# Fourteen Ways to Read the PID for the Local Security Authority Subsystem Service (LSASS)


mdsec.co.uk/2022/08/fourteen-ways-to-read-the-pid-for-the-local-security-authority-subsystem-service-lsass

2 August 2022

ActiveBreach

## Introduction

Process enumeration is necessary prior to injecting shellcode or dumping memory. Threat actors tend to favour using CreateToolhelp32Snapshot with Process32First and Process32Next to gather a list of running processes. And if they're a bit more tech-savvy, they'll use the NtQuerySystemInformation system call directly.

Although this post will focus on obtaining a PID specifically for LSASS, the methods described here can be adapted to resolve PIDs for any process. Some of these are well known and have been discussed before, but there's also a few new ones that many readers won't be familiar with. Just to be clear, there are ways to obtain PIDs using Window objects, but LSASS doesn't have any and won't be discussed here.

## 1. NtQuerySystemInformation / NtQuerySystemInformationEx

CreateToolhelp32Snapshot() and EnumProcesses both use this system call with the SystemProcessInformation class. It's common to see this syscall used when avoiding Win32 API. The data returned also contains thread information, so it's used by Thread32First and Thread32Next. Modules, on the other hand, must be read manually from the PEB of each process, so Module32First and Module32Next require additional syscalls like NtOpenProcess and NtReadVirtualMemory.

```
//
// Use NtQuerySystemInformation(SystemProcessInformation)
//
DWORD
GetLsaPidFromName(void) {
    NTSTATUS Status;
    DWORD ProcessId = 0, Length = 1024;
    std::vector ProcessList(1024);

    do {
        do {
            Status = NtQuerySystemInformation(
                        SystemProcessInformation,
                        ProcessList.data(),
                        Length,
                        &Length
                        );

            if (Status == STATUS_INFO_LENGTH_MISMATCH) {
                ProcessList.resize(Length);
            }
        } while (Status == STATUS_INFO_LENGTH_MISMATCH);

        if (!NT_SUCCESS(Status)) {
            SetLastError(RtlNtStatusToDosError(Status));
            printf("NtQuerySystemInformation(SystemProcessInformation) failed : %ld",
GetLastError());
            break;
        }

        DWORD Offset = 0;
        PSYSTEM_PROCESS_INFORMATION ProcessEntry = NULL;

        do {
            ProcessEntry = (PSYSTEM_PROCESS_INFORMATION)(ProcessList.data() +
Offset);

            if (!lstrcmpiW(ProcessEntry->ImageName.Buffer, L"lsass.exe")) {
                ProcessId = HandleToLong(ProcessEntry->UniqueProcessId);
                break;
            }

            Offset += ProcessEntry->NextEntryOffset;
        } while (ProcessEntry->NextEntryOffset);
    } while(FALSE);

    return ProcessId;
}
```

## 2. Windows Terminal Services (WTS)

The WTSEnumerateProcesses API uses an RPC service to obtain a list of processes. Internally, the service uses NtQuerySystemInformation(SystemProcessInformation), but since the call is made in another process, it may at one time have been useful for evading process enumeration detections.

```
//
// Use Windows Terminal Services
//
DWORD
GetLsaPidFromWTS(void) {
    DWORD ProcessId = 0;

    do {
        PWTS_PROCESS_INFO info = NULL;
        DWORD cnt = 0;

        BOOL Result;
        Result = WTSEnumerateProcesses(WTS_CURRENT_SERVER_HANDLE, NULL, 1, &info,
&cnt);

        if (!Result) {
            printf("WTSEnumerateProcesses() failed : %ld\n", GetLastError());
            break;
        }

        for(DWORD i=0; i<cnt; i++) {
            if (!lstrcmpiW(info[i].pProcessName, L"lsass.exe")) {
                ProcessId = info[i].ProcessId;
                break;
            }
        }
        WTSFreeMemory(info);
    } while (FALSE);

    return ProcessId;
}
```

