GREPEXEC: Grepping Executive Objects from Pool Memory

May 30, 2006

bugcheck
chris@bugcheck.org

Contents

Chapter 1

Foreword

Abstract:

As rootkits continue to evolve and become more advanced, methods that can be used to detect hidden objects must also evolve. For example, relying on system provided APIs to enumerate maintained lists is no longer enough to provide effective cross-view detection. To that point, scanning virtual memory for object signatures has been shown to provide useful, but limited, results. The following paper outlines the theory and practice behind scanning memory for hidden objects. This method relies upon the ability to safely reference the windows system virtual address space and also depends upon the building and locating effective memory signatures. Using this method as a base, suggestions are made as to what actions might be performed once objects are detected. The paper also provides a simple example of how object-independent signatures can be built and used to detect several different kernel objects on all versions of Windows NT+. Due to time constraints, the source code associated with this paper will be made publicly available in the near future.

Thanks:

Thanks to skape, Peter, and the rest of the uninformed hooligans; you guys and gals rock!

Disclaimer:

The author is not responsible for how the papers contents are used or interpreted. Some information may be inaccurate or incorrect. If the reader feels any information is incorrect or has not been properly credited please contact the author so corrections can be made. All content refers to the `Windows XP Service Pack 2` platform unless otherwise noted.

Chapter 2

Introduction

As rootkits become increasingly popular and more sophisticated than ever before, detection methods must also evolve. While rootkit technologies have evolved beyond API hooking methods, detectors have also evolved beyond the hook detection ages. At first rootkits such as FU[5] were detected using various methods which exploited its weak and proof-of-concept design by applications such as Blacklight[2]. These specific weaknesses were addressed in FUTo[7]. However, some still remain excluding the topic of this paper.

RAIDE[1], a rootkit detection tool, uses a memory signature scanning method in order to find EPROCESS blocks hidden by FUTo. This specific implementation works, however, it too has its weaknesses. This paper attempts to outline the general concepts of implementing a successful rootkit detection method using memory signatures.

The following chapters will discuss how to safely enumerate system memory, what to look for when building a memory signature, what to do once a memory signature has been found, and potential methods of breaking memory signatures. Finally, an accompanying tool will be used to concretely illustrate the subject of this paper.

After reading the following paper, the reader should have an understanding of the concepts and issues related to kernel object detection using memory signatures. The author believes this to be an acceptable method of rootkit detection. However, as with most things in the security realm, no one technique is the ultimate solution and this technique should only be considered complimentary to other known detection methods.

Chapter 3

Scanning Memory

Enumerating arbitrary system memory is nowhere near a science since its state can change at anytime while you are attempting to access it. While this is true, the memory that surrounds kernel executive objects should be fairly consistent. With proper care, memory accesses should be safe and the chance of false positives and negatives should be fairly minimal. The following sections will outline a safe method to enumerate the contents of both the system's `PagedPool` and `NonPagedPool`.

3.1 Retrieving Pool Ranges

For the purpose of enumerating pool memory it is unnecessary to enumerate the entire system address space. The system maintains a few global variables such as `nt!MmPagedPoolStart`, `nt!MmPagedPoolEnd` and related `NonPagedPool` variables that can be used in order to speed up a search and reduce the possibility of unnecessary false positives. Although these global variables are not exported, there are a couple ways in that they can be obtained.

The most reliable method on modern systems (`Windows XP Service Pack 2` and up) is through the use of the `KPCR->KdVersionBlock` pointer located at `fs:[0x34]`. This points to a `KDDEBUGGER_DATA64` structure which is defined in the `Debugging Tools For Windows`[8] SDK header file `wdbgexts.h`. This structure is commonly used by malicious software in order to gain access to non-exported global variables to manipulate the system.

