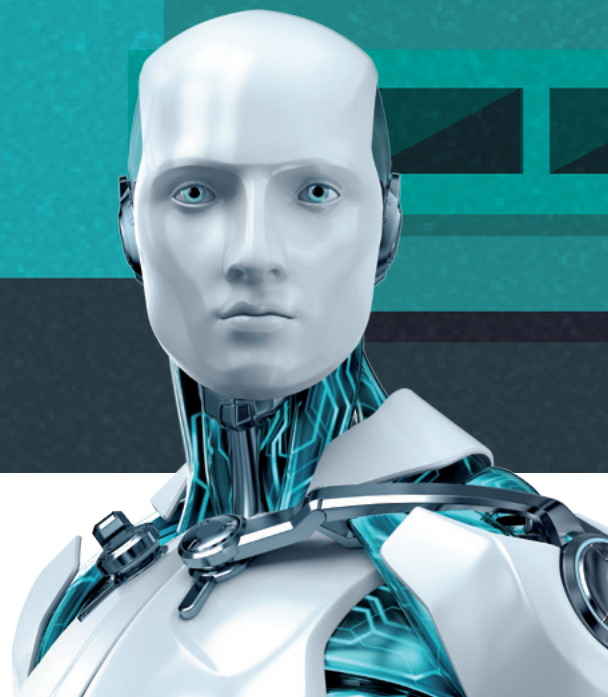# WIN32/INDUSTROYER
## A new threat for industrial control systems

**Anton Cherepanov, ESET**
Version 2017-06-12

# CONTENTS

# WIN32/INDUSTROYER:
# A NEW THREAT FOR INDUSTRIAL CONTROL SYSTEMS

Win32/Industroyer is a sophisticated piece of malware designed to disrupt the working processes of industrial control systems (ICS), specifically industrial control systems used in electrical substations.

Those behind the Win32/Industroyer malware have a deep knowledge and understanding of industrial control systems and, specifically, the industrial protocols used in electric power systems. Moreover, it seems very unlikely anyone could write and test such malware without access to the specialized equipment used in the specific, targeted industrial environment.

Support for four different industrial control protocols, specified in the standards listed below, has been implemented by the malware authors:

- IEC 60870-5-101 (aka IEC 101)
- IEC 60870-5-104 (aka IEC 104)
- IEC 61850
- OLE for Process Control Data Access (OPC DA)

In addition to all that, the malware authors also wrote a tool that implements a denial-of-service (DoS) attack against a particular family of protection relays, specifically the _Siemens SIPROTEC_ range.

All this considered, the Win32/Industroyer malware authors show an intensive focus that suggests they are highly specialized in industrial control systems.

The capabilities of this malware are significant. When compared to the toolset used by threat actors in the 2015 attacks against the Ukrainian power grid which culminated in a black out on December 23, 2015 (BlackEnergy, KillDisk, and other components, including legitimate remote access software) the gang behind Industroyer are more advanced, since they went to great lengths to create malware capable of directly controlling switches and circuit breakers. We have seen indications that this malware could have been the tool used by attackers to cause the power outage in Ukraine in December 2016, although at the time of writing, it is not confirmed, and the investigation is still ongoing. The infection vector remains unknown.

The malware contains multiple modules, as analyzed and described in the next sections of this whitepaper. However, before diving into those details, the following simplified schematic shows the connections between the components of the malware.
See Fig. 1 below.

While some components (e.g. Data wiper) are similar in concept to the _2015 BlackEnergy attacks_ against power grid companies in Ukraine, we don't see any link between those attacks and the code in this malware.
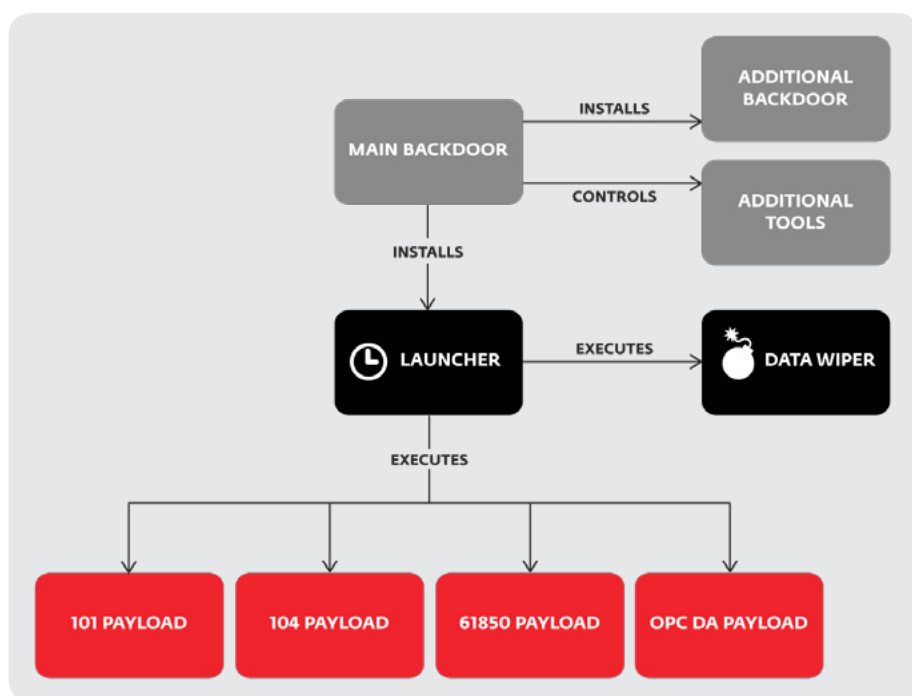


**Figure 1.**
Simplified schematic of Win32/
Industroyer components.

# MAIN BACKDOOR

We refer to the core component of Industroyer as the main backdoor. The main backdoor is used by the attackers behind Industroyer to control all other components of the malware.

