Linux Rootkits

Clemens Paul
Betreuer: Holger Kinkelin, Simon Stauber
Seminar Future Internet SS2013
Lehrstuhl Netzarchitekturen und Netzdienste
Fakultät für Informatik, Technische Universität München
Email: clemens.paul@in.tum.de

KURZFASSUNG

Rootkits sind Programme, die sich selbst sowie laufende Prozesse, Dateien, Module und Systeminformationen vor dem System verbergen können. Eine weitere häufig verbreitete Funktion ist das Erhöhen von Zugriffsrechten. Zumeist werden Rootkits dazu benutzt, Schadprogramme wie Viren, Würmer und Spyware unerkannt einzuschleusen und auszuführen. Einmalige Rootrechte sind die Grundvoraussetzung dafür, dass Rootkits installiert werden können. In der Folge bleiben sie mit diesen Rechten dauerhaft ausgestattet und verbergen in der Regel ihre Präsenz. Um dies zu erreichen, ersetzen User-Mode-Rootkits Binärdateien. Bei Kernel-Mode-Rootkits gibt es verschiedene Methoden, um den Kernel zu kompromittieren, z.B. das Hinzufügen eines *Loadable Kernel Modules* (*LKM*) oder das Patchen des Kernels durch die Devices */dev/mem* und */dev/kmem*. Bei Kernel-Mode-Rootkits werden häufig System-Calls verwendet, um Schadcode auszuführen. Dadurch wird bei Zugriffen von Anwendungen auf bestimmte System-Befehle und Ressourcen jedesmal das Rootkit mit ausgeführt.

Rootkits stellen eine große Gefahr für Computersysteme dar, da sie schwer zu entdecken sind und auf jegliche Ressource des Systems zugreifen können.

Schlüsselworte

Linux, Rootkit, Malware, System Call, Kernel, IDT, Loadable Kernel Module

1. EINLEITUNG

Ein Rootkit ist ein Programm, mit dem laufende Prozesse, Dateien oder Systeminformationen verborgen werden können. Raúl Siles Peláez definiert Rootkits folgendermaßen: „[...] a rootkit is 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.“ [1, p. 8] Ein Rootkit wird also nicht verwendet, um Root-Zugriff zu erlangen, sondern um diesen aufrecht zu erhalten. Ein vorausgehender Root-Zugriff ist überhaupt die Voraussetzung, um ein Rootkit auf einem System zu installieren. Typische Funktionalitäten eines Rootkits sind [1, p. 9]:

- Stealth-Funktionalität: verbirgt Spuren des Angreifers, u.a. Prozesse, Dateien, Netzwerk-Aktivitäten
- Backdoor: ermöglicht dem Angreifer jederzeit wieder auf das System zugreifen zu können
- Sniffer: zum „Abhören“ verschiedener System-Komponenten, z.B. Netzwerk, Keylogger

Der Begriff Rootkit setzt sich zusammen aus „root“, der Bezeichnung für Unix-Anwender mit Administratorrechten, und dem Wort „kit“, welches sich auf die Software-Komponenten bezieht, aus denen das Programm besteht.

Zur Klassifikation kann eine Einteilung in Rootkits, die auf der Anwender-Ebene ausgeführt werden, und Rootkits, die auf der Ebene des Kernels ausgeführt werden, vorgenommen werden [1, p. 12]. Der Fokus dieser Arbeit liegt auf den Kernel-Mode-Rootkits.

Rootkits gibt es für eine Vielzahl von Betriebssystemen wie Linux, Solaris, Mac OSX [34] und diverse Versionen von Microsoft Windows. Häufig verändern sie Elemente des Betriebssystems oder installieren sich als Treiber. Üblicherweise verbirgt ein Rootkit Logins, Prozesse, Dateien und Logs. Es kann auch Code beinhalten, der Daten von Terminals, Netzwerk-Verbindungen und Tastatureingaben abfängt bzw. aufzeichnet. Im Rahmen der Klassifikation von Schadprogrammen werden die meisten Rootkits zu den Trojanern gezählt [15, p. 890].

Ursprünglich auch für neutrale Anwendungsbereiche entwickelt, werden Rootkits in den vergangenen Jahren immer häufiger zum unbemerkten Einschleusen und Ausführen von Schadprogrammen verwendet [15, p. 890].

Der erste PC-Virus „Brain“ nutzte bereits rootkit-ähnliche Eigenschaften, um sich selbst zu verbergen [2, p. 17]. Die ersten moderneren Rootkits wurden von Hackern in den frühen 1990er Jahren entwickelt, um Unix-Systeme anzugreifen [2, p. 17]; auf Linux-Systemen tauchen sie ab Mitte der 1990er Jahre auf [15, p. 890].

In den vergangenen Jahren verwendeten Rootkits immer wieder neue und durchdachtere Techniken, wodurch sie schwerer zu entdecken sind. Dabei werden sie auf unterschiedlichen System-Ebenen bis hin zu den tiefsten Schichten des Kernels ausgeführt [15, p. 890].

Rootkit-Angriffe sind eine ernstzunehmende Bedrohung für Computer-Systeme. Im Verbund mit Schadprogrammen wie Würmern, Viren und Spyware stellen sie eine größere Gefahr denn je dar, indem sie es Schadprogrammen erlauben, vom System unentdeckt zu bleiben. Ohne passende Tools zum Aufspüren von Rootkits können Schadprogramme auf befallenen Systemen über lange Zeiträume unentdeckt ausgeführt werden [25, p. 1].

Der Rest der Arbeit ist wie folgt aufgebaut:

Abschnitt 2 gibt einen allgemeinen Überblick zur Funktionsweise und Klassifikation von Rootkits und geht in weiterer Folge näher auf User-Mode- und Kernel-Mode-Rootkits ein. Der Fokus liegt

doi: 10.2313/NET-2013-08-1_04

dabei auf Kernel-Mode-Rootkits, deren Hauptmethoden das Patchen des Kernels durch die Devices */dev/mem* und */dev/kmem* sowie *Loadable Kernel Modules* im Einzelnen dargestellt werden.

Abschnitt 3 befasst sich mit der Funktionsweise von System-Calls. Nach einem allgemeinen Überblick wird auf die Hooking-Methoden IDT Hooking, System-Call-Table Hooking und System-Call-Handler Hooking näher eingegangen.

In Abschnitt 4 wird die Funktionalität eines selbst entwickelten Rootkits erläutert. Anhand dieses Beispiels wird das Verbergen von Dateien, Prozessen und Modulen sowie das Erhöhen von Zugriffsrechten demonstriert.

