A Taxonomic Evaluation of Rootkit Deployment, Behavior and Detection

A Thesis

Presented in Partial Fulfillment of the Requirements for the

Degree of Master of Science

with a

Major in Computer Science

in the

College of Graduate Studies

University of Idaho

by

Maxine Major

Major Professor: Jim Alves-Foss, Ph.D.

Committee Members: Daniel Conte de Leon, Ph.D.; Sara Eftekharnejad, Ph.D.

Department Administrator: Gregory Donohoe, Ph.D.

July 2015

Authorization to Submit Thesis

This Thesis of Maxine Major, submitted for the degree of Master of Science with a major in Computer Science and titled **"A Taxonomic Evaluation of Rootkit Deployment, Behavior and Detection"**, has been reviewed in final form. Permission, as indicated by the signatures and dates given below, is now granted to submit final copies to the College of Graduate Studies for approval.

Major Professor	_____ Jim Alves-Foss, Ph.D.	Date _____
Committee Members	_____ Daniel Conte de Leon, Ph.D.	Date _____
	_____ Sara Eftekharnejad, Ph.D.	Date _____
Computer Science Department Administrator	_____ Gregory Donohoe, Ph.D.	Date _____

Abstract

Increased inter-connectivity between cyber and cyber-physical systems increases the danger of Advanced Persistent Threat (APT) cyber attacks, against which perimeter-focused defenses are no longer sufficient. Rootkits are debatably the most important piece of malicious software to the success of an APT. Rootkits are are often planted through social engineering, which intend to bypass perimeter–focused defenses. APTs, the most dangerous of cyber attacks, is facilitated by one of the least-detected attack methods.

In order to further the practice of detecting rootkits and aid with early detection, this thesis presents a taxonomy of rootkit activities through each stage of installation and exploitation. Correspondingly, this thesis presents a taxonomy of rootkit detection methods to address rootkit infection vectors. These taxonomies are then applied to a real-world rootkit example to demonstrate how combined application of rootkit detection tools and techniques can provide full-coverage of the possible rootkit-targeted attack surface.

Acknowledgments

I would first like to thank my advisor, Dr. Jim Alves-Foss, for his support, encouragement, and patient guidance throughout my graduate studies.

I would also like to thank my other committee members, Dr. Conte de Leon, and Dr. Sara Eftekharnejad, for their valuable input and comments on my thesis.

I would like to thank Dr. Paul Oman for launching my interest in cyber security, and structuring courses which encouraged curiosity and allowed me to explore.

I would like to thank all my instructors for their hard work and dedication in providing me with a comprehensive and valuable education.

I would like to thank the department chair, Dr. Gregory Donohoe, and Mrs. Arvilla Daffin and other staff in the Department of Computer Science and the Center for Secure and Dependable Systems for their help during my study in the department.

I wish to acknowledge the National Science Foundation CyberCorps® Scholarship for Service, for supporting me during the course of my graduate studies.

Last, but certainly not least, I would like to thank my family and all the friends I have met while at University of Idaho for their support, encouragement, and love which have helped me make this thesis a reality.

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

The security of computing systems is difficult to ensure in the modern era. With the increased complexity of operating systems, applications, network configurations, and combinations of proprietary hardware and software, the number of vulnerabilities in any given system seems endless. Security researchers and system administrators attempt to discover and patch vulnerabilities as they become aware of their existence, but malicious actors are also actively searching for any vulnerability they can exploit. For each vulnerability that is patched, many other vulnerabilities exist which remain undiscovered and unreported. This seeming futility of attempting to defend computer systems against malicious cyber threats is casually referred to as "the Great Wall of Swiss Cheese".

1.1 The Problem Space

There are several methods by which a malicious actor can infiltrate, gain control, and maintain that control over a given system. The ideal scenario for an attacker is to infiltrate a system and then take additional actions to ensure that their activity remains undetected while keeping a "backdoor" open to maintain consistent access to that system. A backdoor is a vulnerability or a communication channel which allows an attacker to maintain access to a system. These types of attacks are called Advanced Persistent Threat (APT) attacks, and are one of the most serious threats in today's increasingly connected cyber world [23] [1].

1.1.1 Advanced Persistent Threats

Detection of APT attacks is difficult due to the fact that the goal of an APT is to remain undetected while gaining as much access and as high a level of privileges in as many systems as necessary in order to achieve its goal. In order to do this, an APT generally does not compromise systems using a noisy method of attack, such as penetration testing [2]. An APT also does not expect to achieve a single goal, such as exfiltrating a credit card database, before exiting the system [1].

An APT is a *blended* attack [71], by which several covert exploits are carried out to infiltrate

a system, hide evidence of access, and quietly attempt to discover and gain access to additional connected resources, while attempting to not trigger any threat detection systems. In addition, an APT will install covert services (i.e., backdoors) that allow quick and easy controlled access to the infiltrated system [1]. A successful APT will compromise several resources, so that if one of its backdoors is discovered and patched, several others still exist which allow the attacker to regain the territory they momentarily lost [28].

1.1.2 Rootkits

It could be argued that the most important component to APTs is the backdoor in any devices they subvert. The backdoor used by an APT is commonly implemented through a *rootkit*: a type of malicious software ("malware") which exploits the host system to gain privileges, hides itself from detection, and keeps a backdoor open for the attacker to continue to carry out malicious activities within the system. The backdoor itself can be either an open network port, or corrupted operating system data which creates a vulnerability that the attacker can exploit for re-entry [42].

Rootkits are often planted through social engineering tactics, usually by convincing an insider to unknowingly download and install a malicious binary or corrupted document [23]. Because a rootkit's activities are focused on evading discovery, it is often discovered only after it has been firmly planted within the victim system and is communicating information and instructions to a Command and Control (C2) server off-site [78].

A system administrator who discovers a rootkit may attempt to remove any obviously infected files and patch any vulnerabilities that enabled its entry. Because rootkits often overwrite or manipulate commands in the system shell or kernel, removing a rootkit is rarely 100 % guaranteed to be successful. Reinstallation of the operating system is the recommended solution [84], but some rootkits, such as those that infect the BIOS, can even survive reboots and reinstallation of the operating system [73]. With the highest privileges and the ability to overwrite kernel-level instructions, the scope of a kernel–level rootkit's activities within a single system is vast.

Rootkits don't always work alone either. Some rootkits can work together as botnets, communicating information between computers around the world [4]. Other rootkits, more dangerously, can collaborate between several compromised systems in the same internal network as part of a laterally–moving APT [32] [28].

Because the majority of intrusion detection and malware defenses aim to protect the perimeter of a network, once rootkits gain access to the inside, they can remain undetected for months, or sometimes even years. Throughout an APT campaign, malware components quietly pass information between services, and slowly compromise additional services, and masquerade their traffic as normal business activities [78].

This poses a problem for system administrators, particularly those who maintain critical infrastructures, such as control systems, or those who are tasked with an enterprise-wide environment with varying levels of user authentication, permissions, and potentially thousands of different proprietary software applications and security controls. Detecting and removing rootkits from a single computer is a challenging task, and to extend that scope of responsibility to an enterprise-wide infrastructure is even more difficult.

1.1.3 The Practice of Rootkit Detection

Some well-respected standalone rootkit detectors such as *GMER* [24] and Malwarebytes' *Anti-Rootkit BETA* [41] perform well in rootkit detection and removal tests. Other tools, such as Kaspersky's *TDSSKiller* [33] aim to remove a single type of rootkit, but may be effective in detecting other types of rootkits as well. These tools, like many other commercial and open source tools, have a limited scope that is not always clearly defined. Even though some rootkit detection tools provide information on what parts of a system are scanned for malicious activity, it is unclear exactly what that means in regards to the completeness of detection coverage.

The problem lies in the fact that rootkit types and activities are not currently meaningfully classified. This hinders detection and removal tools from being applied efficiently and completely. Rootkit classification is generally high level, either based on the permissions limiting rootkit activity, user-mode versus kernel-mode, or based on a generalization of the domain of the rootkit (e.g., *root*kit versus *boot*kit). There is no standardized classification through which to understand rootkit activities, so that rootkits can be understood and identified by the domain of their activities. There is also no classification through which detection methods can be clearly applied to these domains. This lack of classification granularity provides very little information for potential rootkit victims to understand how much coverage their rootkit detection solution provides, nor what tools to use to gain complete coverage.

Current rootkit detection methods and the application of those methods would be better facilitated through the following contributions to the practice of rootkit detection and elimina-

tion:

1. A classification which clearly translates rootkit behavior to system-level activities and groups these activities by the types of modifications inflicted on the system.

2. A classification of rootkit detection methods which can be directly applied to each domain of rootkit activity.

1.2 Motivation

Initial research for this project involved a short-term study to determine what requirements would be necessary in order to detect a cyber event at the earliest stages. Literature on this subject contains several papers detailing experimental methods to detect malicious activity as it enters a network or to detect malware exhibiting certain behaviors. However, research rarely classifies the domains of malware activity within a computer, and does not draw a distinction between rootkit behavior and other malware. Detection methods tend to consist of small single-purpose tools, proof-of-concept research solutions, or are integrated into Security Information and Event Management (SIEM) proprietary software – of which there is a lack of publicly available technical documentation. Further discussion of this literature is provided in Chapter 5.

The focus of industry's most cutting-edge tools, usually in the form of Intrusion Detection Systems (IDS), Intrusion Prevention Systems (IPS), or SIEM systems aims to detect malicious software and infiltration attempts as they enter the network. Host-based solutions, while included as a feature of the SIEM, seem to be an afterthought. Additionally, the goal of defending against APTs and other more immediate threats is to move from a *reactive* stance of patching holes in the Great Wall of Swiss Cheese to a *proactive* stance, by which we can detect and defend against malware threats before they occur. By enabling the detection of a rootkit threat before it is able to carry out its malicious actions, the state of malware detection will move one step closer to becoming predictive.

1.3 Objectives

Objectives for this thesis are as follows:

1. Provide guidelines by which rootkits may be classified through identifying its core behaviors and objectives.

2. Provide a systematic way of categorizing the activities performed by a rootkit throughout each stage of its lifespan, from creation to fulfillment of its mission.

3. Examine rootkit behaviors and activities within a system to classify rootkit activity by domain affected such that detection methods may be applied according to these classifications.

4. Identify and classify rootkit detection methods which apply to each domain of rootkit activity. Develop a method of displaying this information which clearly identifies areas of rootkit activity with insufficient detection coverage.

5. Examine and draw conclusions about the current state of rootkit detection as related to the identified domains of rootkit activity, and propose future research projects or industry related activities which could help bring rootkit detection to a feasible, real-time, full-coverage state.

1.4 Thesis Impact

This research is intended to provide a logical and complete mid-level taxonomic view of rootkit activity and corresponding detection methods. The taxonomic views presented here provide a classification with which to bridge the knowledge gap between existing high-level concepts and the lower-level specifics of rootkit detection implementations. This research also provides a method that can be used to categorize and compare rootkit detection and removal implementations. Also, this research frames the rootkit problem space in a way that helps facilitate both understanding and communication regarding rootkit actions and responses. Additionally, this classification provides a way to classify the coverage space of existing rootkit detection solutions, to distinctly identify and communicate the areas of coverage and deficiencies in existing rootkit detection tools, and identify vectors to add coverage functionality.

This taxonomic classification can assist both academia and industry in developing more complete rootkit detection tools and methods by providing a centralized source of information about rootkit activities and corresponding detection techniques. The taxonomies presented here provide categories with which to classify and compare each rootkit infiltration method by the attack space it addresses and the techniques it uses, as well as a way to classify detection tools which address each of these attack spaces. This taxonomy can also be used to evaluate

and compare existing and future rootkit detection methods, and provide a starting point for which tool coverage and efficiency may be computed.

This thesis contributes to the scientific community by:

1. Providing a taxonomic classification through which rootkit objectives, activities, and detection methods can be more easily understood and communicated. This taxonomy will help research and industry to organize and frame their specific interests within this space so that it may be refined, expanded upon, or derived from, as the practice of rootkit development and detection progresses.

2. Making it easier to understand exactly where existing rootkit detection tools provide rootkit detection coverage and for which rootkit activities these tools could be further developed in order to improve coverage. Additionally, information provided in this thesis will help facilitate decision-making in regards to which detection methods may be more applicable.

3. Providing structured diagrams which increase understanding of rootkit activities and how they relate to their respective attack spaces and to each other. These visualizations also help facilitate understanding of how detection methods apply to the rootkit attack space, and will demonstrate both overlaps and oversights as related to existing rootkit detection techniques.

1.5 Thesis Overview

The remainder of this thesis is organized as follows. Chapter 2 provides background on APTs and the malware lifecycle demonstrated through the kill chain model. The kill chain model presented in this thesis is inspired by the Cyber Kill Chain® model developed by Lockheed Martin [40], and is sometimes called the "Intrusion Kill Chain" in other literature [29] [8]. This chapter also provides an overview of the rootkit as a kernel-level threat.

Chapter 3 provides a detailed discussion of each stage of rootkit activities corresponding to the Stages 1–3 of the kill chain: rootkit preparation and delivery.

Chapter 4 provides a detailed discussion of each stage of rootkit activities corresponding to Stages 4–7 of the kill chain: rootkit installation and system-level activities. This chapter provides taxonomic representations of rootkit attack vectors and other rootkit strategies, such

as process hiding and backdoor vulnerability creation. This chapter also discusses the system kernel and addresses rootkit activity within this space.

Chapter 5 presents an overview of rootkit detection methods and demonstrates how the current practice of rootkit detection relates to the detectability of a rootkit's presence and activities. This chapter concludes by directly correlating detection methods with each of the rootkit attack vectors identified in Chapter 3.

Chapter 6 applies the taxonomies presented in Chapters 4 and 5 to two Metasploit rootkits and the ZeroAccess rootkit, demonstrating their effectiveness at framing the problem space and simplifying the detection space.

Chapter 7 presents conclusions revealed by these taxonomies and the scope of their effectiveness. Then future projects are proposed to extend this work to help advance the science of rootkit detectability.

Chapter 2
Background

There are three major topics encompasing the background of this work, which are summarized in this chapter. First, Section 2.1 is an overview of the problem space, in particular the dangers of *Advanced Persistent Threat (APT)* attacks, their prevalence and the danger posed by these attacks. Included in this discussion is a description of how *rootkits* are debatably the most critical piece to the survivability of APTs. However, due to the perimeter-based focus of defenses, it's easy for rootkits to penetrate defenses, using system vulnerabilities in combination with social engineering attacks. Finally, Section 2.2 describes the *kill chain*, the sequence of steps which nearly all cyber attacks follow, including that of rootkits.

Rootkits will be discussed in greater detail in Chapter 3.

2.1 Advanced Persistent Threats

An APT attack is a long-term, mission-based cyber attack which aims to compromise several services, usually belonging to a single target, while remaining undetected. According to the National Institute of Standards and Technology (NIST), APTs are "a long-term pattern of targeted, sophisticated attacks" [53]. Symantec describes APTs as using "multiple phases to break into a network, avoid detection, and harvest valuable information over the long term" [75].

An APT can utilize several different attack vectors, and compromise multiple devices in a particular domain, while evading detection.

2.1.1 APT Goals

Rather than disrupt services, such as is the objective of DDoS attacks, APTs attempt to gain access to the target's resources and maintain persistence within the target's system [78]. APTs are "goal oriented" [13] and often work covertly in order to exfiltrate valuable or sensitive data from high-value targets.

Because of the high financial cost and many man hours required to carry out these attacks, APTs are usually carried out by two types of actors: a) organized criminal groups and b) state actors [78] [70].

The goals of an APT can range from political espionage, targeting military intelligence, corporate research and intellectual property (IP), exfiltration of credit card or banking information, to theft of personally identifiable information (PII), which is commonly referred to as "identity theft" [80] [36]. APT motives are often financial gain for criminal organizations, and military and IP for state actors [70].

According to Verizon's 2014 Data Breach Report [78], 87% of external actors (i.e., not insider threats) committing cyber espionage were state-affiliated, and 11% of external actors were related to oranized crime. All other actors combined accounted for the remaining 2% of cyber espionage-associated attacks. In 2013, a full 54% of cyber espionage attacks were aimed directly at the United States, with the next closest nation victimized by cyber espionage (South Korea) at the receiving end of 6% of these attacks.

The external actors committing cyber-espionage were roughtly attributed to Eastern Asia (49%), Unattributed (25%), and Eastern Europe (21%). The rest of the world accounted for the remaining 5% of cyber espionage attacks [78]. This would seem to indicate that certain parts of the world are highly invested in infiltrating the United States in order to obtain research data, trade secrets, human data (PII), and even government and military secrets.

2.1.2 APT Persistence

In attacks which involve simple goals which can be executed quickly, the attacker can hide in plain sight, and the use of covert measures may not even be necessary. However, with APTs, this is a different matter [60].

Sensitive, highly desirable data is likely to be heavily protected, segregated from public-facing and commonly-accessed internal services. Because of this, the methods required to infiltrate the target system may be quite complex, and involve several exploitation methods, slowly gaining ground by building on previous exploits. The best way to carry out this type of campaign is to proceed slowly so that all activity evades detection completely [2]. Altshuler *et al.* demonstrated that very low levels of attack aggressiveness were the most rewarding over time, which explains why APTs are successful at exfiltrating secrets [2].

2.1.3 APT Launch and Detection

Active APTs are difficult to detect, and even harder to detect after they have successfully infiltrated a victim's network. Often, APTs start with a single piece of malware, such as a *rootkit*, emailed to an insider at the target organization. If the insider opens the email

attachment, they unknowingly launch a cyber attack [42]. This is known as a type of *social engineering*, which is a very common method by which attackers trick an insider into helping them breach external defenses [14].

The vector for malware infection in 2013 was overwhelming via malicious email attachments (87%), followed by "web drive-by" (20%), which implies that casual users browsing the web were infected by visiting infected websites [78]. The fact that the majority of malware was installed by email attachments is indicative of the level of success of social engineering [78]. These infection methods are discussed further in Chapter 3, Section 3.2.2 "Social Reconnaissance", and 3.3 "Weaponization".

Most cyber espionage attempts go undetected for months (62%), and according to the Verizon 2014 Data Breach Report, the most common way these attempts are discovered is "ad hoc notification" from organizations which observe the victim's network traffic communicating with the Command and Control (C2) servers of a known threat group. 67% of "unrelated" parties are the vector for revealing the existence of these attempts [78].

Clearly APT attacks can pose a serious threat, and current detection implementations and practices may not be up to the task of detecting many APT attacks. Regardless of how skilled the human adversary may be, malware - such as rootkits - are vital to the success of an APT. These rootkits can be very difficult to detect, which is another reason why APTs are so successful. By understanding the APT threat's operational strategy, it may be possible to identify some vectors which are more valuable than others toward the detection of an APT. An APT launches and initially deploys much the same as any other malware or cyber threat, and the model by which an APT targets, enters, and maintains persistence inside a target is called the kill chain [8] [29]. Rootkits also follow this same creation, deployment, and target-focused actions. The kill chain is discussed from a high level view in Section 2.2.

2.1.3.1 Zero-Day Threats

For the most part, cyber exploits fall into two categories: those that should be able to be detected, and those for which there is no precedent and existing defenses cannot be expected to detect that attack.

For cyber attacks that exploit known vulnerabilities, the exploit should theoretically be able to be detected because knowledge exists about that particular attack; the exploitability lies in whether or not a patch or set of mitigations exist for that vulnerability. Exploits that have

not been seen before are called *"zero-day"* exploits, because, as of the time that the exploit is carried out, system administrators have had zero days to defend against this particular attack [67] [80] [14].

Zero day attacks are a hot commodity on the black market, and can fetch prices up to hundreds of thousands of dollars, depending on the target system, level of covertness, and level of access gained through the use of this exploit [6]. While zero day attacks may not be a serious concern for the individual whose digital property may not be worth the $100K price tag, high-profile targets can expect to fall victim to zero day attacks [59]. According to the security firm FireEye, "Zero-day attacks are an important weapon in every APT arsenal" [21].

Because of the high cost, zero-day exploits are not commonly used except where absolutely necessary. Flame made use of two zero day exploits [79], while Stuxnet claimed use of up to *four* zero-days throughout the entire attack campaign [50] [72]. However, to merely establish a presence within a victim's network, social engineering combined with a known vulnerability can be equally effective. These famous APTs are further discussed in Section 2.1.4.

2.1.3.2 Social Engineering

Social engineering is a type of attack in which an adversary uses subtle and manipulative means in order to convince an employee working at the target organization to help the adversary compromise that target [71] [78]. For example, an adversary can place a phone call during business hours to an employee, posing as the IT department, and invent some technical problem which requires that employee's password. Another example of social engineering involves "piggybacking"(also known as "tailgating" or "drafting"), a method by which an adversary poses as an employee, sometimes by carrying paperwork, a box, or coffees, and physically follows employees through a security checkpoint that normally requires key-card access. Because of natural complacency and social pressure to trust fellow employees the adversary knows that legitimate employees are not likely to question his attempted entry [35].

The same methods of deceiving an insider to believe the attacker's request for access is legitimate hold true for more cyber-only methods of social engineering. Despite an organization's best attempts to train employees to be suspicious of links in emails and to only open attachments from known senders, employee complacency and trust still work in favor of the adversary [23] [2]. Additionally, once an employee's machine is compromised, the attacker can use that employee's privileges and identity to leap-frog through the organization, spreading

malware from "trusted" accounts [32] [28] [78] [42] [29].

Often these attacks are carried out using spearphishing. *Spearphishing* attacks are targeted social engineering attacks which utilizes the attacker's direct knowledge of the target company to send phishing emails to hand-picked recipients. These emails contain information that appears relevant to the company and recipient [3] [23].

For example, the attacker sends a contextually relevant email that appears to originate from within the company, which contains what appears to be a legitimate attachment – often a PDF or Microsoft Office document. This attachment contains a malicious executable which will usurp the user's privileges to bury and hide itself in the victim's machine, while crafting a backdoor for the attacker to gain access directly to that machine [42] [23] [78] [85].

Ultimately, this strategy can be successful regardless of both network and host-based defenses in place. Even a payload can go undetected at the network level because of encoding schemes which change the signature of the malware. In the end, any system can be compromised if the users "choose" to install or download malware or malicious updates [82], which is the strength of social engineering. Strategies which defeat the network and trick the end user are further detailed in Section 3.3.

2.1.4 Famous APTs

APTs have risen in prominence in the past decade, with high-profile attacks making headlines around the world. The following sections describe a few of the most famous APTs, some of which will also be referenced throughout this thesis. As can be seen from these examples, these APTs were likely developed by state-sponsored actors on high-stakes targets, revealing the scope of damage possible when APTs succeed.

2.1.4.1 Stuxnet

Stuxnet was an APT first discovered in 2009, which appeared to be designed to attack the Iranian Nuclear Program's Natanz uranium enrichment plant. Stuxnet specifically caused physical damage by targeting the Industrial Control Systems (ICS)'s Programmable Logic Controllers (PLC) that managed the uranium enrichment centrifuges. The infected PLCs caused the centrifuges to operate at speeds outside designated limits, ruining the present batch of uranium and setting the Iranian nuclear program back four years [79] [72].

The initial infection vector for Stuxnet is not known, but may have been the result of spearphishing emails to contracted employees [85].

Stuxnet likely spread like a worm through networks and connected devices, and it is likely that the targeted PLCs, which were not connected to the internet, were infected via USB [67]. Once connected, Stuxnet automatically executed and gained access to critical resources [79].

Stuxnet primarily infected Windows systems running Siemens Step-7 software. The malicious binaries were hidden by rootkits, and the PLC code was modified to present normal operating values to monitoring software. New PLC drivers were signed using compromised digital certificates. Stuxnet scanned the infected system for the presence of security monitoring software, and depending on what service it detected, it would inject its payload in a way to avoid detection. It used encryption for some of the payload and to communicate information about the infected system to C2 servers when internet access was available [79].

It is believed that the team behind Stuxnet was state-sponsored, because the complexity needed to ensure success on such a specific target would require a realistic test environment modeling the uranium enrichment facility, experts with extensive knowledge on the operation of such equipment in a nuclear facility, highly skilled programmers and researchers who either personally developed, likely had financial backing, and may have had direct access to a repository of zero-day exploits [79].

2.1.4.2 Duqu

Duqu was an APT discovered in 2011 which infected very few targets – approximately 50 worldwide. Duqu had a structure similar to Stuxnet, but rather than industrial sabotage, its mission was espionage.

Duqu was initially spread through spearphishing emails [85] which contained infected Microsoft Word files, but did not auto-propagate. Like Stuxnet, it hid its code using rootkits and used compromised digital certificates [79].

Duqu facilitated espionage through a keylogger, which captured user credentials such as passwords, and then communicated the keylogged data to offsite C2 servers using steganographic encryption. After 36 days, unless the attacker sent specific instructions to do otherwise, Duqu self destructed [79].

2.1.4.3 Flame

Flame was an APT first detected in 2012, but is suspected to have already been active for many years prior. It is believed that Flame compromised several thousand Windows systems in the Middle East, and is notable for its small size – just under 20 megabytes [79].

Flame's mission was espionage, with modules that engaged in keylogging, taking screenshots, capturing emails, recording audio using the computer's internal microphone, and gathering information about nearby Bluetooth devices. The most impressive feature of Flame is its ability to impersonate a windows update server. In order to forge signatures for Flame's "software updates", it had to perform a cryptanalytic attack against Microsoft's Terminal Services licensing certificate authority in order to generate valid digital signatures [79].

Flame's initial infection vector was likely targeted spearphishing emails [85] and it did not self-propagate but spread via connected devices, such as USB. Although its overall design and mission is different from Stuxnet, the use of two of the exact same zero day vulnerabilities as Stuxnet, as well as a similar keylogging module to the one used by Duqu, lead researchers to believe that they were developed by collaborating teams [79].

2.1.4.4 Red October

Red October is an APT that was detected in 2012, but may have been active since 2007. Red October targeted both government and industry for the purposes of both political espionage and IP theft, particularly in Russian-speaking countries [78].

Red October was spread through targeted emails containing malicious Microsoft Word and Excel files. Using a minimalist architecture, Red October downloaded modular functionality as needed from C2 servers. It was able to steal information from both PCs and mobile devices such as Nokia phones and iPhones. Activities included keylogging, email interception, and recovery of deleted files [79].

2.2 The Kill Chain

Almost every malicious cyber attack which penetrates internal defenses from outside the victim's network follows the pattern of the kill chain model [8] [29]. Each one of these steps much be performed in order for a cyber attack to be considered successful [13]. A simplified version of this model is shown in Figure 2.1.

Figure 2.1: Kill Chain

The steps of the kill chain model are as follows:

1. **Reconnaissance**: Also known as the "information gathering" stage, the attackers gather

data on the intended target, and likely use many publicly-available resources in order to remain undetected. This can involve visiting corporate websites, blogs, and making friends and collecting information about employees of the organization, as well as their personal and public interests.

2. **Weaponization**: The attacker crafts the malicious payload, which includes exploits, malware, and other activities which will be used to compromise the victim.

3. **Delivery**: The malicious payload is sent to the victim and successfully received.

4. **Exploitation**: The exploit is executed. Often this is triggered by a legitimate user unknowingly authorizing the installation of malware by clicking on an unsafe URL or opening a malicious email attachment.

5. **Installation**: The exploit is used to install malware on the victim machine. This step is only relevant if malware is used, as is the scenario explored in this paper.

6. **Command and Control**: The malware (rootkit) contacts a Command and Control (C2) server hosted by the attackers, to report a successful installation and obtain further instructions. Often the C2 server is used to download additional malware with enhanced functionality onto the victim's machine.

7. **Action**: This is the step in the kill chain where the malicious actor uses the access gained through victim's machine to achieve their goals. In the case of APTs, this stage can be very quiet and may consist of several small subsequent exploits to elevate privileges, make subtle system changes, and eventually infiltrate other machines and services across the network.

According to the kill chain model, it is nearly impossible to detect malicious activity taking place at Steps 1 and 2, Reconnaissance and Weaponization, respectively. Step 3, Delivery, is the stage at which detection can occur at the network level, but as explained in Section 2.1.3.2, often the deployment mechanisms for these attacks bypass network defenses. Steps 4 and 5, Exploitation and Installation, are often initiated by a user of the soon-to-be compromised system, and at this stage, host-based resources should be able to detect malicious activities taking place. At Step 6, C2, the malware is already calling "home" and exfiltrating information about the compromised host. By the time Step 7 "Actions" occurs, the attacker or malware is

already achieving their malicious objectives. Unfortunately, it is often the case that few, if any, defense mechanisms or reactionary measures occur anywhere between Steps 4-7 [18].

2.3 Perimeter-Focused Defenses

Currently, the industry focus is on intrusion prevention through network monitoring strategies. This is evident in the prolific number of Security Information and Event Management (SIEM) systems. Many of these enterprise-wide solutions incorporate network monitoring, both signature and behavior-based IDS solutions, and in-depth, yet "passive" firewall solutions [70]. Each system claims to be a complete package due to advanced proprietary algorithms correlating IDS and firewall logs with a network monitoring solution that uses machine learning to train to recognize what is considered normal network traffic and what traffic may be suspicious [1] [6].

Unfortunately, while these solutions may stop the vast majority of intrusion attempts there will always be other attacks which slip through the cracks. These attacks could slip through by negligence or human error if network defenses aren't configured properly, or updated with the latest malware definitions. "Zero-day" attacks and those carried out through *social engineering*, can be very successful despite a system being well-maintained and up to date. Both of these kinds of attacks are commonly seen as components of APTs.

2.3.0.5 The Keys to the Castle

Layered defenses are an ideal, but not the reality. Past the outer layer of defenses consisting of firewalls and various flavors of intrusion detection systems, the fortifications at the client machine level are usually very weak, particularly where they are connected to other nodes in the same system. Once the attacker has gained access to a victim machine, it is only a matter of time before they gain access to more resources [23] [32] [78] [42].

The payload, activated by the employee, has inherited a minimum of user-level privileges, and now has insider access to the same resources as that user, including networked storage and the projects the user is working on. If the payload includes a keylogger, the attacker can then harvest the employee's credentials and perform activities later, impersonating as that employee.

Access to just a single machine in the internal network can give an attacker an enormous amount of information not seen from the outside. This includes emails and lists of contacts, knowledge of the company infrastructure, levels of permissions, certificates used, human knowledge such as company lingo and terminology, and internal organization, providing an attacker

with a much more robust set of information on new targets for social engineering.

Knowledge gained from user–level access can assist an attacker in exploiting the compromised machine to elevate privileges and attempt to move laterally, accessing and exploiting other systems throughout the victim's network.

Because of the dangers posed by these host-based attacks, it makes sense to fortify host systems with additional services to detect attacks not previously flagged by either network or host-based IDS and anti-malware solutions. However, many solutions which aim to detect OS/kernel-based malware are limited in their scope.

Operating under the assumption that the system has already been compromised, the ideal scenario is to monitor the host machine in some way to detect that suspicious behavior is taking place. There are several methods available in both research and industry, but exactly what they monitor, and how complete that coverage may be is not defined.

In order to develop a more complete solution to detecting APTs before they are able to enter the Actions Step and spread to other hosts in the victim's environment, it is useful to break the problem down into manageable units.

1. Define activities performed by rootkit malware.
2. Categorize these activities into domains of activity.
3. Apply known detection methods to each of these domains.
4. Analyze solutions for full-coverage, and identify malware-affected regions lacking coverage.
5. Propose solutions to provide full coverage. Provide metrics to evaluate full coverage.

The research presented in this thesis is tasked with the first three points: to define rootkit activities and categorize them according to domains of activity so that detection methods can be directly associated with each of those domains. These are addressed in Chapters 3, 4 and 5. The application of detection methods to rootkit activity domains is presented specifically in Chapter 6. The remaining steps are recommended to be addressed in future research and by industry, and are discussed in further detail in Chapter 7.

Chapter 3
Rootkit Preparation and Delivery

This chapter provides background information regarding the development of kernel-level rootkits prior to rootkit activities within the target system. The focus of this chapter is on the considerations and types of preparation an attacker must make for the rootkit to be effective. First, Section 3.1 provides a definition and some semantics addressing what a rootkit is, and what it is not, and addresses basic features and objectives of a rootkit from a high-level perspective. In particular, Section 3.1.1 relates the kill chain to rootkit-specific activities. Rootkits are discussed in this thesis as relating to the kill chain. Sections 3.2 "Reconnaissance", 3.3 "Weaponization", and 3.4 "Delivery" provide detail about rootkits during the first three steps of the kill chain. Subsequent steps in the kill chain are addressed in Chapter 4.

Table 3.1 provides a simple roadmap for topics discussed in this chapter.

Topic	Section
Rootkit Overview	3.1
Step 1: Reconnaissance	3.2
Step 2: Weaponization	3.3
Step 3: Delivery	3.4

Table 3.1: Chapter 3 Roadmap

3.1 Rootkit Overview

A rootkit is a type of malware that covertly infiltrates a host computer system and opens a door up for more malware to further compromise the host. Bravo and Garcia define a rootkit as *"Any software that gives continued privileged access to a computer while actively hiding its presence and other information from administrators by subverting standard operating system functionality or other applications"* [9]. Ries states that the goal of a rootkit is to provide an attacker or malicious code with a permanent and undetectable presence on a computer [63]. Rootkits are notorious for assisting APT attacks, particularly for industrial espionage [6].

Rootkits proceed through four main phases of operation: 1) covertly enter a victim's system, 2) exploit the host to elevate privileges, 3) install and persist while hiding evidence of itself, and 4) perform other simple malicious activities, which usually includes opening a backdoor for a command and control (C2) server or human attacker. These phases can be broken down into specific steps following the structure of the kill chain, as addressed in Section 3.1.1.

A rootkit can be confused or identified as another type of malware. For example, a rootkit is often considered to be the same as a *trojan*, which is malware disguised as legitimate software. Rootkits sometimes do hide within or are disguised as software applications, and trojans often install backdoors, but trojans don't always attempt to elevate privileges or enable a communications channel. Rootkits, however, can be installed via both exploits or trojans. In this way, rootkits and trojans contain some similar features and functionality.

Additionally, a rootkit and a "backdoor" are not always the same thing, even though they are often both referred to as a "backdoor". A backdoor describes an alternate channel which attackers can use to gain access to a system, but its existence may not be intentionally malicious. For example, an unintentionally misconfigured firewall could be used by attackers. A rootkit contains more functionality than a simple "backdoor", as it is capable of other activities, such as masking the presence of that backdoor.

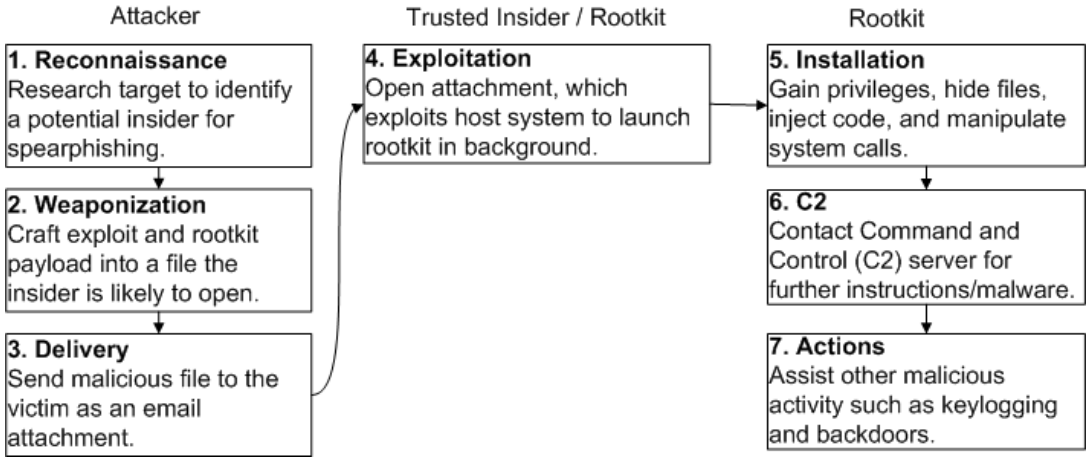

Figure 3.1: Stages of the Rootkit Kill Chain

Often rootkits are bundled with additional malware payloads which may assist in elevating permissions, opening ports, installing keyloggers, and even facilitating contact with the C2 server and downloading even more malware. These activities make the system just a little bit "noisier" and could help expose the existence of the rootkit, but rootkit discovery is by no means easy. Even more difficult than detecting a *noisy* rootkit, is detecting one that has been planted, but is merely sitting and waiting, with no ports actively open, and no additional malware processes running. Because there is almost a limitless number of possible payloads that could accompany a rootkit, for the sake of approaching some standardized model of basic rootkit detection, including dormant rootkit threats, this research focuses on rootkits with minimal functionality.

