Anti-Debugging
A Developers Viewpoint

March 13, 2008

VERACODE

Presenter Background

- Currently
  - Sr. Security Researcher for Veracode, Inc.
- Previously
  - Security Consultant - Symantec
  - Security Consultant - @Stake
  - Incident Response and Forensics Handler – US Government
- Wishes He Was
  - Infinitely Rich
  - Personal Trainer to hot Hollywood starlets
  - General all-around rockstar

What we will cover…

- Definition of Problem Statement and Terms
- Why Bother?
  - Isn't this a futile effort?
- Standing On The Shoulders of Giants
  - Brief reference of anti-debugging
- Classes of Anti-Debugging Methods
  - The six styles of anti-debug fu
- Some Known Anti-Debugging Methods
  - Not enough time to cover them all

Problem Statement and Definition of Terms

VERACODE

Problem Statement

Anti-debugging methods are consistently presented as machine code or assembly language constructs making knowledge transfer difficult and limiting widespread use of the techniques.

Let's Do An Experiment

Can you tell what the following code does in five seconds or less?

Be Honest!

Assembly Dump

```
push   ebp
mov    ebp, esp
sub    esp, 0C0h
push   ebx
push   esi
push   edi
lea    edi, [ebp+var_C0]
mov    ecx, 30h
mov    eax, 0CCCCCCCCh
rep stosd
mov    esi, esp
call   ds:__imp__IsDebuggerPresent@0
cmp    esi, esp
call   j___RTC_CheckEsp
test   eax, eax
jz     short loc_411400
mov    esi, esp
push   0          ; uType
push   offset Caption
push   offset Text
push   0          ; hWnd
call   ds:__imp__MessageBoxW@16
cmp    esi, esp
call   j___RTC_CheckEsp
short loc_41141D
```

```
loc_411400:                    ; CODE XREF: antidebug(void)+2Fj
mov    esi, esp
push   0          ; uType
push   offset aNoDebugger
push   offset aNoDebuggerDete
push   0          ; hWnd
call   ds:__imp__MessageBoxW@16
cmp    esi, esp
j___RTC_CheckEsp

loc_41141D:                    ; CODE XREF: antidebug(void)+4Ej
pop    edi
pop    esi
pop    ebx
add    esp, 0C0h
cmp    ebp, esp
call   j___RTC_CheckEsp
mov    esp, ebp
pop    ebp
Retn
antidebug@@YAXXZ endp
```

Assembly Dump

Assembly Dump – Highlight Important Bits

```
var_C0 = byte ptr -0C0h
push    ebp
mov     ebp, esp
sub     esp, 0C0h
Push    ebx
Push    esi
Push    edi
lea     edi, [ebp+var_C0]
mov     ecx, 30h
mov     eax, 0CCCCCCCCh
Rep stosd
mov     esi, esp
Call    ds:__imp__IsDebuggerPresent@0
cmp     esi, esp
Call j___RTC_CheckEsp
Test    eax, eax
jz      short loc_411400
mov     esi, esp
push    0 ; uType
push    offset Caption
push    offset Text
push    0 ; hWnd
call    ds:__imp__MessageBoxW@16
cmp     esi, esp
call    j___RTC_CheckEsp
jmp     short loc_41141D

loc_411400:
mov     esi, esp
push    0 ; uType
push    offset aNoDebugger
push    offset aNoDebuggerDete
push    0 ; hWnd
call    ds:__imp__MessageBoxW@16
cmp     esi, esp
call    j___RTC_CheckEsp
loc_41141D:
pop     edi
pop     esi
pop     ebx
add     esp, 0C0h
cmp     ebp, esp
call    j___RTC_CheckEsp
mov     esp, ebp
pop     ebp
Retn
?antidebug@@YAXXZ endp
```

Assembly Dump

How About In C

```
if (IsDebuggerPresent())
{
   MessageBox(NULL, L"Debugger Detected Via
       IsDebuggerPresent", L"Debugger Detected", MB_OK);
}  else
{
   MessageBox(NULL, L“No Debugger Detected“, L“No Debugger ",
       MB_OK);
}
```

Definition of Terms

- Debugger (Debugging)
  - The act of detecting and removing bugs in an application.