Punkt 5 schließlich gibt Hinweise auf verwandte bzw. weiterführende Arbeiten.

Der letzte Punkt gibt einen Ausblick auf die Entwicklung von Rootkits und beschreibt verschiedene Gefahrenquellen.

2. Rootkits

Dieser Abschnitt gibt einen Einblick in die Struktur des Linux-Kernels und beschreibt, wie User- und Kernel-Mode-Rootkits aufgebaut sind. Bei den Kernel-Mode-Rootkits werden sowohl das Patchen des Kernels durch die Devices */dev/mem* und */dev/kmem* als auch die Methode *Loadable Kernel Modules* (*LKM*) erläutert.

Der Linux-Kernel wurde 1991 von Linus Torvalds entwickelt und dient als Kern für verschiedene Open Source-Betriebssysteme. Der Kernel wird unter der GNU Public License (GPL) veröffentlicht [26]. Eines der Hauptmerkmale des Kernels ist die monolithische Architektur, welche durch das zusätzliche Einbinden von Modulen in den Kernel komplettiert wird [1, p. 19]. Dieser Mechanismus wird auch von einigen Rootkits (*LKM*) verwendet.

Der Kernel dient als Kommunikationsschnittstelle zwischen den Software-Anwendungen (User Space) und der Hardware. Um diese Schnittstelle zu schützen, müssen alle Prozesse mit Hilfe von Systembibliotheken mit dem Kernel kommunizieren. Diese senden anschließend System-Calls, um in den Kernel-Mode zu gelangen. Dort wird die System-Call-Table aufgerufen, die je nach System-Call oder Funktion mit dem Text- oder Data-Segment des Kernels interagiert. Wenn dies ausgeführt wurde, kommuniziert der Kernel entsprechend mit der Hardware [5].

2.1 User-Mode-Rootkits

User-Mode-Rootkits, auch „Traditional Rootkits“ [1, p. 12] genannt, infizieren das Betriebssystem außerhalb des Kernel-Levels [2, p. 18]. Dabei wird der Fokus auf das Ersetzen spezifischer Systemkomponenten gesetzt, um bestimmte Informationen des Systems zu extrahieren, wie z.B. laufende Prozesse oder Netzwerkverbindungen, Inhalte des Dateisystems, Treiber oder DLL-Files. Dies verfolgt zwei unterschiedliche Ziele: Zum einen soll der unautorisierte Zugriff erhalten und verborgen bleiben; zum anderen sollen Administrator-Rechte wiedererlangt werden [1, p. 12] [2, p. 18]. Des Weiteren können User-Mode-Rootkits System-Calls zwischen dem Kernel und den Software-Programmen abfangen, um sicherzustellen, dass die weitergegebenen Informationen keinen Nachweis für die Existenz des Rootkits liefern [2, p. 18]. Aus diesen Gründen werden User-Mode-Rootkits häufig als klassische Beispiele für Trojaner bezeichnet [1, p. 12]. Um beispielsweise Prozesse vor dem Anwender zu verbergen, kann das Kommando *ps* durch eine Schadvariante ersetzt werden. Wenn der Anwender allerdings das Kommando *top* verwenden würde, um Prozesse anzuzeigen, würde der Angriff fehlschlagen. Aus diesem Grund müsste der Angreifer auch das Kommando *top* und zahlreiche andere Programme, die Prozesse anzeigen können, durch eine Schadvariante ersetzen. Dieses Beispiel zeigt (aus der Sicht des Angreifers) das Hauptproblem von User-Mode-Rootkits: Es müssen zu viele Binärdateien ersetzt werden [1, p. 13]. Dies kann durch Programme wie Tripwire [3], die auf Integrität prüfen, leicht entdeckt werden. Des Weiteren sind Binärdateien sehr betriebssystem-spezifisch und müssen für eine bestimmte Plattform kompiliert werden.

User-Mode-Rootkits laufen, bezogen auf die Unix Ring-Terminologie, in Ring 3, welcher als „Userland“ bezeichnet wird. In diesem Ring laufen die Benutzer-Anwendungen sowie nicht vertrauenswürdige Programme, da das Betriebssystem dieser Ebene die niedrigsten Zugriffsrechte gibt. Aus diesem Grund ist die Erkennung von und Vorbeugung gegen User-Mode-Rootkits einfacher als bei Kernel-Mode-Rootkits [27].

Eines der am häufigsten verwendeten User-Mode-Rootkits ist das „Login“ Rootkit, das in der Regel ein Backdoor mit einem Passwort für den Root-Zugriff besitzt. Die ersten solchen Rootkits beinhalteten das Passwort im Klartext, wodurch es mit dem *string*-Befehl gefunden werden konnte. Neuere Versionen dieses Rootkits verwenden andere Methoden, um das Passwort zu verschleiern, wie z.B. das Passwort in einer Binärdatei abzuspeichern, Permutation des Passworts vorzunehmen oder das Passwort aus einer Datei zu lesen [1, p. 12]. Beim „Login“ Rootkit wird das „Login“ Binary des Systems mit dem Rootkit ersetzt. Dies ermöglicht das Überwachen und Aufzeichnen sämtlicher Logins. Um auf die Liste zugreifen zu können, kann sie beispielsweise zu einem Server gesendet werden [35, p. 4]. Zwei weitere sehr bekannte User-Mode-Rootkits sind T0rnkit (verwendet vom Wurm Lion im März 2001) und LRK (The Linux Rootkit) [1, p. 13].

2.2 Kernel-Mode-Rootkits

Bei Kernel-Mode-Rootkits müssen im Gegensatz zu den User-Mode-Rootkits nicht mehrere Programme verändert werden – was zu mehr Arbeit für den Angreifer führt –, sondern es muss nur der Kernel modifiziert oder ersetzt werden. Dabei bieten Kernel-Mode-Rootkits alle Features von User-Mode-Rootkits auf einer tieferen Ebene an und können durch verschiedene Techniken jegliche Analysetools für den User-Mode täuschen [1, p. 14]. Bezogen auf die Ring-Terminologie befinden sich Kernel-Mode-Rootkits in Ring 0, also jenem mit den höchsten Rechten. Dadurch sind Kernel-Mode-Rootkits schwieriger zu entdecken als User-Mode-Rootkits, vor allem vom infizierten System aus [5].

