Blackswan Technical Writeup

Microsoft 0-day Vulnerabilities

Erik Egsgard
Principal Security Developer

Published: October 14, 2021

FIELD EFFECT

Executive Summary

Six privilege elevation vulnerabilities and one information leak vulnerability were discovered that allow code running as a non-privileged user to gain full access to kernel memory, and therefore full access to the entire system. These vulnerabilities were confirmed to be present in all versions of Windows since early 2007, introduced with Windows Vista. Reliable exploits were demonstrated for four of the privilege escalation vulnerabilities and the information leak.

Microsoft has proceeded to patch all vulnerabilities between July 13th, 2021 and October 12th, 2021.

Table of Contents

Table of Contents

Definitions

AFD - Ancillary Function Driver

ALPC - Advanced Local Procedure Call

BSOD - Blue Screen of Death

DoS - Denial of Service

EoP - Elevation of Privilege

InfoLeak - Information Leak or Information Disclosure

IOCTL - Device Input and Output Control

IPC - Inter-Process Communication

LPC - Local Procedure Call

LPE - Local Privilege Elevation

QoS - Quality of Service

RCE - Remote Code Execution

RPC - Remote Procedure Call

TDI - Transport Driver Interface

TDX - TDI Extension

TOCTOU - Time-of-Check Time-of-Use

Introduction

During the course of my normal day job, I needed a deeper understanding of the internals of Windows ALPC. Over a couple of weeks of research this ended up leading down a rabbit hole to the reverse engineering of several undocumented Windows kernel components and the discovery of multiple privilege elevation vulnerabilities and one information leak.

This paper will cover some of the areas of the Windows kernel that were explored and describe the details of the discovered vulnerabilities and how they can be exploited.

In order to understand the details of the vulnerabilities the reader should be familiar with several Windows and vulnerability exploitation concepts:

- **Windows Kernel IO**
- **Advanced Local Procedure Call**
- **Windows Sockets**
- **Time-of-Check Time-of-Use Vulnerabilities**

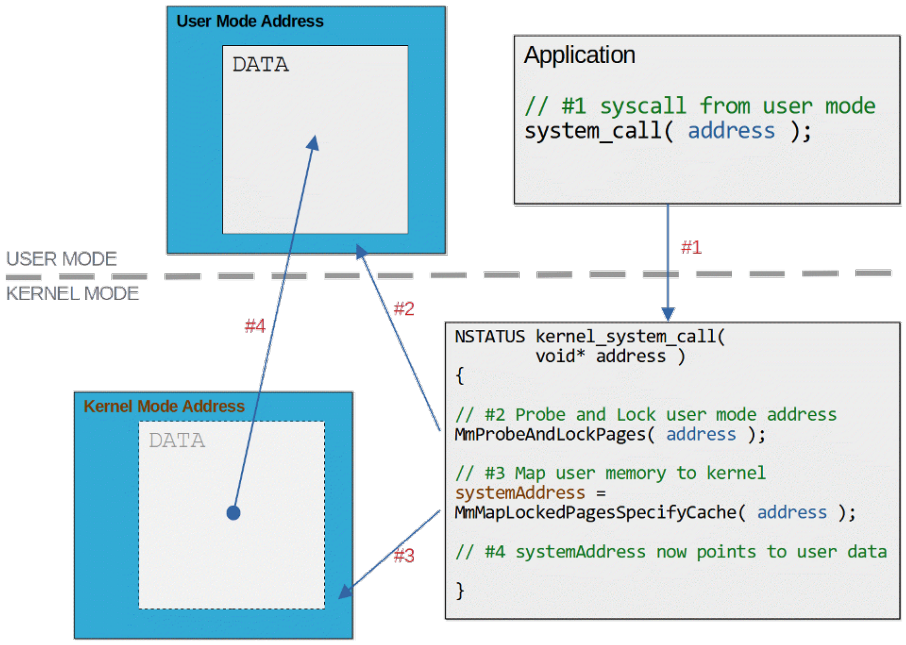
Windows Kernel IO

User mode applications interact with the kernel by making system calls. Applications normally use higher level Windows APIs (such as Win32) but they can still make system calls directly. Most system calls take arguments from user mode and often involve passing pointers to memory buffers. The kernel is responsible for safely capturing the data from user mode memory (or copying data to user mode memory) and handling malicious applications that might unmap memory or change its contents.

The kernel provides a set of helper functions to safely access user memory (*ProbeForRead, ProbeForWrite*, etc.). These functions ensure that the memory addresses provided reside in the user mode address space, are properly aligned and are mapped. Once probed, the kernel can read the data in user memory, although any access must be done inside an exception handler to avoid attempted tampering, such as the user mode application unmapping the memory.

To protect against malicious user mode code modifying data, kernel code should read the data once and save a copy of it for processing. For performance reasons, it is sometimes not desirable to copy all of the user mode data into kernel memory. The other common pattern is to lock the user memory *(MmProbeAndLockPages)*, which prevents it from being unmapped, and map an address in the kernel address space to point at the same pages of memory *(MmMapLockedPagesSpecifyCache)*. This allows kernel code to safely access the memory outside of an exception handler. However, using an exception handler is still recommended as the user mode application can still modify the data.

FIGURE 1: KERNEL IO - LOCK AND MAP MEMORY



Device Driver IOCTL Buffer Handling

One way for user mode to communicate with the kernel is to open a kernel device and then use IO Control Messages (IOCTLs) via the *NtDeviceIoControlFile* system call. Just like other system calls a driver must safely access user memory when handling IOCTLs.

When processing IOCTLs a driver has several options for deciding how to define messages, based on the IO type: *Buffered*, *Direct* or *Neither*. With *Buffered* IO any user buffers are fully copied into kernel memory, whereas with *Direct* IO the user mode buffers are locked into memory and mapped to a kernel address. With the final type, *Neither* IO, the driver receives the original user buffers and is responsible for capturing/writing the memory safely.

Buffered IO is the safest IO type as there is no risk of user mode unmapping the memory or modifying its contents, however, it has more processing overhead than *Neither* IO. *Direct* IO protects from memory being unmapped, and also has more processing overhead than *Neither* IO and does not protect against user mode applications from modifying data.

ALPC

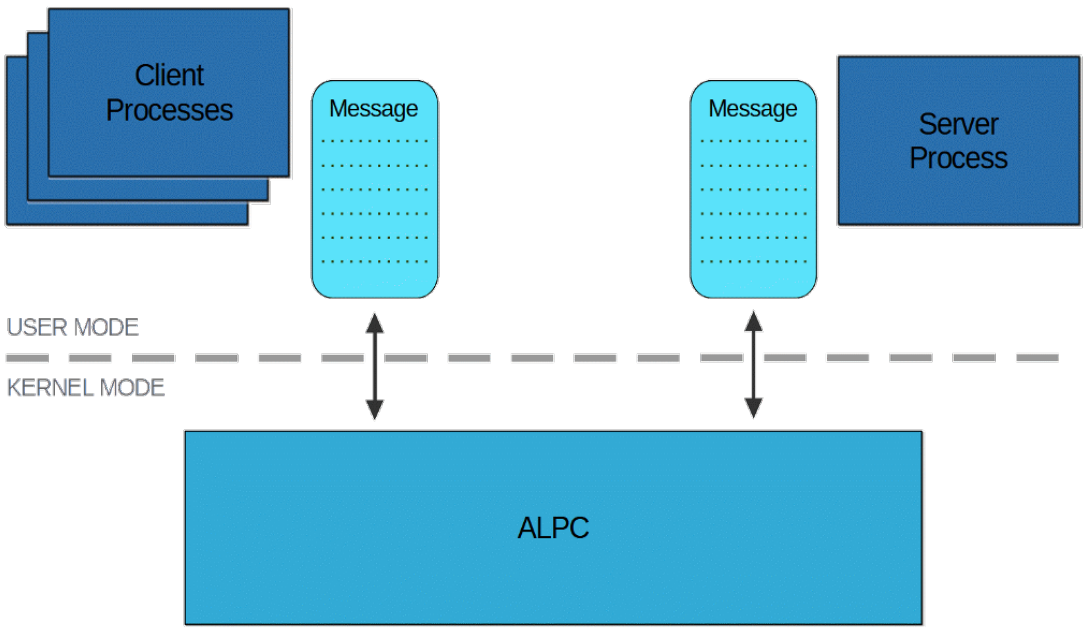
Local Procedure Call (LPC) is a communication mechanism used by Windows for communication between two user mode processes, a user mode process and the kernel, or between two kernel drivers. An example of LPC would be applications sending messages to LSASS.exe (Local Security Authority Subsystem Service) to complete SSL handshakes.

Microsoft introduced the Advanced Local Procedure Call (ALPC) system with Windows Vista as a higher performance method of Inter-Process Communication (IPC). The older LPC interface enforced synchronous communication between client and server endpoints, while all communication in ALPC is asynchronous. The LPC interface remains in Windows for backwards compatibility but internally it is redirected to the newer ALPC system. Both the LPC and ALPC interfaces are not documented and intended only for internal use by other Windows components.

ALPC is accessible through kernel system calls whose usage is very similar to traditional socket programming. The server process creates a named ALPC port to begin listening for client connections, with one or more clients connected concurrently. When a client connects, the server decides to accept or reject the connection and if accepted messages can start to be exchanged.

One of the performance improvements in ALPC is the optional use of IO Completion Ports which provide an efficient method of processing asynchronous IO requests.

FIGURE 2: BASIC ALPC ARCHITECTURE



ALPC System Calls

ALPC exposes a relatively small API to user mode applications. The API is undocumented and applications should use a higher level API for IPC, such as Remote Procedure Call (RPC). But the ALPC system calls can be invoked directly if desired and they are usually accessible from any sandbox or process context.

