Detecting "Certified Pre-owned" Software

Tyler Shields, Veracode
Chris Wysopal, Veracode
March 25, 2009

Introduction

Backdoors in software, whether maliciously inserted or carelessly introduced, are a risk that should be detected prior to the affected software or system being deployed. We call software and devices that come with malicious functionality built in, "Certified Pre-Owned".

Modern static analysis methods can detect many classes of common vulnerabilities. Static analyzers do this by building a semantic model of the software which typically includes control flow and data flow graphs. This model is then scanned for patterns that typically lead to vulnerabilities such as buffer overflows. Static analysis methods can also be targeted at detecting code that offers backdoor functionality to an attacker who knows of its existence. Binary static analysis has the powerful capability of being able to use static analysis techniques when source code is not available, which is the typical case when a consumer is concerned about detecting a backdoor in a product they have purchased.

Special credentials, hidden commands, and unintended information leakage are a few of the types of application backdoors that have been discovered in commercial and open source software. Rules can be created for a static analyzer to inspect for these patterns. Rules can also be created that inspect for evidence that someone is hiding functionality in software by looking for rootkit behavior and anti-debugging functionality. If the analyzer finds these "anti-reverse engineering" patterns, further inspection should be performed to determine what the software is hiding.

Technical Summary

Backdoors are a method of bypassing authentication or other security controls in order to access a computer system or the data contained on that system. Backdoors can exist at a number of different levels including the system level, in a cryptographic algorithm, or within application code. We have concentrated on application backdoors which are embedded within the code of a legitimate application. We define application backdoors as versions of legitimate software modified to bypass security mechanisms under certain conditions. These legitimate programs are meant to be installed and running on a system with the full knowledge and approval of the system operator.

Application backdoors can result in the compromise of the data and transactions performed by an application. They can also result in system compromise.

Application backdoors are often inserted in the code by someone who has legitimate access to the code. Other times the source code or binary to an application is modified by someone who has compromised the system where the source code is maintained or the binary is distributed. Another method of inserting an application backdoor is to subvert the compiler, linker, or other components in the development tool chain used to create the application binary from the source code. Ken Thompson famously performed this feat with his C compiler for UNIX. He modified it to insert a backdoor into the UNIX login program whenever is compiled login.c.

Application backdoors are best detected by inspecting the source code or statically inspecting the binary. It is impossible to detect most types of application backdoors dynamically because they use secret data or functionality that cannot be adequately tested for using only dynamic methods .

Application backdoor analysis is imperfect. It is impossible to determine the intent of all application logic. Well known backdoor mechanisms can be heavily obfuscated and novel mechanisms can certainly be employed. Another angle of detection is to look for signs that the malicious actor is trying to hide their tracks with rootkit behavior, anti-debugging and/or code and data obfuscation techniques.

In the past, backdoors in source code have been detected quickly while backdoors in binaries often hide from detection for years. A survey of the last 15 years of discovered backdoors bears this out. One famous backdoor that was in a program only available in binary form was the “special credential” backdoor in Borland Interbase. A famous open source backdoor was a modification attempted on the Linux kernel code. The Linux kernel “uid=0” backdoor attempt was quickly discovered but the Borland Interbase backdoor lasted for many years until the software was open sourced. In general, backdoors in open source software tend to be discovered quickly while backdoors in binaries can last for years.

For compiled software, a subverted development tool chain or compromised distribution site requires binary analysis for backdoor detection since the backdoor only exists after compilation or linking. In addition, modern development practices often dictate the usage of frameworks and libraries where only binary code is available. When backdoor reviews are performed at the source code level there are still significant portions of the resultant software that are not getting reviewed. For these reasons we have chosen to implement our static detection techniques on binary executables.

By researching backdoors that have been discovered, it is possible to create rules for a static analyzer to inspect for backdoor behavior. One such example is a “special credential” back door. This is when the software has a special username, password,

password hash, or key that is embedded within the software. If an attacker knows that special credential they can authenticate to the software no matter what the contents of the authentication store. A static analyzer can inspect the cryptographic functions that are used by the authentication routine and the data flow graph that connects to these function calls. A special credential can be detected by looking for static or computed values that do not come from the authentication store yet allow authentication.