3.1.1 Rootkit Kill Chain

A rootkit's activities closely follow the same stages as any other attack, best expressed through the kill chain.

First, the attacker performs *Reconnaissance* and *Weaponization*, and then uses social engineering for *Delivery*. After successful delivery to a victim's system, a trusted insider is necessary to launch the rootkit, which attempts *Exploitation* to gain footing on the host system for *Installation*. Afterward, the rootkit contacts the *C2* server and performs malicious activities in the *Actions* stage.

Figure 3.1 shows the stages of the kill chain, as performed through rootkit activity. The stages are categorized by the actor or component initiating activity at that stage.

Figure 3.2 depicts how the kill chain directly relates to a rootkit's operational goals.

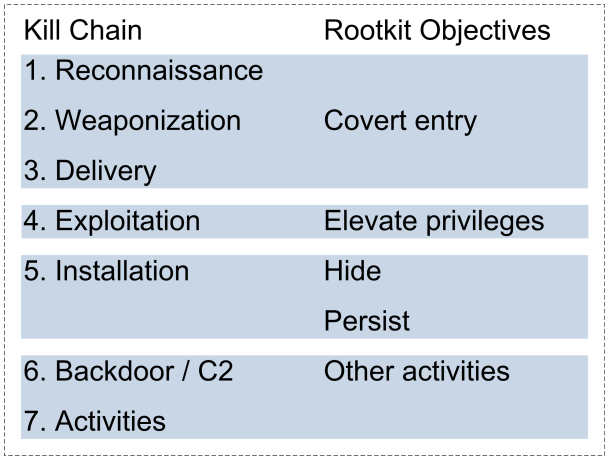

Figure 3.2: Rootkit Objectives as Related to the Kill Chain

3.1.2 Breakdown of Rootkit Activities

Subsequent sections in this chapter and in Chapter 4 roughly follow the steps of the kill chain for rootkits. These stages address the following topics of interest.

1. Rootkits specialize in taking measures to avoid triggering alarms while entering a victim system. The method of selecting unsuspecting targets via reconnaissance is addressed in Section 3.2. Methods used to allow the rootkit to bypass network and host-based defenses is addressed in Section 3.3, and delivery to the victim is discussed in Section 3.4.

2. During the installation process, the rootkit may combine exploitation tactics with installation instructions in order to gain access to protected areas within the host operating system. The installation process is detailed in Section 4.1, and exploitation strategies used to gain system privileges are briefly discussed in Section 4.1.6.

3. To maximize infiltration into a victim's computer, rootkits try to install themselves in the kernel of an operating system. A discussion of what the kernel is and how it relates to rootkit activity is detailed in Section 4.1.2.

4. The installation process also includes tactics for a rootkit to hide its code and configurations. Installation hiding methods are discussed in section 4.1.8, "Rootkit Hiding Methods".

5. Rootkits can include payloads (i.e., additional malware) to provide support functionality, but these payloads can be considered to be separate from the rootkit itself. Rather than

act as a carrier for malware, rootkits tend to remain small, and additional malware or functionality can instead be delivered to the victim through the backdoor that the rootkit creates. This backdoor is discussed in Section 4.2.

6. Once a rootkit has been installed, it often facilitates two primary activities: 1) prepare the OS to receive more malware, usually by subverting defenses, and 2) hide evidence of rootkit and related malware activity. A discussion of rootkit activity is detailed in Section 4.3. The rootkit will also use its system level access to ensure that its files and actions are not noticed by system monitoring utilities. Rootkit hiding measures which disable security services are detailed in Section 4.3.1.

The following sections discuss rootkit preparations corresponding to the first three steps of the kill chain. Taxonomies for the major concepts are provided in order to better structure the problem space surrounding each stage, and to facilitate understanding of what indicators of rootkit activity may later be detected. kill chain stages 4–7 are addressed in Chapter 4, and detection methods as related to these topics are addressed in Chapter 5.

3.2 Reconnaissance

There are several methods by which an attacker can gather information about a potential target. Assuming that the attacker has no inside accomplice, and has no other information, such as stolen credentials, they can still gain quite a bit of knowledge from public information. According to the Penetration Testing Execution Standard [57], "open source intelligence (OSINT) is a form of intelligence collection management that involves finding, selecting, and acquiring information from publicly available sources and analyzing it to produce actionable intelligence".

Generally an attacker can choose to probe external services of a potential target either via the use of freely available penetration testing tools to discover vulnerabilities in a victim's external facing systems or through the use of social engineering by convincing an insider to help provide the attacker with access.

Table 3.2 provides a high level taxonomic overview of reconnaissance techniques discussed in this section.

3.2.1 Cyber Reconnaissance

There are three types of OSINT reconnaissance: passive, semi-passive, and active. Passive information gathering only gleans information from public sources, and involves no direct cy-

Kill Chain Phase	Topic	Section
Step # 1: Reconnaissance		3.2
	Cyber	3.2.1
	Passive	
	Semi–Passive	
	Active	
	Social	3.2.2
	Passive	
	Active	

Table 3.2: Section 3.2 Roadmap

ber contact with the target's systems. Semi-passive methods lightly probe the target with seemingly-innocuous network traffic, such as internet traffic. Active methods are noisier [57]. For example, an active style of reconnaissance might probe the potential victim's entire known IP range in order to discover all externally facing systems, identify the services visibly being used, and which ports on those services are open. This kind of information is easily gathered using tools such as nmap [55], but it can be noisy, depending on the intensity and duration of the scans.

Knowledge of the victim's network infrastructure, even just at the perimeter of the network, can be useful for attacking the victim's externally-facing services. However, because external network defenses tend to be the most heavily fortified, any direct scanning or probing of external systems can be easily noticed, for example, changes in LDAP queries [18]. This is why publicly available information, such as DNS registrations or version information about servers and website content management systems are good sources of information to find potential vulnerabilities [14].

Often the option of a direct cyber attack is discarded in favor of using social engineering to initiate the attack from inside the network. This is the assumption made for the examples provided in this thesis.

3.2.2 Social Reconnaissance

When a malicious actor is planning an attack, it is best if all information gathered about the victim is collected as quietly as possible [2]. The premise of social engineering is that an attacker can persuade someone who has access to a potential victim's internal resources to provide the attacker with an easier way in. This type of attack can be carried out physically

or electronically, and with or without the insider even knowing that they helped facilitate the attack.

The quietest way to perform social engineering on an insider, such as an employee, is to engage the insider in a way such that they: 1) provide the attacker with an easy way in and 2) have no suspicion that they have done so because they think that the action they took was part of normal business or social operations. This often occurs when an insider opens a business-related document containing malware or visits a website which downloads malware in the background. These methods are discussed in Section 3.3, "Weaponization".

Information useful for social engineering is publicly available and requires little cyber reconnaissance by the attacker. For example, most businesses, government and military entities have publicly-facing websites with information about their industry, interests, partnerships, and personnel. Names, roles, phone numbers and email addresses of persons in various positions throughout these entities are sometimes publicly available.

Additionally, with social media and other online services such as Facebook, Twitter, LinkedIn, Tumblr, Blogger, and others, an employee's personal interests and social life are often available for an attacker to determine what may be considered "relevant" for the purposes of social engineering. In extreme cases, the attacker may even forge an internet presence for the purpose of befriending a potential target. The goal of this reconnaissance is to eventually get the insider to somehow provide the attacker with account credentials, knowledge of business operations, visit a malicious website, or open a malicious file from work, which will then launch hidden malware on a machine inside the network.

Social engineering strategies can either either indirectly attempt to lure potential victims to download or open malicious content, or the strategy can directly target specific individuals, with the intention that these hand-selected insiders will launch the malware on their own work machines. These strategies are discussed further in Section 3.4.1.

Once the attacker has selected their method of social engineering and determined that enough information has been collected in order to successfully launch an attack, then the attack enters the *Weaponization* phase.

Table 3.3 describes logical notation used to indicate the applicability of each branch of the taxonomies presented in this thesis.

Figure 3.3 shows the taxonomy of reconnaissance methods used to gather information about a potential victim.

Symbol	Logic	Intent
•	AND	All options apply.
+	OR	One option applies.
$\bigoplus$	XOR	One or more options apply.
OPTIONAL	These features are included at the discretion of the attacker.	

Table 3.3: Logic Notation for Taxonomies

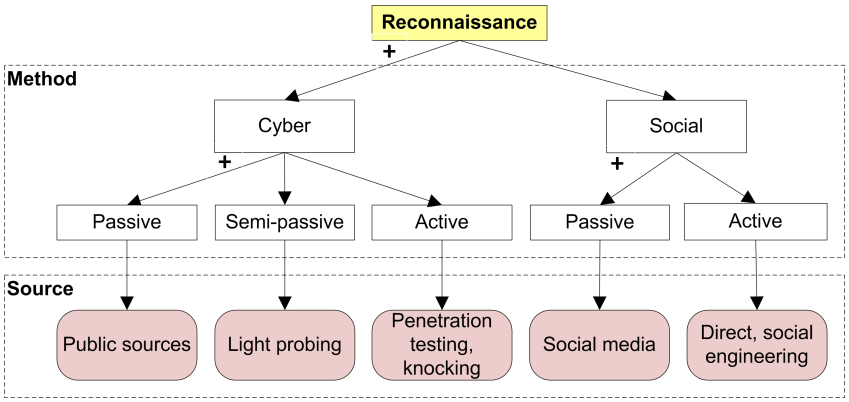

Figure 3.3: Taxonomy of Reconnaissance Methods

3.3 Weaponization

Weaponization is the process of transforming something benign into something that will compromise the security of the recipient. The delivery of rootkits is often facilitated by two highly-effective methods: 1) downloaded from a compromised website or 2) encapuslated within a seemingly legitimate attachment. Links to the website or a copy of the attachment are usually distributed via email through social engineered phishing or spearphishing attacks.

At the weaponization stage, information obtained during reconnaissance can be useful in order to determine the method of delivery, and the payload to be delivered [72].

3.3.1 Delivery Considerations

During the reconnaissance stage, the attacker has gleaned information about the target's business operations, and ideally a few target employees' job roles and interests. The attacker may choose to employ a phishing attack, and so needs to decide how to advertise the bait to the targets. This decision may lead the attacker to compromise a website, for which a URL may be sent as spam mail to company employees. The attacker may even attempt to compromise the company's own website, although this is a riskier move. Methods for compromising a website

Kill Chain Phase	Considerations		Section
Weaponization			3.3
	Delivery Considerations		3.3.1
		Weaponized Websites	3.3.1.1
		Email Distribution	3.3.1.2
		Weaponized Content	3.3.1.3
	Exploits and Payloads		3.3.2
	Exploit Kits		3.3.3
		Blackhole	3.3.3.1
		Metasploit	3.3.3.2
	Obfuscation		3.3.4
		Code Scrambling	3.3.4.1
		Compression	3.3.4.2
		Packaging	3.3.4.3
		Encryption	3.3.4.4

Table 3.4: Section 3.3 Roadmap

are discussed below, in Section 3.3.1.1.

If choosing a spearphishing strategy, the attacker needs to decide how to present the attack to the target, so that the target is cooperative and unsuspecting. In addition to emailing potential victims a URL, the attacker could directly send an attachment. The contents of the attachment will appear to be relevant so that when the victim opens the attachment, they get exactly what they expected, unaware of any background processes launched in the background.

Additionally, any attachments sent through a corporate firewall and through a business's security and scanning systems must not raise any alarms. Methods for avoiding network and host based IDSs and security systems are listed in Section 3.3.4. Additionally, stand-alone payloads, such as executables, are much more likely to be detected by antivirus products [45].

An attack which attempts to foil IDS and other security programs is known as a *content-based evasion attack*. This type of attack uses several different methods to obfuscate the actual properties of the content. These methods include manipulating the structural, lexical or temporal composition of the content, for example, emails, spreadsheets, ad-hoc documents, etc.

3.3.1.1 Weaponized Websites

If the attacker decides that the best delivery method is via a compromised URL, the attacker decides whether the website hosting malicious content is owned by the attacker or another

party [78] [21], and if the victim is to know whether or not they are downloading anything from that site [23] [78].

There are advantages and disadvantages of an attacker owning the site hosting the malicious content. Should the attacker choose to host their own site, they run the risk of being identified should that site come under suspicion. It has been observed, however, that phishing attacks from China tended to be launched from personally-owned domains rather than hacked websites [36].

Malicious actors can choose to rent space on servers available for malicious purposes, such as described in Section 3.3.3.1 on the Blackhole exploit kit. Additionally, the site could either be a simple web hosting service or a site that forges a legitimate web presence, which could require considerable setup.

Regardless of ownership, the site hosting malicious content needs to be relevant to the industry and needs of the target. Often attackers will utilize "watering hole" attacks, which are attacks on websites that are known to consistently draw users from the target industry [21]. This was a popular method that targeted government websites in 2013, and in the same year, a "burst" of Internet Explorer zero day attacks were used to facilitate these attacks as well [21].

A malicious actor who compromises an existing website gains access to all hosted content and regular visitors, as well as an established reputation in the web community. The attacker can be removed from the website at any time, but malicious content hosted may remain. The attacker can either trojanize existing content on the site, or post new malicious content for victim-initiated downloads. For example, user guides, games, and even drivers or software updates hosted on a legitimate website could be malicious. The attacker could also redirect this site to another malicious one, or weaponize the site to initiate the malware download in the background.

Many of the C2 servers used by the MiniDuke APT were legitimate websites that had been compromised to deliver further instructions to the MiniDuke malware visiting that site [79].

Because background downloads are a popular way to distribute malware, ad servers, which have a presence in many sites are a very popular type of web service to compromise, especially if the ads serve high-traffic websites. Even a single ad from a compromised ad server, loaded onto a legitimate webpage, can compromise many visitors to that page [90].

Communications protocols also each have special ways in which they can be exploited. For example, SMTP servers established as open relays can be used to spread spam, and FTP and HTTP servers can become repositories for malware, as reported by Sood *et al.* [72].

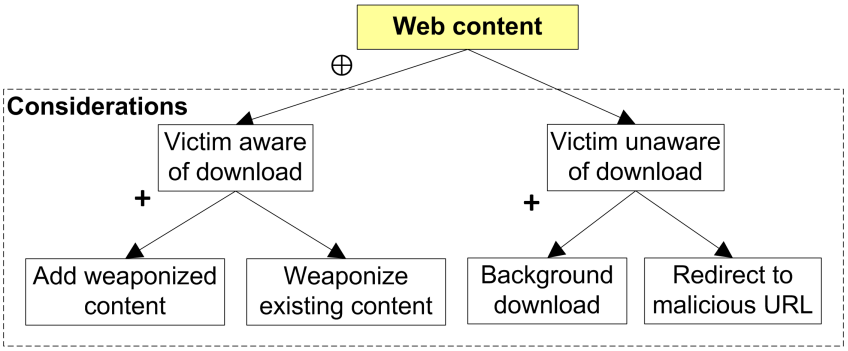

Figure 3.4: Types of Website Weaponization

If the intended target is to be made unaware of the malicious download, there is less of a need to present the victim with anything more than the website, and the malicious content must only be able to pass basic network security measures by not appearing to be a malicious executable. However, if the attacker chooses to have the victim deploy the malware, then the rootkit or other malware must be packaged as a file that the target will open.

The above strategies for a compromised website, which apply to an attacker-owned or a compromised third-party website, are reflected in Figure 3.4.

3.3.1.2 Email Distribution

In addition to an attacker taking over a website as the distribution channel for content, email is the other primary method of distributing malware, and is considered to be highly effective, and can easily slip past network security services such as firewalls and IDSs [72].

Bulk spamming entire companies is increasingly more difficult to carry out due to spam blacklists and rules that can detect mass, indiscriminate distribution. Smaller sets of specific email targets are more likely to make it through to a human recipient, which is why spearphishing is desirable for implanting malware. According to the Verizon 2014 Data Breach Investigations Report [78], spearphishing was the most prolific method used to gain access to a victim's environment.

The primary concerns for an attacker aiming to submit malicious content to a recipient are how to disguise themselves, the sender, as non-suspicious, and whether they are sending the target a link to a website or an actual attachment.

Identity: The identity of the sender of the email can be forged quite easily using an exploit kit, as described below in Section 3.3.3.

URL: The URL address contained in the email can either be an honest representation of the

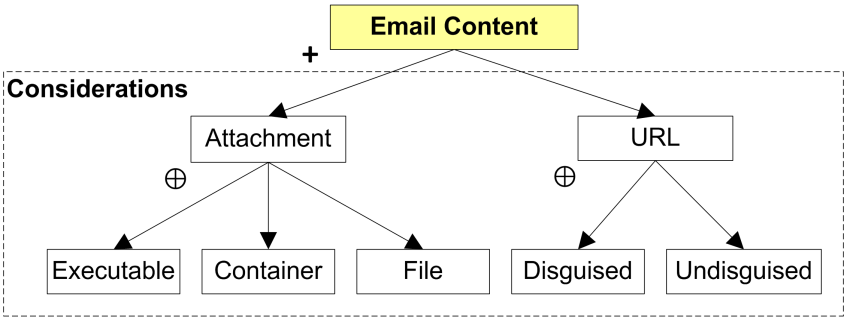

Figure 3.5: Types of Email Weaponization

web address, or the URL could be disguised in emails displaying HTML or some other graphical format. Although the use of forged addresses can be successful, often businesses concerned with security will block HTML and formatting content and only display the text presented. In this case, the target may select the URL presented, and visit a legitimate site, not the one the attacker intended.

Attachment: Attachments sent to the target must be appropriate for the victim to be more likely to open the file than to be suspicious. There are three main types of attachments. The attachment could be an *executable*, such as a trojanized software program. In this case, the victim is aware that a program is about to run on their machine, and therefore they are more prone to caution than with other attachment types. The attachment could be either a file or a software program embedded in a *container* of sorts, such as a zip or rar file. The program used to unzip or decompress the contents is usually the carrier for the exploit. The attachment could be a *file*, like a spreadsheet or a PDF document, which will exploit whatever version of the program that the victim has installed to open the document. Through this method, the victim is less likely to associate any amount of danger with the act of opening the file. These attachment types are discussed in the following section.

While these are considerations for developing defenses, detection mechanisms for these items are not addressed in this thesis.

3.3.1.3 Weaponized Content

There are three types of content which can be used to distribute malware: *executables*, *containers*, and *files*. There are functional differences between these types in terms of detectability, the type of exploit used, and even the likelihood the target will open the attachment or suspect it to be a culprit if a rootkit or malware is detected later.

Executables: Executables attempting to pass through network–based IDS usually face

high levels of scrutiny, and so an attacker would need to take extra care to ensure that it passes all likely network–based IDS. An executable which is known to be malicious and which does not use obfuscation is likely to be stopped by signature–based detection systems, as discussed in Section 5.2.1. Even if obfuscation is used, the behavior of the executable may reveal its malicious intention through defense tools which pre–execute the binary to observe its behavior before allowing it to pass through the network perimeter [69].

Some networks employ virtual machine introspection (VMI), which prevents any executable from passing the perimeter until it has executed it on a virtual machine to monitor its behavior. A file which does not execute as expected, or which causes unallowable behavior on that virtual machine will not be forwarded on to the intended recipient. Executables with delayed malicious activity, or those which only respond to certain environmental conditions may run, but not display any malicious behavior [69]. Section 5.4.2 describes VMI in more detail.

Falsifying the file extension may also raise flags at the network level, but can also prevent the executable (and embedded exploit) from even running if the recipient or system does not recognize how to open the file. There are methods by which an executable can prevent itself from being run in a virtual environment, but these methods are also more likely to result in the attachment being blocked rather than delivered.

One of the best ways to deliver malware via an executable is to inject malware into an existing legitimate executable, such as a driver, or a common office product add-on or update. It would perform as expected on a VMI appliance - minus additional functionality. A downside to this method is that known software and executables can be compared to a published signature of the original content (e.g., MD5, SHA1). Any changes to the content will result in the creation of a new signature, and if network defenses or the intended target check these signatures, the executable will not be delivered.

The ZeroAccess rootkit was deployed via trojanized copies of a DivX Plus 8.0 keygen and copies of the game Skyrim [90]. Both of these executables targeted persons attempting to access commercial software without paying for it. As a result, victims were more likely to download the executables from home and were not likely to be protected by enterprise-level network defenses.

Containers: The attacker can choose to embed the malicious content in a zip or rar file. One advantage to this approach is that the container can hold any kind of content and thus has no known signature to check, unlike executables. However, VMI implementations can still unpack zipped or compressed files and run the contents. Additional security could be added

to the compressed package by encrypting it with a password, but this can raise more suspicion by the recipient, and any unnecessary complexity added to the delivery process can cause the recipient to not want to open the file.

The container can contain a malicious executable, trojanized files, and even hidden files with malicious functionality. The weapon itself could be contained in any content within the container or a hidden executable which launches at the time the container is opened, exploiting the host-based extractor and starting the exploitation process on the target's machine.

Files: The most common carriers for malware are standard office documents, such as .doc(x), .xls(x) or .pdf files. Content distributed via these files is a natural part of conducting business and can be customized for the intended target's current business circumstances or interests. It should be noted that the content of the file is only relevant to convince the target to open the file.

When the victim downloads or launches the file from their inbox a common host-based application appropriate to that file type will attempt to open the file. The malicious content embedded in the file or document will then exploit a weakness in that host application. The type of exploit used depends on the expected host application. For example, Adobe Reader version 9 can be exploited to produce a reverse TCP shell.

There is a large number of possible exploits and classifying host-based application exploit vectors is not attempted in this thesis. However, regardless of the type of exploit used, the goal for the malware or rootkit is to attempt installation, which is discussed in depth in Section 4.1, "Rootkit Installation".

3.3.2 Exploits and Payloads

There are two main components to the weaponized package: the *exploit*, and the *payload*. An *exploit* is carried out by taking advantage of some weakness in a software program or operating system. The *payload* is the malicious software which can install and run after the system has been exploited.

There are two different uses for exploits in the deployment of a rootkit to a victim:

1. The first exploit occurs when the victim opens the malicious content and a vulnerability in the host program allows the hidden malware to start executing in the background.

2. The second exploit is usually contained in the payload and it launches as part of the rootkit installation process to provide the rootkit with elevated privileges.

The *payload* is the rootkit and all helper functions and files. Rootkits tend to be small; their addition to the file or application should not affect the expected size significantly, and because rootkits establish a connection to the attacker's server, additional malware can wait until the rootkit has established itself and created a friendly environment for new malware installation.

The possible range of vulnerabilities an exploit can target or the range of potential payloads which may be included are outside the scope of this thesis, and so they are not discussed in depth here.

3.3.3 Exploit Kits

Often rootkits are deployed from a suite of tools designed for use in for malware creation and exploitation. Exploit kits are an effective way for malicious actors with minimal level of technical experience to distribute malware. This section introduces two tools which greatly assist an attacker with both the weaponization and delivery process.

3.3.3.1 The Blackhole Exploit Kit

The Russian-developed Blackhole exploit kit is a framework for exploits and social engineering tools, and was responsible for generating the vast majority of spam and hacking in 2012 [27]. Blackhole also included a Browser Exploit Pack (BEP) which included tools to compromise and host malware on websites. Exploits developed in Blackhole could be crafted for specific operating systems, target country, browser, and other criteria [72].

Blackhole was used to plant some notorious malware including *Zeus*, the *TDSS* rootkit, the *ZeroAccess* rootkit, and various ransomware [27].

Blackhole is unique in its business model: rather than charge the user a flat rate to purchase the kit Blackhole and use of the Blackhole server is "rented" to the malicious user for a monthly fee. The actor responsible for the creation and marketing of Blackhole was arrested in Russia in 2013 [78] but there are many tools similar to Blackhole available in the wild.

3.3.3.2 Metasploit

Metasploit is a popular penetration testing tool which often comes preloaded alongside other exploitation tools in the penetration testing OS Kali Linux, and can be used to manually craft and deploy exploits and payloads [46]. An alternative to the Blackhole exploit kit, Metasploit and its related tools are free to use, although, unlike Blackhole, server space is not provided.

The Metasploit framework (MSF) contains a database of exploits targeting known vulnerabilities in software programs and operating systems, as well as a database of payloads to further compromise the target. Weaponization can be performed both manually and through an automated interface. Manual weaponization uses `msfexploit` and `msfvenom`, which load exploits and payloads, and obfuscates attack code. Automation is achieved through the use of the *Social Engineer Toolkit (SET)*, a collection of tools designed to simplify the social engineering weaponization process, which is integrated with Metasploit to utilize the exploits, payloads, and other tools within Metasploit.

SET presents Metasploit functionality in a menu-based format (as opposed to a command line interface) with the intent to simplify the social engineering weaponization process. Menu options include commands such as "`Mass Mailer Attack`" and "`Website Attack Vectors`", and make it easy for a non-expert adversary to craft a malicious payload into a file and mail it anonymously. The Mass Mailer Attack gives the attacker the option to send an email to a single address or to recipients in a mailing list. Metasploit's SET also includes attacks specifically for Arduinos, SMS messages, wireless access points, and other attacks, including the Infectious Media Generator, which enables a CD/DVD/USB to auto-run malicious content when loaded [49].

3.3.4 Obfuscation

Obfuscation is a method by which malware morphs or rearranges its code so that it cannot be assessed to be malicious by standard detection mechanisms. This is typically achieved through code scrambling, encryption, or by packaging the malware inside another object. Some obfuscation efforts go as far as to implement code to inhibit debugging or virtual execution of the binary [72].

The most common tools that aid the process of obfuscation are *packers* which compress the malware and reduce its size. *Packagers* embed malware inside legitimate software, and *crypters*, which encrypt the malware, both tools changing the signature of the code [72].

3.3.4.1 Code Scrambling

Obfuscation by scrambling rearranges the order of the code so that it is less likely to generate a known signature but the functionality of the code remains the same. Each of Metasploit's `msfvenom`'s scrambling algorithms is named and ranked within the application. One popular obfuscation algorithm is the `shikata_ga_nai` algorithm [**?**].

Polymorphic malware commonly automates code scrambling for each successive copy in such a way to change its signature [74]. However, some types of "polymorphism" can also be achieved through manual encryption, or repackaging [66] [28].

3.3.4.2 Compression

Packers compress malware, changing the signature and reducing the size. Malware is usually packed with a type of complex, polymorphic packer [90]. RogueAV is one example of malware which uses a dynamic packer, effectively evading signature-based and some pattern-based defenses [85].

3.3.4.3 Packaging

Packagers bundle malware within legitimate software, turning the application into a trojan [85]. Injecting malicious code into a legitimate application is also called "trojanizing" the application [83] [45] [9]. Trojanizing is a popular method of distributing malware within highly popular binaries, but is more likely to be detected in enterprise environments which verify the legitimacy of applications through integrity verification [72].

Malware is often embedded within common business applications, such as Adobe PDF, Microsoft Word, or even ZIP files. Due to the variable nature of the content of these documents, a signature is not infeasible to compare against for integrity checking [66]. For example, an email sent to Mandiant employees in 2012 was suspected to originate from China's People's Liberation Army (PLA) Unit 61398, also known as "APT1". The email contained a malicious ZIP file called "Internal_Discussion_Press_Release_In_Next_Week8.zip", which contained a custom backdoor [42].

3.3.4.4 Encryption

Crypters can be used to obfuscate software by not only rendering the signature detection ineffective, but also by making the code unable to be analyzed for functionality if the target's detection system debugs or executes binaries externally prior to delivery. Even though functionality cannot be determined, the detection of encryption alone can raise suspicion [72] [51]. Additionally, the code must be decrypted before it can be executed, making the installation method just a little bit more complicated.

According to the Websense 2013 Threat Report, Blackhole used "criminal encryption", which is described as a type of encryption that anti-virus engines and deobfuscation tools have

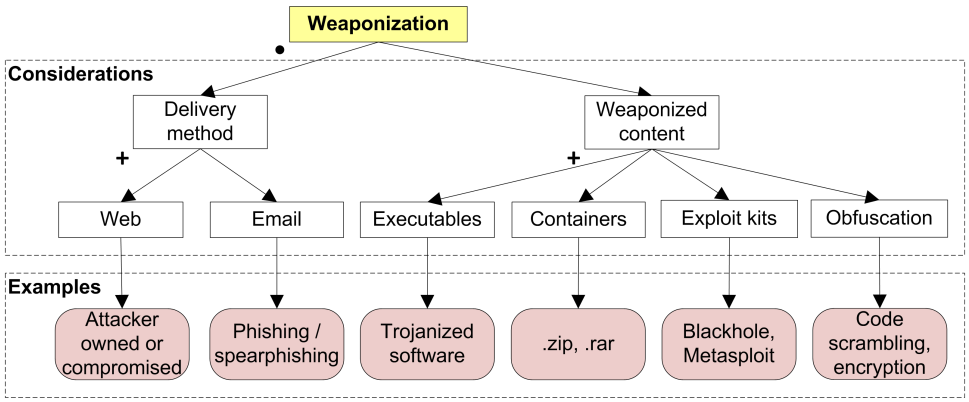

Figure 3.6: Taxonomy of Weaponization Methods

trouble detecting [85]. However, the presence of an unknown encryption should raise suspicion.

Figure 3.6 shows the taxonomy of weaponization methods used to prepare a malicious package to both host the rootkit and not raise suspicion by the target.

3.4 Rootkit Delivery

Delivery: *Delivery* can be defined as the stage in which the rootkit is transmitted, successfully enters the victim's network, and eventually resides on the victim's machine. The method of transmitting the rootkit ideally raises no alarm by any firewall rules, network or host based IDS, or by any other defenses. From the perspective of the network, there is no indication that the rootkit or its container is in any way malicious. Additionally, if the rootkit requires human intervention in order to install, then it must be delivered so that the recipient knows of its presence and also believes it to be benign and necessary so that they will eventually take action to open it and (knowingly or unknowingly) launch the installation process.

The Weaponization stage addresses mechanisms which can be built into or around the rootkit, such as obfuscation, to prevent it from being stopped by both network and host-based defenses, as well as social engineering strategies which can help effect delivery. The Delivery stage addresses the application of some social engineering strategies as applied to both active and passive delivery methods, and physical and cyber delivery methods, in order to ensure that the rootkit is safely delivered to the victim.

This section discusses various delivery methods by which a rootkit could be transmitted and received to the target machine. Rootkit transmission methods can be boiled down to active or passive methods used in conjunction with physical or cyber methods.

Kill Chain Phase	Considerations		Section
Delivery			3.4
	Active vs. Passive Delivery		3.4.1
		Active	3.4.1.1
		Passive	3.4.1.2
	Physical vs. Cyber Delivery		3.4.2
		Cyber	3.4.2.2
		Physical	3.4.2.1

Table 3.5: Section 3.4 Roadmap

- **Active** delivery directly interacts with the target in order to deliver malware.
- **Passive** delivery requires planting malware in a way such that the target is likely to inadvertently stumble upon or discover the malware and install it themselves.
- **Cyber** delivery only requires digital access, usually through a network, in order to deliver malware.
- **Physical** delivery requires physical access to some aspect of a victim's hardware in order to deliver the malware.

Both active and passive delivery can be facilitated by cyber or physical means as discussed in Section 3.4.2. Likewise, both physical and cyber delivery can each be actively or passively distributed.

The methods discussed below are not all equally likely to be implemented, but are important to discuss in order to understand the range of options for malware delivery. The rest of this thesis, starting with Section 4.1 "Installation and Exploitation", simplifies delivery assumptions to the scenario of a rootkit embedded in a seemingly relevant email attachment, delivered via spearphishing.

Table 3.5 provides a high level taxonomic overview of weaponization techniques discussed in this section.

3.4.1 Active vs. Passive Delivery

Malware delivery can be evaluated by the level of interaction the attacker chooses to have with individual targets.

3.4.1.1 Active Delivery

Active malware delivery is ideal for situations necessitating a specific target and thus more direct interactions.

Rather than hoping for some arbitrary victim to inadvertently click a link or stumble upon a relevant website, actors choosing to use active delivery know who the victim should be, and how they may likely become infected. Often active delivery is preceded by high levels of reconnaissance so that more information about the victim will help ensure the success of the delivery. For example, this includes information about the target's political or personal interests for more effective social engineering and knowledge of the target's hardware and/or operating system to ensure the rootkit can effectively install.

Social engineering via spearphishing has proved to be a useful active delivery method [85] [78]. For example, an attacker could email a malicious PDF document containing legitimate information about upcoming tax changes to personnel in the Human Resources department of a company, or send a malicious link about a mutual hobby to an unwitting social media "friend" on the inside to share with their supervisor. An attacker can use a document posted online by a legitimate business, or even on the target's own website or blog, add the malware to the document, and then send it to the target [13].

In 2009 the GhostNet rootkit was distributed via an email containing "contextually relevant" information with a malicious attachment that dropped a trojan connecting to six different C2 servers in China and Hong Kong [37]. Famous APTs Flame, Stuxnet, and Red October are believed to also have been initiated via spearphishing messages to specific individuals within certain targeted organizations [85].

In 2011, political figures in Hong Kong received a high volume of emails containing malicious documents, usually during major political events. For example, several malicious attachments were delivered while massive rioting was occurring in Guangzhou province. The timeliness of these emails led investigators to believe that the goal of these attempted APT attacks was political espionage. The attackers were highly invested in their social engineering attempts, even scheduling political meetings in order to send relevant content in the spearphishing emails and attachments. File extensions for these emails included `.pdf`, `.doc(x)`, `.xls(x)`, and `.chm` [37].

3.4.1.2 Passive Delivery

Passive delivery methods are less concerned with who the end victim is, as long as the malware is delivered somewhere. Distribution methods are more broad than in active methods, such as bulk spam or phishing emails, or a compromised website. The victims could be indiscriminately chosen, such as spam email sent to as many addresses as the attackers can obtain. Victim selection could be narrowed down to a target industry or business resulting in phishing emails or watering hole attacks.

One example of an effective passive delivery method involves attackers setting up a malicious website, as discussed in Section 3.3.1.1. When a victim visits the site, either by casually landing on the page or directly clicking a malicious link, the website's server backend will attempt to launch an exploit against that machine. Common exploits for Windows target applications such as Internet Explorer, Adobe Acrobat, Flash Player, and Java [90].

Passive attacks often consist of "phishing" attacks, where the attacker sends an email containing a malicious URL to every employee within a company, or a "watering hole" attack where the attacker hijacks a popular niche-tech website so that visitors end up with malware downloading in the background [78] [21].

Other passive methods involve planting compromised files on peer-to-peer (P2P) services, establishing rogue Wi-Fi services, and even the possible compromise of shared storage such as cloud services [72]. The type of delivery method is also indicative of the goals of the attacker. Spam messages tend to lead victims to a variety of highly dangerous exploits, while phishing tends to be very specific, primarily for banking or backdoors and botnets [85].

The lines can be blurred between active and passive delivery methods when the target business is specific, but the individual targets are indiscriminately chosen. For example, the Aurora attack, which targeted Google, was distributed via a malicious URL sent to Google employees via phishing emails [67].

3.4.2 Physical vs. Cyber Delivery

Both active and passive methods of malware delivery can also employ both physical and cyber delivery strategies.

3.4.2.1 Physical Delivery

Physically-delivered malware can be *Actively* planted through either direct physical access or indirectly through the supply chain. While this thesis focuses mostly on cyber-only strategies, it is important to note that emails and websites are not the only places from which malware can be sourced.

Physical access is often not considered when planning cyber defenses, but a single piece of compromised hardware connected inside the DMZ can be uncontested by security defenses. Physical delivery methods highlight the importance of vetting supply chain vendors and contractors, as both can access physical resources with less scrutiny than in-bound network traffic.

Active-physical malware delivery requires direct physical access to a specific target's hardware resources. For example, a malicious contractor doing network upgrades could intentionally launch malware from a connected device, or knowingly install malware on a laptop being repaired.