FIGURE 3: ALPC SYMBOLS FROM WINDBG

```
1: kd> x /1 /p /n nt!NtAlpc*
nt!NtAlpcAcceptConnectPort<no parameter info>
nt!NtAlpcCancelMessage(void)
nt!NtAlpcConnectPort<no parameter info>
nt!NtAlpcConnectPortEx<no parameter info>
nt!NtAlpcCreatePort<no parameter info>
nt!NtAlpcCreatePortSection(void)
nt!NtAlpcCreateResourceReserve(void)
nt!NtAlpcCreateSectionView(void)
nt!NtAlpcCreateSecurityContext(void)
nt!NtAlpcDeletePortSection(void)
nt!NtAlpcDeleteResourceReserve<no parameter info>
nt!NtAlpcDeleteSectionView(void)
nt!NtAlpcDeleteSecurityContext(void)
nt!NtAlpcDisconnectPort<no parameter info>
nt!NtAlpcImpersonateClientContainerOfPort<no parameter info>
nt!NtAlpcImpersonateClientOfPort(void)
nt!NtAlpcOpenSenderProcess(void)
nt!NtAlpcOpenSenderThread(void)
nt!NtAlpcQueryInformation(void)
nt!NtAlpcQueryInformationMessage(void)
nt!NtAlpcRevokeSecurityContext<no parameter info>
nt!NtAlpcSendWaitReceivePort(void)
nt!NtAlpcSetInformation(void)
```

To illustrate a simple use case of the ALPC system calls, the following is example pseudocode of client and server connecting, respectively:

FIGURE 4: SERVER ALPC ENDPOINT CODE EXAMPLE

```
RtlInitUnicodString( &serverPortName, L"AlpcExampleServer" );
InitializeObjectAttributes( &objAttr, &serverPortName, 0, NULL, NULL );
status = NtAlpcCreatePort( &serverPort, &objAttr, NULL );

if( STATUS_SUCCESS == status )
{
    // Wait for a connection request message
    status = NtAlpcSendWaitReceivePort( serverPort,
                                        0,
                                        NULL,
                                        NULL,
                                        &connectMessage,
                                        &messageLength,
                                        &messageAttr,
                                        NULL );

    if( STATUS_SUCCESS == status && LPC_REQUEST == connectMessage.type )
    {
        status = NtAlpcAcceptConnectPort( &clientPort,
                                          serverPort,
                                          0,
                                          NULL,
                                          NULL,
                                          NULL,
                                          &connectMessage,
                                          &messageAttr,
                                          TRUE );

        // ... Exchange messages with client ...

        NtAlpcDisconnectPort( clientPort, 0 );
        CloseHandle( clientPort );
    }

    CloseHandle( serverPort );
```

FIGURE 5: CLIENT ALPC ENDPOINT CODE EXAMPLE

```
// Connect to the server with no initial message
RtlInitUnicodString( &serverPortName, L"AlpcExampleServer" );
status = NtAlpcConnectPort( &clientPort,
                            &serverPortName,
                            NULL,
                            NULL,
                            0,
                            NULL,
                            NULL,
                            NULL,
                            NULL,
                            NULL,
                            NULL );

if( STATUS_SUCCESS == status )
{
    // ... Exchange messages with server ...

    CloseHandle( serverPort );
```

Both the server and client endpoints can perform optional security checks when establishing connections, including checking expected SIDs and privileges.

Messaging

ALPC messages contain application specific data, and can also contain additional message metadata called attributes. The metadata information can include information such as: message IDs, impersonation details, and shared memory information.

The normal operation to send a small message is:

01 Message copied from sender's process into kernel memory

02 Copy of message saved on receive port's queue

03 Receiving process asks for new messages

04 Message is copied from kernel memory into receiver's process.

If larger messages are going to be sent, then a shared memory view can be created between the client and server and message data exchanged via the view.

Completion Lists

One of the performance improvements included in ALPC is the use of shared memory between client and server, thus reducing the number of times a message has to be copied. Even if the client is not using shared memory the server can map a shared region with the kernel called a completion list. The kernel copies incoming messages into this list until the user mode server is ready to receive them.

User mode sets the shared memory for the list which must be page aligned and a multiple of page size in length. Allowable sizes are between 4kB and 1GB. The completion list is made up of several sections, each list starts with a header that contains offsets for each section and some other common fields.

FIGURE 6: COMPLETION LIST HEADER

```
typedef struct {
    uint64_t StartMagic;
    uint32_t TotalSize;                 // Length of shared region
    uint32_t ListOffset;                // Region offset to List section
    uint32_t ListSize;
    uint32_t BitmapOffset;              // Region offset to Allocation Bitmap
    uint32_t BitmapSize;
    uint32_t DataOffset;                // Region offset to Message Data section
    uint32_t DataSize;
    uint32_t AttributeFlags;            // Set when the completion list is created
    uint32_t AttributeSize;
    _ALPC_COMPLETION_LIST_STATE State;  // Contains Head & Tail index for List
    uint32_t LastMessageId;
    uint32_t LastCallbackId;
    uint32_t PostCount;
    uint32_t ReturnCount;
    uint32_t LogSequenceNumber;
    _RTL_SRWLOCK UserLock;
    uint64_t EndMagic;
```

The List section is for keeping track of pending messages in the completion list and consists of offsets into the Data section. Indices for the head and tail of the list are stored in the State header field.

Allocation of chunks in the Data section is tracked using the Bitmap section. Each bit in this section represents a 64-byte chunk in the Data section.

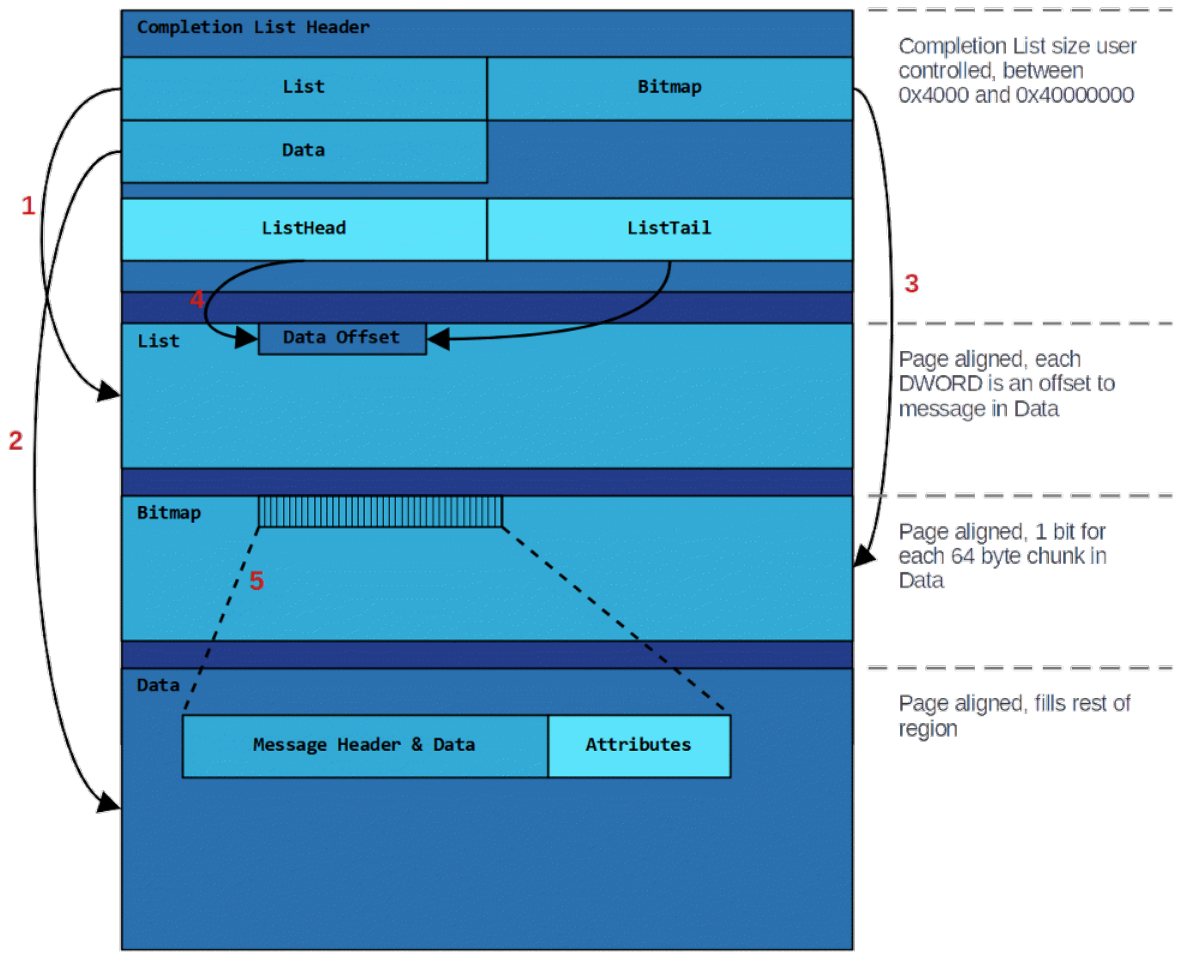
FIGURE 7: ALPC MESSAGE HEADER

```
typedef struct {
    int16_t DataLength;          // Length of data
    int16_t TotalLength;         // Length of data & header
    int16_t Type;
    int16_t DataInfoOffset;
    _CLIENT_ID ClientId;
    uint32_t MessageId;
    uint64_t ClientViewSize;
} _PORT_MESSAGE;
```

Messages all have a common header containing metadata (size, type, etc). All messages in a completion list have the same set of attributes. Any attribute data is stored directly after the message in the Data section.

The figure below shows the region offsets in the completion list header pointing to the List Section (#1), Data Section (#2) and Bitmap Section (#3). There is only one message in the completion list and the List Section contains the offset of this message. The start of the List is indicated by the ListHead field in the header (#4). Space for the message has been reserved by setting the corresponding bits in the Bitmap Section (#5).

FIGURE 8: COMPLETION LIST MEMORY LAYOUT

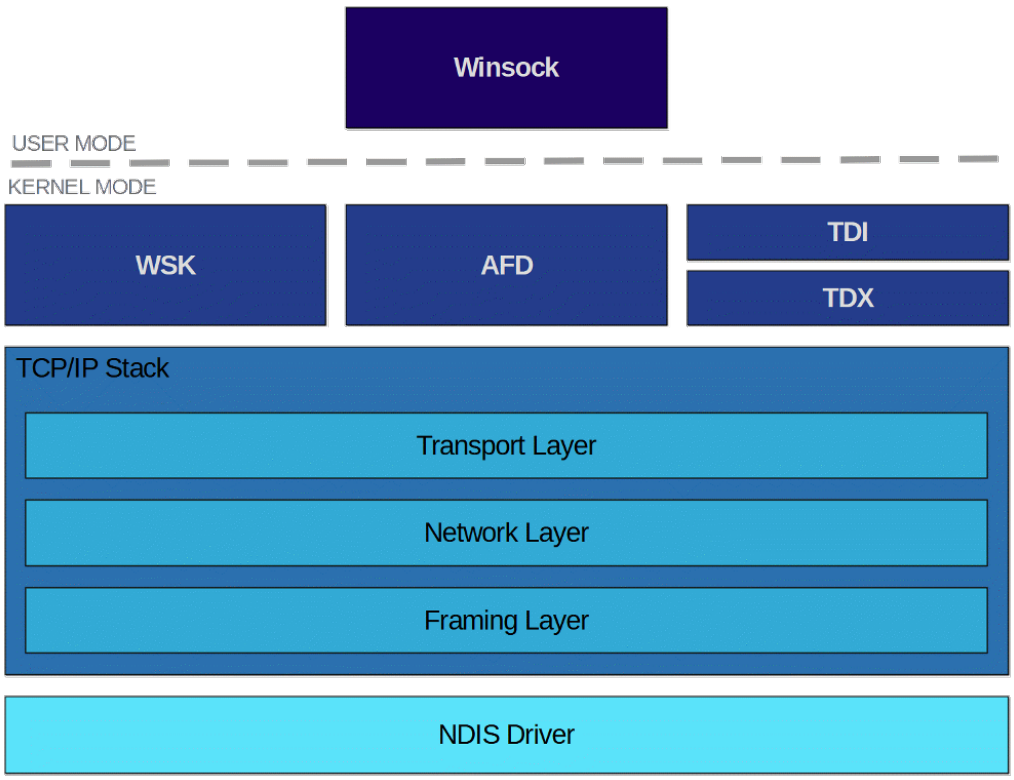


Windows Sockets

Windows socket programming is done using the winsock API. Most other operating systems use the POSIX socket API for socket programming which is usually implemented as a syscall interface to the kernel. Microsoft eventually added a mostly POSIX socket compliant API to winsock, but the kernel interface is still the largely undocumented Ancillary Function Driver (AFD).