As backdoors go, this component is pretty straightforward, connecting to its remote C&C server using HTTPS and receiving commands from the attackers. All analyzed samples are hardcoded to use the same proxy address, located in the local network. Thus, the backdoor is clearly designed to work only in one specific organization. It is also worth mentioning that most of the C&C servers used by this backdoor are running Tor software.

Perhaps the most interesting feature of this backdoor is that attackers can define a specific hour of the day when the backdoor will be active. For example, the attackers can modify the backdoor in this way so it will communicate with its C&C server only outside working hours. This can make detection based only on network traffic examination harder. However, all the samples analyzed so far are set to work 24 hours round the clock.

```
1 int main_loop()
2 {
3   struct _SYSTEMTIME SystemTime; // [esp+0h] [ebp-14h]@4
4   DWORD dwMilliseconds; // [esp+10h] [ebp-4h]@2
5
6   SetLastError(0);
7   if ( GetLastError() != ERROR_ALREADY_EXISTS )
8   {
9     dwMilliseconds = 5000;
10    SetUnhandledExceptionFilter(TopLevelExceptionFilter);
11    if ( !GetSystemMetrics(SM_CLEANBOOT) )
12    {
13      if ( create_1map1_handle() )
14      {
15        while ( 1 )
16        {
17          do
18          {
19            Sleep(dwMilliseconds);
20            GetLocalTime(&SystemTime);
21          }
22          while ( SystemTime.wHour >= 24u );
23          c2_connect_and_execute_cmd(&dwMilliseconds);
24        }
25      }
26    }
27  }
28  return 0;
29 }
```

**Figure 2.**
The decompiled main backdoor code has a check for time-of-day.

Once connected to its remote C&C server, the main backdoor component sends the following data in a POST-request:

- the globally unique identifier (GUID) string for the current hardware profile retrieved via `GetCurrentHwProfile`
- the version of the malware: 1.1e
- the hardcoded ID of the sample
- the result of any previously-received command

The hardcoded ID is used by the attacker as an identifier for the infected machine. Across all analyzed samples we found the following hardcoded ID values:

- DEF
- DEF-C
- DEF-WS
- DEF-EP
- DC-2-TEMP
- DC-2
- CES-McA-TEMP
- CES
- SRV_WSUS
- SRV_DC-2
- SCE-WSUS01

The main backdoor component supports the following commands:

| Command ID | Purpose |
|---|---|
| 0 | Execute a process |
| 1 | Execute a process under a specific user account. Credentials for the account are supplied by the attacker |
| 2 | Download a file from C&C server |
| 3 | Copy a file |
| 4 | Execute a shell command |
| 5 | Execute a shell command under a specific user account. Credentials for the account are supplied by the attacker |
| 6 | Quit |
| 7 | Stop a service |
| 8 | Stop a service under a specific user account. Credentials for the account are supplied by the attacker |
| 9 | Start a service under a specific user account. Credentials for the account are supplied by the attacker |
| 10 | Replace "Image path" registry value for a service |

Once the attackers obtain administrator privileges, they can upgrade the installed backdoor to a more privileged version that is executed as a *Windows service program*. To do this they pick an existing, non-critical Windows service and replace its `ImagePath` registry value with the path of the new backdoor's binary.

The functionality of the main backdoor that works as a Windows service is the same as just described. However, there are two small differences: first the backdoor's version is 1.1s, instead of 1.1e, and second, there is code obfuscation. The code of this version of the backdoor is mixed with junk assembly instructions.

```
.text:00403FD2 main_func proc near          ; CODE XREF: WinMain(x,x,x,x)+14↑p
.text:00403FD2                              ; .text:004038C4↑p
.text:00403FD2         call    $+5
.text:00403FD7
.text:00403FD7 loc_403FD7:                   ; CODE XREF: main_func+57↓j
.text:00403FD7                              ; main_func+5F↓j
.text:00403FD7         add     esp, 4
.text:00403FDA         push    ebp          ; lpOverlapped
.text:00403FDB         mov     ebp, esp
.text:00403FDD         cmp     edx, 142F9F9Ah
.text:00403FE3         jz      short loc_404023
.text:00403FE5         push    ecx
.text:00403FE6         push    ecx
.text:00403FE7         mov     eax, [ebp+10h]
.text:00403FEA         mov     dword_416190, eax
.text:00403FEF         mov     eax, [ebp+8]
.text:00403FF2         mov     dword_416194, eax
.text:00403FF7         mov     eax, [ebp+0Ch]
.text:00403FFA         cmp     edx, 0B5B93EF3h
.text:00404000         jz      short loc_404023
.text:00404002         mov     lpOverlapped, eax
.text:00404007         mov     [ebp-8], eax
.text:0040400A         lea     eax, [ebp-8]
.text:0040400D         push    eax                      ; lpServiceStartTable
.text:0040400E         mov     dword ptr [ebp-4], offset ServiceMain
.text:00404015         call    ds:StartServiceCtrlDispatcherW
.text:0040401B         xor     al, al
.text:0040401D         mov     esp, ebp
.text:0040401F         pop     ebp
.text:00404020         retn
```

