EternalBlue – Everything There Is To Know

By deugenio

Published: 2017-09-29 · Archived: 2026-04-07 15:29:23 UTC

Research By: Nadav Grossman

Introduction

Since the revelation of the EternalBlue exploit, allegedly developed by the NSA, and the malicious uses that followed with WannaCry, it went under thorough scrutiny by the security community. While many details were researched and published, several remained in the dark, and an end-to-end explanation of the vulnerability and exploit is nowhere to be found.

For these reasons, we endeavored to conduct a comprehensive research of the vulnerabilities and exploitation related to EternalBlue, by using reverse engineering, kernel debugging and network analysis. The results of this research appear below. We hope that this paper will enhance the understanding of EternalBlue, and help the security community combat it, and the likes of it in the future.

Background

The EternalBlue exploitation tool was leaked by “The Shadow Brokers” group on April 14, 2017, in their fifth leak, “Lost in Translation.” The leak included many exploitation tools like EternalBlue that are based on multiple vulnerabilities in the Windows implementation of SMB protocol.

EternalBlue works on all Windows versions prior to Windows 8. These versions contain an interprocess communication share (IPC$) that allows a null session. This means that the connection is established via anonymous login and null session is allowed by default. Null session allows the client to send different commands to the server.

The NSA created a framework (much like Metasploit) named FuzzBunch, which was part of the leak. The purpose of this framework is to configure, for example, set victim `ip`, and execute the exploitation tools.

Microsoft released patches for the vulnerabilities in the leak, under the **MS17-010** (Microsoft Security Bulletin).

CVE-2017-0144 is the CVE ID in MS17-010 that is related to EternalBlue.

SMB Protocol

Server Message Block (SMB), one version of which is also known as Common Internet File System (CIFS), operates as an application-layer network protocol mainly used for providing shared access to files, printers, and serial ports and miscellaneous communications between nodes on a network.

It also provides an authenticated inter-process communication mechanism. Most SMB use involves computers running Microsoft Windows, where it was known as the “Microsoft Windows Network” before the subsequent introduction of Active Directory. Corresponding Windows services are LAN Manager Server (for the server component) and LAN Manager Workstation (for the client component) [Wikipedia].

Bug Explanations

EternalBlue exploits 3 bugs (named as Bug [A,B,C]) to achive RCE, the explanation for each bug are listed below:

- Bug A (i.e. “Wrong Casting Bug”)
- Bug B (i.e. “Wrong Parsing Function Bug”)
- Bug C (i.e. “Non-paged Pool Allocation Bug”)

Bug A (i.e. “Wrong Casting Bug”):

A bug in the process of converting FEA (File Extended Attributes) from Os2 structure to NT structure by the Windows SMB implementation (srv.sys driver) leads to buffer overflow in the non-paged kernel pool.

A bit about FEA (File Extended Attributes):

The FEA structure is used to describe file characteristics. For more information, see https://en.wikipedia.org/wiki/Extended_file_attributes.

You can think about the FEA structure as key and value pairs, where the key is `AttributeName` and the value is `AttributeValue`.

For example:

In Os2 format, the struct looks like this:

After you convert the FEA from Os2 format to Windows format (Nt Format), the structure looks like this:

Note– These structures are not valid `Structs`, because the length of the AttributeName and AttributeValue changes (is not a constant). However, it's simpler to understand the structures using this illustration.

Functionality:

The functions listed below are part of the srv.sys driver and they are related to the bug.

`SrvOs2FeaListToNt` – Converts Os2 FEA List to NT FEA List.

The logics of this function are:

1. Gets `Os2FeaList`
2. Calls `SrvOs2FeaListSizeToNT` to get the appropriate size for `NtFeaList`.
3. Allocates a buffer from the non-paged pool according to the size returned from `SrvOs2FeaListSizeToNT`.
4. Iterates over the `Os2FeaList` until it reaches the `SizeOfListInBytes` (from `Os2FeaList`). In each iteration, it calls `SrvOs2FeaToNT` to convert the `Os2Fea` record to NT format and add it to `NtFealist`.

`SrvOs2FeaListSizeToNT` – Calculates the size needed to convert `Os2FeaList` structures into the appropriate `NtFeaList` structures.

The logics of this function are:

1. Calculates the needed buffer size for `NtFeaList`. The buffer size is calculated according to the required size to convert `Os2FeaList` struct to `NtFeaList` struct (this is the return value from the function).