While AFD is the user mode interface to the kernel TCP/IP stack, there are two interfaces that can be used by kernel drivers: the Transport Device Interface (TDI) and Winsock Kernel (WSK). TDI is a legacy interface for drivers to access the transport layer of the windows networking stack (which originally included TCP/IP, NetBIOS and AppleTalk transport providers). The networking stack was redesigned with Windows Vista and the TDI Extension (TDX) driver maps TDI functionality to the new TCP/IP interface. WSK provides a socket-like interface to the TCP/IP stack for all newer drivers.

FIGURE 9: WINDOWS TCP/IP STACK (VISTA AND NEWER)



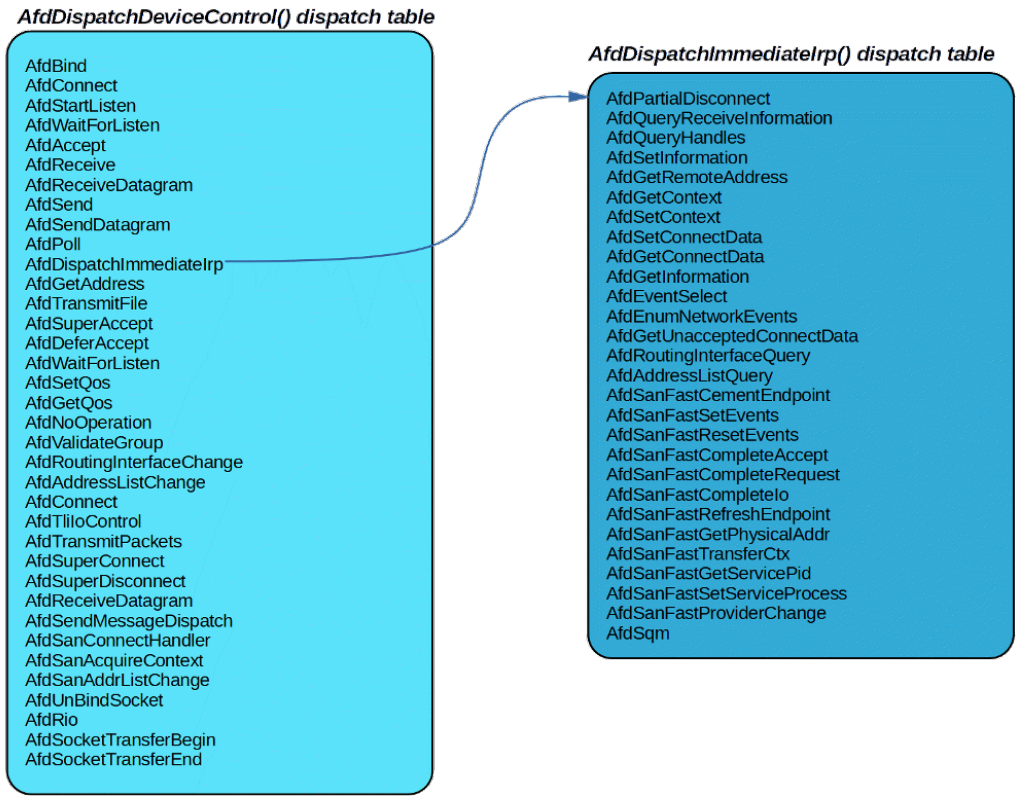
Ancillary Function Driver

AFD exposes a relatively complex interface to user mode which is only intended for use by the user mode winsock library. However, any application that can use sockets can also communicate with AFD directly.

To communicate with AFD, a handle must first be opened to the desired resource. For a socket handle, this is done by opening the AFD device with extended attributes set to an open packet structure. The open packet specifies transport parameters (i.e. name, address family, socket type, protocol) which are used to match with transport for socket creation. Default transport layer providers (in *AfdTlTransportListHead*) are: TCP, UDP, Raw, and Unix sockets.

Communication with AFD is done through IOCTL calls, which are dispatched to one of 79 different functions. The following figure shows the list of functions in the dispatch and immediate dispatch tables.

FIGURE 10: DISPATCH TABLE FOR AFD IOCTLS



Socket Creation

In order to send IOCTL calls to the AFD driver, a handle must be opened. Several types of endpoints can be opened depending on the contents of the extended attributes buffer. Normal socket endpoints are created by using an extended attribute named *'AfdOpenPacketXX'* where the contents are a structure specifying the typical socket creation options (Family, Type & Protocol) and an optional transport name.

The created endpoint has options and flags set based on the contents of the open packet and these determine how the endpoint will behave when handling IOCTL calls. The transport name is the path of a lower level TDI device object (i.e. *'\Device\Tcp'*), and if specified, an internal handle to the device will be opened.

Socket Options

A common operation on socket handles is to set and retrieve properties of different layers of the TCP/IP stack, the typical API functions used are: getsocketopt, setsocketopt and ioctlsocket. These functions call through the AfdTliIoControl IOCTL. This interface is interesting because the requests are passed down to multiple endpoints on lower-level drivers (i.e. TCPIP.sys).

When calling AfdTliIoControl the following structure is passed in:

FIGURE 11: SOCKET IOCTL STRUCTURE

```
typedef struct
{
    uint32_t Type;   // 0 = internal only, 1 = setsockopt(),
                     // 2 = getsockopt(), 3 = ioctlsocket()
    uint32_t Level;  // i.e. SOL_SOCKET = 0xffff
    uint32_t IoctlCode;
    uint8_t  Endpoint;
    void*    InputOutputBuffer;
    uint32_t BufferLength;
} AFD_TL_IO_CONTROL;
```

The kernel validates this structure to make sure that only allowed combinations of Type and Level are permitted using the following call:

FIGURE 12: SOCKET IOCTL VALIDATION FUNCTION

```
char AfdAllowedUserIOControlRequest(int Type, int Level, char IsEndpoint)
{
    // Only allow Endpoint calls
    if (IsEndpoint == 1) {
        if (Type == 0x1 || Type == 0x2) {
            // Restrict SOL_* value for setsockopt()/getsockopt()
            if (Level != 0xfffc && Level != 0xfffd) {
                return 1;
            }
        }
        else {
            // Allow all ioctlsocket()
            if ((Type == 0x3) && (Level == 0x0)) {
                return 1;
            }
        }
    }
    return 0;
}
```

There is also some special handling for a couple of specific IOCTL codes: SIO_SET_QOS and SIO_RESERVED_1. The special handling functions are *AfdTliIoControlHandleSetQos* and *AfdTliIoControlHandleAssociateQos* and both process the input buffer and rewrite with kernel pointers.

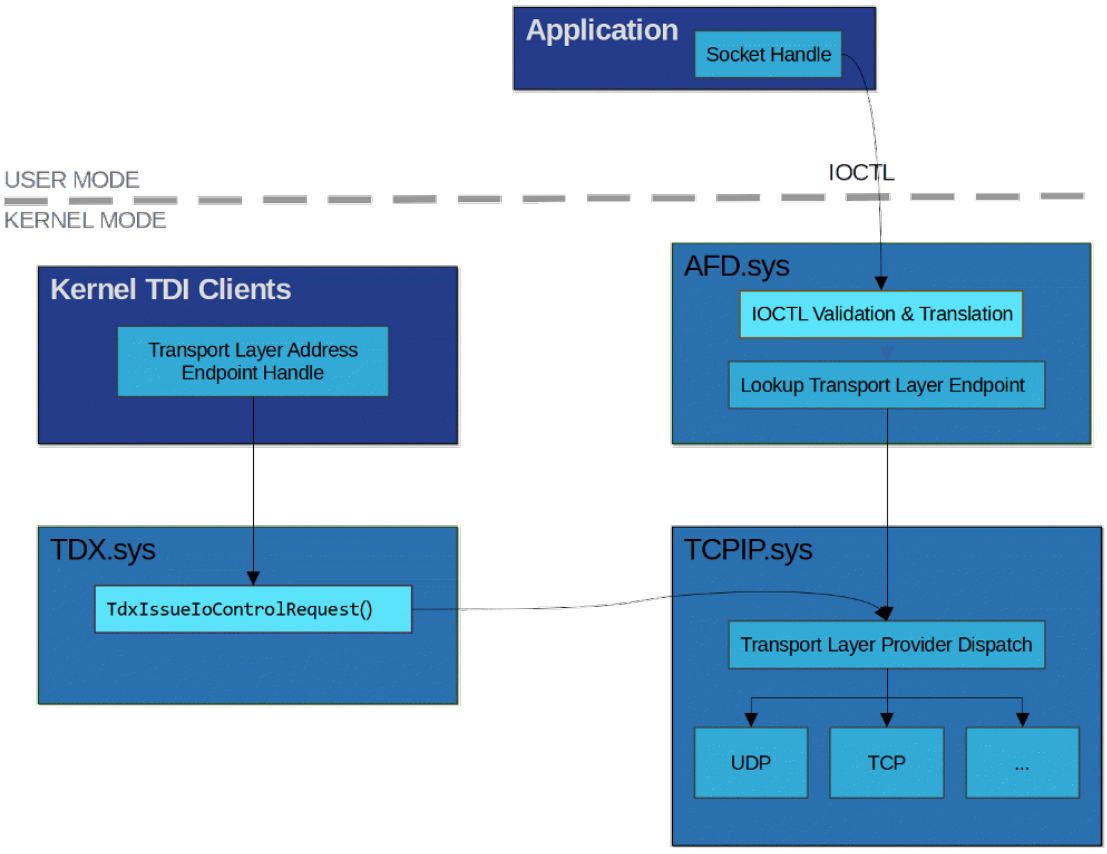
Once validated the data is forwarded to lower-level drivers (i.e. *TCPIP.sys*), in particular the QOS options with special handling are eventually processed by *PACER.sys*.

Transport Driver Interface Extension

As previously mentioned, the TDI driver is the legacy kernel mode interface to the network stack and the TDX driver maps the legacy functionality to the TCP/IP stack. The TDX device can be opened directly from user mode but most of the limited functionality is only available through one of the TDI transport devices: *'/Device/Udp'*, *'/Device/Tcp6'*, etc. User mode programs are only able to open control channel endpoint to the TDI transports (see *TdxCreateControlChannel*), while kernel mode can create connection and network address endpoints.

Kernel mode clients can issue a wider range of commands to TDX endpoints, including internal IO control requests to transport address and connection endpoints. These internal IO requests, handled by *TdxIssueIoControlRequest*, are passed down the TCP/IP stack in the same way as *ioctlsocket & setsockopt* calls but – as kernel clients are implicitly trusted – they don't go through the validation and translation functions in AFD.