**Figure 3.**
The obfuscated assembly code of the main backdoor that works as a Windows service.

```
.text:01004AD5   lea    eax, [ebp+var_50]             .text:01004AD5   lea    eax, [ebp+var_50]
.text:01004AD8   push   eax                           .text:01004AD8   push   eax
.text:01004AD9   lea    eax, [ebp+h]                  .text:01004AD9   lea    eax, [ebp+h]
.text:01004ADC   push   eax                           .text:01004ADC   push   eax
.text:01004ADD   push   0B0h                          .text:01004ADD   push   0B0h
.text:01004AE2   push   hWnd                          .text:01004AE2   push   hWnd
.text:01004AE8   mov    stru_100A680.lStructSize, 58h .text:01004AE8   mov    stru_100A680.lStructSize, 58h
.text:01004AF2   mov    stru_100A680.hwndOwner, edx   .text:01004AF2   mov    stru_100A680.hwndOwner, edx
.text:01004AF8   mov    stru_100A680.nMaxFile, 104h   .text:01004AF8   mov    stru_100A680.nMaxFile, 104h
.text:01004B02   mov    stru_100A500.lStructSize, 28h .text:01004B02   mov    stru_100A500.lStructSize, 28h
.text:01004B0C   mov    stru_100A500.hwndOwner, edx   .text:01004B0C   mov    stru_100A500.hwndOwner, edx
.text:01004B12   call   esi ; SendMessageW            .text:01004B12   call   esi ; SendMessageW
.text:01004B14   push   [ebp+var_50]                  .text:01004B14   pusha
.text:01004B17   push   [ebp+h]                       .text:01004B15   pushf
.text:01004B1A   push   0B1h                          .text:01004B16   neg    ebx
.text:01004B1F   push   hWnd                          .text:01004B18   shr    eax, 1
.text:01004B25   call   esi ; SendMessageW            .text:01004B1B   dec    ebx
.text:01004B27   push   ebx                           .text:01004B1C   mov    eax, 17B200AFh
.text:01004B28   push   ebx                           .text:01004B21   mov    edi, 71CFC28h
.text:01004B29   push   0B7h                          .text:01004B26   or     edi, dword_10095C7
.text:01004B2E   push   hWnd                          .text:01004B2C   xor    esi, 1C779E91h
.text:01004B34   call   esi ; SendMessageW            .text:01004B32   xor    eax, eax
.text:01004B36   push   ebx                           .text:01004B34   dec    edi
.text:01004B37   call   ds:GetKeyboardLayout          .text:01004B35   rol    esi, 5
.text:01004B3D   and    ax, 3FFh                      .text:01004B38   and    esi, edi
.text:01004B41   cmp    ax, 11h                       .text:01004B3A   and    esi, edi
.text:01004B45   jnz    short loc_1004B58             .text:01004B3C   rol    edx, 6
.text:01004B47   push   1                             .text:01004B3F   neg    eax
.text:01004B49   push   1                             .text:01004B41   xor    esi, eax
.text:01004B4B   push   0D8h                          .text:01004B43   neg    ebx
.text:01004B50   push   hWnd                          .text:01004B45   shr    ebx, 5
.text:01004B56   call   esi ; SendMessageW            .text:01004B48   mov    ecx, 5E95422h
```

**Figure 4.**
Comparison between original Notepad binary code (at the left) and backdoored binary code.

# ADDITIONAL BACKDOOR

The additional backdoor provides an alternative persistence mechanism that allows the attackers to regain access to a targeted network in case the main backdoor is detected and/or disabled.

This backdoor is a trojanized version of the Windows Notepad application. This is a fully functional version of the application, but the malware authors have inserted malicious code that is executed each time the application is launched. Once the attackers gain administrator privileges, they are able to replace the legitimate Notepad manually.

The inserted malicious code is heavily obfuscated, but once the code is decrypted it connects to a remote C&C server, which is different to the one linked in the main backdoor, and downloads a payload. This is in the form of shellcode that is loaded directly into memory and executed. In addition, the inserted code decrypts the original Windows Notepad code, which is stored at the end of the file, and then passes execution to it. Thus, the Notepad application works as expected. See Fig. 4 on previous page.

# LAUNCHER COMPONENT

This component is a separate executable responsible for launching the payloads and the Data wiper component.

The Launcher component contains a specific time and date. Analyzed samples contained two dates, 17th December 2016 and 20th December 2016. Once one of these dates is reached the component creates two threads. The first thread makes attempts to load a payload DLL, while the second thread waits one or two hours (it depends on the Launcher component version) and then attempts to load the Data wiper component. The priority for both threads is set to THREAD_PRIORITY_HIGHEST, which means that these two threads receive a higher than normal share of CPU resources from the operating system.

The name of the payload DLL is supplied by the attackers via a command line parameter supplied in one of the main backdoor's "execute a shell command" commands. The Data wiper component is always named haslo.dat. The expected command lines are of the form:

```
%LAUNCHER%.exe %WORKING_
DIRECTORY% %PAYLOAD%.dll
%CONFIGURATION%.ini
```

Each argument on the command line represents the following:

- %LAUNCHER%.exe is the filename of the Launcher component
- %WORKING_DIRECTORY% is the directory where the payload DLL and configuration is stored
- %PAYLOAD%.dll is the filename of the payload DLL
- %CONFIGURATION%.ini is the file that stores configuration data for the specified payload. The path to this file is supplied to the payload DLL by the Launcher component

The payload and Data wiper components are standard Windows DLL files. In order to be loaded by the Launcher component they must export a function named Crash as seen in Figure 5.