There are several categories of application backdoors which can be detected using automated static analysis:

- Special credentials
- Unintended network activity
- Deliberate information leakage

Other constructions in the code that indicate that a backdoor or other malicious code may be present can also be detected statically. These include:

- Embedded Shell Commands
- Time Bombs
- Rootkit-like Behavior
- Code or Data Anomalies
- Self-modifying Code
- Anti-debugging

Looking for Backdoor Indicators

In this paper we will concentrate on looking for indicators that the software is trying to hide its behavior from dynamic run time detection.

Two categories of backdoor indicators that are designed to evade detection by run time analysis are rootkit behavior and anti-debugging. Rootkit behavior modifies the functionality of the operating system the software is running on so that instrumentation and system administration tools cannot detect the existence of a backdoor. An example of rootkit behavior is a backdoor that opens up a network port listening for incoming connections and also modifies the operating system functions that enumerate all listening network ports. Anti-debugging is an approach that makes runtime inspection of the running code more difficult. If the software containing the backdoor can stop a debugger from single stepping through the code or inspecting

internal data structures it makes it difficult to determine if the software has unwanted backdoor functionality at execution time.

One of the solutions to this challenge we propose is to look directly for the code that implements the anti-debugging or rootkit behavior using static analysis.

Rootkit Behavior

The following section details common techniques that rootkits use to hide their behavior from the instrumentation a reverse engineer or system administrator might use. These techniques include: DLL injection, IAT hooking, and Inline function hooking. A scan rule can be created for each of the following coding patterns such that a static analyzer can search for the pattern in the code.

Windows DLL Injection

DLL Injection can be used for many legitimate purposes; however its usage is a red flag that should cause the application metadata to be analyzed. If the scanned application is intended to inject DLLs into external processes then there is a chance these are false positive returns; however if the scanned application should not be executing DLL injection of any manner, positive hits on these scan tests are highly likely to be indicative of a userland rootkit or subversive function.

Via the Registry

Injecting a DLL into a target process can be achieved via modification of a registry entry. A static analysis engine needs to flag any registry modifications to the following key:

HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Windows \AppInit_DLLs = *

This key will load a DLL into all processes as they are created. An attacker would use this to hook function calls in injected processes. This requires a reboot to affect all processes; however as soon as the key is modified, all future processes will be impacted.

Modification of this registry key can be accomplished via calls to RegCreateKey, RegCreateKeyEx, RegLoadKey, and RegOpenKey followed by RegSetValueEx. Static engine detection should detect modification to the specific registry keys using any of the above stated functions.

Using Windows Hooks

It is also possible to inject a DLL via windows hook calls. The call SetWindowsHookEx will hook a target process and load a DLL of our choosing into the target process. This DLL could then hook the IAT or execute inline hooking as desired.

```
HHOOK SetWindowsHookEx
{
        int idHook,
        HOOKPROC lpfn,
        HINSTANCE hMod,
        DWORD dwThreadId
};
```

For example:

```
myDllHandle = Rootkit DLL
SetWindowsHookEx(WH_KEYBOARD, myKeyBrdFuncAd, myDllHandle, 0)
```

Rootkit DLL has the myKeyBrdFuncAd defined and written.

Using Remote Threads

It is possible to inject a DLL into a target process by creating and using remote threads. First we programmatically find the process ID of the target process by using the OpenProcess call. We then allocate some memory in the target process for our thread via VirtualAllocEx and then write our DLL name into the target process memory space. Finally we create a remote thread by calling the CreateRemoteThread with the PID and memory location of our injected DLL name and a reference to the LoadLibraryA function.

```
PID = OpenProcess(DWORD dwDesiredAccess, BOOL bInheritHandle, DWORD
dwProcessId);
// This is used to find the PID of our target process

ADDRESS = GetProcAddress(GetModuleHandle(TEXT( "Kernel32")), "LoadLibraryA");
// This is used to find the address of LoadLibraryA in our current process. We assume
that
// the base is the same in our target thus keeping the function location the same.

BASEAD = VirtualAllocEx(PID, NULL, len_of_our_dll_name_string, MEM_COMMIT |
MEM_RESERVE, PAGE_READWRITE)
// The above allocates some memory in our target process

WriteProcessMemory(PID, BASEAD, Pointer to BUF containing "c:\path\to\our\dll", size,
NULL)
CreateRemoteThread(PID, NULL, 0, ADDRESS, BASEAD, 0, NULL)
```

DLL injection simply injects the DLL, it does not actually execute the IAT or inline hook. An example DLL that we could use with the injection techniques outlined in this section is below.

