# Anti-Debugging – A Developers View

Tyler Shields

tshields@veracode.com

*Veracode Inc., USA*
*4 Van de Graaff Drive, Burlington, MA 01803*

*Abstract*— **Anti-debugging is the implementation of one or more techniques within computer code that hinders attempts at reverse engineering or debugging a target binary. Within this paper we will present a number of the known methods of anti-debugging in a fashion that is easy to implement for a developer of moderate expertise. We will include source code, whenever possible, with a line by line explanation of how the anti-debugging technique operates. The goal of the paper is to educate development teams on anti-debugging methods and to ease the burden of implementation.**

*Keywords*— **anti-debugging, security, debugging, copy protection, anti-piracy, reverse engineering.**

## I. Introduction

Anti-debugging, when implemented properly, can be a significant deterrence to would be reverse engineers and software pirates. There is no foolproof solution to thwart the dedicated reverse engineer; however, making the task as arduous and difficult as possible increases the time and expertise required for full analysis of the binary application.

Application developers should not be required to spend significant amounts of time understanding and examining the specifics of a software protection scheme. Straight forward implementation of a best of breed solution helps to achieve the aforementioned goals while leaving the developer additional time to implement features and other necessary application components.

The majority of data on the topic of anti-debugging has been presented from the vantage point of a reverse engineer. Anti-debugging methods typically have been presented in assembly language dumps with minimal explanation as to the high level code constructs involved in the technique. Unless the developer is adept at reading and comprehending assembly language code, the anti-debugging method is incomprehensible and thus will not be implemented.

The goal of this paper is to present a number of anti-debugging methods in an easy to comprehend manner. The average developer should be able to read this paper, grasp the concepts described, and readily use the source code provided to implement a myriad of different anti-debugging methods. Education of the developer will lead to a stronger understanding of the basic anti-debugging methods that can be used to limit the effectiveness of a reverse engineer's primary tool, the debugger.

## II. Background Terms and Definitions

The definition of debugging is the act of detecting and removing bugs in an application. Bugs exist in code due to syntactic or logic errors that make the program operate differently than intended. The vast majority of times, debugging is done with obvious intentions. The intent is typically to pinpoint the exact location that is causing an error in the program.

Over time, debugging has become an overloaded term, taking on additional meanings. In the reverse engineering specialty of information security, debugging has come to mean the act of using a debugging tool on a target process to determine exactly what or how a piece of code operates. Debugging is especially useful in the area of malware analysis where a thorough understanding of how a piece of malicious code operates can help to develop strategies for detection and eradication.

Finally, practitioners in the area of software piracy must be skilled at the nuances of reverse engineering and debugging. When popular new software is released, it is immediately attacked by reverse engineers in an attempt to remove any copy protection that may have been put in place by the development team. While there is no way to completely protect software from a skilled reverse engineer, it is possible to layer on defenses that can fill the road with potholes and make the trip to cracking your software a much bumpier ride.

It is with these concepts in mind that we will discuss anti-debugging. Anti-debugging is an implementation of one or more techniques, within the code and thus compiled binary file, which hinders the debugging or reverse engineering process. These anti-debugging methods typically fall into one of six major categories: API based anti-debugging, exception based anti-debugging, direct process and thread detections, modified code detection, hardware and register based detection, and timing based anti-debugging.

*We will not be describing anti-dumping via on-disk PE modification techniques and will only focus on the Microsoft Windows operating system in an attempt to limit the scope and length of this paper. These additional areas of research may be covered in a supplemental paper at a later date.*

## III. API Based Anti-Debugging

API based detections use Win32 function calls to detect the presence of a debugger and act accordingly. Specifically, these detection mechanisms do not directly access memory regions or index sections of memory, instead relying upon the functionality of both documented and undocumented Microsoft API function calls. The majority of the time, the presented API based detection mechanisms will rely upon underlying operating system calls to directly access memory and as such there may be overlap between the methods outlined in this section and other methods detailed later in the document.

### A. *IsDebuggerPresent*

The first anti-debugging method that most new reverse engineers discover is the Microsoft API call, IsDebuggerPresent. This function call analyses the running process environment block (PEB) and looks at the DebuggerPresent flag. The function returns the value located at this flag. If the return value is zero, there is no debugger present; however, if the value returned is non-zero, a debugger is attached to our process.

```
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);
}
```

Appendix: isDebuggerPresent.sln

### B. *CheckRemoteDebuggerPresent*

Conceptually similar to the IsDebuggerPresent method, the CheckRemoteDebuggerPresent function checks the PEB block of the target process for the BeingDebugged flag. The CheckRemoteDebuggerPresent API call takes two parameters, the first of which is a handle to the target process, and the second being the return value indicating if the target process has a debugger attached. This API call requires Windows XP service pack one or later to be installed.

```
CheckRemoteDebuggerPresent(GetCurrentProcess(),
&pbIsPresent);

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

Appendix: CheckRemoteDebuggerPresent.sln

### C. *OutputDebugString*

The OutputDebugString anti-debugging method requires Microsoft Windows 2000 or newer to operate. The OutputDebugString function is sensitive to whether a debugger is attached to the running process and will return an error code if our process is not currently running under a debugger. To detect the presence of a debugger we can make a call to SetLastError() with an arbitrary value, followed by a call to OutputDebugString(). If the arbitrary value remains when we check GetLastError() then we know that the OutputDebugString() was successful and the process is being debugged.

```
DWORD Val = 666;
SetLastError(Val);
OutputDebugString(L"anything");
if (GetLastError() == Val) {
   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);
}
```

Appendix: OutputDebugString.sln

### D. *FindWindow*

The FindWindow detection method is different in that it does not specifically detect the presence of a debugger attached to our process; instead it retrieves a handle to the top-level window whose class name matches a specified string. The common debuggers can be checked for by executing FindWindow with their class name as the parameter. The debugger WinDbg can be detected by calling FindWindow with a class name parameter of “WinDbgFrameClass”. In the code example below, a FindWindow call is made passing the string OLLYDBG, and the return handle is evaluated.

```
Hnd = FindWindow(L”OLLYDBG”, 0);
if (hnd == 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);
}
```

Appendix: FindWindow.sln

### E. *Registry Key*

Searching through the registry is another method we can used to detect the presence of a debugger. This method does not detect the attaching of a debugger to a target process, nor does it even indicate that a particular debugger is running. Instead this method simply indicates to the program that debugger is installed on the system. Since this technique has limited effectiveness, one should only use it as a supporting piece of information when deciding how to act upon other, more definitive detection mechanisms. There are three registry keys that can be used to indicate the installation of a debugger on the system. If either of the first two keys exists, OllyDbg has been configured as a shell extension to open target files by right clicking them.

