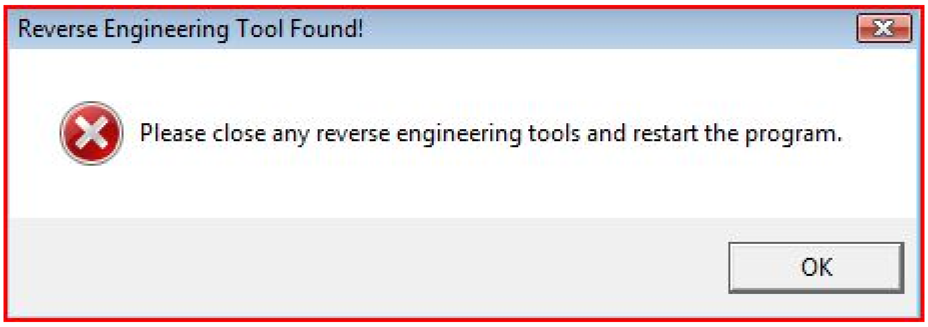
An Anti-Reverse Engineering Guide

Joshua Tully

9 Nov 2008 CPOL

A look into what goes into the area of preventing reverse engineering, and gives developers some functions and ideas about preventing reversing engineering of their programs.

Download source code - 4.87 KB


Reverse Engineering Tool Found!

Please close any reverse engineering tools and restart the program.

Table of Contents

Introduction

In my previous article, I gave a short introduction into some Anti-Debugging/Debugger Detection techniques that primarily involved the use of Win32 API functions. In this article, I plan to travel a bit deeper into the interesting world of reverse engineering and explore some more intermediate level techniques for annoying reverse engineers. Some comments in my previous article noted that the techniques I presented could, and are most of the time, easily bypassed by intermediate level reversers; one statement I would like to make is that there is an ongoing battle between the coders who develop programs that protect against cracking and reverse engineering and the engineers themselves. Every time the protectors release a new technique, the engineers find a way around that specific method. This is the driving force behind the cracking "scene" and anti-reverse engineering fields. Most of the techniques here can easily be bypassed, and some of the others aren't as easily taken out of the picture; however, all of them can in one way, shape, or form be bypassed. I'm presenting these methods here to share the knowledge, and perhaps inspire others to find ways to apply these methods and utilize them in new and creative ways that challenge contemporary methodology.

Background

Anyone who is interested in the field of reverse engineering needs a strong understanding of Assembly language, so if your ASM is a little rusty or if you're just beginning to learn, here are some sites that can assist:

- Introduction to Assembly Language
- IA-32 Instruction Reference
- Iczelion's Win32 Assembly Homepage

Inline Functions

I didn't feel this side note required its own section; however, when reading this article or the attached source, one will notice the functions being marked inline. While this can cause bloat inside an executable, it is important in anti-reverse engineering. If there are very detailed function entries and sections, then the job for the reverse engineer just got much easier. Now, he or she knows exactly what is happening when that function is called. When in-lining, this doesn't happen, and the engineer is left guessing as to what is actually happening.

Breakpoints

There are three types of breakpoints available to a reverse engineer: hardware, memory, and `INT 3h` breakpoints. Breakpoints are essential to a reverse engineer, and without them, live analysis of a module does him or her little good. Breakpoints allow for the stopping of execution of a program at any point where one is placed. By utilizing this, reverse engineers can put breakpoints in areas like Windows APIs, and can very easily find where a badboy message (a messagebox saying you entered a bad serial, for example) is coming from. In fact, this is probably the most utilized technique in cracking, the only competition would be a referenced text string search. This is why breakpoint checks are done over important APIs like `MessageBox`, `VirtualAlloc`, `CreateDialog`, and others that play an important role in the protecting user information process. The first example will cover the most common type of breakpoint which utilizes the `INT 3h` instruction.

INT 3

`INT 3h` breakpoints are represented in in the IA-32 instruction set with the opcode CC (0xCC). This is the most common expression of this type of breakpoint; however, it can also be expressed as the byte sequence 0xCD 0x03 which can cause some troubles. Detecting this type of breakpoint is relatively simple, and some source would look like the following sample. However, we should be careful because using this method of scanning can lead to false positives.

C++

```cpp
bool CheckForCCBreakpoint(void* pMemory,  size_t SizeToCheck)
{
     unsigned char *pTmp = (unsigned char*)pMemory;
    for (size_t i = 0; i < SizeToCheck; i++)
```

```
     {
         if(pTmp[i] == 0xCC)
             return true;
     }

    return false;
}
```

Here's another obfuscated method for checking for `INT 3` breakpoints. It is important to remember that the code shown above would stick out like a sore thumb to even new reversers. By adding another level of indirection, you, the protector, are improving your chances of successfully protecting the application.

C++

```
bool CheckForCCBreakpointXor55(void* pMemory,  size_t SizeToCheck)
 {
     unsigned char *pTmp = (unsigned char*)pMemory;
    unsigned char tmpchar = 0;

    for (size_t i = 0; i < SizeToCheck; i++)
     {
        tmpchar = pTmp[i];
        if( 0x99 == (tmpchar ^ 0x55) ) // 0xCC xor 0x55 = 0x99
            return true;
     }

    return false;
 }
```

Memory Breakpoints

Memory breakpoints are implemented by a debugger using guard pages, and they act like "a one-shot alarm for memory page access" (Creating Guard Pages). In a nutshell, when a page of memory is marked as `PAGE_GUARD` and is accessed, a `STATUS_GUARD_PAGE_VIOLATION` exception is raised, which can then be handled by the current program. At the moment, there's no accurate way to check for memory breakpoints. However, we can use the techniques a debugger uses to implement memory breakpoints to discover if our program is currently running under a debugger. In essence, what occurs is that we allocate a dynamic buffer and write a `RET` to the buffer. We then mark the page as a guard page and push a potential return address onto the stack. Next, we jump to our page, and if we're under a debugger, specifically OllyDBG, then we will hit the `RET` instruction and return to the address we pushed onto the stack before we jumped to our page. Otherwise, a `STATUS_GUARD_PAGE_VIOLATION` exception will occur, and we know we're not being debugged by OllyDBG. Here is an example in source:

C++

```
bool MemoryBreakpointDebuggerCheck()
{
     unsigned char *pMem = NULL;
     SYSTEM_INFO sysinfo = {0};
     DWORD OldProtect = 0;
     void *pAllocation = NULL; // Get the page size for the system

    GetSystemInfo(&sysinfo); // Allocate memory
    pAllocation = VirtualAlloc(NULL, sysinfo.dwPageSize,
                        MEM_COMMIT | MEM_RESERVE,
                         PAGE_EXECUTE_READWRITE);

    if (pAllocation == NULL)
        return false;

    // Write a ret to the buffer (opcode 0xc3)
    pMem = (unsigned char*)pAllocation;
    *pMem = 0xc3;

    // Make the page a guard page
    if (VirtualProtect(pAllocation, sysinfo.dwPageSize,
                    PAGE_EXECUTE_READWRITE | PAGE_GUARD,
                    &OldProtect) == 0)
    {
        return false;
    }

    __try
    {
        __asm
        {
            mov eax, pAllocation
            // This is the address we'll return to if we're under a debugger
            push MemBpBeingDebugged
            jmp eax // Exception or execution, which shall it be :D?
        }
    }
    __except(EXCEPTION_EXECUTE_HANDLER)
    {
        // The exception occured and no debugger was detected
        VirtualFree(pAllocation, NULL, MEM_RELEASE);
        return false;
    }

    __asm{MemBpBeingDebugged:}
    VirtualFree(pAllocation, NULL, MEM_RELEASE);
    return true;
}
```

Hardware Breakpoints

Hardware breakpoints are a technology implemented by Intel in their processor architecture, and are controlled by the use of special registers known as Dr0-Dr7. Dr0 through Dr3 are 32 bit registers that hold the address of the breakpoint. Dr4 and 5 are reserved by Intel for debugging the other registers, and Dr6 and 7 are used to control the behavior of the breakpoints (Intel1). There is a little bit too much information for me to cover how the Dr6 and Dr7 registers affect breakpoint behavior. However, anyone who is interested should read the Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide for an in-depth explanation of how the registers work.

