Quis custodiet ipsos custodes?

Wer wird die Wächter selbst bewachen?

Benedikt Peter
Betreuer: Holger Kinkelin
Hauptseminar - Innovative Internettechnologien und Mobilkommunikation SS2013
Lehrstuhl Netzarchitekturen und Netzdienste
Fakultät für Informatik, Technische Universität München
Email: peterb@in.tum.de

KURZFASSUNG

Rootkits und ihre Fähigkeit, sich selbst und andere Malware zu verbergen, stellen ein wichtiges Hilfsmittel für Angreifer dar und bedeuten somit ein großes Risiko für Administratoren, die ihre Systeme frei von Schadsoftware halten müssen. Heutige *Host-based Intrusion Detection Systeme* (HIDS) bieten in der Regel die Möglichkeit, das Auftreten bestimmter Rootkits zu erkennen, können jedoch durch entsprechend ausgerüstete Rootkits getäuscht werden. Diese Arbeit gibt anhand des Linux-Betriebssystems eine kurze Einführung in die Art und Weise, wie Rootkits arbeiten, um im Anschluss auf die Funktionsweise von HIDSen und im Besonderen auf deren Rootkit-Erkennung einzugehen. Anschließend werden die Grenzen von klassischen HIDSen aufgezeigt und Methoden vorgestellt, die mit Hilfe von *Virtual Machine Introspection* versuchen, diese Grenzen zu überwinden. Es zeigt sich jedoch als Ergebnis, dass auch durch eine solche erweiterte *Intrusion Detection* nicht jedes Rootkit erkannt werden kann.

Schlüsselworte

Security, Malware, Rootkits, Kernel, Intrusion Detection, OSSEC, Samhain, Virtual Machine Introspection, Blue Pill

1. EINLEITUNG

Während die Problematik von Schadsoftware weitgehend mit Windows-Systemen assoziiert wird, sind sehr wohl auch andere Betriebssysteme wie Mac OS X und Linux betroffen. Da Linux besonders im Server-Bereich sehr weit verbreitet ist und durch die Kompromittierung eines solchen Systems potentiell mehr Schaden angerichtet werden könnte als mit einem herkömmlichen PC, den nur wenige Personen benutzen, besteht für Angreifer eine große Motivation, Linux-Systeme zu übernehmen.

Ein wichtiges Hilfsmittel für Angreifer stellen Rootkits dar. Diese verbergen den Angriff vor den Besitzern eines Systems, sodass diese keinen Verdacht schöpfen. Rootkits existieren für verschiedene Betriebssysteme und werden immer wieder in freier Wildbahn entdeckt. Ein Beispiel stellt das **sshd**-Rootkit dar, das im Frühjahr 2013 auf einer Reihe von Linux-Servern entdeckt wurde und dort die eigentliche Funktion der Malware, Login-Informationen zu stehlen, verbarg (siehe hierzu auch [19]).

Um derlei Angriffe und bereits erfolgte Infektionen zu erkennen und nicht auf manuelle, zeitraubende Kontrollen angewiesen zu sein, existieren Programme, die als eine Art Wächter fungieren. Diese informieren den Administrator über ungewöhnliche Vorkommnisse und stellen so sicher, dass Angriffe nicht unbemerkt vonstatten gehen. Dies wird als *Intrusion Detection* bezeichnet. Doch um umfassenden Schutz zu gewährleisten, muss sichergestellt sein, dass ein solcher Wächter Rootkits zuverlässig erkennt.

Diese Arbeit stellt die Methoden, die Rootkits und *Intrusion Detection Systems* verwenden, um ihre jeweilige Aufgabe zu erfüllen, gegenüber. Dazu werden im folgenden Abschnitt 2 zunächst die Grundlagen von Rootkits und Malware erläutert. In Abschnitt 3 werden klassische *Host-based Intrusion Detection* Systeme anhand zweier Beispiele eingeführt, während in Abschnitt 4 der Schritt hin zu *Intrusion Detection* gemacht wird, die auf Virtualisierungstechniken zurückgreift. Zudem werden auch die Grenzen dieses Ansatzes aufgezeigt. Abschnitt 5 gibt einen Überblick über die verwendeten Ansätze und Erkenntnisse anderer Paper zu diesem Thema. Abschnitt 6 schließlich fasst die Erkenntnisse dieser Arbeit zusammen und gibt einen kurzen Ausblick.

2. GRUNDLAGEN VON ROOTKITS

2.1 Definition eines Rootkits

Peláez definiert ein Rootkit in [11] zusammenfassend als „a tool or set of tools used by an intruder to hide itself masking the fact that the system has been compromised and to keep or reobtain administrator-level (privileged) access inside a system.“ Ein Rootkit ist damit zumeist Teil einer Strategie eines Angreifers oder einer Malware und erlaubt es dieser, ein System unbemerkt auszuspähen, zu manipulieren oder zu stören.

Um dies zu ermöglichen, ist es Aufgabe und Ziel eines Rootkits, Spuren des Angreifers innerhalb des Systems zu verbergen. Dazu gehören auch die der Malware zugehörigen Prozesse, Dateien, geladene Kernel-Module, offene Ports, Log-Einträge oder ähnliche Anomalien, die bei einer Analyse des Systems durch den Administrator ansonsten auffallen würden. Rootkits lassen sich einteilen in User-Mode- und Kernel-Mode-Rootkits, auf die in den folgenden beiden Abschnitten genauer eingegangen wird.

2.2 User-Mode-Rootkits

Bei einem User-Mode-Rootkit handelt es sich um ein Tool, das im User-Mode, also wie eine gewöhnliche Anwendung,

doi: 10.2313/NET-2013-08-1_10

ausgeführt wird und insbesondere den Kernel selbst nicht direkt beeinflusst (siehe [11]).

Eine klassische Methode, die im vorigen Abschnitt 2.1 genannten Ziele mit Hilfe solcher Rootkits zu verwirklichen, besteht darin, Programmdateien des Betriebssystems durch eigene, modifizierte Versionen zu ersetzen. Hierdurch kann das Verhalten von anderen Programmen, die diese Systemdateien bzw. -programme verwenden, im Sinne des Angreifers beeinflusst werden (vgl. auch [11]).

Beispielsweise könnte das Unix-Programm **ps**, das Informationen über die laufenden Prozesse des Systems ausgeben kann, mit einer eigenen Version überschrieben werden, die sich nahezu identisch zur Originalversion verhält, aber Prozesse mit einem bestimmten Namen (z.B. den Malware-Prozess selbst) nicht anzeigt. Für Programme oder Benutzer, die sich ausschließlich auf die Ausgabe dieses Programms verlassen, sind nun die verborgenen Prozesse nicht mehr auffindbar.

Genauso könnte etwa der System-Log-Daemon derart modifiziert werden, dass er verdächtige Aktivitäten, die ansonsten einen Log-Eintrag generieren würden, ignoriert.

