Rootkits

Stefan Liebald
Betreuer: Simon Stauber
Seminar Innovative Internet Technologien und Mobilkommunikation WS2012
Lehrstuhl Netzarchitekturen und Netzdienste
Fakultät für Informatik, Technische Universität München
Email: liebald@in.tum.de

KURZFASSUNG

Rootkits sind eine Sammlung von Softwaretools, welche es Außenstehenden erlauben, unerkannt die Kontrolle über ein Computersystem zu behalten und dieses zu manipulieren, ohne dass legitimierte Nutzer in der Lage sind dies zu erkennen. Gerade Schadsoftware verwendet Rootkittechnologie zur Tarnung, diese sind also eine allgegenwärtige Gefahr.
Dieses Paper dient dazu, dem Leser die Eigenschaften und Gefahren von Rootkits näher zu bringen, so dass dieser in der Lage ist, die Grundlegenden Konzepte zu verstehen. So wird auf zwei verschiedene Rootkit-Typen eingegangen und speziell Kernel-Mode Rootkits einer genaueren Betrachtung unterzogen. Abschließend wird erläutert auf welche Weise ein Rootkit das Betriebssystem manipulieren kann und welche Möglichkeiten zu Abwehr und Erkennung existieren.

Schlüsselworte

User-Mode Rootkits, Kernel-Mode Rootkits, LKM, Linux Kernel, Kernel Manipulation

1. EINLEITUNG

Angriffe auf Rechensysteme sind eine allgegenwärtige Gefahr. Neben dem Ausspionieren von sensitiven Daten (wie z.B. Kreditkartendaten) wollen Angreifer den Zugriff auf ein System oft für eine längere Zeit behalten (z.B. für Botnetze). Hierzu versuchen sie die Kompromittierung des Rechners vor dem Benutzer zu verschleiern um Gegenmaßnahmen zu vermeiden.
Ein Mittel hierzu stellen Rootkits dar. Diese können, nach dem erstmaligen Erlangen von Rootrechten, auf einem System platziert werden und gewähren dem Angreifer ab diesem Zeitpunkt fortgesetzten privilegierten Zugriff durch Schaffung einer Hintertür. Des Weiteren sind sie in der Lage sich selbst und andere Schadsoftware vor Entdeckung zu schützen indem sie sich im System verstecken.
Ein prominentes Beispiel für den Einsatz von Rootkits stammt aus dem Jahr 2005. Das Musiklabel “Sony BMG” verwendete heimlich ein Rootkit als Teil seines Kopierschutz [15]. Musik CDs mit diesem “Schutz” wurden millionenfach verkauft und von den Käufern auf dem PC abgespielt, wodurch das Rootkit installiert wurde, welches auch durch andere Schadsoftware verwendet werden konnte.
Dieses Beispiel zeigt, Rootkits sind nicht nur eine theoretische Gefahr für die Sicherheit von Computersystemen, sondern auch eine ganz reale. Dieses Paper soll einen Überblick über Geschichte und Funktion von Rootkits vermitteln sowie einige Schutzmöglichkeiten aufzeigen. Hierbei liegt der Fokus auf Unix Rootkits.
Dazu gibt Abschnitt 2 eine Definition von Rootkits, sowie einem Überblick über ihre Entwicklung und erläutert skizzenhaft verschiedene Rootkit-Typen. Abschnitt 3 zeigt beispielhaft die Funktionsweise eines Kernel-Mode Rootkits, erläutert wie dieses im Kernel integriert werden kann und zeigt eine Möglichkeit diesen zu manipulieren. Abschnitt 4 stellt Optionen zur Erkennung und Abwehr von Rootkits vor. In Abschnitt 5 endet dieses Paper mit einem kurzen Ausblick.

2. HINTERGRUND

Um sich dem Thema Rootkits anzunähern, wird im folgenden Abschnitt eine Definition von Rootkits vorgestellt. Anschließend wird ein Blick auf die Historie der Rootkits geworfen und auf deren Taxonomie eingegangen.

2.1 Definition

Da Rootkits je nach Umfang und Zweck verschiedene Funktionen eines Betriebssystems beeinflussen können ist eine präzise Definition schwer zu finden. Hoglund [11] verwendet folgende grundlegende Definition der Eigenschaften eines Rootkits:

“Ein Rootkit ist dabei ein ’Kit’ aus kleinen, nützlichen Programmen, die einem Angreifer den fortgesetzten Zugriff auf root erlauben, dem mächtigsten Benutzer auf einem Computer.[. . .] Ein Rootkit ist ein Satz von Programmen und Code, der eine dauerhafte und nicht aufzuspürende Präzens auf einem Computer erlauben.”

Ein einmal integriertes Rootkit ist in der Lage sich selbst und andere Software vor dem rechtmäßigen Administrator eines Systems zu verbergen. Im Idealfall ist sich der Benutzer der Kompromittierung nicht bewusst. Nach einer Infizierung des Systems kann ein Rootkit (je nach Umfang) theoretisch alle Interaktion eines Benutzers mit diesem abfangen, kontrollieren und gegebenenfalls manipulieren. In vielen Fällen bietet ein Rootkit außerdem eine Hintertür zum System, mittels derer ein Angreifer wiederholt versteckten Rootzugriff erlangen kann. Ein Rootkit dient **nicht** dem erstmaligem Erlangen von Rootrechten, welche nötig sind um ein Rootkit im System zu integrieren. Diese Rechte müssen vor der Integrierung des Rootkits auf andere Art erlangt werden (z.B. mittels Schadsoftware, Social Engineering).
Aber nicht nur Schadsoftware verwendet Rootkits, sondern auch hostbasierte Überwachungssysteme zum Schutz

doi: 10.2312/NET-2013-02-1_08

eines Computers vor Angreifern können diese Software verwenden.

