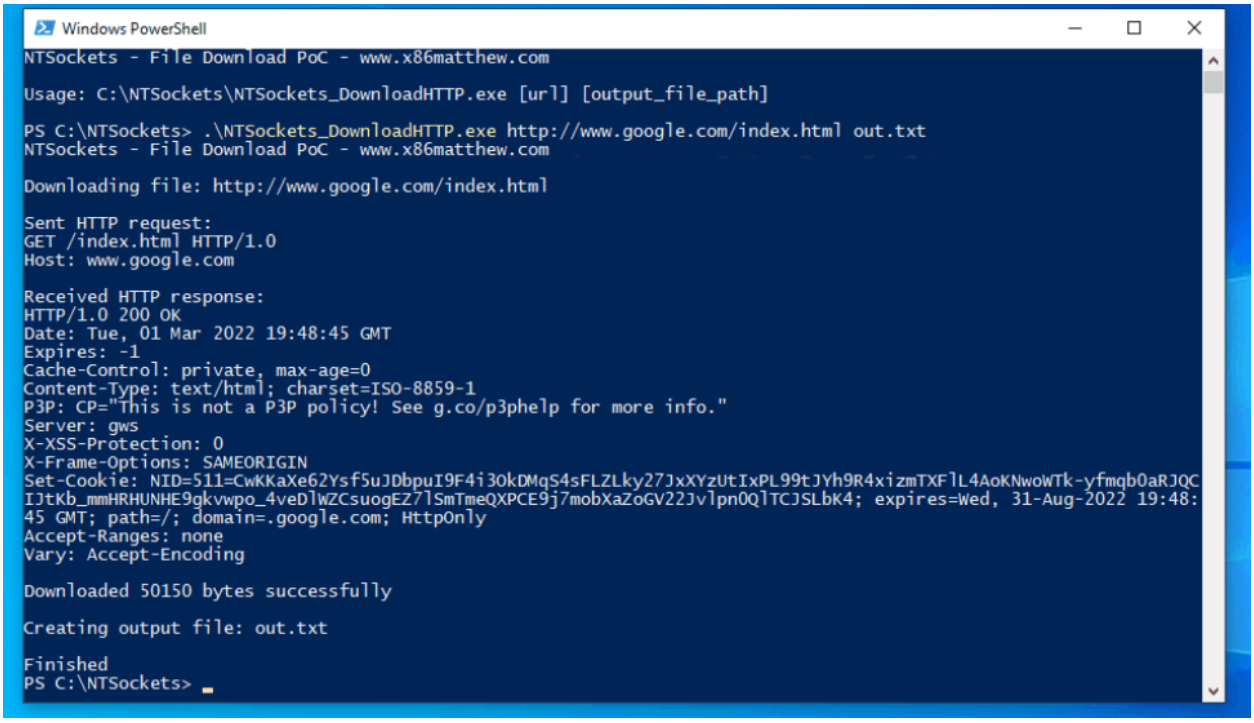
NTSockets - Downloading a file via HTTP using the NtCreateFile and NtDeviceIoControlFile syscalls

Author: x86matthew | **Twitter**: @x86matthew | **E-Mail**: x86matthew@gmail.com

Posted: 01/03/2022

Link: https://www.x86matthew.com/view_post?id=ntsockets

This post demonstrates how to create TCP sockets and transmit/receive data using only ntdll exports.



The functions that we will be using are `NtCreateFile` and `NtDeviceIoControlFile`. The Winsock library uses these functions to communicate with the AFD driver, but we will bypass Winsock and call them directly. These functions could also be called using direct syscall instructions for added stealth against user-mode hooks, although it is still easy to detect network traffic at the kernel level. To demonstrate this concept, I have created a tool that downloads a file via HTTP.

I have only reverse-engineered the internal AFD data structures as much as is necessary for this proof-of-concept. I would expect that further information about the AFD structures can be found elsewhere.

To create a socket, we call `NtCreateFile` to open the `\Device\Afd\Endpoint` object. The socket attributes (address family, protocol type, etc) are specified using an undocumented structure that is passed to `NtCreateFile` as "extended attributes". I have hard-coded these attributes to create an IPv4 TCP socket:

```
DWORD NTSockets_CreateTcpSocket(NTSockets_SocketDataStruct *pSocketData)
{

    IO_STATUS_BLOCK IoStatusBlock;
    HANDLE hEvent = NULL;
    HANDLE hSocket = NULL;
    OBJECT_ATTRIBUTES ObjectAttributes;
    NTSockets_SocketDataStruct SocketData;
    UNICODE_STRING ObjectFilePath;
    DWORD dwStatus = 0;
```

```
BYTE bExtendedAttributes[] =
{
    0x00, 0x00, 0x00, 0x00, 0x00, 0x0F, 0x1E, 0x00, 0x41, 0x66, 0x64, 0x4F,
    0x70, 0x65, 0x6E, 0x50,0x61, 0x63, 0x6B, 0x65, 0x74, 0x58, 0x58, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,0x02, 0x00, 0x00, 0x00,
    0x01, 0x00, 0x00, 0x00, 0x06, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x60, 0xEF, 0x3D, 0x47, 0xFE
};



// create status event
hEvent = CreateEvent(NULL, 0, 0, NULL);

if(hEvent == NULL)
{
    // error
    return 1;
}


// set afd endpoint path
memset((void*)&ObjectFilePath, 0, sizeof(ObjectFilePath));
ObjectFilePath.Buffer = L"\\Device\\Afd\\Endpoint";
ObjectFilePath.Length = wcslen(ObjectFilePath.Buffer) * sizeof(wchar_t);
ObjectFilePath.MaximumLength = ObjectFilePath.Length;


// initialise object attributes
memset((void*)&ObjectAttributes, 0, sizeof(ObjectAttributes));
ObjectAttributes.Length = sizeof(ObjectAttributes);
ObjectAttributes.ObjectName = &ObjectFilePath;
ObjectAttributes.Attributes = 0x40;


// create socket handle
IoStatusBlock.Status = 0;
IoStatusBlock.Information = NULL;
dwStatus = NtCreateFile(&hSocket, 0xC0140000, &ObjectAttributes,
&IoStatusBlock, NULL, 0, FILE_SHARE_READ | FILE_SHARE_WRITE, 1, 0,
 bExtendedAttributes, sizeof(bExtendedAttributes));

if(dwStatus != 0)
{
    // error
    CloseHandle(hEvent);

    return 1;
}


// initialise SocketData object
memset((void*)&SocketData, 0, sizeof(SocketData));
```

```
    SocketData.hSocket = hSocket;
    SocketData.hStatusEvent = hEvent;

    // store socket data
    memcpy((void*)pSocketData, (void*)&SocketData, sizeof(SocketData));

    return 0;
}
```