The supply chain can also act as a player in the distribution of malware. According to FireEye, the supply chain is one of the top ten vertical targets for organizations worldwide [21]. International supply chains are frequently utilized by vendors of hardware components. Hardware is assembled in one country, with parts originating from other countries, and is then re–branded, marketed and sold in yet others [67].

An adversary with influence in a foreign manufacturing facility could take advantage of an organization's purchase request to manufacture PC chips by adding malicious circuits which initiate keylogging. A malicious vendor could choose to ship that same malicious hardware to a target customer [3] [67].

Malware compromise through physical delivery can also be facilitated through *passive* means. A system can be accidentally compromised if a third-party contractor performs system maintenance unknowingly using a previously compromised device, or unknowingly installs an adversary-compromised hardware device. This is a real danger for companies who permit devices to connect to the company network that are not held to the same standard of patching, maintenance, and vetting as company-maintained devices. The supply chain can also be a vector for indirect malware distribution. Honest vendors may ship unknowingly compromised hardware from a malicious supply chain manufacturer to a target customer. Vendor credibility alone does not mean that hardware will not introduce malware threats [11].

	Active	Passive
Physical	Insider PnP abuse Malicious vendor Malicious 3rd party contractor	Employee PnP/hardware misuse, Compromised vendor, Compromised 3rd party contractor
Cyber	Spearphishing Malicious insider Cyber attack	Phishing & spam Employee email/web abuse Web drive-by, watering-hole Connected services (e.g. cloud)

Table 3.6: Active-Passive and Physical-Digital Delivery Examples

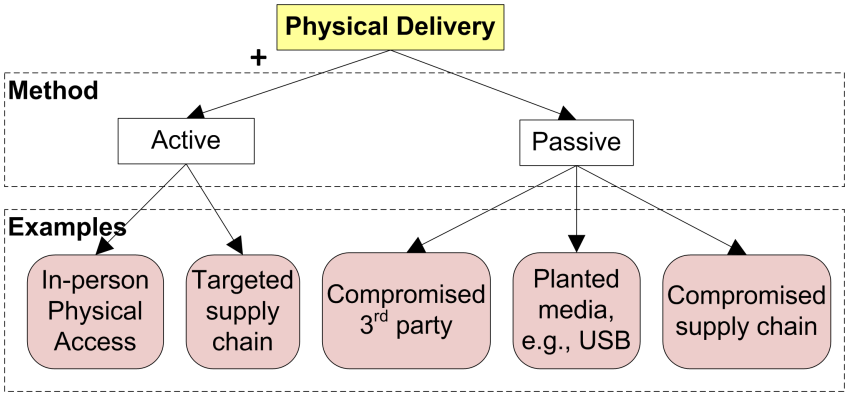

Figure 3.7: Taxonomy of Physical Delivery Methods

In addition to supply chain and third party contractors, another major vector for malware infection results from employee misuse of company resources. One of the most prevalent passive forms of physical malware infection results from unintentional insertion of compromised physical media, such as USBs and other Plug-n-Play (PnP) devices, which are considered misuse via "unapproved hardware". Employees can also misuse resources by taking company hardware (e.g., a work laptop or external storage device) home, or utilize cyber misuse by checking personal email and browsing unsafe sites from within the company firewall [78].

Examples of malware spread via USB include the `W32/Fanbot.A@mm` worm, which also spread itself by email and P2P services [75], the Stuxnet worm [79], and quite possibly the Flame bot [79].

3.4.2.2 Cyber Delivery

Cyber delivery encompasses any method of malware delivery which does not necessitate physical means. Cyber-only malware delivery is most often performed through web-based malware downloads and through emails containing malicious URLs or compromised attachments. How-

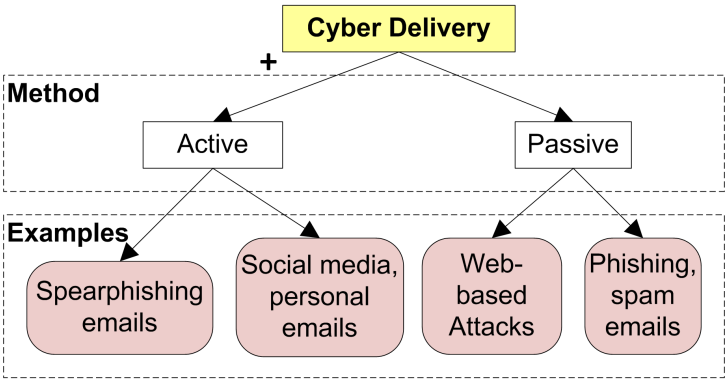

Figure 3.8: Taxonomy of Cyber Delivery Methods

ever, cyber-only delivery can occur through any network-connected service on a user's machine. This includes synchronized cloud services, such as Dropbox [15], P2P services, Bluetooth [91], and even Wi-Fi interactions [2].

Table 3.6 lists some of the malware delivery methods which can result from combinations of physical-cyber and active-passive delivery mechanisms.

Figure 3.7 shows the taxonomy of physical delivery methods and Figure 3.8 shows the taxonomy of cyber delivery methods.

After the malicious package is delivered to the victim, the next stage in the kill chain is for the package to be opened so that the rootkit may begin installation. Chapter 4 addresses rootkit activities after it has been successfully delivered to the victim's machine.

Chapter 4
Kernel–mode Rootkits

There are several different types of rootkits, such as those that affect hardware, run only in memory, or remain entirely in user mode. This thesis focuses on "kernel–mode" rootkits, which primarily operate through kernel and system feature manipulations and related resources.

This chapter provides further detail on kernel–mode rootkit activities once inside the target system. Descriptions of the regions affected and resulting behaviors are provided along with taxonomies for the mechanisms used to compromise a system.

Section 4.1 details both rootkit installation and exploitation actions, and comprises the bulk of rootkit activity. Section 4.2 discusses rootkit C2 channel and communications. Section 4.3 details rootkit attack activities as an extension of rootkit capabilities. Section 4.4 discusses specialized rootkits, particularly those that are not limited to kernel–mode activities, and rootkits which specifically target industrial control systems. Table 4.1 provides a simple roadmap for topics discussed in this chapter.

Chapter 5 addresses the background and taxonomies of rootkit detection mechanisms, as related to rootkit information discussed in this chapter.

Kill Chain Phase	Section
Steps 4 & 5: Installation and Exploitation	4.1
Step 6: Backdoor Creation	4.2
Step 7: Rootkit Activity	4.3
Specialized Rootkits	4.4

Table 4.1: Chapter 4 Roadmap

Kill Chain Phase	Topic	Section
Installation / Exploitation		4.1
	Operational Goals	4.1.1
	The Kernel	4.1.2
	Installation Techniques	4.1.4
	Installation Process	4.1.5
	Rootkit Privilege Escalation	4.1.6
	Rootkit Persistence	4.1.7
	Rootkit Hiding Methods	4.1.8

Table 4.2: Section 4.1 Roadmap

4.1 Rootkit Installation and Exploitation

Stages 4 and 5 of the kill chain, Installation and Exploitation, directly follow rootkit delivery. These stages are discussed together in this section because the installation process involves alternately planting files and in turn subverting the victim's system in order to obtain necessary privileges for further installation. Where rootkits are concerned, exploitation is a natural part of the installation stage. Application–specific exploits are not the focus of this research, but kernel-level workarounds are exploits of rootkit installation activity, and are addressed.

There are two major stages to rootkit installation: the first stage requires the malicious package to be executed, through which arbitrary exploits against applications on the host machine grant privileges to allow the rootkit to install itself. The second stage is the installation, which can take many forms and infect many locations on the victim machine, both in user space and at the kernel level. Low or kernel level manipulations that allow the rootkit to gain additional levels of access to further its installation may or may not be executed through kernel space or through additional application–specific exploits.

Table 4.2 provides a roadmap for topic discussed in Section 4.1.

4.1.1 Operational Goals

Prior to discussing the actions that take place during rootkit installation, it's necessary to understand the intention behind each action. As discussed in Section 3.1, rootkits have four primary stages of operation: 1) covertly enter a victim's system, 2) exploit the host to elevate privileges, 3) install and persist while hiding evidence of itself, and 4) perform other simple malicious activities.

Section 3.4 "Delivery" addressed rootkit entry to the target's system, and Section 4.3, "Rootkit Activity", addresses additional malicious activities. This section addresses the in-between stages, which are all a part of rootkit installation.

Each activity performed at the installation stage prepares the rootkit to fulfill some aspect of its operational goals. The following are rootkit operational goals, which guide the rootkit installation process. The only operational goal that is not specifically handled at installation is that of backdoor creation.

1. **Rootkits attempt to gain elevated privileges.** A rootkit is rarely intended to act alone, but as a vehicle for other malware or adversaries to exploit. Regardless of the mode into which the rootkit process was started, through exploits the rootkit gains access to kernel and OS resources. Privilege escalation is gained during the installation process, and is further discussed in Section 4.1.6, "Rootkit Privilege Escalation".

2. **Rootkits aim to persist.** In order to remain useful as an APT asset, a rootkit will do whatever it can to ensure that it survives reboots, configuration changes, and even attempts to remove it directly. This is performed by hiding its file, processes, and even integrating itself into the system so that removal could be devastating to the system as a whole. Rootkit persistence methods are discussed in Section 4.1.7, "Rootkit Persistence".

3. **Rootkits attempt to conceal their presence.** The first mission for a rootkit is to deploy without being detected. A rootkit will do whatever it can to hide its own existence from process lists, the directory structure, and will even attempt to disable programs that it knows may expose its processes or attempt to remove it. Details on rootkit hiding methods are discussed in Section 4.1.8, "Rootkit Hiding".

4. **Rootkits create a backdoor.** A rootkit will often be sent alone, to serve the covert act of introducing a vulnerability into a system so that other malware or the adversary may

later achieve access and further the exploit process. Backdoors can consist of an open network port, a compromised network-facing application, or even a vulnerability for an attacker to exploit. The backdoor is often some form of network channel for a remote server connection to send further instructions or malware downloads. The method by which a rootkit establishes backdoors is handled separately from initial installation, as Step 6 of the kill chain, and is discussed in Section 4.2, "Backdoor Creation".

In order to understand the methods rootkits use in order to infiltrate a victim system, it is necessary to understand the space in which installation and exploitation take place. Section 4.1.2 discusses the kernel, which is where the most critical rootkit activities take place. The following sections discuss strategies rootkits use to take action within the user space. Section 4.1 concludes with specific ways that rootkits combine these methods in order to fulfill their operational goals.

4.1.2 The Kernel

The kernel is the part of the operating system which handles low–level instructions, and facilitates I/O operations between the OS and hardware.

Many successful rootkits exploit kernel features, because custom instructions executed through the kernel can manipulate the flow of information through the operating system. This allows a rootkit to modify the way that the system responds to commands by altering the content or calling order of kernel objects or by placing new instructions or jumps to attacker–placed code.

This section provides a simple overview of the structure of the Windows and Linux kernels, and then discusses kernel features which are most commonly used by rootkits to fulfill malicious objectives.

Table 4.3 provides a high level view of kernel structures discussed in this section.

4.1.2.1 Windows Kernel

The Windows kernel is a microkernel, where several utilities reside in user space, and then interface with the kernel to complete requests. User space services communicate to kernel level services through device drivers and application interprocess communication (IPC).

The Intel x86 architecture is designed with four rings as layers of permission or protection around the kernel. However, Windows systems use only two of the rings: Ring 0 is kernel

Kill Chain Phase	Topic	Section
Installation / Exploitation		4.1
	The Kernel	4.1.2
	Windows Kernel	4.1.2.1
	Linux Kernel	4.1.2.2
	Terminology	4.1.2.3
	Kernel Structures	4.1.3
	Kernel Objects	4.1.3.1
	Address Tables	4.1.3.2
	Libraries/APIs	4.1.3.3
	Configuration Files	4.1.3.4
	Drivers	4.1.3.5

Table 4.3: Section 4.1.2 Roadmap

mode (highest privilege) and Ring 3 is user mode (lowest privilege). Each of these modes use virtualized memory addresses through a hardware abstraction layer [39].

User–mode is primarily utilized by high level applications, which typically require user interaction and run with lowest privilege. User mode applications include standard third–party applications, services, and applications which a user interface.

Windows user–level applications are each assigned their own *process*, *virtual address space*, and *handle table*. The address space of user–mode applications is limited to prevent access (and possible damage) to OS data in the kernel address space [47]. Because an application's virtual address space is private, one application cannot alter data that belongs to another application.

Crashes of user level processes generally only bring down the affected process, and are recoverable as they do not affect the core operating system [76] [47].

Kernel–mode describes low–level operating system applications and processes, which run at highest privilege. The kernel mode address space in Windows is shared by all kernel–level processes, making it easy for a single corrupted kernel–level process to corrupt the memory space of other processes [76]. Due to this simple, two–layered architecture, code running in kernel mode is implicitly trusted [73].

Driver crashes at the kernel level can bring the entire system down. This is one reason why kernel–mode malware is typically more difficult to write, as any obvious signs of system malfunction could raise alerts to its presence [47].

4.1.2.2 Linux Kernel

Linux is a monolithic kernel system with applications residing in user space and the entire operating system residing in kernel space. The Linux kernel is similar to the Windows kernel in that the hardware utilizes two main levels of privilege. Consisting of a 2–ring layered architecture, Ring 0 is considered to be kernel/supervisor mode (highest privilege), and Ring 3 is user mode (least privileged). Permissions level are divided into a less-privileged user space and a high–privilege "root" space which has access to system resources. Although user and root permissions roughly parallel those of the ring architecture, the relationship is not precise, and context switches to those of the rings themselves are more the domain of hardware–based rootkits [81], which are not discussed in depth in this thesis.

User space in Linux–like systems is the domain of user applications, system daemons, and some application–specific libraries.

Kernel space in Linux is the residence of device drivers, the virtual file system (VFS) and virtual memory, memory management, the scheduler and IPC.

Kernel–mode rootkits in Linux operate similarly to that of Windows kernel–mode rootkits and use techniques that modify resources that may in themselves be defined as user–mode, kernel–mode or the system. Therefore the space in which the rootkit is operating is more defined by the resources accessed, such as libraries or dynamic kernel modules, and less that of an assigned privilege level. Kernel–mode rootkits manipulate resources at all of these levels, provided an exploit has provided access to that resource.

The Linux kernel is modular, and so specific functionality, such as drivers and file system accesses can be implemented as kernel modules [65].

4.1.2.3 Terminology

The following terms are used frequently throughout this thesis. Although the use of one term may appear with greater frequency in reference to one particular operating system, these terms can generally apply to the same concept in other operating systems. It should be noted that the majority of literature focuses on Windows rootkits, and thus the majority of examples presented in this thesis apply to the Windows domain.

- **Kernel mode** is the term used to refer to activities primarily occurring at the kernel level of an OS, but which also frequently use user space and OS resources. The level of permission assumed to access this broad range of regions is frequently referred to in

literature as "kernel–level" [9] [77] [73] [63]. According to Vasisht *et al.*, kernel–mode rootkits operate with "unrestricted accesses at the root privilege level" and can manipulate any component of the system [77].

- **Kernel object/kernel module** are significant components within kernel space. Often this term refers to a function, or piece of code with specific features or functionality. Kernel objects/modules can be callers or callees at the kernel level of an operating system.
- **Kernel code** consists of both code and instructions inside kernel modules and data structures. Kernel code dictates the logic by which instructions are executed.
- **Kernel data** are the value of variables or returned values of kernel objects or other data sources. Kernel data is semantically relevant information by which other modules and kernel objects and applications make decisions.
- **System calls** are the method by which applications or programs request a service from the kernel, including requests for hardware access, process creation and execution, and scheduling. These calls can be made by objects within the kernel, although this term usually refers to a service outside the system making requests for the system to handle a service.
- **Application Programming Interfaces (API)** are the building blocks for software programs, including parameters, tools, and protocols. Each platform's API provides a set of functions which can be used to create and modify software programs and other platform–dependent utilities.
- **Inter–Process Communication (IPC)** is the process by which applications and services communicate with each other, in particular where the individual process memory spaces are virtualized or cannot be accessed via memory address alone.

4.1.3 Kernel Structures

This section discusses the structures and utilities used by rootkits. The categories of kernel structures discussed below are present with some variation in both Windows and Linux–based operating systems.

4.1.3.1 Kernel Objects

The term "kernel object" can be used to describe several different types of kernel data structures and devices, including high level device drivers and the file system, processes, threads, events, semaphores, queues, and timers. Kernel objects can also be both static or dynamic [73]. Although this term may be broadly applied, kernel objects are generally considered to be either part of the functionality of the system kernel, or are modular and extend the functionality of the kernel.

The accessibility of kernel objects depends on the level of privilege of the service calling the object. User level applications in Windows do not reference kernel objects directly (i.e., by pointers) but rather by handles, which provides some layer of security between the user layer and the kernel–space data structure. However, in kernel mode, objects can be accessed directly, by their pointers. The **Access Control List (ACL)** maintains each kernel object's handle, and interfaces with the Security Reference Monitor (SRM) to ensure objects are only being accessed by assigned permissions [73]

In Linux, the system call interface (SCI) mediates user level application access to kernel level structures. The system call lookup table contains the addresses for kernel objects,

Each module has parameters. These parameters can provide additional functionality that can change their behavior. If the module is already running, existing parameters must be unloaded before new parameters can be loaded, or new custom parameters can be directly loaded. Each module also has dependencies, either calling or being called by other modules, and for the most part tends to interact frequently with a certain subset of other relevant modules.

In Linux, dynamic kernel modules are generally called "loadable kernel modules" (LKM) [4] [84] [77], while in Windows the same term is applied more specifically to objects such as device drivers [73]. LKMs are compiled as object code and may be ready for use when needed.

In user mode, applications can crash and restart with minimally corrupted user–level data. LKMs, when crashed, can bring the entire system down. Regardless of OS, the kernel is a difficult domain to exploit, as malware and legitimate modules need to operate nearly perfectly in order to avoid a system crash. This is especially true for rootkit activities, as any slight abnormality caused by the code can reveal the presence of non–standard functionality.

4.1.3.2 Address Tables

Address tables within the kernel provide pointer information to kernel objects and user level process handlers. There are three primary tables which are used by rootkits: SSDT, IAT, and ICT.

- **Import Address Table (IAT)/Global Offset Table(GOT)**:
  The IAT (Windows) and GOT (Linux) are global tables which contain offsets to the pointers for library functions [81]. Windows user level applications call libraries using a function handle to access addresses indirectly [73] [63]. The GOT similarly resolves addresses without requiring function handles.

- **The System Service Descriptor Table (SSDT) or System Call Table**:
  The System Service Descriptor Table (SSDT) is a dispatch table in Windows which contains the pointers to system functions (APIs) that implement system services (located in `ntoskrnl.exe`). This is a kernel-level address table which the System Service Dispatcher uses to find a system service's code [63]. The Linux equivalent is the System Call Table [4].

- **Interrupt Descriptor Table (IDT)**:
  The IDT is a data structure in Windows that stores the handlers for software interrupts [73]. The CPU checks the IDT when interrupts occur in order to determine what code will be called to handle that particular event [63] [77]. The Linux equivalent is also called the Interrupt Descriptor Table [4].

4.1.3.3 Libraries/APIs

The Linux API consists of system calls and subroutines which programs use to access system resources. Libraries contain frequently-used code that can be shared by multiple services within an operating system.

Windows uses dynamically linked libraries (DLLs) which include functions and data that other modules can use. The Linux equivalent to Windows' DLLs could be the Executable and Linking Format (ELF) files, which is the shared object (.so) format.

4.1.3.4 Configuration Files

Configuration settings stored for diverse applications are often contained in a standardized region of the OS. The Windows Registry is one highly useful domain for configuration information

for applications and procedures. In Linux, configuration information is often stored in the `/etc` folder or in an individual application's initialization(`.ini`) or configuration(`.conf`) files.

4.1.3.5 Drivers

Drivers are objects which enable communication between applications and I/O devices. One notable type of driver is the *filter* driver, which exploits the Windows layered device driver architecture. A filter driver placed between two layers of drivers can intercept and censor information in a way that benefits the attacker [73].

In the Windows microkernel structure, kernel–mode device drivers can perform direct memory access (DMA) through which they have the ability to write directly to physical memory [87]. Memory protection mechanisms may be in place to ensure that a user–mode driver does not attempt to write within kernel space, but these mechanisms are imperfect [51].

4.1.4 Installation Techniques

When a rootkit installed, it plants malicious code or instructions via one of the three following types of code placement. These forms will become important later in the discussion of rootkit detection, as some detection methods are better than others at detecting different types of rootkit code. After introducing these methods of kernel level code placement, specific kernel-modification strategies are discussed. Each strategy identifies the domain of activity as related to kernel structures, and the type of code placement used.

Injected code or data. This type of rootkit code is planted inside an already existing host, with the assumption that when the host runs, this code will be executed as part of the host's instruction set. However, rather than writing over existing instructions, injected code is planted as *additional* functionality, rather than replacing existing functionality.

Overwritten or altered code or data. In this scenario, the rootkit actively changes the instructions within a kernel module, binary, or even the address of a pointer [84]. Sometimes the overwritten code is lost forever, but with many rootkits, the original code is stored elsewhere to be called via a malicious filter [44] [81]. This is commonly implemented as pointer overwriting in address tables to modify execution flow [84] [73].

Standalone code or data. This code takes some form of relatively complete, self-sufficient code, which implements instructions on its own. This can take either the form of a kernel module with malicious instructions, a malware executable, or even a brand new entry in a table

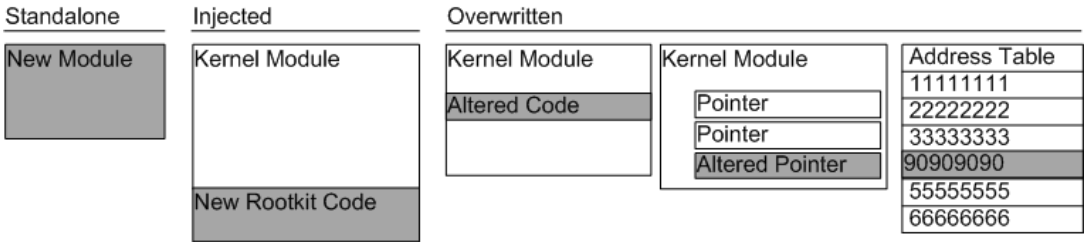

Figure 4.1: Data/Code Modification Types

or registry. Standalone code is new, occupying previously unallocated space. Standalone data can be planted with legitimate data to be used to create malicious logic.

Using complex combinations of these three code planting styles, rootkits can achieve their objectives.

Although a rootkit can only perform kernel level activities through incrementally gained access via successive exploitation actions, the installation process is not always linear. The activities presented in the following sections, unless otherwise specified, may be performed at any stage in the rootkit installation process, and as a result, are not presented chronologically.

Additionally, the following activities which help fulfill rootkit objectives, while commonly used by many rootkits, are not all used by all rootkits. Rootkit activities discussed in this section are primarily described in terms of installation activities, and additional post-installation functionality is discussed in Section 4.3.

The techniques described below provide a broad overview of how kernel manipulation activities can be combined in order to perform some of the most common rootkit activities, These techniques are not exhaustive, but provide a broad range of information to which detection techniques, discussed in the next chapter, can be applied.

These activities can be simplified to a) placing hooks in the system in order to b) control the flow of information and actions performed by the system. The term "**hooking**" can be used to imply that a rootkit has modified a system feature such that it has overwritten or placed code within that feature such that it has control over some aspect of that feature's activities. There are several types of features which can be hooked, such as kernel modules, address tables, and configuration settings, as described below. **Control flow modification** ultimately is how the rootkit controls activities within the kernel, by hooking processes such that they deviate from the actions originally intended in order to execute malicious activities on behalf of the rootkit.

Table 4.4 provides a high level view of kernel structures discussed in this section.

Kill Chain Phase	Technique	Section
Installation / Exploitation		4.1
	Installation Techniques	4.1.4
	API Detouring	4.1.4.1
	DKOM	4.1.4.2
	Configuration/Registry	4.1.4.3
	Code Injection/Patching	4.1.4.4
	Filtering	4.1.4.5
	Process/Thread Injection	4.1.4.6
	Malicious Logic	4.1.4.7

Table 4.4: Section 4.1.4 Roadmap

4.1.4.1 API Detouring

API detouring, also known as a "redirection attack" or "control flow attack", is an abuse of system APIs in order to modify the execution flow of system instructions. Kernel level rootkits implement execution redirection both through hooks and injection [51].

API detouring is generally started by overwriting a pointer in one of the system address tables, from where rootkits can either access and make modifications to kernel code, or change the order of instructions. API hooking must be handled carefully, because if an instruction is called, and the malicious codes does not return to the original (or next) intended address, the entire system could crash. Despite the caution required to implement API detours, this method is shown to be highly effective. Wang *et al.* reported that 96% of Windows and Linux rootkits were persistent kernel–mode rootkits which used control flow modifications [84].

Rootkits "Aphex" and "Hacker Defender" utilize API detours to modify the return address on the stack so that as the called API returns, it calls the malicious code, so the results can be modified to benefit the rootkit before fully returning [4] [83]. ProBot SE, a keylogging rootkit modified the Service Dispatch Table entries for kernel-mode [83].

API detouring usually starts with modifications to tables containing addresses for system API functions and kernel modules.

- **SSDT**:

  The System Service Descriptor Table (SSDT) is a table in Windows which contains the pointers to resolve API system calls for system services (located in `ntoskrnl.exe`).

  The rootkit overwrites an address in this table to point to its own malicious code, and so

the operating system will run that malicious code with highest privileges [63] [76]. SSDT hooking can be used to intercept requests for the registry, file system, processes, threads, and even memory [63].

The pointer in the SSDT could either be redirected to the attacker's own code, somewhere in memory, or in order to create malicious logic, the pointer could redirect to a different API than the one originally intended. It only takes a single address overwrite in the SSDT in order for a rootkit to gain global access to the entire system [73] [62].

Additionally, some rootkits do not change the address in the SSDT but change the assembly instructions of the first bytes in the target API (in ntoskrnl then) to point to the hooking module. This is a different method, called an *inline hook*, which is covered later in this section [39].

- **IAT**: The Import Address Table (IAT) is a table in Windows which contains pointers to all the functions that a particular program might use. IAT deviation, also known as IAT "patching" is a popular method for user-mode rootkits to gain kernel-level access. If an attacker is able to overwrite or create pointers within this table, when the targeted program is executed, the first address it jumps to can actually jump to whatever function the attacker specifies. The Urbin and Mersting rootkits perform IAT deviation to point API calls to a trojanized import function instead [83]. This method can also be used to hide files, as the files displayed within a directory are listed via IAT-listed API functions.

- **IDT**:

  The Interrupt Descriptor Table (IDT) handles interrupts, the handlers for which can be intercepted and forwarded on to rootkit code [82]. IDT interrupts are often used for keylogging, as they can intercept keyboard input interrupts [77].

Address table modification attacks fall into the category of **pointer manipulation** attacks, which overwrite a pointer to redirect calls to that address to rootkit code instead. Pointer attacks can either overwrite address table pointers, or overwrite pointers within a kernel module. Execution flow can also be modified if a pointer to rootkit code also is added to the sequence of instructions.

Code redirection can also occur through **inline function hooks**, which place a JMP instruction over the first five bytes of an API function, along with the 32 bit address of where to jump.

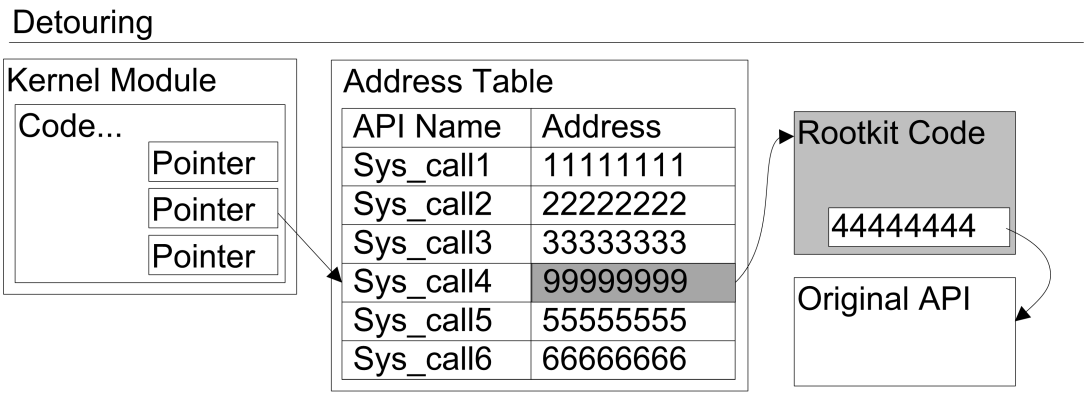

Figure 4.2: API Detouring

4.1.4.2 Direct Kernel Object Manipulation (DKOM)

Direct kernel object manipulation (DKOM) is the term used to encompass any sort of attack that manipulates kernel objects within some arbitrary OS. This includes modification or addition of kernel objects or their contents, malicious calling sequences, and attacks which affect or change the perception of an object's existence or liveness [58]. Kernel level rootkits use DKOM, as it is highly effective and not easily detected [51].

There are several ways that kernel objects can be manipulated to benefit a rootkit. The simplest method is by overwriting a pointer in one of the system address tables, and redirecting control flow either to an attacker-planted malicious kernel module, or to a different system module to create malicious logic flow [73]. Additionally, kernel code itself can be injected with additional instructions or overwritten, but DKOM attacks avoid these tactics as they are easily detected [73].

DKOM attacks make good use of existing kernel data structures, often by "hooking" an existing kernel module , either by overwriting a pointer to a function or injecting malicious code so that its activities and actions are implemented when that module is called. Kernel hooks include mechanisms that load code or data into memory modules [9]. Both the ZeroAccess and TDL rootkits hooked processes in critical functions [72]. For example, ZeroAccess overwrote 704 bytes of `ScRegisterTCPEndpoint` from "services.exe" [44]

Kernel objects can also be duplicated, with malicious functionality added to the copy. Rather than redirecting calls to malicious code and then calling to the original module, a malicious duplicate can provide both original functionality with malicious features. Duplicate kernel objects can include drivers, modules, and even calling patterns.

4.1.4.3 Configuration / Registry

Configuration settings are an ideal location for rootkits to hook further into a system. The Windows Registry is a popular location for rootkits to hide configuration settings and persistence information.

In a sample of 30 malware, Wang *et al.* found that each malware used at least one Registry hook, of which most were hidden to prevent removal [83]. Registry APIs `Advapi32!RegEnumValue` and `NtDll!NtEnumerateKey` are commonly hooked in order to hide not only files, but to intercept and filter Registry query information that may reveal the existence of the rootkit's Registry settings [83].

Auto Start Extensibility Point (ASEP) hooks can load and run processes on startup, load drivers, or even hook multiple processes at the same time, through a shared resource (e.g., a DLL) [4] [83]. Rootkits Urbin and Mersting hook the `AppInit_DLLs` ASEP to load their DLL into every process that loads `User32.dll`, and then hide the ASEP hook [83]. Hacker Defender hides its ASEP hooks for the service `hxdef100.exe` and the driver `hxdefdrv.sys` [83].

4.1.4.4 Code Injection / Patching

Injection or "patching" attacks add functionality to an existing module or system call. This is performed by either adding code or instructions to an existing module or API, or by modifying the control flow to other services. This attack can utilize API Detouring, Overwriting, and Injection, depending on how it is carried out.

Injection attacks which use API detouring can be performed through the following methods:

- **Return–oriented programming**, also known as a "return to library" technique, involves overwriting the return address of a library routine to create a chain of malicious logic that concludes by returning to the original return address location. This method makes use of sets of existing machine instructions ("gadgets"), generating a malicious result through the instruction calling order [7].

  Return–to–libc is a common return–oriented attack which overflows a buffer to overwrite the address to point to a particular function in the `libc` library. Overwriting stack data can then result in the flow of several subsequent addresses to functions called such that it results in a malicious flow of logic [81].

- **Jump–oriented programming** builds up an attack through chaining functional gad-

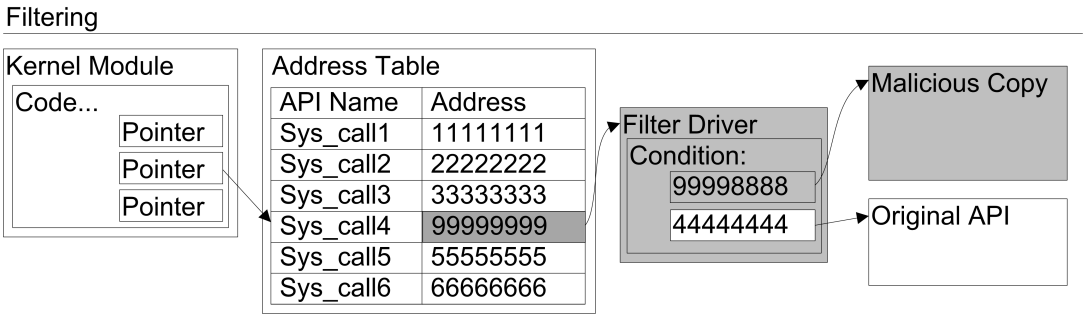

Figure 4.3: Conditional Filtering

gets, but ends indirectly (no return), relying on the dispatcher to schedule and execute the malicious logic.

In addition to initiating detours through an OS, it is also possible to place additional code directly inside kernel objects. However, this type of injection, also called "patching", is much more difficult to do than generating detours. This is because it is much harder to find and work with raw code than it is to simply find the address of a module. When the code is modified, nothing critical can be overwritten or added, and the injected/patched code must be in non-paged memory, because if the rootkit code is temporarily moved to disk, it can cause an unrecoverable fault [63].

4.1.4.5 Filtering

Kernel drivers can be used not only to interface with devices, but play a key role in kernel-level rootkits. According to Musavi *et al.*, "most rootkits in the wild prefer to use malicious drivers" [51].

Rootkits which filter data often attack the API interface for device–independent services like `read` and `write`, and lower level drivers that serve as an abstract interface to the hardware below. *Filter* drivers can be placed between high and low level drivers to intercept and filter data between the layers [73]. Filter drivers can intercept user data, like log keystrokes or intercept passwords entered through network activity. The file management interface in particular is attacked to hide files or directories [73].

Due to Windows allowing a layered driver functionality, each driver filtering data from the driver one layer above, it is easy for rootkits to also implement their own filter driver [63]. Filter drivers may also be installed into the networking stack to create low level backdoors [63]

4.1.4.6 Process/Thread Injection

Malicious processes can also be loaded into new or running threads. One of the advantages to loading a malicious process directly into a running thread is so that the process does not only run when hijacked system modules are called. This effectively hides the running code from the process/module query APIs [83]. Sparks *et al.* describe the Win32 API as providing a "Rootkit API" as user mode functions `CreateRemoteThread` and `WriteProcessMemory` can be readily injected by an attacker. Thread and process resources may be accessed in Windows through the Object Manager, Process & Thread Manager, or the Memory Manager, which also handles paging [73]. Using the `libc` library, attackers can easily create new threads and launch "arbitrary" processes [81].

A political espionage malware, upon installation, called the `CreateRemoteThread` API to inject its malicious "`msvcr.dll`" directly into the running process `explorer.exe`. This malicious DLL waited for three more malicious binaries to be downloaded from the C2 server, and then took the initiative to launch at least two of them [37].

4.1.4.7 Malicious Logic

Any of the activities a rootkit takes within the OS ultimately creates a series of malicious activities, which can be considered "malicious logic", however this term is more specifically applied to a scenario where legitimate, unmodified system modules are called in an order that produces malicious results, such as that of return–oriented or jump–oriented programming. However, a malicious outcome can be effected not only by the calling order of a set of instructions, but also by data values submitted by compromised modules.

Semantic value manipulation (SVM) is malicious logic created by modifying or filtering data inputs and outputs so that the system behaves as the attacker desires. Because of the vast number of modules and processes within an OS which produce outputs and make decisions on the state of other reported modules, the attack space for SVM attacks is "vast" and full coverage detection is not an easy task to achieve [58]. The PLCs in the Stuxnet attack were programmed to return values within the desired thresholds while the centrifuges were sent commands to operate outside those defined limits. This prevented the system from responding appropriately, and so alarms were not raised, and no counter–instructions were sent to the centrifuges to bring them back within limits [79].