Thread Suspend and Hijack

It is also possible to inject a DLL by directly modifying the thread execution of a process and injecting data of our choosing. To execute this injection the attacker first must pick a process and walk the threads of the process by using calls to CreateToolhelp32Snapshot, Thread32First, and Thread32Next. Once the target thread has been determined, the attacker suspends the handle that was acquired by executing a call to SuspendThread.

After the target thread has been suspended, a call to VirtualAllocEx followed by WriteProcessMemory occurs. Using these two calls we write a small section of assembly code to our allocated memory within the suspended target thread. This assembly code executes the LoadLibraryA function and looks like the following:

pushfd; push EFLAGS
pushad; push general purpose registers

push &<dll_path>; push the address of the DLL path string we have already injected
call LoadLibraryA; call the LoadLIbraryA function.

popad; pop the general purpose registers
popfd; pop EFLAGS
jmp Original_Eip; resume execuation at the original EIP value

The dll_path value is dependent upon the return results of VirtualAllocEx and where the data is written to memory. The same is true for the original EIP value. These two values should be programmatically determined when the injection is being made.

Triggering our injected payload occurs by direct modification of the EIP register of the remote thread to point to the address of our injected code. Next a call to SetThreadContext makes the change to the EIP permanent. Finally we tell the system to resume the thread via ResumeThread.

Sample DLL for IAT/Inline Injection

```
BOOL APIENTRY DllMain(HANDLE hModule, DWORD ul_reason_for_call, LPVOID
lpReserved)
{
	if (ul_reason_for_call == DLL_PROCESS_ATTACH)
	{
		// EXECUTE THE IAT OR INLINE HOOK HERE
	}
	return TRUE:
}

__declspec (dllexport) LRESULT myKeyBrdFuncAd (int code, WPARAM wParam, LPARAM
lParam)
```

```
{
        return CallNextHookEx(g_hhook, code, wParam, lParam)
}
```

Windows Userland Rootkits

Windows userland based rootkits target the hooking and modification of processes in ring three. By not having a requirement to go to a lower operating level, such as ring one, these are the simplest forms of rootkit technologies that can be implemented. Static detection of userland process modifications target specific sequences of calls that are indicative of the existence of a hooking utility or rootkit. Each method below outlines the API calls that should be identified and flagged on when using a static analysis engine for the detection of these types of activities.

IAT Hooking

IAT hooking is the modification of the Import Address Table for a binary that has been loaded into memory. This is typically done by injecting a DLL into the running process and executing the IAT modification code in the context of the target process. The code first locates the IAT table within the loaded image by using the following reference:

```
(IMAGE_DOS_HEADER->e_lfanew)->OptionalHeader-
>DataDirectory[IMAGE_DIRECTORY_ENTRY_IMPORT]
// This is using the image base as a starting reference point and the other values as
offsets.
```

Once the reference to the IAT is discovered, code is used to walk the IAT table entries until the entry that matches the function we would like to patch is discovered. At this point we execute the following code to unlock the memory protections of the segment of memory we will be modifying, overwrite the target function pointer, and then revert the memory protections that were originally in place.

```
/* Unlock read only memory protection */
VirtualProtect((LPVOID)(&pIteratingIAT-
>u1.Function),sizeof(DWORD),PAGE_EXECUTE_READWRITE,&dwProtect);

/* OVERWRITE API address! :) */
(DWORD*)pIteratingIAT->u1.Function = (DWORD*)pApiNew;

/* Restore previous memory protection */
VirtualProtect((LPVOID)(&pIteratingIAT-
>u1.Function),sizeof(DWORD),dwNewProtect,&dwProtect);
```

To detect IAT hooking we should identify any reference to the IAT structure noted above that is followed by memory unprotect, write, and re-protect code. Alternate code to unprotect and re-protect memory is demonstrated within the SSDT hooking portion of this document. This is interchangeable with the above VirtualProtect, write,

VirtualProtect method above. Static analysis should check for both moethods in all places where one is used.