Now that we have a valid socket handle, we can communicate with the AFD driver using `NtDeviceIoControlFile`. I have created the following generic function that processes AFD driver messages:

```
DWORD NTSockets_SocketDriverMsg(NTSockets_SocketDataStruct *pSocketData, DWORD
dwIoControlCode, BYTE *pData, DWORD dwLength, DWORD *pdwOutputInformation)
{

    IO_STATUS_BLOCK IoStatusBlock;
    DWORD dwStatus = 0;
    BYTE bOutputBlock[0x10];

    // reset status event
    ResetEvent(pSocketData->hStatusEvent);

    // send device control request
    IoStatusBlock.Status = 0;
    IoStatusBlock.Information = NULL;
    dwStatus = NtDeviceIoControlFile(pSocketData->hSocket, pSocketData->hStatusEvent,
    NULL, NULL, &IoStatusBlock, dwIoControlCode, (void*)pData, dwLength,
    bOutputBlock, sizeof(bOutputBlock));

    if(dwStatus == STATUS_PENDING)
    {

        // response pending - wait for event
        if(WaitForSingleObject(pSocketData->hStatusEvent, INFINITE) !=
        WAIT_OBJECT_0)
        {
            // error
            return 1;
        }

        // complete - get final status code
        dwStatus = IoStatusBlock.Status;
    }

    // check for errors
    if(dwStatus != 0)
    {
        // error
        return 1;
    }


    if(pdwOutputInformation != NULL)
    {
```

```
        // store output info
        *pdwOutputInformation = (DWORD)IoStatusBlock.Information;
    }

    return 0;
}
```

We can call `NTSockets_SocketDriverMsg` with the corresponding `dwIoControlCode` value for the operation that we want to perform - connect, send, receive, etc. The event object waits for the function to complete if it returns a pending status code.

When finished, we can close a socket using `CloseHandle` (or `NtClose`):

```
DWORD NTSockets_CloseSocket(NTSockets_SocketDataStruct *pSocketData)
{

    // close handles
    CloseHandle(pSocketData->hSocket);
    CloseHandle(pSocketData->hStatusEvent);

    return 0;
}
```

I have created the following library of functions that perform all of the actions that we need for this proof-of-concept:

`NTSockets_CreateTcpSocket` - Create a TCP socket (equivalent to `socket()`) `NTSockets_ConvertIP` - Converts an IP string to a binary address (equivalent to `inet_addr()`) `NTSockets_Swap16BitByteOrder` - Converts a 16-bit integer to/from network byte order (equivalent to `htons()` / `ntohs()`) `NTSockets_Connect` - Connect to a remote host (equivalent to `connect()`) `NTSockets_Send` - Send data to socket (equivalent to `send()` - note: the function doesn't return until all bytes are sent) `NTSockets_Recv` - Receive requested number of bytes from socket (equivalent to `recv()` - note: the function doesn't return until all bytes are received) `NTSockets_CloseSocket` - Close socket (equivalent to `closesocket()`)

For this proof-of-concept, I have created a very simple HTTP client to download a file from a remote webserver. This doesn't support HTTPS or 301/302 status redirects, etc.

In addition to this, I also created a basic DNS client to perform name lookups - this is because we can't use the `gethostbyname()` function as this is part of the Winsock library. The DNS server is currently hard-coded to `8.8.8.8`, but you could read the default DNS server from the registry if preferred.