Wie bereits in Punkt 2 erläutert, kontrolliert der Kernel jegliche Anwendung, die auf dem System läuft. Wenn eine Applikation versucht eine Datei zu lesen oder zu schreiben, müssen die notwendigen System-Ressourcen vom Kernel bereit gestellt werden. Dieses Bereitstellen erfolgt durch System-Calls, die vom Kernel kontrolliert werden. Ein Kernel-Mode-Rootkit kann diese System-Calls modifizieren und dadurch Schadcode ausführen [4, p. 4]. Um beispielsweise Dateien vor dem Anwender zu verbergen, kann der Angreifer den System-Call *getdents64* manipulieren, anstatt das Programm *ls* zu ersetzen.

Kernel-Mode-Rootkits können nicht nur ihre eigene Präsenz gegenüber dem User verbergen, sondern auch jede andere Art von

doi: 10.2313/NET-2013-08-1_04

Malware, die das Rootkit eingeschleust hat. Damit Rootkits nach dem Neustart des Systems immer noch vorhanden sind, werden sie beim Booten des Kernels mit geladen. Um den ursprünglichen Zustand eines System nach solch einer Kernel-Mode-Rootkit-Attacke wiederherzustellen, ist die Neuinstallation des Betriebssystems erforderlich, meint Johannes B. Ullrich, Chief Research Officer des SANS Institute, einer Organisation für Computer-Sicherheit [2, p. 18].

Das Ziel von Kernel-Mode-Rootkits besteht darin, Schadcode in den Kernel zu laden, die Quellen des Kernels zu modifizieren oder durch eine andere Möglichkeit den laufenden Kernel zu manipulieren. Auf zwei dieser Möglichkeiten wird in dieser Arbeit im Detail eingegangen. Unter Punkt 2.2.1 wird das Patchen des Kernels durch die Devices */dev/mem* und */dev/kmem* beschrieben. In Punkt 2.2.2 wird das Konzept des *Loadable Kernel Modules* (*LKM*) erläutert [1, p. 14f].

2.2.1 Patchen des Kernels durch die Devices /dev/mem *und* /dev/kmem

Das */dev/mem* Device bildet den physikalischen Speicher des Systems ab. Dagegen repräsentiert das */dev/kmem* Device den virtuellen Speicher, der vom laufenden Kernel genutzt wird, inklusive *Swap*. Da Linux für den User *root* sowohl Schreib- wie auch Lese-Zugriff auf die Devices */dev/mem* und */dev/kmem* erlaubt, kann der laufende Kernel modifiziert werden [1, p. 68]. Es können dazu die Standard APIs *read()*, *write()* und *mmap()* genutzt werden.

Eines der Probleme dabei ist, dass das Verfahren relativ aufwändig ist, da ein spezielles Element, das abgeändert werden soll, im unstrukturierten Speicher gefunden werden muss.

Der Zugriff auf */dev/kmem* kann nur durch das Patchen des Kernels verhindert werden. Solch ein Patch wurde in „Phrack“ [8] veröffentlicht. Allerdings kann trotz dieses Patches auf das Device */dev/kmem* geschrieben werden. Die erfolgt durch Verwendung von *mmap()* anstelle einer direkten Manipulation der Datei via I/O [6].

„SuckIT“ (Super User Control Kit) ist eines der bekanntesten Kernel-Mode-Rootkits, das das */dev/kmem* Device verwendet. Im Gegensatz zu vielen anderen Kernel-Mode-Rootkits, die die System-Call-Table manipulieren, bleibt diese beim „SuckIT“-Rootkit unverändert. Dadurch kann das Rootkit nicht durch Überprüfung der originalen System-Call-Table entdeckt werden. „SuckIT“ gelingt dies, indem der System-Call-Interrupt (System-Call-Funktion) modifiziert wird. Dieser Interrupt wird von jedem User-Mode-Prozess, der einen System-Call benötigt, automatisch aufgerufen. Der Pointer zur normalen System-Call-Table wird zu einer vom Rootkit neu erstellten System-Call-Table umgeleitet. Diese neue System-Call-Table beinhaltet sowohl Adressen zu System-Calls, die durch Schadcode modifiziert wurden, als auch original Adressen zu den unveränderten System-Calls [8].

Einige Funktionen, die „SuckIT“ bereitstellt sind: Prozesse, Dateien und Netzwerk-Sockets (TCP, UDP, RAW) verbergen oder das Sniffen von TTYs [7, p. 111f].

2.2.2 Loadable Kernel Module

Kernel-Module sind Code-Sequenzen, die die Funktionalität eines Kernels erweitern und bei laufendem Betrieb (ohne Neustart) in den Kernel geladen und anschließend wieder aus dem Kernel entfernt werden können. Wenn dies nicht möglich ist, muss der komplette Kernel neu kompiliert werden, um eine zusätzliche Funktionalität einzubauen. Dies führt auch zu dem Nachteil, dass das Kernel-Image sehr groß wird. Ein Beispiel für ein Modul ist ein Gerätetreiber, durch den das System mit der Hardware kommunizieren kann [9, p. 2]. Alle zurzeit geladenen Module können bei einem Linux-System mit dem Befehl *lsmod* angezeigt werden. Dabei liest *lsmod* die Datei */proc/modules*, um die Informationen zu erhalten [9, p. 2]. Es besteht die Möglichkeit den Kernel so zu kompilieren, dass das *LKM* Interface nicht länger unterstützt wird [6].

Das Listing 1 zeigt den Quellcode eines Kernel-Moduls, das „Hello“ bei der Initialisierung und „Goodbye“ beim Entladen des Moduls auf den Bildschirm schreibt:

```
#include <linux/module.h>    // used by every module
#include <linux/kernel.h>    // used by KERNEL_ALERT
#include <linux/init.h>      // used by macros

static int init(void) {
        printk(KERN_ALERT "Hello\n");
        return 0;
}
static void exit(void) {
        printk(KERN_ALERT "Goodbye\n");
}
module_init(init);
module_exit(exit);
```

Listing 1: Zeigt den Quellcode des Kernel-Moduls „hello.c“.

Anschließend kann das Modul mit dem Programm *make*, sofern ein entsprechendes Makefile vorhanden ist, kompiliert werden. Das fertige Kernel-Modul kann nun mit dem Befehl *insmod* in den Kernel geladen und anschließend wieder mit dem Befehl *rmmod* entfernt werden. Diese Befehle benötigen Root-Rechte.

