FUTo

Peter Silberman, C.H.A.O.S.

12/2005

Contents

Chapter 1

Foreword

Abstract:

Since the introduction of FU[2], the rootkit world has moved away from implementing system hooks to hide their presence. Because of this change in offense, a new defense had to be developed. The new algorithms used by rootkit detectors, such as BlackLight[1], attempt to find what the rootkit is hiding instead of simply detecting the presence of the rootkits hooks. This paper will discuss an algorithm that is used by both Blacklight and IceSword[3] to detect hidden processes. This paper will also document current weaknesses in the rootkit detection field and introduce a more complete stealth technique implemented as a prototype in FUTo.

Thanks:

Peter: would like to thank bugcheck, skape, thief, pedram, F-Secure for doing great research, and all the nologin/research'ers who encourage mind growth.

C.H.A.O.S.: would like to thank Amy, Santa (this work was three hours on Christmas day), lonerancher, Pedram, valerino, and HBG Unit.

Chapter 2

Introduction

In the past year or two, there have been several major developments in the rootkit world. Recent milestones include the introduction of the FU rootkit, which uses Direct Kernel Object Manipulation (DKOM); the introduction of VICE, one of the first rootkit detection programs; the birth of Sysinternals Rootkit Revealer and F-Secures Blacklight, the first mainstream Windows rootkit detection tools; and most recently the introduction of Shadow Walker, a rootkit that hooks the memory manager to hide in plain sight.

Enter Blacklight and IceSword. The authors chose to investigate the algorithms used by both Blacklight and IceSword because they are considered by many in the field to be the best detection tools. Blacklight, developed by the Finnish security company F-Secure, is primarily concerned with detecting hidden processes. It does not attempt to detect system hooks; it is only concerned with hidden processes. IceSword uses a very similar method to Blacklight. IceSword differentiates itself from Blacklight in that it is a more robust tool allowing the user to see what system calls are hooked, what drivers are hidden, and what TCP/UDP ports are open that programs, such as netstat, do not.

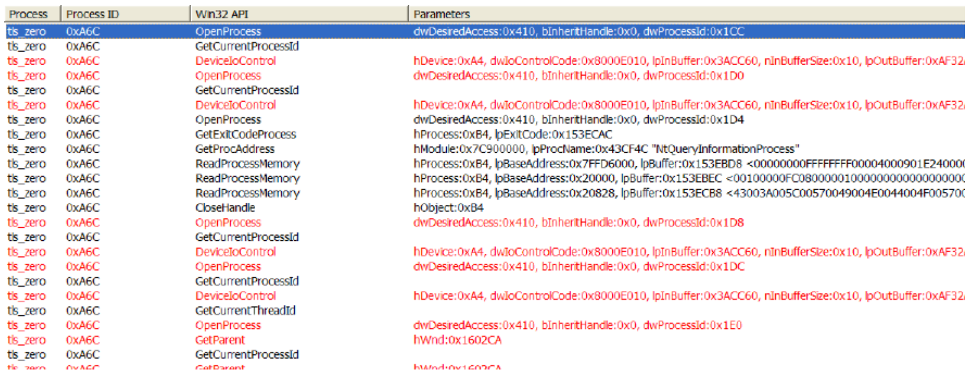
Chapter 3

Blacklight

This paper will focus primarily on Blacklight due to its algorithm being the research focus for this paper. Also, it became apparent after researching Blacklight that IceSword used a very similiar algorithm. Therefore, if a weakness was found in Blacklight, it would most likely exist in IceSword as well.