Almost every rootkit installation technique can employ the basic three installation methods:

Installation Technique	Injection	Overwriting	Standalone
API Detouring	Injected pointers add functionality to a module.	Overwritten pointers in an address table.	Standalone APIs add new or faux-duplicate functionality to OS.
DKOM	Injected modules contain additional instructions.	Overwritten modules perform tasks in new malicious ways.	Standalone modules planted somewhere in memory.
Code Injection / Patching	New code or instructions placed inside existing modules or data structures.	Overwrite existing pointer in table or module (API detouring).	N/A
Configuration / Registry	Injected configuration parameters can add additional rules.	Overwritten configurations operate under altered parameters.	Standalone entries add new processes or tasks to startup.
Filtering	Injected drivers or APIs filter data cause malicious logic.	Data overwritten in existing drivers can conditionally filter calls or data.	Standalone drivers filter messages between other drivers and modules.
Process / Thread Injection	New malicious processes can be injected into a currently executing thread.	N/A	New threads can be created and loaded using Win32 API/`libc`.
Malicious Logic	Injecting new instructions or pointers to both legitimate (OS) and malicious modules/APIs can effect a malicious sequence of events.	Overwriting code or data causes malicious instructions to execute when called.	New modules or malicious copies of system modules/APIs perform malicious actions.

Table 4.5: Core Installation Methods Applied to Rootkit Installation Techniques

injection, overwriting, and planting standalone code, as depicted in Table 4.5. This is important to note as these basic three techniques are the foundation for detection as discussed in Chapter 5.

The techniques discussed in this section can now be applied to the actual rootkit installation process, discussed in the following section.

4.1.5 Installation Process

Rootkit installation involves breaking out of the carrier application, unloading its code and processes, and performing activities which help prepare it to fulfill its mission effectively and covertly.

Some malicious payloads are relatively self-executing (e.g., many viruses and worms), but rootkits which have been planted via social engineering methods require that a user or program on the victim system initiate the rootkit installation process. For example, a PDF with a malicious payload, which has been emailed to the victim, will not execute itself, but requires that 1) the human user attempt to open the file manually, after which 2) an application on the host machine associated with that file type starts its own processes to execute this file.

The rootkit is launched when malicious instructions embedded within that file take advantage of a vulnerability within the associated application. These embedded exploits usually target specific versions of the expected application. For example, CVE-2013-0640/0641 exploits Adobe Reader, and allows malicious code to bypass sandboxing [21]. CVE-2013-0633/0634 used embedded Flash exploits to bypass sandboxing in Microsoft 2008 Office files [21]. The MiniDuke APT was able to exploit all versions of Adobe Reader as well [79].

Table 4.6 provides a high level view of kernel–level rootkit objectives achieved through the use of methods discussed in this section.

4.1.6 Rootkit Privilege Escalation

According to Musavi *et al.*, there are two methods used to get into kernel space: use a driver or use an exploit that targets an unpatched vulnerability [51]. The strategy used to escalate privileges also can depend on the access level into which the rootkit is opened. Exploits used by the rootkit to gain elevated access to the victim system can be either a continuation of the original container exploit, some arbitrary secondary application exploit, or with the use of drivers, can directly tap into kernel resources and solidify a high-level foothold deep inside the system.

If the rootkit is loaded into an environment with restricted privileges then often an arsenal of arbitrary exploits may be necessary to gain sufficient privileges. This is expected, and so the privilege escalation exploit is usually another aspect of the application-specific exploits which allowed it to start installation in the first place. These exploits can take almost any form.

For example, privilege escalation exploits can be authentication-based, using stolen or default credentials or bypassing authentication altogether. For example, MSF exploit module `exploit/windows/http/SolarWinds_fsm_userlogin` is an application-specific exploit which allows an authentication bypass on systems running Solarwinds Firewall Security Manager 6.6.5. [48].

Kill Chain Phase	Objective	Considerations	Example	Section
Installation / Exploitation				4.1
	Privilege Escalation			4.1.6
		Mode	User Mode	
			Privileged Mode	
		Method	Malicious Driver	
			Exploit / Vulnerability	
	Persistence			4.1.7
		Non-volatile storage	Config. settings, file system, kernel loaded	4.1.7.1
		Difficult to access	Kernel loaded	4.1.7.2
			Outside file system	
			Configuration settings	
		Dangerous to access	Kernel loaded	4.1.7.3
		Restore copy	Local backup	4.1.7.4
			Network / C2 backup	
	Evade Detection (hiding)			4.1.8
		Physical Evidence (within file system)	Obscure location	4.1.8.1
			False identity	
			Unbrowsable location	
			Unenumerated	
			Injected / trojanized	
		Physical Evidence (outside file system)	Nonvolatile memory location	
		Conceal process Evidence	False Identity	4.1.8.2
			False intent / misreporting	
			Unenumerated	
			Disable security services	
			Injected / trojanized	

Table 4.6: Installation Objectives Techniques and Roadmap

If a rootkit is initially loaded into user space, in order to perform lower level activities, it often must exploit a victim process. A user-mode rootkit can also hook system API functions, usually performing IAT deviation, and overwriting code in order to detour activity lower in the

kernel [9].

According to Musavi *et al.*, kernel-level drivers are one of the primary approaches used to penetrate Windows systems [51]. To gain privileges, the 32-bit installer for the ZeroAccess rootkit loaded its code into the kernel by overwriting an existing driver [90].

Rootkits may come bundled with multiple privilege escalation options, depending on the privileges into which it is launched. Exploits which assume Administrator privileges may be attempted first, as these attempts are less complicated, and thus less risky. For example, the 64-bit installer for ZeroAccess first attempted to call `RtlAdjustPrivilege` to give itself `SE_DEBUG_PRIVILEGES`, which would be successful if the user were logged into the computer as Administrator. However, if this failed, ZeroAccess would then copy a legitimate version of Adobe Flash Installer, load malicious code into that space, and then generate a User Account Control (UAC) pop-up to require an Administrator to authorize the installation of what appeared to be a legitimate version of Adobe Flash [90].

Kernel mode rootkits are loaded into memory as a dynamic kernel module, and then the kernel module uses detouring or modifies system data structures to carry out actions. As a module operating as part of the system kernel access and privileges are gained merely by navigating the flow of system calls and carrying out further kernel level exploits, such as overwriting pointers to redirect to malicious code that furthers the rootkit's hold in the system.

4.1.7 Rootkit Persistence

Approximately 96% of rootkits establish some kind of persistence in order to achieve their goals. Persistence can be defined as a rootkit having altered control flow such that the rootkit is not easily removed or disabled, or the rootkit's core functions can be easily restored after removal [81]. A rootkit's objectives for gaining persistence require that it survive a) system reboots, b) attempts to disable it, and c) attempts to remove it, including those which may occur by the system. Remaining undetected is critical for a rootkit to withstand being disabled/removed by both the user and the system. Methods used to conceal rootkits are discussed in Section 4.1.8.

Methods of obtaining persistence ultimately boil down to a rootkit placing code in 1) non–volatile locations that are either a) difficult to access, b) are critical to the operation of the system and therefore dangerous to access or 2) by storing copies by which the rootkit may be reinstalled, either from local code or via the C2 channel. The rootkit must somehow also ensure

that the rootkit processes restart when the system returns to operation, usually through boot scripts or adding the rootkit to the OS's list of startup applications.

4.1.7.1 Non-volatile Storage Locations

The first criteria for a successful rootkit installation is one that it will survive system reboots, and so kernel–mode rootkits aim to persist by not running in memory, but rather store instances in places that are unlikely to be overwritten or erased. As such, a rootkit is installed such that it is not dependent on volatile memory and likely to be swapped out through paging, but is instead stored in non–volatile memory, on the hard disk.

4.1.7.2 Difficult–to–Remove Locations

Rootkit persistence locations include those that are difficult to remove, often because the location is either nonstandard or complicated for individual users to navigate.

Configuration settings can help a rootkit gain persistence for both existence and functionality. For example, kernel modules in Linux can store persistent configuration settings in `/etc/modprobe.d/`, which facilitates loading the affected modules directly into the kernel [61].

The Windows Registry is a highly valuable location for rootkits to both hide and initiate their services. To survive and start running after a system reboot, rootkits plant a key in the registry called an Auto Start Extensibility Point (ASEP). Some registry keys which can be hooked include `HKLM\SYSTEM\CurrentControlSet\services` which auto-starts drivers and services, and `HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run` which auto-starts other processes [83].

Other storage locations that are difficult to access include uncommon folders within the file system, in storage locations outside the browsable file system, or as parts of the system kernel itself. In addition to selecting a good non–volatile location to place rootkit code, other strategies for the concealment of rootkit code and objects are used to ensure that these storage locations remain undiscovered.

One common location rootkits reside in the Windows file system is in a user's AppData folder, which only requires user-level privileges [6]. However, with Administrator or system privileges, any location within the file system is a viable storage location, provided other rootkit utilities, particularly any services responsible for launching the malicious application know the address or file path. Some rootkits will even store drivers within hidden file systems and then reload them into memory after reboot [51].

Vogl *et al.* demonstrated that, although difficult, it is possible for malware to only inject data to gain persistent functionality through the use of return oriented programming (ROP) [81].

4.1.7.3 Dangerous–to–Remove Locations

Rootkit persistence in critical regions are more dangerous to access because they tend to be closer to the heart of the operating system. For example, a rootkit installed as kernel objects must be handled carefully. If adding, modifying, or removing these modules results in broken links or bad logic at the kernel level the entire system can crash.

With adequate permissions, an attacker or malware can easily load a new Linux kernel module using `modprobe` [61]. Some rootkits can use the `/dev/kmem` and `/dev/mem` interfaces, which provide access to non–virtualized system memory, to write directly into the kernel [4] [82].

Injecting code or adding function calls into kernel modules is more difficult to perform than loading and unloading entire modules [63]. Removing rootkit code injected into essential system modules could present even more difficulty; if the operation is not performed perfectly, missing data or unexpected functionality could result in a system crash.

4.1.7.4 Reinstallation / Restoration

If the bulk of a rootkit is discovered and removed, it may persist if copies of the rootkit are still accessible. These copies could be stored in a location that automatically reinstalls the rootkit if it is detected as missing at boot time.

For example, the MiniDuke APT created encrypted payloads of itself, using a custom key based on the CPU, drive, and computer name of the victim machine. Because this payload was encrypted it was difficult to discover. If rootkit functionality was removed MiniDuke could decrypt its own payload and reinstall itself [79].

Similar to MiniDuke's host–based backup strategy, a rootkit compromised U.S. Air Force web server was discovered to contain up to seven bootable kernels. Each kernel was a successive upgrade to the previous kernel, and could be reverted with whatever compromises existed at the time of that kernel's operation. The only difficulty in restoration on this particular machine, however, was that the attacker would require manual intervention to initiate the kernel reversion process [45].

An example of networked backups, the drivers for the Festi rootkit are memory resident, and are downloaded and reinstalled from the C2 server every time the machine boots [51].

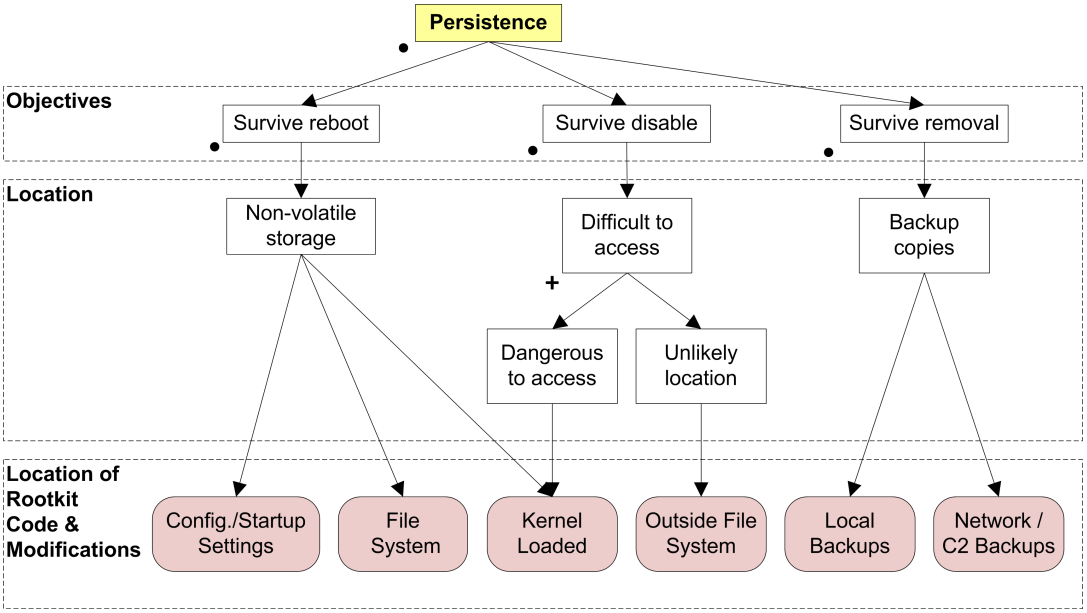

Figure 4.4: Rootkit Persistence

Other rootkits, such as BIOS rootkits, are able to survive reinstallation of the OS. However, these are outside the scope of this thesis and are not discussed in depth here.

4.1.8 Rootkit Hiding Methods

Part of the survivability of a rootkit depends on its ability to stay hidden. Rootkits employ several techniques to hide their physical presence and their active processes from services that might disclose evidence of malicious activity.

4.1.8.1 Physical Evidence

Methods a rootkit can use to conceal its files, binaries, and other hijacked features are diverse and require extensive knowledge of the domain in which it will be installing. Rootkit resources can be placed in the normal directory structure, or any similar known and enumerable location, and then hidden by preventing file enumeration APIs from displaying the rootkit files [4] [83]. Rootkit resources can also be hidden within nonstandard locations on disk, the location of which only the rootkit could naturally know.

Hidden in File System Rootkit code and files can either be hidden "in plain sight" where a user or system process could theoretically be able to locate the rootkit code within enumerable regions, or the rootkit can use kernel–level exploits to make the rootkit undiscoverable. The strategies the rootkit uses depend on if the rootkit attempts to hide its files or if it tries to falsify its identity. Multiple concealment methods may be used together as multiple layers of

security for the rootkit.

- **"Plain Sight": Obscure Location**

  Files can be hidden in plain sight by planting the malicious files among many other files in directories which are unlikely to be accessed or noticed by the user. This strategy also includes registry entries and directly linked kernel objects for which enumeration utilities exist.

  Objects merely hidden from a user's view are still vulnerable to discovery by host-based malware scanners. The use of encryption and compression can greatly reduce the chances of detection through automated tools, although the very presence of obfuscated contents of code can be suspicious [62].

- **"Plain Sight": Identity/Intent Falsification**

  It is a common technique for kernel rootkits to use file masquerading, by which the rootkit modifies its name or properties to falsify its identity as benign or even useful software [51]. For example, the Tapin worm (`W32.Tapin`) copies itself as `paint.exe`, and then modifies the attributes of other helper executables to "Hidden" and "System" (reserved for system use and not to be modified or removed) [38].

  Rootkits can also use obfuscation on files and binaries in an attempt to appear benign to malware detection tools that perform code emulation [62]. However, obfuscation is not often used in a rootkit's kernel–level drivers because of the need for debugging, maintenance and crash dump analysis [51]. These techniques are discussed in Section 3.3.4, "Obfuscation".

- **Hidden: Unbrowsable Locations**

  Locations in the file system which file enumeration utilities do not consider to be part of the directory structure are convenient places for rootkit code to hide. For example, ZeroAccess placed files inside the Global Assembly Cache (GAC), the contents of which are interpreted as cache entries, and not a browsable directory structure. However, this technique only hides files from users browsing directories via the Windows GUI. These entries are viewable through the command line interface and would not prevent host–based file enumeration tools from viewing the contents of the GAC [90].

ZeroAccess rootkit also hid files by first creating a directory under `%systemroot%`, and then using the Windows API `ZwCreateSymbolicLinkObject` to convert the directory to a symbolic link, making it inaccessible to other programs [90]. The Okray rootkit stored its SSH keys as ASCII text in file `root/dev/srd0` which is commonly understood to be a directory, not a file [45].

The existence of a static malicious kernel module can be effectively erased through pointer manipulation. Pointers to the kernel module are not listed in the IAT or other tables containing addresses to modules that a particular function may call. Rather, the attacker often stores the module in a nonstandard memory location and then hard-codes the pointer to this module within some other rootkit-owned memory location.

- **Hidden: Forcibly Unenumerated**

  One of the more effective techniques rootkits can employ is that of hijacking file enumeration APIs so that when the contents of a directory or registry entry is requested rootkit code and objects are not listed [39]. Resources that can be hidden through API enumeration hijacking include files, folders and registry entries [83]. This is often performed through trojanizing IAT file enumeration APIs so that physical evidence and dynamic rootkit processes fail to be listed [84]. Additionally, abusing the use of APIs `FindFirstFile` and `FindNextFile` from the `Kernel32` DLL can skip over any rootkit objects which would normally be listed.

  To hide Windows Registry entries, rootkits can hook the Registry API functions using custom drivers which target the *NtEnumerateValueKey* API, which is the routine that retrieves the value entries of an open key. If the name of the value to be hidden is found, the API is called a second time with instructions to omit that particular value from the list shown. [76]. Rootkits can use Registry–specific APIs such as `RegEnumKey`, but for more universal effectiveness rootkits can also choose to hook system-wide NT API functions such as `NtEnumerateKey` [83] [64].

  Another file hiding method which is particularly effective takes advantages of restrictions on file names which are unallowed by the Win32 API but not the NTFS file system. These files are planted as part of the NTFS file system, but files with long full pathnames, trailing dots or spaces, special characters, etc. are not considered legitimate, and as a result ommitted by the Win32 enumeration APIs [83].

The same methods rootkits use to hide files and processes can be used either indirectly or intentionally to hijack antivirus or malware scanners' enumeration utilities. This ensures that rootkit files are ignored before they can be evaluated [39].

- **Hidden: Injected**

  Rootkits can inject code into a legitimate application or process to either execute when that process runs, or to store code or data for another rootkit process. Rootkits can also plant or create a compromised copy of the original program, either to replace the original, or gain access to resources as that program. For example, the ZeroAccess rootkit successfully ran one of its privilege escalation exploits by creating a copy of the legitimate tool Adobe Flash Player and then using the memory space allocated to the duplicate program to display a malicious UAC pop-up requesting Administrator credentials [90].

  In addition to trojanizing binaries, rootkits can also hide code or data in Alternate Data Streams (ADS), a secondary location within files that can contain data not viewable to the user when the file is accessed using its normal application [83].

  Note that injection/trojanizing could be considered another form of identity forgery (discussed above), but is discussed separately in this thesis, as detection methods applied to identity forgery of standalone rootkit code differs significantly from that of rootkit code trojanized within another application.

Outside File System:

To completely evade concerns from file system enumeration, Rootkits can store code and data outside the file system and on the hard disk. Examples of nonstandard locations include file slack space, which runs the risk of being overwritten, and in bad disk sectors [83].

The difficulty in placing code in locations identified as unallocated by the OS is that the rootkit must 1) be able to identify unallocated memory locations of an appropriate size, 2) ensure that this memory location is not pageable, and 3) store the address for this nonstandard memory location so that it can call this code when needed [73].

4.1.8.2 Process Evidence

Simply hiding physical evidence is not enough to prevent rootkits from being detected. A successful rootkit will also take measures to remove all traces of running processes and suspicious activity occurring in real time on a compromised system. Some of the methods used to

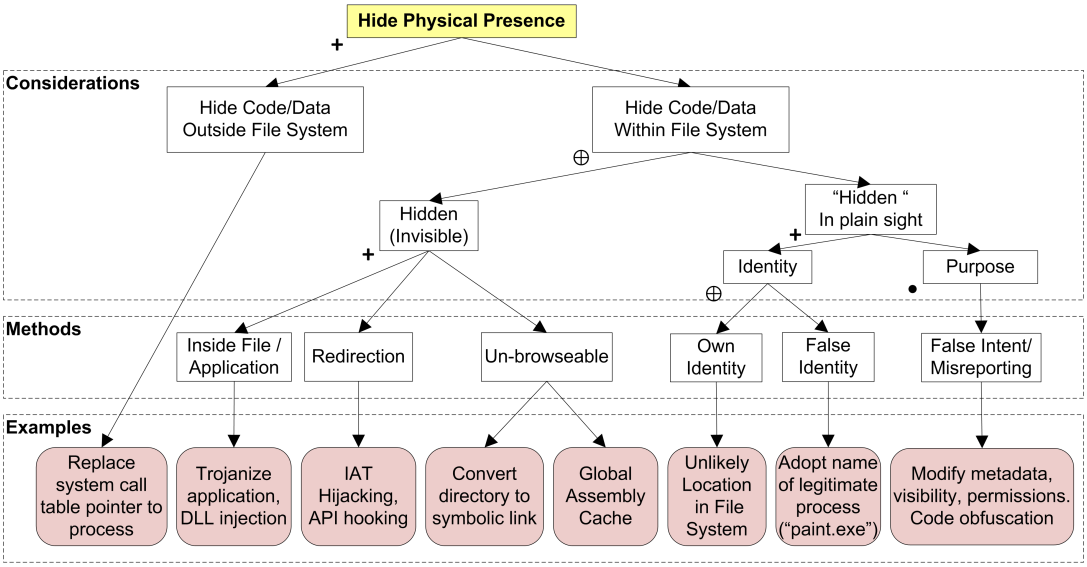

Figure 4.5: Concealment Methods for Physical Existence

hide physical evidence can also be used to hide running processes such as API enumeration interception, but additional methods are needed to hide dynamic rootkit activity.

The most obvious evidence of dynamic rootkit activity can be seen in running process lists, or in the type of data or amount of resources consumed by trojanized applications, therefore these are the locations rootkits attempt to remove evidence. It should be noted that a rootkit cannot effectively remove all evidence of its activity as the very methods used to induce invisibility are also additional activities prone to detection.

Similar to physical evidence, rootkit processes can either be visible ("plain sight") and forge their identity, and falsify data and status information about trojanized processes, or they can attain invisibility through removal from process list enumeration and other tricks. Process hiding can be summed up to the two following strategies: 1) falsify information about the process responsible for the malicious behavior to make it seem non-malicious, or 2) make malicious processes invisible.

The following are ways in which a rootkit may hide live data and processes.

- **"Plain Sight": Identity/Intent Falsification**

  Attempts at invisibility are more critical for processes than files, as there are hundreds of thousands of files, but only hundreds of processes which is much easier for a human operator to query at a glance [83]. Because of this, it is effective for a rootkit process to adopt the name of a benign or helpful process instead.

- **"Plain Sight": Intent Falsification/Misreporting**

  Malicious processes can falsify the intentions of a running process through obfuscation attacks, including control flow evasion and data falsification such as liveness misreporting.

  Some of the most common obfuscation attacks include control flow obfuscation, run-time loading of OS kernel modules, and the use of encrypted commands, callees, and strings [51].

  Control flow obfuscation is a method of obscuring the instructions executed by a malware module by inserting false code and unnecessary API calls. These false instructions attempt to foil flow analysis detection methods, which detect malicious activity based on data outputs communicated during live activity [69].

  Control flow obfuscation is difficult to do in kernel space, however, because driver structures tend to be modular, with a known signature, providing minimal options for obfuscation [51]. However, because kernel control flow is dynamic, the very fact that control has deviated from the original set of instructions is very difficult to detect [62].

  Compromised kernel objects can also be programmed to report false data to any monitoring services. This was famously done by Stuxnet, which sent malicious instructions to the PLCs to spin centrifuges at unacceptable limits, but reported normal parameters and results back to the console [91].

  One very useful type of data misreporting is that of an application or kernel object misreporting their liveness status. Misreported liveness can allow malicious kernel processes to tell process list APIs, and malware scanners, as well as other services that the module is *not* running, when in fact it is [62]. Liveness reporting can also be abused in the reverse method where services that the malware has disabled, such as anti-virus and IDS programs, report that they are active to the OS.

- **Hidden: Forcibly Unenumerated**

  In order to hide evidence of a standalone malicious process rootkits ensure that entries about their processes are not reported or displayed in the system call list [82], Windows Task Manager and other process list utilities. [83]

  Similar to API hijacking which prevents static resources from being enumerated, particularly within the file system and active processes along with dynamically loaded modules

can also be hidden from process enumeration APIs [83]. For example, the Aphex rootkit modifies the IAT entry for the `NtDll!NtQuerySystemInformation` API to intercept and filter process list queries. [83]

Rootkits "Hacker Defender" and "Berbew" place a JMP instruction in the `NtDll!NtQuerySystemInforma` in-memory code in order to hijack the enumeration queries. ProBot SE modifies the process dispatch entries in the Service Dispatch Table in the kernel [83].

In Windows, there are two process lists: the active process list and the process list used by the scheduler. The rootkit only needs to be recognized by the scheduler. The active process list is the only list queried for enumeration. Therefore, a rootkit removing itself from the active process list effectively hides evidence while allowing it to run as scheduled [73].

Windows also uses doubly linked lists for loaded modules. If a malicious loaded module modifies the pointers of adjacent modules, it can remove itself from the list, while still running [73]. The FU rootkit hides processes by removing the entry directly from the Active Process List kernel data structure [83].

- **Hidden: Disable Security**

  Another method rootkits can employ in order to remain undetected is to completely disable all services that are capable of detecting the rootkit in the first place. For example, the ZeroAccess rootkit hides itself from security services by disabling Windows Updates, Windows Firewall, Windows Defender, and other services. This is further discussed in Section 6.1.

  It is worth noting, however, that unless liveness statuses are meticulous about reporting that these services are live, evidence of a completely disabled suite of security software can also be a very strong indicator that malicious activity is occurring within the system.

- **Hidden: Injected**

  Trojanizing a popular utility is a way for a rootkit process to remain undetected, serving as as additional malicious functionality in conjunction with a legitimate service. For example, a SANS investigation of a compromised web server revealed the `mYrk` rootkit disguised inside a legitimate Linux utility called `zic` [45]. DLL injection, particularly in commonly accessed libraries, is also an effective way for existing kernel processes to sponsor rootkit functionality.

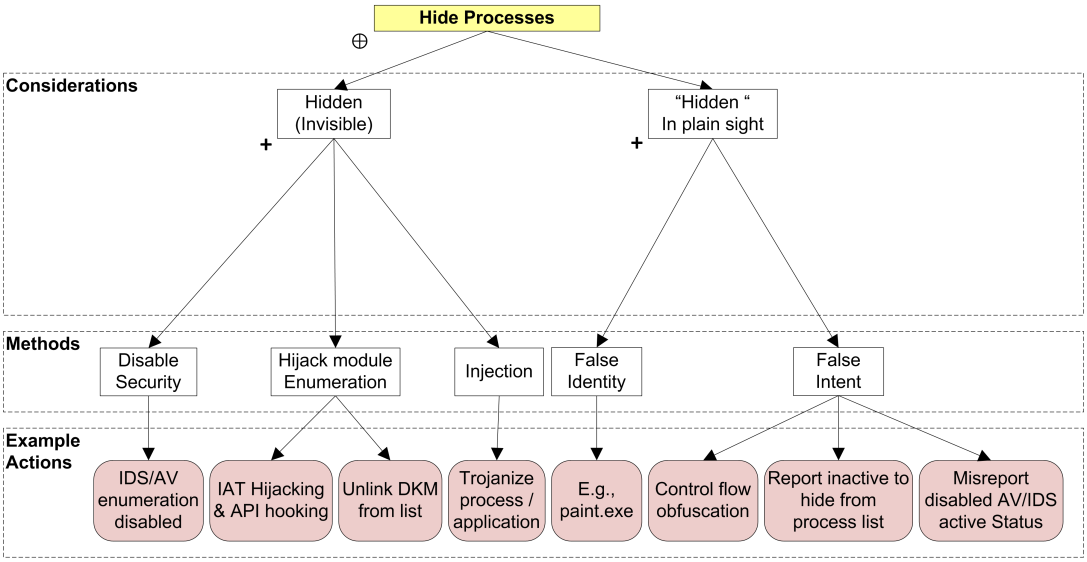

Figure 4.6: Concealment Methods for Process Evidence

Frequently called kernel modules can be injected with malicious code [9]. Rootkits can also bring their own compromised versions of system utilities. For example, Bravo *et al.* provide the example of a custom version of `ps` that omits the presence of malicious processes [9].

4.2 Backdoor Creation

The backdoor is a channel opened by the rootkit specifically for malicious communications with sources outside the victim's machine. This channel is used to transmit information about the compromised machine, download updates or additional malware, and is sometimes used to connect to other compromised machines as part of a botnet.

Section 4.2.1 discusses methods that rootkits may use to establish a covert communication channel. Section 4.2.2 applies general rootkit objectives to the C2 channel and how C2 activities hide and maintain persistence. Section 4.2.3 discusses the types of communication and services to which these channels communicate, and information transmitted and activities performed through the backdoor channel.

Table 4.7 presents a roadmap to topics discussed in this section.

4.2.1 Backdoor Implementations

Backdoors are classified into three different types given their placement method: 1) vulnerabilities planted for an attacker to manually exploit to gain access, 2) software planted on the victim's machine which facilitates communication or direct access, or 3) backdoors that are

Kill Chain Phase	Topic	Section
Backdoor Creation		4.2
Backdoor Implementation		4.2.1
	Vulnerability Creation	4.2.1.1
	Remote Administration Software	4.2.1.2
	Custom Backdoors	4.2.1.3
C2 Objectives		4.2.2
	C2 Persistence	4.2.2.1
	C2 Hiding	4.2.2.2
C2 Communication		4.2.3
	Communication Target and Services	4.2.3.1
	Communication Methods	4.2.3.2
	Communication Content	4.2.3.3

Table 4.7: Section 4.2 Roadmap

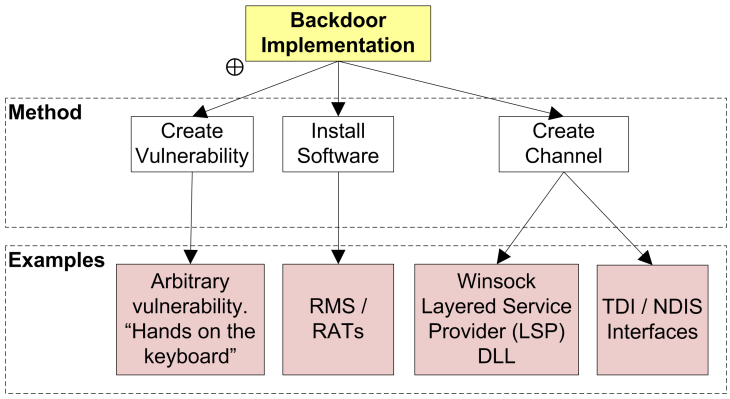

Figure 4.7: Backdoor Implementation Methods

manually created by the rootkit. These backdoors can communicate a variety of information and instructions between the target system and servers or actors offsite.

Figure 4.7 depicts the methods used by rootkits to establish a backdoor into a victim's machine.

4.2.1.1 Vulnerability Creation

A rootkit could modify the host system to create a vulnerability that would allow a human attacker to manually use knowledge of that vulnerability to penetrate the system [52].

4.2.1.2 Remote Administration Software

A malicious actor can use a rootkit to implement a software tool such as a Remote Administration Tool (RAT) or Remote Manipulation System (RMS). These services allow an attacker

full access to all services on the victim machine whenever desired. Two popular RATs used by the APT1 threat are Gh0st RAT and Poison Ivy [42] [21].

SecureList reported a bank hack in which a modified version of an RMS was used to transfer funds. The initial infecting malware, `Backdoor.Win32.RMS`, initiated the download of `Backdoor.Win32.Agent`, which provided virtual network services (VNS) as remote access to the computer [59].

While RAT/RMS services may provide an entire suite of options for activities which can be performed on a target, they require targeted human interaction and are therefore easier to detect than a custom backdoor [21].

4.2.1.3 Custom Backdoors

Rootkits can also choose to manually install a communication service by choosing a protocol and communication channel in the same way any other application or service. This is not difficult, as a malware process can easily open ports simply by hooking certain functions within the kernel [39].

Rootkit installers can also come with custom payloads containing backdoors in the form of executables or code, which are planted and executed as part of the rootkit installation process. For example, a malicious PDF sent to the Lockheed Martin Computer Incident Response Team (LM–CIRT) contained an installer `fssm32.exe` with two backdoor components, `IEUpd.exe` and `IEXPLORE.hlp`. These components communicated to the C2 server using HTTP requests [29]. APT1 actors have been known to use software tools such as Poison Ivy and Gh0st RAT but more often create their own custom backdoors on systems they infiltrate [42].

The APT1 threat has developed several backdoors throughout its cyber campaigns. Mandiant classifies the APT1 custom backdoors into two categories: "beachhead backdoors" and "standard backdoors". Beachhead backdoors are a minimal-service type of backdoor which allows attackers to retrieve files, gather system information, or launch a standard backdoor. The beachhead backdoor is a "WEBC2" backdoor, which downloads a web page from a C2 server, and then interprets any data written between special HTML tags as commands (this is how the APT1 group earned the nickname "Comment Crew") [42].

APT1's standard backdoors communicated both via HTTP (to hide C2 traffic among normal web traffic) and via a custom protocol. This backdoor allowed APT1 actors to almost completely control victim systems by gaining full access and permissions to files, directories, processes, the

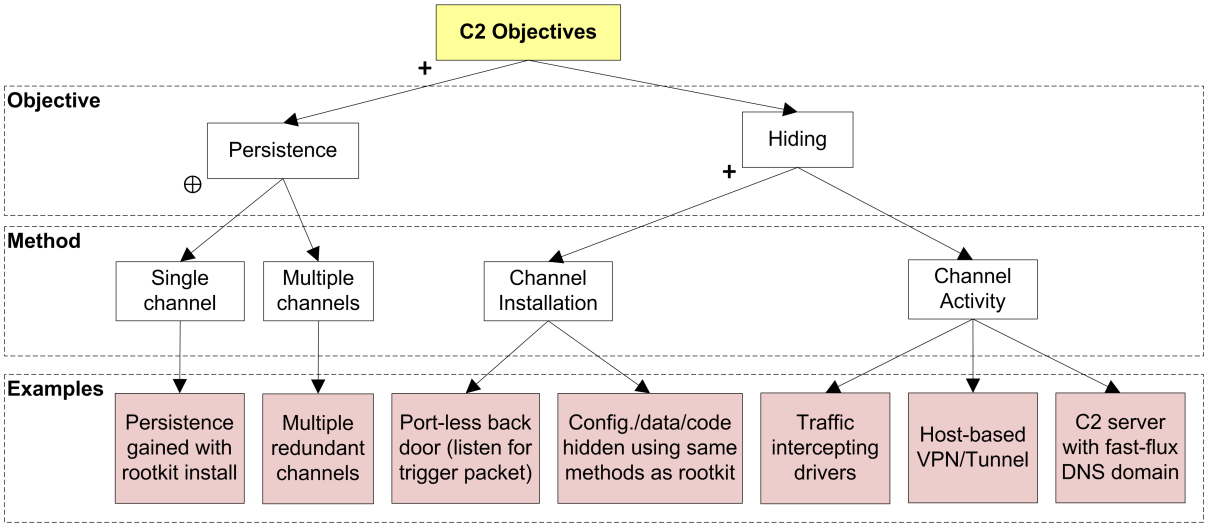

Figure 4.8: C2 Objectives: Persistence and Hiding

registry, keylogging and capturing mouse movement, and even logging off the current user or shutting down the system. One example of a standard backdoor developed by APT1 is the "BISCUIT" backdoor, named for the command `bdkzt`, which is believed to have been in use since at least 2007 [42].

Custom backdoors are used both to communicate autonomously with C2 servers and to be controlled manually by a human actor. However, backdoors which communicate autonomously with C2 servers are the most common. According to the Verizon 2014 Data Breach Report, backdoors as vulnerabilities only accounted for 4% of offsite malicious communication, but 86% of that traffic consisted of C2 server communication [78].

4.2.2 C2 Objectives

Once a rootkit is established in the target system it has the ability to immediately open a channel to the C2 server. However, without taking precautions to hide this new modification, the vulnerability is open to detection and immediate removal. Actions must be taken to both hide and establish persistence of this channel before use. Even a reboot of the machine could destroy the channel [13].