Full code below:

```
#include <stdio.h>
#include <windows.h>

struct IO_STATUS_BLOCK
{
    union
    {
        DWORD Status;
        PVOID Pointer;
    };

    DWORD *Information;
};
```

```
struct UNICODE_STRING
{
    USHORT Length;
    USHORT MaximumLength;
    PWSTR Buffer;
};

struct OBJECT_ATTRIBUTES
{
    ULONG Length;
    HANDLE RootDirectory;
    UNICODE_STRING *ObjectName;
    ULONG Attributes;
    PVOID SecurityDescriptor;
    PVOID SecurityQualityOfService;
};


struct NTSockets_ConnectDataStruct
{
    DWORD dwUnknown1;
    DWORD dwUnknown2;
    DWORD dwUnknown3;
    sockaddr_in SockAddr;
};


struct NTSockets_BindDataStruct
{
    DWORD dwUnknown1;
    sockaddr_in SockAddr;
};


struct NTSockets_DataBufferStruct
{
    DWORD dwDataLength;
    BYTE *pData;
};


struct NTSockets_SendRecvDataStruct
{
    NTSockets_DataBufferStruct *pBufferList;
    DWORD dwBufferCount;
    DWORD dwUnknown1;
    DWORD dwUnknown2;
};


struct NTSockets_SocketDataStruct
{
    HANDLE hSocket;
    HANDLE hStatusEvent;
```

```
};


struct DNSClient_HeaderStruct
{
    WORD wTransID;
    WORD wFlags;
    WORD wQuestionCount;
    WORD wAnswerRecordCount;
    WORD wAuthorityRecordCount;
    WORD wAdditionalRecordCount;
};


struct DNSClient_RequestQueryDetailsStruct
{
    WORD wType;
    WORD wClass;
};


struct DNSClient_ResponseAnswerHeaderStruct
{
    WORD wName;
    WORD wType;
    WORD wClass;
    WORD wTTL[2];
    WORD wLength;
};


DWORD (WINAPI *NtDeviceIoControlFile)(HANDLE FileHandle, HANDLE Event, VOID
*ApcRoutine, PVOID ApcContext, IO_STATUS_BLOCK *IoStatusBlock, ULONG IoControlCode,
PVOID InputBuffer, ULONG InputBufferLength, PVOID OutputBuffer, ULONG OutputBufferLength);

DWORD (WINAPI *NtCreateFile)(PHANDLE FileHandle, ACCESS_MASK DesiredAccess,
OBJECT_ATTRIBUTES *ObjectAttributes, IO_STATUS_BLOCK *IoStatusBlock,
LARGE_INTEGER *AllocationSize, ULONG FileAttributes, ULONG ShareAccess,
ULONG CreateDisposition, ULONG CreateOptions, PVOID EaBuffer, ULONG EaLength);


DWORD NTSockets_CreateTcpSocket(NTSockets_SocketDataStruct *pSocketData)
{

    IO_STATUS_BLOCK IoStatusBlock;
    HANDLE hEvent = NULL;
    HANDLE hSocket = NULL;
    OBJECT_ATTRIBUTES ObjectAttributes;
    NTSockets_SocketDataStruct SocketData;
    UNICODE_STRING ObjectFilePath;
    DWORD dwStatus = 0;
    BYTE bExtendedAttributes[] =
    {
        0x00, 0x00, 0x00, 0x00, 0x00, 0x0F, 0x1E, 0x00, 0x41, 0x66, 0x64, 0x4F,
```

```
    0x70, 0x65, 0x6E, 0x50,0x61, 0x63, 0x6B, 0x65, 0x74, 0x58, 0x58, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,0x02, 0x00, 0x00, 0x00,
    0x01, 0x00, 0x00, 0x00, 0x06, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x60, 0xEF, 0x3D, 0x47, 0xFE
};


// create status event
hEvent = CreateEvent(NULL, 0, 0, NULL);
if(hEvent == NULL)
{
    // error
    return 1;
}


// set afd endpoint path
memset((void*)&ObjectFilePath, 0, sizeof(ObjectFilePath));
ObjectFilePath.Buffer = L"\\Device\\Afd\\Endpoint";
ObjectFilePath.Length = wcslen(ObjectFilePath.Buffer) * sizeof(wchar_t);
ObjectFilePath.MaximumLength = ObjectFilePath.Length;


// initialise object attributes
memset((void*)&ObjectAttributes, 0, sizeof(ObjectAttributes));
ObjectAttributes.Length = sizeof(ObjectAttributes);
ObjectAttributes.ObjectName = &ObjectFilePath;
ObjectAttributes.Attributes = 0x40;


// create socket handle
IoStatusBlock.Status = 0;
IoStatusBlock.Information = NULL;
dwStatus = NtCreateFile(&hSocket, 0xC0140000, &ObjectAttributes, &IoStatusBlock,
NULL, 0, FILE_SHARE_READ | FILE_SHARE_WRITE, 1, 0, bExtendedAttributes,
sizeof(bExtendedAttributes));

if(dwStatus != 0)
{
    // error
    CloseHandle(hEvent);
    return 1;
}


// initialise SocketData object
memset((void*)&SocketData, 0, sizeof(SocketData));
SocketData.hSocket = hSocket;
SocketData.hStatusEvent = hEvent;

// store socket data
memcpy((void*)pSocketData, (void*)&SocketData, sizeof(SocketData));

return 0;
```

```
}


DWORD NTSockets_SocketDriverMsg(NTSockets_SocketDataStruct *pSocketData, DWORD
dwIoControlCode, BYTE *pData, DWORD dwLength, DWORD *pdwOutputInformation)
{
    IO_STATUS_BLOCK IoStatusBlock;
    DWORD dwStatus = 0;
    BYTE bOutputBlock[0x10];

    // reset status event
    ResetEvent(pSocketData->hStatusEvent);

    // send device control request
    IoStatusBlock.Status = 0;
    IoStatusBlock.Information = NULL;

    dwStatus = NtDeviceIoControlFile(pSocketData->hSocket, pSocketData->hStatusEvent,
    NULL, NULL, &IoStatusBlock, dwIoControlCode, (void*)pData, dwLength,
    bOutputBlock, sizeof(bOutputBlock));

    if(dwStatus == STATUS_PENDING)
    {
        // response pending - wait for event
        if(WaitForSingleObject(pSocketData->hStatusEvent, INFINITE) != WAIT_OBJECT_0)
        {
            // error
            return 1;
        }

        // complete - get final status code
        dwStatus = IoStatusBlock.Status;
    }

    // check for errors
    if(dwStatus != 0)
    {
        // error
        return 1;
    }

    if(pdwOutputInformation != NULL)
    {
