Raytheon
Blackbird Technologies

Direct Kernel Object Manipulasiton (DKOM) Proof-of-Concept (PoC) Outline

For

SIRIUS Task Order PIQUE

Submitted to:

U.S. Government

Submitted by:

Raytheon Blackbird Technologies, Inc.
13900 Lincoln Park Drive
Suite 400
Herndon, VA 20171

21 November 2014

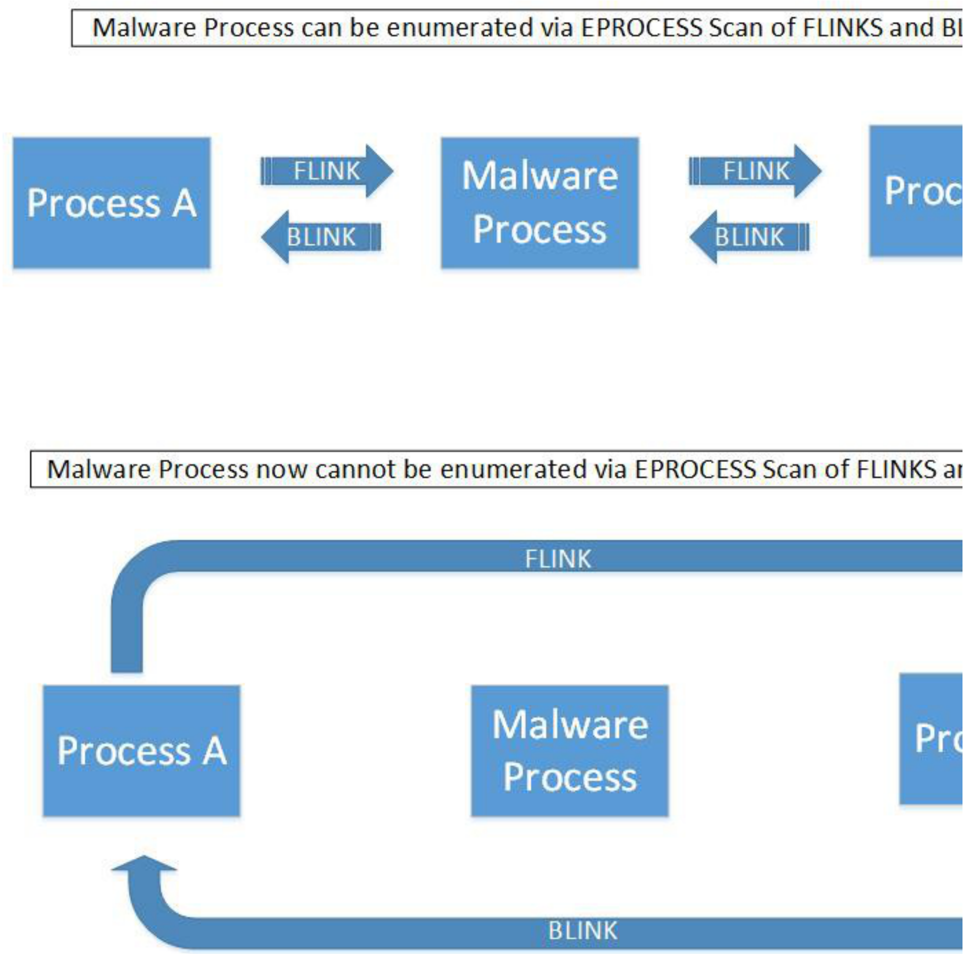
(U) Table of Contents

(U) List of Figures

(U) List of Tables

(U) Executive Summary

(U) Direct Kernel Object Manipulation (DKOM) is a rootkit technique for hiding processes, drivers, and files from the system task manager and event scheduler. Process hiding via DKOM is accomplished by modifying the doubly linked list of active threads and processes so that forward and backward pointers (FLINK and BLINK) of items adjacent to the process so that they "point around" the process to be hidden. The task manager and event scheduler use EPROCESS, which relies on enumeration of the FLINKs and BLINKs to identify running processes, and if the FLINKs and BLINKs are modified processes become "hidden" from the task manager and event scheduler in **Figure 1**.