Now, in order to detect and/or remove hardware breakpoints, there are two methods we can utilize: the Win32 `GetThreadContext` and `SetThreadContext`, or using Structured Exception Handling. In the first example, I'll show how to use the Win32 API functions:

C++

```
// CheckHardwareBreakpoints returns the number of hardware
// breakpoints detected and on failure it returns -1.
int CheckHardwareBreakpoints()
{
    unsigned int NumBps = 0;

    // This structure is key to the function and is the
    // medium for detection and removal
    CONTEXT ctx;
    ZeroMemory(&ctx, sizeof(CONTEXT));

    // The CONTEXT structure is an in/out parameter therefore we have
    // to set the flags so Get/SetThreadContext knows what to set or get.
    ctx.ContextFlags = CONTEXT_DEBUG_REGISTERS;

    // Get a handle to our thread
    HANDLE hThread = GetCurrentThread();

    // Get the registers
    if(GetThreadContext(hThread, &ctx) == 0)
        return -1;

    // Now we can check for hardware breakpoints, its not
    // necessary to check Dr6 and Dr7, however feel free to
    if(ctx.Dr0 != 0)
        ++NumBps;
    if(ctx.Dr1 != 0)
           ++NumBps;
    if(ctx.Dr2 != 0)
           ++NumBps;
    if(ctx.Dr3 != 0)
        ++NumBps;

    return NumBps;
}
```

The SEH method of manipulating the debug registers is much more commonly seen in anti-reverse engineering programs, and is implemented easier in ASM, as shown in the following example:

ASM

```
; One quick note about this little prelude; in Visual Studio 2008
; release builds are compiled with the /SAFESEH flag which helps
; prevent exploitation of SEH by shellcode and the likes.
; What this little snippet does is add our SEH Handler to a
; special table containing a list of "safe" exceptions handlers , which
; if we didn't in release builds our handler would never be called,
; this problem plauged me for a long time, and im considering writing
; a short article on it

ClrHwBpHandler proto
 .safeseh ClrHwBpHandler

ClearHardwareBreakpoints proc
     assume fs:nothing
     push offset ClrHwBpHandler
    push fs:[0]
    mov dword ptr fs:[0], esp ; Setup SEH
     xor eax, eax
     div eax ; Cause an exception
     pop dword ptr fs:[0] ; Execution continues here
     add esp, 4
     ret
```

```
ClearHardwareBreakpoints endp

ClrHwBpHandler proc
     xor eax, eax
    mov ecx, [esp + 0ch] ; This is a CONTEXT structure on the stack
     mov dword ptr [ecx + 04h], eax ; Dr0
     mov dword ptr [ecx + 08h], eax ; Dr1
     mov dword ptr [ecx + 0ch], eax ; Dr2
     mov dword ptr [ecx + 10h], eax ; Dr3
     mov dword ptr [ecx + 14h], eax ; Dr6
     mov dword ptr [ecx + 18h], eax ; Dr7
     add dword ptr [ecx + 0b8h], 2 ; We add 2 to EIP to skip the div eax
     ret
ClrHwBpHandler endp
```

Timing Attacks

The theory behind timing attacks is that executing a section of code, especially a small section, should only take a miniscule amount of time. Therefore, if a timed section of code takes a greater amount of time than a certain set limit, then there is most likely a debugger attached, and someone is stepping through the code. This genre of attacks has many small variations, and the most common example uses the IA-32 `RDTSC` instruction. Other methods utilize different timing methods such as `timeGetTime`, `GetTickCount`, and `QueryPerformanceCounter`.

RDTSC

`RDTSC` is an IA-32 instruction that stands for Read Time-Stamp Counter, which is pretty self-explanatory in itself. Processors since the Pentium have had a counter attached to the processor that is incremented every clock cycle, and reset to 0 when the processor is reset. As you can see, this is a very powerful timing technique; however, Intel doesn't serialize the instruction; therefore, it is not guaranteed to be 100% accurate. This is why Microsoft encourages the use of its Win32 timing APIs since they're supposed to be as accurate as Windows can guarantee. The great thing about timing attacks, in general, though is that implementing the technique is rather simple; all a developer needs to do is decide which functions he or she would like to protect using a timing attack, and then he or she can simply surround the blocks of code in a timing block and can compare that to a programmer set limit, and can exit the program if the timed section takes too much time to execute. Here is an example:

C++

```cpp
#define SERIAL_THRESHOLD 0x10000 // 10,000h ticks

DWORD GenerateSerial(TCHAR* pName)
{
    DWORD LocalSerial = 0;

    DWORD RdtscLow = 0; // TSC Low
    __asm
    {
        rdtsc
        mov RdtscLow, eax
    }

    size_t strlen = _tcslen(pName);
    // Generate serial

    for(unsigned int i = 0; i < strlen; i++)
    {
        LocalSerial += (DWORD) pName[i];
        LocalSerial ^= 0xDEADBEEF;
    }

    __asm
    {
        rdtsc
        sub eax, RdtscLow
        cmp eax, SERIAL_THRESHOLD
        jbe NotDebugged
        push 0
        call ExitProcess
        NotDebugged:
    }
    return LocalSerial;
}
```

Win32 Timing Functions

The concepts are exactly the same in this variation except that we have different means of timing our function. In the following example, `GetTickCount` is used, but as commented, could be replaced with `timeGetTime` or `QueryPerformanceCounter`.

```C++
#define SERIAL_THRESHOLD 0x10000 // 10,000h ticks


GenerateSerialWin32Attack(TCHAR* pName)
{
    DWORD LocalSerial = 0;
    size_t strlen = _tcslen(pName);

    DWORD Counter = GetTickCount(); // Could be replaced with timeGetTime()

     // Generate serial
    for(unsigned int i = 0; i < strlen; i++)
    {
        LocalSerial += (DWORD) pName[i];
        LocalSerial ^= 0xDEADBEEF;
    }

    Counter = GetTickCount() - Counter; // Could be replaced with timeGetTime()
    if(Counter >= SERIAL_THRESHOLD)
        ExitProcess(0);

    return LocalSerial;
}
```

Windows Internals

The following methods of anti-reverse engineering utilize the peculiarities of the Windows Operating System in order to implement some sort of protection, ranging from hiding a thread from a debugger, to revealing the presence of a debugger. Many of the functions used in the following examples are exported from *ntdll.dll*, and are not guarenteed by Microsoft to behave consistently in different versions of the Operating System. Therefore, some caution should be taken when using these examples in your own programs. That being said, I have yet to see one of these APIs change drastically in behavior, so do not take the previous statement as a commandment to avoid these implementations.