```
HKEY_CLASSES_ROOT\dllfile\shell\Open with
Olly&Dbg\command
HKEY_CLASSES_ROOT\exefile\shell\Open with
Olly&Dbg\command
```

If the final key has been set, the value of the Debugger name/value pair represents the debugger that has been configured as the just in time debugger for the system. In the event of a program crash, this is the debugger that will be called. Visual studio is represented as vsjitdebugger.exe while OllyDbg will be OLLYDBG.EXE.

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows
NT\CurrentVersion\AeDebug Debugger=
```

Appendix: Registry.sln

### *F. NtQueryInformationProcess (ProcessDebugPort)*

The NtQueryInformationProcess function is located within ntdll.dll. This undocumented call is used to retrieve information about a target process, including but not limited to, debugging related data. The function takes five parameters, the first two of which are of the most interest. The first parameter is the handle of the process to interrogate. In our example we use the value -1. This value tells the function to use the current running process. The second parameter is a constant value indicating the type of data we wish to retrieve from the target process. A call to this function using a handle to our currently running process, along with a ProcessInformationClass of ProcessDebugPort (0x07) will return the debugging port that is available. If the target process is currently being debugged, a port will already be assigned and that port number will be returned. If the process does not have a debugger attached, a zero value will return indicating that no debugger is currently attached. Since NtQueryInformationProcess is intended to be internal to the operating system, we have to use run time dynamic linking to be able to call this functionality. This is achieved by calling LoadLibrary and GetProcAddress and then executing the returned function pointer.

```
hmod = LoadLibrary(L"ntdll.dll");
_NtQueryInformationProcess = GetProcAddress(hmod,
"NtQueryInformationProcess");

status = (_NtQueryInformationProcess) (-1, 0x07,
&retVal, 4, NULL);

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);
}
```

Appendix: NtQProcessDebugPort.sln

### *G. NtQueryInformationProcess (ProcessDebugFlags)*

The ProcessDebugFlags check also uses the NtQueryInformationProcess function call to detect a debugger. Instead of calling the function with a second parameter of 0x07 (ProcessDebugPort), we submit the call with a second parameter value of 0x1f (ProcessDebugFlags). When calling the function with this constant we are returned a value indicative of the debug flags that are present on the target thread. The function returns the inverse of the NoDebugInherit value, which means that a return of 0 indicates that a debugger is currently attached to the process.

```
hmod = LoadLibrary(L"ntdll.dll");
_NtQueryInformationProcess = GetProcAddress(hmod,
"NtQueryInformationProcess");
status = (_NtQueryInformationProcess) (-1, 31,
&debugFlag, 4, NULL); // 31 is the enum for
DebugProcessFlags and = 0x1f in hex

if (debugFlag == 0x00000000) MessageBox(NULL,
L"Debugger Detected via ProcessDebugFlags",
L"Debugger Detected", MB_OK);

if (debugFlag == 0x00000001) MessageBox(NULL, L"No
Debugger Detected", L"No Debugger", MB_OK);
```

Appendix: ProcessDebugFlags.sln

### *H. NtSetInformationThread Debugger Detaching*

The NtSetInformationThread call is a wrapper around the ZwQueryInformationProcess system call. This function takes four parameters. Of interest to us for anti-debugging purposes are the first and second parameters, which contain the target handle and the ThreadInformationClass constant. By setting this constant to 0x11 (ThreadHideFromDebugger) and submitting the call we can disconnect a debugger from our running process.

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

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

MessageBox(NULL, L"Debugger Detached", L"Debugger
Detached", MB_OK);
```

Appendix: NtSetInformationThread-Detach.sln

### *I. Self Debugging with DebugActiveProcess*

One possible way that a process can determine if it is being run under a debugger is to attempt to debug itself. A process can only have one debugger attached to it at a time thus if a process is successful at debugging itself, then it can be sure that no other debugging tool is currently attached. To achieve this goal, a process must first create a child process. This child process can determine the process ID of its parent using any of a number of methods, at which point it will attempt to call DebugActiveProcess with its parent's process ID as the target parameter. If this is successful we can be sure that the parent process is not currently being debugged. In our example code we spawn a child process of ourselves and pass in the process ID of the parent via command line arguments. The new process then checks for a debugger on the parent and if no debugger exists would continue to execute.

```
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);
```

Appendix: Self-Debugging.sln

### J. NtQueryInformationProcess (ProcessDebugObjectHandle)

With the release of Windows XP, the debugging model was modified to create a handle to a debug object when a process is being debugged. It is possible to detect the existence of this handle by calling NtQueryInformationProcess with a second parameter of 0x1e. The 0x1e constant represents the value for ProcessDebugObjectHandle. If this function call returns a non-zero value we can be sure that the target process is being debugged and can act accordingly.

```
hmod = LoadLibrary(L"ntdll.dll");
_NtQueryInformationProcess = GetProcAddress(hmod,
"NtQueryInformationProcess");

status = (_NtQueryInformationProcess) (-1, 0x1e,
&hDebugObject, 4, NULL); // 0x1e is the enum for
ProcessDebugObjectHandle

if (hDebugObject) MessageBox(NULL, L"Debugger
Detected via ProcessDebugFlags", L"Debugger
Detected", MB_OK);
if (!hDebugObject) MessageBox(NULL, L"No Debugger
Detected", L"No Debugger", MB_OK);
```

Appendix: ProcessDebugObjectHandle.sln

### K. OllyDbg OutputDebugString() Format String

An interesting bug exists in the current version of OllyDbg[1]. This particular debugger has a format string vulnerability within its handling of the OutputDebugString() function. When OllyDbg is attached to a process that calls OutputDebugString with a parameter of %s, the debugger will crash. To execute this vulnerability and anti-debugging method we simply make a call to OutputDebugString(TEXT(“%s%s%s%s%s%s%s”) within a structured exception handler. We safely handle the exception that is thrown within our code while simultaneously crashing any attached OllyDbg instance.

```
__try {


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

Appendix: OllyDbgOutputDBString.sln

### L. SeDebugPrivilege OpenProcess

When a process is run in or attached to by a debugger, the SeDebugPrivilege token is given to the target process. Some debuggers properly remove that permission and revert the process back to its original privilege state, while other debuggers fail to complete this step. If our process is being debugged by a debugger that does not properly revoke this privilege we can use this information to determine the debugger's existence.

To check for the existence of this privilege set the process simply tries to open a process such as csrss.exe with PROCESS_ALL_ACCESS rights. Normally our process would not be allowed to execute an OpenProcess() call with this as our target and the PROCESS_ALL_ACCESS rights; however, with the elevated privilege set granted by the debugger, we are able to open this file.

The first step in this process is determining the process identifier for the csrss.exe process. This can be achieved a number of ways. In our sample code we use an undocumented function within ntdll called CsrGetProcessId(). By dynamically loading the ntdll.dll library and then finding the function address for this call we can execute this function. This function returns the PID for the running csrss.exe process.

```
hmod = LoadLibrary(L"ntdll.dll");
_CsrGetProcessId = GetProcAddress(hmod,
(LPCSTR)"CsrGetProcessId");

pid = (_CsrGetProcessId)();
```

After we have the PID for the target process we attempt to open this process using a call to OpenProcess() and a permission set of PROCESS_ALL_ACCESS. If the result of this function call is successful we know that we are running at an elevated privilege level and most likely have a debugger attached.

```
HANDLE Csrss = 0;

Csrss = OpenProcess(PROCESS_ALL_ACCESS, FALSE, pid);

if (Csrss == 0) {
  printf("Result is 0 Error Code: %d\n",
GetLastError());
  MessageBox(NULL, L"No Debugger Detected or
Debugger Does Not Assign SeDebugPrivilege", L"No
Debugger", MB_OK);
}
if (Csrss != 0) MessageBox(NULL, L"Debugger Detected
via OpenProcess SeDebugPrivilege", L"Debugger
Detected", MB_OK);
```

Appendix: SeDebugPriv.sln

### M. OllyDbg OpenProcess String Detection

All versions of OllyDbg contain a static DWORD at a static offset within the running process. This is most likely implemented to ensure that OllyDbg does not attempt to debug itself by mistake. We can use this information to detect running instances of OllyDbg. We begin this process by walking the process list checking all images for the DWORD 0X594C4C4F at offset 0x4B064B from the main thread base.