UNCLASSIFIED

Figure 1. (U) Hiding a Process by Modifying FLINK and BLINK

(U) There are two methods of performing DKOM:

Load a kernel driver

Use the ZwSystemDebugControl() application programming interface (API) from user-mode

(U) Naturally, the preferred approach to a DKOM PoC is via user-mode API calls to ZwSystemDebugControl() as it obviates the need to install drivers on target.

(U) Description of the PoC Coding Approach

(U) We will write the DKOM PoC in C++ using Visual Studio 2013 using standard Microsoft Windows APIs and libraries. We will write a user-mode application that will perform the following:

Call SeDebugPrivilege() to enable calls to ZwSystemDebugControl()

Locate the base address of the kernel module via ZwQuerySystemInformation(SystemModuleInformation) similar to the proof-of-concept (PoC) code listed in **Figure 2**.

```
PVOID
KernelGetModuleBase(
    PCHAR  pModuleName
    )
{
    PVOID pModuleBase = NULL;
    PULONG pSystemInfoBuffer = NULL;

    __try
    {
        NTSTATUS status = STATUS_INSUFFICIENT_RESOURCES;
        ULONG    SystemInfoBufferSize = 0;

        status = ZwQuerySystemInformation(SystemModuleIn
            &SystemInfoBufferSize,
            0,
            &SystemInfoBufferSize);

        if (!SystemInfoBufferSize)
            return NULL;

        pSystemInfoBuffer = (PULONG)ExAllocatePool(NonPa

        if (!pSystemInfoBuffer)
            return NULL;

        memset(pSystemInfoBuffer, 0, SystemInfoBufferSiz

        status = ZwQuerySystemInformation(SystemModuleIn
            pSystemInfoBuffer,
            SystemInfoBufferSize*2,
            &SystemInfoBufferSize);

        if (NT_SUCCESS(status))
        {
            PSYSTEM_MODULE_ENTRY pSysModuleEntry =
                ((PSYSTEM_MODULE_INFORMATION)(pSystemInf
            ULONG i;
            for (i = 0; i <((PSYSTEM_MODULE_INFORMATION)
            {
                if (_stricmp(pSysModuleEntry[i].ModuleNa
                             pSysModuleEntry[i].ModuleNa
                {
                    pModuleBase = pSysModuleEntry[i].Mod
                    break;
                }
            }
        }

    }
    __except(EXCEPTION_EXECUTE_HANDLER)
    {
        pModuleBase = NULL;
    }
    if(pSystemInfoBuffer) {
        ExFreePool(pSystemInfoBuffer);
    }

    return pModuleBase;
} // end KernelGetModuleBase()
```