Inline Function Hooking (Runtime Patching)

Runtime patching is the modification of the function to be called directly in memory. By runtime patching an exported library call, the execution of that call can be subverted and arbitrary code of our choosing executed in its place. Runtime patching modifies the first bytes of the destination function, typically with a jmp instruction, while saving the original bytes for later use. The first bytes of the target function are modified to jmp to a function of our choosing within our injected DLL. Upon completion of the rerouted function, we call the saved bytes and execute a jmp back to the subverted function plus the appropriate offset of our injected bytes.

The most direct method of detection for this type of rootkit activity is to check for the existence of a function that is declared (naked), meaning no epilog and prolog will exist around the function, which ends in a jmp statement. Standard functions will have epilog and prolog data surrounding the function within the disassembly; however to be able to properly insert our function we would have to end the function without a prolog and on a jmp statement such that the function would properly return to the original hijacked function call.

```
__declspec(naked) my_function_detour_ntdeviceiocontrolfile()
{
	__asm
	{
		//exec missing instructions
		push ebp
		mov ebp, esp
		push 0x01
		push dword prt [ebp+0x2c]

		// do anything we want here

		// end on a jmp statement to jump back to the original
		// functionality. The address is stamped in later by the rootkit
		_emit 0xEA
		_emit 0xAA
		_emit 0xAA
		_emit 0xAA
		_emit 0xAA
		_emit 0x08
		_emit 0x00
	}
}
```

As a point of interest, it should be noted that this is the same method that the Microsoft Detours API package utilizes when creating its inline hooks. This type of detection should work against the MS Detours API by default.

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 binary. These techniques are used by commercial executable protectors, packers, and malicious software, to prevent or slow-down the process of reverse-engineering. Out static analysis engine has implemented scans for generic and targeted anti-debugging and anti-tracing techniques as outlined below.

API

The most straightforward method of anti-debugging uses operating system provided API functions to determine the existence or operation of a debugger. Some of the API calls are documented features provided by the operating system itself, while others are unpublished functions that can be linked at runtime from various system DLL files. Occasionally it is difficult to align a particular anti-debugging technique with only one class so some overlap between classes may exist.

IsDebuggerPresent Windows API

The IsDebuggerPresent API call checks to see if a debugger is attached to the running process. This is a Windows specific API call that checks the process environment block (PEB) for the PEB!BeingDebugged flag and returns its value.

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

CheckRemoteDebuggerPresent Windows API

The CheckRemoteDebuggerPresent API call takes two parameters. The first parameter is a handle to the target process while the second parameter is a return value indicating if the target process is currently running under a debugger. The word “remote” within CheckRemoteDebuggerPresent does not require that the target process be running on a separate system. The API call uses a call to ntdll! NtQueryInformationProcess with ProcessInformationClass set to ProcessDebugPort.

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

OutputDebugString on Win2K and WinXP

The function OutputDebugString operates differently based on the presence of a debugger. The return error message can be analyzed to determine if a debugger is

present. If a debugger is attached, OutputDebugString does not modify the GetLastError message.

```
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);
}
```

FindWindow

OllyDbg by default has a window class of "OLLYDBG". This can be detected using a function call to FindWindow with a first parameter of "OLLYDBG". WinDbg can be detected with an identical method instead searching for the string WinDbgFrameClass.

```
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);
}
```

OllyDbg OpenProcess HideDebugger Detection

The "Hide Debugger" plugin for OllyDbg modifies the OpenProcess function at offset 0x06. The plugin places a far jump (0xEA) in that location in an attempt to hook OpenProcess calls. This can be detected programmatically and acted upon.

```
hMod = GetModuleHandle(L"kernel32.dll");
procAdd = (void*) GetProcAddress(hMod, "OpenProcess");
ptr = (byte *)procAdd;
ptr = ptr + 0x06; // Offset to the byte modified by HideDebugger Plugin
if (*ptr == 0xea) {
	MessageBox(NULL, L"Debugger Detected Via 0xEA Olly Hide Debugger Check",
L"Debugger Detected", MB_OK);
} else {
	MessageBox(NULL, L"No Debugger Detected", L"No Debugger", MB_OK);
}
```