- Anti-Debugging
  - Anti-debugging is the implementation of one or more techniques within computer code that hinders attempts at reverse engineering or debugging a target process.

Definition of Terms

- Dumping
  - Dumping of a running process in memory to disk so that the image can be executed as a separate binary.

- Anti-Dumping
  - Technique that is used to inhibit the dumping of a running process from memory to disk

Definition of Terms

- Anti-Anti-Debugging
  - Techniques used to detect and bypass anti-debugging efforts
- Anti-Anti-Dumping
  - Techniques used to detect and bypass anti-dumping efforts.

- Anti-Anti-Anti-…. You get the picture

Why Bother?

Isn’t this a futile effort

VERACODE

Why Should I Care?

- Protecting your intellectual property
  - Laws can't possibly work
    - Neither do EULA or other soft methods
  - Slow down subversion of software registration
    - Cracking
  - Impede application feature and code theft
- Low cost
  - Short code segments equals quick implementation
  - Write a basic anti-debugging library for reuse

Why Should I Care?

- Raise the bar
  - Implement multiple layers of defense
  - Makes reversing an arduous task
  - It's no longer as simple as a debugger + disassembler + time
- Malware analysis
  - Knowledge of anti-debugging techniques can help your incident response and analysis
  - Debug the random binary found on your accounting system
  - Implement IDS/IPS signatures in shorter timeframes
- Exercise the brain
  - Keeps the synapses firing
  - Implementation (and bypass) is FUN!

Standing On the Shoulders Of Giants

A Brief Reference of Anti-Debugging

VERACODE

References

- Bania, P, "Antidebugging for the (m)asses - protecting the env."
- Brulez, N., "Crimeware Anti-Reverse Engineering Uncovered"
- "Lord Julus", "Anti-Debugger and Anti-Emulator Lair"
- Liston, T., and Skoudis, E., " Thwarting Virtual Machine Detection"
- Bania, P., "Playing with RTDSC"
- Quist, D., Smith, V., "Detecting the Presence of Virtual Machines Using the Local Data Table"
- Omella, A. A., "Methods for Virtual Machine Detection"
- Tariq, T. B., "Detecting Virtualization"
- Rutkowska, J., " Red Pill... or how to detect VMM using (almost) one CPU instruction"

