Berufsakademie Sachsen
Staatliche Studienakademie Leipzig

Kryptografische Sicherung der /boot-Partition bei Systemen mit vollverschlüsselter Root-Partition

Studienarbeit
in der Studienrichtung *Informatik*

Eingereicht von:
Daniel Böhmer
Heinrich-Mann-Weg 14
04289 Leipzig
Seminargruppe IT2007
Matrikel 207828

Leipzig, 20. August 2010.

Inhaltsverzeichnis

1 Einleitung **4**

2 Techniken für Festplattenverschlüsselung **5**

2.1 Verschlüsselte Dateien 5

2.2 Verschlüsselte Dateisysteme 6

2.3 AES-Festplatten 7

3 Aufbau des Zielsystems **8**

3.1 Hardware 8

3.2 Das Partitionslayout 8

3.3 Linux Unified Key Setup und dm-crypt 9

3.4 Logical Volume Manager 10

3.5 Die Dateisysteme 11

4 Angriffsszenarien I **12**

4.1 Modifikation von `cryptsetup` 12

4.2 Modifikation des Login-Prompts 12

4.3 Einschleusen von Spyware 13

4.4 Proof of Concept *Evil Maid Attack* 13

5 Verteidigungsstrategien **14**

5.1 `/boot`-Partition auf einem Wechseldatenträger 14

5.2 Anlegen und Überprüfen von Hash-Werten 14

5.2.1 Was Hash-Funktionen sind 15

6 Überlegungen zur Implementierung **17**

6.1 Speicherort des Hash-Werte 17

6.2 Verwendete Hash-Algorithmen 17

6.3 Eingabewert für die Algorithmen 19

6.4 Auffüllen des Speicherplatzes 20

6.5 Aufruf des Programms 21

6.5.1 Einbindung in `/etc/rc.local` 21

6.5.2 Eigenes Init-Skript 21

6.5.3 Einbindung in bestehende Programme 22
6.6 Deployment 23

7 Angriffsszenarien II 24
7.1 Herbeiführen von Hash-Kollisionen 24
7.2 Vortäuschen einer falschen `/boot`-Partition durch Kernel-Modifikation 25
7.3 Modifikation von Hash-Programmen oder -Bibliotheken 26
7.4 Verhinderung der Ausführung des Programms 26

8 Fazit 28

1 Einleitung

Diese Studienarbeit beschäftigt sich mit der Sicherheit von Computersystemen mit sogenannter *vollverschlüsselter Festplatte*, auch Full Disk Encryption (FDE) genannt. Das grundlegende Problem der notwendigerweise unverschlüsselten `/boot`-Partition soll diskutiert und Lösungsmöglichkeiten erarbeitet werden.

Betroffenen von dieser Problematik sind alle mobilen oder unbeaufsichtigten Computersysteme, die so konstruiert wurden, dass Daten auf Massenspeichern wie herkömmlichen Festplatten oder modernen Flash-Speichern (in dieser Arbeit unter dem Begriff Festplatte zusammengefasst) verschlüsselt abgelegt werden. Die Gründe für den Einsatz dieser kryptografischen Sicherung können vielfältig sein. Zum Erreichungen des angestrebten Niveaus an Datenschutz ist es jedoch generell angeraten, nicht nur einzelne Dateien mit sensiblem Inhalt zu verschlüsseln, sondern die gesamte Dateistruktur des Computersystems inklusive aller Systemdateien derartig abzusichern. Andernfalls können unbemerkt sensible Informationen in nicht gesicherten Bereichen wie temporären Verzeichnissen abgelegt werden oder die Sicherheit des Systems wird durch Modifikationen an den offenliegenden Bereichen kompromittiert.

Bei Software-basierten Verschlüsselungssystemen kommt eine Besonderheit dazu: Weil ein Computersystem nach dem Einschalten zuerst das Betriebssystem laden und starten muss, benötigt es uneingeschränkten Zugang zu einem Teil des Speichers, der deshalb nicht verschlüsselt sein kann. Nachdem ein minimaler Teil des Betriebssystems gestartet wurde, der den kryptografischen Schlüssel als Eingabe erhält, kann auf den größeren verschlüsselten Teil des Speichers zugegriffen werden.

Diese Arbeit beschäftigt sich mit der Sicherheitsproblematik, die sich aus diesem Umstand ergibt.

2 Techniken für Festplattenverschlüsselung

Bevor die spezifischen Attacken auf FDE-Systeme betrachtet werden, sollen an dieser Stelle verschiedene Technologien kurz dargestellt und ihre Vertreter vorgestellt werden.

2.1 Verschlüsselte Dateien

Das am einfachsten anzuwendende Verschlüsselungsverfahren ist das Chiffrieren einzelner Dateien, beispielsweise um sie über unsichere Kanäle wie eine E-Mail zu verschicken. Dazu ist ein einfaches Programm notwendig, dass die Originaldatei einliest und mit einem gegebenen Schlüssel eine verschlüsselte Version derselben Datei erzeugt.

Diese Vorgehensweise ist selbst für ungeübte Anwender praktikabel, weil sie simpel ist und beispielsweise dem Benutzen von Dateikompressionsprogrammen ähnelt. Außerdem erfordert es neben der eventuell notwendigen Installation eines Verschlüsselungsprogramms keine weiteren Systemanpassungen.

Eines der bekanntesten Programme für diesen Einsatz ist Truecrypt[1]. Genauso wie FreeOTFE[2] beherrscht es noch eine Reihe weiterer Betriebsarten, erlaubt aber auch das einfache Ver- und Entschlüsseln einzelner Dateien. Einen tiefergreifenden Ansatz verfolgt ecryptfs[3]: Dabei handelt es sich um eine Dateisystem für Linux, welches ein Verzeichnis mit verschlüsselten Dateien an einer anderen Stelle mit den entschlüsselten Äquivalenten der Dateien einblendet. ecryptfs wird beispielsweise unter Ubuntu Linux verwendet, um verschlüsselte private Verzeichnisse zu realisieren, ohne dafür einen Container mit fixer Größe anlegen zu müssen.

Alle genannten Lösungen haben den Nachteil gemein, dass die Größen und Dateinamen der Dateien transparent im Ursprungsdateisystem liegen – lediglich der Inhalt der Dateien bleibt geheim und ist nur lesbar, wenn sie mit Hilfe des geheimen

[1] `http://www.truecrypt.org`
[2] `http://www.freeotfe.org`
[3] `http://launchpad.net/ecryptfs`

Schlüssels entschlüsselt werden. Im Fall von ecryptfs bedeutet das, dass das ecryptfs-Dateisystem eingehängt (englisch *mounted*) ist.

Wie bereits begründet, ist diese Methode als ungeeignet für FDE zu betrachten, bietet sich jedoch für den sicheren Austausch einzelnener Dateien zwischen zwei Computern über einen unsicheren Kanal an.

2.2 Verschlüsselte Dateisysteme

Wenn Dateien nicht zwischen zwei Parteien übertragen werden sollen, sondern auf einem Computer für längere Zeit verschlüsselt abgelegt werden und zwischendurch verändert werden sollen – wie es für Systemdateien und lokale Benutzerdateien üblich ist – erscheint es sinnvoller, all jene Dateien in einem verschlüsselten Container abzulegen, statt sie einzeln zu verschlüsseln.

Viele populäre Verschlüsselungsprogramme unterstützen diese Betriebsart und können entweder Dateien in einem anderen Dateisystem oder ganze Partitionen oder Blockdatenträger als Container verwenden. Das Programm erhält mit Hilfe des geheimen Schlüssels Zugang zum Container und stellt dessen Inhalt als virtuellen Datenträger bereit. Unter Microsoft Windows wäre dies eine zusätzliche Festplatte. In einem Unix-System wird dieser Datenträger als zusätzliches Blockgerät (englisch *block device*) repräsentiert, welches ohne Unterschied wie jedes andere Blockgerät angesprochen werden kann.