Figure 4.8 depicts the methods used by rootkits to fulfill basic channel supporting objectives.

4.2.2.1 C2 Persistence

Persistence of a C2 channel is not often distinguished from that of the rootkit itself, as installation of the trojan or backdoor is considered to be the primary persistence technique for

APTs [29]. In the same way, the same mechanisms which rootkits use to persist also apply to settings or configurations used in order to create the C2 channel, including storing files and configurations in non-volatile locations.

Redundancy, however, is commonly used to keep C2 channels open. Attackers often create redundant channels in the case that security configurations render the existing active channel inoperable [23]. Redundancy can also be implemented with newly developed backdoors installed on multiple systems within a compromised domain [42].

4.2.2.2 C2 Hiding

The same methods used to hide existence and processes of a rootkit are applied to conceal the existence and communication of a C2 channel. In particular, C2 channels aim to hide both the channel and the activity occurring over that channel. While there are several methods of concealing or misleading the contents of network traffic, the only methods discussed in this thesis are directly related to rootkit communications, as established between a rootkit and its C2 server.

The code and modifications required to establish a rootkit channel can be hidden using the same methods as the rest of the rootkit. For example, the `Backdoor.Win32.RMS` malware used for bank hacking hid RMS configurations in the Windows Registry. The registry key containing the RMS configuration was modified from `HKLM\SYSTEM\Remote Manipulator System\v4` to `HKLM\SYSTEM\System\System\Remote\Windows` [59].

The user space Winsock Layered Service Provider (LSP) DLL is easy to use, but highly visible. In order to secure a C2 channel covertly, rootkits will often use the kernel space "TDI" and "NDIS" interfaces. The TDI interface operates at a lower level than user space security monitors, and the NDIS driver allows a process access to raw packets, which can bypass security services including firewalls [51].

A portless backdoor can also be used so that there is no evidence of any C2 activity until a trigger packet "wakes up" the backdoor service [45]. In order to both hide and establish persistence for C2 services the actors behind the Duqu 2.0 APT deployed special drivers on any internet–facing firewalls, gateways, and other services. These drivers appear as a normal system service which avoids creating log records and consistently allows the attackers direct access to the internal network. For example, the `portserv.sys` driver listens to network traffic and waits for packets containing a special keyword, such as "romanian.antihacker". It then forwards any

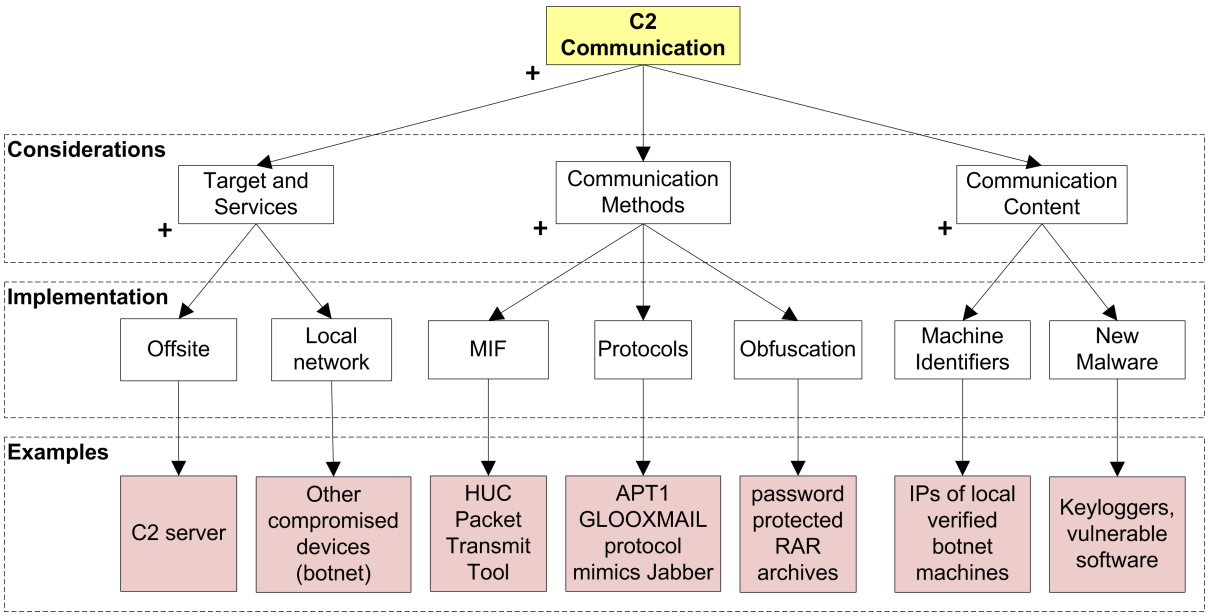

Figure 4.9: C2 Communication Considerations

traffic from that IP to either port 445 (SMB) or 3389 (Remote Desktop), effectively tunneling the attacker's traffic [34].

The C2 server can also take measures to conceal its identity. Fast–flux DNS domains are a common method used by C2 services to hide their infrastructure and identity. Additionally, the communications themselves can be encrypted, which when used in combination with a fast–flux domain C2 server make it difficult to determine what types of communication are occurring through the rootkit's channel. For example, the Duqu APT used steganography to attach data to JPEG images prior to transmitting them to the C2 server [79].

4.2.3 C2 Communication

This section discusses the types of services, methods, and types of information transmitted via a C2 channel.

Figure 4.9 depicts communication considerations and implementations a rootkit may use.

4.2.3.1 Communication Target and Services

The most common service with which rootkits communicate is to an offsite server dedicated to controlling compromised systems [85]. In 2013, Mandiant reported that in the previous two years the APT1 group established at least 937 C2 servers hosted on 849 distinct IP addresses around the world. From January 2011 to January 2013 Mandiant also detected APT1 actors from 832 distinct IP addresses logging into this C2 infrastructure using Remote Desktop [42].

C2 communications need not necessarily communicate with only offsite servers, however. Once a channel has been established it can be used for lateral communication with other compromised machines in the same network or with other machines as part of a botnet.

For example, the ZeroAccess botnet operated as a peer-to-peer botnet communicating with a C2 server while maintaining a list of 256 other actively infected IP addresses. The bot kept the C2 server updated on other compromised hosts' infection status, but also distributed malware and instructions to other bots [90].

4.2.3.2 Communication Methods

Malware infection frameworks (MIF) can be used to facilitate communications between the C2 and compromised machine. MIFs control the C2 server, send commands, and interpret responses from the victim. Botnets SpyEye and Zeus used MIFs in order to target victims for banking information exfiltration [72]. APT1 actors used the HUC Packet Transmit Tool (HTRAN) to communicate between C2 servers and victims, as part of their attack infrastructure [42].

Protocols used to transmit the C2 channel can be just as important as disguising the messages themselves. C2 channels commonly disguise their malicious communications as legitimate web traffic by using the HTTP protocol, but other communication protocols may be used as well. Some APT1 backdoors which mimic web traffic include: `MACROMAIL` mimics MSN Messenger, `GLOOXMAIL` mimics Jabber and XMPP, and `CALENDAR` mimics Gmail Calendar's traffic. The APT1 threat is known to use FTP to offload data as well [42]. Hiding among legitimate web traffic to log into malicious web portals is a common way for the rootkit to communicate with the attackers and download malware without raising suspicion [42].

Obfuscation of the communication carried out by the C2 channel is often in the form of encryption and is used to mask communications between C2 servers and victim systems. APT1 attacks are known to frequently use SSL encryption, although this method of illicit communications can be easily detected once the certificates used for SSL become known [42]. Information being exported can also be broken into smaller file sizes or hidden inside innocuous traffic. APT1 often preferred to pack information into password protected RAR archives, often using batch scripts to automate the process, and then splitting the files into 200MB portions before uploading the data [42].

4.2.3.3 Communication Content

Although a rootkit's purpose is to enable attackers to control a machine, access other machines and services, and eventually use the machine's resources and information to further the attacker's goals, these specific missions are often carried out separate from the rootkit itself. For example, if a malicious actor's goal is political or industrial espionage the method of discovering relevant documents or capturing keystrokes is performed by a special malware module or performed manually by the attacker. The rootkit is often limited to only communicating basic information that helps enable the attacker to further the mission.

Rootkit communications fall into two categories: 1) information about the victim machine and 2) downloading malware modules to fulfill the attacker's mission in ways that the rootkit alone is not programmed to do.

Machine identifiers are commonly created and submitted to the C2 server or other compromised machines. As referenced in Sections 4.2.3.1 and 6.1.5, each victim maintained a list of 256 known active infected machines' IP addresses which were used for communication to other bots and to report this list to the C2 server. The bot also calculated its own identifier using a Domain Generation Algorithm (DGA), with which it authenticated its communication to the C2 server [90]. These identifiers can also be used so that the malware will only execute on the tagged machine, and cannot be easily studied on any other device. Additionally, malware tagged with a specific machine identifier makes the malware signature unique which can foil signature based detection systems [89] .

Malware modules help further the attacker's mission by performing actions that the rootkit alone is not designed to do. Rootkits are often kept small with limited capabilities, but the rootkit backdoor opens up a window do download and install almost any kind of malware the attacker deems useful [72] [85].

4.3 Rootkit Actions

After a rootkit has been installed on a victim's machine, and has established connection to a C2 server, there are still several activities it can perform. A rootkit rarely acts alone and although the rootkit does not provide these services itself it still helps facilitate the further spread of other malware and provides hiding and persistence support for APT attacks.

Additional rootkit activities which are related to a greater APT mission include lowering further defenses, disguising network traffic, keylogging and capturing user data, continuing to

Kill Chain Phase	Topic	Section
Additional rootkit activities		4.3
	Disable Security Services	4.3.1
	Malware Dissemination	4.3.2
	Information Capture	4.3.3

Table 4.8: Section 4.3 Roadmap

intercept and disable security services, and exfiltrating information [82] [51] [8]. Although the C2 channel could be considered an additional activity, because most kill chain models treat backdoor channel activities as a separate stage, it is presented separately, in Section 4.2.

Table 4.8 presents a roadmap to topics discussed in this section.

It should be noted that although the topics in this section address some of the most common APT support activities performed by rootkits the the list of items addressed here is by no means complete. With a complete kernel–level installation and a secure backdoor channel rootkits could potentially initiate and support nearly any type of malicious activity.

4.3.1 Disable Security Services

Kernel–level rootkits have the ability to be highly conscious of their environment, with access to all information and privileges for monitoring and manipulating other applications on the host OS. Some rootkits can even detect if they are on an emulated system, and if so, may refuse to exhibit malicious behavior to avoid analysis [51].

With high environmental awareness and system level privileges rootkits often manipulate or disable security services. This includes firewalls, anti–virus, IDS software, and other services of which the rootkit is aware may be able to detect its activities [4] [82]. The GMER rootkit detection tool, referenced in Section 1.1.3, provides a randomized name for each new implementation because many kinds of malware actively seek out a GMER installation with the intention of disabling it. After the Zeus (or "Zbot") malware source code was published variants of Zeus have been modified to evade anti–virus software, in particular signature–based detection methods.

The disabling of security services may reveal the presence of the rootkit but if a rootkit manages to hook into these services and provide false data, then the risk is much higher, as the system is falsely reporting the absence of malicious activity. Security software that hook into the SSDT for access to system resources can in turn be hooked by a rootkit. These hooks can

cause the security software to conditionally report false information back to the system [63]. Operating system APIs in general tend to be shared by both security services and rootkits alike, and so in the presence of a rootkit, these services cannot be completely trusted [73].

4.3.2 Malware Dissemination

Occasionally a rootkit performs tasks beyond that of installation and opening up a C2 channel. The `mYrk` rootkit as detailed by SANS was "capable of hiding processes, files, network sniffing, modifying firewall rule-sets, and key logging" [45].

However, as addressed in Section 4.2.3.3, rootkits may not contain all the malicious functionality that the attacker desires. In order to further an APT campaign the the rootkit can instead use its C2 channel to download malware from the C2 servers and launch it on the compromised system [85].

For example, the GhostNet cyber attack spread as a trojan attached to a targeted email. After installation, the GhostNet trojan then downloaded the Ghost Remote Administration Toolkit to the victim's machine so the machine could be controlled remotely [72] [37].

In addition to downloading and installing new malicious services a rootkit can also extend its own protections to other malware as well. Rootkits are often used to mask the presence of files and processes of other malware modules [51].

4.3.3 Information Capture

A substantial black market exists for personal information, banking credentials, and user account information, such as names, social security numbers, credit card numbers, passwords, and personal information. This information can be used for political or industrial espionage, or can be sold on the black market for financial gain [72]. Stolen credentials and password hashes can also be used to facilitate lateral movement through a victim's network [4].

While it is possible for victims to unintentionally download spyware programs using any of the social engineering delivery methods as discussed in Section 3.4 and even though rootkits could download and install these same programs through the use of the C2 server, as discussed in Section 4.3.2, kernel–mode rootkits have a natural advantage to gather information from a compromised machine. Information can be collected through RAM scraping utilities, which search memory for patterns that are likely to match credit card numbers, social security numbers, and other information that matches predictable patterns [78]. However, for information

that is not found through searching, a rootkit can hook keyboard input functions, also known as "keylogging" to glean information [39] [4].

In Windows, the I/O Manager is responsible to provide device I/O services and device driver communication. Manipulating the I/O Manager or by hooking the IDT, kernel mode rootkits can intercept and issue commands to hardware and copy information from every keystroke [73]. Keylogging to gather banking and personal information is often implemented in botnets which can gather massive quantities of information that can later be used or sold [72].

According to the Verizon 2014 Data Breach Report, although keyloggers have slipped in popularity compared to RAM scraping, they are the third most common threat in crimeware and in the top ten threat varieties of cyber espionage [78].

4.4 Specialized Rootkits

This section discusses two types of rootkits which go beyond those that just target casual operating systems: rootkits which are not limited to kernel space alone and rootkits which specifically target cyber–physical systems.

4.4.1 Advanced Rootkits

The rootkits discussed in this thesis are primarily operating system dependent, and as such can be detected by changes within the underlying operating system and kernel. However, other types of rootkits are possible, including those that are operating system independent. Although some of these have not been seen in the wild all of the rootkits addressed in the following section have at a minimum been proven possible as proof-of-concept implementations through research.

These rootkits may install themselves in locations outside the operating system, including in hardware and virtualization platforms, and interact directly with hardware, bypassing the operating system altogether. Although these rootkits are not addressed in depth here, future research may develop a taxonomic approach to understanding the scope of operations of these rootkits, in order that specialized detection methods for these rootkits may also be evaluated.

Some of the different types of rootkits include:

- *Virtualization rootkits* which affect virtualized hardware platforms and processor virtualization extensions [77] [73].

- *Memory–based rootkits*, which only run in memory, and do not survive reboot, but can be persistently reloaded by other active malware modules [77].

- *PCI rootkits*, which hide in the firmware of PCI cards [73] [9].
- *SMM rootkits*, which launch by abusing BIOS shadowing to install a keylogger in System Management Mode (SMM) using a System Management Interrupt (SMI) [86] [73] [9].
- *Bootkits*, or *MBR rootkits*, which copy themselves into the Master Boot Record. Bootkits add code to execute at machine boot time, before the operating system has loaded, and so are outside the scope of this report, which deals with covert malware at the kernel–mode. This also includes BIOS bootkits [22] [9].
- *GPU–based rootkits*, such as the Jellyfish rootkit, which can run without standard kernel hooks or modifications [25].
- *ICS/SCADA rootkits*, as addressed in Section 4.4.2.

Of particular concern are rootkits which use normal kernel–mode exploits to compromise industrial control systems and cyber–physical infrastructures. These rootkits are discussed in further detail in Section 4.4.2.

4.4.2 SCADA/ICS Rootkits

Industrial Control Systems (ICS), also known as Supervisory Control and Data Acquisition (SCADA) systems, consist of software administering the actions of a physical system that provides physical services. For example, SCADA systems automate the machinery in factories, electric power dams, water treatment facilities, and other public and private services [91]. The interdependence of physical systems and cyber systems creates "cyber-physical systems" and an only recently-exploited terrain for cyber threats [70].

These systems have been traditionally considered "airgapped" because initially they were not connected to the internet. Additionally, these systems initially were constructed with proprietary equipment and in-house communication protocols. Hence cyber threats in the form of malware could not be easily mass-produced.

However, to enable remote monitoring and maintenance of ICS, machines which host the human-machine interfaces (HMI) - the software that regulates machine actions - are increasingly being connected to the internet. Industrial equipment is starting to include Wi-Fi and Bluetooth capabilities in order to be maintained by mobile HMI devices and communicate with other machinery. Equipment and protocols used in control systems are also becoming standardized [91].

Internet-connected ICS systems can be easy to find through customized string searches in popular search engines such as Google and Bing, through the "internet of things" search engine Shodan [68], through data mining such as the Internet Census 2012, or through SNMP searches. To easily facilitate attacks on thse devices, Metasploit modules started appearing as of 2012 [10].

Cyber-physical ICS/SCADA systems are highly vulnerable as a single error can have a great impact in the physical world. For example a misreported data value or a small memory leak can cause the entire system to respond abnormally and even cause the system to fail [91]. Cyber threats for ICS/SCADA systems can target the OS on which the HMI is hosted, the HMI itself, the programmable logic controllers (PLC) which send instructions to hardware and report values back to the HMI, as well as theoretically any other node in the system.

Other characteristics of ICS/SCADA systems include:

- Control systems are time-critical/hard real-time systems where a failure to meet a single deadline is the same as failure of the entire system.
- Edge nodes (physical devices) are just as critical to the success of the control system as the HMI, databases, server, and any other node (even though only the perimeter is likely to be protected).
- Misinformation of data can cause a device or human operator's *appropriate response* to result in the system responding destructively. Misreporting can hide incidents or alerts altogether.
- Some ICS-specific applications, (e.g. the VxWork embedded OS) can be implemented as a monolithic kernel, through which all applications are performed as kernel tasks, run with highest privileges, and low memory protection [91].

Malware that exploits these characteristics has already been seen in the wild. In 2010, The United States Industrial Control Systems Cyber Emergency Response Team (ICS-CERT) reported a trojan spread via USB which targeted Siemens ICS software including the HMI SIMATIC® WinCC and PCL programming and configuration modules SIMATIC® STEP 7 [30].

This malware became famously known as Stuxnet, the first major ICS-targeting malware. Stuxnet used standard windows rootkits to hide itself on the HMI (hosted on a Windows machine), but also implemented rootkit functionality in the targeted PLCs. The library file

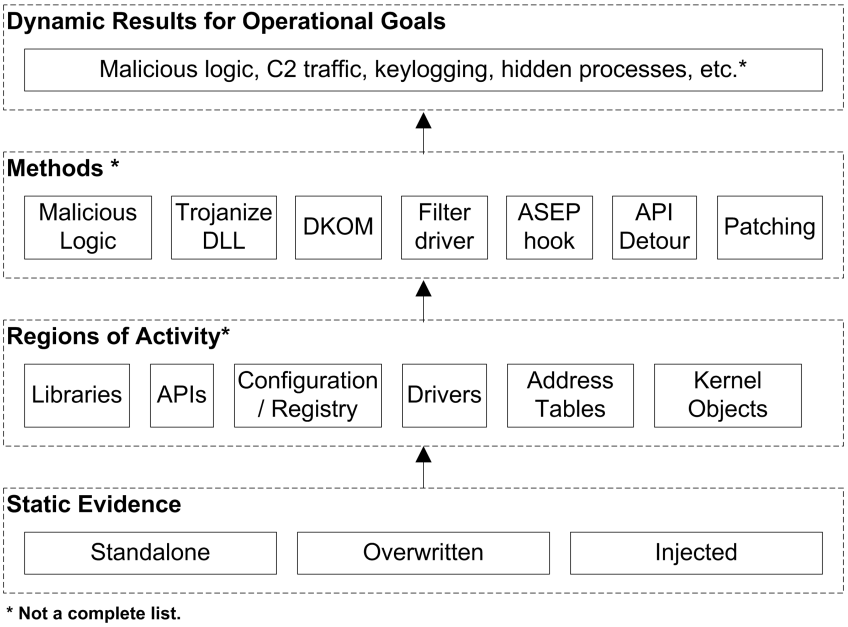

Figure 4.10: Layers of Rootkit Installation Indicators

`(s7otbxdx.dll)` which the infected HMI machine would use to communicate with the PLC, was replaced with Stuxnet's modified version `(s7otbxsx.dll)`, which contained mostly the same exports as the original, but with additional functionality. Stuxnet copied its code to the attached PLC which contained 70 encrypted code blocks that modified or added PLC functionality. These PLC function blocks were hidden from the HMI by hijacking any request used by the HMI to view those code blocks [20] [19].

In 2014, a variant of BlackEnergy, a "highly modular" malware, spread to ICS systems running GE's Cimplicity HMI, Siemens WinCC HMI, or the Advantech/BroadWin WebAccess software. These systems were all connected directly to the internet and they were suspected to have been discovered through the use of automated tools [31].

In 2014, the Air Force Institute of Technology (AFIT) created a proof-of-concept PLC rootkit, which could be loaded via any known vector such as a USB or the supply chain. This rootkit was "vendor agnostic", persisted through the bootloader, and could be triggered through a time bomb in the firmware or remotely [26] [17].

4.5 Summary of Rootkit Activities

Rootkits use three basic methods to install on a host OS: injection, overwriting, and placement of standalone code. These three methods are used on kernel structures, such as modules,

APIs, address tables, and other OS–specific resources and regions in order to generate malicious logic to carry out their mission. Some of the methods used include trojanizing library functions, replacing pointers to legitimate APIs with newly–planted malicious ones, and hooking enumeration functions to hide both rootkit code and running processes.

These malicious activities help the rootkit fulfill its primary functions of hiding itself and its resources, persisting where it cannot be easily removed, and setting up a communications channel in order to offload information and spread additional malware.

Figure 4.10 displays the different layers at which rootkit activities can operate. *Static evidence* are the basic components of any rootkit activity, and combined with each other in different *regions of activity* results in the *methods* used to achieve *operational goals*. Each of these categories of rootkit activity can be detected by some combination of rootkit detection methods, as addressed in Chapter 5.

Chapter 5
Rootkit Detection

In the same way in which rootkits employ diverse strategies for installation detection methods also differ in variety and in execution. This chapter discusses detection methods applied to indicators of a rootkit's installation and activities discussed in Chapter 4.

Section 5.1 presents a high-level overview of the concepts behind rootkit detection techniques. Subsequent sections provide further detail on each of these high-level concepts and present specific techniques. Finally, Section 5.5 discusses how rootkit activity relates directly to detection methods and provides a framework for detection of each level of rootkit activity. Table 5.1 provides a simple roadmap for topics discussed in this chapter.

Topic	Sub-topic	Section
Rootkit Detection Methods Overview		5.1
	Detection Technique Categorization	5.1.1
	Detection Metrics and Considerations	5.1.2
	Detection Method Constraints	5.1.3
Static Rootkit Detection Techniques		5.2
	Signatures	5.2.1
	Static Heuristics	5.2.2
	Static Memory Forensics and Mapping	5.2.3
Dynamic Rootkit Detection Techniques		5.3
	Dynamic Behavior Analysis	5.3.1
	Crossview Detection	5.3.2
	Dynamic Memory Forensics and Mapping	5.3.3
Detection Execution Platform		5.4
	Local Machine	5.4.1
	Virtualization	5.4.2
	Hardware	5.4.3
Rootkit Detection Methods Evaluation		5.5
	Coverage Evaluation	5.5.1
	Conclusions	5.5.2

Table 5.1: Chapter 5 Roadmap

5.1 Rootkit Detection Methods Overview

This section describes foundational concepts for rootkit detection methods. Section 5.1.1 provides a categorization of rootkit detection techniques. Section 5.1.2 addresses how data collected in the detection process are evaluated and determine malicious behavior. Section 5.1.3 provides some constraints applied to the rootkit detection methods described in this thesis.

5.1.1 Detection Technique Categorization

According to Lockheed Martin, there are three types of indicators of malicious activity: atomic, computed, and behavioral [29]. This thesis breaks down these indicators into two primary categories of rootkit detection: *static* and *dynamic* analysis, and two secondary approaches: *code*–based or *data*–based analysis. Additional types of analysis such as *memory forensics* are also commonly used in combination with other techniques. These methods can be used alone or in combination to detect evidence of rootkit activity or presence. This list is not comprehensive.

- **Static** analysis examines the atomic "physical" features of the OS and attempts to identify

the hard-coded presence of a rootkit. This includes static kernel objects, APIs, content residing both within and outside the file system. Static analysis ultimately attempts to uncover evidence of the rootkit installation itself: injection, overwriting, or the planting of standalone code.

Static analysis can be performed on a byte-for-byte snapshot of a system and does not require that the system be actively running. Behavioral characteristics can be detected through heuristic static analysis. Static analysis techniques to reveal rootkits are further discussed in Section 5.2.

- **Dynamic** analysis attempts to gather information about rootkit activity from a system currently in operation. Information collected from dynamic analysis includes loaded dynamic kernel modules, system calls placed in real-time, active applications and reported liveness statuses, and any changes in system activity as a result of data flow through the system. Dynamic analysis attempts to discover evidence that can only be obtained through an actively operational system and not through static analysis. Dynamic analysis techniques to reveal rootkits are further discussed in Section 5.3.

- **Code**–based analysis is closely related to static analysis and relates to the infrastructure of the OS and its components and applications. Code–based analysis is concerned with the data structures and instructions available for execution, but not the individual properties of these structures themselves.

  For example, code–based analysis may identify byte-strings of a known malware but is not concerned with the addresses to which it is calling are overwritten or out of bounds.

- **Data**–based analysis, or "semantic analysis", is concerned with the flow of data throughout the system. This includes both hard–coded data (static), and reported data values for run–time decision making (dynamic). "Data" can include information such as pointers and properties of data structures. Data-based detection methods can identify malicious inputs and outputs even where the sizes of data structures or sequence of instructions appear to be unchanged [62].

- **Memory forensics** is a method of mapping memory accesses, regions, and allocation and deallocation events. Understanding the memory landscape within an OS can reveal the placement of malicious code or nonstandard accesses to malicious objects or for malicious

logic. Memory forensics applies to both static and dynamic detection techniques.

The goal of rootkit detection is to reveal a rootkit's installation so that the roorkit may be ultimately erased and the system reverted to its pre–rootkit state. Without a physical foothold somewhere within the system, dynamic behaviors which carry out the rootkit mission would not be possible.

From this perspective, it can be argued that static detection is the most important type of rootkit detection as it reveals the rootkit installation itself. However, many hard–coded modifications cannot be determined by themselves to be malicious. Hence the behaviors caused by those modifications reveal the malicious properties of that static data. This is why dynamic detection methods are also necessary to uncover rootkits. According to Dube *et al.*, static and dynamic analysis complement one another and provide a "full spectrum defense against malware with reduced effective scan and detection time" [16].

Detection methods addressed throughout this chapter may use static analysis, dynamic analysis, or a combination of both. Some methods evaluate the entire operating system for evidence of a particular malicious characteristic, but it is often the case that both commercial and research techniques combine techniques in order to focus on limited regions of effectiveness.

5.1.2 Detection Metrics and Considerations

In order to determine that some observable configuration or behavior is malicious definitions of "normal" and "malicious" must be be determined. Comparing what is actually observed in a system to these definitions allows detection mechanisms to determine whether or not what is observed is malicious or allowable. This includes both behavioral (*dynamic*) and structural (*static*) deviations. These deviations are then evaluated by other pre–defined metrics such as regions affected, number of occurrences, and likelihood of the deviation being erroneous or malicious.

Methods of defining a baseline of acceptable behavior fall into two overlapping categories: heuristic rulesets, which often define signs of malicious behavior, and thresholds, which establish a range of values within which indicators are determined to be either malicious or benign.

5.1.2.1 Heuristics

Heuristic methods define a set of rules or features to which code is evaluated for compliance. Heuristics can be applied both statically and dynamically. Static heuristics are addressed in

Section 5.2.2, and dynamic heuristics are addressed in Section 5.3.1.1.

Heuristic rules can be used by detection tools to make precise judgements on indicators because the rules define what behavior is explicitly unallowed. For example, an anomaly detection device that monitors the integrity of the SSDT table would detect changes to addresses in that table and then compare those changes to any anticipated acceptable or unacceptable scenarios. These scenarios could include factors such as which service is authorizing the change, the range of addresses to which the overwrite is allowed to point, and other pre–defined parameters. If an address has changed, and the authorizing service is not on the list of acceptable services, then this change is flagged as suspicious.

Although a precise determination on the maliciousness of an activity is ideal, covering all possible rootkit installation scenarios using specific rulesets is not practical. This type of coverage fails on two counts: 1) it is not feasible to completely safeguard entire kernels because of their size and complexity [82], and 2) low–level monitoring rulesets would need to be adapted to the specifics of every system for which it is deployed.

Additionally, strict rules are not practical to detect rootkit installation vectors that mimic the behaviors of legitimate applications and services. For example, an anomaly detection rule could dictate that any new Windows Registry modifications generates an alert. This would result in more false positives than useful information as the contents of the registry are highly dynamic and more frequently used for legitimate purposes. A rules–based detection metric would not be able to discern whether or not a normal system call is legitimate or part of a sequence of instructions used to create malicious logic.

Heuristics can also be applied for gathering information about a system, so that a threshold–based analysis may be performed. For example: the number of calls a module makes to a particular system call can be evaluate to be acceptable or suspicious. Thresholds are discussed in Section 5.1.2.2.

5.1.2.2 Thresholds

Thresholds are a detection metric that sets boundaries for acceptable behavior in order to distinguish malicious activities from identical legitimate activities.

For example, a context-based detection framework developed by Giura *et al.* identifies specific events that are correlated via a ruleset, which is a heuristic method. These events are evaluated for metrics “confidence” and “risk” by predetermined thresholds. If risk and confi-

dence levels are evaluated to be out of bounds then they are considered to be in the "alarm zone" and an alert is raised. Thresholds used for this experiment are parameters specific to each environment. As new events appear in the system, and match the characteristics of other events related to APT actions, the confidence level that these activities are malicious is increased [23].

One of the greatest difficulties in both dynamic and static models is how to define appropriate thresholds. Classification accuracy requires appropriate parameters to be selected based on the type of kernel in the system being monitored [66]. Thresholds also often yield false positives, but if the boundaries are too lenient, malicious activity will go undetected. Because of the dynamic nature of the kernel, not all malicious behavior can be explicitly defined to be anomalous. For example, a detection system may need to know how many calls to a particular module is considered benign, or exactly how many instructions a particular command should execute before setting off an alarm.

Thresholds are an imprecise method of detecting malicious behavior because they are biased toward known behaviors that consistently act outside normal system operation. If a malicious actor manages to only infrequently perform activities that happen to be monitored by a threshold metric then there is a chance that the activity will only deviate slightly outside normal behavior and therefore, will not be identified as malicious. Staying within acceptable thresholds helps APT threats maintain their covert presence in a compromised system.

5.1.3 Detection Method Constraints

Although the detection methods addressed in this thesis attempt to provide as complete coverage as possible, the following constraints apply to detection methods discussed here.

- Detection methods are often not limited to specific rootkit activities, but on detecting that type of activity applied throughout the OS. For example, rather than detecting only SSDT hooks a technique may detect all overwritten pointers in system–defined address tables.

- Detection methods are evaluated independent of the mode from which the rootkit initially launches. Detection methods assume the scenario of some arbitrary rootkit which is capable of assuming full privileges at some stage in its implementation and is capable of any rootkit defined activity.

- Detection methods do not aim to detect a rootkit's objectives but rather focus on detecting

evidence of system modifications, which rootkits use to achieve their objectives.

- Detection methods do not aim to detect or identify specific (named) rootkits but rather to understand what information is available to indicate the existence of some arbitrary rootkit. Some detection tools which detect specific rootkits exist, such as TDSS Killer [33], but the methods which these tools use are part of a broader detection scenario.

- Detection methods do not aim to block a rootkit's process or perform rootkit removal. Some detection tools may perform these actions but these strategies are outside the scope of this thesis and are not discussed here.

- Detection methods are limited to discussing kernel/OS rootkits. Advanced OS–independent rootkits, such as BIOS and SMM rootkits, are outside the scope of this thesis and are not addressed.

This thesis addresses each detection technique, provides examples, and evaluates the scope of coverage and effectiveness for each in uncovering areas of rootkit activity, and in particular how that relates to the original installation.

5.2 Static Rootkit Detection Techniques

Static analysis searches through a system looking for hard–coded modifications to an OS which may indicate rootkit activity. This is usually performed by parsing code and data structures to extract a set of features to analyze for the likelihood of malicious intent [51].

The three main installation techniques used by kernel level rootkits are: injection, overwriting, and planting of standalone code and data. These techniques are all likely to be detected by static methods alone. Essentially, static analysis attempts to discover the physical modifications which enable rootkit behaviors.

Static analysis can be performed by looking for byte strings that match a known malware signature, heuristically analyzing code for potentially abnormal "behavioral" intent, or ad–hoc pattern matching, as is often the case in research settings. Rootkit indicators may be revealed by changes in code size, configuration, or presence of abnormal code, file formats, and data structures.

One of the main benefits of static analysis is that the rootkit or any related malicious binaries need not be active in order to perform static analysis [73] [51]. Static analysis is also

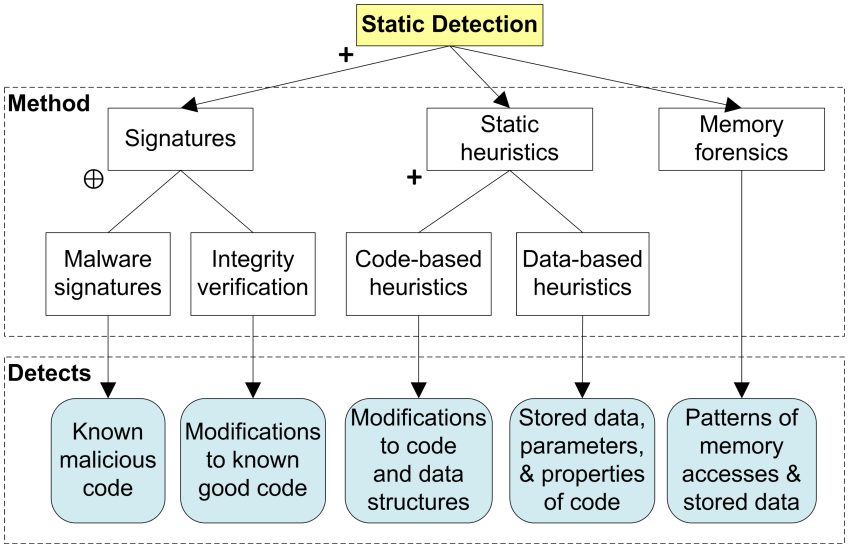

Figure 5.1: Static Rootkit Detection Taxonomy

advisable where possible because it is less resource intensive than dynamic behavioral analysis on a running system and can be performed directly on the host system [51].

There are two primary applications for static analysis: **signature matching**, which attempts to identify specific byte–strings, and **heuristic** methods, which applies rules to evaluate code properties for malicious intent, which cannot be ascertained by signature matching alone. Heuristic analysis in particular is aided by information gathered through *static memory forensics and mapping* [16].

Figure 5.1 demonstrates a high level association between the relationships between static and dynamic detection methods and what kind of rootkit indicators they target.

5.2.1 Signatures

Signatures consist of a unique string of bytes created by a hash or extracted from some defined byte pattern. These calculated values are then compared to known values to make a determination on the legitimacy or intent of the code. Signatures are used to detect known malware. Signatures can also be compared to ensure the integrity of known legitimate code and binaries

5.2.1.1 Malware Signatures

Malware signatures help host-based defenses find evidence of that malware in infected files and stop the vast majority of malware from entering a system, provided that virus and malware definitions are up to date.

Host-based IDSs and anti-virus software often contain an up-to date database of known

malicious strings. Applying and evaluating the signature can involve a database of regular expressions and a string matching engine. Each regular expression in the database can identify a known piece of malware or malicious functionality [66].