2.3 Kernel-Mode-Rootkits

Bei einem Kernel-Mode-Rootkit handelt es sich um ein Tool, das Code im Kernel-Mode, also innerhalb des Betriebssystem-Kerns, ausführt oder dort Veränderungen an Datenstrukturen durchführt, um seine Ziele zu umzusetzen.

Es existieren verschiedene Möglichkeiten, um dies zu erreichen. Viele Betriebssysteme bieten die Möglichkeit, Code in Form von Treibern oder Modulen dynamisch in den Kernel zu laden und dort auszuführen. Alternativ, etwa wenn das Laden solcher Module nicht möglich ist, kann stattdessen der Code des Kernels direkt manipuliert werden. Dies wiederum kann auf flüchtige Art und Weise im Arbeitsspeicher oder permanent auf der Festplatte geschehen (vgl. [17]). Um auf den Kernel-Speicher zugreifen oder Module bzw. Treiber laden zu können, muss das Rootkit in der Regel zunächst Administratorprivilegien erhalten.

Besitzt das Rootkit Zugriff auf den Kernel, kann es diesen nutzen, um das Verhalten des Kernels auf gewünschte Art und Weise zu verändern.

Eine eher statische Art der Veränderung stellt das Manipulieren von Kernel-Datenstrukturen dar. Der Kernel verwaltet, ähnlich wie normale User-Mode-Anwendungen, Informationen in Datenstrukturen wie Bäumen oder Listen im Arbeitsspeicher, darunter etwa Listen mit laufenden Prozessen, geöffneten Datein, etc. Entfernt oder ändert das Rootkit Einträge in einer solchen Datenstruktur, beeinflusst dies das Verhalten von Kernel-Funktionen (*System Calls*). Wird beispielsweise ein Prozess aus der internen Prozessliste des Kernels entfernt, verschwindet der Prozess auch aus der Sicht von User-Mode-Anwendungen wie z.B. dem im vorigen Abschnitt 2.2 erwähnten **ps** (siehe auch [11]).

Eine weitere Art der Manipulation stellt das *Hooking* dar. Dabei werden Kernel-interne Funktions-Pointer auf Funktionen des Angreifer umgebogen, sodass diese stattdessen ausgeführt werden. Beliebtes Ziel solcher Hooks sind *System Calls*, also die Funktionen des Kernels, die von User-Mode-Anwendungen aus aufgerufen werden können (vgl. [11]).

Unter Linux sind die Funktions-Pointer auf die verfügbaren *System Calls* in einem Array, der *System Call Table*, gespeichert. Veranlasst nun eine Anwendung einen *System Call*, indem sie etwa auf einem x86-System den Interrupt *0x80* auslöst, wechselt der Prozessor vom User- in den Kernel-Mode und ruft den *Interrupt Handler* auf. Dieser ist für die Verarbeitung von Interrupts (Hardware- bzw. Prozessor-Ereignissen) zuständig und ruft in diesem Fall die Hilfsfunktion *system_call()* auf, die schließlich den gewünschten Funktions-Pointer aus dem Array ausliest und die gewünschte Funktion ausführt (vgl. [11]). Dieser Vorgang ist auch in Abbildung 1 zu sehen.

Um eine bestimmte *System-Call*-Funktion durch eigenen Code zu ersetzen, reicht es dem Angreifer, den Eintrag in der *System Call Table* mit einer eigenen Funktionsadresse zu überschreiben (siehe Abbildung 1 in der Mitte). Er könnte jedoch auch die Funktion *system_call()* (auf der linken Seite) oder den eigentlichen *System Call* (*sys_close()* auf der rechten Seite) manipulieren. Um diesen Eingriff möglichst unbemerkt vonstatten gehen zu lassen, kann der Angreifer die ursprüngliche Adresse vor dem Überschreiben sichern. Auf diese Art und Weise kann er den Original-Code in seiner eigenen Funktion aufrufen, z.B. wenn sich der modifizierte *System Call* in bestimmten Situationen exakt wie der Kernel verhalten soll (siehe [11]).

Neben *System Calls*, mit denen sich z.B. Prozesse und Dateien verstecken lassen, können auch andere Funktionen des Kernels, etwa der *Interrupt Handler* selbst, der Netzwerk-Stack (zum Verbergen von Ports) oder unter Linux das VFS[1] (ebenfalls für Dateien) Ziel eines Rootkits sein (vgl. [11]).

Da der Kernel selbst mächtiger ist als eine User-Mode-Anwendung, da er jederzeit auf den gesamten physischen Speicher Zugriff besitzt, erscheint es erstrebenswert, Rootkits ausschließlich im Kernel-Mode zu betreiben. Da User-Mode-Rootkits jedoch leichter zu entwickeln und zu warten sind und sich zudem auch besser an verschiedene Zielsysteme anpassen lassen, setzen nicht alle Rootkits auf den Kernel als Ziel. Welche Funktionen genau von einem Rootkit manipuliert werden müssen und auf welche Art das Rootkit dies erreicht, ist letztlich abhängig von den Zielen, die das Rootkit und damit der Angreifer verfolgt.

3. HOST-BASED INTRUSION DETECTION UND ROOTKITS

Die Aufgabe von *Intrusion-Detection*-Systemen (IDS) ist es, einen Angriff, etwa durch eine Malware, zu erkennen, sodass Gegenmaßnahmen ergriffen werden können. Unter einem *Host-based Intrusion Detection System* (HIDS) wiederum versteht man einen Wächter, der lokal auf der zu überwachenden Maschine läuft und den Zustand dieser Maschine überwacht (siehe [7]). Der restliche Abschnitt beschäftigt

[1]Das *Virtual File System* (VFS) des Linux-Kernels stellt eine Schicht für Dateisystemzugriffe dar, die die tatsächliche Implementierung von Dateioperationen abstrahiert (vgl. [6]).

