable discovery, evaluation and mitigation of cyberattack surface risks
    - Facilitate evaluation of supplier risk
- Cortex XDR and XSIAM are designed to:
    - Accurately detect threats with behavioral analytics and reveal the root cause to speed up investigations
    - Better protect against malware discussed in this article through Cortex XDR, including WildFire, Behavioral Threat Protection and the Local Analysis module
- Unit 42 researchers at Palo Alto Networks are committed to discovering new malware and threats. We share our findings and feed the results back into our products and services, so our customers are better protected.

If you think you may have been compromised or have an urgent matter, get in touch with the Unit 42 Incident Response team or call:

- North America Toll-Free: 866.486.4842 (866.4.UNIT42)
- EMEA: +31.20.299.3130
- APAC: +65.6983.8730
- Japan: +81.50.1790.0200

Palo Alto Networks has shared these findings with our fellow Cyber Threat Alliance (CTA) members. CTA members use this intelligence to rapidly deploy protections to their customers and to systematically disrupt malicious cyber actors. Learn more about the Cyber Threat Alliance.

Indicators of Compromise

SHA256 hash:

- 5d2e4fd08f81e3b2eb2f3eaae16eb32ae02e760afc36fa17f4649322f6da53fb
- File size: 3.7 MB (3,699,200 bytes)
- File type: PE32+ executable (console) x86-64 (stripped to external PDB), for MS Windows
- File description: Windows executable file for FrostyGoop malware

SHA256 hash:

- a63ba88ad869085f1625729708ba65e87f5b37d7be9153b3db1a1b0e3fed309c
- File size: 2.4 MB (2,439,680 bytes)
- File type: PE32+ executable (console) x86-64 (stripped to external PDB), for MS Windows
- File description: Windows executable file for FrostyGoop malware

SHA256 hash:

- 2fd9cb69ef30c0d00a61851b2d96350a9be68c7f1f25a31f896082cfbf39559a
- File size: 3.4 MB (3,359,232 bytes)
- File type: PE32+ executable (console) x86-64 (stripped to external PDB), for MS Windows
- File description: Windows executable file for FrostyGoop malware

SHA256 hash:

- c64b67c116044708e282d0d1a8caea2360270a7fc679befa5e28d1ca15f6714c
- File size: 2.0 MB (1,951,232 bytes)

- File type: PE32+ executable (console) x86-64 (stripped to external PDB), for MS Windows
- File description: Windows executable file for FrostyGoop malware

SHA256 hash:

- 91062ed8cc5d92a3235936fb93c1e9181b901ce6fb9d4100cc01167cdc08745f
- File size: 2.5 MB (2,516,480 bytes)
- File type: PE32+ executable (console) x86-64 (stripped to external PDB), for MS Windows
- File description: Windows executable file for FrostyGoop malware

SHA256 hash:

- a25f91b6133cb4eb3ecb3e0598bbab16b80baa40059e623e387a6b1082d6f575
- File size: 2.5 MB (2,515,968 bytes)
- File type: PE32+ executable (console) x86-64 (stripped to external PDB), for MS Windows
- File description: Windows executable file for FrostyGoop malware

SHA256 hash:

- 9cf30d82a86a9485f7bbd0786a5de207cf4902691a3efcfc966248cb1e87d5b7
- File size: 1.8 MB (1,773,568 bytes)
- File type: PE32+ executable (console) x86-64 (stripped to external PDB), for MS Windows
- File description: Windows executable file for go-encrypt.exe, likely used during previous FrostyGoop activity

SHA256 hash:

- 06919e6651820eb7f783cea8f5bc78184f3d437bc9c6cde9bfbe1e38e5c73160
- File size: 0.4 KB (379 bytes)
- File type: JSON text data
- File description: JSON file named task-test.json likely used to test go-encrypt.exe in July 2024 FrostyGoop attack

Additional Resources

- Impact of FrostyGoop ICS Malware on Connected OT Systems – Dragos Intelligence Brief
- Simple 'FrostyGoop' malware responsible for turning off Ukrainians' heat in January attack – SANS Blog
- go-json: Fast JSON encoder/decoder compatible with encoding/json for Go – goccy (Masaaki Goshima) on GitHub
- Queues: A simple sync.WaitGroup like queue and goroutine execution controller – hsblhsn (Hasibul Hasan) on GitHub
- How to Find and Probe ENCO PLCs on the Internet Just Like FrostyGoop malware – ZeroNtek
- CVE-2023-33538 Detail – NIST NVD

Updated Nov. 19, 2024, at 9:00 a.m. PT to align statements with Dragos report.

Updated Nov. 20, 2024, at 10:00 a.m. PT to correct PANW product names.

Updated Nov. 20, 2024, at 10:54 a.m. PT to correct Modbus code.

Updated Dec. 3, 2024, at 10:23 a.m. PT to correct typo.