UNCLASSIFIED

Figure 2. (U) Locate Base Address of the Kernel Module

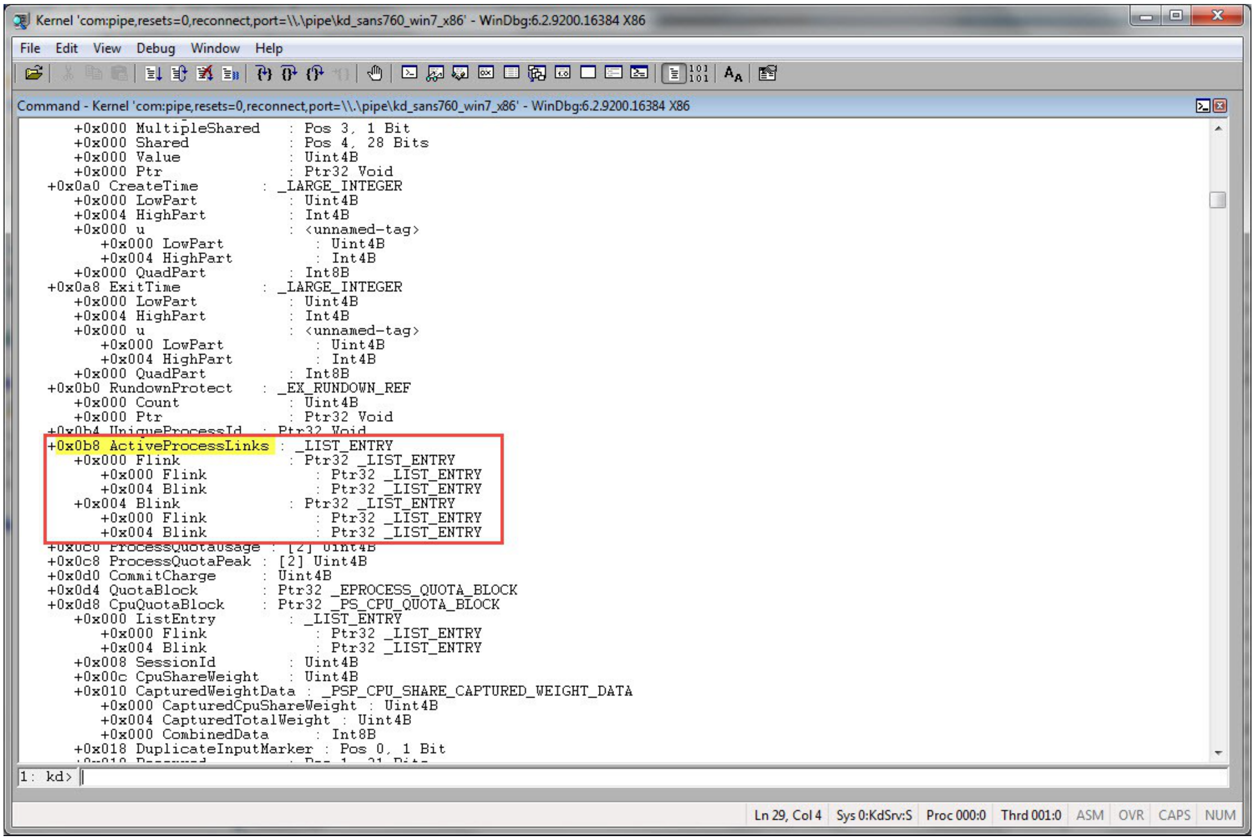
Find PsInitialSystemProcess

Walk the linked list of Executive Process (_EPROCESS) objects until it finds a process ID (PID) matching the process to hide, which will be obtained via GetCurrentProcessId(). We will use the appropriate offset in the _EPROCESS structure for the ActiveProcessLinks substructure to locate the FLINK and BLINK. For example, the offset to the ActiveProcessLinks for Windows 7 32-bit and Windows 8 64-bit are in **Table 1** and shown in the Windows windbg screen capture in **Figures 3** and **4**.