Beim Ausführen des Befehls *insmod* führt das System folgende Prozesse aus [10]:

- Laden der Objektdatei, in diesem Beispiel hello.o
- Aufrufen des System-Calls *create_module*, um entsprechend Speicher zu verlagern
- Auflösen von Referenzen, die noch nicht aufgelöst wurden, durch Kernel-Symbole mit dem System-Call *get_kernel_syms*
- Initialisieren des *LKMs*, in dem die *init_module(void)*-Funktion ausgeführt wird, mit dem System-Call *init_module*

Eines der Hauptprobleme von *LKMs* besteht darin, dass sie sich nach einem Neustart des Systems nicht mehr im Kernel befinden, sofern sie nicht während des Boot-Prozesses in den Kernel geladen werden. Dies verlangt das Verändern eines System-Boot-Scripts, das einfach durch eine Integritäts-Prüfung entdeckt werden kann. Zwei sehr nützliche Methoden, um einen System-Neustart zu überstehen, sind [1, p. 65, 143]:

- Ein bereits auf dem System laufendes *LKM* infizieren. Bei diesem Angriff werden zwei *LKMs* miteinander verbunden. Dies funktioniert dadurch, dass *LKMs* ELF-Objekte (Executable and Linking Format) sind. Diese

doi: 10.2313/NET-2013-08-1_04

Objekte können mit dem *Linker* verbunden werden. Dabei ist zu beachten, dass zwei Symbole (z.B. Funktionen) nicht den gleichen Namen haben dürfen, z.B. müssen die *init_module()* und *exit_module()* Funktionen im Rootkit einen anderen Namen haben als im zu infizierenden Modul. Für den erfolgreichen Angriff wird in der *init_module()* Funktion des Rootkits die *init_module()* Funktion des originalen *LKMs* aufgerufen. Nachdem die zwei *LKMs* verbunden wurden, muss der *.srttab* Abschnitt des verbundenen ELF-Objekts verändert werden, um die *init_module()* Funktion des Rootkits und nicht die *init_module()* Funktion des originalen LKMs standardmäßig aufzurufen. Ein zu infizierendes Modul könnte beispielsweise der Sound-Treiber sein [11].

- Das Kernel-Image auf der System-Platte patchen (siehe [12]).

Abschließend drei Beispiele für sehr verbreitete *LKM*-Rootkits. Die Rootkits „adore“ und „adore-ng“ des Entwicklers Stealth können Dateien, Prozesse und Services verbergen und einem Prozess Root-Rechte geben. Die Rootkits verbergen auch ihre eigene Präsenz und können somit nicht mehr mit dem Befehl *rmmod* entfernt werden. Gesteuert werden sie mit dem zusätzlichen Programm *ava* [13]. Damit „adore-ng“ schwerer zu entdecken ist, wird nicht die System-Call-Table verwendet, sondern die Kernel-Ebene des Virtual-File-System (VFS) manipuliert. Es läuft unter der Kernel-Version 2.6 [1, ch. 4].

Ein weiteres Rootkit ist „knark“, das von Creed entwickelt wurde und für die Linux Kernel-Versionen 2.2 und 2.4 funktioniert [14]. Es bietet ähnliche Funktionalität wie die bereits genannten Rootkits. „Knark“ wird von mehreren Hilfs-Programmen gesteuert und kann Befehle, die von einem Remote-Host geschickt werden, ausführen [13].

3. System-Call

Dieser Abschnitt der Arbeit befasst sich mit System-Calls. Dabei wird zuerst beschrieben, was ein System-Call ist und wie dieser in einem Linux-System erfolgt. Des Weiteren werden verschiedene Möglichkeiten erläutert, wie ein Rootkit Schadcode durch System-Calls ausführen kann.

3.1 Allgemeines zu System-Calls

Wie bereits erwähnt werden die Applikationen auf einem Computer in Ring 3 und der Kernel in Ring 0 ausgeführt. Wenn nun Applikationen auf bestimmte System-Ressourcen zugreifen müssen, wird eine Kommunikation mit dem Kernel über System-Calls hergestellt. Der Kernel kann auf alle System-Ressourcen zugreifen und schickt das Ergebnis an die Applikation zurück [15, p. 1]. Beispiele für System-Calls sind *sys_read*, *sys_write* und *sys_fork*. Ein Rootkit könnte beispielsweise den *sys_read* System-Call hooken, wodurch jedesmal Code des Rootkits ausgeführt wird, wenn dieser System-Call aufgerufen wird. Damit könnte das Rootkit als Keylogger arbeiten und jeden Tastendruck des Systems aufzeichnen.

Es gibt zwei verschiedene Methoden, wie ein System-Call in einer x86-Architektur aufgerufen werden kann. Dies sind die Assembler-Instruktionen *int 0x80* und *sysenter* (wird seit Linux Kernel 2.6 unterstützt) [16, p. 401].

Jeder System-Call hat seine eigene Nummer, die im EAX-Register übermittelt wird. Zusätzlich werden die Register EBX, ECX, EDX, ESI, EDI und EBP als Übergabeparameter verwendet sowie die Register ESP, SS, CS, EIP, EFLAGS vom User-Mode-Stack auf den Kernel-Mode-Stack gespeichert, bevor der Interrupt-Handler läuft. Anschließend wird durch Aufrufen der entsprechenden C-Funktion, die als System-Call-Service-Routine bezeichnet wird, der System-Call abgearbeitet. Nach Beendigung des System-Calls werden die Register wieder zurückgeschrieben [15, p. 1].

Das Ergebnis ist ein Sprung zu einer Assembler-Funktion, die als System-Call-Handler bezeichnet wird. Alle System-Calls geben einen Integer als Ergebnis zurück, wobei ein positiver Wert oder eine 0 für eine erfolgreiche Ausführung und ein negativer Wert für einen Fehler steht [16, p. 399f].

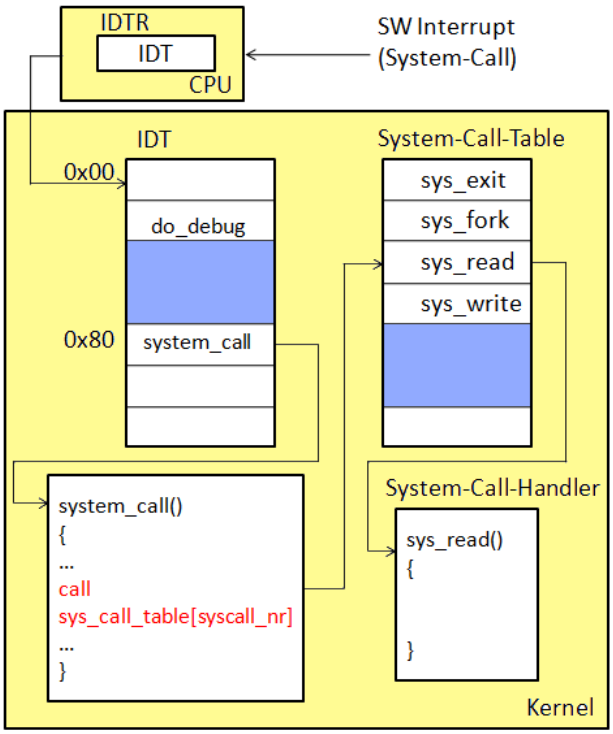

Abbildung 1: Stellt einen System-Call-Hook schematisch dar. [Grafik erstellt nach 28]