There are numerous methods to list and analyze process on a system. We chose to gain access to the process list via the psapi library. This library can be included in the Visual Studio project by adding a dependency for psapi.lib. Using a call to

[1] At the time of authoring the current version of OllyDbg is v1.10.

EnumProcesses within this library we are able to enumerate all processes currently running on the system. We then pass each process in turn to the checkProcess() function.

```
if ( !EnumProcesses( aProcesses, sizeof(aProcesses),
&cbNeeded ) ) return;

cProcesses = cbNeeded / sizeof(DWORD);
for (i = 0; i < cProcesses; i++) {
  flag = checkProcess( aProcesses[i] );
  if (flag == 1) { break; }
}

if (flag == 0) {
  MessageBox(NULL, L"No Debugger Detected", L"No
Debugger Detected", MB_OK);
}
```

Appendix: OllyDbgStaticString.sln

The checkProcess function opens the target process and checks the static offset for the DWORD that indicates a running OllyDbg process. If the value exists it returns one otherwise it returns zero.

```
HMODULE hMod = NULL;

HANDLE hProcess = OpenProcess(
PROCESS_QUERY_INFORMATION | PROCESS_VM_READ, FALSE,
processID );

if ( hProcess != NULL ) {
  if (ReadProcessMemory(hProcess, (LPCVOID)0x4B064B,
&value, (SIZE_T) 4, (SIZE_T *)&read)) {
    if (value == 0x594C4C4F) {
      MessageBox(NULL, L"Debugger Detected via
Static OpenProcessString()", L"Debugger Detected",
MB_OK);
      return 1;
    }
  }
}
return 0;
```

Appendix: OllyDbgStaticString.sln

### *N. OllyDbg Filename Format String*

OllyDbg also contains a flaw in the way it parses the name of the process it is attempting to debug. The parsing function contains a format string vulnerability that can cause OllyDbg to crash. By naming our process with a %s character we can cause OllyDbg to be unable to load and parse our file. While this is a simple method to implement, it is also trivial to bypass. An attacker can simply rename the file before loading it into the debugger. To combat this issue the process should programmatically check its own process name and ensure that it is the original file name that contains the format string character.

To achieve the name check we use the function GetModuleFileName() to get access to our current running process name and then compare this against our expected value using wcsncmp().

```
GetModuleFileName(0, (LPWCH) &pathname, 512);
filename = wcsrchr(pathname, L'\\');

if (wcsncmp(filename, L"\\%s%s.exe", 9) == 0) {
  MessageBox(NULL, L"No Debugger Detected - Original
Name Found", L"No Debugger Detected", MB_OK);
} else {
  MessageBox(NULL, L"Debugger Detected - File Name
Modification Occured", L"Debugger Detected", MB_OK);
}
```

## IV. Direct Process and Thread Block Detections

When direct API calls are insufficient or an industrious debugger is hooking the calls and returning falsified data, the anti-debugging effort must go lower than the intervening methods and directly query the process and thread block information. Much of the results from the API models above can be retrieved by directly accessing details about the running process. Additionally, the lower and closer to the operating system our anti-debugging effort resides, the more difficult it will be to bypass.

### *A. IsDebuggerPresent Direct PEB*

As described in section III. A., a basic anti-debugging technique is to use the Microsoft API IsDebuggerPresent to check for the existence of the BeingDebugged byte within the process environment block (PEB). Please refer to the reference section for a detailed listing of the PEB structure. A similar method is to bypass the API call and directly access the details of the running process block via the process environment headers. When a process is executed, a copy of the executable code as well as all associated header information is stored in memory. This header information can be queried directly, without the help of an API, to verify the value within the BeingDebugged byte.

Multiple methods exist to access the information stored within the PEB. The easiest way is to use built in Microsoft API calls to retrieve a pointer to the data stored within the PEB. A call to NtQueryInformationProcess with a second parameter of ProcessBasicInformation will return a pointer to the process information block (PIB) structure for the target process. Again please refer to the references section for a link to a detailed listing of the PIB structure. Once we have a pointer to the PIB structure we reference the PebBaseAddress member of the PIB structure to gain a pointer to the PEB structure. Finally the BeingDebugged member of the PEB structure is compared against a value of one to determine if we are running within a debugger.

```
hmod = LoadLibrary(L"Ntdll.dll");
_NtQueryInformationProcess = GetProcAddress(hmod,
"NtQueryInformationProcess");

hnd = OpenProcess(PROCESS_QUERY_INFORMATION, FALSE,
GetCurrentProcessId());
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);
  }
}
```

Appendix: IsDebuggerPresent-DirectPEB.sln

### *B. IsDebuggerPresent Set/Check*

As noted in the IsDebuggerPresent method, the process environment block holds a byte that indicates the debugging status of the running process. This byte, called the BeingDebugged byte, contains a non-zero value if the target process is being debugged. Many debuggers and anti-debugging detection plugins will hook the IsDebuggerPresent API call and always return a zero value to the requesting process. In this fashion a debugger can hide from the target process IsDebuggerPresent calls. It is possible to detect these hooks by setting the BeingDebugged byte to an arbitrary value and then issuing an IsDebuggerPresent function call. If the arbitrary value that was previously set is returned from IsDebuggerPresent, then the function is not hooked, however if we are returned a zero, we know that the process is running under a debugger that is attempting to hide its existence.

```
hnd = OpenProcess(PROCESS_QUERY_INFORMATION, FALSE,
GetCurrentProcessId());
status = (_NtQueryInformationProcess) (hnd,
ProcessBasicInformation, &pPIB,
sizeof(PROCESS_BASIC_INFORMATION), &bytesWritten);

if (status == 0) {
  pPIB.PebBaseAddress->BeingDebugged = 0x90;
} // Sets the BeingDebugged Flag to arbitrary value

retVal = IsDebuggerPresent(); // Retrieve value
if (retVal != 0x90 ) {
  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);
}
```

Appendix: IsDebuggerPresent-SetCheck.sln

### *C. NtGlobalFlag*

Processes, when started under a debugger, run slightly differently than those started without a debugger attached. In particular, debugged processes create memory heaps in a different fashion than those not being debugged. The information that informs the kernel how to create heap structures is stored within the PEB structure at offset 0x68. When a process is started from within a debugger, the flags FLG_HEAP_ENABLE_TAIL_CHECK (0x10), FLG_HEAP_ENABLE_FREE_CHECK (0x20), and FLG_HEAP_VALIDATE_PARAMETERS (0x40) are set for the process. We can use this information to determine if our process was started from within a debugging tool by referencing the value located at offset 0x68 within the PEB structure. If the bits corresponding to 0x70 (the sum of the above flags) are set then we can be certain of our debugging status.

```
hmod = LoadLibrary(L"Ntdll.dll");
_NtQueryInformationProcess = GetProcAddress(hmod,
"NtQueryInformationProcess");

hnd = OpenProcess(PROCESS_QUERY_INFORMATION, FALSE,
GetCurrentProcessId());
status = (_NtQueryInformationProcess) (hnd,
ProcessBasicInformation, &pPIB,
sizeof(PROCESS_BASIC_INFORMATION), &bytesWritten);

value = (pPIB.PebBaseAddress);
value = value+0x68;
printf("FLAG DWORD: %08X\n", *value);

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);
}
```

Appendix: NtGlobalFlag.sln

### *D. Vista TEB System DLL Pointer*

This anti-debugging technique is unique to Microsoft Windows Vista. When a process is started without a debugger present, the main thread environment block (TEB) contains a pointer to a Unicode string referencing a system DLL such as kernel32.dll. The pointer is located at offset 0xBFC in the TEB structure and is followed directly by the Unicode string at offset 0xC00. If the process is started under a debugger, that system DLL name is replaced with the Unicode string "HookSwitchHookEnabledEvent".

