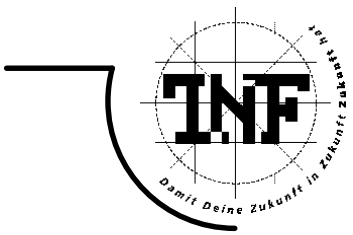
JOHANNES KEPLER
UNIVERSITÄT LINZ
Netzwerk für Forschung, Lehre und Praxis



File Slack Analyse unter Linux

Seminararbeit

Sicherheitsaspekte in der Informatik

LVA-Nummer: 353.053, SS2006

Angefertigt am *Institut für Informationsverarbeitung und Mikroprozessortechnik*

Betreuung:

Prof. Jörg R. Mühlbacher
Dipl.Ing. Rudolf Hörmanseder

Eingereicht von:

Florian Wörter

Linz, October 16, 2006

Kurzfassung

Diese Arbeit beschäftigt sich mit der Analyse von File Slack unter Linux. In der Einleitung wird darauf eingegangen was File Slack ist, welche Arten von File Slack es gibt, und wie es zur Entstehung von File Slack kommt. Anschließend werden die in den Tests verwendeten Tools kurz vorgestellt. Im Hauptteil der Arbeit werden die Dateisysteme *Ext2*, *Ext3*, *FAT32* und *ReiserFS* auf ihre File Slack Eigenschaften untersucht. Als Abschluss wird ein Algorithmus präsentiert, welcher beschreibt, wie man die File Slack Eigenschaften eines beliebigen Dateisystems untersucht.

Abstract

This paper describes the analysis of file slack in a Linux environment. At first a short introduction is given about what file slack is, which types of file slack exist and why file systems produce file slack. After that the tools are presented which were used during testing. In the main part of this paper several file systems (*Ext2*, *Ext3*, *FAT32* and *ReiserFS*) are presented and investigated upon their file slack properties. At the end an algorithm is shown which helps you to investigate any file system upon its file slack properties.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Allgemeines

File Slack ist ein Thema, das in vielen Büchern und Publikationen angedeutet wird, jedoch wird nirgends definitiv erläutert, welche Daten die einzelnen Betriebs- und Dateisysteme in die nicht benötigten Bereiche der Datenblöcke schreiben. Ziel dieser Arbeit ist es, zu erklären was *File Slack* ist und einige Linux Dateisysteme auf ihre *File Slack* Eigenschaften zu untersuchen. Am Ende wird ein abstrakter Algorithmus vorgestellt, wie man ein beliebiges Dateisystem auf *File Slack* Eigenschaften prüfen kann.

1.2 Was ist File Slack?

Wenn ein Benutzer eine neue Datei angelegt, so hängt ihre tatsächliche Größe vom Inhalt der Datei ab. Jedoch teilen moderne Dateisysteme aus Gründen der Performanz und Effektivität den zur Verfügung stehenden Datenträgerspeicher in Cluster oder Datenblöcke ein. Wird nun die neu erstellte Datei auf der Festplatte gespeichert, so wird die Datei nur in seltenen Fällen genau in n Clustern Platz findet. Meistens wird ein gewisser Teil des letzten für die Datei reservierten Datenblockes nicht benutzt. Genau dieser nicht benutzte Bereich vom Ende des tatsächlichen Dateiinhaltes bis zum Ende des letzten für die Datei reservierten Clusters wird laut Armor Forensics [6] als *File Slack* bezeichnet. Andrew S. Tanenbaum [8] zeigt, dass die Clustergrösse von dem involvierten Betriebssystem und im Fall von Windows 9x auch von der Größe der logischen Partition abhängig ist. Je grösser die Clustergrösse gewählt wird, desto mehr Verschnitt und somit auch mehr *File Slack* wird sich auf dem Datenträger wiederfinden. Dieser Umstand stellt zwar ein sicherheitstechnisches Problem dar, die *File Slack* Analyse wird dadurch jedoch um einiges einfacher und wird auch mehr Daten enthalten als bei kleineren Clustergrössen. In den *File Slack* Bereich werden vom Betriebssystem aus Gründen der Performanz (nicht alle Bytes in einem Puffer werden auf 0 gesetzt) potentiell Speicherinhalte hineingeschrieben. Dies ist der Fall, da ein Betriebssystem immer gepuffert auf den Datenträger schreibt. Die Größe des Schreibpuffers entspricht normalerweise der Sektorgrösse des Speichermediums. Cluster wiederum bestehen aus einer Menge