FIGURE 13: AFD IOCTL PROCESSING



When calling *TdxIssueIoControlRequest* the following structure is passed in, which is very similar to the structure used by AFD in *AfdTliIoControl*:

FIGURE 14: TDX IOCTL STRUCTURE

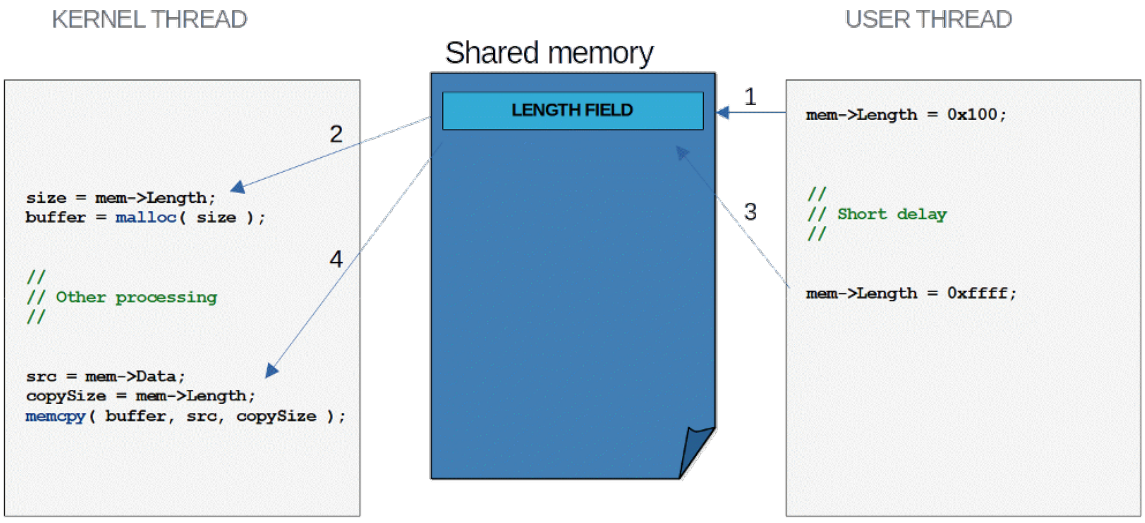
```
typedef struct
{
    uint32_t Type;   // 0 = internal only, 1 = setsockopt(),
                     // 2 = getsockopt(), 3 = ioctlsocket()
    uint32_t Level;  // i.e. SOL_SOCKET = 0xffff
    uint32_t IoctlCode;
    uint8_t  Endpoint;
    void*    InputBuffer;
    uint32_t InputLength;
    void*    OutputBuffer;
    uint32_t OutputLength;
} TDX_IO_CONTROL;
```

TOCTOU Vulnerabilities

It is important at this point to describe a class of vulnerabilities known as Time-Of-Check-Time-Of-Use or TOCTOU vulnerabilities. When processing untrusted input, an application must check that the input is within the expected range of values before using the data. A TOCTOU vulnerability is a race condition where the input can be changed by an attacker between the initial validation of the input and its use. This is a well-known class of vulnerability for kernel system calls that processes user input. Despite being well known these types of bugs are relatively hard to spot.

As a performance enhancement many system calls do not copy all input from user mode into kernel buffers. Instead, the user mode addresses may be accessed directly from kernel or the backing pages may be locked and mapped to kernel addresses. In both cases, a malicious user mode application can modify the memory while the kernel is accessing it thus exposing potential TOCTOU vulnerabilities if the kernel code is not careful.

FIGURE 15: EXAMPLE TOCTOU



1. User thread sets Length to 0x100
2. Kernel thread reads Length and allocates 0x100 bytes
3. User thread sets Length to 0xffff
4. Kernel thread reads new length and copies 0xffff bytes **<== Overflow**

CVE-2020-7460 - FreeBSD TOCTOU Vulnerability

The following code snippet shows the CVE-2020-7460 vulnerability from the *freebsd32_copyin_control* function which illustrates how TOCTOU vulnerabilities work.

This function parses an array of socket control messages and copies them to a kernel buffer. The incoming buffer is in user mode, and the *copyin* function ensures the buffer is valid user memory. The control messages are iterated through, calculating the total size required (#1). Then enough kernel memory is allocated to store all the control messages (#2). A second loop is used to copy the control messages into the kernel buffer, but the length of each control message is read from the user mode buffer a second time (#3). At this point the user application could have altered the length to be larger than when it was first read (#1). If this happens, then when the control message is copied into the kernel buffer (#4), an overflow will occur.

FIGURE 16: CVE-2020-7460 FREEBSD TOCTOU

```
static int
freebsd32_copyin_control(struct mbuf **mp, caddr_t buf, u_int buflen)
{
    // ...<snip>...
    while (idx < buflen) {
        error = copyin(buf + idx, &msglen, sizeof(msglen)); // (1) Length of each CMSG read
                                                            // from input buffer
        if (msglen < sizeof(struct cmsghdr))
            return (EINVAL);
        msglen = FREEBSD32_ALIGN(msglen);
        if (idx + msglen > buflen)
            return (EINVAL);
        // <snip> advance to next CMSG
    }
    if (len > MCLBYTES)
        return (EINVAL);

    m = m_get(M_WAITOK, MT_CONTROL); // (2) Memory allocation done with total length
    if (len > MLEN)
        MCLGET(m, M_WAITOK);
    m->m_len = len;

    md = mtod(m, void *);
    while (buflen > 0) {
        error = copyin(buf, md, sizeof(struct cmsghdr)); // (3) Length of each CMSG read a
                                                         // second time from input buffer
        if (error)
            break;
        msglen = *(u_int *)md;
        msglen = FREEBSD32_ALIGN(msglen);

        // <snip> Modify the message length to account for alignment.
        msglen -= FREEBSD32_ALIGN(sizeof(struct cmsghdr));
        if (msglen > 0) {
            error = copyin(buf, md, msglen); // (4) Second length used to copy into
                                             // buffer, possible overflow
            // <snip> advance to next CMSG
        }
    }

    // ...<snip>...
    return (error);
```

CVE-2021-34514 ALPC TOCTOU LPE

When an application creates an ALPC port, it has the option to specify a shared memory region to use for received messages as a performance enhancement. When a message is sent to the ALPC port, the kernel will copy the message and associated metadata into this shared region, known as a completion list. This is handled by the *AlpcpCompleteDispatchMessage* function.

FIGURE 17: CVE-2021-34514 *ALPCPCOMPLETEDISPATCHMESSAGE*

```
//
// Heavily simplified version of vulnerable function
void AlpcpCompleteDispatchMessage( _ALPC_DISPATCH_CONTEXT *DispatchContext )
{
    _ALPC_PORT *port;
    _KALPC_MESSAGE *message;
    _ALPC_COMPLETION_LIST *completionList;
    _ALPC_MESSAGE_ATTRIBUTES* attributes;
    _PORT_MESSAGE *userMappedMessage;
    void *userMappedMessageData;
    uint32_t completionBufferOffset;
    uint32_t bufferLength;
    uint32_t alignmentPadding = 0;

    port = DispatchContext->TargetPort;
    message = DispatchContext->Message;
    completionList = port->CompletionList;
    bufferLength = message->PortMessage.u1.s1.TotalLength;
    bufferLength += completionList->AttributeSize + alignmentPadding;

    // Finds free space in the completion list using the bitmap
    completionBufferOffset = AlpcpAllocateCompletionBuffer( port, bufferLength );

    userMappedMessage = (_PORT_MESSAGE *)((uintptr_t) completionList->Data +
                                          completionBufferOffset);

    // Message header is copied into shared user memory (** #1 **)
    *userMappedMessage = message->PortMessage;
    userMappedMessageData = userMappedMessage + 0x1;

    // Copy message body into shared user memory
    if (message->DataUserVa == (void *)0x0)
    {
        AlpcpReadMessageData(message, userMappedMessageData);
    }
    else
    {
        AlpcpGetDataFromUserVaSafe(message, userMappedMessageData);
    }

    if (completionList->AttributeFlags != 0x0)
    {
        // TotalLength (the message length) is read from shared user memory. (** #2 **)
        // *** If changed by a malicious application between #1 & #2 then
        // attributes can point past end of buffer ***
        attributes = (_ALPC_MESSAGE_ATTRIBUTES *)( (uintptr_t) userMappedMessage +
                                                   userMappedMessage->u1.s1.TotalLength +
                                                   alignmentPadding );

        attributes->AllocatedAttributes = completionList->AttributeFlags;
        attributes->ValidAttributes = 0;
        AlpcpExposeAttributes(Port, 0, message, completionList->AttributeFlags,
            attributes );
    }
```

This function allocates space in the completion list using the bitmap section, then copies the message header and message body into the shared user mode region. If the message contains attributes, then they will also be copied. Attributes are normally stored immediately after the message data. However, when calculating this position, the message length (*TotalLength*) is read from the message header in the shared user mode region.

If a malicious application changes the message length in the shared region after the kernel has copied the message header (location #1 in Figure 17), but before the attribute destination pointer is calculated (location #2 in Figure 17), then a write past the end of the buffer will occur. This can lead to either memory corruption or an access violation and Blue Screen of Death (BSOD).

FIGURE 18: CALL STACK FOR CVE-2021-34514 BSOD

```
nt!AlpcpCompleteDispatchMessage+0x534:
fffff805`04acdf94 448901          mov     dword ptr [rcx],r8d
    ds:ffffe781`61c5efff=????????

 # RetAddr           Call Site
00 fffff805`04acd975 nt!AlpcpCompleteDispatchMessage+0x534
01 fffff805`04acd286 nt!AlpcpDispatchNewMessage+0x255
02 fffff805`04aca75a nt!AlpcpSendMessage+0x4b6
03 fffff805`04837975 nt!NtAlpcSendWaitReceivePort+0x21a
04 00007ffd`eb0e4424 nt!KiSystemServiceCopyEnd+0x25
```

While investigating methods to exploit this vulnerability, I explored many different areas of the Windows kernel, looking for ways to control the kernel address space. One of these areas, which turned out to be more interesting, was Windows Sockets.

Socket IOCTL Validation Bypass

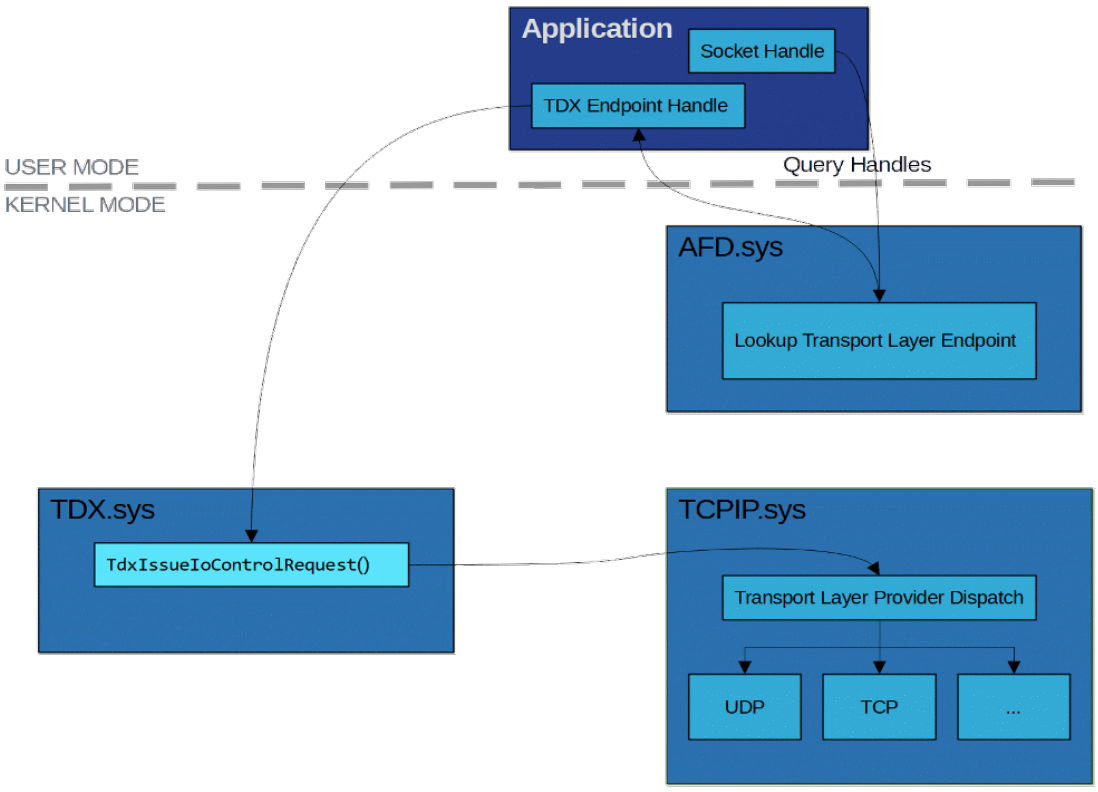
When a socket is created with a transport name specified, a TDX handle will be opened to an address endpoint when the socket is bound to an address. If the correct flags are set in the socket create packet, the address handle can be returned to user mode with the *AfdQueryHandles* IOCTL.