Traditionally this method is only applied to objects in a file system, but it can also be used to find a specific byte pattern in virtual memory modules as well [9]. Signatures can also be used as a basis to identify other similar malware that shares some of the same known malicious byte patterns [66].

This type of detection often attempts to stop malware before it is permitted to execute on a system, and is considered a preventative measure [9]. The main problem with signature-based methods is that the malware must be previously known. Unknown, polymorphic, or obfuscated malware defeats signature-based detection methods [51]. However, signature-based methods may also be used to detect malicious changes to previously known legitimate software. This method is known as **integrity verification** and is addressed in the next section.

5.2.1.2 Integrity Verification

Software integrity verification is a method of ensuring that changes to a piece of software (or similar self-contained object such as a static kernel module or DLL) can be shown to be untampered. Often this method is implemented by first creating a hash (a signature) of that object at initialization, and then consistently compare the original hash to a current hash of the file to verify that there are no differences. If an object's hash differs from the one stored at creation, then it is assumed that the object has been compromised.

The hash generated for comparison could be a checksum, SHA, MD5, or some other signature-creation algorithm. For example, Wang *et al.* built a hash by scanning the ELF headers for the code segment and computing an MD5 hash using Linux's `md5sum` utility [82]. Some software vendors can provide a certificate of integrity, or the administrator installing the software could verify the integrity of the object prior to installation, and then check it at intervals throughout the lifetime of that software. This method could fail, however, if the original download is compromised, as the original signature will vouch for a compromised application. Additionally, the signature will change with each update to the software, so updating integrity verification can be an ongoing maintenance task [82].

5.2.2 Static Heuristics

Static heuristics may identify modifications to code to which signatures do not easily apply. Heuristic rulesets can be applied to analyze nearly any aspect of a static image, particularly where signatures alone fail, and can evaluate the static "behavior", or intent, of code. For example, obfuscated or mutated code may not generate an exact signature match to a known binary, but because the malware retains the original functionality it will utilize the same opcode features as the original [51]. Similarly, the discovery of pointers to code outside the loaded kernel would detect address table hooks [73].

Static heuristic methods focus on non–runtime indicators and are often rule–based more so than threshold–based. These indicators include structural anomalies, program disassembly, and n-grams [16]. While the possibilities for the application of heuristics is vast there are several indicators that may reveal the presence of a rootkit. For example, a heuristic method may examine: the presence of obfuscation, pointers and the locations of addresses in memory, the number of instructions or jumps loaded to or from a kernel module, and the presence of nonstandard characters in code (e.g., "*") [51]. For example, a rule–based heuristic could determine that any JMP overwrite at the start of a function is definitively malicious based on the premise that most functions do not start with a jump instruction [73].

Heuristic methods can directly detect modifications to a system such as encrypted files and code, DKOM hooks and other hard–coded memory accesses to nonstandard locations. In specific detection scenarios, the use of nonstandard system APIs can be considered suspicious. For example, the presence of obfuscation of either code or intent can be a good indicator of malicious activity.

Musavi *et al.* demonstrated that static heuristics can be applied to detect kernel level filter drivers. Using a training set of 2200 legitimate drivers and then applying a custom set of static analysis tools on 2200 known malicious drivers common traits shared by many of the malicious drivers revealed metrics which can be detected by static analysis [51]:

- Injection was used by 27% of rootkit drivers but only by 1.4% of legitimate drivers.
- File modification activity was used by 19% of malicious drivers but only by 7% of legitimate drivers.
- Filters were used by 98% of malicious drivers and only by 19% of legitimate drivers.

- Write protection was bypassed by 7% of the malicious rootkit drivers but by only 3 out of the 2200 legitimate drivers.

One of the downsides to the detection method proposed by Musavi *et al.* is that attempts to distinguish rootkit drivers from legitimate drivers by analyzing use of the NDIS library functions to bypass security products. Interestingly, only 1.12% of rootkit drivers used the NDIS library, while 19% of legitimate drivers did. The authors propose that the complexities in handling the network protocol are not a great concern for rootkit drivers, and so code size would be a better metric for analysis in this case [51].

Musavi *et al.* also demonstrated how DKOM can be detected through a well–developed set of static heuristics by enumerating system calls used to obtain kernel objects, and the number of accesses to the offsets of those objects [51].

The Musavi static driver analysis experiment highlights the complicated role that thresholds play in the detection of rootkits. Without a defined threshold both legitimate and malicious drivers would be considered suspect, resulting in a high number of false positives. However, with low thresholds, both malicious and legitimate drivers would be considered safe.

The difficulty in employing heuristics is that in order to obtain confidence that a particular modification is related to malicious activity, it must be known beforehand exactly what is allowable for that specific system. Solutions are often limited in scope and are experimental. Additionally, heuristic–based methods detect unknown malware, but are "inefficient and inaccurate" and are unable to detect new forms of malware [66]. For example, according to Wang *et al.*, code hooks, which are hard-coded modifications to kernel text are "easily detected" but the possibilities of detecting which kernel objects are hooked are vast because theoretically any kernel instruction could be overwritten [84].

However, new implementations of malware still aim to achieve the same goals, and so they display the same intents and behaviors of traditional malware. Detecting malware and rootkits by their behaviors can be performed by dynamic analysis, which is discussed in Section 5.3.

5.2.2.1 Code–based Heuristics

Static code-based approaches look for evidence within rootkit data structures or code segments. While a code–based approach is very similar to heuristics in general, a code-based approach strictly looks for evidence of a rootkit's functionality through which data structures, system calls, or instructions are executed, regardless if they are new, overwritten, or injected code, or

called via malicious logic. Code–based methods are not concerned with "properties" of data structures. For example, the types of values passed between modules or the legitimacy or possible values of the data.

Code–based approaches are generally not applicable to dynamic detection strategies except in the case of dynamically loaded kernel modules.

5.2.2.2 Data–based Heuristics

While data–based detection approaches are more easily defined and applied in the arena of dynamic detection (See Section 5.3.3.1), data values stored for malicious use could theoretically exist for detection purposes. For example, a filter driver which returns a predefined set of values would be hard-coded to pass data values within a particular range. If these values are hard coded then they could potentially be detected using static heuristic methods [62].

5.2.3 Static Memory Forensics and Mapping

Kernel memory mapping analyzes memory utilized or accessed by a kernel module, which is the most thorough way to generate a view of the flow of information through the kernel. A well-formed map of memory accesses can reveal both out-of-bounds and nonstandard memory accesses and other evidence of malware infection [58] [73].

Memory forensics typically looks for two types of memory-related evidence: out-of-bounds memory accesses and nonstandard memory accesses. Out-of-bounds accesses are performed by examining locations of memory accesses,often in suspicion of pointer manipulation, compared with the results of bounds checking arrays and kernel object stacks. The discovery of an out-of-range pointer may reveal the presence of address table hooking or redirection attacks to malicious code [77] [69].

Nonstandard memory accesses typically look for jumps to new memory locations, even legitimate locations, and can discover attempts at malicious logic. This type of memory forensics requires a pre-established baseline of expected memory accesses with which to compare actual accesses. For example, the discovery that a kernel module is placing more calls per operation can reveal additional malicious functionality as a result of overwritten or injected code. If a particular instruction is not being called as often as expected, but the standard callees are submitting the same amount of requests, then it is possible that a filter is redirecting traffic to a malicious copy of that module [73] [9] [69] [84].

Dynamic memory forensics is discussed in Section 5.3.3.

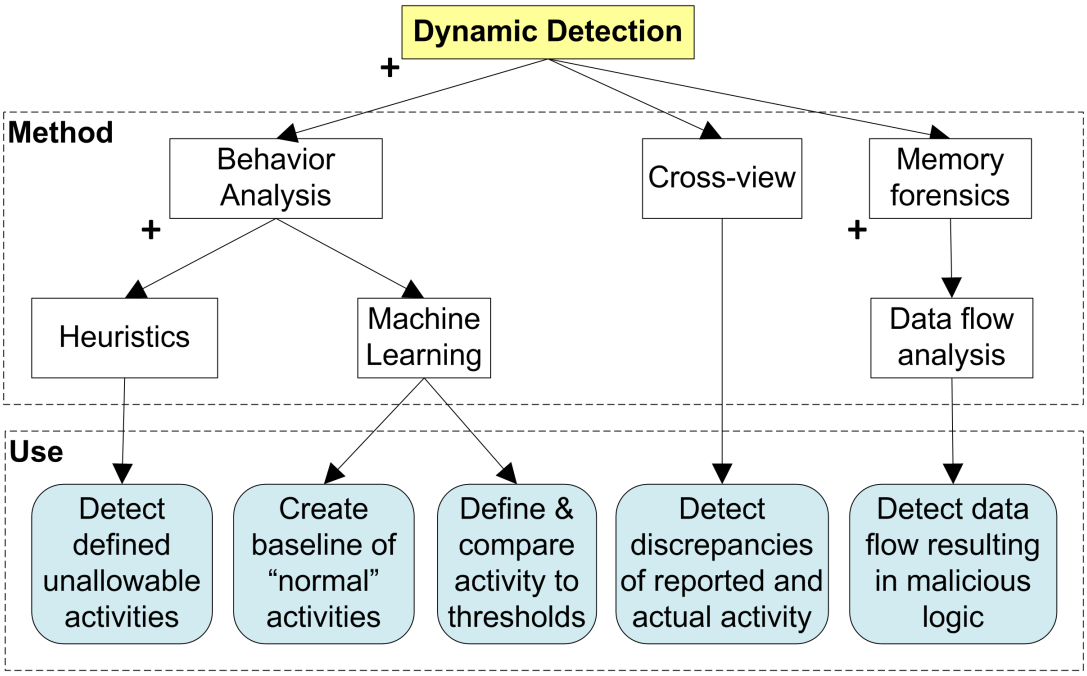

Figure 5.2: Dynamic Rootkit Detection Taxonomy

5.3 Dynamic Rootkit Detection Techniques

Dynamic detection methods exist to detect conditions that only occur while a system is operational. Information accessed through dynamic analysis is continually changing, and so detection techniques also adapt to changes in the environment. It is often the inconsistencies in system operation that provide indication that malicious activity is underway.

The one major downside to dynamic detection is that it is a purely reactive technique. Any indicators of malicious activity discovered are proof that a rootkit has already infiltrated the system to the point where it can successfully perform those activities. However, without dynamic detection many static indicators of rootkit activity would go undetected as a result.

Dynamic detection methods are broken into the following categories. *Behavior analysis* searches for evidence of rootkit objective fulfillment. *Dynamic heuristics* involves the rules and rule–creation for defining malicious activities. *Dynamic memory forensics and mapping* reveal ways of detecting suspicious activity through memory accesses and allocation events [62].

Figure 5.2 demonstrates a high level association between the relationships between static and dynamic detection methods, and what kind of rootkit indicators they target.

5.3.1 Dynamic Behavior Analysis

Rootkit activities that result in the rootkit fulfilling its intended objectives, such as gaining persistence, hiding objects and resources, initiating a C2 channel or disabling security devices, can also create a unique set of indicators that reveal the presence of rootkit activity. Behavior analysis attempts to detect a rootkit's presence not by revealing its installation indicators but by the activities performed as a result of the installation.

Behavior analysis can be performed through two main strategies: *dynamic heuristics*, which look for specific indicators of a particular rootkit activity, and *machine learning*, which takes the opposite approach, to define how a system behaves normally, and then looks for any deviation from normal behavior. Machine learning can be used to create a baseline from which heuristic rules are then created [16] [63].

5.3.1.1 Heuristics

Heuristic rule sets can be used to detect dynamic indicators of rootkit activity. Often, these rule sets are applied to types of activity, numbers or patterns of system calls seen in real time, as defined to be unallowable or potentially malicious. Often these indicators are obtained from virtual environments [16]. The dynamic application of heuristics often reveals indicators of a rootkit actively fulfilling its operational goals, such as process hiding and modifying data and code in protected regions of a system. Dynamic heuristics expand on the successes of static heuristics as they can provide results and adapt to events in real time.

Dynamic heuristics may monitor the integrity of certain regions of kernel code for evidence that they may have changed. For example, the SHARK hardware–based rootkit detection tool continually verifies the hashes for critical kernel regions and process identifiers [77].

Execution path analysis is a dynamic heuristic rootkit detection method which counts the number of instructions executed and then looks for statistical deviations from the expected number of instructions. This is performed by using the x86 single step mechanism to interrupt after each instruction in order to count each instruction. This information is then statistically analyzed and compared to hooked and unhooked versions of kernel APIs. Contrary to execution path analysis, however, heuristics are not ideal for detecting DKOM attacks, which may not manipulate execution paths at all [73].

Heuristic analysis may fail if the rules analyzing the system do not dynamically update with system activity. According to Ahn *et al.*, APT and zero day threats cannot be detected

strictly through pattern matching methods, which include those based off signatures, rules, and blacklisting. This necessitates the need for dynamic detection methods which include both heuristics and data mining techniques [1]. These techniques are discussed in Section 5.3.1.2.

5.3.1.2 Machine Learning

Machine learning is a technique which employs flexible rule sets based off interpretations of large amounts of data, in which both the dataset and the interpretations are continually updated. Machine learning is closely related to big data, data mining, and other sciences built around the gathering and interpretation of large amounts of dynamic data inputs [16]. Machine learning as applied to possible indicators of rootkit activity is a way of detecting and predicting kernel attacks, even those not seen in the wild before [66] [5]. As a result, heuristic analysis benefits from interpretations of this data.

Machine learning is especially useful to generate predictions of possible attacks based off classification, text mining, clustering, and association [1]. According to Ahn *et al.*, big data analysis applied in machine learning involves four stages: prediction, classification, relation, and analysis of atypical data [1]. Prediction of attack possibilities is based off known past attacks using regression analysis from attack logs. Classification can group new attacks based on markers from similar attacks. Relation links events based off user activities, and sequences based on time flow Atypical data consists of other information such as pictures, text, video, and even social mining [1] [2].

In addition to the collection and classification of large amounts of data, machine learning is a continually developing field with no defined methodology, but several experimental techniques to determine an effective way to convert massive amounts of data into actionable information.

For example, the malware target recognition (MaTR) architecture proposed by Dube *et al.*, addresses malware detection by employing both static and dynamic techniques with machine learning for an attempt at a complete detection solution [16]. MaTR employs a three–layer pyramid structure with static detection serving as the foundation, followed by dynamic detection methods, and finally results which are determined by human operators. Static methods are used to quickly scan for candidates for further analysis based off structural anomalies such as feature sets, nonstandard section names, entry points, and imports/exports of both code and data. Candidates for dynamic analysis are extracted from this set, and processed according to machine learning. Machine learning in MaTR is applied as a decision tree with a machine

learning classifier. Classification decisions are implemented as tree branches and nodes with each branch applying a different classifier to the sample, and the path from root to leaf determining the final classification of the sample [16].

Additionally, data collection to support data mining occurs across a wide vector of resources, not just the local machines. Some of these resources are Firewalls/NIDS, HIDS, databases, web, application, and other sources. Consequently, machine learning is typically applied to malicious activity across an array of vectors, not just in the search for rootkits alone [1].

5.3.2 Crossview Detection

Crossview detection specializes at uncovering rootkit concealment activities by comparing two different views from the same machine. This is also called "difference–based comparison", and often reveals evidence of rootkit code and process hiding. For example, to detect hidden files, a crossview detection utility could compare the results of a raw object enumeration with that of an application using the system API [9] [77].

One notable project, Strider GhostBuster [83], attempts stealth malware detection by implementing a crossview, difference-based approach. The GhostBuster utility generates a high level view of a system, which is expected to miss information that the stealth software has hidden about itself, and then a low level view of the system which attempts to collect the "truth" about files and processes through data collection methods that bypass system resources, such as enumeration APIs [83].

GhostBuster obtains a high level view of the processes presented by the system by using the `NtDll!NtQuerySystemInformation` API. The low level view is obtained by using a driver to manually traverse the Active Process List. This lower scan, however, is not entirely accurate because it does not intercept processes hidden via DKOM, and so a second scan traverses a separate kernel data structure, which provides process list information for non-enumeration OS functions [83].

These scans can be performed "inside-the-box" or "outside-the-box". The outside-the-box approach requires making the relevant memory address space available external to the system with the assumption that data could be retrieved via Direct Memory Access (DMA) and not alert an infected system. However, there is no guarantee that a stealth software would not intercept and trap events presented to GhostBuster [83].

The GhostBuster project was successful in detecting Aphex, Hacker Defender, and Berbew

through comparisons centered around the Active Process List. The FU rootkit could only be detected with advanced configuration settings but because the Vanquish rootkit injected its process into several DLLs the GhostBuster detection tool discovered many instances of these injections. Overall, GhostBuster aims to discover "the truth" via low-level scans but only makes the assumption that what is discovered is only a "truth approximation" [83].

5.3.3 Dynamic Memory Forensics and Mapping

Dynamic memory forensics methods are much the same as static memory forensics, except detection methods require a running system and are used to monitor memory accesses that only occur while the system is in operation.

Certain types of information flow can reveal distinct evidence of malicious activity. For example, memory allocation and deallocation events can be used to generate a good baseline for cross-view detection: if dynamic kernel modules are being loaded but do not report the activity, then this is suspected to be liveness falsification [62].

To detect SSDT hooks one possible method is to load a driver that scans the SSDT and compares the addresses to those in `ntoskrnl`. If an SSDT address is outside this range, then there is a good chance that address is associated with a hooked module. Removal of an SSDT hook is a little more difficult than detecting the hook. The original address of the API must be retrieved, and re-written over the malicious address in the SSDT [63]. However, this does not remove the original malicious code, leaving it available to be used should that address be known by other malicious modules.

5.3.3.1 Data Flow Analysis

In contrast to static data, which consists of hard–coded values or configuration information, dynamic data must be observed in real time. Data flow analysis techniques examine data inputs and outputs throughout the system to detect malicious behavior resulting from those values. Although malware can reuse existing data structures maliciously or obfuscate the content of code in order to hide the fact that the code is being used for a malicious purpose, passing data bypasses the need to modify data structures. Because many detection tools look for changes to code malicious data is often overlooked [58].

Data-focused detection methods can detect semantic value manipulation (SVM) attacks and do not require detection of injected code or malicious code sequences [58] [62]. Data flow analysis is considered to be a method of "semantic aware detection".

Semantic aware detection is a method of determining that malware is malicious based on its response when provided certain inputs. Krueger *et al.* statically detect kernel level drivers by having sample code symbolically execute inputs intended to discover "improper kernel memory accesses" [51]. The fuzz testing utility, MOSS, developed by Prakash *et al.*, evaluates the mutability of semantic fields by mutating data inputs [58].

Malicious logic resulting from bad data can mask the true effects of a sequence of system calls, or result in falsified reporting of active processes. These discrepancies can be hard to detect as it is often the case that misreported or malicious values do not result in deviant behavior every time information is passed. For example, the Stuxnet rootkit reprogrammed the PLCs to cause the centrifuges to intermittently spin faster than was allowable. In theory, if an externally applied cross-view system were able to monitor the values from the PLC's commands, this attack may have been detected because reported centrifuge operation values did not match the actual values [79].

Data-based detection scenarios can text the outputs and effects of data values from these seemingly legitimate data structures to detect malicious behavior [62]. One downside to data-based detection, however, is that value-based integrity checking does not always work. Malicious values may fall within established thresholds [62].

Data access patterns can also be used to generate a limited type of malware signature. The DataGene utility [69] proposed a way to monitor data in memory that is owned and accessed by malicious code. Access patterns can reveal the presence of the malware, but this approach is limited enough by parameters in the operating environment to not produce a universally useful or robust signature [69].

5.4 Detection Execution Platform

There are three primary types of platforms from which a detection tool can interact with the host operating system: 1) 3) host–based or networked, 2) hardware–based, or 3) virtual. Local, or networked machines are generally the easiest platforms from which to host a detection tool, but are vulnerable to the kernel–level malware. Virtual environments are commonly used to test incoming files and binaries for malicious behavior before forwarding them to the intended recipient. Hardware–based solutions can provide immunity to the malware, but are less developed.

Figure 5.3 displays a simple taxonomy of the types of execution platform available, from

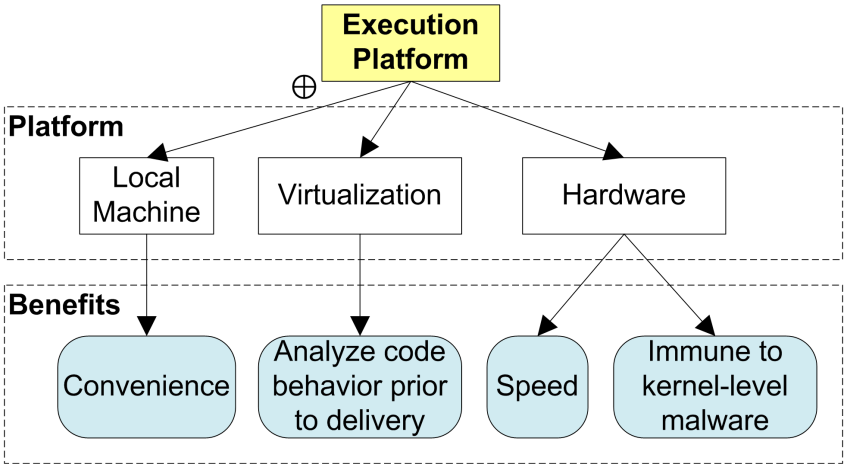

Figure 5.3: Rootkit Detection Execution Platform Options

which detection utilities can operate.

5.4.1 Local Execution

As addressed in Section 4.3.1, any kernel–based malware detection system can be subverted by rootkit activity.

Host–based rootkit detection software, which hook into the same kernel–level data structures as rootkits, have the advantage of observing and detecting rootkit activity. However, they are vulnerable to being hooked by the rootkit as they rely on the compromised system to provide information about itself [9] [4].

Additionally, malware detection tools that run at the application level must rely on the operating system to provide information about the system in order to determine that malicious activity is occurring. These tools can either be directly subverted by the rootkit or merely rendered inactive through the presentation of false information by the rootkit–hooked OS.

For example, the Windows Memory Manager, which is responsible for both paging and translating a process's virtual address space into real address space, provides useful information about memory resources being accessed. The problem lies in that if the rootkit has compromised the Memory Manager, then information reported by this component may not be fully trusted [73]. The Tripwire program, for example, detects file system modifications, but is a user level program, and thus susceptible to rootkit manipulation and deception [4].

A falsified view of the OS, however, can be very useful for crossview detection. For example, the GhostBuster tool creates a high level scan of the OS using the `ls` command line utility. In contrast, the low level view used for comparison is an out–of–the–box image of a clean bootable

CD distribution of the same OS being scanned [83].

Interestingly, the development team for the GhostBuster crossview detection tool was aware that GhostBuster could also be directly targeted by malware. Wang *et al.* decided to incorporate rootkit techniques into GhostBuster so that it could operate without being detected. A kernel–level GhostBuster DLL was created, which then injected into running processes, giving OS utilities such as Windows Explorer, Task Manager, and RegEdit GhostBuster abilities [83].

Ultimately, host–based malware detection applications and kernel utilities are still prone to infection by the rootkit itself, and cannot be fully trusted.

5.4.2 Virtualization

Virtual machine introspection (VMI) is a malware detection technique in which potentially malicious files are executed in a virtual environment while the host OS monitors the guest OS for malicious activities. This technique can be used locally or by network–based IDSs to intercept and execute incoming files in a sandbox–like environment before determining whether the file may be forwarded to its original destination [4].

For example, FireEye's NX 10000 appliance [54] is an external facing IDS that uses a suite of virtual machines (VM) to execute files and look for malicious behavior before the packets are even allowed into the internal network. The VMs are multiple versions of standard operating systems and browsers, so that files can be executed in an environment closely matching the target.

The dAnubis program [51], which is an extension to the Anubis malware analysis utility, uses VMI to look for kernel drivers. Rather than automating the detection process, however, dAnubis monitors all communications with system memory, and then prints out the results for human analysis to make determinations on whether the results reveal a malicious driver.

The Strider GhostBuster cross–view detection utility can be launched using virtualization. As this utility uses an older style of virtualization, the proposed VM is booted from a CD, and then performs both the high level and low level scans in two separate stages. First, GhostBuster injects an ASEP hook to the infected boot drive, and then launches a high level scan on the powered-on machine. After the scan completes, the VM is powered down, the ASEP hook removed, and then the outside-the-box scan is performed. The lower level scan is compared to the high level scan, and possible rootkit activity is reported [83].

There are several benefits to executing malware using virtualization. Virtual machines are

cost effective and convenient, easy to manage, and allows consolidation of resources as existing hardware can be reused for new VMs [4] [84]. Performance is increased, and threats are isolated from affecting internal resources [4]. Additionally, unlike host–based sandbox environments, the target VM is a complete OS, within which the rootkit is functionally uninhibited [84]. Even some utilities claim to not slow down business operations. FireEye claims that the NX10000 device is capable of performing virtual execution of incoming traffic in an enterprise environment, and then forwarding acceptable content at speeds averaging 4Gbps to 10Gbps [54].

In addition to executing files and binaries prior to allowing them to execute on the target machine, VMI can also be used for a type of cross–view comparative detection. The VM can execute the file, then generate a baseline of results for various processes which can then be compared to the execution of the same file or process on another machine, real or VM. However, the obvious problem with this method is that the malware is executing on the victim machine before a comparison between the two images can raise an alert.

Some rootkit samples are able to detect the presence of a VM environment and change their actions. Often rootkits will detect a VM, and then decide to not behave maliciously [84]. Some even have alternate (legitimate) functionality which takes place until it detects that it has been installed on a legitimate host machine.

Some rootkits can even directly attack the virtualized environment. Prakesh *et al.* also warns that the VMI appliance is reading data directly from VM memory, and so if the guest kernel was also compromised, then threat data cannot be considered accurate [58].

A virtual machine monitor (VMM), for example, which consists of a small code size, is considered to be a convenient platform from which to monitor host machine integrity. However, the BluePill rootkit [77] can subvert the VMM and exploit the hardware virtualization support. BluePill implements its own hypervisor and then loads the host OS or original hypervisor as a guest and takes over the entire virtual environment.

Although virtualization provides a cost–effective and convenient method of scanning for rootkits, and is less susceptible to rootkit subversive activities than the target machine, virtualization is not a complete solution.

5.4.3 Hardware

A hardware–based approach is ideal in order to observe the kernel without risk of compromise by kernel–level malware, and to generate a highly accurate raw enumeration for crossview

detection [9].

Hardware-based detection utilities have a substantial advantage over the integrity of local and virtual execution, as they cannot be hooked by kernel–mode–only rootkits. Architectural support is necessary to establish a direct line of communication between hardware and a detection utility, bypassing the kernel altogether [77].

Some possible hardware implementations are a PCI add–in card, or possibly with a Myrinet NIC to retrieve valuable information via Direct Memory Access (DMA) [83].

For example, the Copilot is a hardware-based coprocessor that resides on a PCI add–in card. Copilot serves as a kernel integrity monitor by polling kernel memory for changes in the hash of critical regions of memory or via pre–defined violations [82] [4] [73]. Copilot continually sends the results of the RAM acquisition to an isolated co–processor, which then checks for violations [77].

The Secure Hardware support Against RootKit (SHARK) [77] tool was developed to provide a direct relationship between each software context and machine hardware. SHARK implements a hardware assisted PID generation (HPID) and then establishes a dependency between the HPID and the execution of the process through process table encryption. The SHARK tool then performs a stealth checker by comparing processes presented to an administrator versus actual processes registered through the hardware.

5.5 Rootkit Detection Methods Evaluation

Detection tools, both in industry and in research, encompass a broad range of styles of detection which aim to detect different types of indicators of rootkit activity.

For example, Musavi *et al.* proposed a method of detecting malicious kernel drivers by training a classifier on specific features or activities which were likely to reveal evidence of the driver's malicious intent. These features include: string activity ratio, anti-analysis, number of system calls, disassembly size, and constants such as kernel memory offsets [51]. The GhostBuster tool implemented a cross–view approach by scanning hand–selected regions for comparison, such as enumeration APIs, and was able to be implemented on both the host and external hardware [83]. MaTR employed both static and dynamic heuristics with machine learning to attempt to classify large amounts of threat data [1].

Figure 5.4 demonstrates a high level association between the relationships between static and dynamic detection methods, and what kind of rootkit indicators they target.

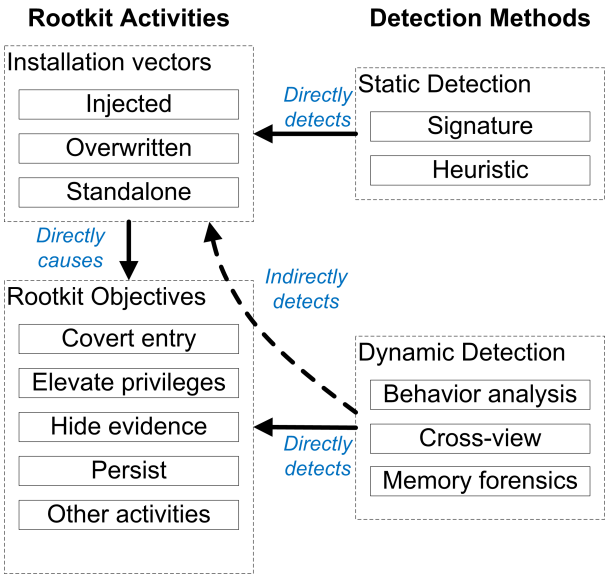

Figure 5.4: Detection Methods Applied to Rootkit Indicators

5.5.1 Coverage Evaluation

To evaluate coverage it is necessary to draw a distinction between the categories of possible indicators of rootkit activity.

It is often the case that combination is the most effective coverage, because no detection method alone can either address all behaviors or all regions within a system. For example:

- Some detection methods may be better than others to detect a particular behavior in one region, but not within another. For example, an integrity verification–based detection tool which monitors the Windows Registry may detect ASEP hooks but the ASEP–hook detection utility would likely not be useful to detect return oriented programming [4].

- Other methods may be applicable to only certain behaviors within a certain region. For example, integrity verification could detect modifications to standard system modules but would not be as effective applied to dynamically loaded kernel modules. Cross–view detection methods may be effective to discover hidden processes, but not necessarily trojanized modules [84].

Figure 5.5 demonstrates how detection methods as expressed in Chapter 5 typically apply to the different types of rootkit indicators. Note that this diagram is not perfect as these detection methods can apply to more than one category of indicator depending on the application of that method.

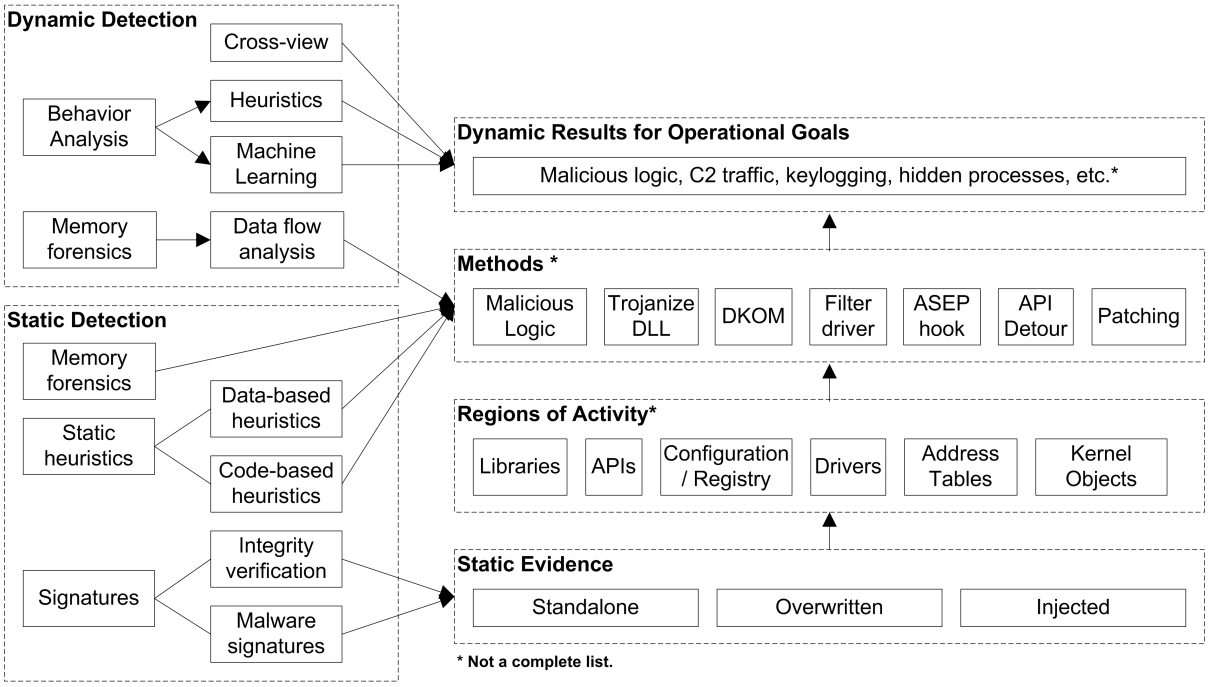

Figure 5.5: Application of Detection Methods to Rootkit Indicators

Detection methods can be used in nearly any combination, applied to nearly any indicator category to detect the rootkit activity itself. Table 5.2 provides an example of a rootkit activity that can be detected through different detection methods, depending on the type of rootkit activity indicator selected.

5.5.2 Conclusions

To gain full detection coverage, it would be easy to say that the goal is to to detect all activity in each of the three basic types of rootkit activity: injection, overwriting, standalone. However, because of the dynamic nature of the kernel not all static modifications can be determined to be malicious. Standalone code which matches malware signatures can be definitively determined to be malicious. Code for which integrity verification fails can be determined to be tampered. However, the operating system is vast, and it is difficult to apply integrity verification to every static component.

Heuristics, both static and dynamic, can identify other types of malicious behavior, such as unallowable system-call patterns or known malicious tampering (e.g., JMP overwrite). However, heuristics are not sufficient to provide definitive rootkit detection but rather tend to collect data that may be 1) previously identified to be malicious and 2) only determined to be through comparison to a range of values.

Indicator Category	Indicator	Detection Method
Simple indicator	Function handle address in SSDT is different than expected address.	Heuristic memory checking rules detect jump to address outside kernel memory region.
Basic static indicator	Overwriting	Integrity verification reveals difference between actual and expected SSDT handle addresses.
OS region	SSDT	Detection tool monitors SSDT specifically for several vectors of infection / modification.
Region type	Address table	Detector monitors all standard address tables for unexpected/unauthorized changes.
Effect	System call rerouted to rootkit code.	Static memory forensics and call mapping, or dynamic heuristics monitoring real time calls, detects abnormal sequence of instructions from originating caller.
Objective / operational goal	Hide malicious activity by filtering inputs so that acceptable values are always returned.	Cross–view detection observes that actual values and reported values differ.

Table 5.2: Detection Methods Applied to Categories of Rootkit Indicators

Crossview detection can detect the presence of misreporting within the system, which is an activity that relies heavily on both assumptions about what data should be monitored and corresponding dynamic information as presented by the system. This strategy fails where there is no alternate data for comparison.

Other methods, such as machine learning, seem to primarily exist to collect data for the purpose of building thresholds so that comparisons may be run against that data.

While signature based strategies and static identification of malicious call sequences help to identify a rootkit's installation, they are not sufficient to provide full detection coverage. Methods which support a thresholds–based analysis, such as machine learning, use real–time data for analysis, and naturally produce and update rules and thresholds for the specific system on which they are implemented.

The combination of these two types of detection do their best to provide full coverage. However, the application of these methods ultimately comes down to the type of detection tool and the type of rootkit indicators the authors of the tool decide to collect. In order to improve the state of rootkit detection, more information is needed regarding the efficiency and difficulty of applying these methods. With metrics that apply to specific systems, the methods discussed

in this thesis can be quantified for efficiency and speed of application, in addition to coverage. Future work in these areas is discussed in Section 7.2.2.

Chapter 6
Application

This chapter demonstrates the rootkit installation and activities taxonomies presented in Chapter 3 and the detection taxonomies developed in Chapter 5 on the ZeroAccess rootkit.

6.1 The ZeroAccess Rootkit

This section describes the ZeroAccess rootkit and provides an overview of the methods utilized by ZeroAccess and possible detection methods to discover ZeroAccess using the taxonomies proposed in this thesis.