        // store output info
        *pdwOutputInformation = (DWORD)IoStatusBlock.Information;
    }

    return 0;
}


DWORD NTSockets_ConvertIP(char *pIP, DWORD *pdwAddr)
{
```

```
char szCurrOctet[8];
DWORD dwCurrOctetIndex = 0;
DWORD dwCompletedOctetCount = 0;
char *pCurrByte = NULL;
DWORD dwEndOfOctet = 0;
DWORD dwEndOfString = 0;
DWORD dwOctet = 0;
BYTE bOctets[4];
DWORD dwAddr = 0;


// read IP string
memset(szCurrOctet, 0, sizeof(szCurrOctet));
dwCurrOctetIndex = 0;
pCurrByte = pIP;
for(;;)
{
    // process current character
    dwEndOfOctet = 0;
    if(*pCurrByte == '\0')
    {
        // end of string
        dwEndOfOctet = 1;
        dwEndOfString = 1;
    }
    else if(*pCurrByte == '.')
    {
        // end of octet
        dwEndOfOctet = 1;
    }
    else
    {
        // ensure this character is a number
        if(*pCurrByte >= '0' && *pCurrByte <= '9')
        {
            if(dwCurrOctetIndex > 2)
            {
                // invalid ip
                return 1;
            }

            // store current character
            szCurrOctet[dwCurrOctetIndex] = *pCurrByte;
            dwCurrOctetIndex++;
        }
        else
        {
            // invalid ip
            return 1;
        }
    }

    // check if the current octet is complete
    if(dwEndOfOctet != 0)
```

```
    {
        if(dwCurrOctetIndex == 0)
        {
            // invalid ip
            return 1;
        }

        // convert octet string to integer
        dwOctet = atoi(szCurrOctet);
        if(dwOctet > 255)
        {
            // invalid ip
            return 1;
        }

        // already read 4 octets
        if(dwCompletedOctetCount >= 4)
        {
            // invalid ip
            return 1;
        }

        // store current octet
        bOctets[dwCompletedOctetCount] = (BYTE)dwOctet;

        // current octet complete
        dwCompletedOctetCount++;

        if(dwEndOfString != 0)
        {
            // end of string
            break;
        }

        // reset szCurrOctet string
        memset(szCurrOctet, 0, sizeof(szCurrOctet));
        dwCurrOctetIndex = 0;
    }

    // move to the next character
    pCurrByte++;
}


// ensure 4 octets were found
if(dwCompletedOctetCount != 4)
{
    // invalid string
    return 1;
}

// store octets in dword value
memcpy((void*)&dwAddr, bOctets, 4);
```

```
    // store value
    *pdwAddr = dwAddr;

    return 0;
}


WORD NTSockets_Swap16BitByteOrder(WORD wValue)
{
    WORD wNewValue = 0;

    // swap byte order - this assumes we are running on an x86-based chip
    *(BYTE*)((DWORD)&wNewValue + 0) = *(BYTE*)((DWORD)&wValue + 1);
    *(BYTE*)((DWORD)&wNewValue + 1) = *(BYTE*)((DWORD)&wValue + 0);

    return wNewValue;
}


DWORD NTSockets_Connect(NTSockets_SocketDataStruct *pSocketData, char *pIP, WORD
wPort)
{
    NTSockets_BindDataStruct NTSockets_BindData;
    NTSockets_ConnectDataStruct NTSockets_ConnectData;
    WORD wConnectPort = 0;
    DWORD dwConnectAddr = 0;

    // bind to local port
    memset((void*)&NTSockets_BindData, 0, sizeof(NTSockets_BindData));
    NTSockets_BindData.dwUnknown1 = 2;
    NTSockets_BindData.SockAddr.sin_family = AF_INET;
    NTSockets_BindData.SockAddr.sin_addr.s_addr = INADDR_ANY;
    NTSockets_BindData.SockAddr.sin_port = 0;

    if(NTSockets_SocketDriverMsg(pSocketData, 0x00012003, (BYTE*)&NTSockets_BindData,
    sizeof(NTSockets_BindData), NULL) != 0)
    {
        // error
        return 1;
    }

    // read connection ip
    if(NTSockets_ConvertIP(pIP, &dwConnectAddr) != 0)
    {
        // error
        return 1;
    }

    // use network byte order for connection port
    wConnectPort = NTSockets_Swap16BitByteOrder(wPort);

    // connect to remote port
    memset((void*)&NTSockets_ConnectData, 0, sizeof(NTSockets_ConnectData));
    NTSockets_ConnectData.dwUnknown1 = 0;
```

```
    NTSockets_ConnectData.dwUnknown2 = 0;
    NTSockets_ConnectData.dwUnknown3 = 0;
    NTSockets_ConnectData.SockAddr.sin_family = AF_INET;
    NTSockets_ConnectData.SockAddr.sin_addr.s_addr = dwConnectAddr;
    NTSockets_ConnectData.SockAddr.sin_port = wConnectPort;
    if(NTSockets_SocketDriverMsg(pSocketData, 0x00012007,
    (BYTE*)&NTSockets_ConnectData, sizeof(NTSockets_ConnectData), NULL) != 0)
    {
        // error
        return 1;
    }

    return 0;
}


DWORD NTSockets_Send(NTSockets_SocketDataStruct *pSocketData, BYTE *pData,
DWORD dwLength)
{
    NTSockets_SendRecvDataStruct NTSockets_SendRecvData;
    NTSockets_DataBufferStruct NTSockets_DataBuffer;
    DWORD dwBytesSent = 0;
    BYTE *pCurrSendPtr = NULL;
    DWORD dwBytesRemaining = 0;

    // set initial values
    pCurrSendPtr = pData;
    dwBytesRemaining = dwLength;

    // send data
    for(;;)
    {
        if(dwBytesRemaining == 0)
        {
            // finished
            break;
        }

        // set data buffer values
        memset((void*)&NTSockets_DataBuffer, 0, sizeof(NTSockets_DataBuffer));
        NTSockets_DataBuffer.dwDataLength = dwBytesRemaining;
