ExaTrack

Hey Uroburos! What's up ?

ExaTrack - Stéfan Le Berre (stefan.le-berre [at] exatrack.com)

Uroburos is a malware / APT, detected in 2014, that had been a big deal in the world of computer security. It distinguished itself from others by his 64b (rootkit) driver for Windows, including a bypass of PatchGuard. Moreover the driver is not signed, the malware leveraged a vulnerability in a third party driver to get kernel execution. For more details about past research on this malware, you can read the Andrzej Dereszowski and Matthieu Kaczmarek's publication in reference [1].

A few months ago now we have identified an Uroburos/Turla sample dating from 2017. After investigation the driver turned out to be an evolution of the one used in 2014. We had a look at this new driver and discovered that it had some serious differences with the original despite an common base. In this paper we present an analysis of some new behaviours of this 64-bit rootkit.

The analysis will focus on the identification of the rootkit from a memory dump (as we would do during a search for compromise), then we will study a part of its new communication protocol. The goal is to be able to spot the presence of the rootkit remotely without requiring an authentication on the server. It should be noted that the rootkit only targets servers.

The code we will analyze is:
https://www.virustotal.com/en/file/f28f406c2fcd5139d8838b52da703fc6ffb8e5c00261d86aec90c28a20cfaa5b/analysis

To put ourself in the situation of a compromise search on a server, we will use the tool Comae DumpIt (https://www.comae.io/) and will analyze the generated crashdump.

Identification of the core compromise

The driver hides itself quite well in the kernel space, it is not present in the list of loaded modules and the integrity of the other modules is correct.

To assist the analysis we will use an internal tool developed by ExaTrack that aims to check the integrity of the kernel and highlight the potential irregularities present.

The Windows Callback system, which allow arbitrary functions to be called during certain events such as process creation, is among the critical components we check.

In our case, by looking at them we identify an anomaly:

```
>>> ccb
# Check CallBacks
 [*] Checking \Callback\TcpConnectionCallbackTemp : 0xfffffa8002f38360
 [*] Checking \Callback\TcpTimerStarvationCallbackTemp : 0xfffffa8004dfd640
 [*] Checking \Callback\LicensingData : 0xfffffa80024bc2f0
[...]
 [*] PspLoadImageNotifyRoutine
 [*] PspCreateProcessNotifyRoutine
  Callback fffffa8004bc2874 -> SUSPICIOUS ***Unknown*** 48895c2408574881ec30010000488bfa
```

The callbacks in the `PspCreateProcessNotifyRoutine` list are called when creating a process. Adding an entry to it is particularly interesting to modify the data, and therefore the behavior, of a new process. In the previous command, the tool has identified an entry that is considered suspicious because it points to a memory address that is not assigned to a driver.