References

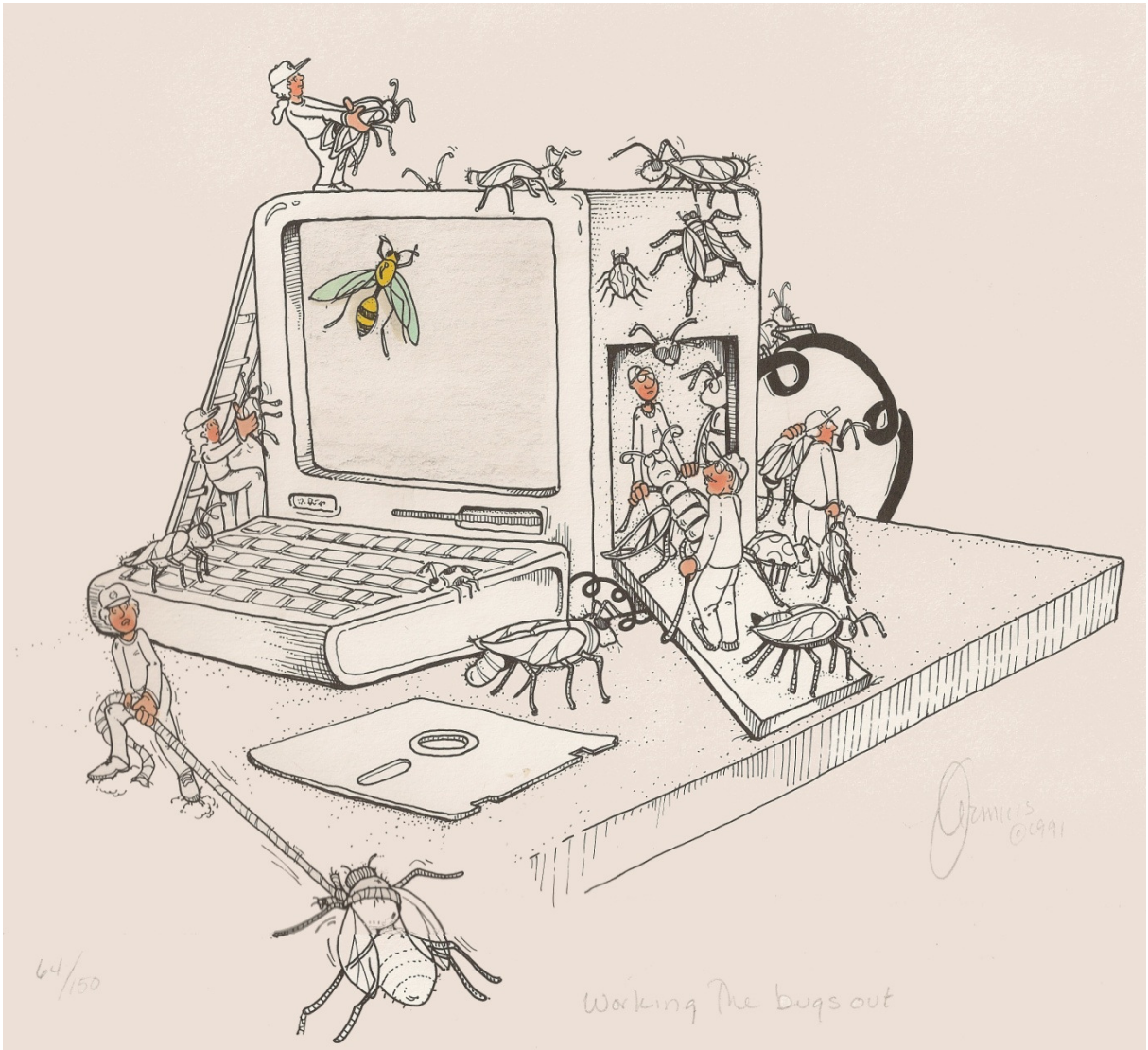
- Jackson, J., "An Anti-Reverse Engineering Guide"
- Falliere, N., "Windows Anti-Debug Reference"
- Gagnon, M. N., Taylor, S., and Ghosh, "Software Protection through Anti-Debugging"
- Ferrie, P., "Anti-Unpacker Tricks"
- OpenRCE: "Anti Reverse Engineering Techniques Database"
- Ferrie, P., "Attacks on More Virtual Machine Emulators"
- Brulez, N., "Anti-Reverse Engineering Uncovered"

Anti-Debugging Classes

VERACODE

VERACODE

Let's Get Started



VERACODE

Let's Get Started

Anti-Debugging Classes

- API Based Detection
- Process and Thread Block Detection
- Hardware and Register Based Detection
- Exception Based Detection
- Modified Code Based Detection
- Timing Based Detection

Anti-Debugging Classes

- API Based Detection
  - Using operating system supplied API calls to determine if a debugger exists and/or is attached to our code
  - No direct access of memory regions
  - Relies upon documented and undocumented operating system functions
- Process and Thread Block Detection
  - Bypassing API calls and directly querying process and thread information to determine discrepancies that indicate the operation of a debugger
  - When direct API calls just aren't enough
  - Avoids userland API hooking
  - Lower in the stack and closer to the OS results in more difficult bypass

Anti-Debugging Classes

- Hardware and Register Based Detection
  - Directly accessing CPU register information specifically debug registers to check for hardware breakpoints
  - No longer relies upon software differences for detection
  - One step lower in the stack
- Exception Based Detection
  - Creating an exception that is handled differently depending on the debugging state of a process