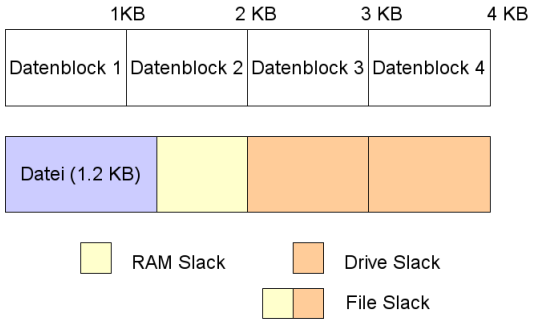


Abbildung 1.1: File Slack

von Sektoren. Wenn die Datei zu klein ist, um den letzten beanspruchten Sektor ganz aufzufüllen, wird dieser unter Umständen mit den Inhalten des Schreibpuffers des Betriebssystems aufgefüllt. Diese zufällig aus dem Arbeitsspeicher kopierten Daten werden auch als *RAM Slack* bezeichnet. Dieser *RAM Slack* kann alle möglichen Informationen beinhalten: Daten, die erstellt, angesehen, heruntergeladen oder kopiert wurden, Teile von E-Mails etc. Potentiell können alle Daten im Arbeitsspeicher stehen, die mit dem Computer seit dem letzten Bootvorgang bearbeitet wurden. Dazu gehören natürlich auch Benutzernamen, Passwörter etc. Die Inhalte des Arbeitsspeichers betreffen nicht nur den aktuell angemeldeten Benutzer. Wie schon erwähnt können alle möglichen Informationen aller seit dem Bootvorgang des Rechners angemeldeter Benutzer im Arbeitsspeicher enthalten sein.
Jedoch wird nur der letzte Sektor der Datei mit Inhalten aus dem RAM aufgefüllt. Sollte der letzte der Datei zugeordnete Cluster noch nicht voll sein, so entsteht eine weitere Art von Slack: *Drive Slack*. Die Inhalte der Sektoren des Drive Slacks werden einfach nicht verändert. Dadurch enthalten sie zwar keine Inhalte des Arbeitsspeichers, dafür aber Inhalte von auf dem Datenträger zuvor gespeicherten Daten. Meistens sind das Überreste von zuvor gelöschten Dateien.
File Slack entsteht beim Erstellen oder Verändern einer Datei. Wird die Datei gelöscht, so bleibt der File Slack auf dem Datenträger gespeichert. Die Cluster der gelöschten Datei werden als frei gekennzeichnet und können vom Dateisystem neuen oder bestehenden Dateien zugeordnet werden. Es ist dabei nicht unwahrscheinlich, dass der *RAM Slack* einer gelöschten Datei auf der Festplatte als Drive Slack einer neuen Datei gespeichert bleibt. In der Abbildung 1.1 werden die Zusammenhänge zwischen den Begriffen *File Slack*, *RAM Slack* und *Drive Slack* veranschaulicht.
File Slack ist ein sicherheitstechnisches Problem, das man nicht unter den Tisch kehren sollte. Zwar wird nur ein Teil eines Sektors mit *RAM Slack* und ein paar weitere Sektoren mit *Drive Slack* belegt, jedoch kann bei einer heute üblichen Festplatte *File Slack* eine Größe von mehreren hundert Megabytes erreichen.

Kapitel 2

File Slack unter Linux

2.1 Die Gerüchteküche brodelt

Wie in der Einführung erwähnt wird, können sich praktisch beliebige Inhalte des Arbeitsspeichers im *File Slack* befinden. Im Prinzip alles von Textdokumenten über E-Mails bis hin zu Netzwerkbenutzernamen. Jedoch stellt sich die Frage, ob bei modernen Betriebssystemen immer noch auf das Löschen von Puffern verzichtet wird. Werden tatsächlich sicherheitskritische Inhalte des Arbeitsspeichers auf den Datebträger geschrieben? Dass gelöschte Dateien beim Löschvorgang nicht wirklich überschrieben werden, ist ein bekannten Phänomen. Auf diese Tatsache bauen alle Werkzeuge zur Datenwiederherstellung auf. In dieser Arbeit wird durch die Analyse einer Testpartition gezeigt, welche Inhalte sich bei den getesteten Dateisystemen tatsächlich auf der Festplatte befinden.