2.2 Geschichte

Erste Software mit der Eigenschaft zur Tarnung trat erstmals in den späten 1980er Jahren auf. Einer der ersten Viren mit Tarnfunktionen war der Brain Virus für DOS [1], welcher sich im Bootsektor eines im DOS FAT formatierten Speichermediums eingenistet hat. Wann immer versucht wurde aus dem infizierten Boot Sektor zu lesen, hat Brain diesen Zugriff umgeleitet und den originalen Boot Sektor vorgezeigt.
Kurze Zeit später traten die ersten Rootkits auf UNIX Systemen auf. Diese Rootkits ersetzten Benutzerprogramme (z.B. *login* oder *ps*) durch eigene Varianten und sorgten dafür, dass Informationen über ihre Tätigkeit aus den Logs entfernt wurden. Eine erweiterte Funktionalität war das Ersetzen von Bibliotheken, welche von den Benutzerprogrammen verwendet wurden. Diese wurden dadurch indirekt beeinflusst, blieben selbst aber unverändert.
Da ersetzte Dateien relativ leicht aufspürbar waren, traten in den späten 90er Jahren erste Rootkits auf, die den Betriebssystemkern (Kernel) beeinflussten. Auf diese Weise waren sie in der Lage Schutzmöglichkeiten zu unterlaufen und waren auf diese Weise noch schwerer aufzuspüren. In dieser Zeit kamen schließlich auch die ersten Windows Rootkits in größerem Maße in Umlauf.

2.3 Taxonomie

Grundsätzlich existieren zwei verbreitete Typen von Rootkits, die User- und die Kernel-Mode Rootkits. Zur Einordnung werden die verschiedenen Berechtigungsstufen eines Systems betrachtet. Die meisten Prozessoren implementieren Privilegierungslevel, die nutzbare Prozessorinstruktionen oder den Zugriff auf bestimmte Speicherbereiche für einen Prozess einschränken. Oft werden diese Level als Ring 0-3 bezeichnet, wobei Ring 3 auch als User-Mode bezeichnet wird. Normale Benutzerprogramme laufen in diesem Modus. Hier gibt es Einschränkungen auf den Zugriff auf Hardware und Speicher, es darf nur auf den zugewiesenen (virtuellen) Speicherbereich zugegriffen werden und nicht auf Speicherbereiche anderer Programme. Auch der Zugriff auf den Kernel ist beschränkt.
Ring 1 und 2 werden nur in seltenen Fällen benutzt, Ring 0 entspricht der höchsten Privilegierungstufe, in welcher keine Einschränkungen gelten und in welchem der Kernel eines Betriebssystems läuft.

Abbildung 1 zeigt ein grundlegendes Modell eines Computersystems. Das Betriebssystem läuft auf der Hardware und verwaltet Zugriffe auf sie über den Kernel. Im Betriebssystem ausgeführte Programme laufen im User-Mode und können dabei auf Kernel Funktionen zugreifen. Ein Rootkit im User-Mode ersetzt diese Programmen oder Bibliotheken, welche von diesen genutzt werden, um das System zu manipulieren. Dabei könnte beispielsweise *ps* dahingehend geändert werden, dass es bei einem Aufruf vom Rootkit angestoßene Prozesse nicht länger auflistet, um diese zu verstecken.

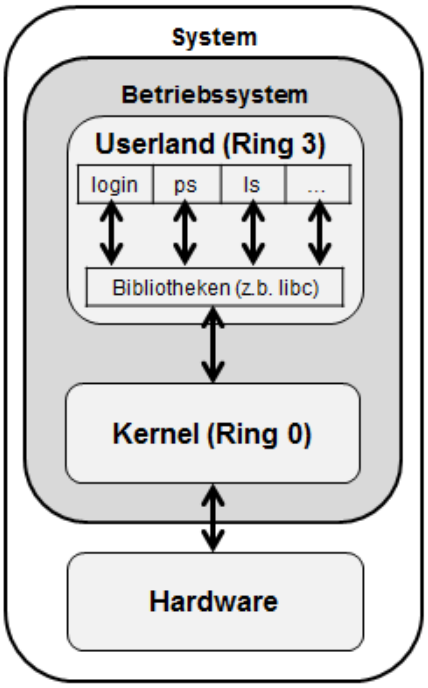

Abbildung 1: Schematische Darstellung der grundlegenden Struktur eines Systems

Ein Beispiel für ein User-Mode Rootkit[1] ist "Jynx2"[3], welches bestimmte Funktionsaufrufe aus Bibliotheken ersetzt, indem es eigene Bibliotheken vor den normalen Bibliotheken lädt ("LD_PRELOAD[2]"). Wird auf diese Art beispielsweise eine eigene Bibliothek mit der Funktion *printf()* vor der Standard C Bibliothek *libc* geladen, so wird die zuerst geladene *printf()* Funktion aus der eigenen Bibliothek verwendet. Nach der Installation versteckt Jynx2 unter anderem alle Dateien und Prozesse, die einer bestimmten Benutzer ID gehören oder deren Name mit einem vorher definierten String beginnt (standardmäßig "XxJynx"). Listing 1 zeigt die Auswirkung beispielhaft:

Listing 1: Verstecken eines Verzeichnisses durch Jynx2

```
1 # ls
2 datei1.txt datei2.txt
3 # mkdir XxJynx_Rootkit
4 # ls
5 datei1.txt datei2.txt
```

Das neu erstellte Verzeichnis *XxJynx_Rootkit* wird durch den Befehl *ls* nicht angezeigt, da es mit dem String "XxJynx" beginnt und deshalb durch das Rootkit verborgen wird.
Die Vielzahl der zu ersetzenden Dateien um alle Benutzerprogramme abzudecken, war einer der Gründe

[1] erschienen Mitte 2012, getestet unter Ubuntu 12.10
[2] LD_PRELOAD ist eine Umgebungsvariable, die eine Liste von Bibliotheksadressen (z.b. "/prelib/libc.so.6") enthält, die beim Starten von Programmen vor allen anderen Bibliotheken geladen werden

doi: 10.2312/NET-2013-02-1_08

für die Entwicklung von Kernel-Mode Rootkits. Wie in Abbildung 1 zu sehen ist, operiert der Kernel in der höchsten Privilegierungsstufe (Ring 0) und regelt den Zugriff von Userland Programmen auf die Hardware. Ein Kernel-Mode Rootkit kann an dieser Stelle ansetzen und das Verhalten des Kernels manipulieren, so dass zwar die Userland Programme wie *ls* nicht ausgetauscht werden und unverdächtig wirken, die von ihnen oder einer eingebundenen Bibliothek verwendeten Kernelfunktionalitäten aber manipuliert sind. Auf diese Weise lassen sich verschiedene Benutzerprogramme zentral manipulieren, ohne diese direkt zu ersetzen.
Ein Beispiel zur Funktionalität und Installation von Kernel-Mode Rootkits wird in Abschnitt 3 beschrieben.

3. KERNEL-MODE ROOTKITS

In diesem Abschnitt werden Kernel-Mode Rootkits näher untersucht. Dazu werden Möglichkeiten aufgezeigt, wie ein derartiges Rootkit in den Kernel eingeschleust werden kann. Hierfür wird die gebräuchliche Vorgehensweise mittels Kernelmodulen untersucht. Im Anschluss wird die Funktionsweise eines Kernel-Mode Rootkits anhand eines Beispiels genauer erläutert.

3.1 Integration in den Kernel

Um ein Rootkit in den Kernel zu bringen, stehen mehrere Möglichkeiten zur Verfügung [16], von denen je nach Kernelversion allerdings nicht immer alle verwendet werden können (siehe Abschnitt 4.1):

- Modifizieren des Abbilds des Kernels im Hauptspeicher (repräsentiert durch /dev/kmem).
- Austausch des Kernelabbilds auf der Festplatte durch eine eigene Version (/boot/vmlinuz).
- Ausführen des Rootkits im User Mode mit Kernel Mode Privilegien (Unterstützung durch Kernel erforderlich, z.B. Kernel Mode Linux [13]).
- Integration in den Kernel durch Kernelmodule (engl.: Loadable Kernel Modules, LKM).

Im Folgenden soll die gebräuchliche Methode der Integration von Kernel-Rootkits in den Kernel mithilfe von LKMs detaillierter betrachtet werden. LKMs sind als Binärdatei gespeichert und erweitern den Kernel um zusätzliche Funktionalitäten (z.B. Gerätetreiber). Sie können während dem laufenden Betrieb in den Kernel mit aufgenommen und von diesem benutzt werden. Der Code wird in der selben Privilegierungsstufe wie der Kernel ausgeführt (Ring 0). Mit LKMs ist es möglich, tiefergreifende Veränderungen im Betriebssystemkern zu bewirken, ohne das System dabei neustarten zu müssen. Der Code eines LKMs muss dabei zwei bestimmte Methoden beinhalten, *init_module()*, welche ausgeführt wird wenn das Modul in den Kernel geladen wird und *cleanup_module()*, welche beim Entladen aus dem Kernel aufgerufen wird. Mittels dieser Aufrufe integriert sich das Modul in den Kernel, so das diesem die neuen Funktionalitäten zur verfügung stehen (siehe Abschnitt 3.3).

3.2 Systemaufrufe

Zum Verständnis der Funktionsweise von Kernel-Mode Rootkits, ist es nötig zu verstehen, wie die für Rootkits relevanten Teile des Kernels funktionieren, beziehungsweise welche Aufgaben der Kernel für gewöhnliche Userland Programme ausführt.
Funktionen des Kernels sind unter anderem Speicher-, Prozess- und Dateisystem Management, sowie die Verwaltung von Gerätetreibern und des Netzwerkzugriffs. Wenn ein Benutzerprogramm Zugriff auf einen vom Kernel verwalteten Bereich benötigt (z.B. zum Versenden eines Netzwerkpakets), muss es auf entsprechende Funktionen des Kernels zugreifen, welche die Anfrage dann behandeln.
Für die Kommunikation mit dem Kernel stehen, je nach Anfrage verschiedene, Systemaufrufe zur Verfügung. Will ein Benutzerprogramm beispielsweise eine Datei öffnen, so muss es dazu, unter anderem, den *sys_open()* Systemaufruf des Kernels nutzen. Im Normalfall werden für diesen Zugriff von Benutzerprogrammen auf den Kernel Funktionen von Bibliotheken im Userland benutzt, wie beispielsweise *fopen()* aus der Standard C Bibliothek *libc*. Die Bibliotheken rufen dann ihrerseits den Systemaufruf auf.
In der Kernelversion 3.5.0 gibt es über 270 verschiedene Systemaufrufe. Die Speicheradressen dieser Funktionen sind in der sogenannten Systemaufrufstabelle abgelegt und werden über diese verwaltet.
Kernel Rootkits basieren darauf, auf diese Tabelle zuzugreifen und bestimmte Systemaufrufe (je nach Ziel des Angreifers) durch eigene, angepasste Varianten zu ersetzen oder neue Systemaufrufe zu erstellen, welche durch Userland Schadsoftware genutzt werden können (siehe Abschnitt 3.3.2).