Blacklight takes a userland approach to detecting processes. Although simplistic, its algorithm is amazingly effective. Blacklight uses some very strong anti-debugging features that begin by creating a Thread Local Storage (TLS) callback table. Blacklights TLS callback attempts to befuddle debuggers by forking the main process before the process object is fully created. This can occur because the TLS callback routine is called before the process is completely initialized. Blacklight also has anti-debugging measures that detect the presence of debuggers attaching to it. Rather than attempting to beat the anti-debugging measures by circumventing the TLS callback and making other program modifications, the authors decided to just disable the TLS routine. To do this, the authors used a tool called LordPE[4]. LordPE allows users to edit PE files. The authors used this tool to zero out the TLS callback table. This disabled the forking routine and gave the authors the ability to use an API Monitor. It should be noted that disabling the callback routine would allow you to attach a debugger, but when the user clicked "scan" in the Blacklight GUI Blacklight would detect the debugger and exit. Instead of working up a second measure to circumvent the anti-debugging routines, the authors decided to analyze the calls occuring within Blacklight. To this end, the authors used Rohitabs API Monitor[6]. Figure 3 shows the API calls made when Blacklight is searching for hidden processes.

In figure 3, notice the failed calls to the API *OpenProcess* (tls zero is Blacklight without a TLS table). Blacklight tries opening a process with process id (PID) of 0x1CC, 0x1D0, 0x1D4, 0x1D8 and so on. The authors dubbed the method



Process	Process ID	Win32 API	Parameters
tls_zero	0xA6C	OpenProcess	dwDesiredAccess:0x410, bInheritHandle:0x0, dwProcessId:0x1CC
tls_zero	0xA6C	GetCurrentProcessId	
tls_zero	0xA6C	DeviceIoControl	hDevice:0xA4, dwIoControlCode:0x8000E010, lpInBuffer:0x3ACC60, nInBufferSize:0x10, lpOutBuffer:0xAF32
tls_zero	0xA6C	OpenProcess	dwDesiredAccess:0x410, bInheritHandle:0x0, dwProcessId:0x1D0
tls_zero	0xA6C	GetCurrentProcessId	
tls_zero	0xA6C	DeviceIoControl	hDevice:0xA4, dwIoControlCode:0x8000E010, lpInBuffer:0x3ACC60, nInBufferSize:0x10, lpOutBuffer:0xAF32
tls_zero	0xA6C	OpenProcess	dwDesiredAccess:0x410, bInheritHandle:0x0, dwProcessId:0x1D4
tls_zero	0xA6C	GetExitCodeProcess	hProcess:0xB4, lpExitCode:0x153ECAC
tls_zero	0xA6C	GetProcAddress	hModule:0x7C900000, lpProcName:0x43CF4C "NtQueryInformationProcess"
tls_zero	0xA6C	ReadProcessMemory	hProcess:0xB4, lpBaseAddress:0x7FFD6000, lpBuffer:0x153EBD8 <00000000FFFFFFFF00004000901E24000
tls_zero	0xA6C	ReadProcessMemory	hProcess:0xB4, lpBaseAddress:0x20000, lpBuffer:0x153EBEC <00100000FC080000010000000000000000
tls_zero	0xA6C	ReadProcessMemory	hProcess:0xB4, lpBaseAddress:0x20828, lpBuffer:0x153ECB8 <43003A005C00570049004E0044004F00570
tls_zero	0xA6C	CloseHandle	hObject:0xB4
tls_zero	0xA6C	OpenProcess	dwDesiredAccess:0x410, bInheritHandle:0x0, dwProcessId:0x1D8
tls_zero	0xA6C	GetCurrentProcessId	
tls_zero	0xA6C	DeviceIoControl	hDevice:0xA4, dwIoControlCode:0x8000E010, lpInBuffer:0x3ACC60, nInBufferSize:0x10, lpOutBuffer:0xAF32
tls_zero	0xA6C	OpenProcess	dwDesiredAccess:0x410, bInheritHandle:0x0, dwProcessId:0x1DC
tls_zero	0xA6C	GetCurrentProcessId	
tls_zero	0xA6C	DeviceIoControl	hDevice:0xA4, dwIoControlCode:0x8000E010, lpInBuffer:0x3ACC60, nInBufferSize:0x10, lpOutBuffer:0xAF32
tls_zero	0xA6C	GetCurrentThreadId	
tls_zero	0xA6C	OpenProcess	dwDesiredAccess:0x410, bInheritHandle:0x0, dwProcessId:0x1E0
tls_zero	0xA6C	GetParent	hWnd:0x1602CA
tls_zero	0xA6C	GetCurrentProcessId	
tls_zero	0xA6C	GetParent	hWnd:0x1602CA