        NTSockets_DataBuffer.pData = pCurrSendPtr;

        // send current block
        memset((void*)&NTSockets_SendRecvData, 0, sizeof(NTSockets_SendRecvData));
        NTSockets_SendRecvData.pBufferList = &NTSockets_DataBuffer;
        NTSockets_SendRecvData.dwBufferCount = 1;
        NTSockets_SendRecvData.dwUnknown1 = 0;
        NTSockets_SendRecvData.dwUnknown2 = 0;

        if(NTSockets_SocketDriverMsg(pSocketData, 0x0001201F,
        (BYTE*)&NTSockets_SendRecvData, sizeof(NTSockets_SendRecvData),
        &dwBytesSent) != 0)
        {
```

```
            // error
            return 1;
        }

        if(dwBytesSent == 0)
        {
            // socket disconnected
            return 1;
        }

        // update values
        pCurrSendPtr += dwBytesSent;
        dwBytesRemaining -= dwBytesSent;
    }

    return 0;
}


DWORD NTSockets_Recv(NTSockets_SocketDataStruct *pSocketData, BYTE *pData,
DWORD dwLength)
{
    NTSockets_SendRecvDataStruct NTSockets_SendRecvData;
    NTSockets_DataBufferStruct NTSockets_DataBuffer;
    DWORD dwBytesReceived = 0;
    BYTE *pCurrRecvPtr = NULL;
    DWORD dwBytesRemaining = 0;

    // set initial values
    pCurrRecvPtr = pData;
    dwBytesRemaining = dwLength;

    // send data
    for(;;)
    {
        if(dwBytesRemaining == 0)
        {
            // finished
            break;
        }

        // set data buffer values
        memset((void*)&NTSockets_DataBuffer, 0, sizeof(NTSockets_DataBuffer));
        NTSockets_DataBuffer.dwDataLength = dwBytesRemaining;
        NTSockets_DataBuffer.pData = pCurrRecvPtr;

        // recv current block
        memset((void*)&NTSockets_SendRecvData, 0, sizeof(NTSockets_SendRecvData));
        NTSockets_SendRecvData.pBufferList = &NTSockets_DataBuffer;
        NTSockets_SendRecvData.dwBufferCount = 1;
        NTSockets_SendRecvData.dwUnknown1 = 0;
        NTSockets_SendRecvData.dwUnknown2 = 0x20;
        if(NTSockets_SocketDriverMsg(pSocketData, 0x00012017,
        (BYTE*)&NTSockets_SendRecvData, sizeof(NTSockets_SendRecvData),
```

```
        &dwBytesReceived) != 0)
        {
            // error
            return 1;
        }

        if(dwBytesReceived == 0)
        {
            // socket disconnected
            return 1;
        }

        // update values
        pCurrRecvPtr += dwBytesReceived;
        dwBytesRemaining -= dwBytesReceived;
    }

    return 0;
}


DWORD NTSockets_CloseSocket(NTSockets_SocketDataStruct *pSocketData)
{
    // close handles
    CloseHandle(pSocketData->hSocket);
    CloseHandle(pSocketData->hStatusEvent);

    return 0;
}


DWORD DNSClient_Query(char *pDNSClient_IP, char *pTargetHost, char *pOutput,
DWORD dwOutputMaxLength)
{
    NTSockets_SocketDataStruct SocketData;
    DNSClient_HeaderStruct DNSClient_RequestHeader;
    DNSClient_RequestQueryDetailsStruct DNSClient_RequestQueryDetails;
    DNSClient_HeaderStruct *pDNSClient_ResponseHeader = NULL;
    DNSClient_ResponseAnswerHeaderStruct *pDNSClient_ResponseAnswerHeader = NULL;
    DWORD dwIpAddrIndex = 0;
    DWORD dwFoundRecord = 0;
    DWORD dwCurrAnswerEntryStartIndex = 0;
    DWORD dwHostLength = 0;
    DWORD dwCurrLabelLength = 0;
    WORD wRequestLength = 0;
    WORD wResponseLength = 0;
    WORD wBlockLength = 0;
    WORD wAnswerCount = 0;
    BYTE bIP[4];
    BYTE bResponseBuffer[4096];
    char szConvertedHost[1024];
    char *pCurrDot = NULL;
    char szIP[32];
```

```
// convert target host name to dns format
memset(szConvertedHost, 0, sizeof(szConvertedHost));
_snprintf(szConvertedHost, sizeof(szConvertedHost) - 1, ".%s", pTargetHost);
dwHostLength = strlen(szConvertedHost) + 1;
for(DWORD i = 0; i < dwHostLength; i++)
{
    // process domain labels
    if(szConvertedHost[i] == '.' || szConvertedHost[i] == '\0')
    {
        // check if a previous separator exists
        if(pCurrDot != NULL)
        {
            // calculate current label length
            dwCurrLabelLength = (DWORD)(&szConvertedHost[i] - pCurrDot);
            dwCurrLabelLength--;
            if(dwCurrLabelLength == 0 || dwCurrLabelLength >= 64)
            {
                return 1;
            }

            // insert label length
            *pCurrDot = (char)dwCurrLabelLength;
        }

        // store current dot position
        pCurrDot = &szConvertedHost[i];
    }
}

// create socket handle
if(NTSockets_CreateTcpSocket(&SocketData) != 0)
{
    // error
    return 1;
}

// connect to DNS server
if(NTSockets_Connect(&SocketData, pDNSClient_IP, 53) != 0)
{
    // error
    NTSockets_CloseSocket(&SocketData);
    return 1;
}

// calculate request length
wRequestLength = sizeof(DNSClient_HeaderStruct) + dwHostLength +
sizeof(DNSClient_RequestQueryDetails);
wBlockLength = NTSockets_Swap16BitByteOrder(wRequestLength);

// set request header details
memset((void*)&DNSClient_RequestHeader, 0, sizeof(DNSClient_RequestHeader));
DNSClient_RequestHeader.wTransID = NTSockets_Swap16BitByteOrder(1);
DNSClient_RequestHeader.wFlags = NTSockets_Swap16BitByteOrder(0x100);