3.3 LKM Kernel-Mode Rootkits

Im folgenden Abschnitt wird gezeigt, wie ein Rootkit als Kernelmodul in den Kernel geladen wird und dort Systemaufrufe manipulieren kann.

3.3.1 Zugriff auf die Systemaufrufstabelle

Um Systemaufrufe zu beeinflussen, benötigt ein Kernel-Mode Rootkit Zugriff auf die Systemaufrufstabelle. Um an die Adresse der Tabelle im Speicher zu gelangen, kann aus */boot/System.map-Kernelversion* die Speicheradresse ausgelesen werden an welcher sich die Tabelle befindet, um sie dann im Rootkit direkt zu adressieren.
Abbildung 2 zeigt den entsprechenden Eintrag in der

```
c15d501f r __func__.22707
c15d5028 r print_trace_ops
c15d5040 R sys_call_table
c15d55e0 r k8_nops
c15d5620 r p6_nops
c15d5660 r intel_nops
```

Abbildung 2: Adresse der Systemaufrufstabelle in System.map

System.map. Die Adresse der Systemaufrufstabelle unterscheidet sich je nach Kernelversion. Der linke Eintrag stellt die Adresse im Speicher dar, der rechte gibt den Symbolnamen der Systemvariable an, welche

doi: 10.2312/NET-2013-02-1_08

sich an dieser Stelle befindet (in diesem Fall sys_call_table als Symbolname für die Systemaufrufstabelle). Das R weist darauf hin, dass der Speicherbereich, in dem die Tabelle liegt, schreibgeschützt (read-only) ist. Um die Tabelle manipulieren zu können, muss dieser Schutz aufgehoben werden, wie es beispielhaft in Abschnitt 3.3.2 gezeigt wird. Ein funktionsfähiges Beispiel eines Rootkits, welches Systemaufrufe ersetzt, findet sich auch im Blog von Styx [19]. Dieser zeigt auch eine Möglichkeit wie ein Rootkit so erweitert werden kann, dass es die Adresse der Systemaufrufstabelle dynamisch während dem Kompilieren bzw. zur Laufzeit aus der System.map auslesen kann [18]. Auf diese Art und Weise ist es nicht länger nötig den Speicherort der Tabelle für verschiedene Systeme manuell zu suchen.

3.3.2 Ersetzen eines Systemaufrufs

Nachdem man Zugriff auf die Systemaufrufstabelle besitzt, kann man verschiedene Systemaufrufe durch eigene Varianten ersetzen. Im Folgenden soll die Funktionalität eines Kernel-Mode Rootkits anhand der wichtigsten Codeausschnitte eines einfachen Beispiels gezeigt werden. Das Rootkit soll den Systemaufruf *sys_nanosleep()* durch eine eigene Variante ersetze. Normalerweise wird der Systemaufruf beispielsweise von der Funktion sleep() verwendet, welche die Ausführung eines Programms um eine angegebene Zeit verzögert. Die Rootkitversion des Aurufs soll nun bei einer bestimmten zu wartenden Zeit anstatt zu warten dem aufrufenden Prozess Rootrechte verleihen. Listing 2 zeigt den Code der zur Ausführung kommt wenn das Rootkit als Modul in den Kernel geladen wird.

Listing 2: init Funktion des Rootkit Moduls

```
1  unsigned long *syscall_table = (unsigned long *)0xc15d5040;
2  asmlinkage int (*original_nanosleep)(struct timespec *, struct timespec *);
3
4  static int init(void) {
5    write_cr0(read_cr0() & (~ 0x10000));
6
7    original_nanosleep = (void *)syscall_table[__NR_nanosleep];
8    syscall_table[__NR_nanosleep] = new_nanosleep;
9
10   write_cr0(read_cr0() | 0x10000);
11   return 0;
12 }
```

In Zeile 1 wird die Systemaufrufstabelle anhand ihrer vorher aus /boot/System.map ausgelesenen Position im Speicher adressiert, so dass sie im Weiteren manipuliert werden kann. Ab Zeile 4 ist der Code zu sehen, der beim Laden des Moduls in den Kernel ausgeführt wird (vgl. Abschnitt 3.1). In Zeile 5 wird zuerst durch die Funktion *write_cr0()* der Schreibschutz des Speicherbereichs aufgehoben, in dem auch die Systemaufrufstabelle liegt. Dies geschieht durch Setzen des 17. Bits im dafür verantwortlichen Register CR0 auf 0 [12]. Anschließend wird die Originaladresse des zu ersetzenden Systemaufrufs (*sys_nanosleep()*) gesichert und der entsprechende Eintrag durch die Adresse der Rootkit Funktion *new_nanosleep()* ersetzt (Zeile 8), welche ebenfalls als Code im Modul vorliegt. Abschließend wird der Schreibschutz des Speicherbereichs wieder reaktiviert.