ProcessDebugFlags

The `ProcessDebugFlags` (0x1f) is an undocumented class that can be passed to the `NtQueryProcessInformation` function. When `NtQueryProcessInformation` is called with the `ProcessDebugFlags` class, the function will return the inverse of `EPROCESS->NoDebugInherit`, which means that if a debugger is present, then this function will return `FALSE` if the process is being debugged. Here's the `CheckProcessDebugFlags`:

C++

```
// CheckProcessDebugFlags will return true if
// the EPROCESS->NoDebugInherit is == FALSE,
// the reason we check for false is because
// the NtQueryProcessInformation function returns the
// inverse of EPROCESS->NoDebugInherit so (!TRUE == FALSE)
inline bool CheckProcessDebugFlags()
{
    // Much easier in ASM but C/C++ looks so much better
    typedef NTSTATUS (WINAPI *pNtQueryInformationProcess)
        (HANDLE ,UINT ,PVOID ,ULONG , PULONG);

    DWORD NoDebugInherit = 0;
    NTSTATUS Status;

    // Get NtQueryInformationProcess
    pNtQueryInformationProcess NtQIP = (pNtQueryInformationProcess)
        GetProcAddress( GetModuleHandle( TEXT("ntdll.dll") ),
        "NtQueryInformationProcess" );

    Status = NtQIP(GetCurrentProcess(),
            0x1f, // ProcessDebugFlags
            &NoDebugInherit, 4, NULL);

    if (Status != 0x00000000)
        return false;

    if(NoDebugInherit == FALSE)
        return true;
    else
        return false;
}
```

Debug Object Handle

Beginning from Windows XP, when a process is debugged, a debug object would be created for that debugging session. A handle to this object is also created, and can be queried using `NtQueryInformationProcess`. The presence of this handle shows that the process is being actively debugged, and this information can be quite a pain to hide since it comes from the kernel. Here's the `DebugObjectCheck` function:

C++

```cpp
// This function uses NtQuerySystemInformation
// to try to retrieve a handle to the current
// process's debug object handle. If the function
// is successful it'll return true which means we're
// being debugged or it'll return false if it fails
// or the process isn't being debugged
inline bool DebugObjectCheck()
{
    // Much easier in ASM but C/C++ looks so much better
    typedef NTSTATUS (WINAPI *pNtQueryInformationProcess)
            (HANDLE ,UINT ,PVOID ,ULONG , PULONG);

    HANDLE hDebugObject = NULL;
    NTSTATUS Status;

    // Get NtQueryInformationProcess
    pNtQueryInformationProcess NtQIP = (pNtQueryInformationProcess)
                GetProcAddress(  GetModuleHandle( TEXT("ntdll.dll") ),
                "NtQueryInformationProcess" );

    Status = NtQIP(GetCurrentProcess(),
            0x1e, // ProcessDebugObjectHandle
            &hDebugObject, 4, NULL);

    if (Status != 0x00000000)
        return false;

    if(hDebugObject)
        return true;
    else
        return false;
}
```

Thread Hiding

In Windows 2000, the guys behind Windows introduced a new class to be passed into `NtSetInformationThread`, and it was named `HideThreadFromDebugger`. It is the first anti-debugging API implemented by Windows, and is very powerful. The class prevents debuggers from receiving events from any thread that has had `NtSetInformationThread` with the `HideThreadFromDebugger` class called on it. These events include breakpoints, and the exiting of the program if it is called on the main thread of an application. Here is the `HideThread` function:

C++

```cpp
// HideThread will attempt to use
// NtSetInformationThread to hide a thread
// from the debugger, Passing NULL for
// hThread will cause the function to hide the thread
// the function is running in. Also, the function returns
// false on failure and true on success
inline bool HideThread(HANDLE hThread)
{
    typedef NTSTATUS (NTAPI *pNtSetInformationThread)
                (HANDLE, UINT, PVOID, ULONG);
    NTSTATUS Status;

    // Get NtSetInformationThread
    pNtSetInformationThread NtSIT = (pNtSetInformationThread)
```

```
        GetProcAddress(GetModuleHandle( TEXT("ntdll.dll") ),
        "NtSetInformationThread");

    // Shouldn't fail
    if (NtSIT == NULL)
        return false;

    // Set the thread info
    if (hThread == NULL)
        Status = NtSIT(GetCurrentThread(),
                0x11, // HideThreadFromDebugger
                0, 0);
    else
        Status = NtSIT(hThread, 0x11, 0, 0);

    if (Status != 0x00000000)
        return false;
    else
        return true;
}
```

BlockInput

This is about as simple as it comes. `BlockInput` does as the names suggests, and blocks mouse and keyboard messages from reaching the desired application; this technique is effective due to the fact that only the thread that called `BlockInput` can call it to remove the block ("BlockInput Function"). This isn't really an anti-reverse engineering technique, but more of a way to mess with someone debugging your application. A simple source code looks like:

C++

```
BlockInput(TRUE); // Nice and simple :D
```

OutputDebugString

The `OutputDebugString` technique works by determining if `OutputDebugString` causes an error. An error will only occur if there is no active debugger for the process to receive the string; therefore, we can conclude that if there is no error (by calling `GetLastError`) after calling `OutputDebugString`, then there is a debugger present.

C++

```
// CheckOutputDebugString checks whether or
// OutputDebugString causes an error to occur
// and if the error does occur then we know
// there's no debugger, otherwise if there IS
// a debugger no error will occur
inline bool CheckOutputDebugString(LPCTSTR String)
{
    OutputDebugString(String);
    if (GetLastError() == 0)
        return true;
    else
        return false;
}
```

Process Exploitation

These techniques exploit the Windows process environment and management system in order to implement protection. Some of these techniques, especially self-debugging, are widely used by many packers and protectors.

Open Process

This debugger detection technique exploits process privileges in order to determine if a process is currently being run under a debugger. This technique works because when a process is attached to or run under a debugger, if the process privileges are not correctly reset by the attaching debugger, the process receives the `SeDebugPrivilege` set which allows the process to open a handle to any process ("How To Use the SeDebugPrivilege to Acquire Any Process Handle"). This includes a vital system process like *csrss.exe*, which we normally wouldn't have access to. Here is some source code to illustrate the technique:

C++

```cpp
// The function will attempt to open csrss.exe with
// PROCESS_ALL_ACCESS rights if it fails we're
// not being debugged however, if its successful we probably are
inline bool CanOpenCsrss()
{
    HANDLE Csrss = 0;

    // If we're being debugged and the process has
    // SeDebugPrivileges privileges then this call
    // will be successful, note that this only works
    // with PROCESS_ALL_ACCESS.
    Csrss = OpenProcess(PROCESS_ALL_ACCESS,
                FALSE, GetCsrssProcessId());

    if (Csrss != NULL)
    {
        CloseHandle(Csrss);
        return true;
    }
    else
        return false;
}
```

Parent Process

Normally, Windows users start a process from a window created or provided by the Windows Shell. In this situation, the child process's parent process is *Explorer.exe*. Therefore, we can retrieve the process ID of *Explorer.exe* and our parent process and compare them. This is, of course, a somewhat risky process since the parent process of your process isn't guaranteed to be *Explorer.exe*; nonetheless, it is still an interesting technique, and here is an example:

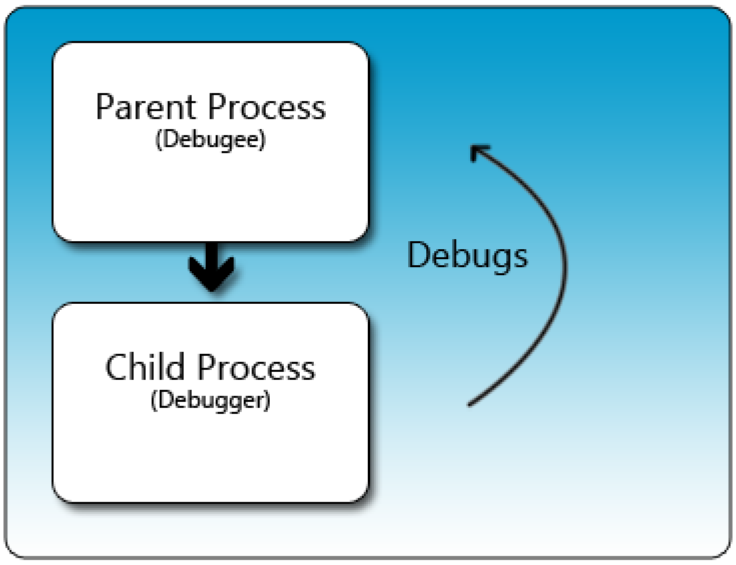
C++

```cpp
// This function returns true if the parent process of
// the current running process is Explorer.exe
```

```cpp
inline bool IsParentExplorerExe()
{
    // Both GetParentProcessId and GetExplorerPIDbyShellWindow
    // can be found in the attached source
    DWORD PPID = GetParentProcessId();
    if(PPID == GetExplorerPIDbyShellWindow())
        return true;
    else
        return false;
        //return GetParentProcessId() == GetExplorerPIDbyShellWindow()'
}
```

Self-Debugging

Self-Debugging is a technique where the main process spawns a child process that debugs the process that created the child process, as shown in the diagram. This technique can be very useful as it can be utilized to implement techniques such as Nanomites and others. This also prevents other debuggers from attaching to the same process; however, this can be bypassed be setting the `EPROCESS->DebugPort` (the `EPROCESS` structure is a struct returned by the kernel mode function `PsGetProcessId`) field to 0. This allows another debugger to attach to a process that already has a debugger attached to it. Here's some sample code:

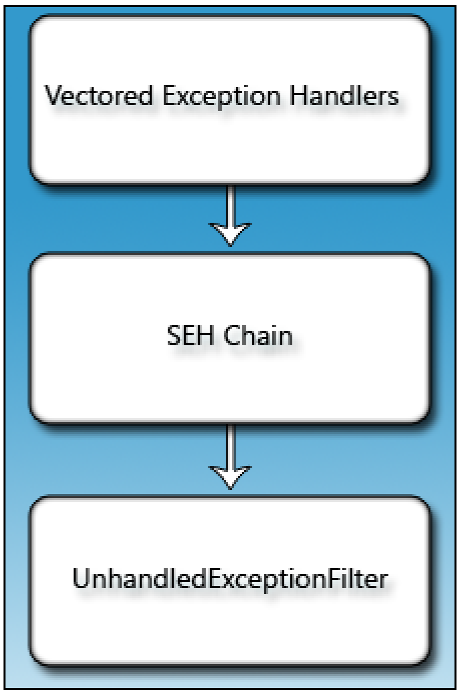


C++

```cpp
// Debug self is a function that uses CreateProcess
// to create an identical copy of the current process
// and debugs it
void DebugSelf()
{
    HANDLE hProcess = NULL;
    DEBUG_EVENT de;
    PROCESS_INFORMATION pi;
    STARTUPINFO si;
    ZeroMemory(&pi, sizeof(PROCESS_INFORMATION));
    ZeroMemory(&si, sizeof(STARTUPINFO));
    ZeroMemory(&de, sizeof(DEBUG_EVENT));

    GetStartupInfo(&si);

    // Create the copy of ourself
    CreateProcess(NULL, GetCommandLine(), NULL, NULL, FALSE,
            DEBUG_PROCESS, NULL, NULL, &si, &pi);
    // Continue execution
    ContinueDebugEvent(pi.dwProcessId, pi.dwThreadId, DBG_CONTINUE);

    // Wait for an event
    WaitForDebugEvent(&de, INFINITE);
}
```

UnhandledExceptionFilter

The `UnhandledExceptionFilter` is the long name for an exception handler that is called when there are no other handlers to handle the exception. The following diagram shows how Windows propagates exceptions. When utilizing the `UnhandledExceptionFilter` technique, one needs to be aware that **if a debugger is attached, that process will exit instead of resuming execution**, which in the context of anti-reverse engineering is quite fine, in my opinion.



C++

```
LONG WINAPI UnhandledExcepFilter(PEXCEPTION_POINTERS pExcepPointers)
{
    // Restore old UnhandledExceptionFilter
    SetUnhandledExceptionFilter((LPTOP_LEVEL_EXCEPTION_FILTER)
          pExcepPointers->ContextRecord->Eax);

    // Skip the exception code
    pExcepPointers->ContextRecord->Eip += 2;

    return EXCEPTION_CONTINUE_EXECUTION;
}

int main()
{
    SetUnhandledExceptionFilter(UnhandledExcepFilter);
    __asm{xor eax, eax}
    __asm{div eax}

    // Execution resumes here if there is no debugger
    // or if there is a debugger it will never
    // reach this point of execution
}
```