Figure 3.1: Output of Blacklight API calls

Blacklight uses as PID Bruteforce (PIDB). Blacklight loops through all possible PIDS calling *OpenProcess* on the PIDs in the range of 0x0 to 0x4E1C. Blacklight keeps a list of all processes it is able to open, using the PIDB method. Blacklight then calls *CreateToolhelp32Snapshot*, which gives Blacklight a second list of processes. Blacklight then compares the two lists, to see if there are any processes in the PIDB list that are not in the list returned by the *CreateToolhelp32Snapshot* function. If there is any discrepancy, these processes are considered hidden and reported to the user.

3.1 Windows OpenProcess

In Windows, the *OpenProcess* function is a wrapper to the *NtOpenProcess* routine. *NtOpenProcess* is implemented in the kernel by NTOSKRNL.EXE. The function prototype for NtOpenProcess is:

```
NTSTATUS NtOpenProcess (
    OUT PHANDLE ProcessHandle,
    IN ACCESS_MASK DesiredAccess,
    IN POBJECT_ATTRIBUTES ObjectAttributes,
    IN PCLIENT_ID ClientId OPTIONAL)
```

The ClientId parameter is the actual PID that is passed by *OpenProcess*. This parameter is optional, but during our observation the OpenProcess function always specified a ClientId when calling NtOpenProcess.

NtOpenProcess performs three primary functions:

1. It verifies the process exists by calling *PsLookupProcessByProcessId*.

2. It attempts to open a handle to the process by calling *ObOpenObjectByPointer*.

3. If it was successful opening a handle to the process, it passes the handle back to the caller.

PsLookupProcessByProcessId was the next obvious place for research. One of the outstanding questions was how does *PsLookupProcessByProcessId* know that a given PID is part of a valid process? The answer becomes clear in the first few lines of the disassembly:

```
PsLookupProcessByProcessId:
mov edi, edi
push ebp
mov ebp, esp
push ebx
push esi
mov eax, large fs:124h
push [ebp+arg_4]
mov esi, eax
dec dword ptr [esi+0D4h]
push PspCidTable
call ExMapHandleToPointer
```

From the above disassembly, it is clear that *ExMapHandleToPointer* queries the PspCidTable for the process ID.

Now we have a complete picture of how Blacklight detects hidden processes:

1. Blacklight starts looping through the range of valid process IDs, 0 through 0x41DC.

2. Blacklight calls *OpenProcess* on every possible PID.

3. *OpenProcess* calls *NtOpenProcess*.

4. *NtOpenProcess* calls *PsLookupProcessByProcessId* to verify the process exists.

5. *PsLookupProcessByProcessId* uses the PspCidTable to verify the processes exists.

6. *NtOpenProcess* calls *ObOpenObjectByPointer* to get the handle to the process.

7. If *OpenProcess* was successful, Blacklight stores the information about the process and continues to loop.

8. Once the process list has been created by exhausting all possible PIDs. Blacklight compares the PIDB list with the list it creates by calling *CreateToolhelp32Snapshot*. *CreateToolhelp32Snapshot* is a Win32 API that takes a snapshot of all running processes on the system. A discrepancy between the two lists implies that there is a hidden process. This case is reported by Blacklight.