doi: 10.2313/NET-2013-08-1_10

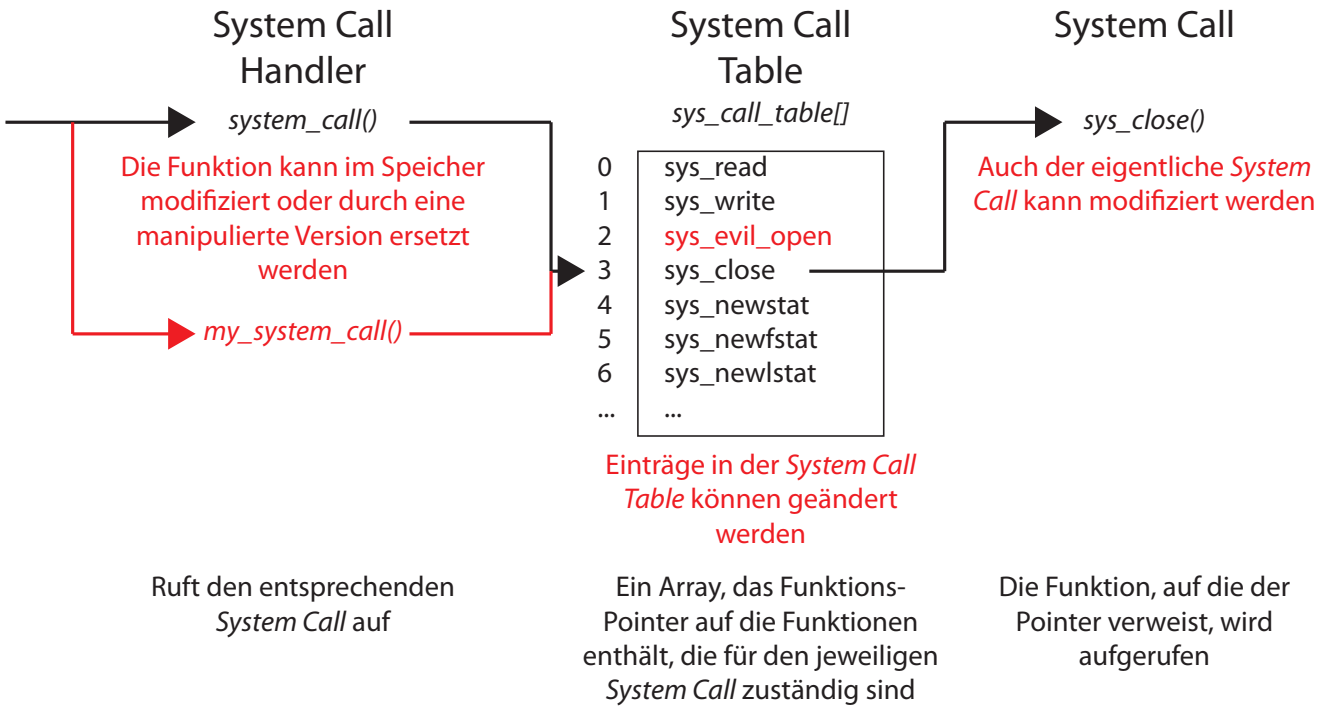

Abbildung 1: Die Abbildung zeigt den typischen Verlauf bei dem Aufruf eines *System Calls*. Die Stellen, an denen ein Kernel-Mode-Rootkit durch *Hooking* das Verhalten des Kernels manipulieren kann, sind rot eingezeichnet.

sich mit solchen HIDSen.

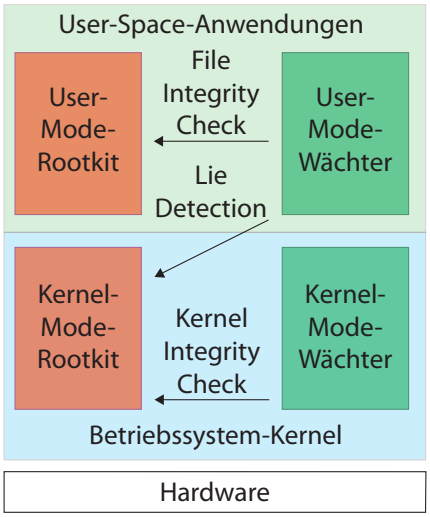

Abbildung 2: Die Abbildung gibt einen Überblick darüber, auf welche Art und Weise Rootkits in einer herkömmlichen Umgebung aufgespürt werden können.

Da Malware sich mit Hilfe eines Rootkits vor der Entdeckung durch ein solches System verstecken könnte, müssen HIDSe wiederum über die Möglichkeit verfügen, entspechende Anomalien zu erkennen und auf die Existenz verborgener Aktivitäten zu schließen. Ähnlich wie bei Rootkits selbst kann man auch bei HIDSen zwischen den Kategorien Kernel-Mode- und User-Mode-HIDS unterscheiden. Ein HIDS könnte unter Umständen einen genaueren Einblick in das Innenleben des Kernels benötigen, als einem User-Mode-Prozess gewährt werden würde, um Manipulationen am Kernel zu erkennen. Eine Übersicht hierzu ist in Abbildung 2 zu sehen. Auch sonst bestehen Ähnlichkeiten zwischen den von Rootkits und HIDSen verwendeten Techniken (vgl. [16]). Wie die Beispiele in diesem Abschnitt zeigen, ist es jedoch für ein HIDS nicht notwendig, im Kernel-Mode zu laufen, um Kernel-Mode-Rootkits zu erkennen. Aus Gründen der Portabilität[2], sowohl zwischen verschiedenen Kernel-Versionen als auch zwischen unterschiedlichen Betriebssystemen, kann es durchaus Sinn machen, sich allein auf eine User-Mode-Implementierung zu konzentrieren.

Im Folgenden sollen nun Methoden, mit denen HIDSe Rootkits aufspüren können, anhand von zwei Beispielen erläutert werden.

3.1 OSSEC

Bei OSSEC (siehe [2]) handelt es sich um ein klassisches HIDS, das mehrere verschiedene Plattformen, darunter Windows und Linux, unterstützt und unter der GPLv3 zur Verfügung gestellt wird. OSSEC verfügt über flexible Möglichkeiten zum Erkennen von Eindringlingen, u.a. durch das Verifizieren von Datei-Prüfsummen anhand einer Datenbank (*File Integrity Checks*) und durch Analyse der Log-Datei-

[2]Während die Schnittstellen zwischen Kernel und User-Mode-Applikationen wohl definiert sind, können sich die internen Strukturen jederzeit ändern. Sowohl Kernel-Mode--HIDSe als auch -Rootkits müssen also an neue Kernel-Versionen angepasst werden, um weiter zu funktionieren.

doi: 10.2313/NET-2013-08-1_10

en (*Log Analysis*). Die lokal installierten Clients lassen sich zentral überwachen und steuern.

Die Rootkit-Erkennung im Besonderen ist in Form des Moduls Rootcheck (siehe [3]) implementiert, das über die allgemeine Funktionalität hinaus Methoden bietet, sowohl User- als auch Kernel-Mode-Rootkits zu erkennen. Rootcheck betreibt die sogenannte *Lie Detection*: Bestimmte Informationen werden über verschiedene, sowohl vertrauenswürdige als auch nicht-vertrauenswürdige Kanäle abgefragt und die Ergebnisse verglichen. Auf diese Art und Weise können Inkonsistenzen und damit mögliche Hinweise auf Rootkits gefunden werden, ohne explizite Informationen darüber zu besitzen, auf welche Art und Weise das Rootkit genau die Manipulationen erreicht hat.

Rootcheck enthält eine Datenbank mit Dateien, die typischerweise von bestimmter Malware angelegt werden und überprüft, ob diese existieren. Um auch versteckte Dateien aufspüren zu können, erfolgt diese Prüfung mit Hilfe von verschiedenen *System Calls* (u.a. *stat()*, *fopen()* und *opendir()*), deren Ergebnisse verglichen werden.