2. Calculates how many records (in bytes) of `Os2Fea` from `Os2FeaList` should be converted to `NtFea` to store them later (by `SrvOs2FeaListToNt`) in `NtFeaList`. The result of the calculation is stored in the member of `Os2FeaList`, named `SizeOfListInBytes`, by overwriting the previous value .
   Note – If there are **no** invalid Os2Fea records, the `SizeOfListInBytes` remains **untouched**.
   However, if there is an invalid\overflowed Os2Fea record(s) in Os2FeaList, the value of `SizeOfListInBytes` is shrunk.
   For example: If part of the FEA is in the range of the `SizeOfListInBytes` and the remainder of the FEA is "out of range" (bigger than `SizeOfListInBytes`). It will ignore this FEA and any further "out of range" FEA and shrink the `SizeOfListInByte` to the size of all the "valid" FEAs.
   Note – `SizeOfListInBytes` doesn't restrict the SMB packet size.
   See the example / illustration below

`SrvOs2FeaToNT` – Converts the `Os2Fea` record to an `NtFea` record. The structure is not identical to the `NtFea`, but it's quite similar.
*Note that the size of the `NtFea` record is bigger than `Os2Fea` because it contains another field named `NextEntryOffset`. There is also an alignment of 4 bytes between the `NtFea` records.

`SrvOs2FeaListSizeToNT` **shrinking illustration:**

Before Shrinking:

After Shrinking: if the size of `SizeOfListInBytes` is below 2^{16}:

After Shrinking (bug): if the size of `SizeOfListInBytes` is above 2^{16}:

The Buggy Code:

In a nutshell: There is a wrong casting in the `SrvOs2FeaListSizeToNT` function, in its second logic (shrinking the `SizeOfListInBytes` member of `Os2FeaList`). That leads to a small buffer (i.e. `NtFeaList`) and large data (out of buffer boundary) that will be stored on it after conversion (i.e. `SizeOfListInBytes`).

Instead of shrinking the `SizeOfListInBytes`, the function enlarges it.
The remainder `SizeOfListInBytes` indicates how many records (in bytes) of `Os2Fea` should be converted to `NtFea`.

However, the size of the `NtFeaList` buffer that returns from the function (`NtFeaListSize`) is calculated correctly for the appropriate size to convert a shrunk `Os2FeaList` to `NtFeaList`.

This leads to a situation in which the size of bytes that should be copied to the buffer (`NtFeaList`, which depends on the updated value `SizeOfListInBytes` member of `Os2FeaLis`) **is bigger** than the buffer size (`NtFeaListSize`). This causes an Out of Bound write.

More details:

`SizeOfListInBytes` is a DWORD size member of `Os2FeaList`. If shrinking is needed, the function treats `SizeOfListInBytes` as a Word member and it updates only the 2 bytes, instead of 4 bytes (with the shrunk size). The 2 most significant bytes remain untouched. This bug causes `SizeOfListInBytes` to be enlarged instead of shrunk.

If the value of `SizeOfListInBytes` is in the range of 2^{16}, there is no problem and the function works as expected. However, if the value of `SizeOfListInBytes` is above the range of 2^{16}, it could be enlarged instead of shrunk.
In the 2 examples below, the first is of a normal scenario (if the bug wouldn't have existed) and the second is of the buggy scenario.

Functions pseudo code:

`* The important lines are marked in Yellow`

`SrvOs2FeaListToNT`:

`SrvOs2FeaListSizeToNT`:

- Line 32 is the buggy line. It updates only the size of Word (LOWORD) from the Dword `SizeOfListInBytes`

`SrvOs2FeaToNT:`

Bug B (i.e. “Wrong Parsing Function Bug”):

When you transmit a file over SMB protocol, there are several data-related functions:

1. SMB_COM_TRANSACTION2: Sub-commands provide support for a richer set of server-side file system semantics. The “ `Trans2` subcommands”, as they are called, allow clients to set and retrieve Extended Attribute key/value pairs, make use of long file names (longer than the original 8.3 file format names), and perform directory searches, among other tasks.

2. SMB_COM_NT_TRANSACT: Sub-commands extend the file system feature access offered by `SMB_COM_TRANSACTION2` , and also allow for the transfer of **very large parameter and data blocks.**

If the data sent via `SMB_COM_TRANSACTION2` or by `SMB_COM_NT_TRANSACT` exceeds the `MaxBufferSize` established during session setup, or `total_data_to_send` is bigger than `transmitted_data` , then the transaction uses the `SECONDARY` sub-command.