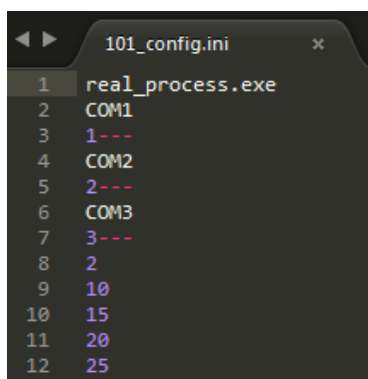**Figure 5.**
Example payload DLL that has internal name Crash101.dll and Crash export function.

# 101 PAYLOAD COMPONENT

This payload DLL has the filename `101.dll` and is named after IEC 101 (aka *IEC 60870-5-101*), an international standard that describes a protocol for monitoring and controlling electric power systems. The protocol is used for communication between industrial control systems and Remote Terminal Units (RTUs). The actual communication is transmitted through a serial connection.

The 101 payload component partly implements the protocol described in the IEC 101 standard and is able to communicate with an RTU or any other device with support for that protocol.

Once executed, the 101 payload component parses the configuration stored in its INI file. The configuration may contain several entries: process name, Windows device names (usually COM ports), the number of Information Object Address (IOA) ranges, and the beginning and ending IOA values for the specified number of IOA ranges. IOA is a number that identifies a particular data element in the device. Figure 6 illustrates a 101 payload configuration file with two defined IOA ranges, 10-15 and 20-25.
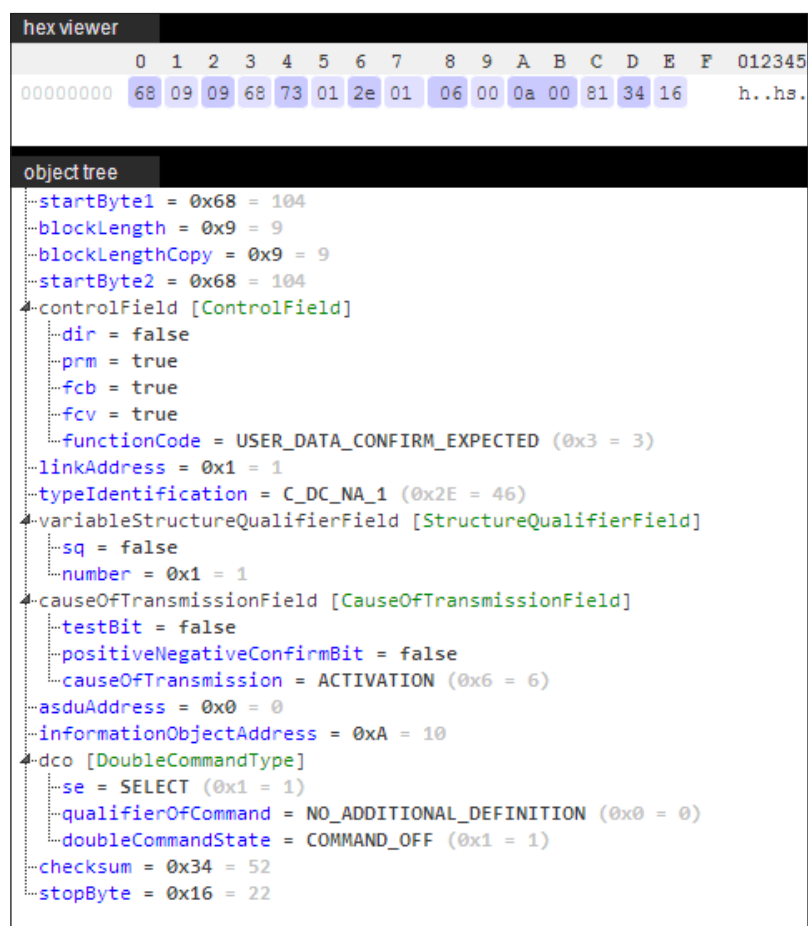
**Figure 6.**
An example of a 101 payload DLL configuration.

The name of the process specified in the configuration belongs to an application the attackers suspect is running on the victim machine. It should be the application the victim machine uses to communicate through serial connection with the RTU. The 101 payload attempts to terminate the specified process and starts to communicate with the specified device, using the `CreateFile`, `WriteFile` and `ReadFile` Windows API functions. The first COM port from the configuration file is used for the actual communication and the two other COM ports are just opened to prevent other processes accessing them. Thus, the 101 payload component is able to take over and maintain control of the RTU device.

This component iterates through all IOAs in the defined IOA ranges. For each such IOA it constructs two "select and execute" packets, one with a single command (`C_SC_NA_1`) and one with a double command (`C_DC_NA_1`) and sends these to the RTU device. The main goal of the component is to change the On/Off state of single command type IOA and double command type IOA. Specifically, the 101 payload has three stages: in the first stage this component attempts to switch IOAs to their Off state, in the second stage it attempts to invert IOA states to On, and in the final stage the component switches IOA states to Off again.
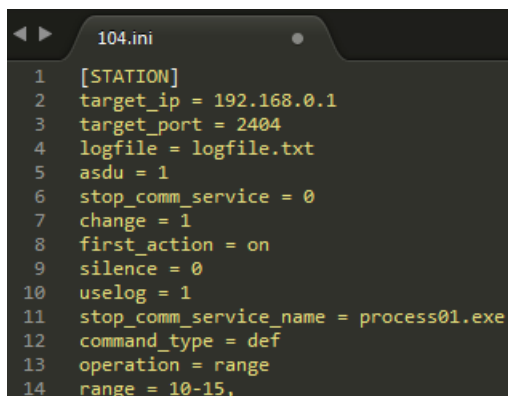
**Figure 7.**
An example of a 101 payload packet, after being dissected in Kaitai Struct WebIDE.

