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

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

Posted: 04/03/2022

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

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
*pBackgroundConsoleInstance)
{

    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,
 "\\\\.\\pipe\\BackgroundConsoleIn_%s", pInstanceName);
hConsoleInputPipe = CreateNamedPipe(szConsoleInputPipeName,
PIPE_ACCESS_OUTBOUND, PIPE_TYPE_BYTE | PIPE_READMODE_BYTE | PIPE_WAIT,
1, 4096, 4096, 0, NULL);

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",
szConsoleInputPipeName);


// launch cmd.exe
if(CreateProcess(NULL, szLaunchCmd, NULL, NULL, 0, CREATE_NEW_CONSOLE,
NULL, NULL, &StartupInfo, &ProcessInfo) == 0)
{
    // 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(BackgroundConsoleInstance.szInstanceName) - 1);
    BackgroundConsoleInstance.hConsoleProcess = ProcessInfo.hProcess;
    BackgroundConsoleInstance.hConsoleInputPipe = hConsoleInputPipe;
    memcpy((void*)pBackgroundConsoleInstance, (void*)&BackgroundConsoleInstance,
     sizeof(BackgroundConsoleInstance));

    return 0;
}


DWORD BackgroundConsole_Close(BackgroundConsoleInstanceStruct
*pBackgroundConsoleInstance)
{

    // 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
*pBackgroundConsoleInstance, char *pCommand, DWORD (*pCommandOutput)(BYTE
*pBufferData, DWORD dwBufferLength, BYTE *pParam), BYTE *pCommandOutputParam)
{
    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,
     "\\\\.\\pipe\\BackgroundConsoleOut_%s",
     pBackgroundConsoleInstance->szInstanceName);
    hCommandOutputPipe = CreateNamedPipe(szCommandOutputPipeName,
    PIPE_ACCESS_INBOUND, PIPE_TYPE_BYTE | PIPE_READMODE_BYTE | PIPE_WAIT, 1,
    4096, 4096, 0, NULL);
    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, szCommandOutputPipeName);
if(WriteFile(pBackgroundConsoleInstance->hConsoleInputPipe,
szWriteCommand, strlen(szWriteCommand), NULL, NULL) == 0)
{
    // 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
    // case this is string data)
    memset(bReadBuffer, 0, sizeof(bReadBuffer));
    if(ReadFile(hCommandOutputPipe, bReadBuffer, sizeof(bReadBuffer) -  1,
    &dwBytesRead, NULL) == 0)
    {
        // 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 -
        pCommandOutput_StoreDataParam->dwTotalSize))
        {
            return 1;
        }

        // copy data
        memcpy((void*)(pCommandOutput_StoreDataParam->pOutputPtr +
        pCommandOutput_StoreDataParam->dwTotalSize), pBufferData, dwBufferLength);
    }

    // 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_StoreDataParam));
    CommandOutput_StoreDataParam.pOutputPtr = NULL;
    CommandOutput_StoreDataParam.dwMaxOutputSize = 0;
    CommandOutput_StoreDataParam.dwTotalSize = 0;
    if(BackgroundConsole_Exec(&BackgroundConsoleInstance,
    "type %windir%\\system32\\ntdll.dll", CommandOutput_StoreData,
    (BYTE*)&CommandOutput_StoreDataParam) != 0)
    {
        return 1;
    }

    printf("ntdll.dll file size: %u bytes - allocating memory...\n",
    CommandOutput_StoreDataParam.dwTotalSize);

    // 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_StoreDataParam));
    CommandOutput_StoreDataParam.pOutputPtr = pNtdllCopy;
    CommandOutput_StoreDataParam.dwMaxOutputSize = dwAllocSize;
    CommandOutput_StoreDataParam.dwTotalSize = 0;
    if(BackgroundConsole_Exec(&BackgroundConsoleInstance, "type
    %windir%\\system32\\ntdll.dll", CommandOutput_StoreData,
    (BYTE*)&CommandOutput_StoreDataParam) != 0)
    {
        return 1;
    }

    printf("Read %u bytes successfully\n", CommandOutput_StoreDataParam.dwTotalSize);

    // (pNtdllCopy now contains a copy of ntdll)

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

    return 0;
}
```