In jedem Fall kann in diesem neuen, virtuellen Datenträger nun ein Dateisystem initialisiert werden, welches danach auf gewöhnliche Weise mit Dateien gefüllt werden kann.

Der Linux-Kernel bietet für diese Art von Verschlüsselung das Modul *dm-crypt*. Es nutzt die im Kernel vorhandenen Kryptografie-Algorithmen und stellt ein unverschlüsseltes virtuelles Blockgerät zur Verfügung, welches auf einem Quellgerät basiert.

Eine Erweiterung dieses Moduls heißt Linux Unified Key Setup (LUKS). Es unterscheidet sich von einem reinem dm-crypt-Datenträger dadurch, dass am Anfang des Blockgeräts ein Header-Bereich eingefügt wird, der Meta-Informationen über den verwendeten Algorithmus und die Schlüssel erhält. Mit LUKS ist es auch möglich, mit mehreren unterschiedlichen, unabhängigen Schlüsseln auf den verschlüsselten Inhalt zuzugreifen. Weil LUKS im Moment den aktuellen Stand der Technik darstellt, soll für die Arbeit angenommen werden, dass die verschlüsselte Festplatte mit LUKS formatiert ist und sich in diesem Container das /-Dateisystem befindet.

Das bereits erwähnte Programm FreeOTFE ist auch unter Microsoft Windows in der Lage LUKS-verschlüsselte Container zu öffnen. Da Windows-Systeme aber nicht von diesem Datenträger starten können und es in dieser Arbeit um die Sicherung

der `/boot`-Partition unter Unix-Systemen geht, wird stattdessen angenommen, dass es sich um ein GNU/Linux oder vergleichbares System handelt.

2.3 AES-Festplatten

Seit einigen Jahren bieten Festplatten- und Zubehörhersteller Produkte an, die die Verschlüsselung der Daten direkt auf Hardware-Ebene durchführen. Dabei wird der geheime Schlüssel beim Systemstart entweder über die Firmware des Computers, das Basic Input Output System (BIOS), eingelesen oder zusätzliche Elektronik am Gerät selbst nimmt die Daten entgegen. Beispielhaft seien hier ein RFID-Tag oder eine integrierte Miniaturtastatur genannt.

Die offensichtlichen Vorteile der Systeme sind, dass die Rechenleistung auf einen externen Prozessor ausgelagert wird und dass die Verschlüsselung unsichtbar für das Betriebssystem oder das BIOS stattfindet.

Allerdings sind solche Produkte aufgrund des kleinen Marktes und der zusätzlichen Entwicklungsarbeit und Elektronikbauteile teurer als klassische Datenträger. Zusätzlich entsteht ein Nachteil aus der Intransparenz der erhältlichen Produkte: Ohne die Elektronik der Datenträger sind die auf ihnen verschlüsselt abgelegten Daten wertlos – das heißt, dass besondere Schwierigkeiten für ein Backup oder das Wiederherstellen von intakten Speichermedien mit defekter Elektronik entstehen. Außerdem haben mehrere Fälle gezeigt[1], dass die angebotenen Geräte die Sicherheit der beworbenen Algorithmen durch Konstruktionsmängel teilweise komplett aushebeln.

Ein grundsätzliches Problem dieser integrierten Verschlüsselungseinheiten ist, dass sie nicht über die Herkunft von Lese- oder Schreibanfragen urteilen können. Ist eine derartige AES-Festplatte einmal initialisiert, können die Daten frei gelesen oder geschrieben werden. Ein Angreifer, der ein derartig geschütztes System in seine Gewalt gebracht hat, kann jedoch durch einen System-Neustart ein eigenes Betriebssystem mit eigenen Programmen starten und so auf die Daten zugreifen, sofern er nicht die Stromversorgung der Festplatte unterbricht und das Computersystem beim Neustart nicht den SATA-Link unterbricht. Es ist unklar, welche verfügbaren Hardware-Implementierungen diese Bedingungen erfüllen, weil die Hersteller keine entsprechenden Informationen veröffentlichen und wegen der hohen Investitionskosten kein Modell zum praktischen Versuch gekauft werden konnte.

3 Aufbau des Zielsystems

Das Zielsystem, für welches in dieser Arbeit weitergehende Schutzmaßnahmen konstruiert werden sollen, basiert auf dem bereits beschriebenen Software-Verfahren mit `dm-crypt` und LUKS.

In diesem Kapitel soll ein Überblick über die vorhandenen Schichten in einem solchen System gegeben werden.

3.1 Hardware

Die physikalischen Komponenten des Systems können gewöhnliche Computer-Komponenten sein. Am stärksten verbreitet ist die x86-Architektur. Ein Standard-PC aus dem einschlägigen Handel, dessen Komponenten vom Linux-Kernel angesprochen werden können, ist ausreichend. Spezielle Anforderungen wegen der Verwendung von Kryptografie werden nicht gestellt.

Der CPU-Hersteller Intel hat jedoch 2010 mit dem Arrandale-Design CPUs eingeführt, die eine AES-Befehlssatzerweiterung zur Hardware-beschleunigten Berechnung des Advanced Encryption Standard (AES) enthalten. Mit Hilfe dieser Instruktionen ist es möglich, Verschlüsselungsoperationen dieses Algorithmus mit spezialisierten Teilen der CPU zu erledigen und damit CPU-Zyklen gegenüber einer Software-implementierten einzusparen. Dies erhöht die Verarbeitungsgeschwindigkeit beziehungsweise senkt die CPU-Auslastung. Auch der direkte Konkurrent Intels, die Firma AMD, hat angekündigt die Instruktionen in ihren CPUs zu implementieren. Damit ist davon auszugehen, dass der Befehlssatz zukünftig durch hohe Verbreitung im x86-Markt zum de-facto-Standard wird, obwohl die Firma VIA bereits seit mehreren Jahren mit ihrer *Padlock Engine* eine ähnliche Technik entwickelt und vertrieben hat.

Die beschriebenen Formen der Hardware-Beschleunigung für AES sind jedoch keine Voraussetzung für die Verwendung von FDE und die Anwendung der in dieser Arbeit entwickelten Darlegungen.

3.2 Das Partitionslayout

Damit eine Software-basierte Verschlüsselungstechnik zum Einsatz kommen kann, muss beim Systemstart zuerst die entsprechende Software geladen werden. Unter

Unix ist Grand Unified Bootloader (GRUB) einer der üblichen Boot-Loader, der bei den meisten modernen Linux-Distributionen Verwendung findet. Dieser wird im Master Boot Record (MBR) der Festplatte installiert.

Sekundäre Teile des GRUB-Programms und initiale Bestandteile des zu startenden Betriebssystems werden in einer dafür angelegten Partition, der `/boot`-Partition, gespeichert. Grundsätzlich kann eine einzelne große `/`-Partition alle Daten aufnehmen. Wenn jedoch der restliche Teil verschlüsselt wird, muss es eine zusätzliche dedizierte `/boot`-Partition geben. Wie in einem späteren Abschnitt dargelegt wird, sollte sie eine minimale Größe haben. Nach Erfahrungswerten des Autors ist eine Größe von 50 bis 100 MiB angemessen.