Die Interrupt-Descriptor-Table (IDT) ist eine Systemtabelle, die jeden Interrupt- und Exception-Vektor mit den jeweiligen Adressen der Interrupt- und Exception-Handler verbindet. Da die IDT aus maximal 256 Einträgen besteht und jede Adresse 8 Byte groß ist, werden für die IDT maximal 2048 Bytes an Speicher benötigt. Das IDT Register (IDTR) spezifiziert die physikalische Base-Adresse und das Limit (maximale Länge) der IDT, wodurch die IDT an einer beliebigen Stelle im Speicher hinterlegt werden kann. Um Interrupts zu ermöglichen, muss die IDT mit dem Assembler-Befehl *lidt* (Load IDT) initialisiert werden [16, p. 140f].

Im Eintrag 0x80 der IDT befindet sich die Funktion *system_call*, in der die Adresse der System-Call-Table aufgerufen wird. Um diese zu erhalten muss die Assembler-Instruktion *sidt* (Store IDT) aufgerufen werden, die die Adresse der IDT im Speicher zurück

doi: 10.2313/NET-2013-08-1_04

gibt. Anschließend muss eine Verschiebung der Adresse erfolgen, um den 128. (0x80) Eintrag bzw. die *system_call* Funktion zu erhalten. In dieser Funktion kann nach der Adresse der System-Call-Table gesucht werden [17, p. 233].

Die Abbildung 1 zeigt schematisch wie ein System-Call ausgeführt wird.

Im nächsten Punkt werden nun verschiedene Möglichkeiten gezeigt, wie ein Rootkit Schadcode mit Hilfe von System-Calls ausführen kann.

3.2 Hooking Methoden

Es gibt verschiedene Methoden, die von Rootkits genutzt werden, um Schadcode durch Sysem-Calls auszuführen (Hooking Methoden). Drei dieser Möglichkeiten werden hier genauer dargestellt.

3.2.1 IDT ersetzen (hooken)

Eine Methode, um Schadcode durch Rootkits auszuführen, ist das Ersetzen der IDT. Jede CPU besitzt ihre eigene IDT [18, p. 225].

Der Angriff eines Rootkits sieht wie folgt aus: Eine neue IDT wird erstellt und der Pointer des IDT Registers entsprechend umgeleitet, damit dieser auf die Adresse der Kopie zeigt. Anschließend kann die Kopie der IDT so manipuliert werden, dass Schadcode ausgeführt wird. Dadurch ist die Integrität der originalen IDT sichergestellt, falls ein Viren-Scanner diese überprüft [18, p. 225, 227].

3.2.2 System-Call-Table ersetzen (hooken)

Diese Art des Angriffs verwendet das „SuckIT"-Rootkit wie in Punkt 2.2.1 beschrieben. Dabei wird eine Kopie der System-Call-Table erstellt und der Pointer in der *system_call* Funktion aus der IDT von der originalen Adresse der System-Call-Table auf die Adresse der Kopie umgeleitet. Die Kopie kann nun so manipuliert werden, dass Schadcode ausgeführt wird. Die originale System-Call-Table bleibt unverändert, wodurch eine Integritätsprüfung erfolgreich verläuft. Sollte ein Viren-Scanner die IDT auf Integrität prüfen wird der Angriff festgestellt, da diese verändert wurde.

3.2.3 System-Call-Handler ersetzen (hooken)

Bei dieser Methode wird die System-Call-Table direkt modifiziert. Die System-Call-Table ist eine Liste, die die System-Call-Nummern (Index) zu den entsprechenden System-Call-Handlern abbildet [10].

Bei diesem Angriff wird ein eigener System-Call-Handler, der den Schadcode beinhaltet, geschrieben. Der Pointer auf die Adresse des originalen System-Call-Handler aus der System-Call-Table wird auf die Adresse des selbst erstellten System-Call-Handler umgeleitet. Diese Methode wird unter anderem vom Rootkit „knark" (in Punkt 2.2.2 beschrieben) angewandt [4, p. 4].

Damit die System-Call-Table verändert werden kann, wird ihre Adresse benötigt. Seit der Linux-Kernel-Version 2.6 wird diese nicht mehr exportiert [1, p. 82, 144] [19]; somit muss eine andere Methode angewandt werden, um die Adresse auszulesen. Auf zwei dieser Methoden wird nun näher eingegangen:

- Statisch: Die Adresse der System-Call-Table kann aus der Datei „/boot/System.map-Kernel-Version" ausgelesen werden [1, p 71f].
- Dynamisch: Die Adresse kann durch einen Brute-Force-Scan des komplett reservierten Kernel-Speichers gefunden werden. In einer 32 Bit-Architektur liegt der Speicher des Kernels zwischen 0xc0000000 und 0xd0000000. Die System-Call-Table wird zwar bei Linux 2.6 nicht mehr exportiert, dafür aber andere System-Calls (z.B. *sys_close*). Die Adresse dieser System-Calls kann mit dem reservierten Kernel-Speicher verglichen und so die Adresse der System-Call-Table ausgelesen werden [20].
  Das Listing 2 zeigt diese Methode:

```
static unsigned long** find_syscall_table(void) {
        unsigned long **sctable;
        unsigned long i;
        for(i = 0xc0000000; i < 0xd0000000; i++) {
                sctable = (unsigned long **)i;
                if (sctable[__NR_close] == (unsigned long *)
                sys_close) {
                        return &sctable[0];
                }
        }
        return NULL;
}
```

Listing 2: C-Funktion, die die Adresse der System-Call-Table findet.

In Abschnitt 4 werden einige Funktionen (inkl. Code-Beispiele) eines Rootkits beschrieben. Bei diesem Rootkit wird die Methode „System-Call-Handler ersetzen" eingesetzt.

4. Funktionalität eines selbst entwickelten Rootkits[1]

Dieser Abschnitt befasst sich mit gängigen Rootkit-Funktionen am Beispiel eines selbst programmierten Rootkits. Das Rootkit wurde als *LKM* mit der Linux-Kernel-Version 2.6.32 entwickelt. Die Funktionen werden neben einer Beschreibung teilweise auch anhand von Code-Beispielen erklärt.