Debugger Registry Key Detection

This is a very basic check to determine if there is a debugger installed on the system. This does not determine if the debugger is currently running. This technique can be used to assist other anti-debugging methods by adding an additional data point to previously existing heuristics.

```
if (RegOpenKeyEx(HKEY_LOCAL_MACHINE, L"SOFTWARE\\Microsoft\\Windows NT\
\CurrentVersion\\AeDebug", 0, KEY_QUERY_VALUE, &hKey)==0) {
	RegQueryValueEx(hKey, L"Debugger", 0, NULL, (LPBYTE)&lpData, &BufferSize);
}

if (RegOpenKeyEx(HKEY_CLASSES_ROOT, L"exefile\\shell\\Open with Olly&Dbg\
\command", 0, KEY_QUERY_VALUE, &hKey)==0) {
	RegQueryValueEx(hKey, NULL, 0, NULL, (LPBYTE)&lpData, &BufferSize);
}

if (RegOpenKeyEx(HKEY_CLASSES_ROOT, L"dllfile\\shell\\Open with Olly&Dbg\
\command", 0, KEY_QUERY_VALUE, &hKey)==0) {
	RegQueryValueEx(hKey, NULL, 0, NULL, (LPBYTE)&lpData, &BufferSize);
}
```

NtQueryInformationProcess ProcessDebugPort Detection

The NtQueryInformationProcess function is located within ntdll.dll. A call to this function using a handle to our currently running process and a ProcessInformationClass value of ProcessDebugPort (7) will return the debugging port that is available. If the returned value is zero, no debugging port is available and the process is not being debugged. If a value if returned via this function, the process is currently being debugged.

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

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);
}
```

OllyDbg Filename Format String

OllyDbg contains a flaw where it crashes if the name of the file that is being opened contains a value of %s. Putting a %s in our filename will stop Ollydbg from functioning.

Thus we can create a file with a “%s” string in the name and within our code we check that our name has not changed. If it has changed we can safely assume that someone is trying to debug our file with OllyDbg.

```
GetModuleFileName(0, (LPWCH) &pathname, 512);
printf("Filename: %ls", pathname);

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);
}
```

OllyDbg IsDebuggerPresent Detection

Many anti-anti-debugging plugins for OllyDbg (and other debuggers) will hook the IsDebuggerPresent function call so that they can always return a value indicating false. This effectively bypasses generic IsDebuggerPresent antidebugging techniques. An attack against this hooking method is to set the PEB!BeingDebugged byte to an arbitrary value and then call IsDebuggerPresent. If the request does not return your arbitrary value, you know that something has hooked that function call and returned a modified response.

```
status = (_NtQueryInformationProcess) (hnd, ProcessBasicInformation, &pPIB,
sizeof(PROCESS_BASIC_INFORMATION), &bytesWritten);

if (status == 0) {
	pPIB.PebBaseAddress->BeingDebugged = 0x90;
}
```

OllyDbg OpenProcess String Detection

OllyDbg has a static string at offset 0x004B064B that contains the value 0x594C4C4F. It's possible to enumerate all processes and walk them looking for this static string at this offset in all processes. If the string is present we know we have a running OllyDbg process on the system.

```
if (ReadProcessMemory(hProcess, 0x4B064B, &value, (SIZE_T) 4, &read)) {
...
	if (value == 0x594C4C4F) {
...
```

NtSetInformationThread Debugger Detaching

By calling NtSetInformationThread with a ThreadInformationClass of 0x11 it is possible to detach our current thread from any attached debugger.

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

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

kernel32!CloseHandle Debugger Detection

The CloseHandle call generates a STATUS_INVALID_HANDLE exception if passed an invalid handle value. This exception will be trapped by the debugger and can be used by a program to determine if it is running inside of a debugger.

```
__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);
```

Self-Debugging

A process can determine if it is being debugged by attempting to debug itself. This is done by creating a child process which then attempts to debug its parent. If the child is not able to attach to the parent as a debugger, it is a strong indicator that our process is being run under a debugger.