Innerhalb der Partition wird ein einfaches Dateisystem wie Ext2 installiert, welches die Dateien aufnimmt. Für ein Standard-Linux-System wäre es ausreichend ein Kernel-Abbild zu speichern. Dieses lädt alle integrierten Hardware-Treiber und kann daraufhin alle weiteren Betriebssystemteile von weiteren Partitionen mit Dateisystemen einlesen. Für den Verschlüsselungsprozess notwendige Dateien müssen jedoch beim Systemstart vorhanden sein. Deswegen wird neben dem Kernel-Abbild ein »initialer RAM-Speicher« (englisch *initial ram disk*) unter `/boot/` abgelegt. Dabei handelt es sich um ein `initramfs`-Archiv, welches alle Dateien enthält, die nach dem Laden des Kernels bereitstehen sollen.

Die Größe der `/boot`-Partition sollte danach bemessen werden, wie viele verschiedene Kernel-Abbilder und dazugehörige `initramfs`-Archive (beispielsweise verschiedene Kernel-Versionen) darin abgelegt werden sollen.

Der gesamte restliche Speicherplatz auf der Festplatte sollte für eine einzelne Partition aufgewendet werden, so dass kein freier Platz übrig bleibt. Darin wird der verschlüsselte Container installiert.

3.3 Linux Unified Key Setup und dm-crypt

LUKS ist ein Container mit Meta-Informationen für die komfortable und sichere Verwaltung verschlüsselter Dateisysteme unter Linux.

Ein solcher Container wird mit dem Programm `cryptsetup` angelegt. Bei der Initialisierung wird einmalig festgelegt, mit welchem Algorithmus und in welchem Betriebsmodus der Container benutzt werden soll. Initial wird nur ein Schlüssel angegeben, die Schlüssel können jedoch später ausgetauscht werden. Neben Passphrasen sind auch Binärdateien als Schlüssel geeignet und es können bis zu acht davon abgelegt werden. LUKS bietet noch weitere Optionen wie hierarchische Schlüssel an, die nicht Gegenstand dieser Arbeit sind.

Nach dem LUKS-Header folgt ein klassischer dm-crypt-Container, der mit den im LUKS gespeicherten Parametern erstellt und betrieben wird. Dieser Bereich enthält

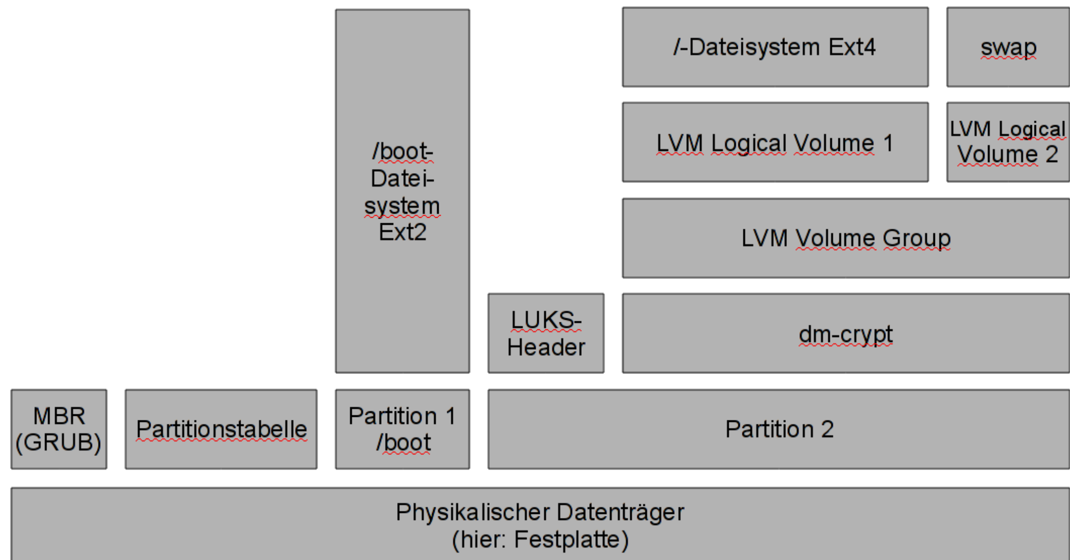

Abbildung 3.1: Die Verteilung der unterschiedlichen Container im Datenträger

ausschließlich AES-verschlüsselte Rohdaten und hat deshalb ähnliche Eigenschaften wie eine Menge von Zufallszahlen.

3.4 Logical Volume Manager

Sollen mehrere Dateisysteme innerhalb eines LUKS-verschlüsselten Containers angelegt werden, wird üblicherweise der Logical Volume Manager (LVM) verwendet. Er wurde ursprünglich geschaffen, um Dateisysteme flexibler verwalten zu können. Mit ihm ist es beispielsweise möglich, Dateisystem unabhängig von Partitionsgrenzen zu erstellen oder sie über mehrere physikalische Datenträger zu verteilen (ähnlich zu RAID-0).

Seine Fähigkeit, mehrere Container für Dateisysteme innerhalb einer Struktur bereitzustellen, kann man sich bei Verwendung von FDE zunutze machen.

Der LUKS-Container wird zuerst erstellt und danach unter Eingabe eines Schlüssels geöffnet. Das bereitgestellte Blockgerät wird im LVM als *physical volume* eingetragen. Über dieser Menge mit einem Element wird nun eine *volume group* erstellt. Das Konzept von LVM sieht vor, dass an dieser Stelle problemlos auch mehrere Datenträger verwendet werden könnten. Innerhalb der *volume group* werden nun *logical volumes* erstellt. Jedes davon kann später ein Dateisystem beherbergen.

Ein zusätzliches Dateisystem kann zum Beispiel eine getrennte `/home/`-Partition oder wie in Abbildung 3.1 dargestellt ein Auslagerungsbereich (englisch *swap*) sein. Bei der Verwendung von nur einem Dateisystem kann selbstverständlich auf den

Einsatz von LVM verzichtet werden.

3.5 Die Dateisysteme

Nach der beschriebenen Kaskade von Schichten und Containern werden nun die Dateisysteme initialisiert. Dies geschieht mit den üblichen Werkzeugen, da der Umgang mit einem *logical volume* oder einem dm-crypt-Container sich nicht vom Umgang mit einer gewöhnlichen physikalischen Partition unterscheidet. Alles sind reguläre Blockgeräte.

4 Angriffsszenarien I

In diesem Kapitel soll beschrieben werden, welche Arten von Angriffen auf FDE-Systeme möglich sind. Dabei geht es dem Titel der Arbeit entsprechend nur um Attacken, die mit der `/boot`-Partition des Computers zu tun haben. Weitere bekannte Formen wie die sogenannte *Cold Boot Attack* werden außer Acht gelassen. Natürlich könnte ebenfalls der geheime Schlüssel durch Unachtsamkeit bekannt werden oder erpresst werden.

Alle im Folgenden beschriebenen Angriffe basieren auf Modifikationen des initramfs-Archivs, welches alle Dateien enthält, die zum Boot-Zeitpunkt vom Bootloader in den Arbeitsspeicher geladen werden. Durch Manipulationen anderer Dateien in der `/boot`-Partition wären vermutlich ähnliche Ergebnisse erzielbar.

Damit der Angreifer einen Nutzen von seiner verdeckten Aktion hat, muss er beim nächsten Benutzen des Computersystems durch den Besitzer, der den geheimen Schlüssel kennt, Kenntnis über Interna des Systems erlangen. Das interessanteste Ziel ist dabei der Schlüssel, mit dessen Hilfe später alle weiteren Details aus der entschlüsselten Festplatte extrahiert werden können.

4.1 Modifikation von `cryptsetup`

In einem mittels LUKS-verschlüsselten Unix-System dient der Befehl `cryptsetup` zur Verwaltung von verschlüsselten Containern. Dazu gehört auch das Einhängen und Abfragen des Schlüssels.

Ein Angreifer könnte das quelloffene Programm so verändern, dass es den Schlüssel nach dem Eingeben auf der Festplatte in einem nicht verschlüsselten Bereich speichert oder später über das Netzwerk an einen Angreifer versendet.