Kapitel 3

Im Test eingesetzte Tools

3.1 The Sleuth Toolkit

Hierbei handelt es sich um eine Reihe von UNIX basierten Kommandozeilentools. Zu ihnen gehören zum Beispiel *fls*, welches gelöschte und nicht gelöschte Dateien eines Verzeichnisses auflistet oder *fsstat*, welches allgemeine Informationen zum Dateisystem anzeigt. Dieses Toolkit bildet die Basis für die meisten *File Slack* bezogenen Tests.
Zu finden ist dieses Toolkit unter [3].

3.2 Autopsy

Dieses Tool ist ein graphisches Frontend für *The Sleuth Toolkit*. Nach Start in der Linux Shell verbindet man sich mit dem Webbrowser auf *localhost*. Man kann neue Untersuchungen (*Cases*) anlegen, wobei ein *Case* einem Dateisystemscan entspricht. Wurde das Dateisystem gescannt, so kann man die einzelnen Bytes des Datenträgers betrachten und gezielte Suchen nach Inhalten vornehmen. Das System arbeitet byteorientiert und kann so auch *File Slack* Bereiche wiedergeben. Erhältlich ist das Tool unter [2].

3.3 Automated Image and Restore

AIR ist ein kleines Tool, welches dazu verwendet wird, bitgenaue Abbilder (Images) von Partitionen zu erzeugen. Dabei werden nicht, wie bei herkömmlichen Kopierprogrammen, die Dateien einzeln kopiert, sondern es wird eine tatsächliche Bit-Kopie des Datenträgers erstellt (inklusive aller *File Slack* Informationen). Dieses Werkzeug wurde für das Erstellen der in den einzelnen Tests benötigten Images der Testpartition verwendet.
Dieses Tool ist erhältlich unter [1].

3.4 Teststrategien

3.4.1 Blackbox Tests

Als Testdatenträger wurde eine 32 MB große Partition gewählt, die abwechselnd mit den verschiedenen Dateisystemen formatiert wurde. Bei FAT32 wurde die Größe auf 128MB erhöht. Die kleine Größe bringt einige Vorteile mit sich. Zum einen minimiert sich dadurch die benötigte Rechenzeit diverser Tools, und zum anderen ist der Datenträger schneller zu füllen.
Nach der Formatierung mit dem jeweiligen Dateisystem wurde ein Image der frisch formatierten Testpartition erstellt. Nach dem Kopieren der Testdaten wird ein neues Image erstellt. Ein Hauptteil der Analyse besteht aus dem Vergleich dieser beiden Images. Dieser Vorgang wird für verschiedene Dateigrößen durchgeführt.

3.4.2 Whitebox Tests

Bei dieser Art von Tests versucht man unter Kenntniss des Quelltextes der einzelnen Dateisystemimplementierungen die einzelnen Tests so zu gestalten, dass *File Slack* auftreten kann. Weiters wird versucht, im Quelltext für *File Slack* relevante Passagen oder Kommentare zu entdecken.

Kapitel 4

Die Dateisysteme im Vergleich

Unter Linux werden eine Vielzahl an unterschiedlichen Dateisystemen eingesetzt. Aus Zeitgründen können in dieser Arbeit leider nur die wichtigsten Vertreter der unter Linux gängigen Dateisysteme untersucht werden. *Ext2/Ext3* wurde ausgewählt, da diese Dateisysteme oft als *die* Linux Dateisysteme bezeichnet werden. Mit *FAT32* wird ein Dateisystem vorgestellt, das eigentlich für Windows entwickelt wurde. Jedoch wird dieses Dateisystem auch im Linux Kernel implementiert und häufig für Datenträger, die von Windows und Linux in Multibootumgebungen verwendet werden, eingesetzt. Den Abschluss bildet das platzeffiziente Dateisystem *ReiserFS*.

4.1 Ext2 / Ext3

Das *Second Extended File System* war über Jahre hinweg das Standarddateisystem von Linux. Heute wird es meistens nur noch für kleinere Partitionen eingesetzt. Es wurde sowohl von seinem Nachfolger *Ext3* als auch von anderen Dateisystemen abgelöst, da diese zusätzlich über ein *Journal* verfügen und so Daten im Falle eines Dateisystem-Crashs besser wiederherstellen können.
Das *Third Extended File System* (*Ext3*) unterscheidet sich nur durch die Verwendung eines Journals von seinem Vorgänger und wird deshalb oft auch als Erweiterung des *Ext2* Systems gesehen. In der Praxis ist es möglich, ein *Ext3* Dateisystem als *Ext2* zu mounten, wobei das *Journal* natürlich nicht zur Verfügung steht.