```
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);
```

ProcessDebugFlags

A call to NtQueryInformationProcess with a second parameter of 31 (ProcessDebugFlags Enum) will return a DWORD indicating if the target process is being debugged. This request returns the inverse of the flag and thus if a 0 is returned, our process is being debugged.

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

ProcessDebugObjectHandle

Just like the DebugProcessFlags method, a call to NtQueryInformationProcess with a second parameter of 0x1e will return a handle to the DebugObject if and only if the target process is being debugged. We can use this to determine the debugging status of a target process and act accordingly.

OllyDbg OutputDebugString Format String Vulnerability

OllyDbg has a format string vulnerability in the handling of OutputDebugString values. It's possible to crash OllyDbg by executing OutputDebugString(TEXT("%s%s%s%s%s%s%s%s%s%s").

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

Exception

Many times a debugger will handle exceptions on behalf of a piece of code and then not properly hand the exception back to the process for internal handling. This exception handling discrepancy can be detected and acted upon. By triggering exceptions that are caught by debuggers our process can determine if its internal exception handler was triggered thus revealing the presence of an attached debugger.

INT 2D Debugger Detection

The debugging interface is accessed by "INT 2d". To convey the service request to the kernel debugger, the trap handler for "INT 2d" constructs an EXCEPTION_RECORD structure with an exception code of STATUS_BREAKPOINT. This exception is ultimately handed to the kernel debugger. We can use this same interrupt call from ring 3 to

trigger an exception. If the application exception occurs, we are not running under a debugger; however if the exception triggers we can be sure that a debugger is present. Additionally, INT 2D can be used as obfuscation of code when being run in a debugger. Depending on the particular debugger in use, it may or may not execute the instruction directly following the INT 2D call.

OllyDbg - run until next breakpoint (if we have any)
Visual Studio - skips one instruction then breaks
WinDbg - stop after INT 2d (always even if we 'Go')

```
__try {
        __asm {
                int 2dh;
        }
}
__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);
```

To detect this type of construct a static analysis engine would have toflag on any inline asm call to INT 2d. Inline asm calls to INT 2d are generally not used in standard coding practices and should have a relatively low rate of false positive identification.

INT3 Exception Detection

If a process that is not running in a debugger triggers an INT 3, an exception will be thrown. However, if the process is running under a debugger, the debugger will capture the INT 3 call and the exception will never occur. Using this discrepancy it is possible to detect if a process is running inside of a debugger.

```
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);
```

Single Step Detection

If the Trap Flag is set within the thread context, the SEH will be called prior to the instruction occurring. By enabling the Trap Flag we can detect the presence of the SEH being fired to accurately determine if the process is running under a debugger.

```
//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
	}
}
```

OllyDbg Memory Breakpoint Detection

OllyDBG interprets PAGE_GUARD as a Memory break-point. Thus if we execute a PAGE_GUARDED page a code exception will occur. If the debugger is present it will trap this as a break point and continue executing code after it. If there is no debugger present we will catch the exception in code and handle it appropriately via the SEH.

```
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);
}
```

Ctrl-C Vectored Exception Handling

When a console program is being debugged, a Ctrl-C command will throw an exception that can be trapped by a vectored exception handler. If the program is not being debugged, no exception is thrown and only a signal handler will be called. We can detect this by registering both a signal handler for Ctrl-C as well as a vectored exception handler. If the exception handler is triggered we are running under a

debugger otherwise we are not. This does not work against OllyDbg. Visual Studio debugger is susceptible.

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

ICE Breakpoint 0xF1

An undocumented opcode within the Intel chipset is the 0xF1 or ICE Breakpoint. This opcode causes a breakpoint to occur that triggers a SINGLE_STEP exception. If this opcode is run while under a debugger, the debugger will trap the exception and not pass it on to the exception handler. If the exception handler receives the opcode directly we are not running in a debugger.

```
__try {
        __asm {
                __emit 0xF1; // ICE BREAKPOINT
        }
}
__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 ICE Breakpoint", L"Debugger
Detected", MB_OK);
```

Prefix Handling

In place inline asm instruction prefixes may be skipped when single stepping with a debugger. If we emit a REP instruction followed by a breakpoint instruction the single stepping debugger will skip the breakpoint instruction. This can be trapped and handled with a SEH.