4.2 Modifikation des Login-Prompts

Auch das Login-Programm des Betriebssystems bietet eine Angriffsfläche für Modifikationen. Im Text-Terminal sorgt üblicherweise das Programm `getty` dafür, ein eingegebenes Passwort entgegenzunehmen und den Benutzer bei korrekter Eingabe anzumelden. Für grafische Umgebungen wie GNOME oder KDE gibt es entsprechende Session-Manager wie GDM beziehungsweise KDM. Mit Hilfe von Anpassungen im

initramfs-Archiv könnten diese Programme ausgetauscht werden und die Angreiferversion könnte das Passwort des Benutzers auf den genannten Wegen veröffentlichen. Weil Menschen dazu neigen, für mehrere Zwecke dasselbe Passwort zu verwenden, ist im Zweifelsfall anzunehmen, dass das Anmeldepasswort des Benutzerkontos im Betriebssystem dem Schlüssel des Crypto-Systems entspricht.

4.3 Einschleusen von Spyware

Weiterhin wäre es leicht möglich, ein eigenes Programm oder Kernel-Modul hinzuzufügen, das dauerhaft Tastatureingaben protokolliert (sogenannter *Keylogger*) und bei Verfügbarkeit über das Netzwerk versendet.

4.4 Proof of Concept *Evil Maid Attack*

Dass die beschriebenen Angriffstechniken in der Praxis durchführbar sind, zeigt das Proof of Concept, der unter dem Namen *Evil Maid Attack* bekannt ist[2].

Die TrueCrypt-Entwickler, die das Programm geschrieben haben, wollten zeigen, dass auch ein mit Truecrypt »vollverschlüsseltes« Computersystem angreifbar ist, weil der Truecrypt-Bootloader wie die `/boot`-Partition eines Unix-Systems unverschlüsselt gespeichert wird.

Mit ihrem Programm auf einem USB-Stick kann man einen Laptop oder anderen PC booten und innerhalb einer Minute ist der Bootloader so manipuliert, dass er ein eingegebenes Passwort auf dem unverschlüsselten Teil abspeichert. Wenn man danach wieder mit dem Programm vom USB-Stick auf den Computer zugreift, zeigt den Schlüssel im Klartext an.

Durch diesen Proof of Concept ist klar, dass derartige Attacken im Bereich des Möglichen sind und bei guter Vorbereitung sogar sehr leicht durchführbar sind. Außerdem illustriert dieses Beispiel, dass es vorteilhaft sein kann, den Computer nicht auszuschalten, wenn es unbeobachtet ist. Das Ausschalten wird zwar als Vorsichtsmaßnahme gegen die *Cold Boot Attack* empfohlen, aber im eingeschalteten Zustand kann der Computer nicht mit einem fremden Betriebssystem gestartet werden, ohne dass der Benutzer die Veränderung mitbekommt.

5 Verteidigungsstrategien

5.1 `/boot`-Partition auf einem Wechseldatenträger

Eine sehr sichere Methode ist das Ablegen der `/boot`-Partition auf einem Wechseldatenträger, der nur zum Starten des Computers angeschlossen wird und ansonsten an einem sicheren Ort hinterlegt oder ständig vom Besitzer mitgeführt wird.

In diesem Fall kann der vollständige Inhalt der Festplatte verschlüsselt werden, so dass ein Angreifer nicht unbemerkt Manipulationen vornehmen kann.

Leider ist diese Vorgehensweise in vielen Fällen nicht praktikabel: Der Wechseldatenträger, beispielsweise in USB-Flash-Speicher ist bei vielen Starts hoher mechanischer Belastung ausgeliefert und ein Benutzen des Computers ist unmöglich, wenn der Wechseldatenträger defekt oder nicht vor Ort ist.

Im Übrigen bedingt das Anschließen des Wechseldatenträgers die Anwesenheit eines Menschen, weshalb unbeaufsichtigte Systeme nicht auf diese Weise gesichert werden können. Ein Server in einem Rechenzentrum, der nach dem Neustart per SSH auf die Eingabe des Schlüssels wartet, könnte nicht aus der Ferne in Betrieb genommen werden, wenn dazu das Anstecken des Datenträgers notwendig wäre. Die `/boot`-Partition muss sich also notwendigerweise auf der Festplatte befinden.

5.2 Anlegen und Überprüfen von Hash-Werten

Aus den genannten Gründen kann es unvermeidlich sein, das Computersystem dem beschriebenen Risiko auszusetzen. Ziel soll es sein, einen erfolgten Angriff, wenn er nicht erfolgreich verhindert werden kann, mindestens mit einer hohen Wahrscheinlichkeit zu erkennen.

Dazu soll beim Start des Computersystem überprüft werden, ob sich die `/boot`-Partition seit dem letzten Betrieb verändert hat. Statt ein vollständiges Abbild der `/boot`-Partition zu speichern und es später Byte-weise zu vergleichen, was Speicherplatz und Zeit zum Schreiben der vollständigen Kopie kostet, sollen kryptografische Methoden eingesetzt werden.

5.2.1 Was Hash-Funktionen sind

Zum Zwecke des Vergleichs werden Hash-Algorithmen benutzt, die auf mathematischen Einweg-Funktionen beruhen. Sie berechnen zu einem gegebenen Eingabewert M einen Vergleichswert h mit einer fixen Länge m von einigen Bytes (üblicherweise 128 bis 160 Bits) [3, S. 491f.]. Für die gleiche Eingabe ist das Ergebnis immer dasselbe. Daher kann man mit dem Vergleichen der Hash-Werte Aussagen über die indirekt verglichenen Eingaben machen.

$$h = H(M) \quad \text{wobei Länge von } h \text{ immer } m \tag{5.1}$$

Obwohl Hash-Funktionen für alle möglichen Eingaben eindeutig definiert und in dieser Hinsicht vollständig deterministisch sind, haben sie mehrere Eigenschaften, die sie für die kryptografische Verwendung nutzbar machen. Die folgenden davon sind für diese Arbeit von Bedeutung.

Hash-Funktionen basieren auf mathematischen Einwegfunktionen, für die kein schneller Algorithmus zur Umkehrfunktion bekannt ist. kann. Deshalb ist es mit heutiger Technik nicht möglich, eine Hash-Funktion in akzeptabler Zeit umzukehren und zu einem gegebenen Hash h eine passende Nachricht M zu finden, die diesen Hash erzeugt. Dieses Problem soll an dieser Stelle Problem A genannt werden.

Zusätzlich müssen Hash-Funktionen eine weitere Art von Berechnung erschweren. Es muss sehr schwer sein, zwei beliebige Nachrichten M und M' zu finden, die denselben Hash-Wert erzeugen, obwohl der Hash-Wert beliebig ist. Das finden von solchen zwei Nachrichten sei Problem B.

Beide Probleme stehen mit dem *Kollisionsresistenz* in engem Zusammenhang. Als Kollision bezeichnet man grundsätzlich zwei Nachrichten mit demselben Hash-Wert. Somit entspricht die Resistenz gegenüber diesem Ereignis einer niedrigen Wahrscheinlichkeit dafür. Je besser die Kollisionsresistenz eines Hash-Algorithmus, desto sicherer ist er.

Unabhängig vom konkreten Hash-Algorithmus kann man sagen, dass das Lösen von Problem B leichter ist als von Problem A. Das liegt daran, dass bei Problem A ein bestimmter Hash-Wert – nämlich der der echten Nachricht M – gesucht ist, während bei Problem B der Suchraum für alle möglichen Wert der Hash-Funktion offen ist. Diese Tatsache ist unter dem Begriff *Geburtstagsproblem* bekannt.

