# Anti-Debug: Assembly instructions

anti-debug.checkpoint.com/techniques/assembly.html

## Contents



## Assembly instructions

The following techniques are intended to detect a debugger presence based on how debuggers behave when the CPU executes a certain instruction.

## 1. INT 3

Instruction `INT3` is an interruption which is used as a software breakpoint. Without a debugger present, after getting to the `INT3` instruction, the exception `EXCEPTION_BREAKPOINT` (`0x80000003`) is generated and an exception handler will be called. If the debugger is present, the control won't be given to the exception handler.

### C/C++ Code

```
bool IsDebugged()
{
    __try
    {
        __asm int 3;
        return true;
    }
    __except(EXCEPTION_EXECUTE_HANDLER)
    {
        return false;
    }
}
```

Besides the short form of `INT3` instruction (`0xCC` opcode), there is also a long form of this instruction: `CD 03` opcode.

When the exception `EXCEPTION_BREAKPOINT` occurs, the Windows decrements `EIP` register to the assumed location of the `0xCC` opcode and pass the control to the exception handler. In the case of the long form of the `INT3` instruction, `EIP` will point to the middle of the instruction (i.e. to `0x03` byte). Therefore, `EIP` should be edited in the exception handler if we want to continue execution after the `INT3` instruction (otherwise we'll most likely get an `EXCEPTION_ACCESS_VIOLATION` exception). If not, we can neglect the instruction pointer modification.

**C/C++ Code**

```
bool g_bDebugged = false;

int filter(unsigned int code, struct _EXCEPTION_POINTERS *ep)
{
    g_bDebugged = code != EXCEPTION_BREAKPOINT;
    return EXCEPTION_EXECUTE_HANDLER;
}

bool IsDebugged()
{
    __try
    {
        __asm __emit(0xCD);
        __asm __emit(0x03);
    }
    __except (filter(GetExceptionCode(), GetExceptionInformation()))
    {
        return g_bDebugged;
    }
}
```

## 2. INT 2D

Just like in the case of `INT3` instruction when the instruction `INT2D` is executed, the exception `EXCEPTION_BREAKPOINT` is raised as well. But with `INT2D`, Windows uses the `EIP` register as an exception address and then increments the `EIP` register value. Windows also examines the value of the `EAX` register while `INT2D` is executed. If it's 1, 3 or 4 on all versions of Windows, or 5 on Vista+, the exception address will be increased by one.

This instruction can cause problems for some debuggers because after the `EIP` incrimination, the byte which follows the `INT2D` instruction will be skipped and the execution might continue from the damaged instruction.

In the example, we put one-byte `NOP` instruction after `INT2D` to skip it in any case. If the program is executed without a debugger, the control will be passed to the exception handler.

**C/C++ Code**

```
bool IsDebugged()
{
    __try
    {
        __asm xor eax, eax;
        __asm int 0x2d;
        __asm nop;
        return true;
    }
    __except(EXCEPTION_EXECUTE_HANDLER)
    {
        return false;
    }
}
```

## 3. ICE

"`ICE`" is one of Intel's undocumented instructions. Its opcode is `0xF1`. It can be used to detect if the program is traced.

If `ICE` instruction is executed, the `EXCEPTION_SINGLE_STEP` (`0x80000004`) exception will be raised.

However, if the program has been already traced, the debugger will consider this exception as the normal exception generated by executing the instruction with the SingleStep bit set in the Flags registers. Therefore, under a debugger, the exception handler won't be called and execution will continue after the `ICE` instruction.

### C/C++ Code

```
bool IsDebugged()
{
    __try
    {
        __asm __emit 0xF1;
        return true;
    }
    __except(EXCEPTION_EXECUTE_HANDLER)
    {
        return false;
    }
}
```

## 4. Stack Segment Register

This is a trick that can be used to detect if the program is being traced. The trick consists of tracing over the following sequence of assembly instructions:

```
push ss
pop ss
pushf
```

After single-stepping in a debugger through this code, the Trap Flag will be set. Usually it's not visible as debuggers clear the Trap Flag after each debugger event is delivered. However, if we previously save `EFLAGS` to the stack, we'll be able to check whether the Trap Flag is set.

### C/C++ Code

```
bool IsDebugged()
{
    bool bTraced = false;

    __asm
    {
        push ss
        pop ss
        pushf
        test byte ptr [esp+1], 1
        jz movss_not_being_debugged
    }

    bTraced = true;

movss_not_being_debugged:
    // restore stack
    __asm popf;

    return bTraced;
}
```

## 5. Instruction Counting

This technique abuses how some debuggers handle `EXCEPTION_SINGLE_STEP` exceptions.