## 3. Windows Management Instrumentation (WMI)

We can obtain a list of instances for the class Win32_Process using WMI. The invocation of NtQuerySystemInformation() occurs within the WMI provider host (wmiprvse.exe). Like WTS, the enumeration occurs within another process and may be useful for evading detections.

```
//
// Windows Management Instrumentation (WMI)
//
DWORD
GetPidForLsass(IWbemServices *svc) {
    IEnumWbemClassObject *ent = NULL;
    DWORD ProcessId = 0;

    do {
        HRESULT hr;

        hr = svc->CreateInstanceEnum(
                L"Win32_Process",
                WBEM_FLAG_RETURN_IMMEDIATELY |
                WBEM_FLAG_FORWARD_ONLY,
                NULL,
                &ent);

        if (FAILED(hr)) {
            printf("IWbemServices::CreateInstanceEnum() failed : %08lX\n", hr);
            break;
        }

        for (;!ProcessId;) {
            ULONG cnt = 0;
            IWbemClassObject *obj = NULL;

            hr  = ent->Next(INFINITE, 1, &obj, &cnt);

            if(!cnt) break;

            VARIANT name;
            VariantInit(&name);

            hr = obj->Get(L"Name", 0, &name, NULL, NULL);

            if (SUCCEEDED(hr)) {
                if (!lstrcmpiW(V_BSTR(&name), L"lsass.exe")) {
                    VARIANT pid;
                    VariantInit(&pid);

                    hr = obj->Get(L"ProcessID", 0, &pid, NULL, NULL);

                    if (SUCCEEDED(hr)) {
                        ProcessId = V_UI4(&pid);
                        VariantClear(&pid);
                    }
                }
                VariantClear(&name);
            }
            obj->Release();
        }
```

```
    } while(FALSE);

    if (ent) ent->Release();

    return ProcessId;
}

//
// Read instances of Win32_Process and filter by Name.
//
DWORD
GetLsaPidFromWMI(void) {
    IWbemLocator  *loc = NULL;
    IWbemServices *svc = NULL;
    DWORD ProcessId = 0;

    do {
        HRESULT hr = CoInitialize(NULL);

        if (FAILED(hr)) {
            printf("CoInitialize() failed : %08lX\n", hr);
            break;
        }

        hr = CoInitializeSecurity(
                NULL,
                -1,
                NULL,
                NULL,
                RPC_C_AUTHN_LEVEL_DEFAULT,
                RPC_C_IMP_LEVEL_IMPERSONATE,
                NULL,
                EOAC_NONE,
                NULL);

        if (FAILED(hr)) {
            printf("CoInitializeSecurity() failed : %08lX\n", hr);
            break;
        }

        hr = CoCreateInstance(
                CLSID_WbemLocator,
                0,
                CLSCTX_INPROC_SERVER,
                IID_IWbemLocator,
                (LPVOID*)&loc);

        if (FAILED(hr)) {
            printf("CoInitializeSecurity() failed : %08lX\n", hr);
            break;
        }
```

```
        hr = loc->ConnectServer(
                    L"root\\cimv2",
                    NULL,
                    NULL,
                    NULL,
                    0,
                    NULL,
                    NULL,
                    &svc);

        if (FAILED(hr)) {
            printf("IWbemLocator::ConnectServer() failed : %08lX\n", hr);
            break;
        }

        ProcessId = GetPidForLsass(svc);
    } while (FALSE);

    if (svc) svc->Release();
    if (loc) loc->Release();

    CoUninitialize();

    return ProcessId;
}
```

## 4. NtQueryValueKey (LsaPid)

We can read the PID directly from the registry value *LsaPid* stored at **HKLM\SYSTEM\CurrentControlSet\Control\Lsa**.

```
//
// Query registry for LSA process ID.
//
DWORD
GetLsaPidFromRegistry(void) {
    UNICODE_STRING                  LsaPath, LsaValue;
    OBJECT_ATTRIBUTES               ObjectAttributes;
    HANDLE                          LsaKey = NULL;
    NTSTATUS                        Status;
    UCHAR                           Buffer[sizeof(KEY_VALUE_PARTIAL_INFORMATION) +
sizeof(DWORD)];
    PKEY_VALUE_PARTIAL_INFORMATION PartialInfo =
(PKEY_VALUE_PARTIAL_INFORMATION)Buffer;
    ULONG                           Length;
    DWORD                           ProcessId = 0;

    do {
        LsaPath =
RTL_CONSTANT_STRING(L"\\Registry\\Machine\\SYSTEM\\CurrentControlSet\\Control\\Lsa");

        InitializeObjectAttributes(
            &ObjectAttributes,
            &LsaPath,
            OBJ_CASE_INSENSITIVE,
            NULL,
            NULL);

        Status = NtOpenKey(&LsaKey, KEY_QUERY_VALUE, &ObjectAttributes);

        if (!NT_SUCCESS(Status)) {
            SetLastError(RtlNtStatusToDosError(Status));
            printf("NtOpenKey() failed : %ld\n", GetLastError());
            break;
        }

        LsaValue = RTL_CONSTANT_STRING(L"LsaPid");

        Status = NtQueryValueKey(
                    LsaKey,
                    &LsaValue,
                    KeyValuePartialInformation,
                    Buffer,
                    sizeof(Buffer),
                    &Length);

        if (!NT_SUCCESS(Status)) {
            SetLastError(RtlNtStatusToDosError(Status));
            printf("NtQueryValueKey() failed : %ld\n", GetLastError());
            break;
        }

        PDWORD pData = (PDWORD)&PartialInfo->Data[0];
```

```
        ProcessId = pData[0];
    } while(FALSE);

    if (LsaKey) NtClose(LsaKey);
    return ProcessId;
}
```

## 5. QueryServiceStatusEx(SAMSS)

LSASS hosts a number of services:

1. CNG KeyIsolation (KeyIso)
2. Security Accounts Manager (SamSs)
3. Credential Manager (VaultSvc)

We can open any of these services to query the status and use SC_STATUS_PROCESS_INFO to obtain the process ID. Like WTS, this information is acquired over RPC and is a good way to obtain the PID of a service.

```
//
// Query samss for LSA process ID.
//
DWORD
GetLsaPidFromService(void) {
    SC_HANDLE              ManagerHandle = NULL, ServiceHandle = NULL;
    SERVICE_STATUS_PROCESS ProcessInfo;
    HANDLE                 Handle = NULL;
    DWORD                  Length, ProcessId = 0;
    BOOL                   Result;

    do {
        ManagerHandle = OpenSCManagerW(
                            NULL,
                            NULL,
                            SC_MANAGER_CONNECT
                        );

        if (!ManagerHandle) {
            printf("OpenSCManager() failed : %ld\n", GetLastError());
            break;
        }

        ServiceHandle = OpenServiceW(
                            ManagerHandle,
                            L"samss",
                            SERVICE_QUERY_STATUS
                        );

        if (!ServiceHandle) {
            printf("OpenService() failed : %ld\n", GetLastError());
            break;
        }

        Result = QueryServiceStatusEx(
                    ServiceHandle,
                    SC_STATUS_PROCESS_INFO,
                    (LPBYTE)&ProcessInfo,
                    sizeof(ProcessInfo),
                    &Length
                );

        if (!Result) {
            printf("QueryServiceStatusEx() failed : %ld\n", GetLastError());
            break;
        }

        ProcessId = ProcessInfo.dwProcessId;
    } while(FALSE);

    if (ServiceHandle) {
        CloseServiceHandle(ServiceHandle);
```

```
    }

    if (ManagerHandle) {
        CloseServiceHandle(ManagerHandle);
    }

    return ProcessId;
}
```

## 6. NtQueryInformationFile(lsass.exe)

If we open a file handle to LSASS.EXE and pass it to NtQueryInformationFile with the FileProcessIdsUsingFileInformation class, we can obtain a list of process IDs that have the file opened, and the first in the list should always belong to the LSASS process. Typically, the LSASS binary is located in C:\Windows\System32\ folder, but it's safer to use something like GetSystemDirectory() to obtain the correct path.

```
//
// Query process path for process ID
//
DWORD
GetLsaPidFromPath(void) {
    IO_STATUS_BLOCK   StatusBlock;
    NTSTATUS          Status;
    WCHAR             LsassPath[MAX_PATH + 1];
    DWORD             ProcessId = 0;
    HANDLE            FileHandle = NULL;
    OBJECT_ATTRIBUTES ObjectAttributes;
    UNICODE_STRING    NtPath;

    do {
        //
        // Get the Windows directory.
        //
        GetSystemDirectoryW(LsassPath, MAX_PATH);
        PathAppendW(LsassPath, L"lsass.exe");

        //
        // Convert DOS path to NT path
        //
        RtlDosPathNameToNtPathName_U(
            LsassPath,
            &NtPath,
            NULL,
            NULL
            );

        //
        // Open file for reading.
        //
        InitializeObjectAttributes(
            &ObjectAttributes,
            &NtPath,
            OBJ_CASE_INSENSITIVE,
            0,
            NULL
        );

        Status = NtOpenFile(
                &FileHandle,
                FILE_READ_ATTRIBUTES,
                &ObjectAttributes,
                &StatusBlock,
                FILE_SHARE_READ,
                NULL
            );

        if (!NT_SUCCESS(Status)) {
            SetLastError(RtlNtStatusToDosError(Status));
```

```
            printf("NtOpenFile() failed : %ld\n", GetLastError());
            break;
        }

        //
        // Get list of process IDs with this process path opened.
        //
        std::vector Buffer;

        for (DWORD Length=4096;;Length += 4096) {
            Buffer.resize(Length);

            Status = NtQueryInformationFile(
                        FileHandle,
                        &StatusBlock,
                        Buffer.data(),
                        Buffer.size(),
                        FileProcessIdsUsingFileInformation
                        );

            if (Status != STATUS_INFO_LENGTH_MISMATCH) break;
        }

        if (!NT_SUCCESS(Status)) {
            SetLastError(RtlNtStatusToDosError(Status));
            printf("NtQueryInformationFile() failed : %ld\n", GetLastError());
            break;
        }

        auto PidFileInfo = (PFILE_PROCESS_IDS_USING_FILE_INFORMATION)Buffer.data();

        if (PidFileInfo->NumberOfProcessIdsInList) {
            ProcessId = DWORD(PidFileInfo->ProcessIdList[0]);
        }
    } while(FALSE);

    if (FileHandle) NtClose(FileHandle);
    return ProcessId;
}
```

## 7. NtFsControlFile(named pipe)

The recommended way of obtaining the PID to a named pipe server is by using the GetNamedPipeServerProcessId API. Internally, this will call the GetNamedPipeAttribute() API, which as of August 2022, remains undocumented by MSDN. When invoked, it will eventually execute the NtFsControlFile syscall. A client or server can use this to obtain a session ID, process ID or computer name from its peer.

```
BOOL GetNamedPipeAttribute(
        HANDLE Pipe,
        PIPE_ATTRIBUTE_TYPE AttributeType,
        PSTR AttributeName,
        PVOID AttributeValue,
        PSIZE_T AttributeValueLength);
```