Die Dateien, die nicht von vornherein von der Datenbank als verdächtig klassifiziert werden, werden auf allgemeine Ungewöhnlichkeiten hin untersucht. Dazu gehören z.B. ungewöhnliche Besitzer- und Rechtekombinationen. Das Rootcheck-Manual [3] bezeichnet etwa Dateien, die dem Root-Benutzer gehören, aber jedem den Schreibzugriff gewähren, als „very dangerous“.

Zudem überprüft Rootcheck unter Linux explizit das Verzeichnis /dev, das Gerätedateien zur Verfügung stellt. Insbesondere spürt es dort normale Dateien auf, die dort nicht hingehören und die dort in der Hoffnung versteckt wurden, ein Administrator würde das /dev-Verzeichnis nicht untersuchen. Auch hier können die Dateirechte der Gerätedateien überprüft werden (siehe [3]).

Um auch versteckte Prozesse aufspüren zu können, geht Rootcheck regelmäßig der Reihe nach alle möglichen *Process Identifiers* (PIDs, unter Linux eine vorzeichenlose 16-Bit-Zahl) durch und überprüft für jede einzelne die Ergebnisse der *System Calls getsid()* und *kill()*, sowie die Ausgabe des Programms **ps**. Ein versteckter Prozess äußert sich z.B. dadurch, dass **ps** für eine bestimmte Zahl keinen Eintrag enthält, der Aufruf von *kill()* mit der entsprechenden PID jedoch keinen Fehler (z.B. „No such process“) verursacht, sondern ausgeführt wird (vgl. [3]).

Auf ähnliche Art und Weise werden auch offene TCP- oder UDP-Ports erkannt. Hierzu wird versucht, über *bind()* jeden einzelnen Port zu belegen. Schlägt dies mit der Meldung fehl, dass der Port bereits in Benutzung sei, zeigt aber **netstat** den Port nicht als benutzt an, so kann dies auf ein Rootkit hindeuten (siehe auch [3]).

Schließlich erkennt Rootcheck laut [3] auch, ob für ein oder mehrere Netzwerk-Interfaces der sogenannte *Promiscuous Mode* aktiviert ist, mit dem ein Sniffer den gesamten Netzwerkverkehr des aktuellen Subnets abhören kann (siehe [11]). Diese Information vergleicht es zusätzlich mit der Ausgabe von **ifconfig**.

OSSEC/Rootcheck beschreitet damit den Weg, auch Kernel-Mode-Rootkits mit Hilfe von Methoden des User-Modes zu erkennen. Hierdurch vermeidet es Probleme wie die *Semantic Gap*, unter der weiter unten besprochene Ansätze zu leiden haben.

3.2 Samhain

Genau wie das oben behandelte OSSEC ist auch Samhain (siehe [4]) ein HIDS für mehrere Plattformen, dessen Hauptfunktionalität die Überprüfung von Dateiintegritäten und Logdateien umfasst. Samhain wird unter der GPLv2 zur Verfügung gestellt.

Auch bei Samhain handelt es sich zunächst um ein User-Mode-Programm. Neben der allgemeinen Funktionalität werden einige Funktionen angeboten, die speziell gegen Kernel-Mode-Rootkits gerichtet sind, während andere Rootkits bereits etwa durch die Integritätsprüfung entdeckt werden könnten.

Zunächst kann Samhain versteckte Prozesse aufspüren, indem wie bei OSSEC systematisch PIDs mit *System Calls* wie *kill()* durchgeprüft werden. Ruft man *kill()* mit der Signalnummer 0 auf, so hat dies keinen negativen Einfluss auf das Programm selbst, wenn ein Programm mit der entsprechenden PID existiert.

Darüber hinaus bietet Samhain dem Benutzer im Gegensatz zu OSSEC die Möglichkeit, Manipulationen am Kernel direkt zu erkennen, ohne den Umweg über die *Lie Detection* zu gehen. Hierfür greift der User-Mode-Prozess von Samhain über die Gerätedatei /dev/kmem auf den Kernel-Bereich des Arbeitsspeichers zu. Während diese Gerätedatei auf älteren Systemen in der Regel dem Administrator zur Verfügung stand, ist dies bei neueren Kernel-Versionen aus Sicherheitsgründen nicht mehr der Fall. Um das Feature zu aktivieren, muss daher entweder der Kernel entsprechend geändert werden, oder es kann auf ein zusätzliches Kernel-Modul, das mit Samhain mitgeliefert wird, zurückgegriffen werden, das genau diese Gerätedatei im /dev-Verzeichnis zur Verfügung stellt. Optional gab es für ältere Kernel-Versionen ein weiteres Modul, das den Samhain-Prozess selbst verstecken konnte, sodass ein Rootkit, das die Fähigkeit besaß, auf aktive Wächter zu reagieren, getäuscht werden konnte (vgl. auch [4]). Diese Funktion zeigt, auch wenn sie inzwischen veraltet ist, in welchem Maße Wächter und Rootkits auf dieselbe Funktionalität zurückgreifen können, um ihrer jeweiligen Aufgaben gerecht zu werden.

Über /dev/kmem kann Samhain in Form eines Blockgerätes auf den virtuellen Kernel-Speicher, also auf die vom Kernel selbst belegten Teile des Arbeitsspeichers zugreifen (siehe [4]). Um mit diesem Speicherbereich etwas anfangen zu können, werden zusätzliche Informationen über die Struktur des Speichers benötigt, die u.a. zum Zeitpunkt des Kompilierens des Kernels bestimmt werden. Dieses Phänomen wird als sog. *Semantic Gap* bezeichnet und tritt insbesondere auch in Abschnitt 4 zu Tage. Hierfür verwendet Samhain die *System.map*-Datei, die genau beim Erstellen des Kernel-Images erstellt wird und die Positionen von Kernel-Symbolen, also von Kernel-Funktionen und globalen Variablen innerhalb des Kernel-Speichers enthält. Samhain liest aus der *System.map* die Positionen der zu überprüfenden

doi: 10.2313/NET-2013-08-1_10

Symbole aus und kann dadurch die entsprechenden Stellen innerhalb von /dev/kmem überprüfen.

Unter den überprüften Kernel-Symbolen befinden sich neben der *Interrupt Descriptor Table*, in der die Handler für die Interrupts stehen, auch die Funktion *system_call()*, die die Ausführung von *System Calls* übernimmt, sowie die eigentliche *System Call Table* (vgl. auch Abschnitt 2.3 und Abbildung 1). Zusätzlich überprüft Samhain die IO-Funktionen, die das /proc-Dateisystem zur Verfügung stellen, das von vielen User-Mode-Programmen genutzt wird, um an Kernel-Informationen zu gelangen (vgl. [4]).