Anti-Debugging Classes

- Modified Code Based Detection
  - Process understands what it *should* look like in memory
  - Compares the current picture against the excepted picture
  - Flags on differences
- Timing Detection
  - Time calls used to determine execution deltas to be compared against a reasonable threshold
  - Primarily used to detect single stepping
  - Can also be used to detect breakpoints within blocks of code

API Anti-Debugging

VERACODE

Technology Background

- X86 Systems and Microsoft Windows Operating Systems
  - Windows 2000/XP/Vista
- Breakpoints
  - Software Breakpoints (Exceptions)
    - Overwrites target location with INT03 (0xcc)
    - Saves original opcode
    - Backs up one operation, replaces original opcode, restarts
  - Hardware Breakpoints (Faults)
    - Debug registers: 4 for addresses, one for control, and one for status
    - Each of the first 4 can hold a memory address that causes a hardware fault when accessed
    - Debug Registers (DR0 – DR7)
- Single Stepping
  - Trap Flag in processor is set
  - Any instruction will trigger a breakpoint when trap flag is set

API Anti-Debugging

- Pros
  - Generally easiest to implement
  - Easiest to understand
  - Short in length

- Cons
  - Easiest to understand
  - Relatively easy to bypass
  - Most frequently used

API Anti-Debugging

- FindWindow

```
HANDLE ollyHandle = NULL;
ollyHandle = FindWindow(L"OLLYDBG", 0);
if (ollyHandle == NULL) {
    MessageBox(NULL, L"OllyDbg Not Detected", L"Not Detected", MB_OK);
} else {
    MessageBox(NULL, L"Ollydbg Detected Via OllyDbg FindWindow()",
    L"OllyDbg Detected", MB_OK);
}
```

- Registry Key Detection

```
SOFTWARE\\Microsoft\\Windows NT\\CurrentVersion\\AeDebug
    Debugger=
exefile\\shell\\Open with Olly&Dbg\\command
dllfile\\shell\\Open with Olly&Dbg\\command
```

API Anti-Debugging

- IsDebuggerPresent() Windows API

```
if (IsDebuggerPresent()) {
    MessageBox(NULL, L"Debugger Detected Via IsDebuggerPresent",
    L"Debugger Detected", MB_OK);
}
```

- CheckRemoteDebuggerPresent() Windows API

```
CheckRemoteDebuggerPresent(GetCurrentProcess(), &pbIsPresent);
if (pbIsPresent) {
    MessageBox(NULL, L"Debugger Detected Via
    CheckRemoteDebuggerPresent", L"Debugger Detected", MB_OK);
}
```

API Anti-Debugging

- OutputDebugString on Win2K and WinXP

```
DWORD AnythingButTwo = 666;
SetLastError(AnythingButTwo);
OutputDebugString(L"foobar");
if (GetLastError() == AnythingButTwo) {
    MessageBox(NULL, L"Debugger Detected Via OutputDebugString",
    L"Debugger Detected", MB_OK);
} else {
    MessageBox(NULL, L"No Debugger Detected", L"No Debugger
    Detected", MB_OK);
}
```

API Anti-Debugging

- NtQueryInformationProcess() Detection

```
status = (_NtQueryInformationProcess) (-1, 7, &retVal, 4, NULL);
printf("Status Code: %08X - DebugPort: %08X", status, retVal);
if (retVal != 0) {
    MessageBox(NULL, L"Debugger Detected Via
    NtQueryInformationProcess ProcessDebugPort", L"Debugger Detected",
    MB_OK);
} else {
    MessageBox(NULL, L"No Debugger Detected", L"No Debugger
    Detected", MB_OK);
}
```

API Anti-Debugging

- NtSetInformationThread

```
lib = LoadLibrary(L"ntdll.dll");
_NtSetInformationThread = GetProcAddress(lib, "NtSetInformationThread");

(_NtSetInformationThread) (GetCurrentThread(), 0x11, 0, 0);
```