Each sub-command has a corresponding sub-command `_SECONDARY` . This `_SECONDARY` is used when the data sent is too big for a single packet. It is therefore split over a few packets to fulfill “the total size of data to be sent” that was declared in the first packet. The packets that follow the first sub-command have the corresponding `_SECONDARY` sub-command set as their command.

Example:

```
SMB_COM_NT_TRANSACT => SMB_COM_NT_TRANSACT_SECONDARY
SMB_COM_TRANSACTION2 => SMB_COM_TRANSACTION2_SECONDARY
```

In `SMB_COM_TRANSACTION2,` the maximum data that can be sent is represented by a parameter in the header of `SMB_COM_TRANSACTION2` in the field of a `Word` size. The same is true for the `SMB_COM_TRANSACTION2_SECONDARY` .

However, in `SMB_COM_NT_TRANSACT` , the maximum data that can be sent is represented by a parameter in the header of `SMB_COM_NT_TRANSACT` in the field of `Dword` size. The same is true for the `SMB_COM_TRANSACTION2_SECONDARY` .

Therefore, there is a difference between the amounts of data that can be sent in `SMB_COM_TRANSACTION2,` where the maximum data length is represented in a `Word` (max `0xFFFF`), and in `SMB_COM_NT_TRANSACT` where the

maximum is represented in a `Dword` (`0xFFFFFFFF`).

However, as there is no validation for which function started the transaction (`SMB_COM_TRANSACTION2` or `SMB_COM_NT_TRANSACT`), parsing is according to the last transaction type.

Thus, it's possible to send:

`SMB_COM_NT_TRANSACT` followed by `SMB_COM_TRANSACTION2_SECONDARY`

This situation can lead to wrong data parsing, and this bug enables `Bug_A` by treating `Dword` as `Word` .
It happens when `SMB_COM_NT_TRANSACT` (`Dword`) is followed by `SMB_COM_TRANSACTION2_SECONDARY` (`Word`).
This leads to parsing the data, wrongly, as if it originally came from a transaction of the `SMB_COM_TRANSACTION2` type.

Bug C (i.e. "Non-paged Pool Allocation Bug"):

In a nutshell: There is a bug that lets you allocate a chunk with a specified size in the kernel non-paged pool with the specified size. It is used in the heap grooming phase when creating a hole that later will be filled with a data size that causes an out of bound write to the next chunk (`Bug A` & `Bug B`).

An `SMB_COM_SESSION_SETUP_ANDX` request MUST be sent by a client to begin user authentication on an SMB connection and establish an SMB session.

This command is used to configure an SMB session. At least one `SMB_COM_SESSION_SETUP_ANDX` MUST be sent to perform a user logon to the server and to establish a valid UID.

There are 2 formats for an `SMB_COM_SESSION_SETUP_ANDX` request:

The first is used for `LM` and `NTLM authentication` , documented here. This is the format of the request:

The second is used for `NTLMv2` (`NTLM SSP`) authentication, documented here. This is the format of the request:

In both formats, the request is split into 2 sections:

- `SMB_Parameters` – Contains parameters of sizes between 1-4 bytes. The `WordCount` field represents the total length of `SMB_Parameters` struct members in a `Word` size.
- `SMB_Data` – Contains data in a variable size. The `ByteCount` field represents the length of the `SMB_Data` struct members section in bytes.

Summing the size of the fields, in the first format, the `WordCount` equals 13 and in the second format (`extended security`), the `WordCount` equals 12.

The server does an integrity check, with a function named `SrvValidateSmb` , for SMB packets that include the format of `SMB_Parameters` and `SMB_Data` .

Although there is no bug in the `SrvValidateSmb` function, there is a bug in the extraction of `SMB_DATA` by a function named `BlockingSessionSetupAndX`.

By triggering the bug, the `BlockingSessionSetupAndX` function wrongly calculates `ByteCount`, which leads to an allocation of controlled size – bigger than the packet data – in the non-paged pool.

Here is the buggy part:

The `SMB_COM_SESSION_SETUP_ANDX` request is handled by the `BlockingSessionSetupAndX` function. Below is pseudo code for handling both request formats (only the relevant part is shown).

From the pseudo code above, we see that if we send an `SMB_COM_SESSION_SETUP_ANDX` request as `Extended Security` (`WordCount` 12) with `CAP_EXTENDED_SECURITY`, but **without** `FLAGS2_EXTENDED_SECURITY`, the request will be processed **wrongly** as an NT Security request (`WordCount` 13, marked in yellow).