To use this technique, the anti-debugging function should first check that it is running on the Windows Vista operating system. After confirming the operating system revision, the technique should locate the thread information block (TIB) by using the following code:

```
void* getTib()
{
  void *pTib;
  __asm {
    mov EAX, FS:[18h] //FS:[18h] is the location of
the TIB
    mov [pTib], EAX
  }
  return pTib;
}
```

Once the location of the TIB is found, the offset 0xBFC is read and the pointer checked. If this value is 0x00000C00 we then read the string at offset 0xC00 and compare this value to the Unicode string "HookSwitchHookEnabledEvent". We check the pointer to ensure that we have a string located in the pointed to address and as a second level of assurance for the accuracy of this method. If we pass this final test we can be sure that our process was started from within a debugger.

```
wchar_t *hookStr =
_TEXT("HookSwitchHookEnabledEvent");

strPtr = TIB+0xBFC;

delta = (int)(*strPtr) - (int)strPtr;
if (delta == 0x04) {
   if (wcscmp(*strPtr, hookStr)==0) {
      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);
}
```

Appendix: Vista-TEB-SystemDLL.sln

### E. *PEB ProcessHeap Flag Debugger*

There are additional flags within the PEB structure that are used to indicate to the kernel how heap memory should be allocated. In addition to the NtGlobalFlag, the ProcessHeap Flag can also be used to determine if a process was started with a debugger attached. By dereferencing a pointer to the first heap located at offset 0x18 in the PEB structure, we can then locate the heap flags which are located at offset 0x10 within the heap header. These heap flags are used to indicate to the kernel that the heap was created within a debugger. If the value at this location is non-zero then the heap was created within a debugger.

```
base = (char *)pPIB.PebBaseAddress;
procHeap = base+0x18;
procHeap = *procHeap;
heapFlag = (char*) procHeap+0x10;
last = (DWORD*) heapFlag;

if (*heapFlag != 0x00) {
  MessageBox(NULL, L"Debugger Detected Using PEB
Process Heap Flag", L"Debugger Detected", MB_OK);
  } else {
  MessageBox(NULL, L"No Debugger Detected", L"No
Debugger Detected", MB_OK);
}
```

Appendix: ProcessHeap.sln

### F. *LDR_Module*

As described in the NtGlobalFlag and PEB ProcessHeap Flag anti-debugging methods, processes that are created from within a debugger use a modified heap creation algorithm when compared to those processes created without a debugger attached. When allocating a heap, while under a debugger, the DWORD 0xFEEEFEEE is created within the memory segment. This occurs because when heaps are created under a debugger, an alternate heap creation algorithm is used that allows for heap corruption detection and heap validation to occur. This value can be used as a signature to determine if a heap, and therefore a process, was created while under the control of a debugger. The easiest method to analyze a heap within a process is to look at the LDR_Module that is pointed to by the PEB. The LDR_Module contains information regarding the modules loaded by the binary. This module is created inside of a heap when the process is started and should exhibit the debugging DWORD if the process was created within a debugger. Below is a memory dump of the end of the LDR_Module in our sample code. The 0xFEEEFEEE DWORD is repeated four times at the end of the heap block.

```
00252e98  abababab abababab 00000000 00000000
00252ea8  000d062b 00ee14ee 00250178 00250178
00252eb8  feeefeee feeefeee feeefeee feeefeee
```

To analyze the LDR_Module for this signature string we locate the start of the LDR_Module by first accessing the PEB base address and then looking at offset 0x0C from the start of the PEB. The DWORD at 0x0C will be a pointer to the LDR_Module for our process.

```
hmod = LoadLibrary(L"Ntdll.dll");
_NtQueryInformationProcess = GetProcAddress(hmod,
"NtQueryInformationProcess");

hnd = OpenProcess(PROCESS_QUERY_INFORMATION, FALSE,
GetCurrentProcessId());
status = (_NtQueryInformationProcess) (hnd,
ProcessBasicInformation, &pPIB,
sizeof(PROCESS_BASIC_INFORMATION), &bytesWritten);

base = (pPIB.PebBaseAddress);
ldr_module = base+0x0c;
ldr_module = *ldr_module;
```

After we have acquired the address of the LDR_Module we scan for the DWORD 0xFEEEFEEE by walking memory one byte at a time. We continue to scan until we have either found the signature string or we trigger an exception. If we find the signature we know the process was created with a debugger attached and if we trigger an exception, we have not found the signature and have walked past our allowable memory read section. We can safely handle the exception and continue execution of the program. When conducting the scan, we actually compare the data pointed to by our variable against the DWORD 0xEEFEEEFE to account for the little endian byte ordering of our x86 system.

```
walk = ldr_module;

__try {
  while (*ldr_module != 0xEEFEEEFE) {
  printf("Value at pointer: %08X\n", *ldr_module);
    walk = walk +0x01; // walk is a byte
    ldr_module = walk;
   }
 }
__except (EXCEPTION_EXECUTE_HANDLER) {
  flag = 1;
  MessageBox(NULL, L"Debugger Not Detected", L"No
Debugger Detected", MB_OK);
}

if (flag == 0) MessageBox(NULL, L"Debugger Detected
via LDR_MODULE", L"Debugger Detected", MB_OK);
```

Appendix: LDR_Module.sln

## V. HARDWARE AND REGISTER BASED DETECTION

Hardware and register based detections differ from API and direct process and thread block detections in that the information that indicates the existence of a debugger is stored within the processors registers on the physical hardware itself. Instead of relying upon software discrepancies to indicate the existence of a debugger, one can directly query the hardware for the required information.

### A. *Hardware Breakpoints*

A breakpoint is a signal that tells the debugger to cease operation of a process at a certain point. Debuggers can create breakpoints in target applications in multiple ways.

Software breakpoints and hardware breakpoints facilitate similar results but operate differently. When a software breakpoint is inserted into a process, the debugger reads the instruction at the breakpoint location, removes the first byte of this instruction, and replaces it with a breakpoint opcode. The original byte is saved in a table and replaced when the breakpoint occurs, thus allowing the execution to continue.