Glücklicherweise spielt das Teilproblem B bei dieser Arbeit keine große Rolle, weil der echte Hash durch den Hash der unveränderten `/boot`-Partition vorgegeben ist. Ein Angreifer kann nicht einen beliebigen Hash-Wert aussuchen und damit Manipulationen durchführen. Dies wirkt sich positiv auf die Sicherheit der kryptografischen Aspekte des Sicherungssystems aus.

Ein Hash-Algorithmus eignet sich also hervorragend als Form der Signatur für

die `/boot`-Partition, mit deren Hilfe schon mit wenigen Bytes an Daten die gesamte `/boot`-Partition auf Integrität überprüft werden kann.

6 Überlegungen zur Implementierung

6.1 Speicherort des Hash-Werte

Eine grundlegende Frage ist, wo der Hash-Wert für die `/boot`-Partition abgespeichert werden soll. Externe Datenträger scheiden aus, weil sonst auch die gesamte `/boot`-Partition darauf gespeichert werden könnte.

Als Speicherort bleiben die unverschlüsselte `/boot`-Partition selbst und der verschlüsselte Teil der Festplatte, als zum Beispiel die `/`-Partition. Es wird davon ausgegangen, dass der vollstände verfügbare Speicherplatz von diesen beiden Teilen eingenommen wird.

Aus offensichtlichen Gründen muss der Hash während des Betriebs des Computersystems im dann offenen `/`-Dateisystem gespeichert werden. Würde er stattdessen auf der `/boot`-Partition abgelegt, könnte der Angreifer mit dem Kernel-Abbild auch den Vergleichswert ändern. Dies würde das Sicherungssystem vollständig kompromittieren.

An dieser Stelle kommt ein Vorteil moderner Verschlüsselungsverfahren zum Tragen: Sie verschleiern nicht nur zuverlässig den Inhalt verschlüsselter Nachrichten, sondern verhindern auch, dass ein Angreifer ohne Kenntnis des Schlüssels durch Modifikationen des Chiffretexts den Klartext zielgerichtet beeinflussen kann. Er kann lediglich durch zufälliges Überschreiben die Daten zerstören oder durch erneutes Verschlüsseln in seine Gewalt bringen. All dies hätte aber keinen Einfluss auf die Integrität des Hash-Werts: Er kann nicht zielgerichtet verändert oder zerstört werden, ohne unbeabsichtigt weitere Teile des Systems anzugreifen, so dass das Betriebssystem nicht mehr funktionsfähig wäre. Das Ziel einer unbemerkten Modifikation ist auf diese Weise unerreichbar.

6.2 Verwendete Hash-Algorithmen

Eine elementare Rolle spielt die Frage, welcher Hash-Algorithmus verwendet werden sollte, um die Prüfsumme zu berechnen.

Algorithmus	Ausgabelänge	bekannte Schwächen
CRC32	32 Bits	keine Hash-Funktion, lediglich Prüfsumme
LM-Hash	128 Bits	Brute-Force stark erleichtert, Rainbow-Tables
MD5	128 Bits	anfällig gegen Kollisionsangriffe
RIPEMD-160	160 Bits	keine
SHA-1	160 Bits	keine
SHA-2-Familie	bis zu 512 Bits	keine

Tabelle 6.1: Verschiedene Algorithmen im Vergleich

Grundsätzlich steigt die Sicherheit mit der Länge des Ausgabewertes. Alle Kombinationen von Eingaben nach einer gültigen zu durchsuchen, ist umso schwerer, desto größer der Ausgaberaum ist. Beispielsweise ist es mit den 160 Bit des SHA-1 viel schwieriger, einen passenden Eingabewert zu einem gegebenen Hash zu finden.

Außerdem gab es in der Vergangenheit Kryptoanalysen, die gezeigt haben, dass es möglich ist, den zu durchsuchenden Schlüsselraum für einen gegebenen Hash-Wert einzugrenzen, oder dass in bestimmten Fällen einzelne Bits für die Hash-Wertberechnung keine Rolle spielen. Diese mathematischen Schwächen dieser Algorithmen machen sie angreifbar und ermöglichen gezielte Angriffe, die möglicherweise schneller sind als das simple Durchsuchen des Eingaberaums.

Die Auswahl des Hash-Algorithmus sollte also nach Länge des Hash-Wertes gehen und bekannte Schwächen der Algorithmen berücksichtigen. Eine gute Auswahl wären die Vertreter der SHA-2-Familie mit bis zu 512 Bit und anerkannter Sicherheit.

Zu weiteren Erhöhung der Sicherheit könnten mehrere Hash-Werte berechnet werden. Wenn man von analytischen Attacken absieht, ist die Wahrscheinlichkeit, dass eine zufällige Eingabe einen bestimmten Hash-Wert liefert,

$$P = \frac{1}{2^m} \tag{6.1}$$

Mit weiteren Hash-Algorithmen (insgesamt n) mit den Ausgabelängen $m_1, m_2, \ldots, m_n$ multiplizieren sich die Wahrscheinlichkeiten

$$P = P_1 \cdot P_2 \cdot \ldots \cdot P_n \tag{6.2}$$

$$P = \frac{1}{2^{m_1+m_2+\ldots+m_n}} \tag{6.3}$$

so dass

$$P \ll P_i \quad \text{für } 1 \leq i \leq n \tag{6.4}$$

Selbst beim Einsatz von grundsätzlich weniger sichereren Hash-Algorithmen wie MD5 oder sehr kurzen Prüfsummen wie bei CRC32, verbessert sich die Sicherheit

des Gesamtsystems, wenn die diese zusätzlich gespeichert werden. Je nach Sicherheitsbedürfnis und verfügbarer Rechenleistung, kann es also auch sinnvoll sein, alle verfügbaren Hash-Algorithmen zu verwenden und Kollisionen somit faktisch auszuschließen.

6.3 Eingabewert für die Algorithmen

Bisher wurde für die Hash-Wertberechnung immer ein einzelner Eingabewert angenommen. Bezogen auf das Problem mit der `/boot`-Partition wäre dieser Eingabewert die Folge aller Bits vom Anfang bis zum Ende der Partition. Diese Folge beinhaltet Blöcke mit Dateiinhalten genauso wie Strukturinformationen des Dateisystems, Dateinamen und Zugriffszeitstempel. Bei der praktischen Verwendung des Systems könnte dies zu einem Problem werden.

Wenn ein Unix-System ein Dateisystem einhängt, wird nicht nur sein Inhalt les- und schreibbar an einer Stelle des virtuellen Dateisystems eingeblendet, sondern es werden auch Statusinformationen direkt im Dateisystem hinterlegt. Dazu gehört beispielsweise ein Zähler für die Anzahl der `mount`-Vorgänge und diverse Zeitstempel – auch für das Lesen von Dateien.

Durch diese Modifikationen ändert sich bereits der gesamte Hash-Wert der Partition. Entweder muss nach jedem Aushängen ein neuer Hash-Wert berechnet werden oder die Partition wird nur lesend eingehangen. Mit der Option `ro` kann man den Kernel anweisen, den Inhalt der Partition nicht zu verändern. Entsprechend sind Dateien darin dann nicht schreibbar. Dies würde allerdings Updates der Dateien, beispielsweise des Bootloaders, unmöglich machen.

Die Alternative ist, Hashes der einzelnen Dateien im Dateisystem anzulegen. Wenn man annimmt, dass von Meta-Informationen wie Zeitstempeln und Blöcken, die als unbenutzt gekennzeichnet sind, keine Gefahr ausgeht, dann bieten die Hashes der Dateien eine equivalente Sicherheit. Damit letztendlich wieder ein Hash-Wert gespeichert werden kann, könnte ein Hash über der Liste von Dateinamen nebst deren Hash-Werten gebildet werden.