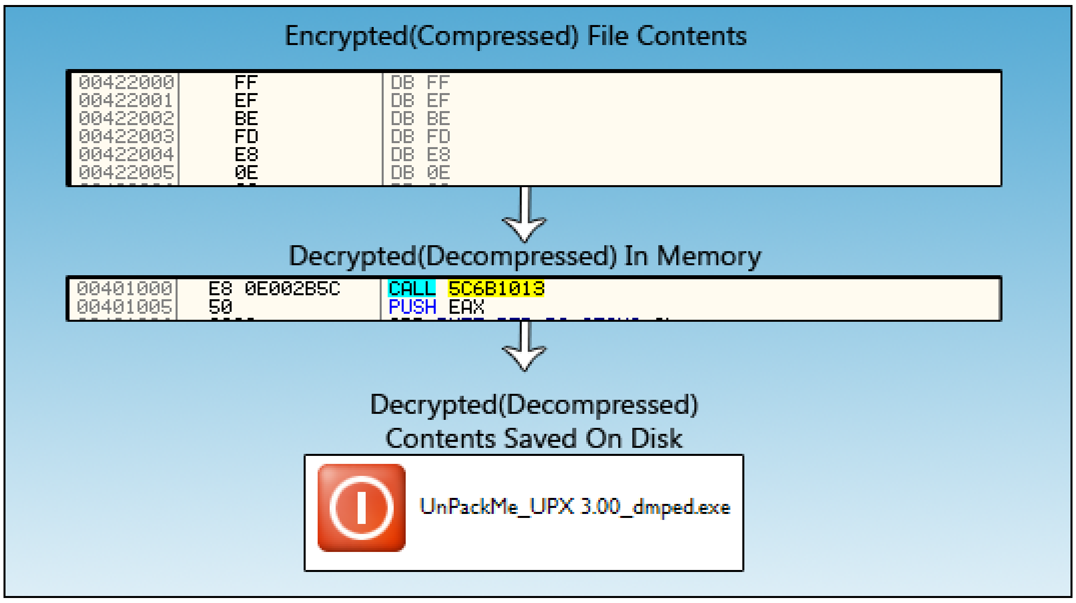
NtQueryObject

The NtQueryObject function, when called with the ObjectAllTypesInformation class, will return information about the host system and the current process. There is a wealth of information to be mined from this function, but we're most concerned with the information given about the DebugObjects in the environment. In Windows XP and Vista, a DebugObject entry is maintained in this list of objects, and most importantly, the number of objects of each type of object. The object and its related information can be expressed as a OBJECT_INFORMATION_TYPE struct. However, calling the NtQueryObject function with the ObjectAllTypesInformation class actually returns a buffer that begins with a OBJECT_TYPE_INFORMATION struct. However, there is more than one OBJECT_INFORMATION_TYPE entry, and traversing the buffer containing these entries isn't as straightforward as array indexing. The source shows that the next OBJECT_INFORMATION_TYPE struct lies after the previous' UNICODE_STRING.Buffer entry. These structs are also padded and DWORD aligned; refer to the source to examine how to navigate the buffer.

```C++
typedef struct _OBJECT_TYPE_INFORMATION {
    UNICODE_STRING TypeName;
    ULONG TotalNumberOfHandles;
    ULONG TotalNumberOfObjects;
}OBJECT_TYPE_INFORMATION, *POBJECT_TYPE_INFORMATION;

typedef struct _OBJECT_TYPE_INFORMATION {
    UNICODE_STRING TypeName;
    ULONG TotalNumberOfHandles;
    ULONG TotalNumberOfObjects;
}OBJECT_TYPE_INFORMATION, *POBJECT_TYPE_INFORMATION;

// ObjectListCheck uses NtQueryObject to check the environments
// list of objects and more specifically for the number of
// debug objects. This function can cause an exception (although rarely)
// so either surround it in a try catch or __try __except block
// but that shouldn't happen unless one tinkers with the function
inline bool ObjectListCheck()
{
    typedef NTSTATUS(NTAPI *pNtQueryObject)
            (HANDLE, UINT, PVOID, ULONG, PULONG);

    POBJECT_ALL_INFORMATION pObjectAllInfo = NULL;
    void *pMemory = NULL;
    NTSTATUS Status;
    unsigned long Size = 0;

    // Get NtQueryObject
    pNtQueryObject NtQO = (pNtQueryObject)GetProcAddress(
                GetModuleHandle( TEXT( "ntdll.dll" ) ),
                "NtQueryObject" );

    // Get the size of the list
    Status = NtQO(NULL, 3, //ObjectAllTypesInformation
                        &Size, 4, &Size);

    // Allocate room for the list
    pMemory = VirtualAlloc(NULL, Size, MEM_RESERVE | MEM_COMMIT,
                    PAGE_READWRITE);

    if(pMemory == NULL)
        return false;

    // Now we can actually retrieve the list
    Status = NtQO((HANDLE)-1, 3, pMemory, Size, NULL);

    // Status != STATUS_SUCCESS
    if (Status != 0x00000000)
    {
        VirtualFree(pMemory, 0, MEM_RELEASE);
        return false;
    }

    // We have the information we need
    pObjectAllInfo = (POBJECT_ALL_INFORMATION)pMemory;

    unsigned char *pObjInfoLocation =
        (unsigned char*)pObjectAllInfo->ObjectTypeInformation;

    ULONG NumObjects = pObjectAllInfo->NumberOfObjects;

    for(UINT i = 0; i < NumObjects; i++)
    {

        POBJECT_TYPE_INFORMATION pObjectTypeInfo =
            (POBJECT_TYPE_INFORMATION)pObjInfoLocation;

        // The debug object will always be present
        if (wcscmp(L"DebugObject", pObjectTypeInfo->TypeName.Buffer) == 0)
        {
            // Are there any objects?
            if (pObjectTypeInfo->TotalNumberOfObjects > 0)
            {
                VirtualFree(pMemory, 0, MEM_RELEASE);
                return true;
            }
            else
            {
                VirtualFree(pMemory, 0, MEM_RELEASE);
                return false;
            }
        }

        // Get the address of the current entries
        // string so we can find the end
        pObjInfoLocation =
            (unsigned char*)pObjectTypeInfo->TypeName.Buffer;

        // Add the size
        pObjInfoLocation +=
                pObjectTypeInfo->TypeName.Length;

        // Skip the trailing null and alignment bytes
        ULONG tmp = ((ULONG)pObjInfoLocation) & -4;

        // Not pretty but it works
        pObjInfoLocation = ((unsigned char*)tmp) +
                        sizeof(unsigned long);
    }

    VirtualFree(pMemory, 0, MEM_RELEASE);
    return true;
}
```

Anti-Dumping

"Dumping", a special term used in the reverse engineering realm, describes the process of taking an executable that has been protected and after the executable has been decrypted into memory, taking what is essentially a snapshot of the program and saving it onto disk, as shown by the following diagram. There are many techniques to prevent the dumping of an executable file that has been encrypted or compressed, and the following techniques are some of the more popular or better documented methods.


Encrypted(Compressed) File Contents
Decrypted(Decompressed) In Memory
Decrypted(Decompressed) Contents Saved On Disk
UnPackMe_UPX 3.00_dmped.exe

```
00422000  FF                  DB FF
00422001  EF                  DB EF
00422002  BE                  DB BE
00422003  FD                  DB FD
00422004  E8                  DB E8
00422005  0E                  DB 0E
```

```
00401000  E8 0E002B5C         CALL 5C6B1013
00401005  50                  PUSH EAX
```

Nanomites

I find this technique to be very interesting, and it works by replacing certain branch instructions, J*cc* instructions, within a program with `INT 3` breakpoints; information about the removed jumps is then stored in a heavily encrypted table. This information includes the destination of the jump, the required CPU flags, and the size of the jump (normally, either two or five bytes). Then, by using self-debugging, when one of these breakpoints is hit, the debugging process will handle the debug exception, and will look up certain information about the debugging break. This information is whether or not the breakpoint is a Nanomite or a real debug breakpoint, whether or not the jump should be taken (this includes comparing the `EFLAGS` registers appropriately for the jump type, i.e.., JNZ needs ZF = 0). If it is, then the address of the jump is retrieved and the execution in the debuggee will resume from there, and if it is not, then the length of the replaced jump is retrieved, and the size of that specific jump is skipped in the debuggee, and execution resumes. Now, to make things even worse, random `INT 3` instructions will be placed at unreachable parts of codes, and entries will be made in the Nanomite table. There are even entries that do not have a corresponding `INT 3` but are placed there to annoy reverse engineers. When used in the correct places inside an executable, this technique is very powerful, and has almost no impact on performance. Unfortunately, the source code for this technique is far too complicated for this article.

Stolen Bytes (Stolen Code)

This is a technique introduced by *ASprotect*, and can be entertaining to someone who has never encountered the method. In the stolen bytes routine, code or bytes from the original process protected by the packer are removed, often from the OEP (Original Entry Point), and are encrypted somewhere inside the packing code. The area where the bytes are is then replaced with code that will jump to a dynamically allocated buffer that contains the decrypted bytes that were "stolen" from the original code; this buffer also contains a jump back to the appropriate address of execution. More often than not, both the area were the bytes were removed from and the dynamically allocated buffer where the original bytes reside are filled with junk code and even more anti-reverse engineering techniques. This can be a powerful technique if the underlying concept is hidden from the reverse engineer; otherwise, it's not too hard to fix.