To obtain the PID, open the named pipe “\Device\NamedPipe\lsass” and use NtFsControlFile to send the FSCTL_PIPE_GET_PIPE_ATTRIBUTE control code with “ServerProcessId” as the input, which should return the process ID.

```
//
// Get the LSA PID from named pipe.
//
DWORD
GetLsaPidFromPipe(void) {
    UNICODE_STRING    LsaName = RTL_CONSTANT_STRING(L"\\Device\\NamedPipe\\lsass");
    HANDLE            LsaHandle;
    IO_STATUS_BLOCK   StatusBlock;
    OBJECT_ATTRIBUTES ObjectAttributes;
    NTSTATUS          Status;
    DWORD             ProcessId = 0;

    do {
        //
        // Open named pipe for reading.
        //
        InitializeObjectAttributes(
            &ObjectAttributes,
            &LsaName,
            OBJ_CASE_INSENSITIVE,
            0,
            NULL
        );

        Status = NtOpenFile(
                    &LsaHandle,
                    FILE_READ_ATTRIBUTES,
                    &ObjectAttributes,
                    &StatusBlock,
                    FILE_SHARE_READ,
                    NULL
                );

        if (!NT_SUCCESS(Status)) {
            SetLastError(RtlNtStatusToDosError(Status));
            printf("NtOpenFile() failed : %ld\n", GetLastError());
            break;
        }

        //
        // Query the server process ID.
        //
        LPSTR Attribute = "ServerProcessId";

        Status = NtFsControlFile(
                     LsaHandle,
                     NULL,
                     NULL,
                     NULL,
                     &StatusBlock,
                     FSCTL_PIPE_GET_PIPE_ATTRIBUTE,
                     Attribute,
```

```
                    lstrlenA(Attribute) + 1,
                    &ProcessId,
                    sizeof(DWORD));

        if (!NT_SUCCESS(Status)) {
            SetLastError(RtlNtStatusToDosError(Status));
            printf("NtFsControlFile() failed : %ld\n", GetLastError());
        }

    } while(FALSE);

    if (LsaHandle) NtClose(LsaHandle);

    return ProcessId;
}
```

## 8. NtQueryOpenSubKeysEx(SAM)

In the same way we can query what process has a file opened, we can use NtQueryOpenSubKeysEx to obtain a list of PIDs with a registry key opened. For LSASS, it's the only process to have HKLM\SAM opened. Unfortunately this method requires the SeRestorePrivilege, but if you're planning to open LSASS, you'll need at least SYSTEM or SeDebugPrivilege enabled anyway.

```
//
// NtQueryOpenSubKeysEx() : Requires Admin rights to enable restore privilege.
//
DWORD
GetLsaPidFromRegName(void) {
    UNICODE_STRING    RegName = RTL_CONSTANT_STRING(L"\\REGISTRY\\MACHINE\\SAM");
    OBJECT_ATTRIBUTES ObjectAttributes;
    NTSTATUS          Status;
    DWORD             ProcessId = 0;

    do {
        //
        // The restore privilege is required for this to work.
        //
        BOOLEAN Old;
        Status = RtlAdjustPrivilege(SE_RESTORE_PRIVILEGE, TRUE, FALSE, &Old);

        if (!NT_SUCCESS(Status)) {
            SetLastError(RtlNtStatusToDosError(Status));
            printf("RtlAdjustPrivilege(SE_RESTORE_PRIVILEGE) failed : %ld\n",
GetLastError());
            break;
        }

        InitializeObjectAttributes(
            &ObjectAttributes,
            &RegName,
            OBJ_CASE_INSENSITIVE,
            NULL,
            NULL
        );

        std::vector OutBuffer(1024);
        ULONG OutLength = 1024;

        Status = NtQueryOpenSubKeysEx(
                    &ObjectAttributes,
                    OutLength,
                    OutBuffer.data(),
                    &OutLength
                );

        if (!NT_SUCCESS(Status)) {
            SetLastError(RtlNtStatusToDosError(Status));
            printf("NtQueryOpenSubKeysEx() failed : %ld\n", GetLastError());
            break;
        }

        PKEY_OPEN_SUBKEYS_INFORMATION SubKeyInfo =
(PKEY_OPEN_SUBKEYS_INFORMATION)OutBuffer.data();
        ProcessId = HandleToUlong(SubKeyInfo->KeyArray[0].ProcessId);
```

```
    } while(FALSE);

    return ProcessId;
}
```

## 9. RegQueryValueExW(HKEY_PERFORMANCE_DATA)

Some of you may know already that pslist by sysinternals uses performance counters to obtain process information. Perhaps the reason this isn't popular for process enumeration right now is because it's quite noisy in obtaining the information. Higher on the list is the fact it's poorly documented. There are of course sources on the internet demonstrating how to use the counters, but they typically date back to the 1990s.