A second method to obtain `PagedPool` values is to reference the per-session `nt!_MM_SESSION_SPACE` found at `EPROCESS->Session`. This contains information about the session owning the process, including its ranges and many other `PagedPool` related values shown here.

```
kd> dt nt!_MM_SESSION_SPACE
   +0x01c NonPagedPoolBytes : Uint4B
   +0x020 PagedPoolBytes   : Uint4B
   +0x024 NonPagedPoolAllocations : Uint4B
   +0x028 PagedPoolAllocations : Uint4B
   +0x044 PagedPoolMutex   : _FAST_MUTEX
   +0x064 PagedPoolStart   : Ptr32 Void
   +0x068 PagedPoolEnd     : Ptr32 Void
   +0x06c PagedPoolBasePde : Ptr32 _MMPTE
   +0x070 PagedPoolInfo    : _MM_PAGED_POOL_INFO
   +0x244 PagedPool        : _POOL_DESCRIPTOR
```

While enumerating the entire system address space is not preferable, it can still be used in situations where pool information cannot be obtained. The start of the system address space can be assumed to be any address above `nt!MmHighestUserAddress`. However, it would appear that an even safer assumption would be the address following the LARGE_PAGE where ntoskrnl.exe and hal.dll are mapped. This can be obtained by using any address exported by hal.dll and rounding up to the nearest large page.

3.2 Locking Memory

When accessing arbitrary memory locations, it is important that pages be locked in memory prior to accessing them. This is done to ensure that accessing the page can be done safely and will not cause an exception due to a race condition, such as if it were to be de-allocated between a check and a reference. The system provides a routine to lock pages named `nt!MmProbeAndLockPages`. This routine can be used to lock either pagable or non-paged memory. Since physical pages maintain a reference count in the `nt!MmPfnDatabase` there is no worry of an outside source unlocking the pages and having them page out to disk or become invalid.

In order to use `MmProbeAndLockPages`, a caller must first build an MDL structure using something such as `nt!IoAllocateMdl` or `nt!MmInitializeMdl`. The MDL creation routines are passed a virtual address and length describing the block of virtual memory to be referenced. On a successful call to `nt!MmProbeAndLockPages`, the virtual address range described by the MDL structure is safe to access. Once the block is no longer needed to be accessed, the pages must be unlocked using `nt!MmUnlockPages`.

A trick can be used to further reduce the number of pages locked when enumerating the `NonPagedPool`. As documented, `MmProbeAndLockPages` can be called at DISPATCH_LEVEL with the limitation of it only being allowed to lock resident memory pages and failing otherwise, which is a desirable side-effect in this

case.

Chapter 4

Detecting Executive Objects

In general, all of the executive components of the NT kernel rely on the object manager in order to manage the objects they allocate. All objects allocated by the object manager have a common header named `OBJECT_HEADER` and additional optional headers such as `OBJECT_HEADER_NAME_INFO`, process quota information, and handle trace information. Let's take a look to see what is common to all executive objects and how we can use the pool block header information to identify an allocated executive object. Lastly, some object specific information will be discussed in terms of generating a useful memory signature for an object.

4.1 Generic Object Information

Since the `OBJECT_HEADER` is common to all objects, let's look at it in detail. A static field here refers to all objects of specific type, not all executive objects in the system.

```
0: kd> dt _OBJECT_HEADER
   +0x000 PointerCount     : Int4B
   +0x004 HandleCount      : Int4B
   +0x004 NextToFree       : Ptr32 Void
   +0x008 Type             : Ptr32 _OBJECT_TYPE
   +0x00c NameInfoOffset   : UChar
   +0x00d HandleInfoOffset : UChar
   +0x00e QuotaInfoOffset  : UChar
```

```
   +0x00f Flags            : UChar
   +0x010 ObjectCreateInfo : Ptr32 _OBJECT_CREATE_INFORMATION
   +0x010 QuotaBlockCharged : Ptr32 Void
   +0x014 SecurityDescriptor : Ptr32 Void
   +0x018 Body             : _QUAD
```