Using this handle, internal socket IO control requests can be issued directly using the function *TdxIssueIoControlRequest* with no restrictions on the types of codes passed in. This function is likely not intended to be reached from user mode.

The validation bypass was patched by copying the validation done in AFD (in the function *AfdAllowedUserIOControlRequest*) to the TDX driver (via a new *TdxTdiAllowedUserIOControlRequest* function).

Three different memory corruption vulnerabilities and an information disclosure were discovered using this validation bypass.

FIGURE 19: AFD IOCTL VALIDATION BYPASS



CVE-2021-38629 Socket Query Security InfoLeak

Socket options calls sent to a TCP address endpoint are received in *TCPIP.sys* by the function *TcpTlEndpointIoControlEndpointCalloutRoutine*, which calls handling routines specific to the type of IOCTL (i.e. *setsockopt*, *ioctlsocket*, internal). If the socket IOCTL validation bypass is used with the internal type, then the request is handled by *TcpIoControlEndpoint*, and a code of 0x20 invokes the *TcpQuerySecurityEndpoint* function. This function reads the security descriptor for the address endpoint and returns a pointer to it in the output buffer.

The security descriptor is allocated from the PagedPool and the leak of this pointer is an ASLR bypass. This information leak can make exploitation of memory corruption bugs easier.

Note that this information disclosure also applies to UDP address endpoints.

FIGURE 20: CALL STACK FOR CVE-2021-38629

```
 # RetAddr           Call Site
00 fffff806`5a595351 tcpip!TcpQuerySecurityEndpoint
01 fffff806`5a4f1da4 tcpip!TcpIoControlEndpoint+0xa2e15
02 fffff806`5808a288 tcpip!TcpTlEndpointIoControlEndpointCalloutRoutine+0x74
03 fffff806`5808a1fd nt!KeExpandKernelStackAndCalloutInternal+0x78
04 fffff806`5a4c4667 nt!KeExpandKernelStackAndCalloutEx+0x1d
05 fffff806`5b4d0bfc tcpip!TcpTlEndpointIoControlEndpoint+0x77
06 fffff806`5b4d372e tdx!TdxIssueIoControlRequest+0x358
07 fffff806`58041cb5 tdx!TdxTdiDispatchDeviceControl+0x37e
08 fffff806`584e1f44 nt!IofCallDriver+0x55
09 fffff806`584e2f0b nt!IopSynchronousServiceTail+0x1d4
0a fffff806`584e22b6 nt!IopXxxControlFile+0xc3b
0b fffff806`58240975 nt!NtDeviceIoControlFile+0x56
```

FIGURE 21: INFORMATION DISCLOSURE EXAMPLE

```
[.] Triggering Query Security Information Disclosure
  [.] Opened '\Device\Afd\Endpoint'
  [.] Socket bind() success
  [.] Queried TDX Handle: 0x009C
  [+] Leaked kernel PagedPool pointer: 0xffff818706ec3b20
[+] Finished with status: SUCCESS
```

CVE-2021-38638 #1
Socket Set Security LPE

Socket options calls sent to a TCP address endpoint are received in TCPIP.sys by the function *TcpTlEndpointIoControlEndpointCalloutRoutine*, which calls handling routines specific to the type of IOCTL (i.e. *setsockopt*, *ioctlsocket*, internal). If the socket IOCTL validation bypass is used with the internal type, then the request is handled by *TcpIoControlEndpoint*, and a code of 0x18 invokes the *TcpSetSecurityEndpoint* function. This function replaces the address endpoint's security descriptor with an arbitrary pointer specified by user mode. A reference count on the security descriptor is incremented with a call to *ObReferenceSecurityDescriptor*, this results in an increment of an arbitrary kernel address that can be leveraged into full read and write access to kernel memory.

Note that this vulnerability also applies to UDP address endpoints.

FIGURE 22: CRASHING CALL STACK FOR CVE-2021-38638 #1

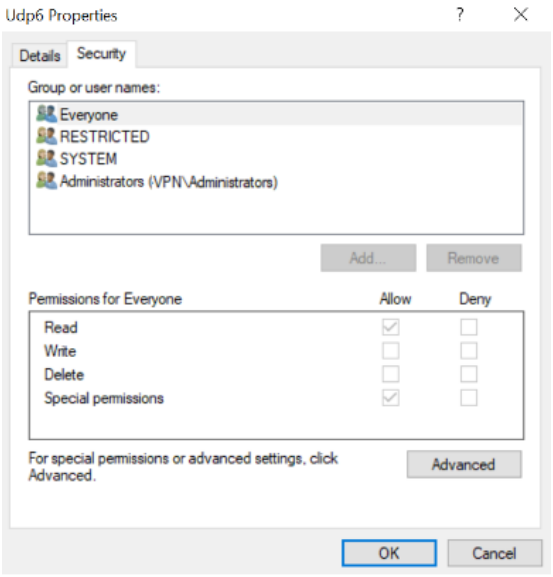
```
nt!ObReferenceSecurityDescriptor+0x2:
fffff803`151bc682 f0480fc141e8    lock xadd qword ptr [rcx-18h],rax
    ds:ffff8888`cccc442c=????????????????

 # RetAddr           Call Site
00 fffff803`171a639c nt!ObReferenceSecurityDescriptor+0x2
01 fffff803`171a62f1 tcpip!TcpSetSecurityEndpointWorkQueueRoutine+0x6c
02 fffff803`1716536c tcpip!TcpSetSecurityEndpoint+0x81
03 fffff803`170c1da4 tcpip!TcpIoControlEndpoint+0xa2e30
04 fffff803`14c7d288 tcpip!TcpTlEndpointIoControlEndpointCalloutRoutine+0x74
05 fffff803`14c7d1fd nt!KeExpandKernelStackAndCalloutInternal+0x78
06 fffff803`17094667 nt!KeExpandKernelStackAndCalloutEx+0x1d
07 fffff803`18060bfc tcpip!TcpTlEndpointIoControlEndpoint+0x77
08 fffff803`1806372e tdx!TdxIssueIoControlRequest+0x358
09 fffff803`14c34cb5 tdx!TdxTdiDispatchDeviceControl+0x37e
0a fffff803`150d4f44 nt!IofCallDriver+0x55
0b fffff803`150d5f0b nt!IopSynchronousServiceTail+0x1d4
0c fffff803`150d52b6 nt!IopXxxControlFile+0xc3b
0d fffff803`14e33975 nt!NtDeviceIoControlFile+0x56
```

Impact

This vulnerability can be exploited to gain full read and write access to kernel memory, allowing complete control over the system (see later section for details). The vulnerable code is reachable from any process that can open a handle to the TCP or UDP transport device (*'\Device\Tcp'*). These devices are accessible to the *'Everyone'* group:

FIGURE 23: PERMISSIONS FOR '\DEVICE\UDP6'



However, most sandboxes or low integrity processes will not be able to access the required device. This also applies to the QoS bugs CVE-2021-38638 #2 and CVE-2021-38638 #3.

Root Cause Analysis

A normal *ioctlsocket* call is processed by *AfdTliIoControl* and this function has a check to ensure user mode is only allowed to call a subset of IOCTL socket codes. This check is done in the *AfdAllowedUserIOControlRequest* function which specifically prevents the IOCTL type required for *TcpIoControlEndpoint* to be invoked.

It is likely that the possibility of sending socket IOCTL messages through the TDX driver — thereby bypassing the validation and copying done in the AFD driver — was not considered. Therefore, the lower layer TCP/IP driver assumed it would always be called from another kernel-mode driver which is not true. This also applies to the information leak.

CVE-2021-38638 #2
Socket Associate QoS LPE

Socket options calls sent to a UDP address endpoint are received in *PACER.sys* by the function *UdpTlEndpointIoControlEndpointCalloutRoutine*, which calls handling routines specific to the type of IOCTL (i.e. *setsockopt*, *ioctlsocket*, internal). If the *ioctlsocket* or *setsockopt* types are used, then the request is handled by *UdpSetSockOptEndpoint*. Using the socket IOCTL validation bypass and a code of *SIO_RESERVED_1* (0x8800001A) will cause the request to be handled by *QimInspectAssociateQoS* but without the message translation that normally happens in *AFD.sys*.

Quality of Service (QoS) functionality is handled by PACER.sys, and the Associate QoS message is eventually passed down to the *PcpValidateAndReferenceFlow* function, which reads an arbitrary object pointer out of the input data. A reference count is incremented on this object in *PcpReferenceFlow*, this results in an increment of an arbitrary kernel address that can be leveraged into full read and write access to kernel memory.

Note that this vulnerability also applies to TCP address endpoints. See the later section on how this vulnerability can be exploited.

FIGURE 24: CRASHING CALL STACK FOR CVE-2021-38638 #2

```
pacer!PcpReferenceFlow:
fffff802`16876384 8b8168020000    mov     eax,dword ptr [rcx+268h]
    ds:002b:f33dfade`f00dfd36=????????????????

 # RetAddr           Call Site
00 fffff802`1687a04e pacer!PcpReferenceFlow
01 fffff802`1687b0e0 pacer!PcpValidateAndReferenceFlow+0x12
02 fffff802`157fbd5d pacer!PcQoSPValidateAndReferenceFlow+0x20
03 fffff802`157f4398 tcpip!EQoSValidateAndReferenceQoSFlow+0x4d
04 fffff802`1578c97c tcpip!QimInspectAssociateQoS+0x40
05 fffff802`156eb6d2 tcpip!UdpSetSockOptEndpoint+0xa1238
06 fffff802`166d0bfc tcpip!UdpTlProviderIoControlEndpoint+0x52
07 fffff802`166d372e tdx!TdxIssueIoControlRequest+0x358
08 fffff802`1323bcb5 tdx!TdxTdiDispatchDeviceControl+0x37e
09 fffff802`136dbf44 nt!IofCallDriver+0x55
0a fffff802`136dcf0b nt!IopSynchronousServiceTail+0x1d4
0b fffff802`136dc2b6 nt!IopXxxControlFile+0xc3b
0c fffff802`1343a975 nt!NtDeviceIoControlFile+0x56
```

Root Cause Analysis

A normal ioctlsocket call is processed by AfdTliIoControl and this function has a special case to handle an Associate QoS request, specifically by calling the AfdTliIoControlHandleAssociateQoS function. This routine validates and makes a copy of the data passed in from user mode, including opening a handle and saving a FILE_OBJECT pointer.