Initially, this method appeared to be very stealthy, but when you realise the amount of operations required to obtain a list of processes, it clearly isn't the best approach. Still, performance data is one area of Windows that doesn't get enough attention. There's a goldmine of data in there.

```
//
// Read LSASS process ID from performance counters.
//
DWORD
GetLsaPidFromPerf(void) {
    DWORD ProcessId = 0;
    std::vector Buffer;

    //
    // Read performance data for each process.
    //
    for (DWORD Length=8192;;Length += 8192) {
        Buffer.resize(Length);

        DWORD rc = RegQueryValueExW(
                        HKEY_PERFORMANCE_DATA,
                        L"230",
                        NULL,
                        0,
                        Buffer.data(),
                        &Length);

        if (rc == ERROR_SUCCESS) break;
    }

    //
    // Read offset for the process ID.
    //
    auto pPerf = (PPERF_DATA_BLOCK)Buffer.data();
    auto pObj = (PPERF_OBJECT_TYPE) ((PBYTE)pPerf + pPerf->HeaderLength);
    auto pCounterDef = (PPERF_COUNTER_DEFINITION) ((PBYTE)pObj + pObj->HeaderLength);

    DWORD ProcessIdOffset = 0;

    for (auto i=0; iNumCounters; i++) {
        if (pCounterDef->CounterNameTitleIndex == 784) {
            ProcessIdOffset = pCounterDef->CounterOffset;
            break;
        }
        pCounterDef++;
    }

    //
    // Read process name and compare with lsass
    //
    auto pInst = (PPERF_INSTANCE_DEFINITION) ((PBYTE)pObj + pObj->DefinitionLength);

    for (auto i=0; iNumInstances; i++) {
        auto pName = (PWSTR) ((PBYTE)pInst + pInst->NameOffset);
        auto pCounter = (PPERF_COUNTER_BLOCK) ((PBYTE)pInst + pInst->ByteLength);

        if (*pName && !lstrcmpiW(pName, L"lsass")) {
```

```
            ProcessId = *((LPDWORD) ((PBYTE)pCounter + ProcessIdOffset));
            break;
        }
        pInst = (PPERF_INSTANCE_DEFINITION) ((PBYTE)pCounter + pCounter->ByteLength);
    }

    RegCloseKey(HKEY_PERFORMANCE_DATA);
    return ProcessId;
}
```