Samhain präsentiert damit einen Hybrid-Ansatz, der auch auf der Kernel-Ebene arbeitet, ohne jedoch selbst Kernel-Code ausführen zu müssen. Damit entfällt die Notwendigkeit, flexibel auf Änderungen innerhalb des Kernels, die auch zwischen kleinen Releases geschehen, reagieren zu müssen. Es ist jedoch nicht auszuschließen, dass Situationen existieren, in denen eine solche Herangehensweise an ihre Grenzen stößt. Als Beispiel sei hier etwa ein Rootkit zu nennen, das innerhalb des Kernels Zugriffe auf /dev/kmem abfängt und Spuren auf etwaige Manipulationen beseitigt. In diesem Fall hätte ein User-Mode-Wächter keine Chance, das Rootkit auf diese Art und Weise zu erkennen. Im Einzelfall muss daher zwischen Kosten und Nutzen abgewogen werden.

4. WER WIRD DIE WÄCHTER SELBST BEWACHEN?

Wie im vorherigen Abschnitt dargelegt wurde, können die in den beiden Abschnitten 3.1 und 3.2 beschriebenen Manipulationen, die Rootkits im System verursachen, sowohl von User-Mode- als auch von Kernel-Mode-Wächtern gefunden werden. Diese Manipulationen beschränken sich jedoch darauf, verdächtige Elemente vor typischen Diagnoseoperationen wie dem Anzeigen der laufenden Prozesse, wie sie u.a. auch die vorgestellten HIDSe vollziehen, zu verstecken.

Sind etwa die Aktionen, die ein bestimmtes HIDS im User-Mode zur *Lie Detection* durchführt und vergleicht, bekannt, kann ein Rootkit gezielt genau diese Funktionen dazu bringen, identische, manipulierte Informationen zu liefern.

Handelt es sich sowohl beim Schadcode als auch beim HIDS um Kernel-Code (oder wie im Beispiel von Samhain um Code, der den Kernel direkt untersucht), spielt es eine entscheidende Rolle, welches der Tools zuerst geladen wird, da der einmal geladene Code im Kernel effektiv verhindern kann, dass nachfolgende Module ebenfalls Änderungen am Kernel durchführen. Unter Linux etwa könnte die Funktion zum Laden von Kernel-Modulen überschrieben werden, sodass das Laden von weiteren Modulen von vornherein abgelehnt wird (vgl. auch [11]). Wird wiederum bösartiger Code im Kernel ausgeführt, ist dieser potentiell allmächtig, da innerhalb des Systems nun keine Beschränkungen mehr existieren. Die Komplexität des Verbergens hängt in diesem Fall nur davon ab, welchen Aufwand der Autor des Rootkits betreiben will, um seine Ziele zu erfüllen. Handelt es sich um einen gezielten Angriff auf ein System, dessen Konfiguration (etwa das verwendete HIDS) bekannt ist, könnte ein Rootkit auch so weit gehen, statt den Kernel das HIDS selbst anzugreifen und zu manipulieren. Um daher auch komplexere Manipulationen am System erkennen zu können, kann es daher nötig sein, die *Intrusion Detection* vom Kernel eine Ebene tiefer zu verlegen.

4.1 VMI-based Intrusion Detection

In den letzten Jahren fand eine starke Verbreitung der Verwendung von Virtualisierungstechniken wie Xen statt. Gerade im Server- und Rechenzentrums-Umfeld bietet Virtualisierung und die dadurch gewonnene Flexibilität und Isolation viele Vorteile.

Virtualisierungssysteme wie Xen führen eine oder mehrere Virtuelle Maschinen (VMs) innerhalb eines physischen Systems aus. Um die Isolation und Ressourcenzuteilung zu gewährleisten, existiert eine zentrale Kontrollinstanz, der sogenannte *Hypervisor* (dieser wird auch als *Virtual Machine Monitor* (VMM) bezeichnet). Der Hypervisor verfügt in der Regel über die Fähigkeit, auf jede VM zuzugreifen und deren Konfiguration zu beeinflussen. Besitzt er weiterhin die Fähigkeit, in einer laufenden VM auf den virtuellen Arbeitsspeicher derselben zuzugreifen, bietet das einen Ansatzpunkt für *Intrusion Detection*. Diese Art des Zugriffs auf VMs wird als *Virtual Machine Introspection* (VMI) bezeichnet (siehe auch Abbildung 3).

Ein Problem stellt dabei jedoch die *Semantic Gap* dar: Der Hypervisor sieht den virtuellen Speicher einer bestimmten VM als zusammenhängenden Datenbereich einer bestimmten Größe, besitzt jedoch zunächst keinerlei Informationen darüber, in welcher Form sich in diesem Bereich die gesuchten Daten befinden. Da der Speicher von einem virtualisierten Kernel des Gast-Betriebssystems verwaltet wird, der Paging oder Swapping betreibt, ist zu erwarten, dass es ein nicht-triviales Problem darstellt, an die gesuchten Daten zu gelangen.

Um diese *Semantic Gap* zu überwinden, existieren verschiedene Methoden. Unter Xen und KVM kann die Bilbiothek *LibVMI*[3] verwendet werden. Diese wurde früher unter dem Namen *XenAccess* entwickelt (siehe hierzu auch [10]) und bietet die Möglichkeit, von außen in laufenden Windows- und Linux-Gästen sowohl auf physische und virtuelle Adressen als auch auf Kernel-Symbole zuzugreifen (siehe [1]). Mit Hilfe der Kernel-Symbole können nun von außerhalb, z.B. von einer dazu autorisierten anderen Virtuellen Maschine, ähnlich wie etwa bei *Samhain* Funktions-Pointer und andere potentielle *Hooking*-Punkte auf Integrität geprüft werden (siehe Abbildung 3).

Da sich der Wächter nun außerhalb des Betriebssystems der Virtuellen Maschine befindet, ist seine Sichtweise nicht mehr von den potentiell manipulierten Funktionen des Kernels selbst abhängig. Während es für ein Rootkit zwar möglich ist, auf die Existenz eines Hypervisors zu schließen (bzw. auf die Existenz einer virtualisierten Umgebung, etwa indem es prüft, ob die Treiber für bestimmte virtuelle Hardware-Geräte geladen wurden), so bleibt die Existenz des HIDS selbst verborgen, da es sich außerhalb der Reichweite der Virtuellen Maschine befindet. Dadurch ist es nun auch einem Kernel-Mode-Rootkit nicht mehr auf einfache Art und Weise möglich, auf die Anwesenheit und auf die Art eines Wächters zu schließen, da Systeme wie Xen oder KVM auch

[3]LibVMI ist Bestandteil der *vmitools*-Sammlung, siehe [5].