3.2 The PspCidTable

The PspCidTable is a "handle table for process and thread client IDs"[7]. Every process' PID corresponds to its location in the PspCidTable. The PspCidTable is a pointer to a HANDLE_TABLE structure.

```
typedef struct _HANDLE_TABLE {
   PVOID        p_hTable;
   PEPROCESS    QuotaProcess;
   PVOID        UniqueProcessId;
   EX_PUSH_LOCK HandleTableLock [4];
   LIST_ENTRY   HandleTableList;
   EX_PUSH_LOCK HandleContentionEvent;
   PHANDLE_TRACE_DEBUG_INFO DebugInfo;
   DWORD        ExtraInfoPages;
   DWORD        FirstFree;
   DWORD        LastFree;
   DWORD        NextHandleNeedingPool;
   DWORD        HandleCount;
   DWORD        Flags;
}
```

Windows offers a variety of non-exported functions to manipulate and retrieve information from the PspCidTable. These include:

ExCreateHandleTable creates non-process handle tables. The objects within all handle tables except the PspCidTable are pointers to object headers and not the address of the objects themselves.

ExDupHandleTable is called when spawning a process.

ExSweepHandleTable is used for process rundown.

ExDestroyHandleTable is called when a process is exiting.

ExCreateHandle creates new handle table entries.

ExChangeHandle is used to change the access mask on a handle.

ExDestroyHandle implements the functionality of CloseHandle.

ExMapHandleToPointer returns the address of the object corresponding to the handle.

ExReferenceHandleDebugIn tracing handles.

ExSnapShotHandleTables is used for handle searchers (for example in oh.exe).

Below is code that uses non-exported functions to remove a process object from the PspCidTable. It uses hardcoded addresses for the non-exported functions necessary; however, a rootkit could find these function addresses dynamically.

```
typedef PHANDLE_TABLE_ENTRY (*ExMapHandleToPointerFUNC)
                     ( IN PHANDLE_TABLE HandleTable,
                       IN HANDLE ProcessId);

void HideFromBlacklight(DWORD eproc)
{
               PHANDLE_TABLE_ENTRY CidEntry;
               ExMapHandleToPointerFUNC map;
               ExUnlockHandleTableEntryFUNC umap;
               PEPROCESS p;
               CLIENT_ID ClientId;

               map = (ExMapHandleToPointerFUNC)0x80493285;

               CidEntry = map((PHANDLE_TABLE)0x8188d7c8,
                         LongToHandle( *((DWORD*)(eproc+PIDOFFSET)) ) );
               if(CidEntry != NULL)
               {
                  CidEntry->Object = 0;
               }
               return;
}
```

Since the job of the PspCidTable is to keep track of all the processes and threads, it is logical that a rootkit detector could use the PspCidTable to find hidden processes. However, relying on a single data structure is not a very robust algorithm. If a rootkit alters this one data structure, the operating system and other programs will have no idea that the hidden process exists. New rootkit detection algorithms should be devised that have overlapping dependencies so that a single change will not go undetected.

Chapter 4

FUTo

To demonstrate the weaknesses in the algorithms currently used by rootkit detection software such as Blacklight and Icesword, the authors have created FUTo. FUTo is a new version of the FU rootkit. FUTo has the added ability to manipulate the PspCidTable without using any function calls. It uses DKOM techniques to hide particular objects within the PspCidTable.

There were some design considerations when implementing the new features in FUTo. The first was that, like the ExMapHandleXXX functions, the PspCidTable is not exported by the kernel. In order to overcome this, FUTo automatically detects the PspCidTable by finding the PsLookupProcessByProcessId function and disassembling it looking for the first function call. At the time of this writing, the first function call is always to ExMapHandleToPointer. ExMapHandleToPointer takes the PspCidTable as its first parameter. Using this knowledge, it is fairly straightforward to find the PspCidTable.

```
PsLookupProcessByProcessId:
mov edi, edi
push ebp
mov ebp, esp
push ebx
push esi
mov eax, large fs:124h
push [ebp+arg_4]
mov esi, eax
dec dword ptr [esi+0D4h]
push PspCidTable
call ExMapHandleToPointer
```