It is likely that the possibility of sending socket IOCTL messages through the TDX driver — thereby bypassing the validation and copying done in the AFD driver — was not considered. Therefore, the lower layer Pacer driver assumed it always receives validated data which is not true. This also applies to the Set QoS LPE vulnerability.

CVE-2021-38638 #3 Socket Set QoS LPE

Socket options calls sent to a UDP address endpoint are received in PACER.sys by the function *UdpTlEndpointIoControlEndpointCalloutRoutine*, which calls handling routines specific to the type of IOCTL (i.e. *setsockopt*, *ioctlsocket*, internal). If the *ioctlsocket* or *setsockopt* types are used, then the request is handled by *UdpSetSockOptEndpoint*. Using the socket IOCTL validation bypass and a code of *SIO_SET_QOS* (0x8800000B) will cause the request to be handled by *QimInspectSetQoS* but without the message translation that normally happens in *AFD.sys*.

The structure expected as input for the IOCTL is:

FIGURE 25: QOS IOCTL STRUCTURE

```
typedef struct _QualityOfService
{
    FLOWSPEC  SendingFlowspec;   /* the flow spec for data sending */
    FLOWSPEC  ReceivingFlowspec; /* the flow spec for data receiving */
    WSABUF    ProviderSpecific;  /* additional provider specific stuff */
} QOS, FAR * LPQOS;
```

Quality of Service functionality is handled by *PACER.sys*, and the Associate QoS message is eventually passed down to the *PcpValidateFlowParameters* function which accesses the *ProviderSpecific* buffer without properly probing and locking this user mode buffer. This can result in an access violation due to an invalid memory access.

More interestingly, the buffer containing the QOS structure is in memory that has been mapped from user mode so it can be changed while being accessed. This results in a TOCTOU bug (see section on TOCTOU) where the length of the *ProviderSpecific* buffer is read to calculate an allocation size and then read again to copy into the buffer. If user mode increases the length after the allocation is made, then a buffer overflow will occur which can be leveraged into full read and write access to kernel memory.

The pseudocode for the TOCTOU bug:

FIGURE 26: TOCTOU PSEUDOCODE

```
// In PcpCreateFlow()
status =  PcpUpdateFlow( flow,
                         var1,
                         bufferLength, // #1 Read from qos->ProviderSpecific.len earlier
                                       // in call stack, in PcQoSPCreateFlow()
                         type,         // 0
                         qos,          // Points to buffer shared with user mode
                         ... );
// ...


NTSTATUS
    PcpUpdateFlow
    (
        void*  Flow,
        void*  Param_2,
        int    BufferLength,
        int    Type,
        QOS*   Qos,
        ...
    )
{
    // ...

    savedQos = ExAllocatePool( NonPagedPoolNx, BufferLength, 'lfcP' );

    // ...
    memcpy( savedQos, Qos, sizeof( QOS ) );
    memcpy( savedQos + sizeof( QOS ),
            Qos->ProviderSpecific.buf,
            Qos->ProviderSpecific.len ); // #2 If this length increases between when it
                                         // was first read in PcQoSPCreateFlow() then a
```

The above code shows the original buffer length read from shared user memory being passed to *PcpUpdateFlow* (#1). The length used in the memory copy (#2) is then read again from shared user memory, thus potentially causing a heap overflow.

FIGURE 27: CRASHING CALL STACK FOR CVE-2021-38638 #3

```
pacer!PcpValidateFlowParameters+0x176:
fffff805`73b8a2ee 418b4e04        mov     ecx,dword ptr [r14+4]
    ds:002b:f00d1c0f`fee1f011=????????????????

 # RetAddr           Call Site
00 fffff805`73b899a7 pacer!PcpValidateFlowParameters+0x176
01 fffff805`73b88438 pacer!PcpUpdateFlow+0x2ff
02 fffff805`73b8abae pacer!PcpCreateFlow+0xf8
03 fffff805`724eb7c3 pacer!PcQoSPCreateFlow+0x8e
04 fffff805`724e453f tcpip!EQoSCreateQoSFlow+0x7f
05 fffff805`7247c975 tcpip!QimInspectSetQoS+0x7b
06 fffff805`723db6d2 tcpip!UdpSetSockOptEndpoint+0xa1231
07 fffff805`73400bfc tcpip!UdpTlProviderIoControlEndpoint+0x52
08 fffff805`7340372e tdx!TdxIssueIoControlRequest+0x358
09 fffff805`6fe45cb5 tdx!TdxTdiDispatchDeviceControl+0x37e
0a fffff805`702e5f44 nt!IofCallDriver+0x55
0b fffff805`702e6f0b nt!IopSynchronousServiceTail+0x1d4
0c fffff805`702e62b6 nt!IopXxxControlFile+0xc3b
0d fffff805`70044975 nt!NtDeviceIoControlFile+0x56
```

Note that this vulnerability also applies to TCP address endpoints.
See an upcoming section on how this vulnerability can be exploited.

CVE-2021-38628 Set Multicast Filter Local Privilege Elevation

Windows provides the option of setting multicast filtering parameters for IPv4 or IPv6 addresses. These parameters can be set using the *ioctlsocket* call with the *SIOCSMSFILTER* code. This request uses the following data structure:

FIGURE 28: SET MULTICAST FILTER IOCTL STRUCTURE

```
//
// Structure for GROUP_FILTER used by protocol independent source filters
// (SIOCSMSFILTER and SIOCGMSFILTER).
//
typedef struct group_filter {
    ULONG gf_interface;         // Interface index.
    SOCKADDR_STORAGE gf_group;  // Multicast address.
    MULTICAST_MODE_TYPE gf_fmode; // Filter mode.
    ULONG gf_numsrc;            // Number of sources.
    SOCKADDR_STORAGE gf_slist[1]; // Source address.
} GROUP_FILTER, *PGROUP_FILTER;
```

When this structure is parsed by the *TCPIP.sys* driver, it has been mapped into memory shared with user mode. Length validation is done on the *gf_numsrc* field of this structure, which indicates the number of included IP addresses. This length is then used to allocate a *NonPagedPool* buffer to hold the IP addresses. However, when the addresses are copied into the new buffer, *gf_numsrc* is read from the original buffer a second time, leading to a TOCTOU vulnerability (see the TOCTOU section).

If user mode increases the length after the allocation is made, then a buffer overflow will occur which can be leveraged into full read and write access to kernel memory.

FIGURE 29: CALL STACK TO READ THE SECOND LENGTH FOR CVE-2021-38628

```
tcpip!IpNlpSetSessionInfo+0xa238a:
fffff806`42b8e2ea 4c8b75a0        mov     r14,qword ptr [rbp-60h]

 # RetAddr           Call Site
00 fffff806`42aebb96 tcpip!IpNlpSetSessionInfo+0xa238a
01 fffff806`42aeb6d2 tcpip!UdpSetSockOptEndpoint+0x452
02 fffff806`43ad5320 tcpip!UdpTlProviderIoControlEndpoint+0x52
03 fffff806`43af3e99 afd!AfdTLIoControl+0x5c
04 fffff806`43ae363d afd!AfdTliIoControl+0x1b7b9
05 fffff806`40641cb5 afd!AfdDispatchDeviceControl+0x7d
06 fffff806`40ae1f44 nt!IofCallDriver+0x55
07 fffff806`40ae2f0b nt!IopSynchronousServiceTail+0x1d4
08 fffff806`40ae22b6 nt!IopXxxControlFile+0xc3b
09 fffff806`40840975 nt!NtDeviceIoControlFile+0x56
```

Impact

This vulnerability can be exploited to gain full read and write access to kernel memory, allowing complete control over the system (see later section for details). The vulnerable code is reachable from any process that creates a UDP socket, which is any process that can open the AFD device (*'/Device/Afd'*). Most sandboxes do not allow socket creation, but the Windows Defender Application Guard Sandbox does.

Root Cause Analysis

The code processing the multicast filter did not guard against TOCTOU race conditions. There are other TOCTOU race conditions in nearby code that lead to BSOD only, but Microsoft does not consider DoS (Denial of Service) bugs to meet their 'bar for servicing' so they may not get patched. Seeing as the *TCPIP.sys* driver is not directly callable by user mode, it is not surprising that these bugs slipped through. Using *Buffered* IO for the AFD IOCTL or making a copy of the input buffer before sending it to the *TCPIP.sys* driver would prevent bugs like this.

Exploitation of Vulnerabilities

This section will discuss how these memory corruption vulnerabilities can be turned into reliable local privilege escalation (LPE) exploits. The four socket-related CVE's have similarities and will be covered together: CVE-2021-38638 #1 and CVE-2021-38638 #2 both allow for the increment of an arbitrary kernel address, while CVE-2021-38638 #3 and CVE-2021-38628 are both buffer overflows in the NonPaged pool. With an understanding of some Windows kernel internals, reliably exploiting kernel pool corruption is relatively easy.

The techniques used work on Windows 10 19H1 and later and were partially based on *Scooping the Windows 10 Pool* by Paul Fariello and Corentin Bayet of Synaktiv. The latest versions of the Windows kernel use the Segment Heap as the allocator for both the Paged and *NonPaged* pools. The Segment Heap has several backends that are used depending on allocation size and alignment; the Variable Size backend will be targeted with these exploits.

FIGURE 30: VARIABLE SIZE BACKEND STRUCTURES

```
typedef struct {
    uint64_t                    Lock;
    _RTLP_HP_LOCK_TYPE          LockType;
    _RTL_RB_TREE                FreeChunkTree;
    _LIST_ENTRY                 SubsegmentList;
    uint64_t                    TotalCommittedUnits;
    uint64_t                    FreeCommittedUnits;
    _HEAP_VS_DELAY_FREE_CONTEXT DelayFreeContext;
    void*                       BackendCtx;
    _HEAP_SUBALLOCATOR_CALLBACKS Callbacks;
    _RTL_HP_VS_CONFIG           Config;
    uint32_t                    Flags;
} HEAP_VS_CONTEXT;
```

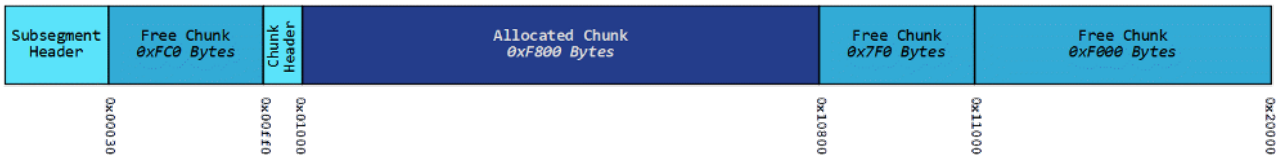
```
typedef struct {
    _LIST_ENTRY ListEntry;
    uint64_t    CommitBitmap;
    uint64_t    CommitLock;
    uint16_t    Size;
    uint16_t    Signature;
} _HEAP_VS_SUBSEGMENT;
```

Variable Size Backend Basics

The Segment Heap's Variable Size (VS) backend is used for allocations greater than 512 bytes and up to 128kB. The VS backend contains a linked list of Subsegments which are split into variable-sized memory chunks. When an allocation request is processed that cannot be satisfied with existing chunks a new Subsegment is created. The Subsegment will be twice the size of the allocation request, rounded up to the closest power of 2 (i.e. an allocation request for 0xf800 bytes would create a Subsegment of size 0x20000). The first page of the Subsegment contains a header (*HEAP_VS_SUBSEGMENT*) and free chunks will be made from any unused memory and stored in the VS *FreeChunkTree*.