doi: 10.2313/NET-2013-08-1_10

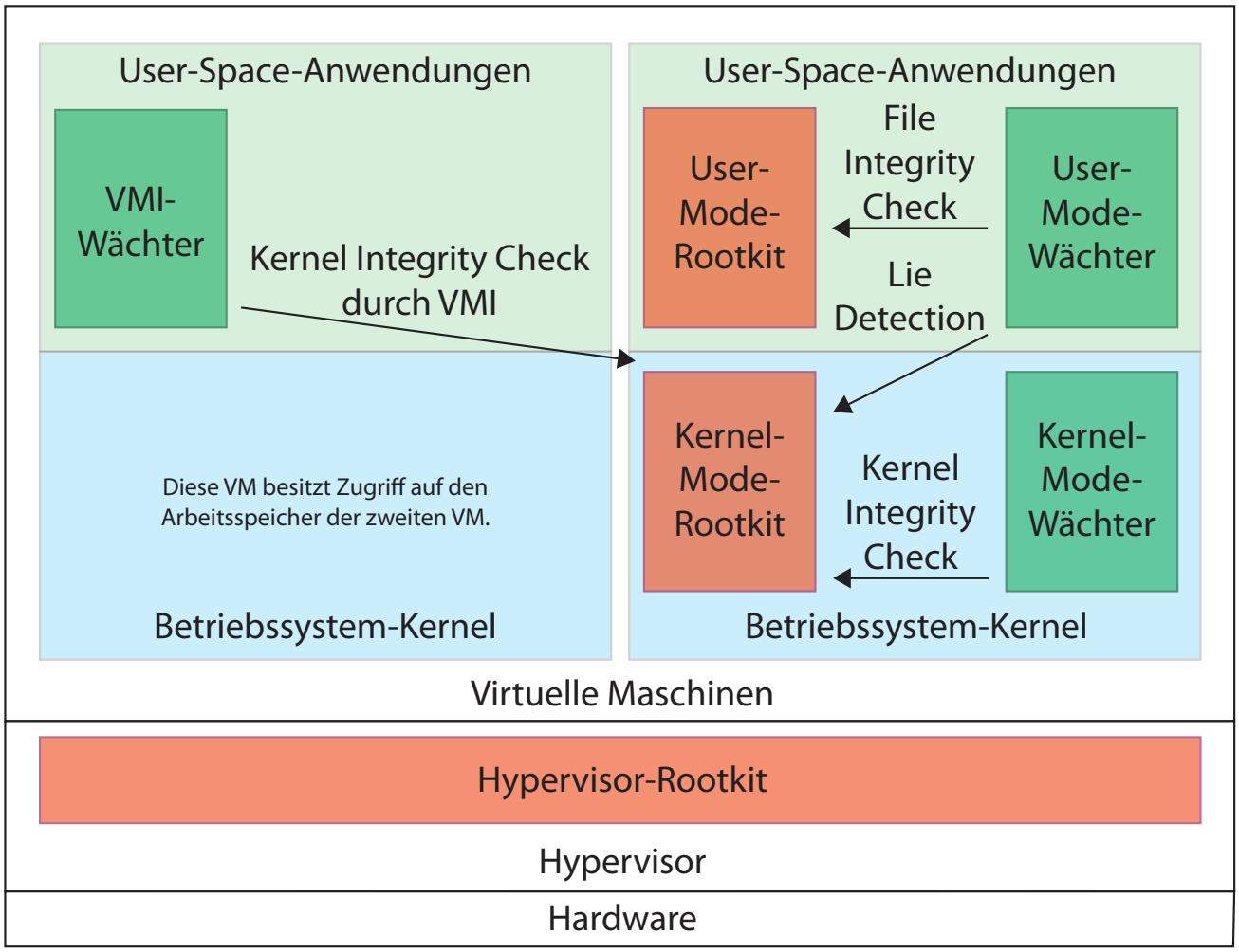

Abbildung 3: Die Abbildung zeigt, auf welche Art und Weise in einer virtualisierten Umgebung bestimmte Typen von Rootkits durch bestimmte Wächter entdeckt werden können. Während die VM auf der rechten Seite normale Anwendungen (etwa einen Web-Server) ausführt, dient die linke VM allein dazu, einen VMI-Wächter auszuführen, der eine oder mehrere andere VMs überprüft.

unabhängig von HIDSen häufig eingesetzt werden.

4.2 Grenzen und Ausblick

Doch auch ein VMI-basierter Wächter kann keine vollständig zuverlässige *Intrusion Detection* garantieren. Wie auch im vorigen Abschnitt ist es essentiell, dass der eigene Hypervisor (ähnlich wie im oberen Fall das HIDS) unmanipuliert gestartet wird, bevor ein Angreifer die Möglichkeit bekommt, Einfluss zu nehmen. Kann durch Schadsoftware der Bootvorgang geändert werden, indem z.B. der Bootloader umkonfiguriert wird, so kann unter Umständen auch der Hypervisor davon betroffen sein. Unter Xen z.B. kann aus der Domain-0 heraus mit Root-Rechten die Boot-Konfiguration der gesamten virtualisierten Umgebung manipuliert werden, indem etwa unter Linux die Einstellungen des Grub-Bootloaders angepasst werden, der wiederum den Xen-Hypervisor bootet.

Laut [13] kann man bei derartigen Rootkits zwischen *Hypervisor-based* und *Virtualization-based Rootkits* unterscheiden.

Die *Hypervisor-based Rootkits*, wie sie etwa in [14] beschrieben werden, manipulieren Datenstrukturen und Funktionen des Hypervisors und orientieren sich damit in der Funktionalität und in der Implementierung stark an Kernel-Mode-Rootkits. Während bei einem solchen in der Regel der Administratorzugriff für den Angreifer ausreichend ist, um das Rootkit zu aktivieren, hängt der Erfolg eines Angriffs auf den Hypervisor von der Implementierung desselben ab (u.a. von den Schutzmaßnahmen, die ergriffen werden, um den Zugriff auf Hypervisor-Speicher aus Virtuellen Maschinen heraus zu verhindern). Für die in [14] gezeigte Demonstration am Beispiel von Xen wurde etwa eine Sicherheitslücke in einem Modul des Xen-Hypervisors verwendet. Da es sich bei dem Xen-Hypervisor um einen eigenen leichtgewichtigen Betriebssystem-Kernel handelt, besitzt dieser auch die typischen Angriffspunkte, die ebenfalls oben beschrieben wurden. Erlangt der Angreifer Schreibzugriff auf den Hypervi-

doi: 10.2313/NET-2013-08-1_10

sor-Speicher, kann er dort etwa den im Hypervisor ebenfalls vorhandenen *Interrupt Handler* oder auch *System-Call*-Äquivalente[4] überschreiben oder manipulieren.

Bei den *Virtualization-based Rootkits* wiederum handelt es sich um Tools, die nicht einen vorhandenen Hypervisor angreifen, sondern versuchen, das anzugreifende System in eine eigene virtualisierte Umgebung zu verschieben, die dann von außen kontrolliert werden kann. Ein bekanntes Beispiel für diese Rootkit-Art stellt etwa Blue Pill dar, ein Rootkit, das ein laufendes Windows-System in eine virtualisierte Umgebung verschieben kann. Das Rootkit selbst dient dabei als leichtgewichtiger Hypervisor, der Hardware-unterstützte Virtualierungstechniken nutzt. Diese virtualisieren für das Gastsystem transparent, sodass es nicht ohne Weiteres möglich ist, diese Manipulation von innerhalb des Gastbetriebssystems festzustellen (siehe [12]). Wie in [13] gezeigt wird, kann diese Art des Angriffs auch auf virtualisierte Umgebungen selbst ausgeweitet werden, sodass ein Xen-Hypervisor oberhalb des Blue-Pill-Hypervisors ausgeführt wird. Wie in den genannten Vorträgen gezeigt wurde, ist das Erkennen solcher Rootkits eine komplexe Aufgabe und zumeist nur über Umwege (wie etwa *Timing Analysis*, siehe auch [12]) möglich.