Diese Funktionalität lässt sich sehr einfach mit Standard-Unix-Programmen zusammensetzen:

```
$ find /boot -type f
/boot/grub/device.map
/boot/grub/stage1
/boot/grub/stage2
[...]
```

```
$ md5sum ‘find /boot -type f‘
27b7d761820b6de1a9589a1c4c0ea69c  /boot/grub/device.map
c05824e4047360389cf7b71296117e54  /boot/grub/stage1
68a8bfd5a1ce16ecd90435b892f54952  /boot/grub/stage2
[...]

$ md5sum ‘find /boot -type f‘ | md5sum
b987b29dce9662b43b9c72a7a659c492  -
```

Das Programm `find` liefert alle Dateien in dem Pfad `/boot` und die Liste der Dateinamen wird an `md5sum` übergeben, welches einen Hash-Wert für jede Datei erzeugt. Zusammen mit den Dateinamen wird diese Liste dann wieder zu einem Hash-Wert zusammengefasst. Das Programm `md5sum` steht hier nur beispielhaft für jeden möglichen Hash-Algorithmus.

Bei einer Manipulation einer einzelnen Datei – beispielsweise des `initramfs`-Archivs – wird die Liste der Hashes verändert und damit ist auch der akkumulierte Hash-Wert ein anderer. Dies gilt auch, wenn Dateien hinzugefügt oder entfernt werden.

Das Bilden der Hash-Werte für alle einzelnen Dateien ist also ebenfalls sehr sicher. Wenn der Benutzer (beispielsweise aus Sicherheitsgründen) die `/boot`-Partition aber grundsätzlich immer nur lesend einhängt, wird sie sich nicht unbeabsichtigt verändern. Daher kann in diesen Fällen der Aufwand reduziert werden, indem nur über die gesamte Partition ein einzelner Hash berechnet wird. Dies ist weniger rechenaufwändig und es ist nicht mit absoluter Sicherheit auszuschließen, dass Modifikationen außerhalb der Dateiinhalte zu einem Angriff genutzt werden könnten. Deshalb sollte ein praktisches Programm beide Arten von Hash-Berechnungen unterstützen, den Hash über der gesamten Partition und die akkumulierten Datei-Hash-Werte.

6.4 Auffüllen des Speicherplatzes

Wie in einem späteren Abschnitt gezeigt wird, kann ein Angriff auf die kryptografische Sicherung darin bestehen, das alte korrekte Abbild der `/boot`-Partition vorzuhalten und laufenden Programmen dieses Abbild zu präsentieren.

Um dies zu verhindern, kann man die Partition soweit mit schwer komprimierbaren Informationen auffüllen, dass es faktisch unmöglich wird, das komprimierte vollständige Abbild der alten Partition zusätzlich zu dem neuen veränderten Kernel in einer Partition derselben Größe zu speichern.

Dazu könnte eine neue Datei erstellt werden, die mit Zufallszahlen angefüllt

wird, bis kein freier Speicherplatz mehr vorhanden ist. Es ist sehr unwahrscheinlich, dass ein guter Zufallsdatenstrom gut komprimierbare Teile enthält. Für zusätzliche Sicherheit könnte der Zufallsdatenstrom durch ein Kompressionsprogramm geschickt werden. Alternativ könnte man in diese Datei bereits bestehende gut komprimierte Daten kopieren. Als Quelle würden sich beispielsweise Videofilme sehr gut eignen. Video-Encoder-Programme haben das Ziel, die Videodaten so gut wie möglich zu komprimieren und einen möglichst großen Informationsgehalt in einen gegebenen Speicherplatz zu packen. Mit solchen Daten angefüllt könnte es sich als unmöglich erweisen, ein Abbild der korrekten Partition in die modifizierte zu integrieren.

6.5 Aufruf des Programms

6.5.1 Einbindung in `/etc/rc.local`

Um beim Systemstart eigene Kommandos ausführen zu lassen, gibt es unter den meisten Linux-Distributionen bestimmte Dateien, die allein dafür bestimmt sind, vom Systemadministrator mit den notwendigen Befehlen gefüllt zu werden und sonst keinerlei Code enthalten. Diese werden üblicherweise benutzt, um bestimmte Aktionen auszulösen, die bei der konkreten Installation notwendig sind und nicht in allgemeiner Form in bestehende Programme implementiert werden können oder sollen.

Die Debian-Distribution besitzt für diesen Zweck die Datei `/etc/rc.local`. Die einfachste Variante für den Aufruf des Testprogramms wäre ein Eintrag an dieser Stelle. Die Datei sähe dann wie folgt aus:

```
#!/bin/sh

# System-notwendige Befehle ...

/usr/bin/testprogramm
```

6.5.2 Eigenes Init-Skript

Aufwändiger, aber sauberer ist die Implementierung mit Hilfe eines eigenen Init-Skripts. Diese werden für gewöhnlich verwendet um im Hintergrund laufende Programme, die Unix-Daemons, zu steuern. Für Debian paketierte Programme liefern oft ein Shell-Skript in `/etc/init.d/` mit, welches die eigentliche Binärdatei starten oder beenden kann. Dafür wird das Skript mit entsprechenden Argumenten wie `start` oder `stop` aufgerufen. Dies ermöglicht einen generischen Umgang mit Daemons durch Verwaltungsprogramme ohne Kenntnis über Eigenheiten der Daemons.

Ein solches Init-Skript könnte für das Testprogramm angelegt werden und mit entsprechenden Soft-Links in den Verzeichnissen für die speziellen Unix-Runlevel würde es bei jedem Systemstart und beim Herunterfahren aufgerufen. Vereinfacht müsste es so aussehen:

```
#!/bin/sh

case "$1" in
    start)
        # Hash ueberpruefen
        # ggf. Warnmeldung ausgeben
        exit 0
        ;;
    stop)
        # Hash speichern
        exit 0
        ;;
    *)
        echo $0 (start|stop)
        exit 1
        ;;
esac
```

Der Vorteil dieser Variante besteht darin, dass die neuen Shell-Skriptdateien in einem Paket für den Paketmanager der Distribution enthalten sein können und so automatisch installiert oder deinstalliert werden. Außerdem ist eine leichte Versionsverwaltung möglich. Es ist der vielleicht sauberste Weg, in eine bestehende Linux-Installation ein neues Programm einzubinden, welches bei Änderungen des Runlevels gestartet werden soll.

6.5.3 Einbindung in bestehende Programme

Eine Alternative stellt die Einbindung des eigenen Codes in ein bereits existierendes Shell-Skript dar, das bereits automatisch gestartet wird.

Allerdings erfordert diese Methode einige Handarbeit, die nicht von Skripten automatisch ausgeführt werden sollte, und sie geht am Paketmanager der Distribution vorbei. Das bedeutet, dass beispielsweise Updates für die betreffende Datei ein Eingreifen des Administrators erforderlich machen. Diese Methode ist als sehr unsauber anzusehen, hat aber einige Vorteile, wie im Abschnitt 7.4 gezeigt wird.

Zudem hängt diese Methode davon ab, welche Pakete auf dem konkreten Zielsystem installiert sind. Es muss ein Daemon vorhanden sein, der beim Systemstart

gestartet und beim Herunterfahren beendet wird. Außerdem sollte er besser nicht zwischendurch neugestartet werden, also kein temporärer Dienst sein.

Der versierte Administrator kann dann den erforderlichen Teil Shell-Code, der üblicherweise in einem Extra-Skript liegt, in ein von ihm gewähltes existentes Shell-Skript einfügen. Dabei ist darauf zu achten, dass enthaltene `exit`-Befehle nicht zum vorzeitigen Abbruch des Skripts führen.