API Anti-Debugging

- Self Debugging with DebugActiveProcess

```
pid = GetCurrentProcessId();
_itow_s((int)pid, (wchar_t*)&pid_str, 8, 10);
wcsncat_s((wchar_t*)&szCmdline, 64, (wchar_t*)pid_str, 4);
success = CreateProcess(path, szCmdline, NULL, NULL, FALSE, 0, NULL,
   NULL, &si, &pi);
…
success = DebugActiveProcess(pid);
if (success == 0) {
   printf("Error Code: %d\n", GetLastError());
   MessageBox(NULL, L"Debugger Detected - Unable to Attach",
   L"Debugger Detected", MB_OK);
}
if (success == 1) MessageBox(NULL, L"No Debugger Detected", L"No
   Debugger", MB_OK);
```

API Anti-Debugging

- ProcessDebugFlags

```
hmod = LoadLibrary(L"ntdll.dll");
_NtQueryInformationProcess = GetProcAddress(hmod,
    "NtQueryInformationProcess");
status = (_NtQueryInformationProcess) (-1, 31, &debugFlag, 4, NULL);
```

- ProcessDebugObjectHandle

```
status = (_NtQueryInformationProcess) (-1, 0x1e, &debugObject, 4, NULL);
```

- OllyDbg OutputDebugString() Format String Vulnerability

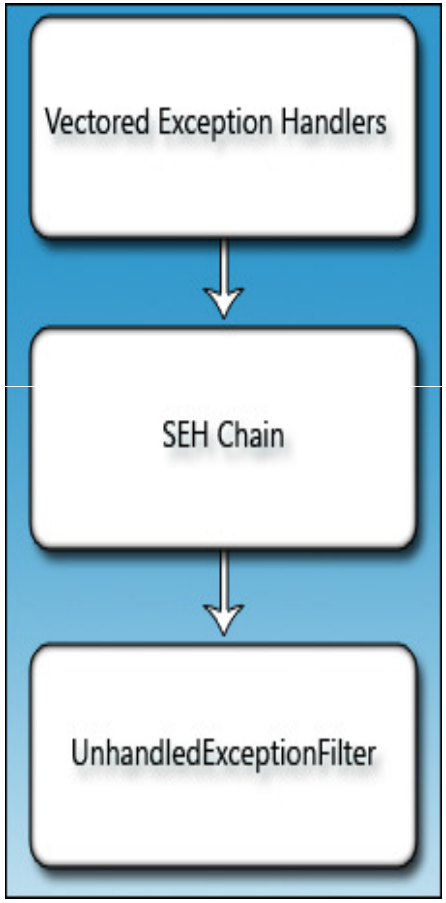
```
OutputDebugString(TEXT("%s%s%s%s%s%s%s%s%s%s%s%s"),
    TEXT("%s%s%s%s%s%s%s%s%s%s%s%s") );
}
__except (EXCEPTION_EXECUTE_HANDLER) {
    printf("Handled Exception\n");
}
```

Exception Anti-Debugging

VERACODE

Technology Background

- Exceptions
  - Vectored Exception Handling
    - Linked list of exceptions inserted before all other exceptions
  - Structured Exception Handling
    - Linked list of exceptions setup on a functional basis
  - Unhandled Exception Filter
    - A final catch all for exceptions to limit program crashes
- Debugger Interaction With Exceptions
  - First chance exceptions
  - Before Vectored Exception Handlers
  - Modify and return execution to process
  - Ignore and continue
  - Pass exception back to process for handling



Exception Based Anti-Debugging

- Pros
  - Slightly more complex
  - Sometimes more difficult to bypass (debugger doesn't know how to ignore the exception)

- Cons
  - Slightly more complex
  - Sometimes trivial to bypass (debugger ignore exception)
  - Typically longer in length

Exception Anti-Debugging

- INT 2D, INT 0x03, 0xF1 (ICE Breakpoint), 0xCC Detection

```
__try {
    __asm {
        int 2dh; // substitute int 3 or __emit 0xF1, 0xCC for other triggers
    }
}
__except (EXCEPTION_EXECUTE_HANDLER) {
    flag = 1;
    MessageBox(NULL, L"No Debugger Detected", L"No Debugger",
    MB_OK);
}
if (flag != 1) MessageBox(NULL, L"Debugger Detected via int2d",
    L"Debugger Detected", MB_OK);
```