DNSClient_RequestHeader.wQuestionCount = NTSockets_Swap16BitByteOrder(1);
```

```
// type A dns request
memset((void*)&DNSClient_RequestQueryDetails, 0,
sizeof(DNSClient_RequestQueryDetails));
DNSClient_RequestQueryDetails.wType = NTSockets_Swap16BitByteOrder(1);
DNSClient_RequestQueryDetails.wClass = NTSockets_Swap16BitByteOrder(1);

// send request length
if(NTSockets_Send(&SocketData, (BYTE*)&wBlockLength, sizeof(WORD)) != 0)
{
    // error
    NTSockets_CloseSocket(&SocketData);
    return 1;
}

// send request header
if(NTSockets_Send(&SocketData, (BYTE*)&DNSClient_RequestHeader,
sizeof(DNSClient_RequestHeader)) != 0)
{
    // error
    NTSockets_CloseSocket(&SocketData);
    return 1;
}
// send host name

if(NTSockets_Send(&SocketData, (BYTE*)szConvertedHost, dwHostLength) != 0)
{
    // error
    NTSockets_CloseSocket(&SocketData);
    return 1;
}



// send host query details
if(NTSockets_Send(&SocketData, (BYTE*)&DNSClient_RequestQueryDetails,
sizeof(DNSClient_RequestQueryDetails)) != 0)
{
    // error
    NTSockets_CloseSocket(&SocketData);
    return 1;
}

// receive response length
if(NTSockets_Recv(&SocketData, (BYTE*)&wBlockLength, sizeof(WORD)) != 0)
{
    // error
    NTSockets_CloseSocket(&SocketData);
    return 1;
}

// swap byte order
wResponseLength = NTSockets_Swap16BitByteOrder(wBlockLength);
```

```
// validate response length
if(wResponseLength < sizeof(DNSClient_HeaderStruct) || wResponseLength
> sizeof(bResponseBuffer))
{
    // error
    NTSockets_CloseSocket(&SocketData);
    return 1;
}

// receive response data
memset((void*)bResponseBuffer, 0, sizeof(bResponseBuffer));
if(NTSockets_Recv(&SocketData, bResponseBuffer, wResponseLength) != 0)
{
    // error
    NTSockets_CloseSocket(&SocketData);
    return 1;
}

// set response header ptr
pDNSClient_ResponseHeader = (DNSClient_HeaderStruct*)bResponseBuffer;

// check flags (expect response, no error)
if(pDNSClient_ResponseHeader->wFlags != NTSockets_Swap16BitByteOrder(0x8180))
{
    // error
    NTSockets_CloseSocket(&SocketData);
    return 1;
}

// validate question count
if(pDNSClient_ResponseHeader->wQuestionCount != NTSockets_Swap16BitByteOrder(1))
{
    // error
    NTSockets_CloseSocket(&SocketData);
    return 1;
}

// get response answer count
wAnswerCount = NTSockets_Swap16BitByteOrder(pDNSClient_ResponseHeader->
wAnswerRecordCount);

// read DNS response answers
dwCurrAnswerEntryStartIndex = wRequestLength;
for(i = 0; i < (DWORD)wAnswerCount; i++)
{
    // validate start index
    if((dwCurrAnswerEntryStartIndex + sizeof(DNSClient_ResponseAnswerHeaderStruct))
    > (DWORD)wResponseLength)
    {
        // error
        NTSockets_CloseSocket(&SocketData);
        return 1;
    }
```

```
    // get current response answer header ptr
    pDNSClient_ResponseAnswerHeader = (DNSClient_ResponseAnswerHeaderStruct*)
    &bResponseBuffer[dwCurrAnswerEntryStartIndex];

    // check if this is a type A record
    if(pDNSClient_ResponseAnswerHeader->wType == NTSockets_Swap16BitByteOrder(1)
    && pDNSClient_ResponseAnswerHeader->wClass == NTSockets_Swap16BitByteOrder(1))
    {

        // ensure value length is 4 (ipv4 addr)
        if(pDNSClient_ResponseAnswerHeader->wLength != NTSockets_Swap16BitByteOrder(4))
        {
            // error
            NTSockets_CloseSocket(&SocketData);
            return 1;
        }

        // validate ip addr index
        dwIpAddrIndex = dwCurrAnswerEntryStartIndex +
        sizeof(DNSClient_ResponseAnswerHeaderStruct);
        if((dwIpAddrIndex + 4) > (DWORD)wResponseLength)
        {
            // error
            NTSockets_CloseSocket(&SocketData);
            return 1;
        }

        // store IP addr
        memcpy((void*)bIP, (void*)&bResponseBuffer[dwIpAddrIndex], 4);

        // set flag
        dwFoundRecord = 1;
        break;
    }
    else
    {
        // check next entry
        dwCurrAnswerEntryStartIndex += sizeof(DNSClient_ResponseAnswerHeaderStruct);
        dwCurrAnswerEntryStartIndex += NTSockets_Swap16BitByteOrder
        (pDNSClient_ResponseAnswerHeader->wLength);

    }
}

// close socket
NTSockets_CloseSocket(&SocketData);

// ensure a valid record was found
if(dwFoundRecord == 0)
{
    return 1;
}

// generate IP string
```

```
    memset(szIP, 0, sizeof(szIP));
    _snprintf(szIP, sizeof(szIP) - 1, "%u.%u.%u.%u", bIP[0], bIP[1], bIP[2], bIP[3]);

    // store value
    strncpy(pOutput, szIP, dwOutputMaxLength);

    return 0;
}


DWORD DownloadFile(char *pURL, BYTE **pOutput, DWORD *pdwOutputLength)
{
    char szProtocol[16];
    char szHostName[256];
    char szRequestHeader[2048];
    char szResponseHeader[2048];
    char *pStartOfHostName = NULL;
    char *pEndOfHostName = NULL;
    char *pRequestPath = NULL;
    DWORD dwAddr = 0;
    char *pHostNamePort = NULL;
    DWORD dwPort = 0;
    char szResolvedIP[32];
    NTSockets_SocketDataStruct SocketData;
    DWORD dwFoundEndOfResponseHeader = 0;
    char szEndOfResponseHeader[8];