# 104 PAYLOAD COMPONENT

This payload DLL has the filename 104.dll and is named after IEC 104 (aka _IEC 60870-5-104_), an international standard. The IEC 104 protocol extends IEC 101, so the protocol can be transmitted over a TCP/IP network.

Due to its highly configurable nature, this payload can be customized by the attackers for different infrastructures. Figure 8 shows what a configuration file may look like.

Once executed, the 104 payload DLL attempts to read its configuration file. As described above, the path for the configuration file is supplied by the Launcher component.

The configuration contains a STATION section followed by properties that configure how the 104 payload should work. The configuration may contain multiple STATION entries.

```
104.ini
1   [STATION]
2   target_ip = 192.168.0.1
3   target_port = 2404
4   logfile = logfile.txt
5   asdu = 1
6   stop_comm_service = 0
7   change = 1
8   first_action = on
9   silence = 0
10  uselog = 1
11  stop_comm_service_name = process01.exe
12  command_type = def
13  operation = range
14  range = 10-15,
```

**Figure 8.**
An example of 104 payload DLL configuration.

Our analysis of this component reveals the following possible configuration properties:

| Property | Expected value | Purpose |
| --- | --- | --- |
| target_ip | IP address | The IP address that will be used for the communication using IEC 104 protocol standard |
| target_port | Port number | Self-explanatory |
| uselog | 1 or 0 | Enables or disables logging to a file |
| logfile | Filename | Specifies the filename for the log, if enabled |
| stop_comm_service | 1 or 0 | Enables or disables termination of the process |
| stop_comm_service_name | Process name | Specifies the process name that will be terminated |
| timeout | Timeout in milliseconds | Specifies timeout between send and recv calls. Default value: 15000 |
| socket_timeout | Timeout in milliseconds | Specify the receiving timeout. Default value: 15000 |
| silence | 1 or 0 | Enables or disables console output |
| asdu | Integer | Specifies ASDU (Application Service Data Unit) address also known as sector |
| first_action | on or off | Specifies the Switch value in ASDU packet for first iteration |
| change | 1 or 0 | Specifies that the Switch value in ASDU packet should be inverted during iterations |
| command_type | def or short or long or persist | Specifies command pulse duration for qualifier of command (QOC) |
| operation | range or sequence or shift | Specifies iteration type for Information Object Addresses (IOA) |
| range | Specific format of IOAs | Specifies range of Information Object Addresses (IOA) |
| sequence | Specific format of IOAs | Specifies sequence of Information Object Addresses (IOA) |
| shift | Specific format of IOAs | Specifies shift of Information Object Addresses (IOA) |

Once the configuration file is read, the 104 payload creates a thread for each STATION section defined in the configuration file. In each such thread, the 104 payload will attempt to communicate with the specified IP address using the protocol described in the IEC 104 standard. Before the connection is made, the 104 payload attempts to terminate the legitimate process that is normally responsible for IEC 104 communication with the device. It does so only if the `stop_comm_service` property is specified in its configuration. By default, the 104 payload terminates the process named `D2MultiCommService.exe`, or the process name specified in its configuration.

The main idea behind the 104 payload is relatively simple. It connects to the specified IP address and starts to send packets with the ASDU address that was defined in its configuration. The goal of this communication is to interact with an IOA of a single command type.

In the configuration file, the attacker can define the `operation` property to specify exactly how single command type IOAs will be iterated.

The first such `operation` mode is the `range` mode. The attackers use this mode in order to discover possible IOAs in the targeted device. The attackers have to take this approach because the protocol described in the IEC 104 standard does not provide a specific method to obtain such information.

The `range` mode has two stages. During the first stage, once the range of IOAs is obtained from the configuration file, the 104 payload connects to the target IP address and starts to iterate through the specified IOAs. To each such IOA the 104 payload sends "select and execute" packets in order to switch the state and to confirm whether the IOA belongs to the single command type. See Fig. 9

Once all possible IOAs from the specified range are iterated, the 104 payload switches to the second stage of `range` mode. If logging is enabled, the payload writes `Starting only success` to the log. The rest of this second stage is an infinite loop that uses the previously discovered IOAs of single command type. In the loop the payload constantly sends "select and execute" packets. In addition, if the option change is defined, the payload flips the On/Off state between loop steps.

Figure 10 demonstrates the log file that was produced by the 104 payload during our analysis. It shows the payload iterated IOAs from 10 to 15, and once IOAs of the single command type were discovered, the payload started to use them in the loop. The configuration had the `change` option enabled, so between loop iterations the payload flipped the switch value from On to Off and wrote it to the log. See Fig. 10 on the right side.

The second `operation` mode is the `shift` mode. This is very similar to the range mode. The attacker defines, in the configuration file, a range of IOAs and shift values. Once the 104 payload is activated it does everything the same way as in `range` mode; however, once all IOAs in the defined range are iterated, it starts to iterate over the new range. The new range is calculated by adding the shift values to the default range values.

The third `operation` mode is the `sequence` mode. It can be used by attackers once they know the values of all IOAs of the single command type that are supported by the connected device. This payload immediately executes an infinite loop, sending "select and execute" packets to the IOAs defined in the configuration file.