Der neue Systemaufruf des Rootkits *new_nanosleep()* ist in Listing 3 zu sehen. Immer wenn der ursprüngliche Systemaufruf angefragt wird, springt stattdessen der Rootkit Code ein.

Listing 3: Rootkitversion des ersetzten Systemaufrufs

```
1  asmlinkage int new_nanosleep(struct timespec *req , struct timespec *rem){
2    struct cred *new;
3    if (req->tv_sec==1234){
4      new = prepare_creds();
5      if ( new != NULL ) {
6         new->uid = new->gid = 0;
7         new->euid = new->egid = 0;
8         new->suid = new->sgid = 0;
9         new->fsuid = new->fsgid = 0;
10        commit_creds(new);
11        }
12       return 0;
13     }
14   else {
15     return original_nanosleep(req, rem);
16   }
17 }
```

Dieser überprüft zuerst wie lange *sys_nanosleep()* schlafen soll (Zeile 3) und verleiht dem aufrufenden Prozess Rootrechte (Zeilen 2,4,6-10) wenn die Wartezeit 1234 Sekunden beträgt. Dazu werden die Werte der Benutzer-ID[3] (=uid) und Gruppen-ID[4] (=gid) auf den Wert 0 gesetzt, was intern dem Root Benutzer entspricht[5]. Für alle anderen Zeiten wird in Zeile 15 der originale Systemaufruf aufgerufen.
Danach besitzt der Prozess der den Systemaufruf ausgeführt hat Rootrechte und kann im Folgenden mit diesen operieren, auch wenn nie das Passwort des eigentlichen Rootbenutzers eingegeben werden musste.
Listing 4 zeigt den C Code für ein (Userland) Programm, bei dessen Ausführung (bei installiertem Rootkit) eine Shell mit Rootzugriff gestartet wird ohne dass eine Authentisierung des Benutzers nötig ist[6].

Listing 4: Programm zum Testen des Rootkits

```
1 int main () {
2  sleep(1234);
3  system ("/bin/sh");
4  return 0;
5 }
```

Oft werden (vor allem in der Forschung) Kernelrootkits zusammen mit einem Userland Steuerprogramm genutzt, mit Hilfe dessen z.B. Funktionen des Rootkits dynamisch (de-)aktiviert werden können. Dieses Steuerprogramm könnte, wie in Listing 4 gezeigt, mittels des *sys_nanosleep()* Systemaufrufs arbeiten und ungenutzte Signale[7] zur Steuerung verwenden, die das Kernelrootkit abfängt. Ein Kernel-Mode Rootkit, welches den Systemaufruf *sys_kill()* nutzt ist KNARK [14] (für die Kernelversionen 2.2 und 2.4), welches standardmäßig mittels *killall -31* die Anwesenheit bestimmter Prozesse versteckt und sie nach dem Kommando *killall -32* wieder anzeigt.

Abbildung 3 verdeutlicht die Funktionsweise des Rootkits und zeigt wie es eine eigene Funktion vor dem eigentlichen Systemaufruf platziert. Alle weiteren Aufrufe laufen nun

[3] repräsentiert intern den Benutzer anstelle des Benutzernamens
[4] Ein Benutzer gehört mindestens einer Gruppe an, welche eine Rolle bei seinen Zugriffsberechtigungen spielt
[5] Die Varianten euid,egidm,suid,sgid, fsuid und fsgid sind zum Verständniss von Rootkits hier nicht weiter relevant
[6] vollständiges Rootkit getestet unter Kernelversion 3.5.0
[7] kurze Nachrichten an Prozesse die bestimmte Reaktionen auslösen sollen und über den Kernel laufen

doi: 10.2312/NET-2013-02-1_08

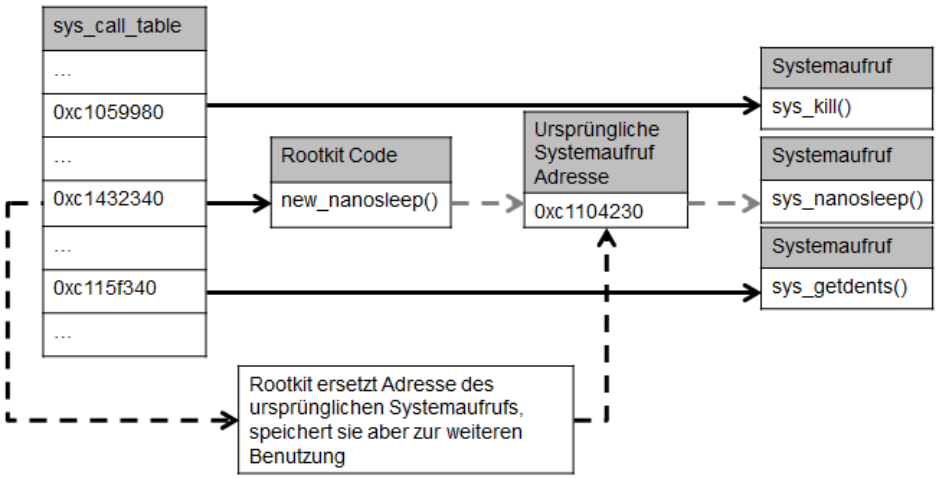

Abbildung 3: Beispiel für das Verhalten eines Kernelrootkits, welches einen Systemaufruf ersetzt