Hardware breakpoints are implemented in the processor hardware itself. The hardware contains registers dedicated to the detection of specific addresses on the program address bus. When the address on the bus matches those stored in the debug registers, a breakpoint signal, interrupt one (INT 1), is sent and the CPU halts the process. There are eight debug registers on the x86 architecture referred to as DR0 through DR7. Registers DR0 through DR3 contain the addresses on which we wish to break while DR7 contains bits to enable or disable each of the DR0 through DR3 breakpoints. DR6 is used as a status register to permit a debugger to know which debug register has triggered.

A call to GetCurrentThreadContext is used to read the debug register information from the chip. We then compare the value in the registers to 0x00 to ensure that no hardware breakpoints are currently set in our process.

```
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);
}
```

Appendix: HardwareBreakpoints.sln

### B. VMware LDT Register Detection

Recent research into virtual machine monitors and in particular VMware has yielded detection mechanisms that allow code running in the guest operating system to determine if it is running within a virtual machine or native on the system. While these methods are not specific to anti-debugging, they are pertinent to the discussion due to the fact that a large portion of reverse engineering takes place within an emulated environment for security purposes, specifically malware related debugging and reverse engineering. Anti-debugging code can be designed to only allow execution within a native operating system, or at a minimum, to help determine a heuristic value indicative of a debugging or reverse engineering session.

Virtual machines must emulate certain hardware resources in order for guest operating systems to boot and function as expected. This includes emulating registers and other processor specific components within the virtual machine monitor. Of interest to the anti-debugging world is the fact that VMware, and presumably other virtual machine monitors, must emulate specific registers that can be read from an unprivileged level. The Interrupt Descriptor Table Register (IDTR), Global Descriptor Table Register (GDTR), and the Local Descriptor Table Register (LDTR), all exhibit this specific behavior. The data held in these registers is the memory location that is specific to that particular data structure. In other words a pointer to the required data is saved for use within the register. Since the virtual machine monitor must emulate these registers, the data pointers must point to different locations. Based on the returned locations for the interrupt, global, and local descriptor tables we are able to determine if the operating system is running within a virtual machine.

GDTR and IDTR are unreliable sources of virtual machine detection due to the fact that multi-core processors, as well as multi-processor systems, will return different values depending on which of the processors is queried. The LDTR register, however, remains static across all cores and processors and is a reliable method of virtual machine detection.

To determine if we are within a virtual machine we use the assembly function *sldt* to store the value of the LDTR into a variable. We then check the first two bytes of the return value and compare them against 0x00. If the host is running as a native operating system, these two bytes will always be 0x00. If the host is running under a virtual environment, these two bytes will differ as the local descriptor table is placed into a different memory location.

```
unsigned char ldt_info[6];
int ldt_flag = 0;

__asm {
  sldt ldt_info;
}

if ((ldt_info[0] != 0x00) && (ldt_info[1] != 0x00))
ldt_flag = 1;

if (ldt_flag == 1) MessageBox(NULL, L"Vmware
Detected via ldt", L"Debugger Detected", MB_OK);
if (ldt_flag == 0) MessageBox(NULL, L"No VMware
Detected", L"No Vmware", MB_OK);
```

Appendix: VMWareSLDT.sln

### C. VMware STR Register Detection

Very similar to the LDT Register method of VMware detection is the STR method. This method uses the store task register (STR) to detect the existence of a virtual operating system. The STR, much like the LDT, GDT, and IDT contains a pointer to a piece of data that is specific to the process running on the hardware. Out of necessity, the virtual operating environment must relocate this information so that it does not overwrite the native system data. Once again this information can be retrieved from an unprivileged level. We can query the location of the stored task data and compare that against the normal expected values when operating in a native environment. When the data is outside of the expected bounds we can be sure that we are operating within a virtual environment.

To execute this detection mechanism we make an assembly call to the *str* mnemonic and store the resultant data within the variable *mem*. If the first byte is equal to 0x00 and the second byte equal to 0x40, we can be confident that we are running within a VMware instance.

```
unsigned char  mem[4] = {0, 0, 0, 0};

__asm str mem;

printf ("\n[+] Test 4: STR\n");
printf ("STR base: 0x%02x%02x%02x%02x\n", mem[0],
mem[1], mem[2], mem[3]);

if ((mem[0] == 0x00) && (mem[1] == 0x40))
  MessageBox(NULL, L"VMware Detected", L"VMWare
Detected", MB_OK);
else
  MessageBox(NULL, L"No VMWare Detected", L"No
VMWare Detected", MB_OK);
```

Appendix: VMWareSTR.sln

## VI. Timing Based Detections

Another class of anti-debugging is timing based detections. These methods use timing based functions to detect latency in the execution between lines or sections of code. When running code within a debugger it is very common to execute the code in a single step fashion. Single stepping is allowing the debugger to execute a single line (step into) or single function (step over) and then return control to the debugger.

When the debugger is executing the code via single stepping, the latency in execution can be detected by using functions that return time or tick counts to the application. Two time function calls can be made in succession and the delta compared against a typical value. Alternatively, a section or block of code may be flanked by timer calls and again the delta compared against a typical execution time value.

Four of the Microsoft windows primary time functions are used to demonstrate how these detections work. In the examples provided we have simply placed the timer calls directly in succession and compared the return value against a rough estimate of a reasonable latency time.

### A. *RDTSC*

The time stamp counter is a 64 bit register that is part of all x86 processors since the creation of the original Intel Pentium. This register contains the number of processor ticks since the system was last restarted.  The x86 assembly language Opcode RDTSC was formally introduced with the Pentium II and was undocumented until then. To access this value from C code we use the function __rdtsc. Our example takes the results of the two calls to __rdtsc and compares the delta to a constant value of 0xff. This is an arbitrary value that returned results with a high level of accuracy when used to detect single step debugging.

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

Appendix: RDTSC.sln

### B. *NTQueryPerformanceCounter*

Modern processors also include hardware performance counters. These performance counters are registers that store counts of hardware related activities within the processor. The value of the hardware performance can be queried using the function QueryPerformanceCounter. We use a nearly identical technique to other timing methods and compute a delta of two calls to QueryPerformanceCounter. If this delta is within a reasonable threshold amount our process is not running within a debugger in single step mode. Again we use an arbitrary value of 0xff for our latency threshold.

```
QueryPerformanceCounter(&li);
QueryPerformanceCounter(&li2);

if ((li2.QuadPart-li.QuadPart) > 0xFF) {
  MessageBox(NULL, L"Debugger Detected via
QueryPerformanceCounter", L"Debugger Detected",
MB_OK);
} else {
  MessageBox(NULL, L"No Debugger Detected", L"No
Debugger", MB_OK);
}
```

Appendix: NTQueryPerformanceCounter.sln

### C. *GetTickCount*

The GetTickCount API functions provided by kernel32.dll return the number of milliseconds that have elapsed since the system was last restarted. This value wraps at 49.7 days. The primary difference between this and other timing methods is that due to the return value being in milliseconds, the threshold value is much lower. In our example case we set the threshold to 0x10 in order to detect single step debugging efforts.

```
li = GetTickCount();
li2 = GetTickCount();

if ((li2-li) > 0x10) {
  MessageBox(NULL, L"Debugger Detected via
QueryPerformanceCounter", L"Debugger Detected",
MB_OK);
} else {
  MessageBox(NULL, L"No Debugger Detected", L"No
Debugger", MB_OK);
}
```

Appendix: GetTickCount.sln

### D. *timeGetTime*

Like GetTickCount, the timeGetTime function call returns a time value in milliseconds. In the case of timeGetTime we are returned the system time as opposed to the elapsed time since the last system restart. The timeGetTime function is part of the WinMM library and must be added to a Visual Studio project as a dependency. Again we use the same method of comparing a delta to a reasonable threshold to detect single stepping of our process.

```
li = timeGetTime();
li2 = timeGetTime();

if ((li2-li) > 0x10) {
  MessageBox(NULL, L"Debugger Detected via
QueryPerformanceCounter", L"Debugger Detected",
MB_OK);
} else {
  MessageBox(NULL, L"No Debugger Detected", L"No
Debugger", MB_OK);
}
```

## VII. Modified Code Detection

Self referencing code can be used to determine if modifications to our process have been made. When a software based breakpoint is place into a process, a byte of the operating code is overwritten with a special instruction that causes the processor to trigger a break point exception that is then trapped and handled by the debugger. By using self referencing code to scan for instruction modifications, it is possible to check for instances of software breakpoints on a running process.