PointerCount	Variable	# of references
HandleCount	Variable	# of open handles
NextToFree	NotValid	Used when freed
Type	Static	Pointer to OBJECT_TYPE
NameInfoOffset	Static	0 or offset to related header
HandleInfoOffset	Static	0 or offset to related header
QuotaInfoOffset	Static	0 or offset to related header
Flags	NotCertain	Not certain
ObjectCreateInfo	Variable	Pointer to OBJECT_CREATE_INFORMATION
QuotaBlockCharged	NotCertain	Not certain
SecurityDescriptor	Variable	Pointer to SECURITY_DESCRIPTOR
Body	NotValid	Union with the actual object

From this it is assumed that the most reliable and unique signature is the `Type` field of the `OBJECT_HEADER` which could be used in order to identify objects of a specific type such as `EPROCESS`, `ETHREAD`, `DRIVER_OBJECT`, and `DEVICE_OBJECT` objects.

4.2 Validating Pool Block Information

Kernel pool management appears to be slightly different from usermode heap management. However, if one assumes that the only concern is dealing with pool memory allocations which are less than `PAGE_SIZE`, it is fairly similar. Each call to `ExAllocatePoolWithTag()` returns a pre-buffer header as follows:

```
0: kd> dt _POOL_HEADER
   +0x000 PreviousSize     : Pos 0, 9 Bits
   +0x000 PoolIndex        : Pos 9, 7 Bits
   +0x002 BlockSize        : Pos 0, 9 Bits
   +0x002 PoolType         : Pos 9, 7 Bits
   +0x000 Ulong1           : Uint4B
   +0x004 ProcessBilled    : Ptr32 _EPROCESS
   +0x004 PoolTag          : Uint4B
   +0x004 AllocatorBackTraceIndex : Uint2B
   +0x006 PoolTagHash      : Uint2B
```

For the purposes of locating objects, the following is a breakdown of what could be useful. Again, static refers to fields common between similar executive objects and not all allocated `POOL_HEADER` structures.

PreviousSize	Variable	Offset to previous pool block
PoolIndex	NotCertain	Not certain
BlockSize	Static	Size of pool block
PoolType	Static	POOL_TYPE
Ulong1	Union	Padding, not valid
ProcessBilled	Variable	Allocator EPROCESS when no Tag specified
PoolTag	Static	Pool Tag (ULONG)
AllocatorBackTraceIndex	NotCertain	Not certain
PoolTagHash	NotCertain	Not certain

The `POOL_HEADER` contains several fields that appear to be common to similar objects which could be used to further verify the likelihood of locating an object of a specific type such as `BlockSize`, `PoolType`, and `PoolTag`.

In addition to the mentioned static fields, two other fields, `PreviousSize` and `BlockSize`, can be used to validate that the currently assumed `POOL_HEADER` appears to be a valid, allocated pool block and is in one of the pool managers maintained link lists. `PreviousSize` and `BlockSize` are multiples of the minimum pool alignment which is 8 bytes on a 32bit system and 16 bytes on a 64bit system. These two elements supply byte offsets to the neighboring pool blocks.

If `PreviousSize` equals 0, the current `POOL_HEADER` should be the first pool block in the pool's contiguous allocations. If it is not, it should be the same as the previous `POOL_HEADER`s `BlockSize`. The `BlockSize` should never equal 0 and should always be the same as the proceeding `POOL_HEADER`s `PreviousSize`.

The following code validates a `POOL_HEADER` of an allocated pool block.

```
//
// Assumes BlockOffset < PAGE_SIZE
// ASSERTS Flink == Flink->Blink && Blink == Blink->Flink
//
BOOLEAN ValidatePoolBlock (
    IN PPOOL_HEADER     pPoolHdr,
    IN VALIDATE_ADDR    pValidator
) {
    BOOLEAN bReturn = FALSE;

    PPOOL_HEADER    pPrev;
    PPOOL_HEADER    pNext;
```

```
    pPrev = (PPOOL_HEADER)((PUCHAR)pPoolHdr
                           - (pPoolHdr->PreviousSize * sizeof(POOL_HEADER)));
    pNext = (PPOOL_HEADER)((PUCHAR)pPoolHdr
                           + (pPoolHdr->BlockSize * sizeof(POOL_HEADER)));

    if
    ((
      ( pPoolHdr == pNext )
    ||( pValidator( pNext + sizeof(POOL_HEADER) - 1 )
     && pPoolHdr->BlockSize == pNext->PreviousSize )
    )
    &&
    (
      ( pPoolHdr  != pPrev )
    ||( pValidator( pPrev )
     && pPoolHdr->PreviousSize == pPrev->BlockSize )
    ))
    {
        bReturn = TRUE;
    }

    return bReturn;
}
```