Bei allen Funktionen, die einen System-Call benötigen, wurde die gleiche Methode angewandt. Es wurde ein neuer System-Call-Handler geschrieben, wofür zuerst die System-Call-Table benötigt wurde. Wie bereits erwähnt, wird diese für die Linux-Kernel-Version 2.6.x nicht mehr exportiert. Wie man die Adresse der System-Call-Table auslesen kann, wurde in Punkt 3.2.3 beschrieben. Außerdem ist die System-Call-Table durch den Kernel schreibgeschützt; somit muss zuerst das Protected Mode-Bit des CR0 CPU Registers umgeschrieben werden, damit die System-Call-Table verändert werden kann. Das Bit 0 des CR0 CPU-Registers wird auch als WP-Bit (write protected) bezeichnet. Wenn das WP-Bit auf 1 gesetzt ist, ist die CPU in einem „write-protected"-Modus; ist der Wert auf 0, dann ist die CPU im „read/write"-Modus. Dies bedeutet, dass wenn das Bit auf 0 gesetzt ist, das Programm Schreib-Zugriff auf die Memory-Pages (inkl. System-Call-Table) hat [21].

Die Funktion *read_cr0* liest den Wert des CR0 Registers und die Funktion *write_cr0* setzt das Bit des Registers auf den übergebenen Parameter. Der schreibgeschützte Modus kann ausgeschaltet

[1] Das hier behandelte Rootkit wurde im WS 2012/13 im Rahmen der Lehrveranstaltung „Rootkit Programming" an der TU München von Christian Eckert und Clemens Paul entwickelt.

doi: 10.2313/NET-2013-08-1_04

werden, indem zuerst das CR0 Bit gelesen und anschließend eine UND-Operation auf den negierten Wert 0x10000 angewandt wird. Um den Schreibschutz wieder einzuschalten, wird ebenfalls zuerst das CR0 Bit gelesen und anschließend eine ODER-Operation mit dem Wert 0x10000 ausgeführt [21].

Mit dieser beschriebenen Methode kann der Schreibschutz der System-Call-Table aufgehoben werden und der Pointer auf einen System-Call-Handler zu einer vom Angreifer geschriebenen Funktion umgeleitet werden. Wie dies für den *read* System-Call aussieht, zeigt Listing 3.

```
// so the sys call table is not write protected anymore
write_cr0 (read_cr0 () & (~ 0x10000));

// save the origin pointer of the sys call table

original_read = (void*)syscall_table[__NR_read];

// set the pointer of the sys call table to the malicious function
syscall_table[__NR_read] = (int)new_read;

// so the sys call table is write protected again
write_cr0 (read_cr0 () | 0x10000);
```

Listing 3: Schreibschutz der System-Call-Table aufheben und den *read* System-Call hooken.

4.1 Verbergen von Dateien (File Hiding)

Um Dateien durch ein Rootkit zu verbergen, gibt es verschiedene Möglichkeiten, beispielsweise das Hooken des *getdents64* System-Calls, da Programme wie *ls* diesen System-Call verwenden, um eine Liste von Dateien zu erhalten. Mit dem Befehl *strace* wird eine Liste der verwendeten System-Calls eines Programms ausgegeben.

Bei diesem Angriff wird ein neuer System-Call-Handler vom Angreifer geschrieben, in dem zuerst der originale System-Call-Handler aufgerufen wird, um die angefragte Liste von Dateien zu erhalten. Diese Liste wird dann vom Schadcode bearbeitet, damit dem Anwender bestimmte Dateien verborgen werden.

4.2 Verbergen von Prozessen (Process Hiding)

Prozesse können in einem Linux-System u.a. mit den Programmen *ps*, *pstree*, *lsof* und *top* angezeigt werden. Für diesen Angriff eines Rootkits gibt es mehrere Methoden. Eine ähnelt der Methode aus Punkt 4.1. Dabei werden die System-Calls *getdents* – für die Programme *ps* und *top* – und *getdents64* – für die Programme *pstree* und *lsof* – bearbeitet. Diese Methode ist sehr oberflächlich und bewirkt nur, dass die Prozesse vor den genannten Programme verborgen werden.

Eine weitere Möglichkeit eines Angriffs ergibt sich aus der Tatsache, dass Programme, die Prozesse anzeigen wie *ps*, eine andere Prozess-Liste verwenden als der Task-Scheduler des Kernels. Jeder Pointer auf einen Prozess, der im System läuft, wird in einer verketteten Liste namens *all-tasks* abgespeichert. Die Datenstruktur zu dieser Liste ist im Kernel als *init_tasks->next_task* hinterlegt. Der erste Eintrag dieser Liste ist der erste vom System erstellte Prozess. Diese Liste wird von Programmen, die Prozesse anzeigen, genutzt, wohingegen der Scheduler eine zweite Liste namens *run-list* verwendet. Die Datenstruktur für diese Liste im Kernel ist *run_queue_head->next*.

Wenn nun ein Angreifer einen bestimmten Prozess vor dem Anwender verbergen möchte, muss nur der Prozess aus der *all-tasks*-Liste gelöscht werden. Damit ist der Prozess für den Anwender zwar unsichtbar, läuft im System aber weiter [22, p. 673].

Auf diesem Prinzip ist auch das Rootkit „Virus:W32/Alman.B“ aufgebaut [23].

4.3 Verbergen von Modulen (Module Hiding)

Für einen Angreifer ist eine der wichtigsten Funktionen eines *LKM*-Rootkits das Verbergen des Moduls vor dem Anwender. Dabei muss das Kernel-Modul aus zwei verschiedenen Listen gelöscht werden, damit es sowohl beim Befehl *lsmod* – verwendet die Datei */proc/modules* – als auch im Verzeichnis */sys/module* – ein Verzeichnis, das alle Module beinhaltet – nicht erscheint. Um das Modul nach dem Befehl *lsmod* nicht mehr angezeigt zu bekommen, muss die Funktion *list_del_init(struct list_head*entry)*, die in *linux/list.h* definiert ist, aufgerufen werden [21].

Seit der Linux-Kernel-Version 2.6 gibt es das *kobject* (Kernel-Object), das im *sysfs*-Dateisystem in */sys* gemountet wird [1, p 83]. Damit das Kernel-Modul auch aus dieser Liste gelöscht wird, muss die Funktion *kobject_del(struct kobject *kobj)*, die in *linux/kobject.h* definiert ist, aufgerufen werden. Listing 4 zeigt einen funktionierender Code für diesen Angriff.

```
list_del_init(&__this_module.list);

kobject_del(&__this_module.mkobj.kobj);
```

Listing 4: Zeigt wie ein Kernel-Modul verborgen werden kann.