The following figure shows a VS Subsegment after an initial allocation of 0xf800 bytes:

FIGURE 31: VS SUBSEGMENT LAYOUT



When memory is freed from a Subsegment it will be merged with any neighbouring free chunks.

Pipe Grooming

As discussed in the *Scooping the Windows 10 Pool paper*, Windows pipes are a useful way to get controlled allocations in the kernel. For grooming the *PagedPool* pipe, attributes can be used. For grooming the *NonPagedPool* pipe, data queue entries can be used.

FIGURE 32: PIPE STRUCTURES

```
typedef struct
{
    LIST_ENTRY  List;
    char*       Name;
    uint64_t    Length;
    char*       Value;
    char        Data[0];
} AttributeEntry;
```

```
typedef struct
{
    LIST_ENTRY  List;
    void*       Irp; // only used for type 1
    void*       SecurityContext;
    uint32_t    Type; // 0 == Buffer, 1 == IRP
    uint32_t    Quota;
    uint32_t    Length;
    uint32_t    Align;
    char        Data[0]; // only used for type 0
} DataQueueEntry;
```

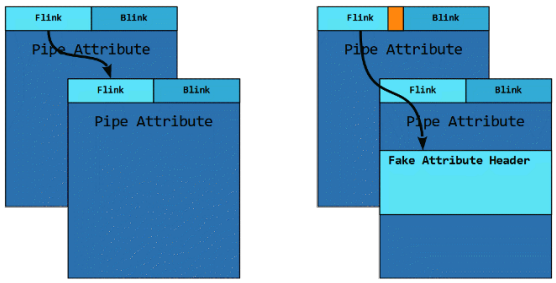
Pipe attributes are added using *NtFsControlFile* with an undocumented IOCTL code. Attributes with a different name are stored as a new entry in the linked list. Data queue entries are created by writing to the pipe and will remain in a linked list until read from the other end of the pipe.

A common feature with both of these structures is that they begin with a doubly linked list entry. If the attacker has full control over the contents of the overflowing data, then the full entry can be controlled directly. If not, then the bottom few bytes of the *Flink* pointer (next entry in list) can be corrupted and the next list entry can point into the body of the following entry, resulting in a fully controlled attribute or data queue entry.

The data in an attribute and a data queue entry can be read back in user mode, so full control over either of these header structures results in an arbitrary kernel read.

When a pipe is created, a context structure (referred to as a CCB) is also created. The CCB contains the linked list heads of the send and receive data queues and attributes, as well as a pointer to the associated file object (See the *NpCreateCcb* function).

FIGURE 33: PIPE ATTRIBUTE CORRUPTION

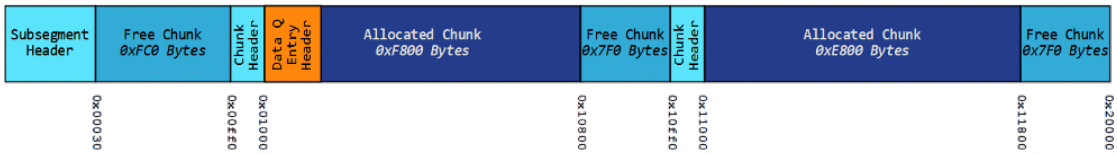


Exploiting NonPagedPool Overflows

Both CVE-2021-38638 #3 and CVE-2021-38628 can result in overflows in the *NonPagedPool* with almost fully controlled data and length. The main difference between these two bugs is that the overflow length for CVE-2021-38628 must be a multiple of the IP address size (4 bytes for IPv4 and 16 bytes for IPv6). This means that it is not possible to corrupt the least significant byte of the *Flink* pointer, so the entire data queue entry header must be overwritten.

Grooming is done by alternating *NonPagedPool* allocations of carefully chosen sizes to obtain the following VS Subsegment layout:

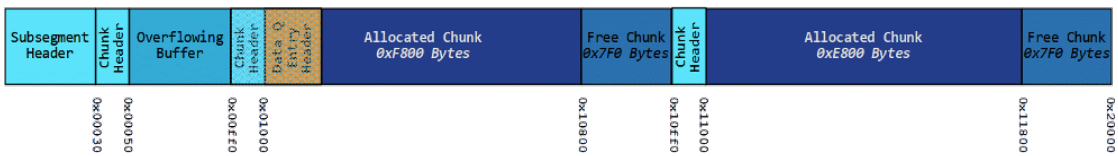
FIGURE 34: NONPAGEDPOOL GROOMING LAYOUT



There will be multiple Subsegments with this layout. The overflow allocation can occur in any Subsegment. Note that the first free chunk is larger (0xfc0 bytes) than the other free chunks in the Subsegment and the grooming ensures that any other free memory blocks of the same size have been used up.

Once the correct heap layout has been obtained, the desired vulnerability can be triggered with an allocation size of 0xf80 and an overflow length of 0x60, which will overwrite the target chunk's heap header and the target data queue entry structure.

FIGURE 34: NONPAGEDPOOL OVERFLOW LAYOUT

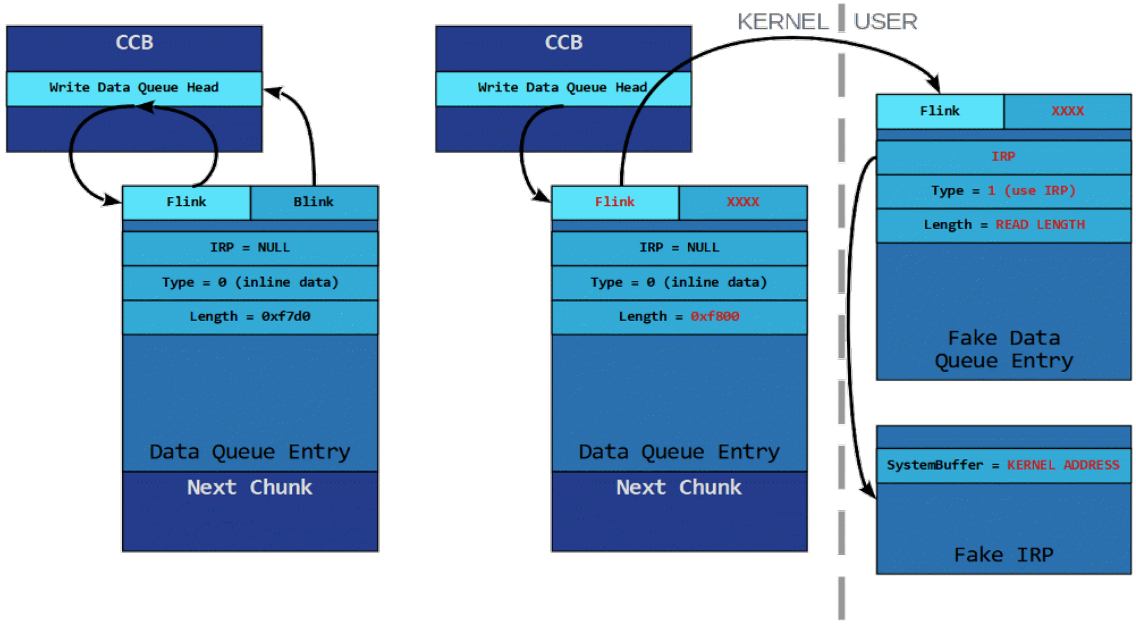


Gaining Arbitrary Read

Once full control of the data queue entry has been obtained, it still only gives a single arbitrary read. In order to get repeated arbitrary reads, the data queue can be extended by pointing the linked list entry to a fake data queue entry in user mode. This allows for repeated reads from arbitrary kernel addresses.

In order to use an arbitrary read, there needs to be some sort of info leak to identify where to start reading. The initial data queue entry corruption can also be used to get an info leak by increasing the entry length so the header of the following memory chunk can be read.

FIGURE 36: DATA QUEUE CORRUPTION TO ARBITRARY READ



At this point, reads from the corrupted pipe will read the first 0xf800 bytes from the corrupted data queue entry which will include the start of the following memory chunk. Any read longer than 0xf800 will read data from the arbitrary address specified in the fake IRP structure in user mode. A normal pipe read will remove the entries from the data queue; instead, it is possible to call *PeekNamedPipe()* which will read from the queue but leave the data entries in place.

As discussed in the *Scooping the Windows 10 Pool* paper, it is possible to get a decrement of an arbitrary address by overwriting the Quota Process Pointer field of an allocated heap chunk header. This could be used to escalate the privileges of a specified process token. However, that would require a second overflow, which would be less reliable, and a full arbitrary kernel write is much more powerful.

With a full arbitrary read and an info leak, the address of the overflown buffer can be calculated. By walking the linked list of data queue entries, the address of the CCB can be obtained as well as the associated file object. This information is enough to set up an arbitrary write.

Gaining Arbitrary Write

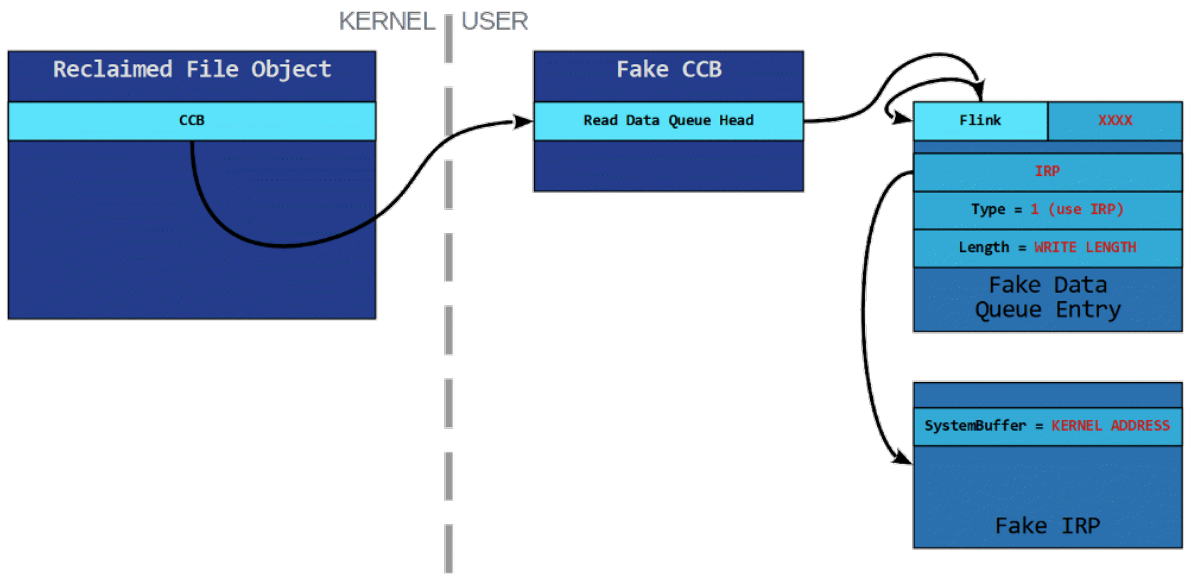
An arbitrary kernel write can be obtained by corrupting data queue entries in a pipe's pending read queue. Entries in the read queue are a small, fixed size as they do not have any associated data and so they are difficult to target with a heap overflow.