über die zwischengeschaltete Rootkitfunktion und können von dieser manipuliert werden.
Neben der beispielhaft beschriebenen Manipulation des Systemaufrufs *sys_nanosleep()* existieren noch weitere Möglichkeiten, das Systemverhalten zu beeinflussen:

- Verstecken von Dateien, Verzeichnissen und Prozessen: Durch Manipulation des Systemaufrufs *sys_getdents()* können Dateien und Verzeichnisse beispielsweise vor dem Programm *ls* verborgen werden.
- Manipulation eines Programmaufrufs: Mit Hilfe eines angepassten *sys_execve()* kann anstelle eines aufgerufenen Programms ein anderes ausgeführt werden.
- Benutzerrechte verändern: Rootrechte mittels *sys_setuid* verleihen
- Netzwerkverbindung verstecken
- Netzwerkpakete abfangen
- ...

Die aufgeführten Möglichkeiten zeigen einen Überblick dessen, wozu Kernel-Mode Rootkits fähig sind. Weitere Möglichkeiten und detailliertere Erläuterungen zeigt beispielsweise Peláez [16].

4. SCHUTZMÖGLICHKEITEN

Es zeigt sich, dass Rootkits eine große Bedrohung für die Sicherheit eines Systems darstellen. Entsprechend müssen Gegenmaßnahmen getroffen werden, um Rootkits die Manipulation des Systems zu erschweren und schon installierte Rootkits zu erkennen. Ein allgemeiner Überblick dazu zeigt z.B. Hannel [8]. Einige Methoden zu Erkennung und Abwehr werden in diesem Kapitel exemplarisch angesprochen.

4.1 Abwehr

Generell ist es wichtig, das System vor dem ersten Angriff zum Erlangen von Rootrechten zu schützen, welche nötig sind, um ein Rootkit zu platzieren.
Zusätzlich können auch Maßnahmen ergriffen werden, um das System zu härten. Dabei soll dem Angreifer die Installation des Rootkits zu erschwert, beziehungsweise die Funktion des Rootkits behindert werden. Nachfolgend sind einige Möglichkeiten aufgezeigt, die teilweise auch schon standartmäßig in Linux Systemen umgesetzt werden.
Um wichtige Dateien vor dem Ersetzen oder Verändern zu schützen, kann durch einen Benutzer mit Rootrechten ihr so genanntes immutable Flag gesetzt werden. Solange das Flag gesetzt ist, kann niemand diese Datei manipulieren bis das Flag wieder entfernt wird. Dadurch wird auch verhindert, dass die Datei durch ein Userland Rootkit ersetzt wird. Allerdings ist das für den Kernel selbst nicht möglich, wodurch Rootkits mittels LKM weiterhin den Kernel um ihren eigenen Code erweitern können.
Um einen Angriff durch LKM Kernel Rootkits zu vermeiden besteht die Möglichkeit den Kernel ohne die Unterstützung von Kernbelmodulen zu kompilieren. Diese Variante hat aber den Nachteil, dass alle benötigten Gerätetreiber (welche meist auf LKMs basieren) direkt in den Kernel integriert sein. Ein dynamisches Nachladen mittels Kernelmodulen ist dann nicht möglich. Systeme werden durch diese Einschränkung sehr unflexibel, da der Kernel für Änderungen immer neu kompiliert werden müsste.
Zusätzlich ist zu beachten, dass Rootkits nicht nur mittels LKMs in den Kernel gelangen können, sondern z.B. auch durch die Veränderung des Kernels im Speicher mittels /dev/kmem. Über diese spezielle Datei besteht ein Schreibzugriff auf den Speicherbereich des Kernels. Diese Möglichkeit ist aber in vielen Linuxdistributionen wie Ubuntu mittlerweile deaktiviert [5], da ein Zugriff über kmem in erster Linie eine Sicherheitsgefährdung darstellte, aber keinen sinnvollen Nutzen für das System mehr erfüllte. Eine mit der Linux Kernel Version 3.7 neu eingeführte Möglichkeit zum Schutz vor schädlichen Kernelmodulen ist die Verwendung von kryptographisch signierten Modulen [7]. Wird diese Option im Kernel aktiviert kann auch ein

doi: 10.2312/NET-2013-02-1_08

Benutzer mit Rootrechten keine Module mehr in den Kernel laden, welche nicht mit dem richtigen Schlüssel signiert wurden.
Wie erläutert wurde, verwenden Kernel Rootkits die Tabelle der Systemaufrufe um diese zu manipulieren. In früheren Kernelversionen (vor 2.6) wurde diese Tabelle noch vom Kernel exportiert und erlaubte so einen einfachen Zugriff für Kernelmodule mittels *extern void* sys_call_table[].*
In neueren Kernel Versionen wird die Tabelle standartmäßig nicht mehr exportiert und ist schreibgeschützt. Dadurch wird dem Rootkit der Zugriff auf die Adresse an der die Tabelle im Speicher liegt erschwert. Einige der möglichen Wege zu der Adresse liegen in der erwähnten System.map, die allerdings für den laufenden Betrieb nicht nötig ist und daher (nach einem Backup) theoretisch gelöscht werden kann[16][S.143].