### A. CRC Checking

The CRC checking method of anti-debugging uses techniques similar to self modifying code; however instead of modifying our code we simply take a CRC or hash value of the target code block to ensure that no changes have taken place. Using this method we can ensure that no software breakpoints have been placed into our tested code segment at the time of the CRC computation.

The CRC checking method is typically implemented via a CRC function that takes as inputs pointers to locations within memory. The first parameter is the address of the start of memory, usually a function pointer, and the second address is normally the range of memory to compute the CRC value for.

In our example, the memory range is computed by subtracting the address of the function following our target function from the address of our target function itself. The result of this computation is the length of the function we wish to determine a CRC value for. The line *"original_crc = CRCCCITT((unsigned char*) &antidebug, (DWORD)&runmycode-(DWORD)&antidebug, 0xffff, 0);*" shows a call to the CRC function with a start address of the function antidebug and a length computation resulting from the delta between the address of runmycode() and antidebug(). Alternative hash or CRC algorithms could be used in place of the CRCCCITT one chosen for our example.

```
void antidebug(int pass)
{
  printf("Location of runmycode = %08X and antidebug
= %08X\n", &runmycode, &antidebug);
  if (pass == 1) {
    original_crc = CRCCCITT((unsigned char*)
&antidebug, (DWORD)&runmycode-(DWORD)&antidebug,
0xffff, 0);
  } else {
    the_crc = CRCCCITT((unsigned char*) &antidebug,
(DWORD)&runmycode-(DWORD)&antidebug, 0xffff, 0);
  }
  return;
}

void runmycode()
{
  if (the_crc != original_crc) {
    MessageBox(NULL, L"Debugger Detected via CRC",
L"Debugger Detected", MB_OK);
  } else {
    MessageBox(NULL, L"No Debugger Detected", L"No
Debugger Detected", MB_OK);
  }
  return;
}
```

Appendix: CRCScan.sln

Once we have a working function to determine the CRC of a target code region, we can run the method and compare the results against an expected hardcoded CRC value. Alternatively, our anti-debugging check can run the CRC scan at multiple times throughout the program and compare the results against the previous executions. If the CRC value changes there is a high probablity that a software breakpoint has been inserted into the target function or code range.

One point of note with regard to self referencing code is to be sure that your Visual Studio project has the incremental linking option disabled. Self referencial code using direct function addressing, as in our model, will not work properly when incrementally linked due to the use of "jump thunks" that redirect program flow using a jump table to the actual function code. Incremental linking uses a table of pointers to reference functions and will cause the function as we have written in our proof of concept to fail.

## VIII. Exception Based Detection

Exception handling based detection relies upon the fact that a debugger will trap certain exceptions and not properly pass them on to the process for internal consumption. In some debuggers, there may be a configurable option to ignore or otherwise return the exception handling to the process, but many times this is not enabled by default. If the debugger doesn't pass the exception back to the process properly this can be detected within the process exception handling mechanism inside of the process and acted upon.

Microsoft Windows uses a chain of exception handlers designed to trap exceptions before they cause a fatal crash of the process or operating system. The exception chain consists of one more vectored exception handlers, followed by the structured exception handler chain, and finally the unhandled exception filter is implemented to catch any exceptions that have fallen through the other methods. The following figure, created by Joe Jackson [3], graphically depicts the flow of exceptions.

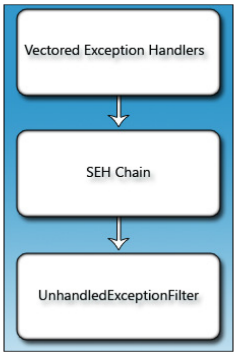

Figure 1: Exception Handling Chain [3]

### A. *INT 3 Exception (0XCC)*

The basic operation of a general debugger is to use a call to interrupt 3 (INT 3) to trigger a software breakpoint. Hardware breakpoints, as discussed previously use a different interrupt value to generate the breakpoint exception. INT 3 generates a call to trap in the debugger and is triggered by opcode 0xCC within the executing process. When a debugger is attached, the 0xCC execution will cause the debugger to catch the breakpoint and handle the resulting exception. If a debugger is not attached, the exception is passed through to a structured exception handler thus informing the process that no debugger is present.

```
int flag = 0;

__try {
  __asm {
    int 3;
  }
}
__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 Int3", L"Debugger Detected", MB_OK);
```

Appendix:INT3.sln

### B. *INT 2D (Kernel Debugger Interrupt)*

Similar in method to INT3, the INT 2D mnemonic is used to access the kernel debugging system. This call creates an exception record and then raises an exception that is trapped by kernel debuggers. We can use this method from ring 3 as well since the call will eventually filter to a ring 3 debugger if no kernel debugger exists. The code to execute this anti-debugging check is identical to the INT 3 method listed above with the only exception being the int 3 assembly call is replaced with int 2dh. See appendix INT2d.sln for full source code.

### C. *ICE Breakpoint*

Yet another method in which a breakpoint can be triggered and the resultant exception inspected is the ICE Breakpoint method. This method uses an undocumented opcode nicknamed the ICE Breakpoint to cause a break state in the same fashion as the INT 3 and INT 2D methods. The ICE Breakpoint is intended to be used with a specific subset of microprocessors; this method is also identical to the previous interrupt based methods. By executing the opcode 0xF1 we can generate a breakpoint that will be trapped by an attached debugger. Once again, we use a structured exception handler to determine if the exception occurs or is handled by the debugger. See appendix ICEBreak.sln for full source code.

### D. *Single Step Detection*

When a process is executing, it is possible to tell the thread to generate a single step exception (EXCEPTION_SINGLE_STEP) after every executed instruction. This indicator is stored within the trap flag bit of the EFLAGS register. We can read this register by pushing the EFLAGS onto the stack with a pushfd instruction. We then set the trap flag bit using a logical OR instruction. Finally we save the EFLAGS by popping the data off the stack with the popfd instruction. When this series of commands is executed, a single step exception will be generated after each instruction. If a debugger is attached to our process, the debugger will intercept the exception thus skipping our structured exception handler and indicating to our process that a debugger is attached.

```
int flag = 0;

//Set the 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);
```

Appendix: SingleStep.sln

### E. *Unhandled Exception Filter*

When an exception occurs within a process executing on the Microsoft Windows operating system, there is a pre-defined set of exception handlers that may be executed. If none of the exception handlers are configured to accept the incoming exception, the final handler is called the unhandled exception filter. This filter is the catch all for exceptions that do not meet the criteria for any other handling mechanism. Using a Microsoft supplied API, we can change the call back function for the unhandled exception filter to a piece of code of our choosing.