4.4 Erhöhen von Zugriffsrechten (Privilege Escalation)

Bei diesem Angriff geht es darum, einem bestimmten Prozess Root-Rechte zu geben. Wie auch in Punkt 4.3 wird bei diesem Angriff kein System-Call benötigt.

Um einem Prozess Root-Rechte zu geben, müssen seine Credentials verändert werden. Da der Init-Prozess (Prozessnummer 1) bereits die notwendigen Credentials bzw. Rechte besitzt, können diese einfach kopiert werden. Um dies zu tun, muss zuerst auf die Datenstruktur des Init-Prozesses und des Prozesses, der Root-Rechte erhalten soll, zugegriffen werden. Diese Datenstruktur der Prozesse, die als *struct task_struct* definiert ist, kann durch eine Hilfsfunktion erhalten werden. Anschließend kann über die Datenstruktur auf die Credentials der Prozesse zugegriffen und dem Prozess können Root-Rechte zugeteilt werden.

In Listing 5 wird gezeigt, wie dieser Angriff aussehen kann.

```
struct task_struct *get_task_struct_by_pid(unsigned pid) {
        struct pid *proc_pid = find_vpid(pid);
        struct task_struct *task;
        if(!proc_pid)
                return 0;
        task = pid_task(proc_pid, PIDTYPE_PID);
        return task;
}
void escalate_root(int pid) {
        struct task_struct *task = get_task_struct_by_pid(pid);
```


doi: 10.2313/NET-2013-08-1_04

```
        // get the task_struct of the init process
        struct task_struct *init_task =
        get_task_struct_by_pid(1);
        if(!task || !init_task)
                return;
        task->cred = init_task->cred; //use the same credentials
        // as the ones of the init process
}
```

Listing 5: Zeigt zwei C-Funktionen mit denen die Zugriffsrechte eines Prozesses erhöht werden können.

5. VERWANDTE ARBEITEN

Während in dieser Arbeit der Fokus auf der Beschreibung von Rootkits und deren Anwendung liegt, wird in der Arbeit „Proactive Detection of Kernel-Mode Rootkits“ [24] von Bravo und Garcìa auf das Aufspüren von Rootkits im Detail eingegangen. Dabei wird eine Methode dargestellt, um Kernel-Mode-Rootkits frühzeitig aufzuspüren.

Die Arbeit „Detecting Kernel-Level Rootkits Using Data Structure Invariants“ [22] von Baliga, Ganapathy und Iftode befasst sich ebenfalls mit dem Aufspüren von Kernel-Mode-Rootkits. Dabei wurde ein Programm namens Gibraltar entwickelt, das sowohl modifizierte „control“- wie auch „noncontrol“-Datenstrukturen erkennt.

Eine ausführliche Erläuterung der in dieser Arbeit vorgestellten Methoden, wie ein Rootkit Schadcode ausführen kann (Punkt 3.2), wird technisch ausführlicher in den Arbeiten „The Research on Rootkit for Information System Classified Protection“ [15] und „Virus Analysis on IDT Hooks of Rootkits Trojan“ [18] beschrieben. In letzterer liegt der Schwerpunkt auf IDT-Hooking; dies wird auch anhand mehrerer Grafiken detailliert veranschaulicht.

6. AUSBLICK

Zum Abschluss der Arbeit werden Gefahren durch Rootkits anhand von aktuellen Beispielen aufgezeigt.

Der Artikel „LynuxWorks Debuts Industry's First Real-Time Zero-Day Rootkit and Bootkit Defense“ vom 26. Februar 2013 [29] beschreibt das System „LynxSecure 5.2“, das unter der Verwendung von Virtualisierungs-Technologie Rootkits und Bootkits in Echtzeit erkennen soll. Der Vizepräsident von Enterprise Security Solutions at LynuxWorks stellt fest, dass Rootkits und Bootkits nicht nur eine der gefährlichsten, sondern auch eine der am meistverbreiteten Gefahren sind. Zusätzlich sagt er, dass es derzeit nur wenige existierende Programme zum Aufspüren und Entfernen von Rootkits gibt und diese nur Teillösungen anbieten.

Bootkits sind eine weitere Art von Rootkit. Sie stellen eine noch größere Gefahr dar, da sie zum Einsatz kommen, bevor der Kernel des Betriebssystems geladen wird. Sie können Programme, die Malware aufspüren, unbrauchbar machen und selbst durch eine Formatierung der Festplatte nicht entfernt werden. Die Arbeit „An Overview of Bootkit Attacking Approaches“ [30] beschreibt verschiedene Arten von Bootkits, u.a. BIOS-Bootkits. Ein BIOS-Bootkit, das von IceLord geschrieben wurde, diente als Grundlage für den ersten derartigen Angriff im Jahr 2011. Webroot Researcher Marco Giuliani bezweifelt dagegen, dass solche Angriffe eine größere Gefahr darstellen, da diese BIOS-Rootkits nur für einen bestimmten Typ von BIOS funktionieren [31].

Durch die immer größer werdende Anzahl an Smartphone-Nutzern werden diese Systeme inzwischen ebenfalls zum Angriffsziel. Vor allem viele Nutzer von Android-Systemen aktualisieren ihr Gerät nicht, wodurch die Sicherheitslücken auf ihren Systemen nicht geschlossen werden können. [32] Bei der diesjährigen Black Hat Europe 2013-Konferenz hielt Thomas Roth einen Vortrag mit dem Titel „Next generation mobile rootkits“. Dabei referierte er über einen Angriff eines Rootkits auf die Trust-Zone von mobilen Geräten [33].

Die angeführten Beispiele verdeutlichen noch einmal die Gefahren, die Rootkits gegenwärtig darstellen. Die Auseinandersetzung mit ihrer Funktionweise und den sich daraus ableitenden Gegenmaßnahmen wird somit auch in Zukunft eine wichtige Aufgabe im Rahmen der IT-Sicherheit sein.

7. LITERATUR

[1] Silez Peláez, R., Linux kernel rootkits: protecting the system's "Ring-Zero". SANS Institute, 2004.

[2] Geer, D., „Hackers get to the root of the problem”. IEEE Journals & Magazines, vol. 39, 5, 17-19, 2006. DOI= 10.1109/MC.2006.163

[3] Kim, G. H. und Spafford, E. H, „The design and implementation of Tripwire: a file system integrity checker,“ in Proc. the 2nd ACM Conference on Computer and Communications Security, 18-29, 1994.

[4] Levine, J. G. , Grizzard, J. B., Hutto, P. W. und Owen, H. W L, III, „A Methodology to Characterize Kernel Level Rootkit Exploits that Overwrite the System Call Table“. IEEE, SoutheastCon, 25-31, 2004. DOI= 10.1109/SECON.2004.1287894

[5] http://www.rootkitanalytics.com/kernelland/. Aufgerufen am 24. März 2013.