Alternatively, the *SecurityContext* field of a data queue entry can be leveraged to free an arbitrary kernel address. During a call of *NpReadDataQueue*, if the *SecurityContext* field is not NULL, then the *SecurityContext* field of the CCB will be freed and replaced with the new context. Using this behaviour, any kernel allocation can be freed – in this case, the file object for the pipe is freed.

By spraying memory with file object-sized allocations, the file object can be reclaimed with fully controlled data. The pointer to the CCB can then be set to a copy of the CCB with the pending read queue pointing to fake entries in user mode. At this point, any data written to the pipe will cause the data to be written to the address specified by the *SystemAddress* field of the fake IRP structure.

Finally, cleanup can be performed to prevent crashes after the exploit finishes. Cleanup includes repairing corrupted heap headers – requiring a leak of the heap cookie – and restoring data queue-linked lists.

FIGURE 37: FAKE FILE OBJECT FOR ARBITRARY WRITE



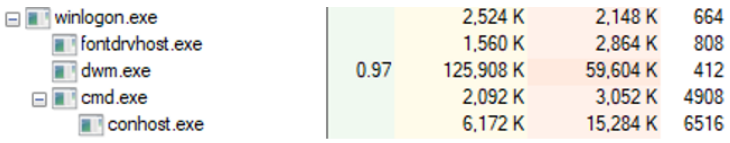
Elevation of Privilege (EoP)

With full arbitrary read and write to kernel memory, an attacker has full control over everything on the system. The simplest elevation method is to alter the *Privileges* field of the process token, giving the process full privileges. This can be abused to access the memory of other processes on the system and/or inject shellcode.

FIGURE 38: SOCKET SET MULTICAST FILTER EXPLOIT OUTPUT

```
[.] Running Socket Set Multicast Filter LPE
  [.] Opened '\Device\Afd\Endpoint'
  [.] Socket created
  [.] Grooming NonPagedPool memory
  [+] Groom complete with status: SUCCESS
  [.] Starting TOCTOU race
  [+] Found corrupted pipe index: 57
  [.] Leaked Flink pointer: 0xffff868f1e44cb58
[+] Arbitrary read setup complete
  [.] CCB address: 0xffff868f1e3aadf0
  [.] File Object address: 0xffff998532f22710
  [.] Replacing File Object
[+] Arbitrary write setup complete
  [.] Process token: 0xffff998532f23390
  [.] Elevating privileges
  [.] Enabled SE_DEBUG_PRIVILEGE
  [.] Found winlogon.exe with PID: 664
  [.] Attached to process: 664
  [.] Starting remote WinExec() thread
[+] Spawned 'cmd.exe' from winlogon
```

FIGURE 39: WINLOGON.EXE AS PARENT OF CMD.EXE

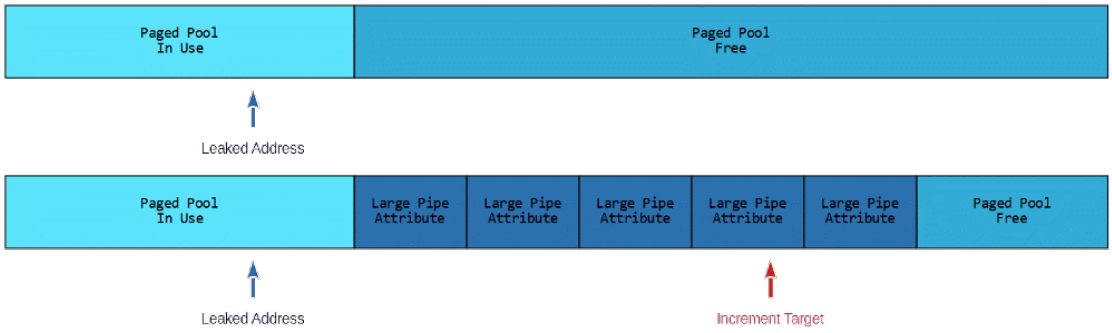


Exploiting Arbitrary PagedPool Increment

The paged pool increment primitives that can be gained with CVE-2021-38638 #1 and CVE-2021-38638 #2 can be exploited with nearly identical methods. Both of these vulnerabilities allow for the increment of a value at an arbitrary address in the kernel. To achieve arbitrary kernel read and write primitives, the pipe attribute entries were targeted.

CVE-2021-38629 can be used to leak an address in the *PagedPool*. This is not enough to predict the location of anything else in the pool by itself, but if the pool is expanded by making many large allocations, then the address of these large allocations can be correctly guessed with high reliability. By taking the available system memory and current usage into account, memory can be groomed to reliably increment the contents of a pipe attribute entry.

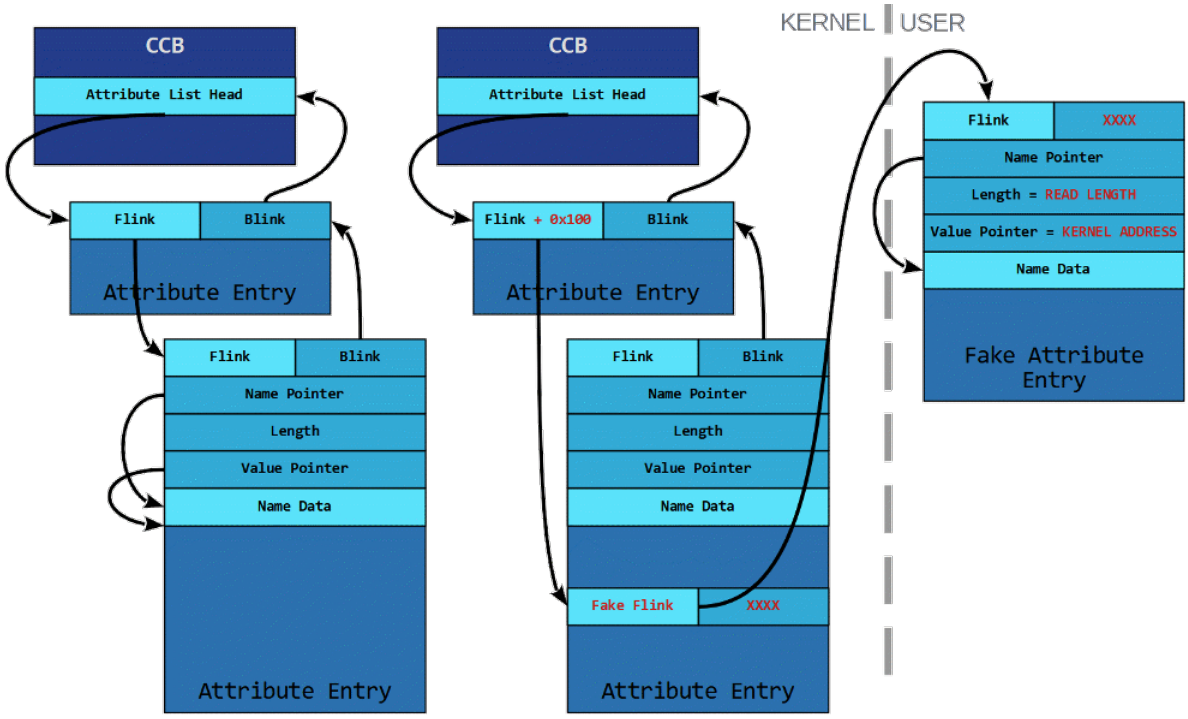
FIGURE 40: ARBITRARY INCREMENT GROOM



Once the data of an attribute entry has been altered with the increment, then the exact address of the attribute entry can be determined by reading back all the attributes and finding the modification. With the address of the attribute entry, the *Flink* pointer can be modified by incrementing the second least significant byte. This will gain full control of the next attribute entry header.

While the pipe attribute entry structures can provide an arbitrary read primitive, there is no way to leverage them for arbitrary writes. The solution is to locate the CCB and the write data queue entry and then use the arbitrary increment to adjust the *Flink* of a data queue entry and take control of the queue. Then an arbitrary write can be gained in the same manner as the *NonPagedPool* overflow.

FIGURE 41: PIPE ATTRIBUTE HEADER INCREMENT TO ARBITRARY READ



Exploiting ALPC Completion List Corruption

Unlike the previously discussed vulnerabilities, CVE-2021-34514 does not result in corruption in the *Paged* or *NonPaged* pool. Instead, the corruption occurs from a buffer allocated in the system address range via *MiReservePtes*. Exploitation of this type of memory corruption has not been published – to the knowledge of the author – and as the vulnerability is reachable from all browser sandboxes, details of exploitation will be left as an exercise for the reader.

Conclusion

A common thread with these vulnerabilities is that they have been present in the Windows kernel since Windows Vista – almost 15 years. Given that they were all found in just over a week-long time period while researching a side project, it is likely that they have been discovered by others and are potentially being exploited in the wild.

By the same reasoning, it is also likely that there are still many other bugs in the Windows kernel that could be exploited. During the course of research, several DoS bugs were found, not all of which were disclosed as Microsoft expressed they were not interested in DoS bugs.

In fact, while writing this paper, another LPE bug (CVE-2021-26442) was found in a separate area of the kernel. More details will follow 30 days after the vulnerability has been patched.

Acknowledgements & References

“Reverse Engineering Windows AFD.sys” – Steven Vittitoe at REcon 2015

https://recon.cx/2015/slides/recon2015-20-steven-vittitoe-Reverse-Engineering-Windows-AFD-sys.pdf

“Scoop the Windows 10 pool!” - Corentin Bayet and Paul Fariello from Synacktiv

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

“Modern Kernel Pool Exploitation: Attacks and Techniques” – Tarjei Mandt at Infiltrate 2011

https://downloads.immunityinc.com/infiltrate-archives/kernelpool_infiltrate2011.pdf

“CVE-2020-7460 FreeBSD TocTou Vulnerability” – m00nbsd c/o ZDI

https://www.zerodayinitiative.com/blog/2020/9/1/cve-2020-7460-freebsd-kernel-privilege-escalation

Disclosure Timeline

2021-04-21	ALPC bug discovered
2021-04-29	Socket bugs discovered
2021-05-04	Sample PoCs and writeups completed and reported to Microsoft
2021-05-10	Working exploits submitted to Microsoft
2021-06-16	CVE-2021-26442 discovered and reported to Microsoft
2021-07-13	Patch released for CVE-2021-34514
2021-09-14	Patches released for CVE-2021-38628, CVE-2021-38629, and CVE-2021-38638
2021-10-12	Patch released for CVE-2021-26442

About Field Effect

Field Effect believes that businesses of all sizes deserve powerful cyber security solutions to protect them. Our threat monitoring, detection, and response platform, along with our training and compliance products and services are the result of years of research and development by the brightest talents in the cyber security industry. Our solutions are purpose-built for SMBs and deliver sophisticated, easy-to-use and manage technology with actionable insights to keep you safe from cyber threats.

Contact information

LetsChat@fieldeffect.com

Canada and the United States
+1 (800) 299-8986

United Kingdom
+44 (0) 800 086 9176

Australia
+61 1800 431418