The idea of this trick is to set hardware breakpoints to each instruction in some predefined sequence (e.g. sequence of `NOP`s). Execution of the instruction with a hardware breakpoint on it raises the `EXCEPTION_SINGLE_STEP` exception which can be caught by a vectored exception handler. In the exception handler, we increment a register which plays the role of instruction counter (`EAX` in our case) and the instruction pointer `EIP` to pass the control to the next instruction in the sequence. Therefore, each time the control is passed to the next instruction in our sequence, the exception is raised and the counter is incremented. After the sequence is finished, we check the counter and if it is not equal to the length of our sequence, we consider it as if the program is being debugged.

**C/C++ Code**

```
#include "hwbrk.h"

static LONG WINAPI InstructionCountingExeptionHandler(PEXCEPTION_POINTERS
pExceptionInfo)
{
    if (pExceptionInfo->ExceptionRecord->ExceptionCode == EXCEPTION_SINGLE_STEP)
    {
        pExceptionInfo->ContextRecord->Eax += 1;
        pExceptionInfo->ContextRecord->Eip += 1;
        return EXCEPTION_CONTINUE_EXECUTION;
    }
    return EXCEPTION_CONTINUE_SEARCH;
}

__declspec(naked) DWORD WINAPI InstructionCountingFunc(LPVOID lpThreadParameter)
{
    __asm
    {
        xor eax, eax
        nop
        nop
        nop
        nop
        cmp al, 4
        jne being_debugged
    }

    ExitThread(FALSE);

being_debugged:
    ExitThread(TRUE);
}

bool IsDebugged()
{
    PVOID hVeh = nullptr;
    HANDLE hThread = nullptr;
    bool bDebugged = false;

    __try
    {
        hVeh = AddVectoredExceptionHandler(TRUE, InstructionCountingExeptionHandler);
        if (!hVeh)
            __leave;

        hThread = CreateThread(0, 0, InstructionCountingFunc, NULL, CREATE_SUSPENDED,
0);
        if (!hThread)
            __leave;

        PVOID pThreadAddr = &InstructionCountingFunc;
        // Fix thread entry address if it is a JMP stub (E9 XX XX XX XX)
```

```
        if (*(PBYTE)pThreadAddr == 0xE9)
            pThreadAddr = (PVOID)((DWORD)pThreadAddr + 5 + *(PDWORD)
((PBYTE)pThreadAddr + 1));

        for (auto i = 0; i < m_nInstructionCount; i++)
            m_hHwBps[i] = SetHardwareBreakpoint(
                hThread, HWBRK_TYPE_CODE, HWBRK_SIZE_1, (PVOID)((DWORD)pThreadAddr +
2 + i));

        ResumeThread(hThread);
        WaitForSingleObject(hThread, INFINITE);

        DWORD dwThreadExitCode;
        if (TRUE == GetExitCodeThread(hThread, &dwThreadExitCode))
            bDebugged = (TRUE == dwThreadExitCode);
    }
    __finally
    {
        if (hThread)
            CloseHandle(hThread);

        for (int i = 0; i < 4; i++)
        {
            if (m_hHwBps[i])
                RemoveHardwareBreakpoint(m_hHwBps[i]);
        }

        if (hVeh)
            RemoveVectoredExceptionHandler(hVeh);
    }

    return bDebugged;
}
```

## 6. POPF and Trap Flag

This is another trick that can indicate whether a program is being traced.

There is a Trap Flag in the Flags register. When the Trap Flag is set, the exception `SINGLE_STEP` is raised. However, if we traced the code, the Trap Flag will be cleared by a debugger so we won't see the exception.

**C/C++ Code**

```
bool IsDebugged()
{
    __try
    {
        __asm
        {
            pushfd
            mov dword ptr [esp], 0x100
            popfd
            nop
        }
        return true;
    }
    __except(GetExceptionCode() == EXCEPTION_SINGLE_STEP
        ? EXCEPTION_EXECUTE_HANDLER
        : EXCEPTION_CONTINUE_EXECUTION)
    {
        return false;
    }
}
```

## 7. Instruction Prefixes

This trick works only in some debuggers. It abuses the way how these debuggers handle instruction prefixes.

If we execute the following code in OllyDbg, after stepping to the first byte `F3`, we'll immediately get to the end of `try` block. The debugger just skips the prefix and gives the control to the `INT1` instruction.

If we run the same code without a debugger, an exception will be raised and we'll get to `except` block.

**C/C++ Code**

```
bool IsDebugged()
{
    __try
    {
        // 0xF3 0x64 disassembles as PREFIX REP:
        __asm __emit 0xF3
        __asm __emit 0x64
        // One byte INT 1
        __asm __emit 0xF1
        return true;
    }
    __except(EXCEPTION_EXECUTE_HANDLER)
    {
        return false;
    }
}
```

## Mitigations

- During debugging:
    - The best way to mitigate all the following checks is to patch them with `NOP` instructions.
    - Regarding anti-tracing techniques: instead of patching the code, we can simply set a breakpoint in the code which follows the check and run the program till this breakpoint.
- For anti-anti-debug tool development: No mitigation.