    char szResponseSuccessStatus[32];
    char szContentLengthParamName[16];
    char *pContentLength = NULL;
    char *pEndOfContentLength = NULL;
    DWORD dwOutputLength = 0;
    DWORD dwOutputAllocLength = 0;
    BYTE *pOutputBuffer = NULL;
    BYTE *pNewOutputBuffer = NULL;
    BYTE bCurrByte = 0;

    // ensure url starts with 'http://'
    memset(szProtocol, 0, sizeof(szProtocol));
    strncpy(szProtocol, "http://", sizeof(szProtocol) - 1);
    if(strncmp(pURL, szProtocol, strlen(szProtocol)) != 0)
    {
        // error
        printf("Error: Invalid protocol\n");
        return 1;
    }

    // copy host name
    pStartOfHostName = pURL;
    pStartOfHostName += strlen(szProtocol);
    memset(szHostName, 0, sizeof(szHostName));
    strncpy(szHostName, pStartOfHostName, sizeof(szHostName) - 1);

    // remove request path from host name
    pEndOfHostName = strstr(szHostName, "/");
```

```
if(pEndOfHostName == NULL)
{
    // error
    printf("Error: Invalid URL\n");
    return 1;
}

*pEndOfHostName = '\0';

// check if the host name contains a custom port number
pHostNamePort = strstr(szHostName, ":");
if(pHostNamePort == NULL)
{
    // no port specified - use port 80
    dwPort = 80;
}
else
{
    // terminate string
    *pHostNamePort = '\0';

    // extract port number
    pHostNamePort++;
    dwPort = atoi(pHostNamePort);
    if(dwPort == 0)
    {
        // error
        printf("Error: Invalid URL\n");
        return 1;
    }
}

// get start of request path
pRequestPath = pStartOfHostName;
pRequestPath += strlen(szHostName);

// check if the host name is a valid ipv4 address
memset(szResolvedIP, 0, sizeof(szResolvedIP));
if(NTSockets_ConvertIP(szHostName, &dwAddr) != 0)
{
    // not ipv4 - try to resolve host using DNS
    if(DNSClient_Query("8.8.8.8", szHostName, szResolvedIP,
    sizeof(szResolvedIP) - 1) != 0)
    {
        // error
        printf("Error: Failed to resolve host name\n");
        return 1;
    }
}
else
{
    // copy original ip
    strncpy(szResolvedIP, szHostName, sizeof(szResolvedIP) - 1);
}
```

```
// create socket handle
if(NTSockets_CreateTcpSocket(&SocketData) != 0)
{
    // error
    printf("Error: Failed to create TCP socket\n");
    return 1;
}

// connect to server
if(NTSockets_Connect(&SocketData, szResolvedIP, (WORD)dwPort) != 0)
{
    // error
    printf("Error: Failed to connect to server\n");
    NTSockets_CloseSocket(&SocketData);
    return 1;
}

// send HTTP request
memset(szRequestHeader, 0, sizeof(szRequestHeader));
_snprintf(szRequestHeader, sizeof(szRequestHeader) - 1,
"GET %s HTTP/1.0\r\nHost: %s\r\n\r\n", pRequestPath, szHostName);
if(NTSockets_Send(&SocketData, (BYTE*)szRequestHeader, strlen(szRequestHeader)) != 0)
{
    // error
    printf("Error: Failed to send data to server\n");
    NTSockets_CloseSocket(&SocketData);
    return 1;
}


printf("Sent HTTP request:\n%s", szRequestHeader);

// get response header
memset(szEndOfResponseHeader, 0, sizeof(szEndOfResponseHeader));
strncpy(szEndOfResponseHeader, "\r\n\r\n", sizeof(szEndOfResponseHeader) - 1);
memset(szResponseHeader, 0, sizeof(szResponseHeader));
for(DWORD i = 0; i < sizeof(szResponseHeader) - 1; i++)
{
    // get next byte
    if(NTSockets_Recv(&SocketData, (BYTE*)&szResponseHeader[i], 1) != 0)
    {
        // error
        printf("Error: Failed to read HTTP response header\n");
        NTSockets_CloseSocket(&SocketData);
        return 1;
    }

    // check if this is the end of the response header
    if((i + 1) >= strlen(szEndOfResponseHeader))
    {
        if(strncmp(&szResponseHeader[(i + 1) - strlen(szEndOfResponseHeader)],
        szEndOfResponseHeader, strlen(szEndOfResponseHeader)) == 0)
        {
```

```
            // found end of response header
            dwFoundEndOfResponseHeader = 1;
            break;
        }
    }
}

// ensure the end of the response header was found
if(dwFoundEndOfResponseHeader == 0)
{
    // error
    printf("Error: Failed to read HTTP response header\n");
    NTSockets_CloseSocket(&SocketData);
    return 1;
}

printf("Received HTTP response:\n%s", szResponseHeader);

// convert response header to upper-case (for the content-length value search below)
for(i = 0; i < strlen(szResponseHeader); i++)
{
    // convert to upper-case (for the content-length value search below)
    szResponseHeader[i] = toupper(szResponseHeader[i]);
}

// check status code
memset(szResponseSuccessStatus, 0, sizeof(szResponseSuccessStatus));
strncpy(szResponseSuccessStatus, "HTTP/1.0 200 OK\r\n",
sizeof(szResponseSuccessStatus) - 1);
if(strncmp(szResponseHeader, szResponseSuccessStatus,
strlen(szResponseSuccessStatus)) != 0)
{
    // error
    printf("Error: Invalid response status code\n");
    NTSockets_CloseSocket(&SocketData);
    return 1;
}

// get content-length value
memset(szContentLengthParamName, 0, sizeof(szContentLengthParamName));
strncpy(szContentLengthParamName, "CONTENT-LENGTH: ",
sizeof(szContentLengthParamName) - 1);