4.3 Object Specific Signatures

So far a few useful signatures have been shown which apply to all executive objects and could be used to identify them in memory. For some cases these may be enough to be effective. However, in other cases, it may be necessary to examine information within the object's body itself in order to identify them. It should be noted that some objects of interest may be clearly defined and documented while others may not be. Furthermore, executive object definitions may vary between OS versions. The following subsections briefly outline obvious memory signatures for a few objects which generally are of interest when identifying rootkit-like behavior. A few examples of object-specific signatures will also be discussed, some of which have been used in previous work.

4.3.1 Process Objects

Here are just a few of the most basic `EPROCESS` fields which can form a simple signature using rather predictable constant values which hold true for all `EPROCESS` structures in the same system.

Pcb.Header.Type	Dispatch header type number
Pcb.Header.Size	Size of dispatcher object
Pcb.Affinity	CPU affinity bit mask, typically # CPU in system
Pcb.BasePriority	Typically the default of 8
Pcb.ThreadQuantum	Workstations is typically 18
ExitTime	0 for running processes
UniqueProcessId	0 if bitwise AND with 0xFFFF0002
SectionBaseAddress	Typically 0x00400000 for non-system executables
InheritedFromUniqueProcessId	Same as UniqueProcessId, typically a valid running pid
Session	Unique on a per-session basis
ImageFileName	Printable ASCII, typically ending in '.exe'
Peb	0x7FF00000 if bitwise AND with 0xFFF00FFF
SubSystemVersion	XP Service Pack 2 is 0x400

Note that there are several other `DISPATCH_HEADER`s embedded within locks, events, timers, etc in the structure which also have a predicable `Header.Type` and `Header.Size`.

4.3.2 Thread Objects

Here are just a few of the most basic `ETHREAD` fields which can form a simple signature using rather predictable constant values which hold true for all `ETHREAD` structures in the same system.

Tcb.Header.Type	Dispatch header type number
Tcb.Header.Size	Size of dispatcher object
Teb	0x7FF00000 if bitwise AND with 0xFFF00FFF
BasePriority	Typically the default of 8
ServiceTable	nt!KeServiceDescriptorTable(Shadow) used by RAIDE
Affinity	CPU affinity bit mask, typically # CPU in system
PreviousMode	0 or 1, which is KernelMode or UserMode
Cid.UniqueProcess	0 if bitwise AND with 0xFFFF0002
Cid.UniqueThread	0 if bitwise AND with 0xFFFF0002

Note that there are several other `DISPATCH_HEADER`s embedded within locks, events, timers, etc in the structure which also have a predicable `Header.Type` and `Header.Size`.

4.3.3 Driver Objects

A tool written previously named MODGREPPER[3] by Joanna Rutkowska of invisiblethings.org used a signature based approach to detect hidden `DRIVER_OBJECT`s. This signature was later 'broken' by valerino described in a rootkit.com article titled "Please don't greap me!"[6]. Listed here are a few fields which a signature could be built upon to detect `DRIVER_OBJECT`s.

Type	I/O Subsystem structure type ID, should be 4
Size	Size of the structure, should be 0x168
DeviceObject	Pointer to a valid first created device object(can be NULL)
DriverSection	Pointer to a nt!_LDR_DATA_TABLE_ENTRY structure
DriverName	A UNICODE_STRING structure containing the driver name

The following fields of the `DRIVER_OBJECT` can be validated by assuring they fall within the range of a loaded driver image such that:

```
DriverStart < FIELD < DriverStart + DriverSize.
```

DriverInit	Address of DriverEntry() function
DriverUnload	Address of DriverUnload() function, can be NULL
MajorFunction[0x1c]	Dispatch handlers for IRP_MJ_XXX, can default to ntoskrnl.exe

4.3.4 Device Objects

For the `DEVICE_OBJECT` structure there are few static signatures which are usable. Here are the only obvious ones.