Diese Methode sollte generell nur von Administratoren benutzt werden, die genau über die Interna der bearbeiteten Shell-Skripte Bescheid wissen und ungewollte Seiteneffekte ausschließen können.

6.6 Deployment

Der Code einer konkreten Implementierung des hier skizzierten Programms kann selbstverständlich ganz einfach als Quellcode-Archiv – für gewöhnlich ein sogenanntes *tarball* (`.tar`-Archivdatei) – bereitgestellt werden. Wird das Programm mit einem Versionskontrollsystem wie Git entwickelt, könnte zusätzlich das entsprechende Repository publiziert werden, um eine Weiterentwicklung oder Korrektur zu vereinfachen.

Um aber das Programm leichter anwenden zu können, ist es empfehlenswert, ein Paket für den Paketmanager einer Distribution herzustellen. Für Debian GNU/Linux wäre dies das `.deb`-Format.

Dazu werden alle zu installierenden Dateien innerhalb eines temporären Verzeichnisses mit vollem Pfad installiert. Dieses Verzeichnis enthält dann die bekannten Unterordner `/bin`, `/usr` etc., aber jene enthalten nur vom Programm bereitgestellte Dateien. Zusätzlich werden Dateien mit Meta-Informationen wie einer Prosa-Beschreibung des Inhalts und Paketabhängigkeiten dort abgelegt. Dieses Verzeichnis wird dann in ein Debian-Paket überführt, wozu es hauptsächlich in ein `.tar`-Archiv gepackt wird. Weitere Details bieten die Dokumentation zu den einzelnen Distributionen und ihrer Paketmanager.

7 Angriffsszenarien II

In Kapitel 4 wurde gezeigt, welche Mittel für weitere Attacken einem Angreifer zur Verfügung stehen, wenn er unbemerkt die `/boot`-Partition modifizieren kann.

Auch bei Einsatz einer Implementierung des in dieser Arbeit dargestellten Testprogramms, ergeben sich weitere potenzielle Schwachstellen, bei denen bestimmte Attacken denkbar sind. In diesem Kapitel soll analysiert werden, welche Attacken auf das Testprogramm möglich sind und welche Gegenmaßnahmen wiederum sinnvoll sind.

7.1 Herbeiführen von Hash-Kollisionen

Wie in Abschnitt 5.2 gezeigt wurde, ist es sehr unwahrscheinlich oder extrem aufwendig, eine veränderte `/boot`-Partition so zu modifizieren, dass sie denselben Hash-Wert wie die unveränderte hat. Natürlich kann diese Möglichkeit nicht vollständig ausgeschlossen werden. Deshalb wurde bereits die Verwendung mehrerer Hash-Algorithmen in Erwägung gezogen.

Versucht ein Angreifer trotzdem eine Hash-Kollision herbeizuführen, benötigt er dafür maximale Zeit mit maximaler Rechenleistung. Da Rechenleistung durch Faktoren wie Investitionsvolumen, Verfügbarkeit et cetera begrenzt ist, führen gute oder viele Hash-Algorithmen zu einer längeren benötigten Zeit bis zum Finden einer Kollision.

Eine zusätzliche Sicherungsmaßnahme könnte deshalb darin bestehen, die `/boot`-Partition bewusst regelmäßig zu verändern und einen neuen Hash-Wert abzuspeichern. Dies begrenzt zusätzlich die verfügbare Zeit, die ein Angreifer hat, um mit Hilfe einer gefundenen Hash-Kollision einen weitergehenden Angriff durchzuführen. Wenn bereits der nächste Hash-Wert als korrekt gilt, ist eine alte gefundene Kollision wertlos geworden.

7.2 Vortäuschen einer falschen `/boot`-Partition durch Kernel-Modifikation

Eine Möglichkeit, die Entdeckung eines Angriffs bei gleichzeitiger ordnungsgemäßer Ausführung des Schutzprogramms zu verhindern, ist eine Kernel-Modifikation, die ein falsches Abbild der `/boot`-Partition wiedergibt.

Unix-Programme kommunizieren nicht selbst mit Geräten wie der Festplatte, sondern über vom Kernel bereitgestellte Gerätedateien, die die eigentlichen Geräte repräsentieren. Obwohl dies grundsätzlich sehr sinnvoll ist, ermöglicht es in diesem speziellen Fall, dass es ein manipulierter Kernel fälschlicherweise das alte Partitionsabbild wiedergibt.

Dazu müsste im Kernel der Treiber für Blockgeräte oder eine ähnliches Modul so umgeschrieben werden, dass es die Ausgaben für bestimmte Adressbereiche auf dem eigentlichen Datenträger umlenkt und stattdessen von einer internen Funktion bereitgestellte Daten kopiert. Diese Funktion müsste die ursprünglichen, unveränderten Daten beispielsweise aus einer komprimierten Datei im Dateisystem der `/boot`-Partition lesen.

Die Datei müsste deshalb komprimiert sein, weil auf die Partition mit der Größe von n Bytes die neuen veränderten Dateien m zuzüglich der vollständigen Information über die alte `/boot`-Partition von n Bytes passen müssen. Der Angreifer kann deshalb nur mit einem Komprimierungsverfahren $K()$ die logische Information in weniger Bytes ausdrücken. Für den Fall, dass die `/boot`-Partition vor dem Formatieren mit Nullen überschrieben wurde (beispielsweise eine neue Festplatte), wäre es auch möglich, dass sich große, gut komprimierbare Bereiche von Nullen im Dateisystem befinden. Die Erfolgsbedingung für den Angreifer wäre dann, ein Komprimierungsverfahren zu finden, so dass gilt:

$$|n| \geq |K(n) \circ m| \tag{7.1}$$

Deshalb wurde im Abschnitt 6.4 eine Methode entwickelt, ungenutzten Speicherplatz mit möglichst viel schlecht komprimierbarer Information zu füllen. Wenn diese Technik angewandt wird, ist es höchst unwahrscheinlich, dass ein Integrieren des alten Abbildes gelingen kann. Offensichtlich sollte die `/boot`-Partition dafür grundsätzlich von minimaler Größe sein und der gesamte verfügbare Speicherplatz sollte von verschlüsselten Dateisystemen belegt sein, damit dort nicht ohne auffällige Seiteneffekte Daten abgelegt werden können.

7.3 Modifikation von Hash-Programmen oder -Bibliotheken

Weil die Sicherheit des beschriebenen Überprüfungssystems auf dem Vergleich von Hash-Werten beruht, ist es grundsätzlich gar nicht notwendig, die gesamte Information über die unveränderte `/boot`-Partition vorzuhalten. Stattdessen könnte es ausreichen, die Berechnung der Hash-Werte zu manipulieren, so dass auch bei einer falschen Eingabe der gesuchte Hash-Wert zurückgegeben wird.

Zur Umsetzung müsste entweder die entsprechende Binärdatei wie `/usr/bin/sha1sum` für SHA-1 ausgetauscht oder das entsprechende Kernelmodul modifiziert werden. Beides ist grundsätzlich möglich und für einen Angreifer, der das `initramfs`-Archiv verändern kann, durchführbar. Es ist jedoch fraglich, ob auf diesem Weg effektiv eine Verschleierung des Angriffs möglich ist.

Für die Implementierung des Hash-Algorithmus ist nicht ohne weiteres zu erkennen, woher die Daten kommen, die beispielsweise über eine Unix-Pipe übergeben werden. Das Programm kann jedoch nicht für alle Eingaben den falschen Hash-Wert liefern, weil sonst viele legitime und notwendige Abfragen einen vermeintlichen Fehler detektieren würden. Der Algorithmus müsste also selektiv für die Byte-Folge der veränderten `/boot`-Partition den Hash-Wert der unveränderten liefern. Es müsste während der Eingabe fortlaufend ein Binärvergleich zwischen den Eingabe-Bytes und den Daten auf der `/boot`-Partition durchgeführt werden. Bei Übereinstimmung würde der falsche, gespeicherte Hash-Wert ausgegeben.