A second anomaly is present in the callbacks, it is less visible because it does not impact deeply the operation of the system, but it slightly modifies it.

```
 [...]
 [*] IopNotifyShutdownQueueHead
  Name : Null
  Driver Object : fffffa80032753e0
    Driver : \Driver\Null
    Address: fffff88001890000
    Driver : Null.SYS
  Name : 000000a6
  Driver Object : fffffa8003d2adb0
    Driver : \Driver\usbhub
    Address: fffff88000da6000
    Driver : usbhub.sys
[...]
>>> cirp \Driver\Null
    Driver : \Driver\Null
    Address: fffff88001890000
    Driver : Null.SYS
    DriverUnload : fffff88001895100  c:\windows\system32\drivers\null.sys
        IRP_MJ_CREATE                   fffff88001895008  Null.SYS
        IRP_MJ_CREATE_NAMED_PIPE        fffff80002abb1d4  ntoskrnl.exe
        IRP_MJ_CLOSE                    fffff88001895008  Null.SYS
        IRP_MJ_READ                     fffff88001895008  Null.SYS
        IRP_MJ_WRITE                    fffff88001895008  Null.SYS
        IRP_MJ_QUERY_INFORMATION        fffff88001895008  Null.SYS
        IRP_MJ_SET_INFORMATION          fffff80002abb1d4  ntoskrnl.exe
        IRP_MJ_QUERY_EA                 fffff80002abb1d4  ntoskrnl.exe
        IRP_MJ_SET_EA                   fffff80002abb1d4  ntoskrnl.exe
        IRP_MJ_FLUSH_BUFFERS            fffff80002abb1d4  ntoskrnl.exe
        IRP_MJ_QUERY_VOLUME_INFORMATION fffff80002abb1d4  ntoskrnl.exe
        IRP_MJ_SET_VOLUME_INFORMATION   fffff80002abb1d4  ntoskrnl.exe
        IRP_MJ_DIRECTORY_CONTROL        fffff80002abb1d4  ntoskrnl.exe
        IRP_MJ_FILE_SYSTEM_CONTROL      fffff80002abb1d4  ntoskrnl.exe
        IRP_MJ_DEVICE_CONTROL           fffff80002abb1d4  ntoskrnl.exe
        IRP_MJ_INTERNAL_DEVICE_CONTROL  fffff80002abb1d4  ntoskrnl.exe
        IRP_MJ_SHUTDOWN                 fffff80002abb1d4  ntoskrnl.exe
        IRP_MJ_LOCK_CONTROL             fffff88001895008  Null.SYS
        IRP_MJ_CLEANUP                  fffff80002abb1d4  ntoskrnl.exe
        IRP_MJ_CREATE_MAILSLOT          fffff80002abb1d4  ntoskrnl.exe
        IRP_MJ_QUERY_SECURITY           fffff80002abb1d4  ntoskrnl.exe
```

```
        IRP_MJ_SET_SECURITY             fffff80002abb1d4  ntoskrnl.exe
        IRP_MJ_POWER                    fffff80002abb1d4  ntoskrnl.exe
        IRP_MJ_SYSTEM_CONTROL           fffff80002abb1d4  ntoskrnl.exe
        IRP_MJ_DEVICE_CHANGE            fffff80002abb1d4  ntoskrnl.exe
        IRP_MJ_QUERY_QUOTA              fffff80002abb1d4  ntoskrnl.exe
        IRP_MJ_SET_QUOTA                fffff80002abb1d4  ntoskrnl.exe
        IRP_MJ_PNP                      fffff80002abb1d4  ntoskrnl.exe
```

The "\Driver\Null" driver is referenced in the list of modules to be called when the system is shut down. This driver is not supposed to be in this list. However its IRP table does not seem to have been modified and even points to ntoskrnl (for IRP_MJ_SHUTDOWN interface). We did not identify any real usefulness to that behavior.

Let's continue troubleshooting, Windows has many filter systems in its Input Output (IO) handlers. Among these IO there is the network that is divided into several parts, we will take a deeper look at one of those: NetIO.

NetIo also offers a system of callbacks which allow to act on the exchanged network data. These callbacks are called "Callout" and are therefore network callbacks, the structures are not documented and do not appear in the Windows symbols. These properties make it a good place to implant a malware. In the dump we can find 5 callbacks that point to code not belonging, again, to any driver.

```
>>> cnetio
  [*] NetIo Callouts (callbacks) : fffffa8004965000 (4790)
  Callback fffffa8004bd9580 -> SUSPICIOUS ***Unknown*** 488bc448895808488950105556574154
  Callback fffffa8004bca6b0 -> SUSPICIOUS ***Unknown*** 33c0c3cc40534883ec20488b89500100
  Callback fffffa8004bd9af8 -> SUSPICIOUS ***Unknown*** 4883ec286683f91474066683f916750f
  Callback fffffa8004bd9ca0 -> SUSPICIOUS ***Unknown*** 48895c2408488974241057 4883ec4048
  Callback fffffa8004bd9de0 -> SUSPICIOUS ***Unknown*** 4c8bdc49895b0849896b104989731857
```

One of these functions will be studied in more detail later in the document.