Type	I/O Subsystem structure type ID, should be 3
Size	Size of the structure, should be 0xb8
DriverObject	Pointer to a valid driver object

Note that the `DriverObject` field must be valid in order for the device to function.

4.3.5 Miscellaneous

So far the memory signatures discussed have been fairly straight forward and for the most part are simply a binary comparison with a specific value. Later in

this paper (6.2), a technique called N-depth pointer validation will be discussed as a method of developing a more effective signature in situations where pointer based memory signatures are attempted to be evaded.

Another way of considering an object field as a signature is to validate it in terms of its characteristics instead of by its value. A common example of this would be to validate an object field `LIST_ENTRY`. Validating a `LIST_ENTRY` structure can be done as follows:

```
Entry == Entry->Flink->Blink == Entry->Blink->Flink.
```

A pointer to any object or memory allocation can also be checked using the function shown previously, named `ValidatePoolBlock`. Even a `UNICODE_STRING.Buffer` can be validated this way provided the allocation is less then `PAGE_SIZE`.

Chapter 5

Found An Object, Now What?

The question of what to do after potentially identifying an executive object through a signature depends on what the underlying goal is. For the purpose of a the sample utility included with this paper, the goal may be to simply display some information about the objects as it finds them.

In the context of a rootkit detector, however, there may be many more steps that need to be taken. For example, consider a detector looking for `EPROCESS` blocks which have been unlinked from the process linked list or a driver module hidden from the system service API. In order to determine this, some cross-view comparisons of the raw objects detected and the output from an API call or a list enumeration is needed. Detectors must also take into consideration the race condition of an object being created or destroyed in between the memory enumeration and the acquisition of the "known to the system" data.

Additionally, it may be desired that some additional sanity checks be performed on these objects in addition to the signature. Do the object fields x,y,z contain valid pointers? Is field c equal to b? Does this object appear to be valid however has signs of tampering in order to hide it? Does the number of detected objects match up with a global count value such as the one maintained in an `OBJECT_TYPE` structure? The following sections will briefly mention some random thoughts of what to do with a suspected object of the four types previously mentioned in this paper in Chapter 4.

5.1 Process Objects

Here is a brief list of things to check when scanning for `EPROCESS` objects.

- Compare against a high level API such as `kernel32!CreateToolhelp32Snapshot`.
- Compare against a system call such as `nt!NtQuerySystemInformation`.
- Compare against the `EPROCESS->ActiveProcessLinks` list.
- Does the process have a valid list of threads?
- Can `PsLookupProcessByProcessId` open its `UniqueProcessId`?
- Is `ImageFileName` a valid string? zeroed? garbage?

5.2 Thread Objects

Here is a brief list of things to check when scanning for `ETHREAD` objects.

- Compare against a high level API such as `kernel32!CreateToolhelp32Snapshot`.
- Compare against a system call such as `nt!NtQuerySystemInformation`.
- Does the process have a valid owning process?
- Can `PsLookupThreadByThreadId` open its `Cid.UniqueThread`?
- What does `Win32StartAddress` point to? Is it a valid module address?
- What is its `ServiceTable` value?
- If it is in a wait state, for how long?
- Where is its stack? What does its stack trace look like?

5.3 Driver Objects

Here is a brief list of things to check when scanning for `DRIVER_OBJECT` objects.

- Compare against services found in the service control manager database.
- Compare against a system call such as `nt!NtQuerySystemInformation`.
- Is the object in the global system namespace?

- Does the driver own any valid device objects?
- Does the drive base address point to a valid MZ header?
- Do the object's function pointer fields look correct?
- Does `DriverSection` point to a valid `nt!_LDR_DATA_TABLE_ENTRY`?
- Does `DriverName` or the `LDR_DATA_TABLE_ENTRY` have valid strings? zeroed? garbage?