Der Angreifer kann aber auf diesem Wege nicht sicherstellen, dass das eingeschleuste manipulierte Programm überhaupt aufgerufen wird. Das Testprogramm könnte eine andere Kopie des Hash-Programms über den vollständigen Pfad aufrufen oder der Hash-Algorithmus wird im Testprogramm selbst implementiert.

Diese Vorgehensweise führt also nicht mit absoluter Sicherheit zum Erfolg.

7.4 Verhinderung der Ausführung des Programms

Eine grundlegende Angriffsstrategie wäre es, zu verhindern, dass das Testprogramm beim Systemstart ausgeführt wird. Wird dieses nicht aufgerufen, kann es nicht die Hash-Wert-Vergleiche durchführen und gegebenenfalls eine entsprechende Warnung ausgeben.

Dieser Weg erscheint zunächst recht zuverlässig und ist grundsätzlich durchführbar. Soll ein neues Programm gestartet werden, sendet ein bestehendes Programm dafür einen entsprechenden Befehl an den Kernel, welcher die ausführbare Datei in den Speicher lädt und einen es in einem neuen Prozess startet. Dafür bekommt der Kernel den Pfad zur ausführbaren Datei übergeben.

Ein Angreifer könnte den Kernel also so modifizieren, dass er selektiv die Ausführung eines bestimmten Programms wie beispielsweise `/usr/bin/testprogramm` verweigert. Stattdessen könnte er an das aufrufende Programm einfach den Rückgabewert `0` melden und so mitteilen, das gewünschte Programm wäre fehlerfrei beendet worden.

Es gibt jedoch einfache Möglichkeiten dem Angreifer diesen Weg stark zu erschweren.

Zuerst könnte man das Testprogramm an einem zufällig gewählten Ort mit einem zufälligen Namen ablegen, so dass weiterführende Vergleiche notwendig wären, um das Testprogramm zu identifizieren. Beispielhaft seien hier die Dateien in `/etc/init.d/` genannt. In diesem Verzeichnis liegen in einem Debian-GNU/Linux-System die Start- und Stop-Skripte für Unix-Daemons und ähnliche Programme. Wird für das Testprogramm ein zufälliger Name gewählt und das Programm dort abgelegt, ist es durch den Aufrufpfad nicht von legitimen Programm zu unterscheiden.

Der Angreifer müsste weiterhin eine Inhaltsanalyse implementieren, um herauszufinden, ob es sich bei dem aufgerufenen Programm, um das Testprogramm handelt. Dies entspricht in etwa der Technik eines signaturbasierten Viren-Scanners, wie er von vielen Software-Herstellern angeboten wird. Im Feld der Computer-Viren haben sich in der Vergangenheit Techniken entwickelt, um eine Identifizierung durch solche Suchprogramme zu verhindern. Dazu gehört das Einfügen von zufälligen Daten, die eine Prüfsumme verändern, oder das Umschreiben von Shell-Code während der Laufzeit des Programms (Polymorphie). Diese Techniken könnten eingesetzt werden, um das Testprogramm für die `/boot`-Partition zu verstecken.

Zusätzlich könnte der Code für die Überprüfung in ein anderes legitimes Shell-Skript eingefügt werden. Der Kernel erhält nur den Befehl, den Shell-Interpreter (beispielsweise `/bin/bash`) mit einem gewöhnlichen Start-Skript als Argument zu starten (`/etc/init.d/kdm`, der Session-Manager von KDE). Der modifizierte Kernel müsste den Befehl tatsächlich ausführen, um keine auffälligen Fehlfunktionen des Computersystems auszulösen, und könnte auf diesem Wege nicht den Test umgehen.

8 Fazit

Full Disk Encryption ist eine praktisch einsetzbare und akzeptabel sichere Methode, vertrauliche Daten vor unbefugtem Zugriff zu schützen. Für den alltäglichen Einsatz auf grundsätzlich gefährdeten Systemen wie mobilen PCs ist eine einfache Verschlüsselung meist ausreichend.

Bei besonders interessanten Computersystemen, die Daten von besonderem Interesse beinhalten, werden weitergehende Sicherheitsüberlegungen wichtig. Wo Daten verarbeitet werden, die Einzelpersonen interessieren, welche konkrete Attacken vorbereiten, kann die `/boot`-Partition ein mögliches Einfallstor zum unbemerkten Brechen der Verschlüsselung sein.

Ein System, wie es in dieser Arbeit konstruiert wurde, kann dabei helfen Manipulationen zu entdecken. Aufgrund des Ausgangsproblems, dass manipulierter Code aus dem unverschlüsselten Bereich der Festplatte *vor* dem sicheren verschlüsselten Code ausgeführt wird, können weitere Attacken gegen das Schutzprogramm selbst nicht ausgeschlossen werden. Eine nachträglich ablaufende Überprüfung kann lediglich ein Indiz für die Unversehrtheit des Systems sein.

Kryptografische Strategien und mathematische Überlegungen machen es einem Angreifer schwerer, eine Veränderung zu verbergen. Indem man Details der Implementierung des Schutzprogramms – beispielsweise die Stelle, an der das Schutzprogramm ausgeführt wird – geheimhält oder variiert, kann man die Sicherheit zusätzlich erhöhen. Dies ist jedoch *Security by Obscurity* und widerspricht teilweise dem Grundgedanken der universitären Kryptografie, dass Verfahren nur dann als sicher gelten können, wenn möglichst viele Experten sich von ihrer Sicherheit überzeugt haben.

Der vorgestellte Sicherheitsmechanismus ist also kein perfekter Schutz, sondern soll das bestehende Risiko dort zu begrenzen helfen, wo alternative Sicherheitsmaßnahmen nicht angewendet werden können.

Literaturverzeichnis

[1] heise online. Verschusselt statt verschlüsselt. Website. Online abgerufen unter `http://www.heise.de/security/artikel/Verschusselt-statt-verschluesselt-270058.html` am 3.8.2010.

[2] Joanna Rutkowska. Evil Maid goes after TrueCrypt! Website. Online abgerufen unter `http://theinvisiblethings.blogspot.com/2009/10/evil-maid-goes-after-truecrypt.html` am 3.8.2010.

[3] Bruce Schneier. *Angewandte Kryptographie.* Pearson Studium, München, 2006.

Abkürzungsverzeichnis

AES Advanced Encryption Standard

BIOS Basic Input Output System

CPU Central Processing Unit

FDE Full Disk Encryption

GDM GNOME Display Manager

GNOME GNU Network Object Model Environment

GRUB Grand Unified Bootloader

KDE K Desktop Environment

KDM KDE Display Manager

LUKS Linux Unified Key Setup

LVM Logical Volume Manager

MBR Master Boot Record

MiB Mebibyte (1 MiB = 2^{20} Bytes $\approx 10^6$ Bytes)

PRNG Pseudo Random Number Generator

RAID Redundant Array of Independent Disks

RFID Radio Frequency Identification

SATA Serial ATA (Advanced Technology Attachment)

SSH Secure Shell

Tabellenverzeichnis

6.1 Verschiedene Algorithmen im Vergleich 18

Abbildungsverzeichnis

3.1 Die Verteilung der unterschiedlichen Container im Datenträger 10

Selbstständigkeitserklärung

Ich versichere, dass ich die vorliegende Studienarbeit ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form weder veröffentlicht, noch einer anderen Prüfungsbehörde vorgelegt.

____________________ ___________________________________

Ort, Datum Unterschrift