## 10. NtDeviceIoControlFile(TCP Table)

Since Vista, it's possible obtain a process ID for a network connection from the Network Store Interface Service (NSI). LSASS typically has at least one port listening for incoming connections and if we know the port, we can read the PID using some code. The example uses NtDeviceIoControlFile with an undocumented IOCTL code and undocumented structures. Another approach for legacy systems is opening the socket handle and sending IOCTL_TDI_QUERY_INFORMATION.

```
typedef struct _NSI_CONNECTION_INFO {
    PVOID               Buffer;
    SIZE_T              Size;
} NSI_CONNECTION_INFO, *PNSI_CONNECTION_INFO;

typedef struct _NSI_CONNECTION_ENTRIES {
    NSI_CONNECTION_INFO Address;
    NSI_CONNECTION_INFO Reserved;
    NSI_CONNECTION_INFO State;
    NSI_CONNECTION_INFO Process;
} NSI_CONNECTION_ENTRIES, *PNSI_CONNECTION_ENTRIES;

typedef enum _NPI_MODULEID_TYPE {
    MIT_GUID = 1,
    MIT_IF_LUID,
} NPI_MODULEID_TYPE;

typedef struct _NPI_MODULEID {
    USHORT            Length;
    NPI_MODULEID_TYPE Type;
    union {
        GUID          Guid;
        LUID          IfLuid;
    };
} NPI_MODULEID, *PNPI_MODULEID;

NPI_MODULEID NPI_MS_TCP_MODULEID = {
        sizeof(NPI_MODULEID),
        MIT_GUID,
        {0xEB004A03, 0x9B1A, 0x11D4, {0x91, 0x23, 0x00, 0x50, 0x04, 0x77, 0x59,
0xBC}}
};

// the following structures were reverse engineered and won't be correct.
typedef struct _NSI_CONNECTION_TABLE {
    DWORD                 Unknown1[4];
    PNPI_MODULEID         ModuleId;
    DWORD64               TypeId;
    ULONG64               Flags;
    NSI_CONNECTION_ENTRIES Entries;
    DWORD                 NumberOfEntries;
} NSI_CONNECTION_TABLE, *PNSI_CONNECTION_TABLE;

typedef union _NSI_IP_ADDR_U {
    sockaddr_in  v4;
    sockaddr_in6 v6;
} NSI_IP_ADDR_U, *PNSI_IP_ADDR_U;

typedef struct _NSI_CONNECTION_ADDRESS {
    NSI_IP_ADDR_U Local;
    NSI_IP_ADDR_U Remote;
} NSI_CONNECTION_ADDRESS, *PNSI_CONNECTION_ADDRESS;
```

```
typedef struct _NSI_CONNECTION_STATE {
    PULONG        ulState;
    ULONG         ulTimestamp;
} NSI_CONNECTION_STATE, *PNSI_CONNECTION_STATE;

typedef struct _NSI_CONNECTION_PROCESS {
    DWORD         dwOwningPidUdp;
    BOOL          bFlag;
    DWORD         liCreateTimestampUdp;
    DWORD         dwOwningPidTcp;
    LARGE_INTEGER liCreateTimestamp;
    ULONGLONG     OwningModuleInfo;
} NSI_CONNECTION_PROCESS, *PNSI_CONNECTION_PROCESS;

#define FSCTL_TCP_BASE     FILE_DEVICE_NETWORK

#define _TCP_CTL_CODE(Function, Method, Access) \
    CTL_CODE(FSCTL_TCP_BASE, Function, Method, Access)

#define NSI_IOCTL_GET_INFORMATION _TCP_CTL_CODE(0x006, METHOD_NEITHER,
FILE_ANY_ACCESS)

//
// Read the LSA PID from TCP table.
//
DWORD
GetLsaPidFromTcpTable(void) {
    DWORD                    ProcessId = 0;
    HANDLE                   NsiHandle = NULL;
    NTSTATUS                 Status;
    IO_STATUS_BLOCK          IoStatusBlock;
    PNSI_CONNECTION_ADDRESS  Address = NULL;
    PNSI_CONNECTION_STATE    State = NULL;
    PNSI_CONNECTION_PROCESS  Process = NULL;

    do {
        //
        // Open handle to Network Store Interface Service. (nsiproxy.sys)
        //
        NsiHandle = CreateFileW(
                        L"\\\\.\\Nsi",
                        0,
                        FILE_SHARE_READ | FILE_SHARE_WRITE,
                        NULL,
                        OPEN_EXISTING,
                        0,
                        NULL);

        if (NsiHandle == INVALID_HANDLE_VALUE) break;

        //
```

```
        // Tell service to return information for TCP connections.
        // The first call obtains the number of entries available.
        //
        NSI_CONNECTION_TABLE TcpTable={0};

        TcpTable.ModuleId = &NPI_MS_TCP_MODULEID;
        TcpTable.TypeId = 3; // TCP connections.
        TcpTable.Flags = 1 | 0x100000000;

        Status = NtDeviceIoControlFile(
                    NsiHandle,
                    NULL,    // no event object. making a synchronous request.
                    NULL,
                    NULL,
                    &IoStatusBlock,
                    NSI_IOCTL_GET_INFORMATION,  // ioctl code to return all
information
                    &TcpTable,          // in
                    sizeof(TcpTable),
                    &TcpTable,          // out
                    sizeof(TcpTable)
                    );

        if (!NT_SUCCESS(Status)) {
            SetLastError(RtlNtStatusToDosError(Status));
            printf("NtDeviceIoControlFile() failed : %ld\n", GetLastError());
            break;
        }

        //
        // Allocate memory for entries.
        //
        Address = (PNSI_CONNECTION_ADDRESS)calloc(TcpTable.NumberOfEntries + 2,
sizeof(NSI_CONNECTION_ADDRESS));
        State = (PNSI_CONNECTION_STATE)calloc(TcpTable.NumberOfEntries + 2,
sizeof(NSI_CONNECTION_STATE));
        Process = (PNSI_CONNECTION_PROCESS)calloc(TcpTable.NumberOfEntries + 2,
sizeof(NSI_CONNECTION_PROCESS));

        //
        // Assign buffers and the size of each structure.
        // Then try again.
        //
        TcpTable.Entries.Address.Buffer = Address;
        TcpTable.Entries.Address.Size = sizeof(NSI_CONNECTION_ADDRESS);

        TcpTable.Entries.State.Buffer = State;
        TcpTable.Entries.State.Size = sizeof(NSI_CONNECTION_STATE);

        TcpTable.Entries.Process.Buffer = Process;
        TcpTable.Entries.Process.Size = sizeof(NSI_CONNECTION_PROCESS);
```

```
        Status = NtDeviceIoControlFile(
                    NsiHandle,
                    NULL,    // no event object. making a synchronous request.
                    NULL,
                    NULL,
                    &IoStatusBlock,
                    NSI_IOCTL_GET_INFORMATION,  // ioctl code to return all
information
                    &TcpTable,          // in
                    sizeof(TcpTable),
                    &TcpTable,          // out
                    sizeof(TcpTable)
                    );

        if (!NT_SUCCESS(Status)) {
            SetLastError(RtlNtStatusToDosError(Status));
            printf("NtDeviceIoControlFile() failed : %ld\n", GetLastError());
            break;
        }

        //
        // Loop through each entry to find LSASS ports.
        // When found, return the Process ID.
        //
        for (DWORD i=0; i= 49664 && lport <= 49667 || lport == 49155) {
                ProcessId = pid;
                break;
            }
        }

    } while (FALSE);

    if (Address) free(Address);
    if (State) free(State);
    if (Process) free(Process);
    if (NsiHandle) NtClose(NsiHandle);

    return ProcessId;
}
```