Much of the detail in this section is summarized from a thorough analysis by Sophos of the most common ZeroAccess derivative in circulation in 2012 [90].

6.1.1 ZeroAccess Background

ZeroAccess, also known as "Sirefef", is a trojan-like kernel-mode rootkit. This trojan is a component of the ZeroAccess botnet, which had been installed more than 9 million times by the time it was discovered in 2012. One million of these infections were still active as of September 2012 [90], and 1.9 million infections were active in 2013 [88]. ZeroAccess also resurfaced in January 2015 with new infections, but with only approximately 50 thousand actively participating infected PCs.

ZeroAccess infects 32-bit and 64-bit versions of Windows primarily to carry out click-fraud and bitcoin mining. For the click-fraud scheme, ZeroAccess displays ads on a computer and then clicks on them as a user-initiated request. These ad clicks generate revenue for the botnet operators [12]. At the peak of ZeroAccess's operations it is believed to have generated in excess of $100,000 per day [88].

6.1.2 ZeroAccess Weaponization and Delivery

ZeroAccess was a popular payload to use through the Blackhole exploit kit. Weaponized and distributed by Blackhole, ZeroAccess infected PCs through infected websites. The other main avenue of distribution was through victim–downloaded trojanized files. ZeroAccess was commonly downloaded through a trojanized keygen for DivX Plus 8.0 which was hosted on upload

sites and as a torrent. ZeroAccess was also distributed through a trojanized copy of the game Skyrim [90].

6.1.3 ZeroAccess Installation

The ZeroAccess rootkit has been trojanized into both copies of the game Skyrim and an NSIS installer for a DivX Plus 8.0 keygen [90]. The trojanized keygen is presented as a Nullsoft Scriptable Install System (NSIS) [56] self extractor, which compiles scripts into highly customizable executable installation programs. These programs include target directory and installation configurations and usually run at the same privilege level as a standard user. The NSIS self extractor containes both the DivX Plus 8.0 keygen and an encrypted 7zip file. When the file is executed, it unpacks the keygen publicly (`%Profile%\Application Data\Keygen.exe'`) and the 7zip file secretly, extracting the ZeroAccess dropper [90].

ZeroAccess queries the `ZwQueryInformationProcess` API to determine which version of Windows is running and then determines which installation path it will use.

6.1.3.1 32-Bit Installation and Hiding

The kernel-mode rootkit for the 32-bit installer for ZeroAccess loads its code into the kernel by overwriting a randomly chosen system driver. The original driver and other ZeroAccess files are encrypted and stored in a location not normally accessed by applications [44] [90].

Other versions of ZeroAccess hid files by creating a hidden volume. As of 2012, ZeroAccess hid files by creating a legitimate directory under %systemroot%. It encrypts the files with RC4, converts the folder into a symbolic link to make it inaccessible to standard Windows APIs, modifies the Access Control List (ACL) for that directory so that its location is unable to be browsed to.

ZeroAccess hides changes made to the infected driver by hooking the `LowerDeviceObject` of the DR0 device (`\Driver\Disk`). Processes attempting to use the driver are presented with a clean copy of the driver, but ZeroAccess directly accesses the malicious version without using the Windows APIs, which only recognize the folder as a symbolic link.

The rootkit driver creates a device named "`ACPI#PNP0303#2&da1a3ff&0`" which accesses and decrypts ZeroAccess's hidden files as needed [90].

6.1.3.2 64-Bit Installation and Hiding

The 64-bit version of ZeroAccess executes in user memory and does not hook the kernel. To ensure persistence a file is stored in the user's AppData folder and a registry entry is created under `HKCU\Software\Microsoft\Windows NT\CurrentVersion\Winlogon`.

ZeroAccess hides its files inside the Global Assembly Cache (GAC), which does not diplay files and folders, but rather launches the Cache Viewer if Explorer is used to view the directory. (The directory structure can be viewed from the command line, however) [90].

6.1.4 Privilege Escalation

The first attempt to escalate privilege happens when ZeroAccess calls `RtlAdjustPrivilege` to give itself `SE_DEBUG_PRIVILEGES`, which will only succeed if the user initiating the ZeroAccess installation is an Administrator. If the user is logged in as a Standard User then ZeroAccess generates a User Account Control (UAC) popup, which requires an Administrator login.

At this step, social engineering assists privilege escalation. ZeroAccess will not generate a UAC popup for the purported keygen or Skyrim install, but for a more common program such as Adobe Flash Installer. ZeroAccess first places a clean, legitimate copy of Adobe Flash Installer into a temporary directory, and then places a malicious DLL *msimg32.dll* into the same directory. Windows prioritizes the execution of local DLLs over that of DLLs in the system directory, which allows ZeroAccess to abuse Windows' DLL load order so that the malicious program can run in the legitimate file's process address space [90].

6.1.5 C2 / Botnet

The ZeroAccess communication channel is initated from within the kernel driver and an injected user memory component, either within `explorer.exe` or `svchost.exe` [90].

Some variants of ZeroAccess communicate to an IP address during the installation process to report information about the compromised machine. Using an HTTP GET request, the domain name displayed in the "Host" field is randomly generated by a Domain Generation Algorithm (DGA) which generates a new name within the `.cn` domain daily. The connection itself does not require this name, but the Host name is used for authentication, by which the C2 servers can verify that the request is from a legitimate instance of ZeroAccess before responding [90].

ZeroAccess operates as a peer-to-peer botnet and while it communicates infected machine information to a C2 server, the payload is connected to a list of peers for downloading additional

files.

Network communication is either initiated directly by the kernel driver or by a component that the infected driver placed in user memory, usually in the address space of `explorer.exe` or `svchost.exe`.

The bot executable file contains a list of 256 peer IP addresses and attempts to contact each one on a fixed port number. It also listens on the same high-numbered TCP port as used for outgoing connections.

When a ZeroAccess bot connects to another bot, communication is sent using RC4 encryption with a fixed key, shared by all variants of ZeroAccess. The bot sends a `GetL` request, which the peer bot responds with RetL: a list of its 256 IP addresses and files it downloaded with associated timestamps. A subsequent `GetF` request lets the bot download each file from that peer. This provides each bot with an updated list of active peers and the most current versions of the exploits and malware for the botnet [90].

6.1.6 Additional Activities

Using its newfound privileges, ZeroAccess now takes liberties to disable several security services. It disables Windows Updates first and then the following:

- Base Filtering Engine (BFE) service, which can disable antivirus scanning capabilities.
- IP Helper service (iphlpsvc), which retrieves network configurations about the local computer.
- Windows firewall service (mpssvc), a host-based network protection and monitoring service.
- Windows Defender (WinDefend), the Windows-based anti-virus/anti-spyware solution.
- Windows Security Center service (wscsvc), the location from which users manage the firewall, malware protection, and other security settings.
- Proxy Auto Discovery service (WinHttpAutoProxySvc), which provides components for facilitating HTTP requests and responses.

A self-defense mechanism included in the 32-bit verson uses a “bait” process [90], also known as a “tripwire” process [44], which contains an alternate data stream. If any process

attempts to execute the bait file, ZeroAccess changes the ACL of the file for that process, then injects shell code into the process which terminates it. This assists in disabling security tools, particularly those which use emulation by sandboxing and executing files to determine if they exhibit malicious behavior [90].

6.2 Taxonomic Evaluation of the ZeroAccess Rootkit

This taxonomic evaluation of the ZeroAccess rootkit is based on the DivX Plus 8.0 keygen method of delivery. The examples presented in this section demonstrate how rootkit activities and detection techniques presented in this thesis relate and can potentially provide full coverage of all rootkit activities.

Not all possible taxonomies for each rootkit activity are presented. Likewise, not all possible detection taxonomies are presented for each indicator. The application presented in this section intends to demonstrate that each basic type of indicator may have multiple applicable activity taxonomies, each of which may have multiple detection taxonomies. This application is not intended to prove the specifics of full coverage, as this could include thousands of indicators, each with several taxonomies, and multiple detection techniques applied to each taxonomy.

Table 6.1 applies rootkit activity and detection taxonomies to one indicator for injection–based rootkit activities. Likewise, Table 6.2 addresses taxonomies applicable to overwriting–based indicators, and Table 6.3 applies taxonomies to standalone indicators.

Rootkit Modifications & Taxonomies		Detection Taxonomies	
Injection–based Indicator	Activity	Static	Dynamic
Modify contents of `%WINDIR%\debug\usermode\userenv.log` [43]	Hide process → Plain sight → False intent → Misreporting	Signature → Integrity Verification Memory forensics → `userenv.log` accessors	Crossview → Event Ommission/Modification Memory forensics → Data flow analysis → Real–time accesses

Table 6.1: Injection–based ZeroAccess Rootkit Indicators and Corresponding Detection

Rootkit Modifications & Taxonomies		Detection Taxonomies	
Overwriting–based Indicator	Activity	Static	Dynamic
Overwrite system driver	Hide physical presence → Within file system → Hidden → Inside application	Signature → Integrity Verification Memory forensics → Memory region accesses	Memory forensics → Data flow analysis Behavior analysis → Machine learning → System call / memory access patterns
	Hide process → Hidden → Invisible → Injection	N/A	
Hook DR0 `LowerDeviceObject`	Hide physical presence → Outside file system → Hook existing object	Signature → Integrity verification Memory forensics → Memory region accesses	Behavior analysis → Heuristics Memory forensics → Data flow analysis
	Hide process → Plain sight → False identity		
Modify Group Policy Registry keys	Hide process → Invisible → Disable security	Signature → Integrity verification	Behavior analysis → Machine learning → Security service behaviors
Modify ACL of file for process accessing bait process.	Hide process → Invisible → Disable security	Signature → Integrity verification Static heuristics → Data–based heuristics	Behavior analysis → Heuristics → Security service liveness Memory forensics → Data flow analysis

Table 6.2: Overwrite–based ZeroAccess Rootkit Indicators and Corresponding Detection

Rootkit Modifications & Taxonomies		Detection Taxonomies	
Standalone–based Indicator	Activity	Static	Dynamic
New malicious DLL in Adobe Flash Installer directory	Hide physical presence →Within file system → Plain sight → False identity → False name	Signature → Integrity verification	Behavior analysis → Heuristics Memory forensics → Data flow analysis
New files in unbrowseable locations within file system.	Persistence → Survive removal → Survive disable → Difficult to access	Static heuristics → Code–based heuristics Memory forensics → Memory accesses to unbrowseable regions	Crossview → File enumeration Memory forensics → Data flow analysis
	Hide physical presence → Within file system → Hidden → Unbrowseable		
New kernel loaded filter driver	Persistence → Survive removal → Survive disable → Dangerous to access	Static heuristics → Code–based heuristics	Behavior analysis → Machine learning Memory forensics → Data flow analysis
	Hide physical presence → Hide physical presence → Hide code outside file system	Signature → Code–based heuristics	Memory forensics → Data flow analysis
New files hidden in GAC	Persistence → Survive removal → Survive disable → Difficult to access	Memory forensics → Memory region accesses	Crossview → File enumeration Memory forensics → Data flow analysis
	Hide physical presence → Hide within file system → Hidden → Unbrowseable		
New application `ACPI#PNP0303#2&da1a3ff&0`	Hide process →Hidden → Plain sight → Own identity	Static heuristics → Calling pattern Signature → Malware signatures Memory forensics → Memory region accesses	Memory forensics → Data flow analysis Behavior analysis → Machine learning

Table 6.3: Standalone–based ZeroAccess Rootkit Indicators and Corresponding Detection

6.3 Application Conclusions

The application of the taxonomies to only a few of the ZeroAccess rootkit indicators reveals a few patterns which may potentially apply to all rootkit indicators.

1. Each static–based indicator can be used in more than one way. For example, overwriting can be used to modify pointers in kernel code or address tables, or it can be used to change the values in the Registry to modify Group Policy settings.

2. Each indicator can be associated with multiple taxonomies. For example, hooking the `DR0 LowerDeviceObject` hides both the physical presence of the malicious activity as the change is unseen, but also hides its process by falsifying its identity as a legitimate system service. Hiding new rootkit files in an unbrowseable location both helps the rootkit persist as well as hide its physical presence.

3. Each taxonomic representation of a rootkit activity can potentially be associated with more than one detection taxonomy. For example, the `ACPI#PNP0303#2&da1a3ff&0` device is "hidden" within the normal Windows file system but it can be detected using multiple methods. This device is known by name and thus could be detected with a malware signature. Memory forensics may discover that this device communicates with an unbrowseable part of the file system, while dynamic machine learning could be trained to recognize a decryptor device.

4. Some detection methods apply to more than one rootkit activity. For example, new files in an unbrowseable folder are placed in that region for both persistence and hiding. Static heuristics can detect the persistence of the files by analyzing all unbrowseable regions of a file system, or static heuristics combined with memory forensics can detect that memory accesses connect to that region.

5. Not all detection methods are useful for detecting all indicators of rootkit activity. For example, crossview detection is not a good method for detecting injection inside kernel modules unless the immediate result of that injection itself goes misreported at another level.

Through this application, it is possible to conclude that rootkit detection techniques can potentially provide full coverage to detect all rootkit activities. However, the efficiency and

practicality of combining these techniques is not yet proven. See Section 7.2.2 for possible future research and projects in order to determine how detection techniques may be efficiently applied to provide full coverage detection of kernel–mode rootkits.

Chapter 7
Conclusions and Future Work

This chapter summarizes the research presented in this thesis and proposes ways that this research could be furthered to benefit the scientific community. Section 7.1 presents a summary of rootkit activities and detection methods, as well as conclusions derived from the research and taxonomies presented in this thesis. Section 7.1.2, in particular, highlights the contributions this thesis provides to the existing rootkit knowledge base. Section 7.2 proposes future research and project ideas based on and inspired by this thesis.

Table 7.1 provides a simple roadmap for topics discussed in this chapter.

7.1 Summary of Work

This section summarizes the work performed in this thesis, describes how it addresses objectives stated in Section 1.3, and meets the contributions identified in Section 7.1.2 in Chapter 1

7.1.1 Research Summary

This section discusses information gleaned from research on both rootkits and their detection techniques, and then presents conclusions based on this research.

7.1.1.1 Rootkit Summary

Rootkits are debatably the most critical component to the success of APTs. However, rootkits often go undetected upon delivery and installation because of social engineering tactics. Rootkits are dangerous because they are able to gain access to the system kernel, and then perform all malicious activity with system–level privileges. Once a rootkit gains access to the kernel, the rootkit completes installation using methods to ensure persistence while hiding evidence of the physical and behavioral presence of the rootkit. After installation is complete, rootkits open up a backdoor for communication to either a server or other compromised machines and download and hide additional malware.

In order to detect and mitigate the threat of APTs, rootkit detection is critical. However, in order to effectively detect kernel–mode rootkits, the scope of behaviors and system resources manipulated by rootkits must first be understood.

Topic	Sub–topic	Section
Summary of Work		7.1
	Research Summary	7.1.1
	Contribution	7.1.2
Future Work		7.2
	Taxonomy Expansion	7.2.1
	Tool Evaluation	7.2.2

Table 7.1: Chapter 7 Roadmap

The rootkit lifecycle can be depicted via the kill chain model, with the steps summarized as follows:

- **Step 1: Reconnaissance** (Section 3.2) is the stage where the attacker gleans as much information about the target as possible to ensure that the malicious package appears relevant to any targeted insiders and is technically able to take root on the targeted systems.

- **Step 2: Weaponization** (Section 3.3) is the stage at which the attacker selects and prepares the malicious package. This package includes the method of delivery, the carrier container or file, the exploit that launches the rootkit process, and the rootkit itself.

- **Step 3: Delivery** (Section 3.4) is the stage at which the attacker deploys the weaponized rootkit. This method could be active or passive, depending on the desired target, and can be via physical, such as a USB, or cyber–only, such as web or email. The scenario commonly used in this thesis is that of a spearphishing attempt, where an attacker sends a malicious business–relevant document through email to a specific individual at the targeted company.

- **Steps 4–5: Exploitation & Installation** (Section 4.1) are the two critical stages for rootkit activities, with exploitation repeatedly occurring throughout the rootkit installation process. At this stage, the rootkit plants code, modifies configuration settings, overwrites pointers, and generates malicious logic through manipulating the order of system calls. These activities allow the rootkit to not only plant its code, but in such a way that the rootkit persists and hides evidence of itself.

- **Step 6: Backdoor / C2** (Section 4.2) is the stage at which the rootkit establishes an

external connection to other rootkit services, typically a C2 server or other compromised machines as part of a botnet.

- **Step 7: Actions** (Section 4.3) is the stage at which the rootkit helps facilitate further compromise. Most commonly, the rootkit uses the backdoor connection to download and launch additional malware, uses techniques to hide evidence of the new malicious activity, including disabling security services.

A rootkit's operational goals as described in Section 4.1.1 consist of 1) covertly entering a victim's system (Steps 1–3), 2) exploit the host to elevate privileges (Step 4), 3) install and persist while hiding evidence of itself (Steps 4–5), and 4) perform other simple malicious activities (Steps 6–7). These goals are directly fulfilled through the rootkit's installation, and these goals also translate more directly to dynamic detection methods than to static detection methods.

A kernel–mode rootkit's installation takes place by modifying static system attributes, specifically through 1) injection, 2) overwriting, and 3) the placement of standalone code or executables. Any run–time rootkit behaviors or activities can be directly attributed to some malicious influence on the system generated by some combination of these three static attributes.

7.1.1.2 Detection Summary

Detection methods can be either analyzing 1) static attributes of a system, 2) dynamic behaviors, or 3) a combination of both. Analysis techniques can emphasize analysis of static code, or static or dynamic data attributes.

Static detection methods can be summed up as either applying 1) signature based detection or 2) pattern–matching heuristics to determine that code or data has been modified or has the potential to perform malicious actions through its inherent logic. Static detection methods can detect rootkit indicators directly relating to the rootkit's static installation, such as evidence of injection, overwriting, and standalone code.

Static heuristics can analyze code intention and detect some instances of malicious logic and filter drivers. Static detection methods are not effective at detecting run–time indicators. Run–time indictors include: injected processes, process hiding, and abnormal numbers of system calls. Additionally, static detection methods cannot always make determinations that a modification is malicious because the use of that modification may not be apparent.

Dynamic detection methods monitor the running system and look for runtime indicators. Behavior analysis analyzes the system for evidence of rootkit activity, particularly the fulfilment of its objectives, namely process hiding. Behavior analysis is achieved through heuristics supplemented by machine learning techniques which adapt to changes in the system over time and make determinations from patterns in large amounts of data. Crossview detection compares two different views of the same system to search for discrepancies in information communicated to the system and to the user, which helps reveal rootkit indicators such as process hiding and liveness falsification.

Dynamic detection methods can detect many of the same indicators as static detection methods, but apply those indicators to a dynamic environment. For example, integrity verification of kernel code can be checked continually in a dynamic environment.

It is possible that the application of both static and dynamic detection techniques can provide full coverage of all rootkit indicators and each of the different detection methods can be used alone or in combination to detect the same rootkit indicator. The application of these methods into modern detection tools is left up to the discretion of the developer. Research surveyed does not clearly indicate the most effective blend of techniques for rootkit detection.

7.1.1.3 Conclusions

1. ***Kernel–mode rootkits are supported by a static installation.*** Because all malicious behaviors of a kernel–mode rootkit are rooted in static configurations, detecting its static installation of injection, overwriting, and standalone code could be considered ideal. Revealing every static indicator of a rootkit's existence would not only reveal the full extent of the rootkit's installation and capabilities, but would also conveniently help facilitate any subsequent removal attempts (notwithstanding the difficulties of restoring each modification to its original state). Additionally, detection of the first stages of a rootkit installation could help advance rootkit detection technology from a purely reactive state to a semi-predictive state, preventing the rootkit from completing its installation and ultimately preventing it from fulfilling its mission.

2. ***It is not feasible to identify a kernel–mode rootkit by static indicators alone.*** There are two major problems with the assumption that detecting a kernel–mode rootkit can be performed through static–only detection. First, not every static modification performed by a rootkit can be seen as definitively malicious. Therefore, dynamic detection

techniques are necessary to observe the behavior of a system in order to determine that certain static attributes are used to generate malicious or deviant behavior. Second, rootkit installations are launched through unpredictable exploits. Depending on the nature of the exploit, installation can unfold in any arbitrary sequence of events, provided adequate privileges to necessary resources have been obtained by the rootkit. Because of this, detection techniques would need to monitor every possible rootkit installation vector within the system in order to detect the earliest stages of installation.

3. ***Detection methods provide full, overlapping coverage, but the most efficient application of these methods is not clear from research.*** Both static and dynamic detection methods can detect multiple aspects of a single rootkit indicator. There were no rootkit indicators for which a detection method did not exist. However, the likelihood of detecting that particular indicator given related indicator interactions and the state of the system is much less clear from the research, for which more information is necessary. Additionally, preferred combinations of rootkit techniques were not addressed in the research surveyed, as many tools tended to focus on detecting a single rootkit indictor using a single method.

4. ***More information is needed to properly evaluate tools and techniques to provide a realistic full-coverage rootkit detection scenario.*** While it is possible to assume a full-coverage scenario by choosing some combination of methods that apply to each vector of rootkit installation and behaviors, there is little data to support the practicality of such a scenario. Detection metrics for various implementations and proposals for which methods would work best in specific environments would greatly enhance decision–making. This topic is addressed as a possible future project in Section 7.2.2.

7.1.2 Contribution

The contribution of this thesis as identified in Section 1.4, aims to contribute to the scientific commmunity in the following ways. Each identified contribution objective is highlighed along with an explanation of how this thesis met that objective.

1. ***Providing a taxonomic framework through which rootkit objectives, activities, and detection methods can be more easily understood and communicated.***

   This thesis provides taxonomic frameworks for rootkit objectives, in particular actions

taken to hide both its dynamic and static presences, and rootkit installation activities. A taxonomic framework for detection methods as applied to rootkit installation and behavioral indicators is also presented.

2. ***Making it easier to understand exactly where existing rootkit detection tools provide rootkit detection coverage and for which rootkit activities these tools could be further developed in order to improve coverage.***

   The detection taxonomy provides a framework for evaluating the capabilities of rootkit detection tools. Although this thesis does not directly evaluate specific rootkit detection tools to evaluate their coverage, the taxonomy presented provides a method by which tools and other malware detection methods may be evaluated for rootkit detection coverage.

3. ***Providing graphical depictions which increase understanding of rootkit activities and how they relate to their respective attack spaces and to each other. These visualizations also help facilitate understanding of how detection methods apply to the rootkit attack space and demonstrate both overlaps and oversights as related to existing rootkit detection techniques.***

   Each rootkit activity and detection method is accompanied by a taxonomic depiction demonstrating the relationships or objectives of each vector of activity.

7.2 Future Work

In order to advance the science of rootkit detection, there are several areas in which more information is necessary in order to determine a) the most effective application of detection methods, and b) areas in which the science is inadequate to make those determinations. These topics are both best addressed through research involving the collection and evaluation of existing information, and through projects, the output of which fills in identified knowledge gaps.

Section 7.2.1 identifies areas in which the taxonomy proposed in this thesis can be expanded to include more robust information, and contribute to decisions on how to apply rootkit detection methods for full coverage. Section 7.2.2 proposes projects in which existing rootkit detection methods and tools can be analyzed in order to improve the knowledge and application of information about current rootkit detection technologies.

7.2.1 Taxonomy Expansion

The kernel–mode rootkit taxonomy presented in this paper is by no means complete. The focus of this thesis is centered primarily on kernel–only rootkits, but this is only a subsection of possible modern rootkit implementations.

Section 4.4 addresses advanced rootkits which utilize some kernel–mode capabilities, but are not limited to this domain of computing. These types of rootkits are not dependent on the OS, are less likely to be detected using standard heuristics, and can only be detected through the use of targeted tools and techniques [73].

Some possible rootkits to add to the existing taxonomy include:

- Virtualization rootkits, the detection of which necessitates feedback that bypasses the compromised kernel. For example, the SHARK architecture uses a crossview detection strategy to compare hardware feedback to OS–provided data to detect malicious activity [77] [73].
- Memory–based rootkits, which generally evade static detection techniques as they run only in memory. These rootkits, however, still rely on other static configurations to some degree [77].
- Hardware–specific rootkits, such as GPU–based rootkits [25], System Management Mode (SMM), PCI, MBR/"bootkits", and BIOS rootkits [86] [73] [9] [22].
- SCADA/ICS rootkits. For example, the rootkit used in the Stuxnet APT [91] [72].

These advanced rootkits, which use a combination of kernel–level techniques and specialized methods, provide a new layer of information that can be researched, evaluated, and appended to the existing rootkit taxonomy. Detection methods for these rootkits should also be evaluated, and determinations of the coverage for each detection method can be either appended to the existing taxonomy, or entered into a new taxonomy to address these advanced threats.

In particular, research on rootkits which specifically target ICS/SCADA systems may necessitate the creation of a new taxonomy to understand both the limited scope of rootkit activity and limited possible implementations of ICS/SCADA rootkit detection methods. Additionally, research should address the difficulties of implementing malware detection platforms in SCADA systems and the scope of damage possible through certain implementations of these detection methods.

7.2.2 Tool Evaluation

One major deficiency in existing information regarding rootkit detection tools and methods is a practical means of comparing these detection resources. It would be highly valuable to develop a framework to understand the applications, coverage, and efficiencies of known tools and techniques. Additionally, information can be created by implementing these various tools and techniques, so that data exists by which efficiencies and coverage may be evaluated.

- A rootkit detection comparison framework would help both research and industry determine the most practical rootkit detection methods, as well as plan future rootkit research projects (both improving existing technology and developing new technologies).

  This framework should facilitate understanding the scope of coverage and application of known rootkit tools and techniques, by comparing and differentiating detection applications across different platforms and by the different rootkit indicators detected. Critical information to be included in this framework is that of resources utilized by any detection system, so that detection capabilities can be evaluated to be practical and provide best coverage. In particular, this detection framework should support quantifiable data, such as resource–specific metrics and the load of both the detection tool and the system under analysis.

- Data collection on existing rootkit detection tools and techniques may be necessary to support a rootkit detection comparison framework. This project would consist of selecting a few rootkit detection techniques – with similar implementations and which intend to detect similar rootkit indicators – and running tests to compare the efficiency and coverage provided by these tools under a fixed set of conditions. Tool comparison information is not readily available and would be highly valuable for decision–making for future research and rootkit detection tool implementation.

  Once sufficient data is collected on tools with similarities to one another, it may be beneficial to expand this research to include different types of implementations. For example, VMI tools versus hardware–based tools for detecting injected processes.

Bibliography

[1] S.-H. Ahn, N.-U. Kim, and T.-M. Chung. "Big data analysis system concept for detecting unknown attacks". In *2014 16th International Conference on Advanced Communication Technology (ICACT)*, pages 269–272.

[2] Y. Altshuler, N. Aharony, A. Pentland, Y. Elovici, and M. Cebrian. "Stealing Reality: When Criminals Become Data Scientists (or Vice Versa)". *IEEE Intelligent Systems*, 26(6), Nov. 2011.

[3] E. Baize. "Developing Secure Products in the Age of Advanced Persistent Threats". *IEEE Security Privacy*, 10(3):88–92, May 2012.

[4] A. Baliga, L. Iftode, and X. Chen. "Automated containment of rootkits attacks". *Computers & Security*, 27(7):323–334.

[5] T. Ban, R. Isawa, S. Guo, D. Inoue, and K. Nakao. "Application of string kernel based support vector machine for malware packer identification". In *The 2013 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8.

[6] A. Beuhring and K. Salous. "Beyond Blacklisting: Cyberdefense in the Era of Advanced Persistent Threats". *IEEE Security Privacy*, 12(5), Sept. 2014.

[7] R. Bhat. "Return Oriented Programming (ROP) Attacks". *InfoSec Institute*, 2015.

[8] P. Bhatt, E. Toshiro Yano, and P. Gustavsson. "Towards a Framework to Detect Multi-stage Advanced Persistent Threats Attacks". In *2014 IEEE 8th International Symposium on Service Oriented System Engineering (SOSE)*, pages 390–395.

[9] P. Bravo and D. Garcia. "Proactive Detection of Kernel-Mode Rootkits". In *2011 Sixth International Conference on Availability, Reliability and Security (ARES)*, pages 515–520.

[10] F. Brown. "SCADA Hacking: Clear and Present Danger". San Diego, CA, Oct. 2014. IT Audit & Controls Conference (ITAC) 2014.

[11] M. Butt. "BIOS integrity an advanced persistent threat". In *2014 Conference on Information Assurance and Cyber Security (CIACS)*, pages 47–50.

[12] L. Constantin. “The ZeroAccess botnet is back in business”. *Computerworld*, Jan. 2015.

[13] A. Dauria. “Battling APTs with the Kill Chain Method | Splunk Blogs”. http://blogs.splunk.com/2014/09/03/battling-apts-with-the-kill-chain-method/, Sept. 2014.

[14] J. De Vries, H. Hoogstraaten, J. van den Berg, and S. Daskapan. “Systems for Detecting Advanced Persistent Threats: A Development Roadmap Using Intelligent Data Analysis”. In *2012 International Conference on Cyber Security (CyberSecurity)*, pages 54–61.

[15] Dropbox Help Center. “How does Dropbox handle viruses or malware?”. *Dropbox*, May 2015.

[16] T. Dube, R. Raines, M. Grimaila, K. Bauer, and S. Rogers. “malware Target Recognition of Unknown Threats”. *IEEE Systems Journal*, 7(3):467–477, Sept. 2013.

[17] S. Dunlap and J. Butts. “PLC Code Protection”. Retrieved from http://www.slideshare.net/dgpeters/plc-code-protection?from=ss_embed. *Scientific SCADA Security Symposium*, Jan. 2014.

[18] D. Eijndhoven. “Mapping Defenses Using the Cyber Kill Chain”. *Norse*, Dec. 2014.

[19] N. Falliere. “Exploring Stuxnets PLC Infection Process”. *Symantec Security Response*, Sept. 2010.

[20] N. Falliere. “Stuxnet Introduces the First Known Rootkit for Industrial Control Systems”. *Symantec Security Response*, June 2010.

[21] FireEye, Inc. “FireEye Advanced Threat Report 2013”. Technical Report SR.ATR.US-EN.082014, 2014.

[22] H. Gao, Q. Li, Y. Zhu, W. Wang, and L. Zhou. “Research on the working mechanism of Bootkit”. In *2012 8th International Conference on Information Science and Digital Content Technology (ICIDT)*, volume 3, pages 476–479.

[23] P. Giura and W. Wang. “A Context-Based Detection Framework for Advanced Persistent Threats”. In *2012 International Conference on Cyber Security (CyberSecurity)*, pages 69–74.

[24] GMER. "GMER - Rootkit Detector and Remover". http://www.gmer.net/, 2015.

[25] D. Goodin. "GPU-based rootkit and keylogger offer superior stealth and computing power". *Ars Technica*, May 2015.

[26] K. J. Higgins. "Air Force Researchers Plant Rootkit In A PLC". *Dark Reading*, Jan. 2014.

[27] F. Howard and U. SophosLabs. "Exploring the Blackhole exploit kit". *Naked Security*, Mar. 2012.

[28] C. Hudel and M. Shehab. "Optimizing search for malware by hashing smaller amounts of data". In *2013 World Congress on Internet Security (WorldCIS)*, pages 112–117.

[29] E. M. Hutchins, M. J. Cloppert, and R. M. Amin. "Intelligence-Driven Computer Network Defense Informed by Analysis of Adversary Campaigns and Intrusion Kill Chains". Retrieved from http://www.lockheedmartin.com/content/dam/lockheed/data/corporate/documents/LM-White-Paper-Intel-Driven-Defense.pdf,
2011.

[30] ICS-Cert. "USB Malware Targeting Siemens Control Software (Update C)". *ICS-CERT*, Aug. 2010.

[31] ICS-CERT. "Ongoing Sophisticated Malware Campaign Compromising ICS (Update B)". *ICS-CERT*, Dec. 2014.

[32] J. Johnson and E. Hogan. "A graph analytic metric for mitigating advanced persistent threat". In *2013 IEEE International Conference on Intelligence and Security Informatics (ISI)*, pages 129–133.

[33] Kaspersky. "TDSSKiller - Rootkit Removal". Retrieved from http://usa.kaspersky.com/downloads/TDSSKiller, 2015.

[34] Kaspersky Labs' Global Research & Analysis Team (GReAT). "The Duqu 2.0 persistence module". *SecureList*, June 2015.

[35] M. Kelly. “Information Security Awareness - Social Engineering”. http://informationsecurityhq.com/information-security-awareness-social-engineering/, Feb. 2011.

[36] N. Kshetri. “Cyberwarfare: Western and Chinese Allegations”. *IT Professional*, 16(1), Jan. 2014.

[37] F. Li, A. Lai, and D. Ddl. “Evidence of Advanced Persistent Threat: A case study of malware for political espionage”. In *2011 6th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 102–109.

[38] Y. Liu. “W32.Tapin Technical Details | Symantec”. *Symantec.com*, Sept. 2011.

[39] D. Lobo, P. Watters, X.-W. Wu, and L. Sun. “Windows Rootkits: Attacks and Countermeasures”. In *Cybercrime and Trustworthy Computing Workshop (CTC), 2010 Second*, pages 69–78, July 2010.

[40] Lockheed Martin. “Cyber Kill Chain®”. Retrievd from http://www.lockheedmartin.com/us/what-we-do/information-technology/cyber-security/cyber-kill-chain.html, 2015.,
2015.

[41] Malwarebytes. “Malwarebytes | Anti-Rootkit BETA”. Retrieved from https://www.malwarebytes.org/antirootkit/, 2015., 2015.

[42] Mandiant. “APT1: Exposing One of Chinas Cyber Espionage Units”. Technical Report Mandiant APT1, 2013.

[43] McAfee. “Virus Profile: ZeroAccess!9AC50C5125DE”. Retrieved from http://home.mcafee.com/virusinfo/virusprofile.aspx?key=742144#none, 2015., Dec. 2011.

[44] McAfee. “McAfee Labs Threat Advisory: ZeroAccess Rootkit”. Retrieved from https://kc.mcafee.com/resources/sites/MCAFEE/content/live/PRODUCT_DOCUMENTATION/23000/PD23412/en_US/McAfee2013., Aug. 2013.

[45] J. Melvin. “Discovery Of A Rootkit: A simple scan leads to a complex solution”. Retrieved from http://digital-forensics.sans.org/community/papers/gcfa/discovery-

rootkit-simple-scan-leads-complex-solution_244,
2005.

[46] Metasploit. "Penetration Testing Software, Pen Testing Security". *Metasploit*, Mar. 2015.

[47] Microsoft Windows Dev Center. "User mode and kernel mode (Windows Drivers)". https://msdn.microsoft.com/en-us/library/windows/hardware/ff554836(v=vs.85).aspx, 2015.

[48] MITRE. CVE-2015-2284. Retrieved from http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-2284, 2015., Mar. 2015.

[49] A. Mohamed. "Phishing and Social Engineering Techniques 3.0 - InfoSec Institute". http://resources.infosecinstitute.com/phishing-and-social-engineering-techniques-3-0/, 2015.

[50] L. Murchu. "Stuxnet Using Three Additional Zero-Day Vulnerabilities". *Symantec Security Response*, Sept. 2010.

[51] S. Musavi and M. Kharrazi. "Back to Static Analysis for Kernel-Level Rootkit Detection". *IEEE Transactions on Information Forensics and Security*, 9(9):1465–1476, Sept. 2014.

[52] T. Mustafa. "Malicious Data Leak Prevention and Purposeful Evasion Attacks: An approach to Advanced Persistent Threat (APT) management". In *Electronics, Communications and Photonics Conference (SIECPC), 2013 Saudi International*, pages 1–5.

[53] National Institute of Standards and Technology (NIST). "NIST Special Publication 800-39: Managing Information Security Risk". Retrieved from http://csrc.nist.gov/publications/nistpubs/800-39/SP800-39-final.pdf, 2015., Mar. 2011.

[54] D. Newman. "Review: FireEye fights off multi-stage malware". *Network World*, May 2014.

[55] Nmap. "Nmap: the Network Mapper - Free Security Scanner". Retrieved from http://nmap.org/, 2015., 2015.