Finally we will search for loaded drivers that try to hide from Windows.

```
>>> pe
[...]
  [OK] fffff88001899000 : \SystemRoot\System32\Drivers\Beep.SYS
  [OK] fffff88000da6000 : \SystemRoot\system32\DRIVERS\usbhub.sys
  [NO] fffffa8004bb8000 (Header overwritten)
  [OK] fffff88006a00000 : \SystemRoot\system32\DRIVERS\E1G6032E.sys
  [OK] fffff880017d2000 : \SystemRoot\System32\Drivers\Npfs.SYS
[...]
>>> dq fffffa8004bb8000 100
 FFFFFA8004BB8000 0000000300000000  0000FFFF00000004  ....?...?...¦¦..
 FFFFFA8004BB8010 00000000000000B8  0000000000000040  ........@.......
 FFFFFA8004BB8020 0000000000000000  0000000000000000  ................
 FFFFFA8004BB8030 0000000000000000  000000D800000000  ................
 FFFFFA8004BB8040 CD09B4000EBA1F0E  685421CD4C01B821  ??.?....!.?L.!Th
 FFFFFA8004BB8050 72676F7270207369  6F6E6E6163206D61  is program canno
 FFFFFA8004BB8060 6E75722065622074  20534F44206E6920  t be run in DOS
```

```
 FFFFFA8004BB8070 0A0D0D2E65646F6D  0000000000000024  mode....$.......
 FFFFFA8004BB8080 095520395A3B417D  0955203909552039  }A;Z9 U.9 U.9 U.
 FFFFFA8004BB8090 095520A609542039  0955203C092E28A6  9 T.. U..(..< U.
 FFFFFA8004BB80A0 0955203B0928E61E  095520510938E61E  ?.(.; U.?.8.Q U.
 FFFFFA8004BB80B0 09552038092FE61E  09552038092DE61E  ?./.8 U.?.-.8 U.
 FFFFFA8004BB80C0 0955203968636952  0000000000000000  Rich9 U.........
 FFFFFA8004BB80D0 0000000000000000  0006866400000000  ............d.?.
 FFFFFA8004BB80E0 000000005900F3CF  202200F000000000  ...Y.........."
 FFFFFA8004BB80F0 00042E000008020B  000000000001BC00  ??....?...?.....
>>> list fffffa8004bb8000 fffffa8004bbb000
    FFFFFA8004BB8000 rwx-
    FFFFFA8004BB9000 rwx-
    FFFFFA8004BBA000 rwx-
```

Here again, an important anomaly is observable. A driver is present in memory and has overwritten its MZ and PE headers probably to hide itself from raw memory search. Its addressing corresponds to the callbacks we encoutered before and it is mapped with RWX rights.

All the elements support the idea that a malware operates from the kernel. We will now analyze some of its code (mainly network communication) to get an idea of how it works.

Driver analysis

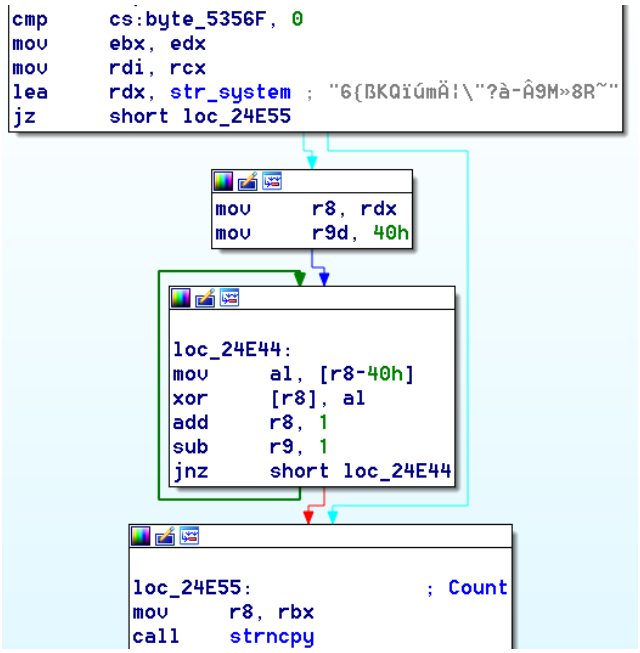
Entry point

During its initialization the driver quickly targets the "Null" device. It retrieves a pointer to the object and registers it in the list of "shutdown" callbacks mentioned previously. It also registers its callback called during process creations.

```
[...]
      if ( (unsigned int)get_top_deviceObjet(L"\\Device\\Null", &device_obj_null)
        && (result = get_top_deviceObjet(L"\\Device\\Beep", &device_obj_null),
(_DWORD)result) )
      {
        __asm { xchg    rbx, qword ptr cs:isNullDeviceFailed }
      }
      else
      {
        v5 = IoRegisterShutdownNotification(device_obj_null);
        if ( v5 || (drvobj_null = device_obj_null->DriverObject, (v5 =
sub_4E21C(byte_1188D)) != 0) )
[...]
            PsSetCreateProcessNotifyRoutine(cbCreateProcess, 0i64);
[...]
```

Encrypting strings

To avoid easy identification by anti-virus all of the Uroburos related strings are encrypted. Each block of encrypted data is 0x40 bytes, a XOR operation is performed with the previous 0x40 bytes.