## 11. Security Event Log

Event ID 4608 AKA ‘Windows is starting up’ contains the process ID of LSASS. It can be extracted using the Windows Event Log API. There are of course other events to consider: e.g. Logon events. 4608 just happens to be consistent from Vista to Windows 11/Server 2022.

```
//
// Query the PID for LSASS from the Security event log.
//
DWORD
GetLsaPidFromEventLogs(void) {
    EVT_HANDLE hResults = NULL, hContext = NULL, hEvent = NULL;
    DWORD      dwProcessId = 0;

    do {
        //
        // Get all records from the Security log with event ID 4608
        //
        hResults = EvtQuery(
                    NULL,
                    L"Security",
                    L"*/*[EventID=4608]",
                    EvtQueryTolerateQueryErrors
                    );

        if (!hResults) {
            printf("EvtQuery(Security, EventID=4608) failed : %ld\n",
GetLastError());
            break;
        }

        //
        // Move position of results to the last entry. (the latest available)
        //
        BOOL Result;
        Result = EvtSeek(hResults, 0, NULL, 0, EvtSeekRelativeToLast);

        if (!Result) {
            printf("EvtSeek() failed : %ld\n", GetLastError());
            break;
        }

        //
        // Read last event.
        //
        DWORD dwReturned = 0;

        Result = EvtNext(
                    hResults,
                    1,
                    &hEvent,
                    INFINITE, 0,
                    &dwReturned);

        if (!Result || dwReturned != 1) {
            printf("EvtNext() failed : %ld\n", GetLastError());
            break;
        }
```

```
        //
        // Create a render context so that we only extract the PID
        //
        LPCWSTR ppValues[] = {L"Event/System/Execution/@ProcessID"};

        hContext = EvtCreateRenderContext(
                    ARRAYSIZE(ppValues),
                    ppValues,
                    EvtRenderContextValues);

        if (!hContext) {
            printf("EvtCreateRenderContext failed : %ld\n", GetLastError());
            break;
        }

        //
        // Extract the PID.
        //
        EVT_VARIANT pProcessId={0};

        EvtRender(
            hContext,
            hEvent,
            EvtRenderEventValues,
            sizeof(EVT_VARIANT),
            &pProcessId,
            &dwReturned,
            NULL
            );

        //
        // Save it.
        //
        dwProcessId = pProcessId.UInt32Val;
    } while (FALSE);

    if (hEvent) EvtClose(hEvent);
    if (hContext) EvtClose(hContext);
    if (hResults) EvtClose(hResults);

    return dwProcessId;
}
```

## 12. Brute Forcing PIDs

PID values are incremented by 4, so we can easily cycle through potential PIDs and query the image.

```
BOOL
GetProcessNameById(DWORD ProcessId, WCHAR ImageName[MAX_PATH + 1]) {
    WCHAR                          ImageBuffer[512];
    SYSTEM_PROCESS_ID_INFORMATION  ProcessInformation;
    NTSTATUS                       Status;
    DWORD                          Length;

    //
    // Query the system for image name of process ID.
    //
    ProcessInformation.ProcessId              = LongToHandle(ProcessId);
    ProcessInformation.ImageName.Buffer       = ImageBuffer;
    ProcessInformation.ImageName.Length       = 0;
    ProcessInformation.ImageName.MaximumLength = sizeof(ImageBuffer);

    Status = NtQuerySystemInformation(
              SystemProcessIdInformation,
              &ProcessInformation,
              sizeof(ProcessInformation),
              NULL
              );

    if (!NT_SUCCESS(Status)) {
        SetLastError(RtlNtStatusToDosError(Status));
        //printf("NtQuerySystemInformation(SystemProcessIdInformation) failed :
%ld\n", GetLastError());
        return FALSE;
    }

    //
    // Strip path and copy name to buffer.
    //
    PathStripPathW(ImageBuffer);
    Length = ProcessInformation.ImageName.Length;

    if (Length > (MAX_PATH * sizeof(WCHAR))) {
        Length = MAX_PATH * sizeof(WCHAR);
    }

    memcpy(
      ImageName,
      ImageBuffer,
      Length
      );

    return TRUE;
}

//
// Run a loop. Increment PID and resolve an image name. Then compare with lsass.exe
//
DWORD
```

```
GetLsaPidFromBruteForce(void) {
    DWORD ProcessId = 0;

    WCHAR ImageName[MAX_PATH + 1]={0};

    //
    // 0xFFFFFFFC is the maximum PID but probably too much for this.
    //
    for (DWORD pid=8; pid<0xFFFFFFFC; pid += 4) {
        if (GetProcessNameById(pid, ImageName)) {
            if (!lstrcmpiW(ImageName, L"lsass.exe")) {
                ProcessId = pid;
                break;
            }
        }
    }
    return ProcessId;
}
```