Table 1. (U) Offsets to ActiveProcessLinks

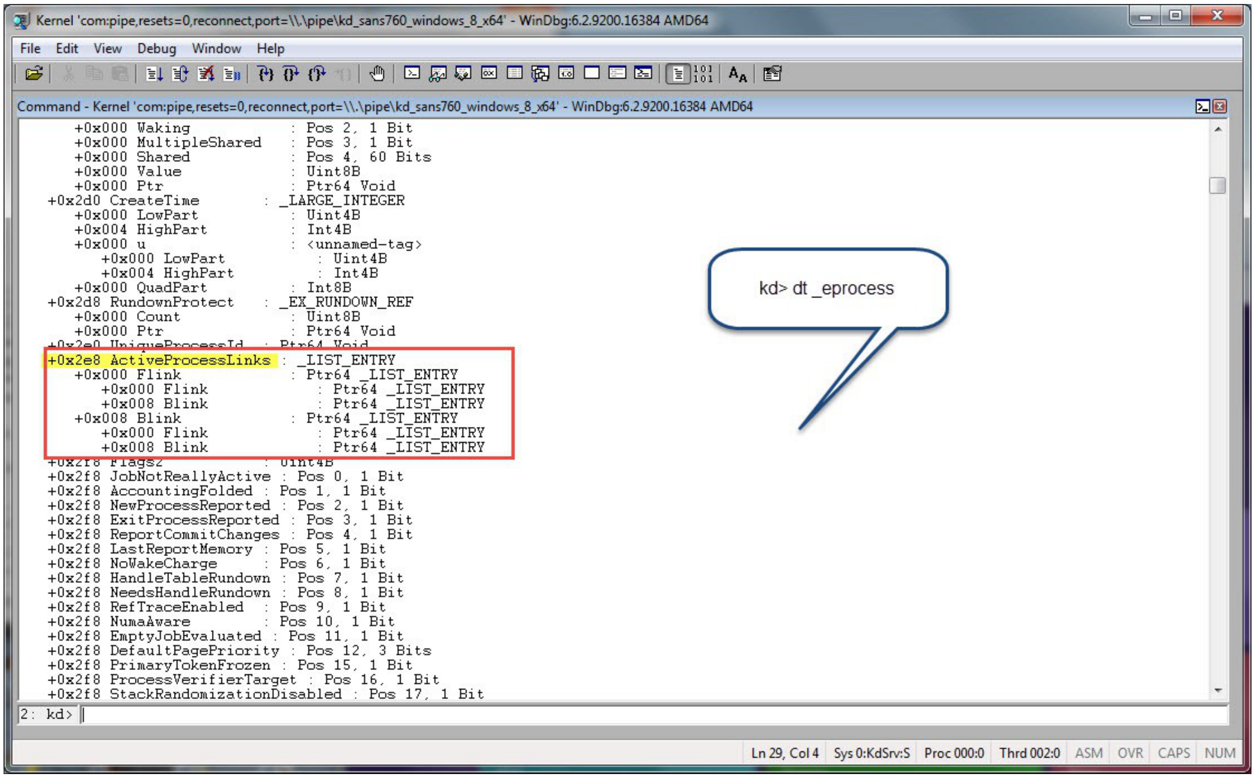
Windows 7 32-bit	0x0b8
Windows 8 64-bit	0x2e8

UNCLASSIFIED



UNCLASSIFIED

Figure 3. (U) Windows 7 32-bit – Offset to ActiveProcessLinks



UNCLASSIFIED

Figure 4. (U) Windows 8 64-bit – Offset to ActiveProcessLinks

We will then call WriteKernelMemory(), which is a wrapper function for ZwSystemDebugControl(), to modify the FLINK and BLINK to effectively hide the target process.

(U) We plan to write PoCs for both 32-bit and 64-bit versions of Windows. There are some code listings in "The Art of Memory Forensics" that were apparently generated by an IDA Pro examination of a malware sample that implements DKOM to hide itself (Prolaco) and decompiled with Hex-Rays decompiler. We will take as much as we can from the Prolaco decompiled code listing in the "Art of Memory Forensics" to enlighten our development of the PoC.