Auch anderweitig kann es Rootkits gelingen, sich selbst in der Hierarchie über dem Kernel zu positionieren. Vorstellbare Ziele sind das BIOS bzw. das UEFI oder die Firmware von SATA-Controllern oder des Mainboard-Chipsatzes (siehe hierzu auch [15]).

Allen diesen Methoden ist gemein, dass das Rootkit für die Manipulation des eigentlichen Zielsystems bzw. des entsprechenden Kernels Wissen über den Aufbau und die Struktur des Speichers besitzen muss. Damit unterliegen die Rootkits denselben Beschränkungen hinsichtlich der *Semantic Gap*, denen man auch bei der Entwicklung von HIDSen begegnet.

Insgesamt kann nicht ausgeschlossen werden, dass es für ein beliebig komplex implementiertes *Intrusion Detection System* nicht eine Methode gibt, um ein Rootkit vor genau dieser Methode zu verstecken und den Vorteil damit wieder zunichte zu machen. Nur wenn für jedes einzelne Glied der gesamten Kette an Programmen, die ab dem Zeitpunkt des Einschaltens des Rechners ausgeführt werden, die Integrität garantiert werden kann, kann für das jeweils nächste Glied diese Integrität überprüft werden, sodass die Integrität des Gesamtsystems gewahrt bleibt. Solche Integritätsmessungen könnten durch den Einsatz spezieller Hardware-Geräte, die Teil von *Trusted Computing* sind, durchgeführt werden.

5. RELATED WORK

Im Folgenden werden einige Arbeiten vorgestellt, die erweiterte Möglichkeiten vorstellen, Rootkits daran zu hindern, unbemerkt ein System zu befallen.

In ihrem Paper „Detecting Kernel-Level Rootkits Through Binary Analysis“ (siehe [8]) stellen die Autoren Kruegel, Robertson und Vigna ihren Ansatz vor, die Ausführung von Rootkits in Form von dynamisch ladbaren Linux-Kernel-Modulen (LKMs) durch *Binary Analysis* zu verhindern. Dabei wird der Code von zu ladenden Modulen auf verdächtiges bzw. bösartiges Verhalten untersucht und das Laden des Moduls bei drohender Gefahr unterbunden, bevor Befehle im Kernel-Mode ausgeführt werden konnten. Für die Analyse wird das Verhalten des Codes durch *Symbolic Execution* simuliert, um Rückschlüsse auf gelesene bzw. geschriebene Speicheradressen zu erhalten. Mit Hilfe dieser Adressen wird schließlich entschieden, ob es sich um legitime oder um verbotene Zugriffe handelt. Die Autoren zeigen, dass sich in der Praxis die zum Zeitpunkt des Papers aktuellen Linux-Rootkits mit sehr hoher Wahrscheinlichkeit erkennen lassen und es zudem nicht zu Fehlalarmen im Bezug auf legitime Kernel-Module kommt.

Wampler und Graham beschreiben in „A normality based method for detecting kernel rootkits“ (siehe [17]) eine Methode, Veränderungen in der Behandlung von *System Calls* im Linux-Kernel aufzuspüren. Die Methode erkennt einerseits das Überschreiben der Adresse der *System Call Table* mit einer anderen, durch das der *System Call Handler* Adressen nicht mehr der Original- sondern einer weiteren, manipulierten Tabelle entnimmt. Im Falle der Abbildung 1 bedeutet dies, dass etwa die Funktion *system_call()* auf der linken Seite so manipuliert wird, dass sie als Tabelle nicht mehr *sys_call_table*, sondern eine Kopie davon verwendet. Andererseits kann auch das Einfügen von Jump-Anweisungen an den Anfang der Kernel-Implementierungen der *System Calls* (in Abbildung 1 auf der rechten Seite), mit dem direkt beim Aufruf der entsprechenden Funktionen Angreifer-Code ausgeführt wird, entdeckt werden. Um dies zu erreichen, werden die Ziele aller Jump-Instruktionen des Kernels statistisch erfasst, um *Outliers*, also Elemente außerhalb der Normalität, zu finden. Die Autoren zeigen anhand des **enyelkm**-Rootkits, dass die Methode geeignet ist, mögliche Manipulationen am Kernel zu erkennen.

Im Gegensatz zu den beiden vorigen Methoden setzen die Autoren Wang, Jiang, Cui und Ning in ihrer Arbeit „Countering kernel rootkits with lightweight hook protection“ (siehe [18]) auf einen Hypervisor-basierten Ansatz. Hierfür stellen sie *HookSafe* vor, ein leichtgewichtiges, auf Xen basierendes System zum Schutz von Kernel-Hooks vor Manipulationen. Basierend auf der Annahme, dass Hooks zwar oft gelesen, aber fast nie geschrieben werden, verschiebt *HookSafe* die Hooks in einen eigenen, dem Kernel nicht direkt verfügbaren Speicherbereich. Auf dieser Art und Weise können Zugriffe auf die entsprechenden Speicherbereiche kontrolliert und die Hooks vor Manipulation geschützt werden. In einer Evaluation mit neun „real world“ Rootkits zeigt sich, dass sich der Ansatz dazu eignet, typische Kernel-Rootkits effektiv daran zu hindern, mit Hilfe von Kernel-Hooks Schaden anzurichten. Da *HookSafe* auf Virtualisierungstechniken wie *Binary Translation* etc. zurückgreift, geht der Einsatz des Systems mit einem Performance-Einbruch von ca. 6% einher.

Auch die Autoren Litty, Lagar-Cavilla und Lie schließlich setzen in ihrem Paper „Hypervisor support for identifying covertly executing binaries“ (siehe [9]) auf einen Hypervisor-basierten und mit Hilfe von Xen implementierten Ansatz. Der Ansatz sieht ein *Patagonix* getauftes System vor, das ausführbaren Code im Arbeitsspeicher analysiert und

[4]Unter Xen gibt es die sogenannten *Hyper Calls*, die es analog zu klassischen *System Calls* den VMs erlauben, Funktionen des Hypervisors aufzurufen.

doi: 10.2313/NET-2013-08-1_10

aus diesen Informationen auf die aktuell laufenden Prozesse innerhalb eines Systems schließt. *Patagonix* ist dabei nur abhängig vom eingesetzten Binärformat der ausführbaren Dateien, gleichzeitig aber unabhängig vom Betriebssystem, sodass sowohl Windows als auch Linux untersucht werden können. Unterstützt werden zwei Betriebsmodi: Während im *Reporting Mode* nur eine Prozessliste erstellt wird, die auf einem unmanipulierten System der Ausgabe von Programmen wie **ps** entsprechen sollte, vergleicht *Patagonix* im *Lie Detection Mode* die selbst errungenen Ergebnisse mit den Ausgaben eines im Betriebssystem laufenden Services, der die Kernel-Schnittstellen zum Erzeugen der Prozessliste verwendet. Auf diese Art und Weise können sowohl versteckte als auch nicht-existierende Prozesse erkannt werden. Die Autoren zeigen, dass eine solche Lösung nur mit einem geringen Einbruch in der Performance einhergeht.