Aside from its logging capability, the 104 payload can output debug information to the console, as seen in Figure 11 on the right side.

```
▷ Internet Protocol Version 4, Src: 192.168.0.1, Dst: 192.168.0.2
▷ Transmission Control Protocol, Src Port: 2404, Dst Port: 49168, Seq: 39, Ack: 45, Len: 16
▷ IEC 60870-5-104-Apci: -> I (2,2)
▲ IEC 60870-5-104-Asdu: ASDU=1 C_SC_NA_1 ActTerm IOA=10 'single command'
    TypeId: C_SC_NA_1 (45)
    0... .... = SQ: False
    .000 0001 = NumIx: 1
    ..00 1010 = CauseTx: ActTerm (10)
    .0.. .... = Negative: False
    0... .... = Test: False
    OA: 0
    Addr: 1
  ▲ IOA: 10
      IOA: 10
    ▲ SCO: 0x01
        .... ...1 = ON/OFF: On
        .000 00.. = QU: No pulse defined (0)
        0... .... = S/E: Execute
```

**Figure 9.**
An example of a 104 payload packet, after being dissected by Wireshark.

**Figure 10.**
Example log file produced by the 104 payload.



**Figure 11.**
The console output of the 104 payload.

# 61850 PAYLOAD COMPONENT

Unlike the 101 and 104 payloads, this payload component exists as a standalone malicious tool comprising an executable named `61850.exe` and the DLL `61850.dll`. It is named after the _IEC 61850_ standard. This standard describes a protocol used for multivendor communication among devices that perform protection, automation, metering, monitoring, and control of electrical substation automation systems. The protocol is very complex and robust, but the 61850 payload uses only a small subset of the protocol to produce its disruptive effect.

Once executed, the 61850 payload DLL attempts to read the configuration file, the path to which is supplied by the Launcher component. The standalone version defaults to reading its configuration from `i.ini`. The configuration file is expected to contain a list of IP addresses of devices capable of communicating via the protocol described in the IEC 61850 standard.

If the configuration file is not present, then this component enumerates all connected network adaptors to determine their TCP/IP subnet masks. The 61850 payload then enumerates all possible IP addresses for each of these subnet masks, and tries to connect to port 102 on each of those addresses. Therefore, this component has the ability to discover relevant devices in the network automatically.

Otherwise, if a configuration file is present and it contains target IP addresses, this component connects to port 102 on those IP addresses and on IP addresses that were discovered automatically.

Once this component connects to a target host, it sends a Connection Request packet using the Connection Oriented Transport Protocol, as seen in Figure 12.



**Figure 12.**
A Connection Request packet, after dissection by Wireshark.

If the target device responds appropriately, the 61850 payload then sends an `InitiateRequest` packet using the *Manufacturing Message Specification* (MMS). If the expected answer is received, it continues, sending an MMS `getNameList` request. Thereby, the component compiles a list of object names in a Virtual Manufacturing Device (VMD).

Next, this component enumerates the objects discovered in the previous step and sends the device domain-specific `getNameList` requests with each object name. This enumerates named variables in a specific domain.

See Fig. 13

Afterwards, the 61850 payload parses data received in response to these requests, searching for variables that contain following combinations of strings:

- **CSW, CF, Pos,** and **Model**
- **CSW, ST, Pos,** and **stVal**
- **CSW, CO, Pos, Oper,** but not **$T**
- **CSW, CO, Pos, SBO,** but not **$T**

The string CSW is a name for logical nodes, which are used to control circuit breakers and switches.

For variables that contain the Model or stVal string the 61850 payload sends an additional MMS `Read` request. For some of the variables this component may also issue an MMS `Write` request that will change its state.

The 61850 payload produces a log file of its operations that contains the IP addresses, MMS domains, named variables and the node states (open or closed) of its targets.
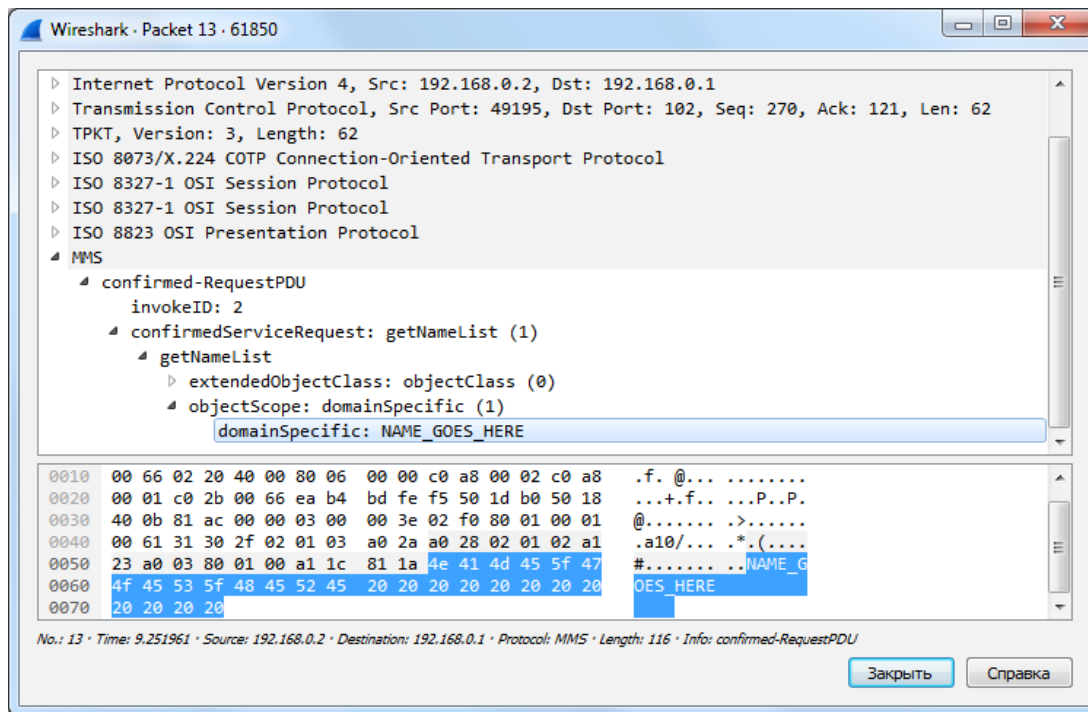