A more robust method to find the PspCidTable could be written as this algorithm will fail if even simple compiler optimizations are made on the kernel. Opc0de wrote a more robust method to detect non-exported variables like PspCidTable, PspActiveProcessHead, PspLoadedModuleList, etc. Opc0des method does not requires memory scanning like the method currently used in FUTo. Instead Opc0de found that the *KdVersionBlock* field in the Process Control Region structure pointed to a structure KDDEBUGGER_DATA32[5]. The structure looks like this:

```
typedef struct _KDDEBUGGER_DATA32 {

    DBGKD_DEBUG_DATA_HEADER32 Header;
    ULONG   KernBase;
    ULONG   BreakpointWithStatus;       // address of breakpoint
    ULONG   SavedContext;
    USHORT  ThCallbackStack;            // offset in thread data
    USHORT  NextCallback;               // saved pointer to next callback frame
    USHORT  FramePointer;               // saved frame pointer
    USHORT  PaeEnabled:1;
    ULONG   KiCallUserMode;             // kernel routine
    ULONG   KeUserCallbackDispatcher;   // address in ntdll

    ULONG   PsLoadedModuleList;
    ULONG   PsActiveProcessHead;
    ULONG   PspCidTable;

    ULONG   ExpSystemResourcesList;
    ULONG   ExpPagedPoolDescriptor;
    ULONG   ExpNumberOfPagedPools;

    [...]

    ULONG   KdPrintCircularBuffer;
    ULONG   KdPrintCircularBufferEnd;
    ULONG   KdPrintWritePointer;
    ULONG   KdPrintRolloverCount;

    ULONG   MmLoadedUserImageList;

} KDDEBUGGER_DATA32, *PKDDEBUGGER_DATA32;
```

As the reader can see the structure contains pointers to many of the commonly needed/used non-exported variables. This is one more robust method to finding the PspCidTable and other variables like it.

The second design consideration was a little more troubling. When FUTo removes an object from the PspCidTable, the HANDLE_ENTRY is replaced with NULLs representing the fact that the process "does not exist." The problem then occurs when the process that is hidden (and has no PspCidTable entries) is closed. When the system tries to close the process, it will index into the PspCidTable and dereference a null object causing a blue screen. The solution to this problem is simple but not elegant. First, FUTo sets up a process notify routine by calling PsSetCreateProcessNotifyRoutine. The callback function will be invoked whenever a process is created, but more importantly it will be called whenever a process is deleted. The callback executes before the hidden process is terminated; therefore, it gets called before the system crashes. When FUTo deletes the indexes that contain objects that point to the rogue process, FUTo will save the value of the HANDLE_ENTRYs and the index for later use. When the process is closed, FUTo will restore the objects before the process is closed allowing the system to dereference valid objects.

Chapter 5

Conclusion

The catch phrase in 2005 was, We are raising the bar [again] for rootkit detection. Hopefully the reader has walked away with a better understanding of how the top rootkit detection programs are detecting hidden processes and how they can be improved. Some readers may ask ”What can I do?” Well, the simple solution is not to connect to the Internet, but a combination of using both Blacklight, IceSword and Rootkit Revealer will greatly help your chances of staying rootkit free. A new tool called RAIDE (Rootkit Analysis Identification Elimination) will be unveiled in the coming months at Blackhat Amsterdam[8]. This new tool does not suffer from the problems brought forth here.

Bibliography

[1] Blacklight Homepage. *F-Secure Blacklight* http://www.f-secure.com/blacklight/

[2] FU Project Page. *FU* http://www.rootkit.com/project.php?id=12

[3] IceSword Homepage. *IceSword* http://www.xfocus.net/tools/200505/1032.html

[4] LordPE Homepage. *LordPE Info* http://mitglied.lycos.de/yoda2k/LordPE/info.htm

[5] Opc0de. 2005. *How to get some hidden kernel variables without scanning* http://www.rootkit.com/newsread.php?newsid=101

[6] Rohitabs API Monitor. *API Monitor - Spy on API calls* http://www.rohitab.com/apimonitor/

[7] Russinovich, Solomon. *Microsoft Windows Internals Fourth Edition.*

[8] Silberman. *RAIDE:Rootkit Analysis Identification Elimination* http://www.blackhat.com/html/bh-europe-06/bh-eu-06-speakers.html#Silberman