When a debugger is in place, it inserts itself above the unhandled exception filter catching unhandled exceptions prior to the final filter being executed. As such, the debugger will intercept exceptions before our assigned call back function allowing us to determine that a debugger is attached to our process. In our example the exception that we generate crashes the application when run under a debugger due to the fact that the debugger does not handle the exception, instead allowing it to execute. When the same code is run without a debugger attached, the registered unhandled exception filter will catch the exception and safely continue program execution.

The first step in executing this anti-debugging method is to set the call back function for an unhandled exception filter. This call back function is registered by a call to the SetUnhandledExceptionFilter() function as demonstrated below. This is then directly followed by inline assembly code that causes a divide by zero error to trigger.

```
int flag = 0;
SetUnhandledExceptionFilter((LPTOP_LEVEL_EXCEPTION_F
ILTER)exHandler);
__asm {
  xor eax, eax;
  div eax;
}
MessageBox(NULL, L"No Debugger Detected", L"No
Debugger", MB_OK);
```

Our callback function simply handles the exception, resets the default unhandled exception filter and continues execution. The function exHandler() is as follows:

```
LONG WINAPI exHandler(PEXCEPTION_POINTERS
pExecpPointers)
{
// Process will CRASH if a debugger is in place
SetUnhandledExceptionFilter((LPTOP_LEVEL_EXCEPTION_F
ILTER)pExecpPointers->ContextRecord->Eax);
pExecpPointers->ContextRecord->Eip += 2;
return EXCEPTION_CONTINUE_EXECUTION;
}
```

Appendix: UnhandledExceptionFilter.sln

### F. CloseHandle

Another trivial way to create an exception that will be trapped by a debugger is to generate an invalid handle exception. To generate this exception a call to CloseHandle() is executed with an invalid handle object. This call directly executes the syscall ZwClose, which in turn generates the exception. As is the case with most exception based anti-debugging techniques, we encase the call within a __try and __except clause to be sure that our program can safely handle the exception. This particular anti-debugging mechanism is slightly different in that the call to CloseHandle() only raises the exception if a debugger is attached to the process. If a debugger is not attached, the call simply returns an error code and continues on. Due to this fact, if our code reaches the except block, we have detected a debugger.

```
__try {
  CloseHandle((HANDLE)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);
```

Appendix: UnhandledExceptionFilter.sln

### G. Control-C Vectored Exception

Vectored exception handlers are a recent feature addition to the exception handling mechanism in the Microsoft Windows operating system. Vectored exception handlers execute first in the exception handling chain and any number of VEH handlers can be chained together. Vectored exception handlers are explicitly added to your code and do not rely upon the __try and __except blocks. When creating vectored exception handlers, a linked list structure is used, allowing the process to install a theoretically unlimited number of exception handlers between the SEH and the final unhandled exception filter.

When a console mode application is being executed under the control of a debugger, typing control-c will create an exception that can be detected and trapped using vectored exception handling. Normally a console application will create a signal handler to properly handle a call to control-c. If the process is not running under the context of a debugger, this signal handler is executed. By creating both a signal handler and an exception handler, it is possible to determine if the exception or the signal trap is executed. When running under Visual Studio debugger, the exception is thrown and executed within our process.

```
AddVectoredExceptionHandler(1,
(PVECTORED_EXCEPTION_HANDLER)exhandler);

SetConsoleCtrlHandler((PHANDLER_ROUTINE)sighandler,
TRUE);

success = GenerateConsoleCtrlEvent(CTRL_C_EVENT, 0);
```

Appendix: CNTRL-C.sln

In the above lines of code we see three function calls of interest. In the first call we add a vectored exception handler which calls back to our function exhandler(). We then add a signal handler using SetConsoleCtrlHander() that calls back to our function sighandler(). And finally we generate a control-c call by executing GenerateConsoleCtrlEvent() with a first parameter of CTRL_C_EVENT. In the signal handler we simply handle the signal and continue executing, while in the vectored exception handler we take action as if a debugger is attached to our process. The code will operate differently based on the existence of a debugger. If the debugger is attached, the exception handler is triggered first and thus we know the process is being debugged. If the VEH does not fire, this is because no debugger is present and we handle the control-c event with our signal handler and continue execution.

### H. Prefix Handling

An interesting issue occurs when debugging an application that uses inline assembly prefixes. Depending on which debugger is in use, these prefixes may not be properly executed. Some debuggers simply step over the byte following a prefix such as rep (repeat) and never actually execute the next instruction. This occurs in our example when the debugger causes the interrupt to not be executed and thus our exception is never run. In this manner we are able to detect that a debugger is attached to our process and act accordingly. Before implementing this anti-debugging method, one should be sure that their target audience debugger will be detected using this method.

```
int flag = 0;
__try {
  __asm {
    __emit 0xF3; // 0xF3 0x64 is PREFIX REP:
    __emit 0x64;
    __emit 0xF1; // Break that gets skipped if
debugged
  }
}
__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 Prefixing", L"Debugger Detected", MB_OK);
```

Appendix: Prefix.sln

### *I. CMPXCHG8B and LOCK*

The LOCK prefix in assembly is used to assert a special pin on the processor during the execution of the subsequent instruction. This pin is used to ensure that the processor is the only processor with access to a shared memory area. The LOCK prefix is used within multi-processor systems that may be affected by processors simultaneously modifying shared memory segments. There is a small subset of instructions that can legally follow a LOCK prefix.

The CMPXCHG8B instruction is a compare instruction that compares values stored in specific registers with a target memory location. If the destination value matches the source value, the source is moved into the targeted memory location, if not, the destination memory data is loaded into the specific registers.

The CMPXCHG8B and LOCK prefix instructions do not operate properly together. If they are executed in succession an invalid instruction error will be generated. If this code is run under a debugger, the debugger will catch the invalid instruction exception and terminate the running process. However; if no debugger exists, we can trap this exception and continue execution gracefully. To do this we set an unhandled exception filter and then execute the instructions in inline assembly.

```
void error()
{
  MessageBox(NULL, L"No Debugger Detected", L"No
Debugger", MB_OK);
  return;
}

...

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

Appendix: LOCKCMPXCHG8B.sln

### *J. OllyDbg Memory Breakpoint*

The OllyDbg system handles some exceptions differently than others. If we protect a memory page with the PAGE_GUARD option and then try to execute within that memory page, OllyDbg will interpret the results as a memory breakpoint as opposed to a memory access exception. If OllyDbg is attached to our code when we execute from the protected memory region, a breakpoint will occur; however since an exception does not actually occurs, our structured exception handler will not be run. Without OllyDbg present, our exception handler will execute thus informing us that we are not inside of a debugger.

The first step is in this anti-debugging method is to allocate a region of memory and fill it with return opcodes. This is done by calling VirtualAlloc() followed by a call to RtlFillMemory with a final parameter of 0xC3. We have to fill the region of memory with the return opcodes so that when OllyDbg continues after the breakpoint, it will execute from the target memory and return back to the original function.