The decryption function can be represented as follows (an oneliner is simpler for IDA). It is thus possible to decipher all the chains.

```
Python>def decrypt(addr, clen): return ''.join(chr(b) for b in
[struct.unpack('B'*clen,idaapi.get_many_bytes(addr,64))[a] ^
struct.unpack('B'*clen,idaapi.get_many_bytes(addr-clen,clen))[a] for a in xrange(clen)])

Python>[decrypt( 0x53530 + (i*0x80) , 0x40).replace("\x00",'') for i in xrange(38)]
['system', 'isapi_http', 'isapi_log', 'isapi_dg', 'isapi_openssl', 'shell.{F21EDC09-85D3-
4eb9-915F-1AFA2FF28153}', 'Hd1', 'Hd2', 'RawDisk1', 'RawDisk2', 'wininet_activate',
'dmtev', 'Ultra3', 'Ultra3', 'services_control', 'fixdata.dat', '$NtUninstallQ817473$',
'fdisk.sys', 'fdisk_mon.exe', '400', '16', '{AAAA1111-2222-BBBB-CCCC-DDDD3333EEEE}',
'~WA434.tmp', '~WA4276.tmp', '.', '~WA356.tmp', 'rasmon.dll', 'rasman.dll', 'user',
'internat', 'NTUSER.DAT', 'ntuser.dat.LOG1', '.', 'mscrt.dll', 'msvcrt.dll', '0', '1',
'.']
```

A YARA rule of the cryptographic function is provided in the appendix.

Network interception

As seen above, network callbacks are installed. They will be registered through the function "FwpsCalloutRegister0" (which allows to add a network filter) and allows the driver to transfer or not the data received.

```
  v20 = addCalloutAddress(
          &stru_14930,
          &a2,
          DeviceObject,
          (__int64)intercept_packet,
          (__int64)&ret_null,
          (__int64)a6,
          (__int64)&v47,
          (__int64)&v34,
          &a9,
          &a10);
```

The "intercept_packet" function (at address fffffa8004bd9580 in the memory dump) will analyze the data passing through the network connections. Interestingly, it will not look at data passing through port 139. For the other ports it will only look at data received and only if the host is the server.

```
  if ( v9 || LOWORD(a1->layerId) == 20 && a1->pIP_infos->src_port == 139 )
    return;
  if ( LOWORD(a1->layerId) == 22 && a1->pIP_infos->src_port == 139 )
    return;
[...]
    fwpsCopyStreamDataToBuffer0(v8, datas_tcp_buffer, *(_QWORD *)(v8 + 48), &v31);
[...]
      buffer_type_2 = find_and_decode_datas(datas_tcp_buffer, v24, *((_DWORD *)v11 +
0x1FF) == 0, &a4a);
```

The "find_and_decode_datas" function is responsible for testing the different accepted communication protocols. In our case we will study an HTTP communication. The goal is to see if it is possible, remotely, to determine if a server is compromised by Uroburos.

The malware validates that the received message is a standard HTTP request. Then he will look for a covert message in one of the arguments of the HTTP header.

```
  if ( space_offset_1 != 3i64
    || ((v18 = *(_WORD *)Buf < (unsigned __int16)str_GET, *(_WORD *)Buf !=
(_WORD)str_GET)
     || (v19 = Buf[2], v18 = (unsigned __int8)v19 < BYTE2(str_GET), v19 !=
BYTE2(str_GET)) ? (v20 = -v18 - (v18 - 1)) : (v20 = 0),
        v20) )
  {
    if ( space_offset_1 != 4 || *(_DWORD *)Buf != str_POST )
      return 0i64;
  }
[...]
    if ( *(_DWORD *)start_word_2 != *(_DWORD *)"http://"
[...]
  if ( v33 != *(_DWORD *)"HTTP/" || (v35 = v32[4], v34 = v35 < aHttp_0[4], v35 !=
aHttp_0[4]) )
[...]
    || !(unsigned int)check_and_decode_buffer(&Buf[nextline], v14, response_tag,
out_decoded_b64, v7) )
[...]
```