**Figure 13.**
The dissected MMS `getNameList` request in Wireshark.

# OPC DA PAYLOAD COMPONENT

The OPC DA payload component implements a client for the protocol described in the *OPC Data Access* specification. *OPC (OLE for Process Control)* is a software standard and specification that is based on Microsoft technologies such as OLE, COM, and DCOM. The Data Access (DA) part of the OPC specification allows real-time data exchange between distributed components, based on a client–server model.

This component exists as a standalone malicious tool with the filename `OPC.exe` and a DLL, which implement both 61850 and OPC DA payload functionalities. This DLL is named, internally in PE export table, `OPCClientDemo.dll`, suggesting that the code of this component may be based on the open source project *OPC Client*.



**Figure 14.**
The PE export reveals the internal DLL name of the OPC DA payload.

The OPC DA payload does not require any kind of configuration file. Once executed by the attacker, it enumerates all OPC servers using the `ICatInformation::EnumClassesOfCategories` method with `CATID_OPCDAServer20` category identifier and `IOPCServer::GetStatus` to identify the ones running.

Next the component uses the `IOPCBrowseServerAddressSpace` interface to enumerate all OPC items on the server. Specifically, it looks for items that contain the following strings in their name:

- ctlSelOn
- ctlOperOn
- ctlSelOff
- ctlOperOff
- \Pos and stVal

The names of these items may suggest that attackers are interested in OPC items provided by OPC servers that belong to solutions from *ABB*, such as their *MicroSCADA range*. Figure 15 demonstrates an example list of OPC items that contain names with similar strings. This list of OPC items is received by the OPC Process Objects List Tool from ABB.
See Fig. 15 on the next page

The attackers use the string `Abdul` when they add a new OPC group. Possibly this string is used by the attackers as a slang term when referring to the ABB solutions.
See Fig. 16 on the next page

On the final step, the OPC DA payload attempts to change the state of discovered OPC items using the `IOPCSyncIO` interface by writing the 0x01 value twice.
See Fig. 17 on the next page

The component writes the OPC server name, OPC item name state, quality code and value to the log file. The logged values are separated with the following headers:

- [*ServerName: %SERVERNAME%] [State: Before]
- [*ServerName: %SERVERNAME%] [State: After ON]
- [*ServerName: %SERVERNAME%] [State: After OFF]

**Figure 15.**
An example of OPC items names in IN field received using OPC Process Objects List Tool.



**Figure 16.**
The disassembled code of the OPC DA component that uses the `Abdul` string.



**Figure 17.**
Disassembled code of OPC DA payload that uses `IOPCSyncIO` interface.

# DATA WIPER COMPONENT

The data wiper component is a destructive module that is used in the final stage of an attack. The attackers are using this component to hide their tracks and to make recovery difficult.

This component has the filename `haslo.dat` or `haslo.exe` and can be executed by the Launcher component or used as a standalone malicious tool.

Once executed it attempts to enumerate all keys in the registry that list Windows services:

• HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services

It attempts to set the registry value `ImagePath` with an empty string in each of the entries found. This operation will make the operating system unbootable.

The next step is actual deletion of file contents. The component enumerates files with specific file extensions on all drives connected to computer, from C:\ to Z:\. It should be noted that during enumeration the component skips files that are located in subdirectory that contains `Windows` in its name.

The component rewrites file content with meaningless data obtained from newly allocated memory. In order to perform this operation thoroughly the component attempts to rewrite files twice. The first attempt happens once the file is found on a drive. If the first attempt is unsuccessful then the wiper malware makes a second attempt, but before that the malware terminates all processes except those included in a list of critical system processes. The list of these processes is displayed in Figure 18.

To speed up the wiping operation this component rewrites only partial file content at the beginning of the file. The amount of data to be rewritten depends on file size: the smallest amount of data will be rewritten for files less than or equal to 1Mb (4096 bytes); the largest amount of data will be rewritten for files less than or equal to 10Mb (32768 bytes).

Finally, this component attempts to terminate all processes (including system processes) except its own. This will result in the system becoming unresponsive and eventually crashing.



```
off_10010E88    dd offset aAudiodg_exe   ; DATA XREF: _terminate_processes:loc_10001470↑r
                                         ; "audiodg.exe"
                dd offset aConhost_exe   ; "conhost.exe"
                dd offset aCsrss_exe     ; "csrss.exe"
                dd offset aDwm_exe       ; "dwm.exe"
                dd offset aExplorer_exe  ; "explorer.exe"
                dd offset aLsass_exe     ; "lsass.exe"
                dd offset aLsm_exe       ; "lsm.exe"
                dd offset aServices_exe  ; "services.exe"
                dd offset aShutdown_exe  ; "shutdown.exe"
                dd offset aSmss_exe      ; "smss.exe"
                dd offset aSpoolss_exe   ; "spoolss.exe"
                dd offset aSpoolsv_exe   ; "spoolsv.exe"
                dd offset aSvchost_exe   ; "svchost.exe"
                dd offset aTaskhost_exe  ; "taskhost.exe"
                dd offset aWininit_exe   ; "wininit.exe"
                dd offset aWinlogon_exe  ; "winlogon.exe"
                dd offset aWuauclt_exe   ; "wuauclt.exe"
```