[56] NSIS. "NSIS Self-Extractor Kit". Retrieved from http://nsis.sourceforge.net/NSIS_Self-Extractor_kit, 2005., June 2005.

[57] Penetration Testing Execution Standard. "Intelligence Gathering - The Penetration Testing Execution Standard". Retrieved from http://www.pentest-standard.org/index.php/Intelligence_Gathering#Level_1_Information_Gathering, 2014., Oct. 2014.

[58] A. Prakash, E. Venkataramani, H. Yin, and Z. Lin. "Manipulating semantic values in kernel data structures: Attack assessments and implications". In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–12.

[59] M. Prokhorenko. "Thefts in Remote Banking Systems: Incident Investigations". *SecureList*, Sept. 2014.

[60] A. Raff. "Solving the Cyber Kill Chain Paradox: Going from Reactive to Proactive". http://www.securityweek.com/solving-cyber-kill-chain-paradox-going-reactive-proactive, June 2014.

[61] Red Hat. "Loading a Customized Module - Persistent Changes". Retrieved from https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Deployment_Guide/sec-Loading_a_Customized_Module-Persistent_Changes.html, 2015., May 2015.

[62] J. Rhee, R. Riley, Z. Lin, X. Jiang, and D. Xu. "Data-Centric OS Kernel Malware Characterization". *IEEE Transactions on Information Forensics and Security*, 9(1):72–87, Jan. 2014.

[63] C. Ries. Inside windows rootkits. *VigilantMinds Inc*, 4736:27, 2006.

[64] Rootkit Analytics. "Hidden Registry Detection". *RootkitAnalytics.com*, 2011.

[65] P. J. Salzman, M. Burian, and O. Pomerantz. "The Linux Kernel Module Programming Guide". http://www.tldp.org/LDP/lkmpg/2.6/html/, May 2007.

[66] M. Shankarapani, K. Kancherla, S. Ramammoorthy, R. Movva, and S. Mukkamala. "Kernel machines for malware classification and similarity analysis". In *The 2010 International Joint Conference on Neural Networks (IJCNN)*, pages 1–6.

[67] P. Sharma, A. Joshi, and T. Finin. "Detecting data exfiltration by integrating information across layers". In *2013 IEEE 14th International Conference on Information Reuse and Integration (IRI)*, pages 309–316.

[68] Shodan. "SHODAN - Computer Search Engine". Retrieved from http://www.shodanhq.com/, 2015., 2015.

[69] A. Shosha, C. Liu, P. Gladyshev, and M. Matten. "Evasion-resistant malware signature based on profiling kernel data structure objects". In *2012 7th International Conference on Risk and Security of Internet and Systems (CRiSIS)*, pages 1–8.

[70] J. Sigholm and M. Bang. "Towards Offensive Cyber Counterintelligence: Adopting a Target-Centric View on Advanced Persistent Threats". In *Intelligence and Security Informatics Conference (EISIC), 2013 European*, pages 166–171.

[71] C. B. Simmons, S. G. Shiva, H. Bedi, and D. Dasgupta. "AVOIDIT: A Cyber Attack Taxonomy". 9th ANNUAL SYMPOSIUM ON INFORMATION ASSURANCE (ASIA14), JUNE 3-4, 2014, ALBANY, NY.

[72] A. Sood and R. Enbody. "Targeted Cyberattacks: A Superset of Advanced Persistent Threats". 11(1):54–61.

[73] S. Sparks, S. Embleton, and C. Zou. *"Chapter 19: Windows Rootkits - A Game of Hide and Seek"*. World Scientific, 1 edition, 2011.

[74] D. Stanley, D. Xu, and E. Spafford. "Improved kernel security through memory layout randomization". In *Performance Computing and Communications Conference (IPCCC), 2013 IEEE 32nd International*, pages 1–10.

[75] Symantec. "Advanced Persistent Threats: How They Work". Retrieved from http://www.symantec.com/theme.jsp?themeid=apt-infographic-1, 2015., 2015.

[76] Tigzy. "KernelMode rootkits: Part 1, SSDT hooks". *Adlice Software*, June 2014.

[77] V. Vasisht and H.-H. Lee. "SHARK: Architectural support for autonomic protection against stealth by rootkit exploits". In *2008 41st IEEE/ACM International Symposium on Microarchitecture, 2008. MICRO-41*, pages 106–116.

[78] Verizon. "2014 Data Breach Investigations Report". Retrieved from http://www.verizonenterprise.com/DBIR/2014/reports/rp_Verizon-DBIR-2014_en_xg.pdf,
2014.

[79] N. Virvilis, D. Gritzalis, and T. Apostolopoulos. "Trusted Computing vs. Advanced Persistent Threats: Can a Defender Win This Game?". In *Ubiquitous Intelligence and Computing, 2013 IEEE 10th International Conference on and 10th International Conference on Autonomic and Trusted Computing (UIC/ATC)*, pages 396–403.

[80] N. Virvilis, O. Serrano, and B. Vanautgaerden. "Changing the game: The art of deceiving sophisticated attackers". In *Cyber Conflict (CyCon 2014), 2014 6th International Conference On*, pages 87–97.

[81] S. Vogl, T. Kittel, and C. Eckert. "Persistent Data-only Malware: Function Hooks without Code". San Diego, CA, Feb. 2014. Internet Society.

[82] L. Wang and P. Dasgupta. "Kernel and Application Integrity Assurance: Ensuring Freedom from Rootkits and Malware in a Computer System". In *21st International Conference on Advanced Information Networking and Applications Workshops, 2007, AINAW '07*, volume 1, pages 583–589.

[83] Y.-M. Wang, D. Beck, B. Vo, R. Roussev, and C. Verbowski. "Detecting stealth software with Strider GhostBuster". In *International Conference on Dependable Systems and Networks, 2005. DSN 2005. Proceedings*, pages 368–377, June 2005.

[84] Z. Wang, X. Jiang, W. Cui, and X. Wang. "Countering Persistent Kernel Rootkits Through Systematic Hook Discovery". Technical report, 2008.

[85] Websense. "Websense 2013 Threat Report". Retrieved from http://www.websense.com/assets/reports/websense-2013-threat-report.pdf, 2013., 2013.

[86] F. Wecherowski. "A Real SMM Rootkit: Reversing and Hooking BIOS SMI Handlers". *Phrack Magazine*, 2009.

[87] Windows Hardware Dev Center. "Windows Driver Frameworks". Retrieved from https://msdn.microsoft.com/en-us/library/windows/hardware/ff557565(v=vs.85).aspx, 2015., 2015.

[88] J. Wyke. "Over 9 million PCs infected - ZeroAccess botnet uncovered". *Naked Security*, Sept. 2012.

[89] J. Wyke. "Notorious "Gameover" malware gets itself a kernel-mode rootkit". *Naked Security*, Feb. 2014.

[90] J. Wyke and SophosLabs, UK. "The ZeroAccess rootkit". *Naked Security*, 2012.

[91] B. Zhu, A. Joseph, and S. Sastry. "A Taxonomy of Cyber Attacks on SCADA Systems". In *Internet of Things (iThings/CPSCom), 2011 International Conference on and 4th International Conference on Cyber, Physical and Social Computing*, pages 380–388.

Appendix A
Acronyms

Acronym	Definition
ACL	Access Control List
ADS	Alternate Data Streams
AFIT	Air Force Institute of Technology
API	Application Program Interface
APT	Advanced Persistent Threat
APT1	PLA Unit 61398
ASEP	Auto Start Extensibility Point
BEP	Browser Exploit Pack
C2, C&C, CnC	Command and Control
DDoS	Distributed Denial-of-Service
DGA	Domain Generation Algorithm
DKOM	Direct Kernel Object Manipulation
DLL	Dynamically Linked Library
DMA	Direct Memory Access
DoS	Denial of Service
ELF	Executable and Linking Format
GAC	Global Assembly Cache
GOT	Global Offset Table
HMI	Human-Machine Interfaces
HPID	Hardware Assisted PID
HTRAN	HUC Packet Transmit Tool
IAT	Import Address Table
ICS	Industrial Control Systems
ICS-CERT	Industrial Control Systems Cyber Emergency Response Team
IDS	Intrusion Detection System
IDT	Interrupt Descriptor Table
IP	Intellectual Property
IPC	Inter–process Communication
LKM	Loadable Kernel Module
LM–CIRT	Lockheed Martin Computer Incident Response Team
LSP	Layered Service Provider
MIF	Malware Infection Framework
MSF	Metasploit Framework
OSINT	Open Source Intelligence
P2P	Peer-to-Peer

PLA	People's Liberation Army
PLC	Programmable Logic Controllers
PnP	Plug-n-Play
RAT	Remote Administration Tool
ROP	Return–Oriented Programming
SCADA	Supervisory Control and Data Acquisition
SCI	System Call Interface
SET	Social Engineer Toolkit
SHARK	Secure Hardware support Against RootKit
SIEM	Security Information and Event Management
SMI	System Management Interrupt
SMM	System Management Mode
SSDT	System Service Descriptor Table
SVM	Semantic Value Manipulation
UAC	User Account Control
VFS	Virtual File System
VM	Virtual Machine
VMI	Virtual Machine Introspection
VMM	Virtual Machine Monitor
VNS	Virtual Network Services

Appendix B
Compilation of Taxonomies

Rootkit Kill Chain Taxonomies

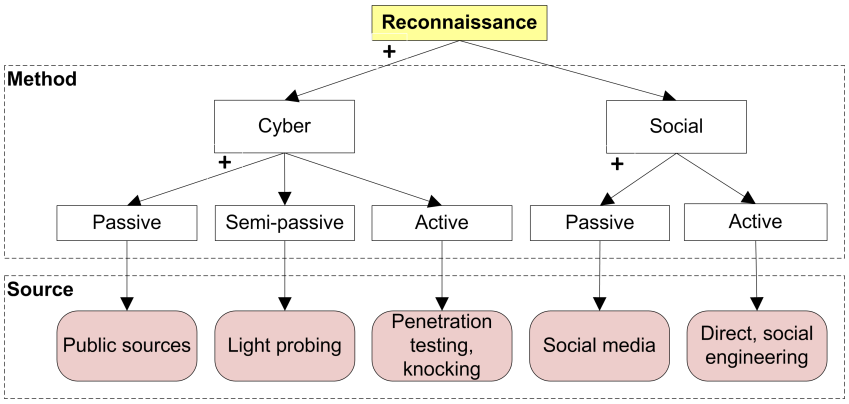

Figure 3.3: Taxonomy of Reconnaissance Methods

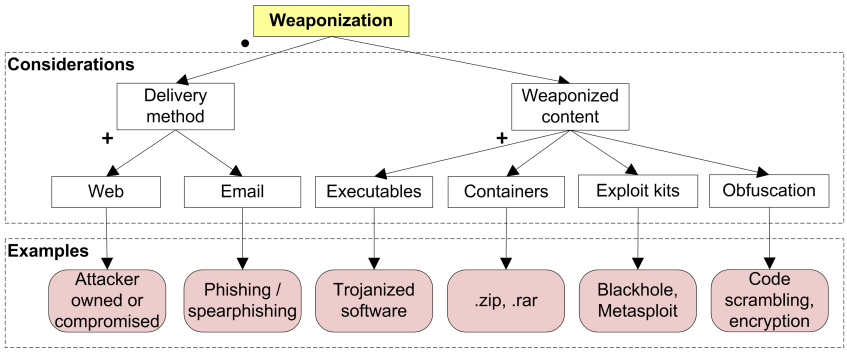

Figure 3.6: Taxonomy of Weaponization Methods

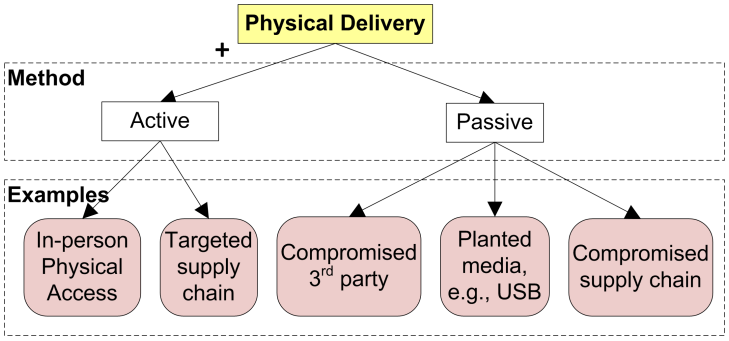

Figure 3.7: Taxonomy of Physical Delivery Methods

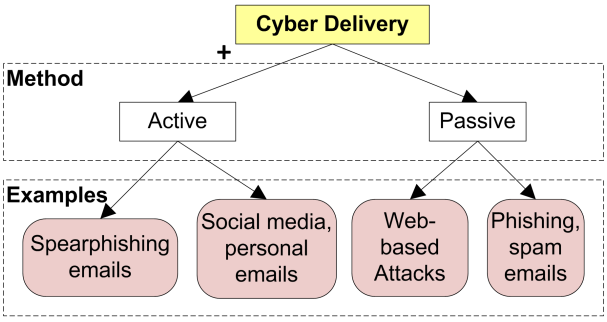

Figure 3.8: Taxonomy of Cyber Delivery Methods

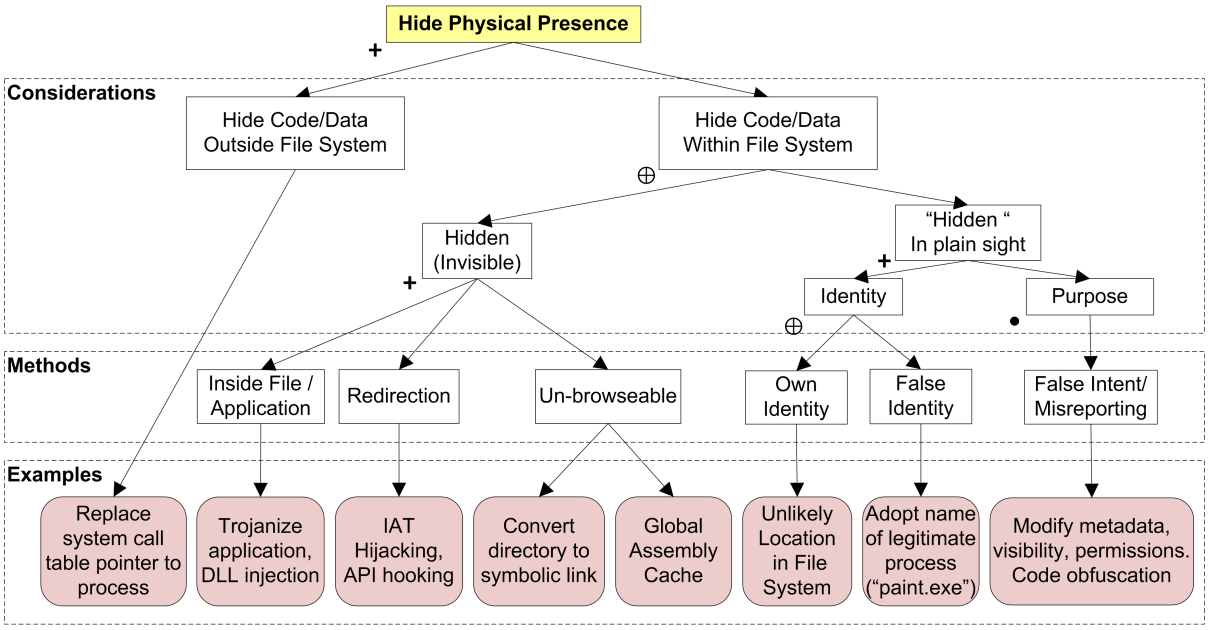

Figure 4.5: Concealment Methods for Physical Existence

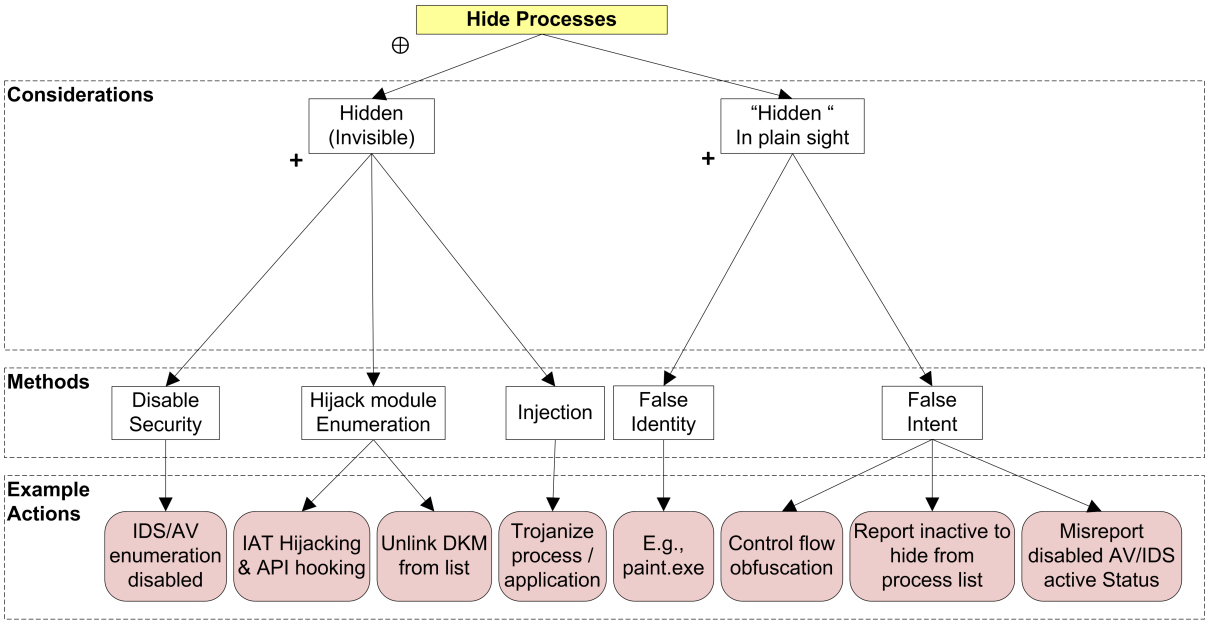

Figure 4.6: Concealment Methods for Process Evidence

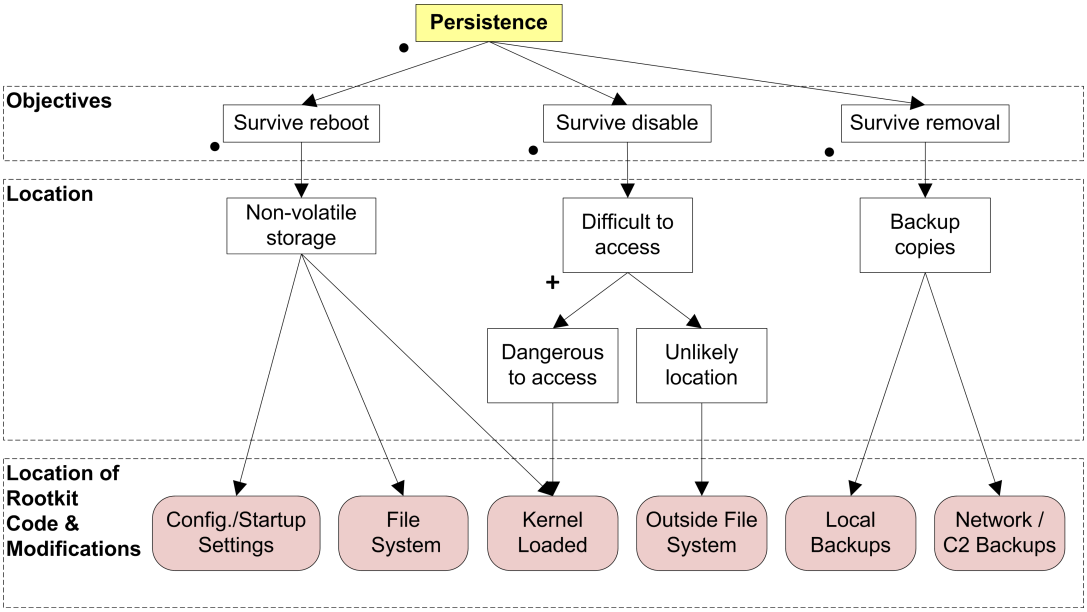

Figure 4.4: Rootkit Persistence

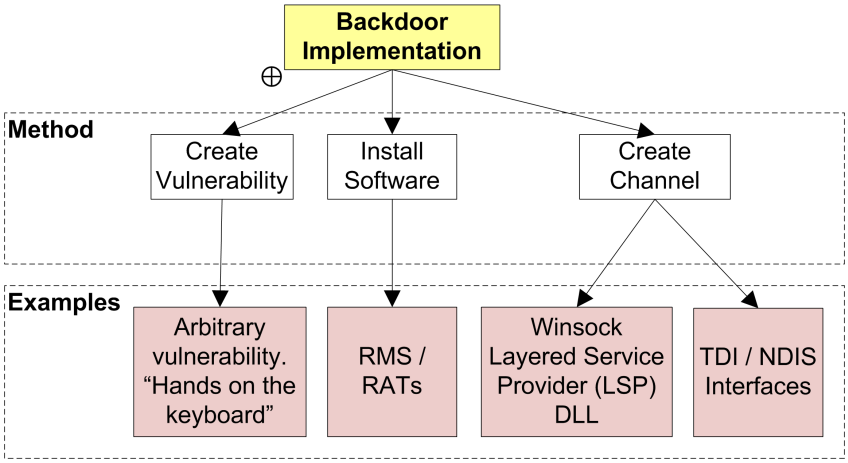

Figure 4.7: Backdoor Implementation Methods

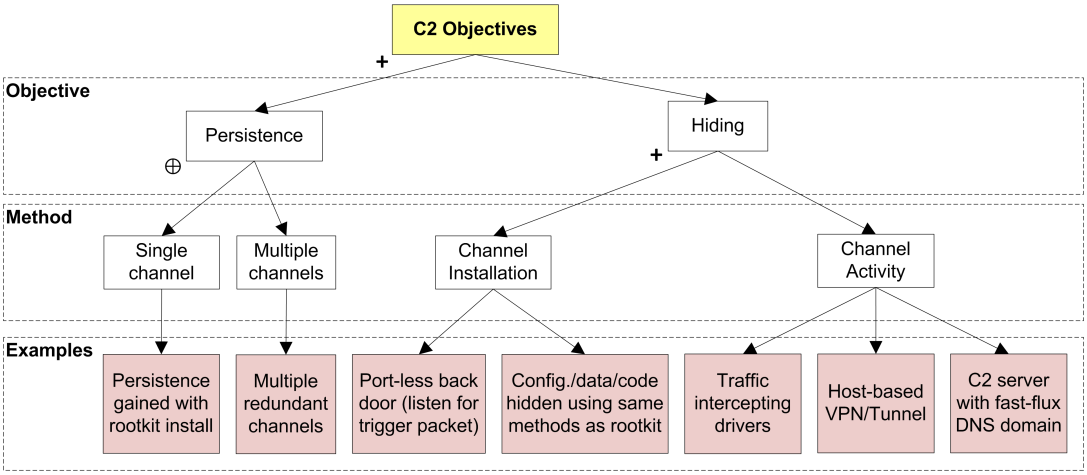

Figure 4.8: C2 Objectives: Persistence and Hiding

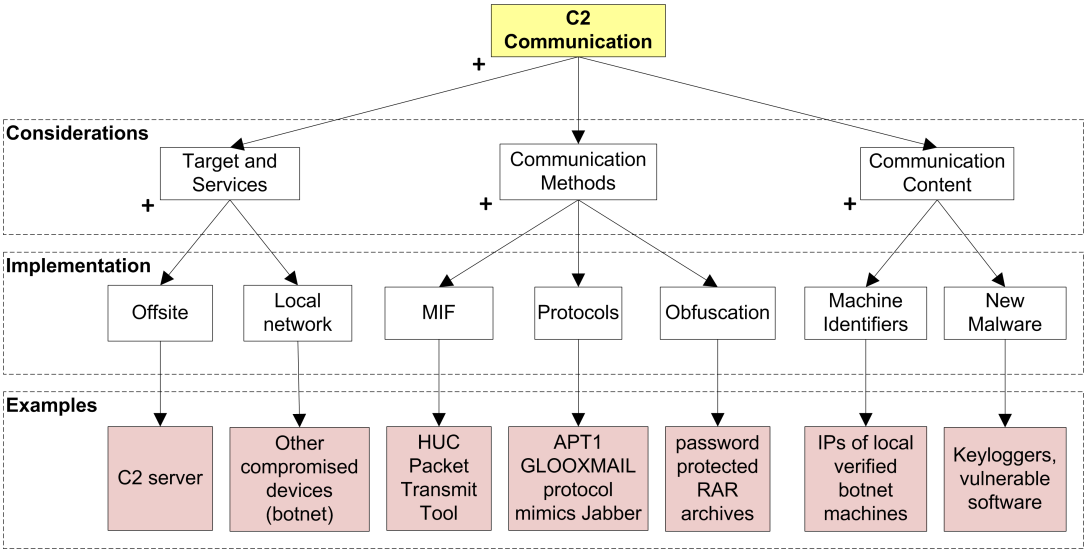

Figure 4.9: C2 Communication Considerations

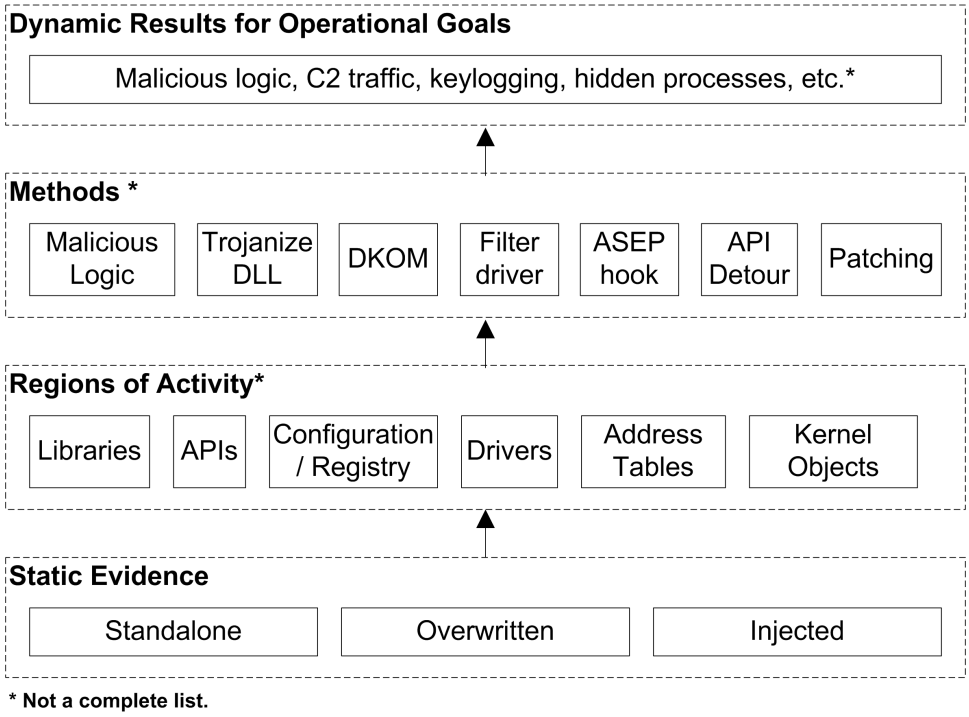

Figure 4.10: Layers of Rootkit Installation Indicators

Rootkit Detection Taxonomies

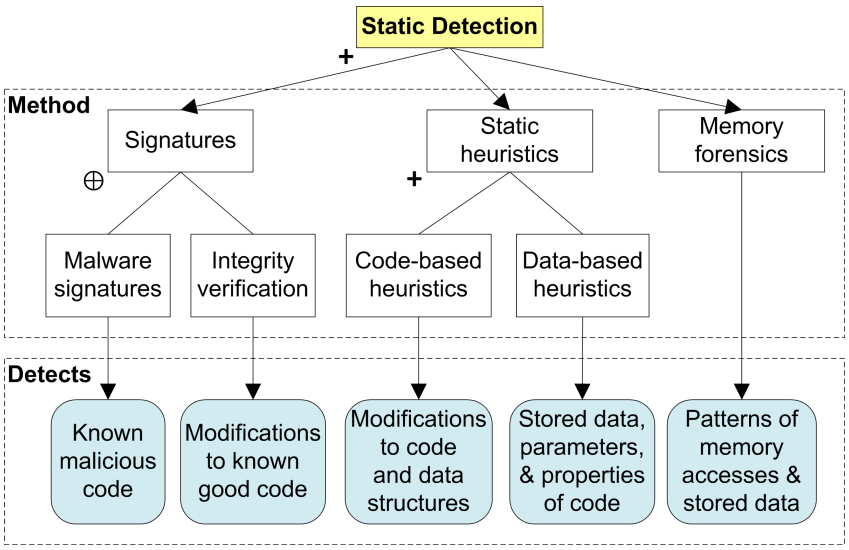

Figure 5.1: Static Rootkit Detection Taxonomy

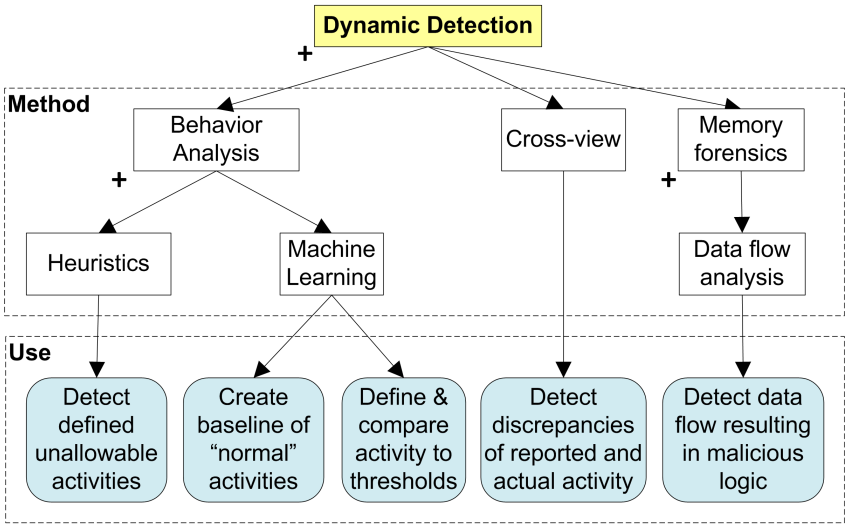

Figure 5.2: Dynamic Rootkit Detection Taxonomy

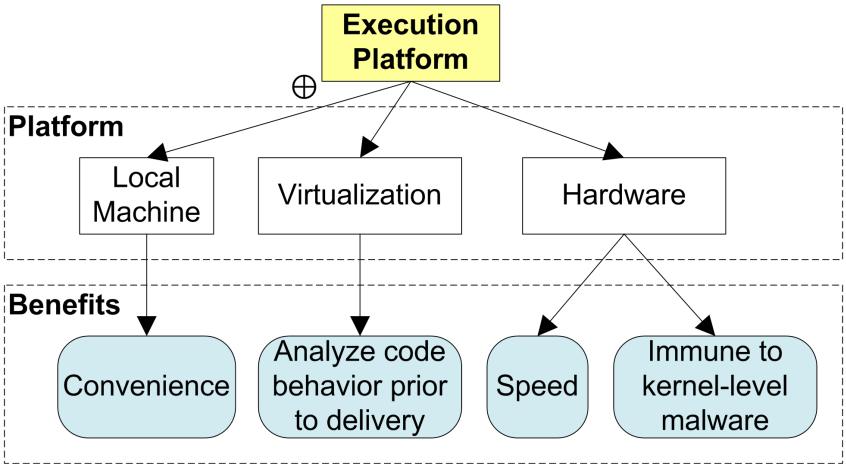

Figure 5.3: Rootkit Detection Execution Platform Options

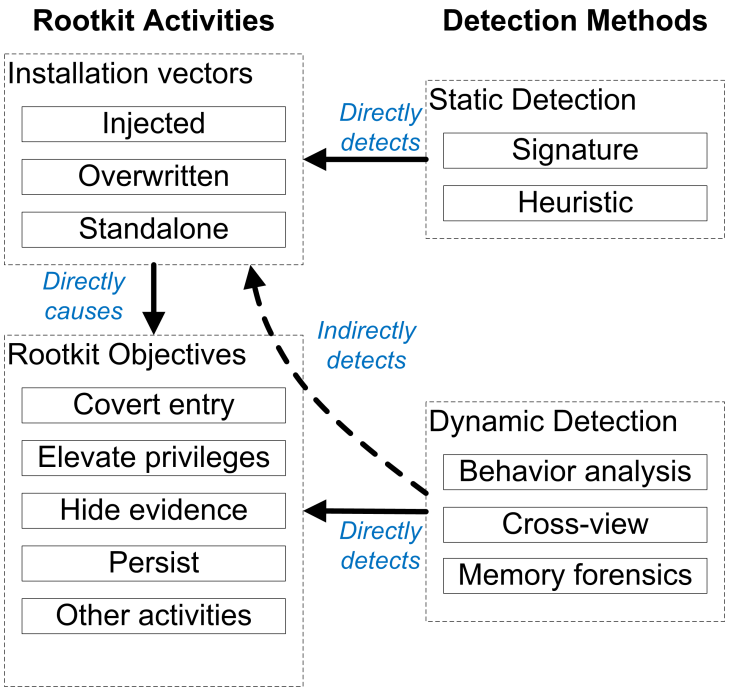

Figure 5.4: Detection Methods Applied to Rootkit Indicators

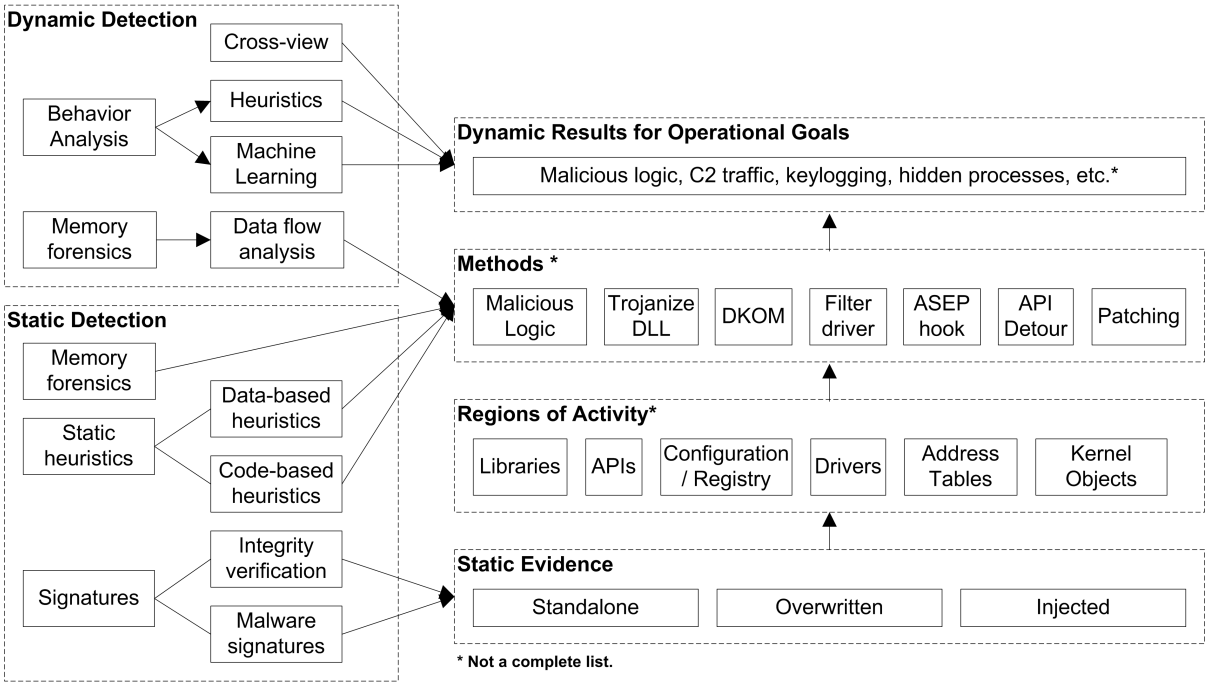

Figure 5.5: Application of Detection Methods to Rootkit Indicators