5.4 Device Objects

Here is a brief list of things to check when scanning for `DEVICE_OBJECT` objects.

- Is the owning driver object valid?
- Is the device named and is it mapped into the global namespace?
- Does it appear to be in a valid device stack?
- Are its `Type` and `Size` fields correct?

Chapter 6

Breaking Signatures

Memory signatures can be an effective method of identifying allocated objects and can serve as a low level baseline in order to detect objects hidden by several different methods. Although the memory signature detection method may be effective, it doesn't come without its own set of problems. Many signatures can be evaded using several different techniques and non-evadable signatures for objects, if any exist, have yet to be explored. The following sections discuss issues and counter measures related to defeating memory signatures.

6.1 Pointer Based Signatures

Using a memory signature which is a valid pointer to some common object or static data is a very appealing signature to use for detection due to its reliability, however is also an easy signature to bypass. The following demonstrates the most simplistic method of bypassing the `OBJECT_HEADER->Type` signature this paper uses as a generic object memory signature. This is possible because the `OBJECT_TYPE` is just an allocated structure of fairly stable data. Many pointer based signatures with similar static characteristics are open to the same attack.

```
NTSTATUS KillObjectTypeSignature (
    IN  PVOID   Object
)
{
    NTSTATUS        ntStatus = STATUS_SUCESS;
    PVOID           pDummyObject;
    POBJECT_HEADER  pHdr;
```

```
    pHdr = OBJECT_TO_OBJECT_HEADER( Object );

    pDummyObject = ExAllocatePool( sizeof(OBJECT_TYPE) );

    RtlCopyMemory( pDummyObject, pHdr->Type, sizeof(OBJECT_TYPE) );

    pHdr->Type = pDummyObject;

    return STATUS_SUCCESS;
}
```

6.2 N-Depth Pointer Validation

As demonstrated in the previous section, pointer based signatures are effective. However, in some cases, they may be trivial to bypass. The following code demonstrates an example which does what this paper refers to as N-depth pointer validation in an attempt to create a more complex, and potentially more difficult to bypass, signature using pointers. The following example is also evadable using the same principal of relocation shown above.

The algorithm assumes a given address is an executive object and attempts validation by performing the following steps:

1. Calculates an assumed `OBJECT_HEADER`
2. Assumes `pObjectHeader->Type` is an `OBJECT_TYPE`
3. Calculates an assumed `OBJECT_HEADER` for the `OBJECT_TYPE`
4. Assumes `pObjectHeader->Type` is `nt!ObpTypeObjectType`
5. Validates `pTypeObject->TypeInfo.DeleteProcedure` == `nt!ObpDeleteObjectType`

```
BOOLEAN ValidateNDepthPtrSignature (
    IN  PVOID           Address,
    IN VALIDATE_ADDR    pValidate
)
{
    PVOID           pObject;
    POBJECT_TYPE    pTypeObject;
    POBJECT_HEADER  pHdr;

    pHdr = OBJECT_TO_OBJECT_HEADER( Address );
```

```
    if( ! pValidate(pHdr) || ! pValidate(&pHdr->Type) ) return FALSE;

    // Assume this is the OBJECT_TYPE for this assumed object
    pTypeObject = pHdr->Type;

    // OBJECT_TYPE's have headers too
    pHdr = OBJECT_TO_OBJECT_HEADER( pTypeObject );

    if( ! pValidate(pHdr) || ! pValidate(&pHdr->Type) ) return FALSE;

    // OBJECT_TYPE's have an OBJECT_TYPE of nt!ObpTypeObjectType
    pTypeObject = pHdr->Type;

    if( ! pValidate(&pTypeObject->TypeInfo.DeleteProcedure) ) return FALSE;

    // \ObjectTypes\Type has a DeleteProcedure of nt!ObpDeleteObjectType
    if( pTypeObject->TypeInfo.DeleteProcedure
            != nt!ObpDeleteObjectType ) return FALSE;

    return TRUE;
}
```

6.3 Miscellaneous

An obvious method of preventing detection from memory scanning would be to use what is commonly referred to as the Shadow Walker[4] memory subversion technique. If virtual memory is unable to be read then of course a memory scan will skip over this area of memory. In the context of pool memory, however, this may not be an easy attack since it may create a situation where the pool appears corrupted which could lead to crashes or system bugchecks. Of course, attacking a function like `nt!MmProbeAndLockPages` or `IoAllocateMdl` globally or specifically in the import address table of the detector itself would work.