```
__try {
        __asm {
                __emit 0xF3; // 0xF3 0x64 is PREFIX REP:
                __emit 0x64;
                __emit 0xF1; // Break that gets skipped if single stepping
        }
}
__except (EXCEPTION_EXECUTE_HANDLER) {
        flag = 1;
        MessageBox(NULL, L"No Debugger Detected", L"No Debugger", MB_OK);
}
```

Using the CMPXCHG8B with the LOCK Prefix

The LOCK prefix, when used with CMPXCHG8B is considered an invalid lock and will cause a debugger to catch an invalid instruction exception. This call will cease

debugging; however an Unhandled Exception Handler will be able to gracefully continue code execution when executed outside of a debugger.

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

Process and Thread Blocks

Some of the API based anti-debugging methods outlined above have equivalent techniques that can be executed by directly accessing the process and thread block data. The process and thread block data holds information pertinent to the execution of the particular process or thread that is used by the operating system.

IsDebuggerPresent Direct PEB Access

A flag is kept within the process environment block (PEB) for the currently running process that indicates if the process is running under a debugger. This can be detected by looking at the PEB for our process and analyzing the DebuggerPresent element of the structure.

```
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);
        }
}
```

NtGlobalFlag Debugger Detection

The PEB structure holds flags at offset 0x68 that contains information regarding the start status of the process. When a process is started under 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, and we can use this

to our advantage to identify if our process is being debugged. It is possible to detect the value located at offset 0x68 in the PEB and act accordingly.

```
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);
}
return;
```

PEB ProcessHeap Flag Debugger Detection

When a process is started under a debugger, the heap headers are created differently than when it is run without a debugger present. As the above PEB analysis method indicates, there are specific flags set that manipulate how ntdll.dll creates heaps. Looking at a heap header at offset 0x10 will indicate if the heap has been created while in a debugger or not. If the heap was created under a debugger, the offset at 0x10 will be a nonzero value.

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

Vista TEB system dll pointer

When a process is created without being under a debugger in Vista, the main thread environment block (TEB) (at offset 0xBFC) contains a pointer to a Unicode string referencing a system DLL. The string directly follows the pointer at 0xC00. If the process is debugged, the Unicode string equals "HookSwitchHookEnabledEvent".

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

TIB = getTib();         // Function to gather a TEB pointer (Can use API or inline asm)
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);
}
```

LDR_MODULE

When a process is created from within a debugger, the heap allocation routines append a hex DWORD of 0xEEFEEEFE to the end of the memory space. This can be detected by looking at the LDR_MODULE section pointer to by the PEB as it is allocated using the heap when a process is started.

```
__try {
	while (*ldr_module != 0xEEFEEEFE) {
		printf("Value at pointer: %08X\n", *ldr_module);
		walk = walk +0x01;
		ldr_module = walk;
	}
}
```

Size Based Anti Dumping

When a PE executable file is loaded into memory, the header of the process (PE Header) contains a value indicating the size of the image in memory. Dumping programs such as LordPE and ProcDump use this information to determine the length of the process in memory when attempting to dump the image. Since this information is not used by the binary itself to execute, it is possible to fool dumping applications into the thinking the image is larger or smaller than it is. The result is that the dumping application may crash or dump an otherwise inoperable binary to disk from memory.

```
PIMAGE_DOS_HEADER image_addr;
PIMAGE_DOS_HEADER dosHeader;
PIMAGE_NT_HEADERS pNTHeader;
...
pNTHeader->OptionalHeader.SizeOfImage = sizeOfImage + 0x3000;
```

Registers

CPU hardware registers contain information that can aid debugging processes. These hardware breakpoints rely upon registers internal to the CPU to hold address and trigger data and to respond when certain addresses are discovered on the bus.

Hardware Breakpoint Detection

There are two different types of breakpoints, software and hardware breakpoints. When a hardware breakpoint is set the CPU debug registers are used to hold the specific breakpoint information. DR0-DR3 holds the address that is used to break the program execution while DR7 holds context information about the breakpoints in DR0-DR3. We can access these debug registers via the GetThreadContext() function and determine if hardware breakpoints have been set and act accordingly.

```
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);
}
```