In this case, the `GetNtSecurityParameters` function is called, but it calculates the `SMB_DATA` wrongly. The request is in `Extended Security` (`WordCount` 12) format, but the function intends to parse it as `NT Security request` (`WordCount` 13).
As a result, it reads `ByteCount` from the wrong offset in the struct, and allocates space in the non-paged kernel pool for `NativeOs` and `NativeLanMan` unicode strings (see the structure above), according to the wrong offset of ByteCount.

This bug allows you to send a small packet that leads to a big allocation in the non-paged pool, which is used to create a big allocation as a placeholder.

This allocation will later be freed (creating a `HOLE`) and allocated again by an `NtFea` chunk that will overflow the next chunk. This issue is explained further in the **exploitation flow** section.

Exploitation Technique

*** See the glossary section for an explanation of terms.**

Primitives:

1. `MDL` (`pMdl1`) Overwrite – When we get a write primitive to an `MDL` , we can map an I/O data to a specific virtual address. In the exploit we have an OOB (out-of-bound) write in `srv allocation` (`Bug A and Bug B`). We can therefore overwrite the header of `srvnet` chunk (using some sort of grooming). Note that in the header of `srvnet` chunk there is a `MDL` As explained in the **glossary section, a** virtual address in the MDL maps the incoming data from the client to a specific virtual address. Therefore, if we change it by some sort of overwriting, the data from the client is mapped to the address that would have been overwritten. It enables the client data (that we control) to be written wherever we want (controlling the `MDL`).
2. `pSrvNetWskStruct` Overwrite – `pSrvNetWskStruct` is located in the `srvnet` header struct and points to the `SrvNetWskStruct` . This struct contains a pointer to the function `HandlerFunction` , which is called when the related `srvnet` connection is closed. If we overwrite the pointer to the `SrvNetWskStruct` struct (`pSrvNetWskStruct`) with an address that contains a **fake** `SrvNetWskStruct` whose values we control, we can control the value of the pointer to the function that is called when the `srvnet` connection is closed (`HandlerFunction`), which leads to RCE.

Using the 2 primitives for RCE:

As explained earlier, if we overwrite the `MDL` (primitive 1), we can control where the data under our control (data sent from the client to the server, over a connection that is related to `srvnet`) would be written.
If we write (overwrite) in the virtual address (`StartVA` field) of the `MDL` a static address like an address within `HAL's Heap` , the data that the client sends will be mapped to a virtual address within the `HAL's Heap` .
(`HAL's Heap` has execute permission in Windows versions prior to Windows 8).
To execute that data, which have been written to the `HAL's Heap` (by primitive 1) we must combine it with primitive 2.
If we change the pointer to `SrvNetWskStruct` (`pSrvNetWskStruct`) to point to the same address that was crafted in the `MDL` (static address on the `HAL's Heap`), in that address (`HAL's Heap`) we can create a **fake struct** (`SrvNetWskStruct`) preceded by the shellcode. The shellcode would then be called upon closing the related srvnet connection, and we achieve RCE.

Notes: The `SrvNetWskStruct` struct contains a pointer to a function (`HandlerFunction`) that is called when `srvnet` connection is closed. The pointer to function (`HandlerFunction`) should point to our shellcode.

Exploitation Flow

Step 0:

Assume that this is the initial state of the non-paged kernel pool and the `HAL's Heap`.

Step 1:

`srv allocation` according to `Bug A and Bug B`. In this step, only the connection for `Os2Fea` transmission is opened.

Step 2:

Fills part of the `OS2Fea` first by `SMB_COM_NT_TRANSACT`. Later, it is filled by `SMB_COM_NT_TRANSACT_SECONDRY` or `SMB_COM_TRANSACTION2_SECONDARY` as described in `Bug A` and `Bug B`, but without sending the last `SECONDARY` packet (this does not yet trigger the OOB write).

Step 3:

Open multiple `srvnet` connections to increase the chances of `srvnet` overwriting (overflow) by preceding the `srv allocation` of converted `OS2Fea` to `NtFea`. Using `Bug A` and `Bug B`, this leads to overflowing the next chunk (`srvnet Header`). This technique is also used as a **grooming technique**.

Step 4:

At this point, the attacker creates a chunk according to `Bug C`. This chunk is used as a placeholder for the converted `Os2Fea` to `NtFea`. It should be the same size as the overflowing `NtFeaList` (wrong calculation, due to `Bug A`). This chunk is later freed by closing the connection, and `NtFea` (write out-of-bound) is allocated instead (fills the hole).