pContentLength = strstr(szResponseHeader, szContentLengthParamName);
if(pContentLength != NULL)
{
    // content-length field exists
    pContentLength += strlen(szContentLengthParamName);
    pEndOfContentLength = strstr(pContentLength, "\r\n");
    if(pEndOfContentLength == NULL)
    {
        // error
        printf("Error: Invalid response header\n");
        NTSockets_CloseSocket(&SocketData);
        return 1;
```

```
    }

    *pEndOfContentLength = '\0';
    dwOutputLength = atoi(pContentLength);

    // process response data
    if(dwOutputLength != 0)
    {
        // allocate output data
        pOutputBuffer = (BYTE*)malloc(dwOutputLength);
        if(pOutputBuffer == NULL)
        {
            // error
            printf("Error: Failed to allocate memory\n");
            NTSockets_CloseSocket(&SocketData);
            return 1;
        }

        // read output data
        if(NTSockets_Recv(&SocketData, pOutputBuffer, dwOutputLength) != 0)
        {
            // error
            printf("Error: Failed to read HTTP response data\n");
            NTSockets_CloseSocket(&SocketData);
            return 1;
        }
    }
}
else
{
    // no content-length field - read until socket closes
    for(;;)
    {
        // read output data
        if(NTSockets_Recv(&SocketData, &bCurrByte, 1) != 0)
        {
            // finished
            break;
        }

        // check if the output buffer is large enough
        if(dwOutputLength >= dwOutputAllocLength)
        {
            // reallocate output buffer - add 8kb
            dwOutputAllocLength += 8192;
            if(pOutputBuffer == NULL)
            {
                // first buffer
                pOutputBuffer = (BYTE*)malloc(dwOutputAllocLength);
                if(pOutputBuffer == NULL)
                {
                    // error
                    printf("Error: Failed to allocate memory\n");
                    NTSockets_CloseSocket(&SocketData);
```

```
                        return 1;
                    }
                }
                else
                {
                    // reallocate existing buffer
                    pNewOutputBuffer = (BYTE*)realloc(pOutputBuffer,
                    dwOutputAllocLength);
                    if(pNewOutputBuffer == NULL)
                    {
                        // error
                        printf("Error: Failed to allocate memory\n");
                        NTSockets_CloseSocket(&SocketData);
                        free(pOutputBuffer);
                        return 1;

                    }

                    // update ptr
                    pOutputBuffer = pNewOutputBuffer;
                }
            }

            // store current byte
            *(BYTE*)(pOutputBuffer + dwOutputLength) = bCurrByte;
            dwOutputLength++;
        }
    }

    // close socket
    NTSockets_CloseSocket(&SocketData);

    // store data
    *pOutput = pOutputBuffer;
    *pdwOutputLength = dwOutputLength;

    return 0;
}

int main(int argc, char *argv[])
{
    BYTE *pOutput = NULL;
    DWORD dwLength = 0;
    char *pURL = NULL;
    char *pOutputPath = NULL;
    HANDLE hOutputFile = NULL;
    DWORD dwBytesWritten = 0;

    printf("NTSockets - File Download PoC - www.x86matthew.com\n\n");

    if(argc != 3)
    {
        printf("Usage: %s [url] [output_file_path]\n\n", argv[0]);
        return 1;
```

```
}

// get param
pURL = argv[1];
pOutputPath = argv[2];

// get NtDeviceIoControlFile function ptr
NtDeviceIoControlFile = (unsigned long (__stdcall *)(void *,void *,void *,
void *,struct IO_STATUS_BLOCK *,unsigned long,void *,unsigned long,void *,
unsigned long))GetProcAddress(GetModuleHandle("ntdll.dll"),
"NtDeviceIoControlFile");

if(NtDeviceIoControlFile == NULL)
{
    return 1;
}

// get NtCreateFile function ptr
NtCreateFile = (unsigned long (__stdcall *)(void ** ,unsigned long,struct
OBJECT_ATTRIBUTES *,struct IO_STATUS_BLOCK *,union _LARGE_INTEGER *,unsigned
long,unsigned long,unsigned long,unsigned long,void *,unsigned long))
GetProcAddress(GetModuleHandle("ntdll.dll"), "NtCreateFile");

if(NtCreateFile == NULL)
{
    return 1;
}

printf("Downloading file: %s\n\n", pURL);

// download file
if(DownloadFile(pURL, &pOutput, &dwLength) != 0)
{
    printf("Failed to download file\n");
    return 1;
}

printf("Downloaded %u bytes successfully\n\n", dwLength);

printf("Creating output file: %s\n", pOutputPath);

// create output file
hOutputFile = CreateFile(pOutputPath, GENERIC_WRITE, 0, NULL, CREATE_ALWAYS,
0, NULL);

if(hOutputFile == INVALID_HANDLE_VALUE)
{
    printf("Failed to create output file\n");
    return 1;
}

// write output data to file
if(WriteFile(hOutputFile, pOutput, dwLength, &dwBytesWritten, NULL) == 0)
{
```

```
        printf("Failed to write output data to file\n");
        return 1;
    }

    // close output file
    CloseHandle(hOutputFile);

    if(dwLength != 0)
    {
        // free buffer
        free(pOutput);
    }

    printf("\nFinished\n");

    return 0;
}
```