SizeOfImage

`SizeOfImage` is a field in `IMAGE_OPTION_HEADER` of a PE file, and by increasing the size of this field in the PEB at runtime, we can cause problems for tools that weren't developed to handle this problem. This method is easily applied to an application, and is easily defeated by reversing applications by enumerating all pages with the `MEM_IMAGE` flag, starting at the application's `ImageBase` page. This works because there cannot be a gap in the pages in memory. Here is some sample code:

C++

```cpp
// Any unreasonably large value will work say for example 0x100000 or 100,000h
void ChangeSizeOfImage(DWORD NewSize)
{
    __asm
    {
        mov eax, fs:[0x30] // PEB
        mov eax, [eax + 0x0c] // PEB_LDR_DATA
        mov eax, [eax + 0x0c] // InOrderModuleList
        mov dword ptr [eax + 0x20], NewSize // SizeOfImage
    }
}
```

Virtual Machines

Virtualization is considered the future of anti-reverse engineering, and has very much already made it into the present. Protectors like Themida and VMProtect already use virtual machines in their protection schemes. The simple use of virtual machines isn't the extent of the technique, however. Themida, for example, uses a technology that creates a unique virtual machine for **every** protected executable that utilizes the virtual machine protection. By implementing virtual machines this way, Themida prevents the use of a generic attack against its virtualization protection. Also, many protection schemes that utilize virtual machines often implement junk code instructions in their virtual machine byte code, much like junk code is inserted in native IA-32 code (Ferrie).

Guard Pages

Packers and protectors can utilize guard pages to implement, what is in essence, an on-demand decryption/decompression system. When the executable is loaded into memory instead of decompressing/decrypting the entire contents of the file at run-time, the protector will simply mark all pages that were not immediately needed as guard pages. After that is done, when another section of code or data is needed, an `EXCEPTION_GUARD_PAGE` (0x80000001) exception will be raised, and the data can be decrypted or decompressed either from file or from encrypted/compressed contents in memory.

This technique has been implemented in two ways, one by hooking `KiUserUserDispatcher` (Shrinker), and by using Self-Debugging (Armadillo's CopyMemoryII) (Ferrie). In the case of Shrinker, when an exception is raised, a check is made, from the hook placed on `KiUserUserDispatcher`, to find where the exception occurred and if the exception occurred in the process space of the protected executable; if it was, then the contents will be decompressed from the disk into the page where the program is either expecting data or executable code. Utilizing this technique can significantly reduce loading times and reduce memory usage for an executable file, because only pages that are needed are ever backed by physical memory (RAM), and only the pages that need to be used are decompressed and decrypted.

Armadillo also implements this technique under the name of CopyMemII, and it behaves in a similar fashion with the exception that it requires the use of self-debbugging; also, instead of having empty pages and loading the pages from disk, CopyMemII simply decompresses the pages into memory. Note that this does not decrypt the pages; therefore, the code and data is still secure. Then, when a page that has not been decrypted is accessed, an `EXCEPTION_GUARD_PAGE` (0x80000001) exception will be raised, and the process that is the debugger will catch the exception and decrypt the page as needed. There is, however, a weakness in Armadillo's implementation of the technique, and that is that once a page is decrypted, it will stay decrypted in memory. By exploiting this weakness, a reverse engineer could force the process to touch every page needed by the program and leave the entire program decrypted in memory and in perfect shape for a dump. In both implementations of this technique, the processes will only decrypt or decompress a page at a time; therefore, if an access spans more than one page, the protector will simply allow for the next exception to occur and decrypt that page. As a final note, if the protectors were to remember the last accessed page and were to discard or erase the last used page before decrypting the next page, then this technique would be extremely powerful.

Removing the Portable Executable Header

This is a simple anti-dumping technique that removes an executable's portable executable from memory at runtime; by doing this, a dumped image would be missing important information such as the RVA (Relative Virtual Address) of important tables (Reloc, Import, Export etc..), the entry point, and other information that the Windows loader needs to utilize when loading an image. One would want to be careful when utilizing this technique, because the Windows API or maybe legitimate external programs may need access to this information which has been removed.

C++

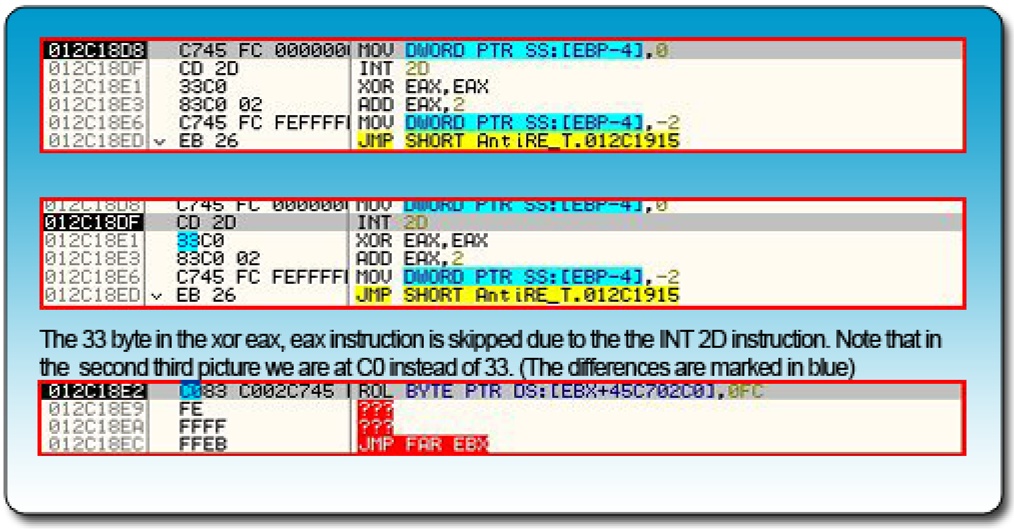
```cpp
// This function will erase the current images
// PE header from memory preventing a successful image
// if dumped
inline void ErasePEHeaderFromMemory()
{
    DWORD OldProtect = 0;

    // Get base address of module
    char *pBaseAddr = (char*)GetModuleHandle(NULL);

    // Change memory protection
    VirtualProtect(pBaseAddr, 4096, // Assume x86 page size
            PAGE_READWRITE, &OldProtect);

    // Erase the header
    ZeroMemory(pBaseAddr, 4096);
}
```

IA-32 Instruction Exploits

The following techniques take advantage of the problems debuggers have dealing with the IA-32 instructions. Most of these methods are low level techniques that aren't used very often.