For memory signatures based on constant or predicable values it may be feasible to either zero out or change these fields and not disturb system operation. For example take the author's enhancements to the FUTo rootkit where it is seen that the `EPROCESS->UniqueProcessId` can be safely cleared to 0 or previously mentioned rootkit.com article titled "Please don't greap me!" which clears `DRIVER_OBJECT->DriverName` and its associated buffer in order to defeat MODGREPPER.

For the case of some pointer signatures a simple binary comparison may not be enough to validate it. Take the above example and using `nt!ObpDeleteObjectType`. This could be defeated by overwriting `pTypeObject->TypeInfo.DeleteProcedure`

to point to a simple jump trampoline which is allocated elsewhere which simple jumps back to `nt!ObpDeleteObjectType`.

Chapter 7

GrepExec: The Tool

Included with this paper is a proof-of-concept tool complete with source which demonstrates scanning the pool for signatures to detect executable objects. Objects detected are `DRIVER_OBJECT`, `DEVICE_OBJECT`, `EPROCESS`, and `ETHREAD`. The tool does nothing to determine if an object has been attempted to be hidden in any way. Instead, it simply displays found objects to standard output. At this time the author has no plans to continue work with this specific tool, however, there are plans to integrate the memory scanning technique into another project. The source code for the tool can be easily modified to detect other signatures and/or other objects.

7.1 The Signature

For demonstration purposes the signature used is simple. All objects are allocated in `NonPagedPool` so only non-paged memory is enumerated for the search. The signature is detected as follows:

1. Enumeration is performed by assuming the start of a pool block.
2. The signature offset is added to this pointer.
3. The assumed signature is compared with the `OBJECT_HEADER->Type` for the object type being searched for.
4. The assumed `POOL_HEADER->PoolType` is compared to the objects known pool type.
5. The assumed `POOL_HEADER` is validated using the function from section 4.2, `ValidatePoolBlock`.

The following is the function which sets up the parameters in order to perform the pool enumeration and validation of a block by a single PVOID signature. On a match, a callback is made using the pointer to the start of the matching block. As an alternative to the PVOID signature, the `poolgrep.c` code can easily be modified to accept either a structure to several signatures and offsets or a validation function pointer in order to perform a more complex signature validation.

```
NTSTATUS ScanPoolForExecutiveObjectByType (
    IN  PVOID           Object,
    IN  FOUND_BLOCK_CB  Callback,
    IN  PVOID           CallbackContext
) {
    NTSTATUS        ntStatus = STATUS_SUCCESS;
    POBJECT_HEADER  pObjHdr;
    PPOOL_HEADER    pPoolHdr;
    ULONG_PTR       blockSigOffset;
    ULONG_PTR       blockSignature;

    pObjHdr         = OBJECT_TO_OBJECT_HEADER( Object );
    pPoolHdr        = OBJHDR_TO_POOL_HEADER( pObjHdr );
    blockSigOffset  = (ULONG_PTR)&pObjHdr->Type - (ULONG_PTR)pObjHdr
                       + OBJHDR_TO_POOL_BLOCK_OFFSET(pObjHdr);
    blockSignature  = (ULONG_PTR)pObjHdr->Type;

    (VOID)ScanPoolForBlockBySignature(  pPoolHdr->PoolType - 1,
                                        0, // pPoolHdr->PoolTag OPTIONAL,
                                        blockSigOffset,
                                        blockSignature,
                                        Callback,
                                        CallbackContext );
    return ntStatus;
}
```