The "check_and_decode_buffer" function will look for the first ':' and will try to find a covert message in the corresponding HTTP argument.

```
      v15 = memchr(v10, ':', (unsigned int)(v14 - 1));
      if ( !v15 || *((_BYTE *)v15 + 1) != ' ' )
        return (unsigned int)v5;
```

This is followed by a check of the message by means of several checksums.

```
    v10 = get_checksum(datas, 12);
    result = (unsigned __int64)reverse_littleendian(v10) == *(_DWORD *)v4;
```

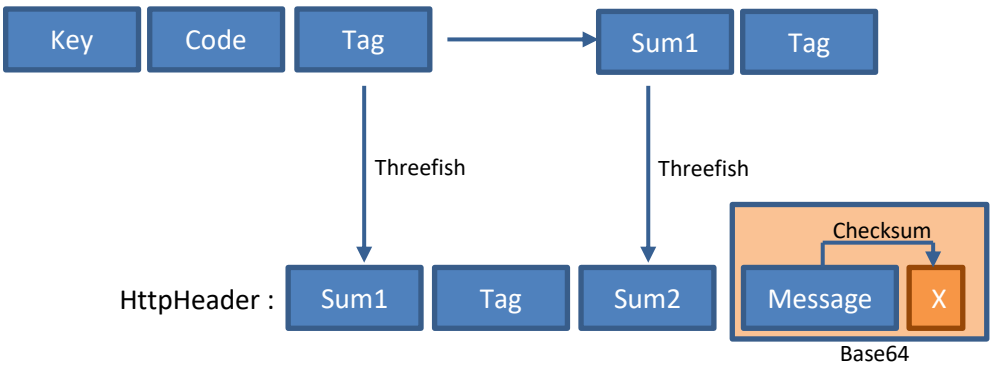
The checksum function use a modified "threefish256" algorithm truncated to 4 bytes.

```
    crypto_it(v3, v6, (v5 - 1) >> 5, 32);
[...]
    memcpy((void *)(v3 + *(_QWORD *)(v3 + 8) + 64), v6, v5);
    *(_QWORD *)(v3 + 8) += v5;
```

The hash function is used three times to obtain a final hash, the first time with a static buffer that will allow to initialize the data, the second with the data to hash and a last with the hash of this data. This final hash will be truncated on 4 bytes and used as checksum. A key is initialized, our guess is that there is one per target. It allows to have a shared secret between the malware and the operator. It will be used in the calculation of hashes but will not be sent in the request.

The message is contained at the end of the line, it is 7 bytes + 1 byte integrity check which is only an sum of the previous 7 bytes. This 8 bytes block is then base64 encoded.

The HTTP request data will be adjusted as follows:



It is interesting to note that the "Code" element is not present in the final query. Four values can be used and they are bruteforced by the malware during the verification.

When a server is contacted with a request of this type (on a port already open) it is the rootkit that answers us (the data will not be transferred to the userland).

If the message corresponds to the expected format the driver will send a response of variable size and padded with random bytes.

```
    if ( reply_datas[6] & 2 )
    {
      v8 = 8 * (rand() % -32);
      v4 = v8;
      v9 = &v21[-v8];
      if ( v8 )
      {
        v10 = v8;
```

```
        do
        {
          *v9++ = rand();
          --v10;
        }
        while ( v10 );
      }
    }
  }
  *(_BYTE *)(v7 + 0xBE0);
  sprintf(Dest, "HTTP/1.1 200 OK\r\nContent-Length: %u\r\nConnection: %s\r\n\r\n",
(unsigned int)(v4 + 8));
```

Only the first 8 bytes respond to a specific format, all other data are random. The integrity of this block is enforced by an additional checksum between the first 7 bytes, with the result stored in byte 8. This cheksum is similar to the one mentioned above.