Interrupt 2D

The `INT 2D` instruction can be used as a general purpose debugger detection method, because when executing the instruction, if no debugger is present, an exception will occur. However, if a debugger is present, no exception will occur, and things get interesting based on the debugger you are using. OllyDBG, as shown in the diagram, will actually skip a byte in its disassembly and will cause the analysis to go wrong. Visual Studio 2008's debugger handles the instruction without issues, and as for other debuggers, we would have to test ourselves.



```
012C18D8    C745 FC 000000( MOV DWORD PTR SS:[EBP-4],0
012C18DF    CD 2D           INT 2D
012C18E1    33C0            XOR EAX,EAX
012C18E3    83C0 02         ADD EAX,2
012C18E6    C745 FC FEFFFF( MOV DWORD PTR SS:[EBP-4],-2
012C18ED v  EB 26           JMP SHORT AntiRE_T.012C1915
```

```
012C18D8    C745 FC 000000( MOV DWORD PTR SS:[EBP-4],0
012C18DF    CD 2D           INT 2D
012C18E1    33C0            XOR EAX,EAX
012C18E3    83C0 02         ADD EAX,2
012C18E6    C745 FC FEFFFF( MOV DWORD PTR SS:[EBP-4],-2
012C18ED v  EB 26           JMP SHORT AntiRE_T.012C1915
```

The 33 byte in the xor eax, eax instruction is skipped due to the the INT 2D instruction. Note that in the second third picture we are at C0 instead of 33. (The differences are marked in blue)

```
012C18E2    C083 C002C745 ( ROL BYTE PTR DS:[EBX+45C702C0],0FC
012C18E9    FE              ???
012C18EA    FFFF            ???
012C18EC    FFEB            JMP FAR EBX
```

C++

```
// The Int2DCheck function will check to see if a debugger
// is attached to the current process. It does this by setting up
// SEH and using the Int 2D instruction which will only cause an
// exception if there is no debugger. Also when used in OllyDBG
// it will skip a byte in the disassembly and will create
// some havoc.
inline bool Int2DCheck()
{
    __try
    {
        __asm
        {
            int 0x2d
            xor eax, eax
            add eax, 2
        }
    }
    __except(EXCEPTION_EXECUTE_HANDLER)
    {
        return false;
    }

    return true;
}
```

Stack Segment

By manipulating the stack segment through the use of `push ss` and `pop ss`, we can cause the debugger to execute instructions unwillingly. In the following function, when stepping over the code with any debugger, the `mov eax, 9` line will execute, but will not be stepped on by the debugger.

C++

```
inline void PushPopSS()
{

    __asm
    {
        push ss
        pop ss
        mov eax, 9 // This line executes but is stepped over
        xor edx, edx // This is where the debugger will step to
    }
}
```

Instruction Prefixes

The following technique takes advantage of the way debuggers handle instruction prefixes. When stepping over this code in OllyDBG or in Visual Studio 2008, we will reach the first emit and immediately be taken to the end of the `__try` block. What happens is that the debugger essentially skips over the prefix and handles the `INT 1`. When running this code without a debugger, there will be an exception that SEH will catch and the program will continue along.

C++

```
// The IsDbgPresentPrefixCheck works in at least two debuggers
// OllyDBG and VS 2008, by utilizing the way the debuggers handle
// prefixes we can determine their presence. Specifically if this code
// is ran under a debugger it will simply be stepped over;
// however, if there is no debugger SEH will fire :D
inline bool IsDbgPresentPrefixCheck()
{
    __try
    {
        __asm __emit 0xF3 // 0xF3 0x64 disassembles as PREFIX REP:
        __asm __emit 0x64
        __asm __emit 0xF1 // One byte INT 1
    }
    __except(EXCEPTION_EXECUTE_HANDLER)
    {
        return false;
    }
    return true;
}
```

OllyDBG Specific

The following techniques can be used specifically to attack OllyDBG, which is probably the most used debugging tool on Windows at the moment. There are more techniques used to detect OllyDBG than what I show here, and probably for every technique discussed in this article, there is a plug-in for OllyDBG that fixes the issue. However, for inexperienced reverse engineers, these tricks can still work.

FindWindow

Using the Win32 API function `FindWindow`, we can check for the existence of OllyDBG's window class, `OLLYDBG`, and if it does exist, then most likely, OllyDBG is open and waiting to attach to a process, or is actively debugging the current process or another process.

```
HANDLE hOlly = FindWindow(TEXT("OLLYDBG"), NULL);

if(hOlly)
    ExitProcess(0);
```

OutputDebugString Exploit

In the world of exploits, there are many ways to exploit a program's security measures or lack thereof, and OllyDBG does have one. It's a format string exploit that has been patched by various custom versions of OllyDBG, but exists in the normal unmodified version which is the prevalent version of OllyDBG. The following code will crash OllyDBG if it is currently attached to the process, and is a very powerful technique.

```
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%s%s%s")
                TEXT("%s%s%s%s%s%s%s%s%s%s%s%s%s%s")
                TEXT("%s%s%s%s%s%s%s%s%s%s%s%s%s%s") );
```

WinDBG Specific

The following technique can be used to detect if WinDBG is running on the host machine. There hasn't been much research done on detecting WinDBG, because reverse engineers tend to favor other debuggers and analysis tools over WinDBG.

FindWindow

This is exactly the same technique as shown above in the OllyDBG example, except with a different window class and works the same way.

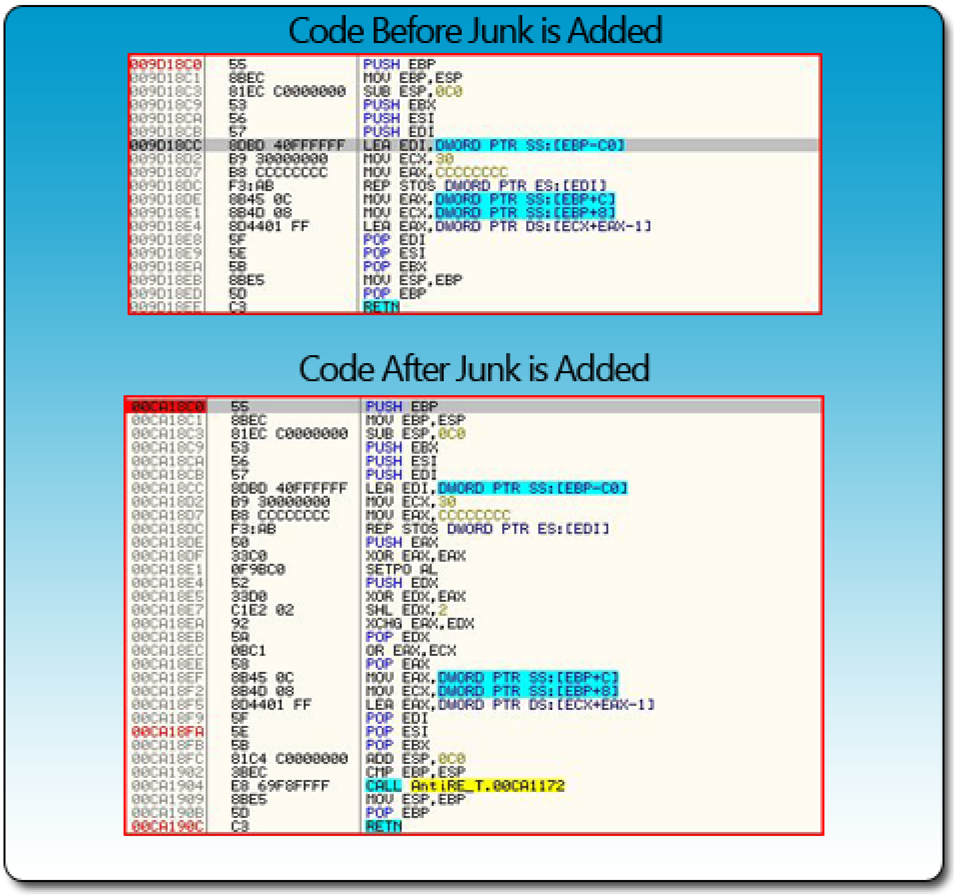
```
HANDLE hWinDbg = FindWindow(TEXT("WinDbgFrameClass"), NULL);

if(hWinDbg)
    ExitProcess(0);
```

Other Techniques

The following techniques didn't really fit into the other categories I covered in the previous section, and since they shared that in common, I'm putting them all into their own unique section.