4.2 Erkennungsmechanismen

Auch wenn ein System entsprechend geschützt bzw. gehärtet wurde, existiert keine Garantie, dass keine Kompromittierung vorliegt. Daher sollte regelmäßig überprüft werden, ob das Systemverhalten beeinflusst wird. Dazu existieren verschiedene Möglichkeiten.
Durch gezieltes Überprüfen der Rückgabe verschiedener Benutzerprogramme eines Systems, welche normalerweise gleiche Ergebnisse liefern (z.B. *ls* und *echo **), können Unterschiede aufgespürt werden, falls nur eines der Programme eine falsche Ausgabe liefert. Dieser Vergleich verschiedener Sichten wird als "Lie Detection" bezeichnet.
Ein weiteres Beispiel aus diesem Bereich ist das Aufspüren von Ports, welche von Rootkits als Hintertür geöffnet wurden und vor dem Benutzer versteckt werden. Dazu kann man die Ausgabe des Systems (*netstat -nlp*) mit einer externen Sicht vergleichen, die ein Portscanner wie zum Beispiel nmap liefert (*nmap -sT -p 1-65535 IP-Adresse*).
Chkrootkit [2] ist ein Tool, welches unter anderem überprüft, ob Prozesse versteckt werden, indem es Lie Detection anwendet. Prozessinformationen werden in Linux durch ein spezielles Dateisystem im Verzeichnis /proc repräsentiert [8]. Chkrootkit überprüft /proc/ und vergleicht den Inhalt mit dem Ergebnis des *ps* Befehls, um verborgene Prozesse aufzuspüren.
Ein weiteres Verfahren von Chkrootkit ist das Aufspüren von Konfigurationsdateien, welche von einigen Rootkits genutzt werden, um ihr Verhalten flexibel anpassen zu können (z.B. bei Jynx2 zum Wählen des Schlüsselworts und der ID die zum Verstecken einer Datei führt).
Eine weitere Möglichkeit zur Rootkiterkennung bietet eine regelmäßige Durchführung einer Integritätsprüfung von wichtigen Programmen oder Dateien. Dadurch können vor allem Userland Rootkits aufgespürt werden, da diese auf dem Ersetzen von Binärdateien basieren. Zu dieser Überprüfung wird die Datei mit einer vertrauenswürdigen Datei des Benutzerprogramms zu verglichen, die entweder früher als Backup gesichert wurde, oder beispielsweise aus einer vertrauenswürdigen, offiziellen Datenbank entnommen wird. Um den Vergleich zu vereinfachen wird meist ein Hashwert der Datei benutzt.
Rkhunter [4] ist ein Schutzprogramm, welches unter anderem dieses Verfahren einsetzt, um mittels MD5/SHA1 Hash Berechnung und einer aus dem Internet aktualisierbare Datenbank von Hashwerten veränderte Binärdateien aufzuspüren.
Einige Rootkits weisen spezielle Eigenschaften auf, anhand derer sie leichter zu identifizieren sind (Rootkit Signatur). KNARK zum Beispiel verwendet ungenutzte Signale an das System (*sys_kill(-31)*), um Prozesse zu verstecken. Diese können überprüft werden und bei unerwarteten Ereignissen (z.B. Verschwinden des Prozesses) eine Meldung ausgegeben werden.
Im Vergleich zu anderer Rootkiterkennungssoftware ist das letzte rkhunter Update relativ aktuell (Mai 2012), was für Rootkiterkennung eine wichtige Rolle spielt, da Rootkitersteller versuchen ihre Software vor Entdeckung zu schützen.
Einige weiterführende Tools zur Systemanalyse und -härtung beschreibt Claudia Eckert [6][Seite 179ff].

Gerade bei Kernel-Mode Rootkits läuft die Erkennung von Rootkits auf ein Wettrüsten zwischen Sicherheitssoftware und Rootkits hinaus. Rootkits versuchen sich möglichst komplett vor Erkennung zu verbergen, Sicherheitssoftware versucht Lücken in diesem "Versteck" zu finden. Wenn Erkennungstools versuchen eine Veränderung der Systemaufrufstabelle zu erkennen (z.B. durch Vergleich mit einem Snapshot aus einem früheren, unveränderten Zustand), können Rootkits darauf verzichten, auf die Originaltabelle zuzugreifen und beispielsweise mit einer Kopie der Tabelle arbeiten und Aufrufe an diese umleiten. Darüber hinaus kann auf die Manipulation der Systemaufrufstabelle verzichtet werden. Stattdessen überschreiben ein Rootkit die ersten Bytes der Systemaufrufe mit einem Sprungbefehl an die Adresse ihrer eigenen Variante.
Um den als Beispiel für Lie Detection erwähnten Portvergleich zu umgehen, kann ein Rootkit beispielsweise diesen erst dann öffnen, wenn davor versucht wurde in einer bestimmten Reihenfolge verschiedene Ports zu adressieren. Spricht ein externer Portscanner die Ports dann nicht zufällig in dieser Reihenfolge an, oder wird der Port gerade von dem Rootkit verwendet, ist er geschlossen und wird nicht erkannt.