## 13. Section Object

LSASS creates a section object for storing information about its performance aptly named “\LsaPerformance” Inside the data is among other things the PID. Performance Monitor Users have read access while SYSTEM and Admins have full access. The structure of data is undocumented and the PID can appear at different offsets, so this would need work to improve reliability.

```
//
// Only tested on Windows 10. The offset of PID differs across versions.
//
DWORD
GetLsaPidFromSection(void) {
    UNICODE_STRING    SectionName = RTL_CONSTANT_STRING(L"\\LsaPerformance");
    OBJECT_ATTRIBUTES ObjectAttributes;
    NTSTATUS          Status;
    DWORD             ProcessId = 0;
    HANDLE            SectionHandle = NULL;
    PVOID             ViewBase = NULL;

    do {
        InitializeObjectAttributes(
            &ObjectAttributes,
            &SectionName,
            OBJ_CASE_INSENSITIVE,
            NULL,
            NULL
        );

        Status = NtOpenSection(
                    &SectionHandle,
                    SECTION_MAP_READ,
                    &ObjectAttributes
                    );

        if (!NT_SUCCESS(Status)) {
            SetLastError(RtlNtStatusToDosError(Status));
            printf("NtOpenSection() failed : %ld\n", GetLastError());
            break;
        }

        LARGE_INTEGER SectionOffset={0};
        SIZE_T ViewSize = 0;

        Status = NtMapViewOfSection(
                  SectionHandle,
                  NtCurrentProcess(),
                  &ViewBase,
                  0,              // ZeroBits
                  0,              // CommitSize
                  &SectionOffset,
                  &ViewSize,
                  ViewShare,
                  0,
                  PAGE_READONLY
                  );

        if (!NT_SUCCESS(Status)) {
            SetLastError(RtlNtStatusToDosError(Status));
            printf("NtMapViewOfSection() failed : %ld\n", GetLastError());
```

```
            break;
        }

        PDWORD data = (PDWORD)ViewBase;

        ProcessId = data[32]; // Windows 10. it appears at different offsets for
different builds.
    } while(FALSE);

    if (ViewBase) NtUnmapViewOfSection(NtCurrentProcess(), ViewBase);
    if (SectionHandle) NtClose(SectionHandle);

    return ProcessId;
}
```

## 14. NtAlpcQueryInformation

Available since Windows 10 version 1909 “19H2” is the ability to query an ALPC service for its session and process ID. The example here uses “\RPC Control\samss lpc” which should be running inside the LSASS process. This isn’t always reliable. Sometimes NtAlpcQueryInformation() will fail with STATUS_INVALID_PARAMETER.

```
//
// Query process ID from session information. Only works on 19H2 builds.
//
DWORD
GetLsaPidFromAlpc(void) {
    HANDLE            AlpcPort = NULL;
    UNICODE_STRING    AlpcName;
    DWORD             ProcessId = 0;
    NTSTATUS          Status;
    OBJECT_ATTRIBUTES ObjectAttributes;

    do {
        //
        // Connect to RPC service.
        //
        AlpcName = RTL_CONSTANT_STRING(L"\\RPC Control\\samss lpc");

        Status = NtAlpcConnectPort(
                    &AlpcPort,
                    &AlpcName,
                    NULL,
                    NULL,
                    0,
                    NULL,
                    NULL,
                    NULL,
                    NULL,
                    NULL,
                    NULL);

        if (!NT_SUCCESS(Status)) {
            SetLastError(RtlNtStatusToDosError(Status));
            printf("NtAlpcConnectPort() failed : %ld\n", GetLastError());
            break;
        }

        //
        // Query session information.
        //
        ALPC_SERVER_SESSION_INFORMATION SessionInfo={0};

        Status = NtAlpcQueryInformation(
                    AlpcPort,
                    AlpcServerSessionInformation,
                    &SessionInfo,
                    sizeof(SessionInfo),
                    NULL);

        if (!NT_SUCCESS(Status)) {
            SetLastError(RtlNtStatusToDosError(Status));
            printf("NtAlpcQueryInformation() failed : %ld\n", GetLastError());
            break;
```

```
        }

        ProcessId = SessionInfo.ProcessId;
    } while(FALSE);

    if (AlpcPort) NtClose(AlpcPort);

    return ProcessId;
}
```

## Summary

As you can see, there are many ways to obtain a PID for a process. For LSASS, reading the ServerProcessId attribute from a named pipe seems like the most elegant approach. It doesn't require any allocation of memory, works on at least Vista up to Windows 11 and should continue working well into the future. One more that wasn't discussed here would involve enumerating handles and counting how many security tokens a process has opened. The entry with the highest score should be LSASS, but of course there's a possibility of identifying the wrong process with this approach too.

This blog post was written by @modexpblog