Junk Code

Junk code is an aptly named technique of code obfuscation, and as its name suggests, it utilizes code that is junk or not needed to confuse a reverse engineer as to what the current code is actually trying to accomplish. When the junk code that is inserted into a routine is convincing and successfully manages to confuse a reverse engineer, then this technique can be rather effective; however, there is a performance penalty for utilizing this technique because the more instructions a routine or function contains, the longer the function will take to complete. Another issue utilizing junk code is that for memory and stack manipulation operations like `push`, `pop`, and `mov ptr []`, there is a decent chance for stack or memory corruption; therefore, these instructions are either placed and utilized carefully, or not used at all. Here is an example of a routine that adds two numbers and subtracts one, but has junk code added in.


Code Before Junk is Added

```
009D18C0   55              PUSH EBP
009D18C1   8BEC            MOV EBP,ESP
009D18C3   81EC C0000000   SUB ESP,0C0
009D18C9   53              PUSH EBX
009D18CA   56              PUSH ESI
009D18CB   57              PUSH EDI
009D18CC   8DBD 40FFFFFF   LEA EDI,DWORD PTR SS:[EBP-C0]
009D18D2   B9 30000000     MOV ECX,30
009D18D7   B8 CCCCCCCC     MOV EAX,CCCCCCCC
009D18DC   F3:AB           REP STOS DWORD PTR ES:[EDI]
009D18DE   8B45 0C         MOV EAX,DWORD PTR SS:[EBP+C]
009D18E1   8B4D 08         MOV ECX,DWORD PTR SS:[EBP+8]
009D18E4   8D4401 FF       LEA EAX,DWORD PTR DS:[ECX+EAX-1]
009D18E8   5F              POP EDI
009D18E9   5E              POP ESI
009D18EA   5B              POP EBX
009D18EB   8BE5            MOV ESP,EBP
009D18ED   5D              POP EBP
009D18EE   C3              RETN
```

Code After Junk is Added

```
00CA18C0   55              PUSH EBP
00CA18C1   8BEC            MOV EBP,ESP
00CA18C3   81EC C0000000   SUB ESP,0C0
00CA18C9   53              PUSH EBX
00CA18CA   56              PUSH ESI
00CA18CB   57              PUSH EDI
00CA18CC   8DBD 40FFFFFF   LEA EDI,DWORD PTR SS:[EBP-C0]
00CA18D2   B9 30000000     MOV ECX,30
00CA18D7   B8 CCCCCCCC     MOV EAX,CCCCCCCC
00CA18DC   F3:AB           REP STOS DWORD PTR ES:[EDI]
00CA18DE   50              PUSH EAX
00CA18DF   33C0            XOR EAX,EAX
00CA18E1   0F98C0          SETPO AL
00CA18E4   52              PUSH EDX
00CA18E5   33D0            XOR EDX,EAX
00CA18E7   C1E2 02         SHL EDX,2
00CA18EA   92              XCHG EAX,EDX
00CA18EB   5A              POP EDX
00CA18EC   0BC1            OR EAX,ECX
00CA18EE   58              POP EAX
00CA18EF   8B45 0C         MOV EAX,DWORD PTR SS:[EBP+C]
00CA18F2   8B4D 08         MOV ECX,DWORD PTR SS:[EBP+8]
00CA18F5   8D4401 FF       LEA EAX,DWORD PTR DS:[ECX+EAX-1]
00CA18F9   5F              POP EDI
00CA18FA   5E              POP ESI
00CA18FB   5B              POP EBX
00CA18FC   81C4 C0000000   ADD ESP,0C0
00CA1902   3BEC            CMP EBP,ESP
00CA1904   E8 69F8FFFF     CALL AntiRE_T.00CA1172
00CA1909   8BE5            MOV ESP,EBP
00CA190B   5D              POP EBP
00CA190C   C3              RETN
```

C++

```cpp
#define JUNK_CODE_ONE        \
    __asm{push eax}            \
    __asm{xor eax, eax}        \
    __asm{setpo al}            \
    __asm{push edx}            \
    __asm{xor edx, eax}        \
    __asm{sal edx, 2}        \
    __asm{xchg eax, edx}    \
    __asm{pop edx}            \
    __asm{or eax, ecx}        \
    __asm{pop eax}

inline int AddSubOne(int One, int Two)
{
    JUNK_CODE_ONE
    return ( (One + Two) - 1 );
}
```

As we can see, this routine does a lot of nothing, and only the final two instructions actually accomplish the goal of the function.

Native Code Permutations

Permutations is defined as "often major or fundamental change (as in character or condition) based primarily on rearrangement of existent elements", which when referring to the world of code means different ways of accomplishing the same goal or task ("permutation"). For those unfamiliar with the concept of permutation, I'm going to explain it first with numbers, and then we'll explore the concept with code.

Permutations of the set {1,5,9} would be:

```
159
195
519
591
915
951
```

When we do permutation of an item or object, we're simply trying to represent the same information or actions in a different way. Now, we'll do the permutation of the `mov m32, imm` instruction:

ASM

```
Original:
mov [mem addr], 7 (mov m32, imm)

Permutation 1:
push 7
pop [mem addr]

Permutation 2:
mov eax, mem addr
mov [eax], 7

Permutation 3:
mov edi, mem addr
mov eax, 7
stosd

Permutation 4:
push mem addr
pop edi
push 7
pop eax
stosd

And on....
```

As we can imagine, this can be a very powerful and flexible method of obfuscation and confusion, especially considering that a lot of people are simply following tutorials when reversing, and even small changes will completely prevent them from reversing your application. Then, when this concept is applied to an executable at runtime, and if at every run the program is permutated and morphed, we can achieve metamorphic code which, along with virtual machines, is the cream of the crop of anti-reverse engineering techniques. However, this technique is very hard to implement correctly, considering an excellent disassembly engine is required, and for those who are familiar with the hell of creating an accurate disassembler, this is quite a task.

Citations

- "Creating Guard Pages". Microsoft Developer Network. Microsoft. 19 Oct 2008.
- Falliere, Nicolas. "Windows Anti-Debug Reference". SecurityFocus. 9 Nov 2008.

- Intel Corporation, "Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide", Intel Corporation, 2008.
- Microsoft Corporation, "BlockInput Function". Microsoft Developer Network. Microsoft Corporation. 22 Oct 2008.
- Microsoft Corporation, "How To Use the SeDebugPrivilege to Acquire Any Process Handle". Microsoft Help and Support. Microsoft Corporation. 21 Oct 2008.OpenRCE Anti Reverse Engineering Techniques Database". OpenRCE.org. 9 Nov 2008.
- OpenRCE, "
- "Permutation". Merriam-Webster Online Dictionary. Merriam-Webster Online. 1 November 2008.
- Peter, Ferrie. Home Page of Peter Ferrie. 26 Oct 2008.

License

Written By

Joshua Tully

Student

United States

This member has not yet provided a Biography. Assume it's interesting and varied, and probably something to do with programming.