5. ZUSAMMENFASSUNG UND AUSBLICK

Rootkits stellen eine große Sicherheitsgefährdung dar, welche sich seit den 80er Jahre kontinuierlich weiterentwickelt haben. Sie sind in der Lage unerkannt die Herrschaft über ein System zu behalten und dieses in dem Sinn ihres Erstellers zu beeinflussen, um diesem heimlichen Root Zugriff zu gestatten, sowie bestimmte Prozesse und Dateien vor Entdeckung zu schützen.
Grundlegend kann zwischen zwei Rootkit Typen unterschieden werden. User-Mode Rootkits setzen auf der Benutzerebene an und ersetzen komplette Programme bzw. Bibliotheken, während Kernel-Mode Rootkits Systemaufrufe des Kernels manipulieren. In diesem Paper wurde ein exemplarisches Kernel-Mode Rootkit vorgestellt, welches in der Lage ist einem gewöhnlichen Benutzer Rootzugriff auf das System zu gewähren.
Außerdem wurden einige Möglichkeiten aufgezeigt, wie basierend auf Lie Detection, Integritätsprüfung bzw. Signaturerkennung infizierte Systeme erkannt

[8]Für den Prozess mit der ID XXXX existiert das Verzeichnis /proc/XXXX/

doi: 10.2312/NET-2013-02-1_08

werden können. Je nach Rootkit können sich die Erkennungsmerkmale allerdings unterscheiden. Neuere Rootkitversionen sind in der Lage bekannte Erkennungsmechanismen zu umgehen, wie das in diesem Paper erwähnte User-Mode Rootkit Jynx2, welches weder von chkrootkit noch von rkhunter, zwei Rootkiterkennungstools, erkannt wurde.

Neuere Entwicklungen zeigen, dass sich Rootkits noch tiefer in einem System einnisten können, als es Kernel-Mode Rootkits tun.
Rootkits, die die Möglichkeiten der Virtualisierung benutzen, verschieben dabei das komplette Betriebssystem in eine virtuelle Umgebung. Sie werden als "virtual machine based Rootkit", kurz VMBR bezeichnet und setzen sich zwischen Hardware und Kernel und täuschen diesem vor, weiterhin volle Kontrolle über die Hardware zu besitzen. Beispiele sind Subvirt[10] oder Bluepill[17], beide vorgestellt im Jahr 2006.
Eine weitere Variante sind Firmware-Rootkits, welche sich direkt in der Hardware Ebene festsetzen und die Firmware beeinflussen. Ein solches Rootkit von John Heasman [9] überlebte sogar eine komplette Neuinstallation des Betriebssystems des kompromittierten Systems, in dem es sich im BIOS festsetzte.
Rootkits sind ein aktuelles Thema, welches gerade wegen den Eigenschaften sich vor Benutzern zu verbergen aufmerksam verfolgt werden sollte, um den Sicherheitsanforderungen von heutigen Systemen gerecht werden zu können.

6. LITERATUR

[1] Brain virus. Webseite. http://www.f-secure.com/v-descs/brain.shtml; besucht am 19.Dezember 2012.

[2] chkrootkit. Webseite. http://www.chkrootkit.org/; besucht am 18.Dezember 2012.

[3] Jynx2 rootkit. Webseite. http://www.blackhatlibrary.net/Jynx; besucht am 18.Dezember 2012.

[4] The rootkit hunter project. Webseite. http://rkhunter.sourceforge.net/; besucht am 18.Dezember 2012.

[5] Ubuntu security features. Webseite. https://wiki.ubuntu.com/Security/Features#dev-kmem; besucht am 19.Dezember 2012.

[6] C. Eckert. *IT-Sicherheit*. Oldenburg Wissenschaftsverlag GmbH, 2012. 7. Auflage.

[7] F. Grosshans. Linux kernel 3.7. Webseite, 2012. http://kernelnewbies.org/Linux_3.7; besucht am 2.Januar 2013.

[8] J. Hannel. Linux rootkits for beginners - from prevention to removal. Technical report, SANS Institute, Januar 2003.

[9] J. Heasman. Implementing and detecting a pci rootkit. *Black Hat DC 2007*, 2007.

[10] heise online. Microsoft demonstriert virtualisierungs-rootkit. Webseite. http://www.heise.de/newsticker/meldung/Microsoft-demonstriert-Virtualisierungs-Rootkit-110190.html; besucht am 17.Dezember 2012.

[11] G. Hoglund and J. Butler. *Rootkits*. Addison-Wesley Verlag, 2006.

[12] Intel. Intel® 64 and ia-32 architectures software developer's manual. Technical report, Intel, 2012.

[13] T. Maeda. Kernel mode linux. Webseite. http://web.yl.is.s.u-tokyo.ac.jp/~tosh/kml/; besucht am 20.Dezember 2012.

[14] T. Miller. Analysis of the knark rootkit. Webseite. http://www.ouah.org/tobyknark.html; besucht am 16.Dezember 2012.

[15] D. K. Mulligan and A. K. Perzanowski. The magnificence of the desaster: Reconstructing the sony bmg rootkit incident. *Berkeley Technology Law Journal [Vol. 22:1157 2007]*, pages 1158–1189, 2007.

[16] R. S. Peláez. Linux kernel rootkits: protecting the system's "ring-zero". Technical report, SANS Institute, Mai 2004.

[17] J. Rutkowska. Introducing blue pill. Webseite. http://theinvisiblethings.blogspot.de/2006/06/introducing-blue-pill.html; besucht am 19.Dezember 2012.

[18] styx. Syscall hijacking: Dynamically obtain syscall table address (kernel 2.6.x). Webseite. http://memset.wordpress.com/2011/01/20/syscall-hijacking-dynamically-obtain-syscall-table-address-kernel-2-6-x/; besucht am 13.Dezember 2012.

[19] styx. Syscall hijacking: Simple rootkit (kernel 2.6.x). Webseite. http://memset.wordpress.com/2010/12/28/syscall-hijacking-simple-rootkit-kernel-2-6-x/; besucht am 13.Dezember 2012.

doi: 10.2312/NET-2013-02-1_08