[6] Wichmann, R., Linux Kernel Rootkits. 2006. http://www.lasamhna.de/library/rootkits/basics.html. Aufgerufen am 24. März 2013.

[7] Levine, John G., Grizzard, Julian B. und Owen, Henry W L, III, „A methodology to detect and characterize Kernel level rootkit exploits involving redirection of the system call table“. IEEE, Information Assurance Workshop, 107-125, 2004. DOI=10.1109/IWIA.2004.1288042

[8] sd und devik, „Linux on-the-fly kernel patching without LKM“. Phrack, vol. 0x0b, 0x3a. 2001. http://www.phrack.org/issues.html?id=7&issue=58. Aufgerufen am 24. März 2013.

[9] Salzman, P. Burian, M. und Pomerantz, Ori, The Linux Kernel Module Programming Guide. Mai 2007. http://www.tldp.org/LDP/lkmpg/2.6/lkmpg.pdf. Aufgerufen am 24. März 2013.

[10] pragmatic / THC, (nearly) Complete Linux Loadable Kernel Modules. März 1999. http://www.thc.org/papers/LKM_HACKING.html. Aufgerufen am 24. März 2013.

[11] truff, „Infecting loadable kernel modules“. Phrack, vol. 0x0b, 0x3d. August 2013. http://www.phrack.org/issues.html?issue=61&id=10. Aufgerufen am 24. März 2013.

[12] jbtzhm, „Static Kernel Patching“. Phrack, vol. 0x0b, 0x3c. Dezember 2002. http://www.phrack.org/issues.html?issue=60&id=8. Aufgerufen am 24. März 2013.

[13] Wichmann, R. Linux Kernel Rootkits. http://www.la-samhna.de/library/rootkits/list.html. 2006 Aufgerufen am 24. März 2013.

[14] Information about the Knark Rootkit. http://www.ossec.net/doc/rootcheck/rootcheck-knark.html. Aufgerufen am 24. März 2013.

[15] Zhi-hong, T., Bai-ling, W., Zixi, Z. und Hong-Li Li, Z., „The Research on Rootkit for Information System Classified Protection“. International Conference on Computer Science and Service System (CSSS), 890-893, 2011. DOI= 10.1109/CSSS.2011.5974667

[16] Bovet, D. P. und Cesati, M., Understanding the Linux Kernel. O'Reilly Media, Auflage: 3rd ed, 2006.

[17] Zhao, K., Li, Q., Kang, J., Jiang, D. und Hu, L., „Design and Implementation of Secure Auditing System in Linux Kernel“. IEEE International Workshop on Anti-counterfeiting, Security, Identification, 232-236, 2007. DOI=10.1109/IWASID.2007.373733

[18] Wang, Y., Gu, D., Li, W., Li, J. und Wen, M., „Virus Analysis on IDT Hooks of Rootkits Trojan“. IEEC '09. International Symposium on Information Engineering and Electronic Commerce, 224-228, 2009. DOI= 10.1109/IEEC.2009.52

[19] Cooperstein, J., Vanishing Features of the 2.6 Kernel. Dezember 2012. http://www.linuxdevcenter.com/lpt/a/2996. Aufgerufen am 24. März 2013.

[20] memset, Syscall Hijacking: Dynamically obtain syscall table address (kernel 2.6.x) #2. März 2011. http://memset.wordpress.com/2011/03/18/syscall-hijacking-dynamically-obtain-syscall-table-address-kernel-2-6-x-2/. Aufgerufen am 24. März 2013.

[21] memset, Syscall Hijacking: Kernel 2.6.* systems. Dezember 2010. http://memset.wordpress.com/2010/12/03/syscall-hijacking-kernel-2-6-systems/. Aufgerufen am 24. März 2013.

[22] Baliga, A., Ganapathy, V. und Iftode, L., „Detecting Kernel-Level Rootkits Using Data Structure Invariants“. IEEE Transactions on Dependable and Secure Computing, vol. 8, 5, 670-684, 2011. DOI=10.1109/TDSC.2010.38

[23] Virus:W32/Alman.B. http://www.f-secure.com/v-descs/fu.shtml. Aufgerufen am 24. März 2013.

[24] Bravo, P. und Garcìa, D. F., „Proactive Detection of Kernel-Mode Rootkits“. Sixth International Conference on Availability, Reliability and Security (ARES), 515-520, 2011. DOI=10.1109/ARES.2011.78

[25] Baliga A et al., „Automated containment of rootkits attacks“. Comput. Secur., 2008. DOI= 10.1016/j.cose.2008.06.003

[26] http://www.gnu.org/copyleft/gpl.html. Aufgerufen am 24. März 2013.

[27] http://www.rootkitanalytics.com/userland/. Aufgerufen am 24. März 2013.

[28] Quade, J., Kernel Rootkits and Countermeasures. http://www.linux-magazine.com/Online/Features/Kernel-Rootkits-and-Countermeasures. Aufgerufen am 24. März 2013.

[29] „LynuxWorks Debuts Industry's First Real-Time Zero-Day Rootkit and Bootkit Defense“. Marketwire, Februar 2013. http://apache.sys-con.com/node/2554490. Aufgerufen am 25. März 2013.

[30] Li, X., Wen, Y., Huang, M. und Liu, Q., „An Overview of Bootkit Attacking Approaches“. Seventh International Conference on Mobile Ad-hoc and Sensor Networks (MSN), 428-231, 2011. DOI= 10.1109/MSN.2011.19

[31] Kaplan, D., „Researchers uncover first active BIOS rootkit attack“. SC Magazine, September 2011. http://www.scmagazine.com/researchers-uncover-first-active-bios-rootkit-attack/article/212035/. Aufgerufen am 25. März 2011.

[32] Platform Versions. http://developer.android.com/about/dashboards/index.html. Aufgerufen am 25. März 2013.

[33] Thomas R., Next generation mobile rootkits. Black Hat Europe, März 2013.

[34] ghalen and wowie, „Developing Mac OSX kernel rootkits“. Phrack, vol. 0x0d, 0x42, Juni 2009. http://www.phrack.org/issues.html?issue=66&id=16. Aufgerufen am 25. März 2013.

[35] Hannel, J., „Linux RootKits For Beginners – From Prevention to Removal“. Januar 2003. http://www.sans.org/reading_room/whitepapers/linux/linux-rootkits-beginners-prevention-removal_901. Aufgerufen am 19. April 2013.