Exception Based Anti-Debugging

- kernel32!CloseHandle

```
__try {
    CloseHandle(0x12345678);
}
__except (EXCEPTION_EXECUTE_HANDLER) {
    flag = 1;
    MessageBox(NULL, L"Debugger Detected via kernel32!CloseHandle",
    L"Debugger Detected", MB_OK);
}
if (flag == 0) MessageBox(NULL, L"No Debugger Detected", L"No
    Debugger", MB_OK);
```

Exception Anti-Debugging

- Single Step Detection (Trap Flag)

```
__try {
    __asm {
        PUSHFD; //Saves the flag registers
        OR BYTE PTR[ESP+1], 1; // Sets the Trap Flag in EFlags
        POPFD; //Restore the flag registers
        NOP; // NOP
    }
}
__except (EXCEPTION_EXECUTE_HANDLER) {
    flag = 1;
    MessageBox(NULL, L"No Debugger Detected", L"No Debugger",
    MB_OK);
}
if (flag == 0) MessageBox(NULL, L"Debugger Detected Via Trap Flag",
    L"Debugger Detected", MB_OK);
```

Exception Anti-Debugging

- OllyDbg Memory Breakpoint Detection

```
memRegion = VirtualAlloc(NULL, 0x10000, MEM_COMMIT,
   PAGE_READWRITE);
RtlFillMemory(memRegion, 0x10, 0xC3);
success = VirtualProtect(memRegion, 0x10, PAGE_EXECUTE_READ |
   PAGE_GUARD, &oldProt);
myproc = (FARPROC) memRegion;
success = 1;
__try {
   myproc();
}
__except (EXCEPTION_EXECUTE_HANDLER) {
   success = 0;
   MessageBox(NULL, L"No Debugger Detected", L"No Debugger
   Detected", MB_OK);
}
```

Exception Anti-Debugging

- Control C Vectored Exception Handling

```
AddVectoredExceptionHandler(1,
   (PVECTORED_EXCEPTION_HANDLER)exhandler);
SetConsoleCtrlHandler((PHANDLER_ROUTINE)sighandler, TRUE);
success = GenerateConsoleCtrlEvent(CTRL_C_EVENT, 0);
```

- Using the CMPXCHG8B with the LOCK Prefix

```
SetUnhandledExceptionFilter((LPTOP_LEVEL_EXCEPTION_FILTER)
   error);
__asm {
   __emit 0xf0;
   __emit 0xf0;
   __emit 0xc7;
   __emit 0xc8;
}
```

Process and Thread Block Anti-Debugging

VERACODE

Technology Background

- Process Environment Block (PEB)
  - Process Base Address
  - Heap Location
  - Memory Pointers
  - Global Flag
  - OS Version Information (Major/Minor)
  - isDebugged Flag
  - …
- Thread Information Block (TIB)
  - Thread ID
  - Process ID
  - Pointer to process PEB
  - Error Information
  - Exception Information
  - …

Process and Thread Block Anti-Debugging

- Pros
  - Direct to the process and thread information
  - Less chance of detection via hooking
  - Harder to bypass overall
- Cons
  - Requires use of undocumented structures
  - Requires understanding of runtime dynamic linking
  - More complex to understand

Process and Thread Block Anti-Debugging

- IsDebuggerPresent() Direct PEB Access

```
status = (_NtQueryInformationProcess) (hnd, ProcessBasicInformation,
   &pPIB, sizeof(PROCESS_BASIC_INFORMATION), &bytesWritten);
if (status == 0 ) {
   if (pPIB.PebBaseAddress->BeingDebugged == 1) {
        MessageBox(NULL, L"Debugger Detected Using PEB!IsDebugged",
   L"Debugger Detected", MB_OK);
   } else {
        MessageBox(NULL, L"No Debugger Detected", L"No Debugger
   Detected", MB_OK);
```