```
memRegion = VirtualAlloc(NULL, 0x10000, MEM_COMMIT,
PAGE_READWRITE);
RtlFillMemory(memRegion, 0x10, 0xC3);
```

Next we add the PAGE_GUARD permission to our target memory region via a call to VirtualProtect().

```
success = VirtualProtect(memRegion, 0x10,
PAGE_EXECUTE_READ | PAGE_GUARD, &oldProt);
```

We then setup the structured exception handler and a function pointer to point to our memory region. We call the function pointer from within the __try part of our exception handler and then handle the resultant exception which would indicate we are not currently running under OllyDbg.

```
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);
}
```

Finally we determine if our exception handler was executed, and if it was not, we know that OllyDbg had executed the memory break point as opposed to the exception handler. This indicates that we are running under the presence of the debugger.

```
if (success == 1) MessageBox(NULL, L"Debugger
Detected Via OllyDbg Memory Breakpoint Detection",
L"Debugger Detected", MB_OK);
```

Appendix: OllyDbgMemoryBreakpoint.sln

### K. *VMware Magic Port*

The virtual machine system VMware uses a "backdoor communication port" to be able to pass data between the host and the guest operating system. This communication port is used to read and write clipboard information, drag and drop between host and guest operating system, and allow file sharing between the two running systems. Communication on this port occurs by using two privileged x86 instructions, *"IN"* and *"OUT"*. These two instructions cannot normally be run from an unprivileged vantage point and would generate an exception; however when running under VMware, the emulation layer has implemented these particular instructions differently allowing them to be executed from an unprivileged vantage point. As such we can use these methods to detect if we are in a VMware virtual environment.

In the inline assembly below we setup a call to the *"IN"* instruction by pushing a number of static values onto the stack. The first parameter of interest is the static string 'VMXh'. This string is the "magic" value that must be present for the virtual machine to know that the request is legitimate. The value 10 is the particular VMware backdoor function that we wish to execute, while the value 'VX' is the default port that the VMware backdoor IO listens for. Finally we execute the *"IN"* call and analyze the return value. If the return value is zero, and we have reached our exception handler, we know that we are not running in a virtual session. If the return value is non zero and we do not reach out exception handler we are running inside of VMware.

```
int flag = 0;

__try {
  __asm {
    push edx;
    push ecx;
    push ebx;

    mov eax, 'VMXh';
    mov ebx, 0; // This can be any value except
MAGIC
    mov ecx, 10; // "CODE" to get the VMware Version
    mov edx, 'VX'; // Port Number

    in eax, dx; // Read port
    //On return EAX returns the VERSION
    cmp ebx, 'VMXh'; // is it VMware
    setz [flag]; // set return value

    pop ebx;
    pop ecx;
    pop edx;
  }
}
__except(EXCEPTION_EXECUTE_HANDLER) {
  flag = 0;
  MessageBox(NULL, L"No VMware Instance Detected",
L"No VMware", MB_OK);
}

if (flag != 0) { MessageBox(NULL, L"VMware
Detected", L"VMware Detected", MB_OK); }
}
```

Appendix: VMwareBackdoorIO.sln

## IX. Conclusions

While we can be assured that this document has not discussed every anti-debugging method in existence we hope that a majority of the more useful methods have been demonstrated. We believe that presenting the information in a manner that is easy to digest for the mid-level developer will help to ease the burden of implementation and increase the frequency of use of techniques such as these. This information will hopefully increase the barrier of entry for would be software pirates and make the path to reverse engineering of legitimate code more difficult.

## Acknowledgment

The author would like to acknowledge Chris Eng and Chris Wysopal for their support and review of this paper throughout the writing process. Additional acknowledgement should be given to Wesley Shields and Andreas Junestam, along with the people on #uninformed who acted as a source of knowledge and research data points.

## References

[1] PEB Structure break out: http://undocumented.ntinternals.net/UserMode/Undocumented%20Functions/NT%20Objects/Process/PEB.html Last accessed March 1, 2009

[2] PIB Structure break out: Appendix PIB.txt

[3] Jackson, J., "An Anti-Reverse Engineering Guide", Code Project Publication, November 9, 2008. http://www.codeproject.com/KB/security/AntiReverseEngineering.aspx, Last Accessed December 11, 2008.

[4] Falliere, N., "Windows Anti-Debug Reference", November 12, 2007, http://www.securityfocus.com/infocus/1893, Last Accessed December 11, 2008.

[5] Gagnon, M. N., Taylor, S., and Ghosh, A. K. 2007. Software Protection through Anti-Debugging. *IEEE Security and Privacy* 5, 3 (May. 2007), 82-84. DOI= http://dx.doi.org/10.1109/MSP.2007.71

[6] Ferrie, P., "Anti-Unpacker Tricks", May 2008, http://pferrie.tripod.com/papers/unpackers.pdf, Last Accessed December 11, 2008.

[7] Website: "OpenRCE Anti Reverse Engineering Techniques Database", http://www.openrce.org/reference_library/anti_reversing, Last Accessed December 11, 2008.

[8] Ferrie, P., "Attacks on More Virtual Machine Emulators", October 2008, http://pferrie.tripod.com/papers/attacks2.pdf, Last Accessed December 11, 2008.

[9] Brulez, N., "Anti-Reverse Engineering Uncovered", March 7, 2005, http://www.codebreakers-journal.com/downloads/cbj/2005/CBJ_2_1_2005_Brulez_Anti_Reverse_Engineering_Uncovered.pdf, Last Accessed December 11, 2008.

[10] Bania, P, "Antidebugging for the (m)asses - protecting the env.", http://www.piotrbania.com/all/articles/antid.txt, Last Accessed December 11, 2008.

[11] Brulez, N., "Crimeware Anti-Reverse Engineering Uncovered", 2006, http://securitylabs.websense.com/content/Assets/apwg_crimeware_antireverse.pdf, Last Accessed December 11, 2008.

[12] "Lord Julus", "Anti-Debugger and Anti-Emulator Lair", 1998, http://vx.netlux.org/lib/vlj03.html, Last Accessed December 11, 2008

[13] Liston, T., and Skoudis, E., " Thwarting Virtual Machine Detection", 2006, http://handlers.sans.org/tliston/ThwartingVMDetection_Liston_Skoudis.pdf, Last Accessed December 11, 2008.

[14] Bania, P., “Playing with RTDSC”, http://www.piotrbania.com/all/articles/playing_with_rdtsc.txt, Last Accessed December 11, 2008.

[15] Quist, D., Smith, V., “Detecting the Presence of Virtual Machines Using the Local Data Table”, http://www.offensivecomputing.net/files/active/0/vm.pdf, Last Accessed December 11, 2008.

[16] Omella, A. A., “Methods for Virtual Machine Detection”, June 20, 2006, http://www.s21sec.com/descargas/vmware-eng.pdf, Last Accessed December 11, 2008.

[17] Tariq, T. B., “Detecting Virtualization”, May 14, 2006, *Talha Bin Tariq Blog,* http://talhatariq.wordpress.com/2006/05/14/detecting-virtualization-2/, Last Accessed December 11, 2008.

[18] Rutkowska, J., " Red Pill... or how to detect VMM using (almost) one CPU instruction”, November 2004, http://www.invisiblethings.org/papers/redpill.html, Last Accessed December 11, 2008.