6. ZUSAMMENFASSUNG

Insgesamt zeigt sich also, dass der Erfolg einer *Intrusion Detection* davon abhängt, welcher Angreifer auf welchen Wächter trifft. Auf beiden Seiten kann durch geschickte Programmierung ein Vorteil gegenüber dem Gegner erreicht werden.

Während es für einen Wächter relativ einfach ist, ein bestimmtes Rootkit mit bekannten Symptomen zu erkennen, muss er bei unbekannten Rootkits darauf vertrauen, dass diese sich auf eine bestimmte Art und Weise verdächtig verhalten. Als Beispiel für das Erkennen von bekannten Symptomen sei hier OSSEC genannt, das eine Liste mit Dateien enthält, die in der Regel von bestimmten Rootkits angelegt werden.

Genau dies kann typischen HIDSen jedoch mit zunehmender Komplexität der Rootkits immer schwerer fallen. Ein Rootkit, das sich als Hypervisor sogar eine Stufe tiefer als das Betriebssystem befindet, in dem das HIDS läuft, kann sich etwa sowohl der *Lie Detection* als auch der Überprüfung des Kernel-Speichers nahezu beliebig entziehen, sodass die Kompromittierung des Systems über lange Zeit unbemerkt bleiben kann. Da auf der anderen Seite die Entwicklung eines solchen Rootkits einen nicht zu vernachlässigenden Aufwand bedeutet, existieren entsprechende Implementierungen wie Blue Pill nach wie vor in der Regel als Forschungsobjekte und nicht in freier Wildbahn.

Allein die Möglichkeit der Existenz solcher Rootkits sollte jedoch Anlass genug dazu geben, auch über produktiv einsetzbare Möglichkeiten nachzudenken, komplexe Rootkits, wie sie im Abschnitt 4.2 beschrieben worden sind, zuverlässig erkennen zu können. Es kann daher durchaus in Betracht gezogen werden, alternativ oder zusätzlich zu einem HIDS auch auf eine Lösung wie *Trusted Computing* (und etwa *Trusted Network Connect*) zu vertrauen, um die Integrität eines Systems zuverlässig überprüfen zu können.

7. LITERATUR

[1] An introduction to libvmi. https://code.google.com/p/vmitools/wiki/LibVMIIntroduction, abgerufen am 25.05.2013.

[2] Ossec - open source host-based intrusion detection system. http://www.ossec.net/, abgerufen am 20.05.2013.

[3] Ossec rootcheck module. http://www.ossec.net/doc/manual/rootcheck/manual-rootcheck.html, abgerufen am 20.05.2013.

[4] Samhain - open source host-based intrusion detection system. http://la-samhna.de/samhain/, abgerufen am 21.05.2013.

[5] vmitools - virtual machine introspection utilities. https://code.google.com/p/vmitools/, abgerufen am 25.05.2013.

[6] Anatomy of the linux file system - a layered structure-based review. http://www.ibm.com/developerworks/linux/library/l-linux-filesystem/, abgerufen am 18.05.2013, Oct. 2007.

[7] H. Debar, M. Dacier, and A. Wespi. Towards a taxonomy of intrusion-detection systems. *Computer Networks*, 31(8):805 – 822, 1999.

[8] C. Kruegel, W. Robertson, and G. Vigna. Detecting kernel-level rootkits through binary analysis. In *Proceedings of the 20th Annual Computer Security Applications Conference*, ACSAC '04, pages 91–100, Washington, DC, USA, 2004. IEEE Computer Society.

[9] L. Litty, H. A. Lagar-Cavilla, and D. Lie. Hypervisor support for identifying covertly executing binaries. In *Proceedings of the 17th conference on Security symposium*, SS'08, pages 243–258, Berkeley, CA, USA, 2008. USENIX Association.

[10] B. Payne, M. de Carbone, and W. Lee. Secure and flexible monitoring of virtual machines. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pages 385–397, 2007.

[11] R. S. Peláez. Linux kernel rootkits: protecting the system's "ring-zero". *SANS Institute*, May 2004.

[12] J. Rutkowska and A. Tereshkin. Subverting vista kernel for fun and profit, presentation at black hat usa. http://blackhat.com/presentations/bh-usa-06/BH-US-06-Rutkowska.pdf, Black Hat USA, abgerufen am 26.05.2013, Aug. 2006.

[13] J. Rutkowska and A. Tereshkin. Bluepilling the xen hypervisor, presentation at black hat usa. http://invisiblethingslab.com/resources/bh08/part3.pdf, Black Hat USA, abgerufen am 26.05.2013, Aug. 2008.

[14] J. Rutkowska and R. Wojtczuk. Preventing and detecting xen hypervisor subversions, presentation at black hat usa. http://invisiblethingslab.com/resources/bh08/part2-full.pdf, Black Hat USA, abgerufen am 26.05.2013, Aug. 2008.

[15] A. Tereshkin and R. Wojtczuk. Introducing ring -3 rootkits, presentation at black hat usa. http://invisiblethingslab.com/resources/bh09usa/Ring%20-3%20Rootkits.pdf, Black Hat USA, abgerufen am 26.05.2013, July 2009.

[16] W.-J. Tsaur and Y.-C. Chen. Exploring rootkit detectors' vulnerabilities using a new windows hidden driver based rootkit. In *Proceedings of the 2010 IEEE Second International Conference on Social Computing*, SOCIALCOM '10, pages 842–848, Washington, DC, USA, 2010. IEEE Computer Society.

[17] D. Wampler and J. H. Graham. A normality based method for detecting kernel rootkits. *SIGOPS Oper. Syst. Rev.*, 42(3):59–64, Apr. 2008.

[18] Z. Wang, X. Jiang, W. Cui, and P. Ning. Countering kernel rootkits with lightweight hook protection. In

Proceedings of the 16th ACM conference on Computer and communications security, CCS '09, pages 545–554, New York, NY, USA, 2009. ACM.

[19] B. Zdrnja. Sshd rootkit in the wild. https://isc.sans.edu/diary/SSHD+rootkit+in+the+wild/15229, Version 3, abgerufen am 30.05.2013, Feb. 2013.

doi: 10.2313/NET-2013-08-1_10