VMware Detection

Multiple techniques exist to detect if our process is running within an instance of VMware. These techniques are based on the fact that some processor specific registers can be accessed from user mode and will return the value that the host system is storing in those registers. Due to this fact, the virtual machine must emulate those registers and will return values that are different than the standard values stored by the host operating system. IDT and GDT values were originally used to detect the presence of a virtual machine but due to the advent of multiple processors they became less accurate. However, checks against the LDT register are impervious to multiple processor differences. As such it is possible to reference this register from code and accurately determine if we are running under an instance of VMware. If the return value for the first and second bytes is not equal to zero we can be certain that we are within a virtual machine.

```
__asm {

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

Timing

Timing based anti-debugging methods are primarily used to detect single stepping through a process. By surrounding sensitive code blocks with time states, or alternatively conducting two time checks in succession, it is possible to determine the

execution latency between lines of code. If the time is too large when compared against a reasonable threshold we know that single stepping is occurring.

RDTSC Instruction Debugger Latency Detection

It is possible to detect a single stepping debugging effort by calculating the delta between two rdtsc calls. If the resultant delta value is greater than 0xFF (arbitrary threshold), then the program is being single stepped.

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

Kernel-Mode Timer NtQueryPerformanceCounter

Similar to the rdtsc technique, QueryPerformanceCounter can be used to determine if a debugger is connected to our process and single stepping. Simply call the QueryPerformanceCounter twice and determine the delta. If the delta is above a reasonable threshold (0xFF in our case) we are single stepping.

```
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);
}
```

GetTickCount Timer

This is the same method as the other timing related anti-debugging techniques. This technique uses GetTickCount functions to determine an execution timing delta. If the delta is too large we can assume that our process is being single stepped manually. The primary difference between this and other timing related techniques is that the threshold must be set much lower as this is a lower precision timer than the others in use.

timeGetTime Timer

Identical to other timer methods using timeGetTime function to determine execution deltas.

Conclusion

The list of techniques a programmer can use to hide from runtime analysis continues to grow as new methods are found over time. Like most areas of security research this is a classic cat and mouse game. rootkit techniques are a fertile ground of research and new operating systems and virtual machines give additional debugging opportunities that the anti-reverse engineer will want to thwart. The list is long and growing. However, when a new technique is discovered it is relatively easy to add a scan or check for the pattern with a rules based static analyzer. Binary static analysis is up to the task of looking for the tell tale signs that a program is trying to hide its behavior from a reverse engineer which is often a tip off that something unwanted is hidden in the software.

References

1. Reflections on trusting trust. Commun. ACM 27, 8 (Aug. 1984), 761-763. DOI= http://doi.acm.org/10.1145/358198.358210, Thompson, K. 1984.
2. http://www.veracode.com/images/stories/static-detection-of-backdoors-1.0.pdf, Chris Wysopal and Chris Eng, August 2007
3. http://www.offensivecomputing.net/files/active/0/vm.pdf, Danny Quist and Val Smith
4. http://www.codeproject.com/KB/security/AntiReverseEngineering.aspx, Josh Jackson, 11/9/2008
5. http://www.securityfocus.com/infocus/1893, Nicolas Falliere, 9/12/2007
6. http://ieeexplore.ieee.org/Xplore/login.jsp?url=/iel5/8013/4218538/04218560.pdf?temp=x, Michael N. Gagnon et al., Jun-07
7. http://pferrie.tripod.com/papers/unpackers.pdf , Peter Ferrie, May-08
8. http://pferrie.tripod.com/papers/unpackers.ppt, Peter Ferrie, May-08
9. http://www.openrce.org/reference_library/anti_reversing, Mixed, Unknown
10. http://pferrie.tripod.com/papers/attacks2.pdf, Peter Ferrie, Oct-08
11. http://www.codebreakers-journal.com/downloads/cbj/2005/CBJ_2_1_2005_Brulez_Anti_Reverse_Engineering_Uncovered.pdf, Nicolas Brulez, 3/7/2005
12. http://www.piotrbania.com/all/articles/antid.txt, Piotr Bania, Unknown
13. http://securitylabs.websense.com/content/Assets/apwg_crimeware_antireverse.pdf, Nicolas Brulez, 2006
14. http://vx.netlux.org/lib/vlj03.html, Lord Julus, 1998
15. http://handlers.sans.org/tliston/ThwartingVMDetection_Liston_Skoudis.pdf, Tom Liston and Ed Skoudis, 2006
16. http://nagareshwar.securityxploded.com/2007/12/16/new-antidebugging-timer-techniques/, Nagareshwar Talekar, 12/16/2007
17. http://www.piotrbania.com/all/articles/playing_with_rdtsc.txt, Piotr Bania, Unknown