Before it is freed, more `srvnet` chunks (new `srvnet` connections) are allocated.

It is freed just before the final packet of `srv allocation` (`SMB_COM_TRANSACTION2_SECONDARY`) that allocates a chunk for storing the `NtFea` converted data.

Most likely, this allocation is stored in this freed chunk. It's part of the **grooming technique**.

Step 5:

New `srvnet` allocation (by a new connection). This `srvnet` chunk is located after “the chunk of `Bug C`.” If the `NtFea` is located in the previous chunk (“the chunk of `Bug C`“), it would lead to an overflow of this `srvnet` chunk.

Step 6:

The “chunk of `Bug C`” is freed.

Step 7:

1. The last `Os2Fea` data is sent (using `SMB_COM_TRANSACTION2_SECONDARY`). This leads to allocation to convert `Os2Fea` to `NtFea`.

2. The `NtFea` is allocated at the free hole (previously was allocated with the chunk according to `Bug C`).

3. `NtFea` overwrites (overflow) the next chunk, which is a `srvnet` chunk. The `srvnet header` is overwritten and modifies these 2 header properties to point to the same address in the `HAL's Heap`:

– `pSrvNetWskStruct`, the pointer to the struct that includes the function that is called when the connection is closed.
– `MDL` that is used to map the next incoming data (from the user) in this `srvnet` connection.

Step 8:

1. Incoming data in the overflowed `srvnet` connection from the user arrives.
2. Because of the previous overwrite of the `MDL`, this data is written to the `HAL's Heap`, instead of the reserve place in the kernel pool (the value of the `MDL` before the overwrite).
3. The data from the user that was written to the `HAL's heap` contains a **fake struct** (`SRVNET_RECV`).
4. `pSrvNetWskStruct` points to the **fake struct** (`SRVNET_RECV`) in the `HAL's heap`. When the connection closes, the `HandlerFunction` from the **fake struct** is called.
5. The data from the user after the fake struct contains the shellcode, `DoublePulsar`. It is written after the fake struct to the `HAL's Heap`.

Step 9:

1. All the `srvnet` connections are closed.
2. In each `srvnet` connection, the `HandlerFunction` pointed at by the `SRVNET_RECV` is executed. However, in the overflowed srvnet connection, the pointer to the `SRVNET_RECV` (`pSrvNetWskStruct`) is fake and points to the fake struct in the `HAL's heap` .
3. The fake `HandlerFunction` is executed, but this function is the **shellcode**.

Network Analysis

This is an edited wireshark pcap of a successful exploitation on Windows 7. It includes only one exploitation try; in reality, it usually takes more than a single attempt. We added relevant filters, coloring and comments. Responses were omitted for readability.

- The comments next to each packet explain the purpose of the packet in the exploitation flow.

Glossary

SMBv1:

Client systems use the Common Internet File System (CIFS) Protocol to request file and print services from server systems over a network. CIFS is a stateful protocol, in which clients establish a session with a server and use that session to make a variety of requests to access files, printers, and inter-process communication (IPC) mechanisms, such as named pipes. CIFS imposes state to maintain an authentication context, cryptographic operations, file semantics such as locking, and similar features.

The Server Message Block (SMB) Version 1.0 Protocol extends the CIFS Protocol with additional security, file, and disk management support. These extensions do not alter the basic message sequencing of the CIFS Protocol but introduce new flags, extended requests and responses, and new information levels. All of these extensions follow a request/response pattern in which the client initiates all of the requests.

SMBv2:

The Server Message Block (SMB) Protocol Versions 2 and 3 supports the sharing of file and print resources between machines. The protocol borrows and extends concepts from the Server Message Block (SMB) Version 1.0 Protocol.

These drivers are related to SMB protocols:

- `srv.sys` – Related to `SMBv1` protocol
- `srvnet.sys` – Related to `SMBv2` protocol

srv allocation:

Some `SMBv1` packets from the client to the server cause allocation in a paged or non-paged kernel pool. `Bug A` and `Bug B` use the `SMBv1` protocol that leads to `srv allocation`. The out-of-bound write happens in `srv allocation`.

srvnet allocation:

Some `SMBv2` packets from the client to the server cause allocation in a paged or non-paged kernel pool. In the exploit, the srvnet chunk is allocated, so we can overflow it with `srv allocation`. By doing this, we overwrite a structure in the srvnet header that leads to RCE (Remote Code Execution)

SRVNET_RECV Struct:

This struct is pointed to by the `pSrvNetWskStruct` variable in the `SRVNET_HEADER`. This is used by `srvnet.sys` to call the handler function (which points to the shellcode address in the EternalBlue scenario) when the connection is closed.