- PEB ProcessHeap Flag Debugger Detection

```
base = (char *)pPIB.PebBaseAddress;
procHeap = base+0x18;
procHeap = *procHeap;
heapFlag = (char*) procHeap+0x10;
last = (DWORD*) heapFlag;
if (*heapFlag != 0x00) { … Found a debugger }
```

Process and Thread Block Anti-Debugging

- NtGlobalFlag Debugger Detection

```
status = (_NtQueryInformationProcess) (hnd, ProcessBasicInformation,
    &pPIB, sizeof(PROCESS_BASIC_INFORMATION), &bytesWritten);
value = (pPIB.PebBaseAddress);
value = value+0x68;
if (*value == 0x70) {
    MessageBox(NULL, L"Debugger Detected Using PEB!NTGlobalFlag",
    L"Debugger Detected", MB_OK);
} else {
    MessageBox(NULL, L"No Debugger Detected", L"No Debugger
    Detected", MB_OK);
}
```

Process and Thread Block Anti-Debugging

- Vista TEB system DLL pointer

```
strPtr = TIB+0xBFC; // Offset into the target structure
delta = (int)(*strPtr) - (int)strPtr; // Ensure that string directly follows pointer
if (delta == 0x04) {
   if (wcscmp(*strPtr, hookStr)==0) { // Compare to our known bad string
        MessageBox(NULL, L"Debugger Detected Via Vista TEB System DLL
   PTR", L"Debugger Detected", MB_OK);
   } else {
        MessageBox(NULL, L"No Debugger Detected", L"No Debugger",
   MB_OK);
   }
} else {
   MessageBox(NULL, L"No Debugger Detected", L"No Debugger",
   MB_OK);
}
```

Hardware and Register Anti-Debugging

VERACODE

Hardware and Register Anti-Debugging

- Pros
  - Directly checks the hardware
  - Harder yet to bypass
  - Fairly easy to implement
- Cons
  - Not a whole lot of research in this area
  - Limited number of techniques available

Hardware Register Anti-Debugging

- Hardware Breakpoint Detection

```
hnd = GetCurrentThread();
status = GetThreadContext(hnd, &ctx);
if ((ctx.Dr0 != 0x00) || (ctx.Dr1 != 0x00) || (ctx.Dr2 != 0x00) || (ctx.Dr3 !=
   0x00) || (ctx.Dr6 != 0x00) || (ctx.Dr7 != 0x00))
{
   MessageBox(NULL, L"Debugger Detected Via DRx
   Modification/Hardware Breakpoint", L"Debugger Detected", MB_OK);
} else {
   MessageBox(NULL, L"No Debugger Detected", L"No Debugger",
   MB_OK);
}
```

Hardware Register Anti-Debugging

- Vmware sldt detection

```
__asm {
   sldt ldt_info;
}
if ((ldt_info[0] != 0x00) && (ldt_info[1] != 0x00)) ldt_flag = 1;
```

Timing Based Anti-Debugging

VERACODE

Timing Based Anti-Debugging

- Pros
  - Super easy to implement
  - Conceptually easy to understand
  - Can monitor blocks or individual instructions
- Cons
  - Very easy to bypass
  - Can be spotted a mile away

Timing Based Anti-Debugging

- RDTSC Instruction Debugger Latency Detection

```
i = __rdtsc();
j = __rdtsc();
if (j-i < 0xff) {
    MessageBox(NULL, L“No Debugger Detected Via RDTSC", L“No
    Debugger Detected", MB_OK);
} else {
    MessageBox(NULL, L"Debugger Detected Via RDTSC", L"Debugger
    Detected", MB_OK);
}
```

- NTQueryPerformanceCounter

```
QueryPerformanceCounter(&li);
QueryPerformanceCounter(&li2);
if ((li2.QuadPart-li.QuadPart) > 0xFF) {
```

- GetTickCount Timing
- timeGetTime Timing

Paper Link

http://www.veracode.com/images/pdf/whitepaper_antidebugging.pdf

Questions

VERACODE

VERACODE

Contact

Tyler Shields
Sr. Security Researcher
Veracode, Inc.
tshields@veracode.com