We are therefore able to develop a PoC to remotely check if a server is compromised by this version of the malware:

```
> request_builder.py 192.168.48.133 8080
datas :
 0000000000000000 E8 F6 E8 4E 72 61 03 EA  C8 B3 DD 8D 25 D0 26 12  ...Nra♥.....%.&↕
 0000000000000010 B7 F9 50 E5 8C D2 01 62  A0 37 2F FB AD C8 91 DA  ..P...☺b.7/.....
 0000000000000020 44 A5 53 C7 1D 76 0E 4D  AC AF F7 18 F4 12 57 A2  D.S.↔v♫M...↑.↕W.
 0000000000000030 A0 75 3B 0F 50 C5 6C 55  31 4B A1 9F D0 2E F4 F4  .u;☼P.lU1K......
 0000000000000040 30 39 93 13 1A DF B8 A2  B4 7C DB 88 55 DE 26 98  09.‼→....|..U.&.
 0000000000000050 98 04 29 6F AF 25 CF 9F  FA F5 90 0D D8 23 E9 97  .♦)o.%.......#..
  [*] checksum OK - Host is compromised
```

This PoC is available on our website [2].

Differences and similarities between Uroburos 2014 and 2017

Several changes and non-change are surprising compared to the 2014 version of the malware. This list is really not exhaustive but allows to realize the state of the project as of today.

The names of the files remained the same and the name of the service too, which makes it easily detectable by any IOC.

The driver is always loaded with a VirtualBox exploit. A kernel vulnerability exploitation is therefore done at each reboot of the system.

The PatchGuard bypass has been removed, which necessarily limits kernel alterations later.

```
Driver 2014:
  if ( v2 )
    installService(v3);
```

```
  v4 = PG_bypass();
  if ( v4 )
    goto LABEL_16;
  ObjectAttributes.Length = 48;
  ObjectAttributes.RootDirectory = 0i64;

Driver 2017:
    if ( v2 )
      installService(cp_DriverObject);
    ObjectAttributes.RootDirectory = 0i64;
    ObjectAttributes.SecurityDescriptor = 0i64;
```

The `\Driver\Null` driver is still used and the `\Device\FWPMCALLOUT` device is still attached to it.

```
>>> drv_stack \Driver\Null
- Stack of device name : \Device\FWPMCALLOUT
  Driver Object : fffffa80032753e0
    Driver : \Driver\Null
    Address: fffff88001890000
    Driver : Null.SYS

- Stack of device name : \Device\Null
  Driver Object : fffffa80032753e0
    Driver : \Driver\Null
    Address: fffff88001890000
    Driver : Null.SYS
```

The string encryption system has remained the same. The registering of the "Null" driver in `IopNotifyShutdownQueueHead` is a good idea in itself but we have not seen its use. A possible use of this callback would be the possibility to inject registry key for persistance at shutdown time.

The checksum system has evolved, the malware uses Threefish and the message's format has evolved since the 2014 version. The goal was probably to escape the network signatures. However the correction of a English typo `221 Service closing transmi`**`tt`**`ion channel` to give `221 Closing transmi`**`ss`**`ion channel` allows us to date the driver as more recent (see reference [1]).

Overall the rootkit still has serious assets, but also some carelessness, such as the names of files and registry keys, suggesting that it will only be used on isolated servers. Despite an apparent decreased popularity, kernel malware (64b) are still present and are not ready to disappear because their presence is harder to identify than userland components.

References

[1] Paper of "Andrzej Dereszowski" and "Matthieu Kaczmarek", one of moste valuables on this subject:
http://artemonsecurity.com/uroburos.pdf

[2] PoC of http request:
http://www.exatrack.com/public/uroburos_poc.py

[3] SNAKE CAMPAIGN & CYBER ESPIONAGE TOOLKIT:
http://artemonsecurity.com/snake_whitepaper.pdf

Appendix

YARA rule :

```
rule Sig
{
  strings:
    $strings_crypt = { 4d 8b c1 41 ba 40 00 00 00 41 ?? ?? ?? 41 ?? ?? 49 83 c0 01 49 83
ea 01 75 ??}
    $hash_part1 = { 49 c1 c3 0e 4e ?? ?? ?? 4c 33 dd 4c 03 c7 4c 03 c1 48 c1 c0 10 49 33
c0 4d 03 c3 48 03 e8 48 c1 c8 0c 48 33 c5 49 c1 cb 07 4d 33 d8 4c 03 c0 49 03 eb 49 c1 c3
17 4c 33 dd 48 c1 c8 18 49 33 c0 4d 03 c3 48 03 e8 49 c1 cb 1b 4d 33 d8 4c 03 df 4c 03 d9
48 c1 c0 05 48 33 c5 4a ?? ?? ?? ??}
  condition:
    1 of them
}
```