SRVNET_HEADER Struct:

This struct resides in the beginning of the srvnet related chunk. In the exploitation, we use/manipulate it by overwriting these fields:

- `pSrvNetWskStruct` : Pointer to the `SRVNT_RECV` struct
- `pMdl1` : Pointer to a `MDL`. This maps received data to virtual address space.
  After the srvnet connection is established, it waits for another data packet from the client. When it arrives, it maps the received data to the address specified by the `MDL`.

MDL:

A memory descriptor list (MDL) is a system-defined structure (kernel structure) that describes a buffer by a set of physical addresses. A driver performs direct I/O receives a pointer to an MDL from the I/O manager, and reads and writes data through the MDL.

An I/O buffer that spans a range of contiguous virtual memory addresses can be spread over several physical pages, and these pages can be discontinuous. The operating system uses an MDL to describe the physical page layout for a virtual memory buffer. For more info, see references 1, 2.

Controlling the `MDL` lets you “write-what-where” the primitive.
When `MDL` is “locked”, specific physical pages are mapped to specific virtual address space.
For example: It is possible to map the entire virtual page range to one physical page. No copy is needed because every request (for example, read/write) to a specific virtual range will fetch the same physical page.

Structure

Illustration

[https://i-msdn.sec.s-msft.com/dynimg/IC454060.gif]

`HAL's Heap` – The Windows HAL's heap is located at 0xffffffff'ffd00000 in 64-bit and at 0xffd00000 in 32-bit (the address was static until Windows10). This area is executable in Windows versions prior to Windows 8 / Server 2012.

Summary

In this research paper, we attempted to shed light on important sections of the EternalBlue vulnerabilities and exploit, and provide a full step-by-step explanation of the causes for the vulnerabilities and their exploitation. It is our belief that with better understanding of the roots of EternalBlue, we will be able to improve the security of users around the world.

As important as EternalBlue is, it is not the first, nor the last major exploit that enables hackers to take complete control over entire networks. Unfortunately, nefarious exploits continue to appear, as was recently shown by Microsoft itself. Users and organizations cannot wait for a second round against cyber threats, and must protect themselves with the most up-to-date protections, capable of detecting and blocking any cyber-attack on its tracks.

I would also like to thank my colleagues Omri Herscovici, Yannay Livneh, and Michael Kajiloti for their help in this research.

The Check Point IPS blade provides protections against these threats:
Microsoft Windows EternalBlue SMB Remote Code Execution

References

https://gist.github.com/worawit/bd04bad3cd231474763b873df081c09a
https://github.com/worawit/MS17-010
http://blog.trendmicro.com/trendlabs-security-intelligence/ms17-010-eternalblue/
https://www.rapid7.com/db/modules/exploit/windows/smb/ms17_010_eternalblue
https://www.risksense.com/_api/filesystem/466/EternalBlue_RiskSense-Exploit-Analysis-and-Port-to-Microsoft-Windows-10_v1_2.pdf
https://en.wikipedia.org/wiki/EternalBlue
https://msdn.microsoft.com/en-us/library/windows/hardware/ff565421(v=vs.85).aspx
https://msdn.microsoft.com/en-us/library/windows/hardware/dn614012(v=vs.85).aspx
https://msdn.microsoft.com/en-us/library/cc246328.aspx
https://msdn.microsoft.com/en-us/library/ee441849.aspx
https://msdn.microsoft.com/en-us/library/cc246233.aspx#gt_e1d88514-18e6-4e2e-a459-20d5e17e9078
https://msdn.microsoft.com/en-us/library/cc246233.aspx#gt_ee1ec898-536f-41c4-9d90-b4f7d981fd67
https://msdn.microsoft.com/en-us/library/ee441741.aspx
https://msdn.microsoft.com/en-us/library/ee915515.aspx
https://msdn.microsoft.com/en-us/library/ff359296.aspx
https://msdn.microsoft.com/en-us/library/ee441551.aspx
https://msdn.microsoft.com/en-us/library/ee442101.aspx
https://msdn.microsoft.com/en-us/library/windows/hardware/ff545793(v=vs.85).aspx
https://en.wikipedia.org/wiki/The_Shadow_Brokers
https://steemit.com/shadowbrokers/@theshadowbrokers/lost-in-translation

---