(U) Conclusion

(U) The DKOM PoC appears to be straightforward and presents low to moderate risk due to complexity. This PoC should provide an effective and robust process hiding capability. However, there are known techniques for discovering this type of DKOM-based hiding method. The code listing in **Figure 5** will detect DKOM-based process hiding.

```
NTSTATUS
ReadKernelMemory(IN PVOID BaseAddress,
OUT PVOID Buffer,
IN ULONG Length)
{
NTSTATUS Status;
SYSDBG_VIRTUAL DbgMemory;

//
// Setup the request
//
DbgMemory.Address = BaseAddress;
DbgMemory.Buffer = Buffer;
DbgMemory.Request = Length;

//
// Do the read
//
Status = NtSystemDebugControl(SysDbgReadVirtual,
&DbgMemory,
sizeof(DbgMemory),
NULL,
0,
NULL);
return Status;
}

PCHAR
FindDriverForAddress(IN PVOID Pointer)
{
NTSTATUS Status;
PRTL_PROCESS_MODULES ModuleInfo;
PRTL_PROCESS_MODULE_INFORMATION ModuleEntry;
ULONG ReturnedLength;
ULONG i;

//
// Figure out how much size we need
//
Status = NtQuerySystemInformation(SystemModuleInformation,
NULL,
0,
&ReturnedLength);
if (Status != STATUS_INFO_LENGTH_MISMATCH) return NULL;

//
```

```
// Allocate a buffer large enough
//
ModuleInfo = RtlAllocateHeap(RtlGetProcessHeap(), 0, ReturnedLength);
if (!ModuleInfo) return NULL;

//
// Now query the data again
//
Status = NtQuerySystemInformation(SystemModuleInformation,
ModuleInfo,
ReturnedLength,
&ReturnedLength);
if (!NT_SUCCESS(Status)) return NULL;

//
// Loop all the drivers
//
for (i = 0; i < ModuleInfo->NumberOfModules; i++)
{
//
// Get the current entry and check if the pointer is within it
//
ModuleEntry = &ModuleInfo->Modules[i];
if ((Pointer > ModuleEntry->ImageBase) &&
(Pointer < ((PVOID)((ULONG_PTR)ModuleEntry->ImageBase +
ModuleEntry->ImageSize))))
{
//
// Found a match, return it
//
return ModuleEntry->FullPathName;
}
}
}

PCHAR
DetectDriver(VOID)
{
BOOLEAN Old;
NTSTATUS Status;
ULONG_PTR MappedAddress;
PVOID KernelBase, TableBase;
UNICODE_STRING KernelName;
ANSI_STRING TableName = RTL_CONSTANT_STRING("KeServiceDescript
RTL_PROCESS_MODULES ModuleInfo;
ULONG Flags;
```

```
KSERVICE_TABLE_DESCRIPTOR ServiceTable;

//
// Give our thread the debug privilege
//
Status = RtlAdjustPrivilege(SE_DEBUG_PRIVILEGE, TRUE, FALSE, &Old);
if (!NT_SUCCESS(Status)) return NULL;

//
// Query the kernel's module entry
//
Status = NtQuerySystemInformation(SystemModuleInformation,
&ModuleInfo,
sizeof(ModuleInfo),
NULL);
if (Status != STATUS_INFO_LENGTH_MISMATCH) return NULL;

//
// Initialize the kernel's full path name
//
Status = RtlCreateUnicodeStringFromAsciiz(&KernelName,
ModuleInfo.Modules[0].FullPathName);
if (!Status) return NULL;

//
// Keep only the short name
//
KernelName.Buffer = KernelName.Buffer +
(KernelName.Length / sizeof(WCHAR)) -
12;

//
// Map the kernel
//
Flags = IMAGE_FILE_EXECUTABLE_IMAGE;
Status = LdrLoadDll(NULL, &Flags, &KernelName, &KernelBase);
if (!NT_SUCCESS(Status)) return NULL;

//
// Find the address of KeServiceDescriptorTable
//
Status = LdrGetProcedureAddress(KernelBase, &TableName, 0, &TableBase);
if (!NT_SUCCESS(Status)) return NULL;

//
// Unload the kernel image, we're done with it
```

```
//
Status = LdrUnloadDll(KernelBase);
if (!NT_SUCCESS(Status)) return NULL;

//
// Get the virtual address we need
//
MappedAddress = (ULONG_PTR)ModuleInfo.Modules[0].ImageBase;
MappedAddress -= (ULONG_PTR)KernelBase;
MappedAddress += (ULONG_PTR)TableBase;

//
// Now read the SSDT
//
Status = ReadKernelMemory((PVOID)MappedAddress,
&ServiceTable,
sizeof(ServiceTable));
if (!NT_SUCCESS(Status)) return NULL;

//
// Setup the argument table
//
ArgumentTable = RtlAllocateHeap(RtlGetProcessHeap(),
0,
ServiceTable.Limit * sizeof(ULONG_PTR));
if (!ArgumentTable) return NULL;

//
// Now fill it up
//
Status = ReadKernelMemory(ServiceTable.Base,
ArgumentTable,
ServiceTable.Limit * sizeof(ULONG_PTR));
if (!NT_SUCCESS(Status)) return NULL;

//
// Now scan it
//
for (i = 0; i < ServiceTable.Limit; i++)
{
//
// Make sure no pointer is outside the kernel area
//
if (ArgumentTable[i] > 0x8FFFFFFF)
{
//


// Find the driver file that this belongs to
//
return FindDriverForAddress(UlongToPtr(ArgumentTable[i]));
}
}
//
// If we got here, then you don't have any rootkit
//
return NULL;
}
```

UNCLASSIFIED

Figure 5. (U) DKOM Detection Code