**Figure 18.**
List of processes that are not terminated on second rewriting attempt.

The filename masks targeted by the data wiper component to be overwritten are:

SYS_BASCON.COM

| | | |
|---|---|---|
| *.v | *.pcmt | *.bkp |
| *.PL | *.ini | *.log |
| *.paf | *.xml | *.zip |
| *.XRF | *.CIN | *.rar |
| *.trc | *.prj | *.tar |
| *.SCL | *.cxm | *.7z |
| *.bak | *.elb | *.exe |
| *.cid | *.epl | *.dll |
| *.scd | *.mdf | |
| *.pcmp | *.ldf | |
| *.pcmi | *.bk | |

This list contains filename extensions that are used in a standard environment, such as Windows binaries (.exe/.dll), archives (.7z /.tar/.rar/.zip), backup files (.bak/.bk/.bkp), Microsoft SQL server files (.mdf/.ldf), and various configuration files (.ini/.xml). In addition, the component also wipes files that may be used in industrial control systems, such as files written using _Substation Configuration description Language_ (.scl/.cid/.scd) and there are many files and file extensions that are used by various products from ABB. For example, a file named `SYS_BASCON.COM` is used by ABB solutions for storing configuration data, and files with the `.paf (Product Authorization File)` filename extension are used to store license data for ABB MicroSCADA products.

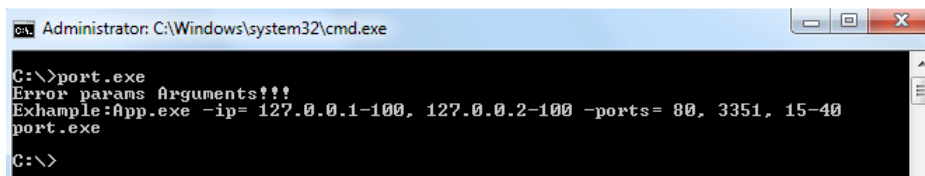# ADDITIONAL TOOLS

## Port scanner tool

The attackers' arsenal includes a port scanner that can be used to map the network and to find computers relevant to their attack. Interestingly, instead of using software already existing, the attackers built their own custom-made port scanner. As is evident from Figure 19, the attacker can define a range of IP addresses and a range of network ports that are to be scanned by this tool.
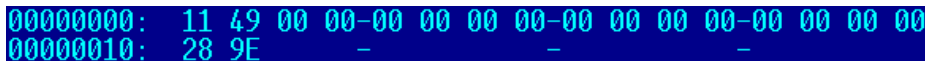See Fig. 19

## DoS tool

Another tool from the attackers' arsenal is a Denial-of-Service (DoS) tool that can be used against Siemens SIPROTEC devices. This tool leverages the _CVE-2015-5374_ vulnerability in order to render a device unresponsive. Once this vulnerability is successfully exploited, the target device stops responding to any commands until it is rebooted manually.

To exploit this vulnerability the attackers hardcoded the device IP addresses into this tool. Once the tool is executed it sends specifically crafted packets to port 50,000 of the target IP addresses using UDP. The UDP packet contains only 18 bytes.
See Fig. 20



**Figure 19.**
The port scanner tool usage example.



**Figure 20.**
Content of UDP packet used during exploitation of CVE-2015-5374.

# CONCLUSION

The investigation behind the Ukrainian power outage last December is still ongoing and it is currently not confirmed that the malware analyzed here was the direct cause. Nevertheless, we believe that to be a very probable explanation, as the malware is able to directly control switches and circuit breakers at power grid substations using four ICS protocols and contains an activation timestamp for December 17, 2016, the day of the power outage.

We can definitely say that the Win32/Industroyer malware family is an advanced and sophisticated piece of malware that is used against industrial control systems. However, it should be noted that the malware itself is just a tool in the hands of an even more advanced and very capable malicious actor. Using logs produced by the toolset and highly configurable payloads, the attackers could adapt the malware to any comparable environment.

The commonly-used industrial control protocols used in this malware were designed decades ago without taking security into consideration. Therefore, any intrusion into an industrial network with systems using these protocols should be considered as "game over".

# INDICATORS OF COMPROMISE (IOC)

## SHA-1 hashes:

```
F6C21F8189CED6AE150F9EF2E82A3A57843B587D
CCCCE62996D578B984984426A024D9B250237533
8E39ECA1E48240C01EE570631AE8F0C9A9637187
2CB8230281B86FA944D3043AE906016C8B5984D9
79CA89711CDAEDB16B0CCCCFDCFBD6AA7E57120A
94488F214B165512D2FC0438A581F5C9E3BD4D4C
5A5FAFBC3FEC8D36FD57B075EBF34119BA3BFF04
B92149F046F00BB69DE329B8457D32C24726EE00
B335163E6EB854DF5E08E85026B2C3518891EDA8
```

## IP addresses of C&C servers:

```
195.16.88[.]6
46.28.200[.]132
188.42.253[.]43
5.39.218[.]152
93.115.27[.]57
```

Warning! Most of the servers with these IP addresses were part of Tor network which means that the use of these indicators could result in a false positive match.

# welivesecurity

news, views and insight from the ESET security community

WeLiveSecurity.com is where ESET experts are. The site is an editorial outlet for internet security news, views and insight. It covers relevant breaking news and aims to cater to all skill levels by offering video tutorials, in-depth features and podcasts.