7.2 Usage

GrepExec usage is pretty straightforward. Here is the output of the help command.

```
*******************************************************
```

```
 GREPEXEC 0.1 * Grepping executive objects from the pool *
 Author: bugcheck
 Built on: May  30 2006
************************************************************

Usage: grepexec.exe [options]

  --help,       -h                    Displays this information
  --install,    -i                    Manually install driver
  --uninstall,  -u                    Manually uninstall driver
  --status,     -s                    Display installation status
  --process,    -p                    GREP process objects
  --thread,     -t                    GREP thread objects
  --driver,     -d                    GREP driver objects
  --device,     -e                    GREP device objects
```

7.3 Sample Output

The standard output is also straightforward. Here is a sample of each supported command.

```
C:\grepexec>grepexec.exe -p
EPROCESS=81736C88 CID=0354 NAME: svchost.exe
EPROCESS=8174E238 CID=0634 NAME: explorer.exe
EPROCESS=81792020 CID=027c NAME: winlogon.exe
...

C:\grepexec>grepexec.exe -t
EPROCESS=817993C0 ETHREAD=815D4A58 CID=0778.077c wscntfy.exe
EPROCESS=8174AA88 ETHREAD=815D6860 CID=0408.0678 svchost.exe
EPROCESS=819CA830 ETHREAD=815F3B30 CID=0004.0368 System
EPROCESS=81792020 ETHREAD=81600398 CID=027c.0460 winlogon.exe
...

C:\grepexec>grepexec.exe -d
DRIVER=81722DA0 BASE=F9B5C000 \FileSystem\NetBIOS
DRIVER=819A4B50 BASE=F983D000 \Driver\Ftdisk
DRIVER=81725DA0 BASE=00000000 \Driver\Win32k
DRIVER=81771880 BASE=F9EB4000 \Driver\Beep
 ...

C:\grepexec>grepexec.exe -e
```

```
DEVICE=81733860 \Driver\IpNat        NAME: IPNAT
DEVICE=81738958 \Driver\Tcpip        NAME: Udp
DEVICE=817394B8 \Driver\Tcpip        NAME: RawIp
DEVICE=81637CE0 \FileSystem\Srv      NAME: LanmanServer
...
```

Chapter 8

Conclusion

From reading this paper the reader should have a good understanding of the concepts and issues related to scanning memory for signatures in order to detect objects in the system pool. The reader should be able to enumerate system memory safely, construct their own customized memory signatures, locate signatures in memory, and implement their own reporting mechanism.

It is obvious that object detection using memory scanning is no exact science. However, it does provide a method which, for the most part, interacts with the system as little as possible. The author believes that the outlined technique can be successfully implemented to obtain acceptable results in detecting objects hidden by rootkits.

Bibliography

[1] Blackhat.com. *RAIDE: Rootkit Analysis Identification Elimination.* http://www.blackhat.com/presentations/bh-europe-06/bh-eu-06-Silberman-Butler.pdf; Accessed May. 30, 2006.

[2] F-Secure. *Blacklight.* http://www.f-secure.com/blacklight/; Accessed May. 30, 2006.

[3] Invisiblethings.org. *MODGREPPER.* http://www.invisiblethings.org/tools.html; Accessed May. 30, 2006.

[4] Phrack.org. *Shadow Walker.* http://www.phrack.org/phrack/63/p63-0x08_Raising_The_Bar_For_Windows_Rootkit_Detection.txt; Accessed May. 30, 2006.

[5] Rootkit.com. *FU.* http://rootkit.com/project.php?id=12; Accessed May. 30, 2006.

[6] Rootkit.com. *Please don't greap me!.* http://rootkit.com/newsread.php?newsid=316; Accessed May. 30, 2006.

[7] Uninformed.org. *futo.* http://uninformed.org/?v=3&a=7&t=sumry; Accessed May. 30, 2006.

[8] Windows Hardware Developer Central. *Debugging Tools for Windows.* http://www.microsoft.com/whdc/devtools/debugging/default.mspx; Accessed May. 30, 2006.