# Techniques for Detecting and Mitigating Native API Usage in Windows

## 1 Introduction

Native API in Windows refers to a set of undocumented low-level functions exposed by `ntdll.dll`. These APIs provide the underlying interface to the Windows NT kernel. While most application developers use the higher-level Win32 API, advanced programmers and malware authors often use Native APIs to access system functionality not available through standard libraries or to evade detection.

## 2 Motivations for Using Native APIs

- **Bypassing security software:** Antivirus and EDR solutions often monitor Win32 API calls. Native API usage can evade such detection.
- **Access to undocumented features:** Some kernel functionalities are only exposed via native calls.
- **Performance and stealth:** Direct system calls reduce overhead and traceability.

## 3 Common Native API Examples

```
#include <Windows.h>
#include <winternl.h>

extern "C" NTSTATUS NTAPI NtQueryInformationProcess(
    HANDLE, PROCESSINFOCLASS, PVOID, ULONG, PULONG);

bool IsDebugged() {
    DWORD debugPort = 0;
    NtQueryInformationProcess(GetCurrentProcess(),
      ↪ ProcessDebugPort, &debugPort, sizeof(DWORD), NULL);
    return debugPort != 0;
}
```

Listing 1: Using NtQueryInformationProcess to detect a debugger

## 4 Assembly-Level Native API Invocation

```
mov eax, 0x2C                ; Syscall number for
   ↪ NtTerminateProcess (example)
mov rcx, -1                  ; Handle to current process
mov rdx, 0                   ; Exit status
syscall                      ; Invoke system call directly
```

Listing 2: Manual syscall to NtTerminateProcess

**Note:** Syscall numbers vary by Windows version. Using them directly requires version checks or dynamic resolution.

## 5 Detecting Native API Usage

Defenders can detect Native API use through:

- **ETW (Event Tracing for Windows)**
- **Sysmon**: Captures suspicious process activity.
- **Process Monitor (ProcMon)**: Logs Native API calls.
- **Inline Hooking**: Injecting code into `ntdll.dll` to monitor function calls.

## 6 Obfuscating Native API Usage

Malware often avoids detection by:

- Resolving function addresses dynamically using `GetProcAddress`.
- Encrypting strings containing API names.
- Using direct syscalls to skip `ntdll.dll`.

```
HMODULE ntdll = LoadLibraryA("ntdll.dll");
auto NtWriteVirtualMemory = (decltype(&::NtWriteVirtualMemory))
    GetProcAddress(ntdll, "NtWriteVirtualMemory");
```

Listing 3: Dynamic resolution of NtWriteVirtualMemory

## 7 Hooking Native API Calls

To monitor or modify behavior, defenders can hook native functions:

```
NTSTATUS HookedNtQueryInformationProcess(...) {
    if (InfoClass == ProcessDebugPort) {
        *(PDWORD)Buffer = 0;
        return STATUS_SUCCESS;
    }
    return OriginalNtQueryInformationProcess(...);
}
```

Listing 4: Hooked NtQueryInformationProcess function

## 8 Kernel-Level Defenses

Advanced techniques include:

- **SSDT hooking** (in older systems).
- **EDR solutions** with behavior-based anomaly detection.
- **Hypervisors** for syscall tracing.

## 9 Conclusion

Native API use is a double-edged sword. It provides powerful low-level access and efficiency but is often abused for stealth and evasion. Awareness of its usage, combined with proactive detection and defense, is essential for maintaining Windows system integrity.