Wie alle klassischen Unix Dateisysteme basieren auch *Ext2* und *Ext3* auf *Blöcken* und *I-Nodes*. *I-Nodes* sind sogenannte Index-Knoten, die jeweils einer Datei zugeordnet sind. In ihnen werden alle Metainformationen zu einer Datei und eine Liste der verwendeten Datenblöcke gespeichert. Das Dateisystem ist in sogenannte Blockgruppen unterteilt. In jeder Blockgruppe befindet sich der *Superblock*, ein *Gruppendeskriptor*, ein *Blockbitmap*, ein *I-Node Bitmap*, die *I-Nodes* und die *Datenblöcke*. Die Bitmaps werden dazu verwendet, die freien Datenblöcke bzw. I-Nodes zu verwalten.

Wie schon erwähnt, arbeitet das *Second Extended File System* auf Block-Basis.

Die standardmäßige Blockgröße beträgt 1024 Bytes. Somit ist die Grundvoraussetzung für *File Slack* gegeben.

Der Testablauf

Als erstes wurde die Testpartition mit Hilfe des Linux Programems *mke2fs* formatiert. Nach der Formatierung wurde mit Hilfe des Tools *AIR* (siehe Kapitel 3 auf Seite 7 „Im Test eingesetzte Tools“) eine exakte Kopie der Partition erstellt und in einem Hexeditor untersucht. Offensichtlich wurde der erste Datenblock nicht verändert (hier würde bei einer bootfähigen Partition der Bootcode stehen). Alle Dateisysteminformationen und *I-Node* Tabellen wurden erstellt, jedoch wurden keine älteren Dateninhalte gelöscht. D.h. Dateisystemverwaltungsstrukturen wurden auf die Partition geschrieben, aber Bereiche, die nicht für die Verwaltung der Daten gebraucht werden, werden auch nicht überschrieben. So ist es sicherlich nach einer Formatierung auch möglich, Daten wiederherzustellen.
Da noch viele Datenbereiche der Testpartition mit Nullbytes gefüllt waren (die Testpartition wurde auf einer noch wenig benutzten Festplatte erstellt) war es notwendig, die Datenbereiche der Festplatte zu füllen, um anschließend feststellen zu können, ob alte Datenbereiche mit Nullbytes gelöscht wurden oder unverändert blieben. Das *Ext2* Dateisystem versucht, Datenblöcke von Dateien in einem Unterverzeichnis in die selbe Blockgruppe zu geben, in der sich das Unterverzeichnis selbst befindet. Um die Datenblöcke effektiv zu füllen, wurde ein Unterverzeichnis erstellt und mehrere Male zufällige Dateien hineinkopiert, bis die Partition gefüllt war. Anschließend wurden die Dateien wieder gelöscht und mit dem Kopieren wieder von vorne begonnen. Um die Sache etwas zu beschleunigen, wurde dieses Füllen und Leeren des Dateisystemes mit Hilfe eines kleinen Shell-Skriptes erledigt, das Dateien verschiedener Größen auf die Testpartition kopierte und anschließend wieder löschte.
Nach einigen Durchläufen wurde wieder ein Image der Testpartition erstellt und in einem Hexeditor untersucht. Das Ergebnis war zufriedenstellend, da nun das Dateisystem auf den ersten Blick bereits eine große Ansammlung von Datenmüll erkennen ließ. Es waren keine längeren Folgen von Nullbytes mehr auszumachen und viele Inhalte bereits gelöschter Dateien waren im Hexeditor noch klar ersichtlich. Anschließend wurde das Testdateisystem mit *Autopsy* (siehe Kapitel 3 auf Seite 7 „Im Test eingesetzte Tools“) untersucht. Tatsächlich ist in diesem Werkzeug eine Liste von Dateinamen ersichtlich, die einmal gelöscht wurden und die wiederhergestellt werden könnten. Dies lässt auf das Vorhandensein von *Drive Slack* schließen. Nun wurde eine Textdatei mit bekannten Inhalt auf die Testpartition kopiert. Nach einem neuen Einlesen der Testpartition durch Autopsy konnten mit Hilfe der Suchfunktion die Inhalte der Testdatei auf dem Datenträger ausgemacht werden. So konnte festgestellt werden, dass alle Bytes, die zwischen Datei- und Datenblockende lagen, auf Null gesetzt wurden.

Ein Vergleich mit dem vor dem Kopieren der Testdatei erstellten Image zeigt, dass an dieser Stelle sich vor dem Kopieren noch keine Nullbytes befunden haben (siehe Abbildung 4.1 und Abbildung 4.2). Der Vorgang wurde einige Male wieder-

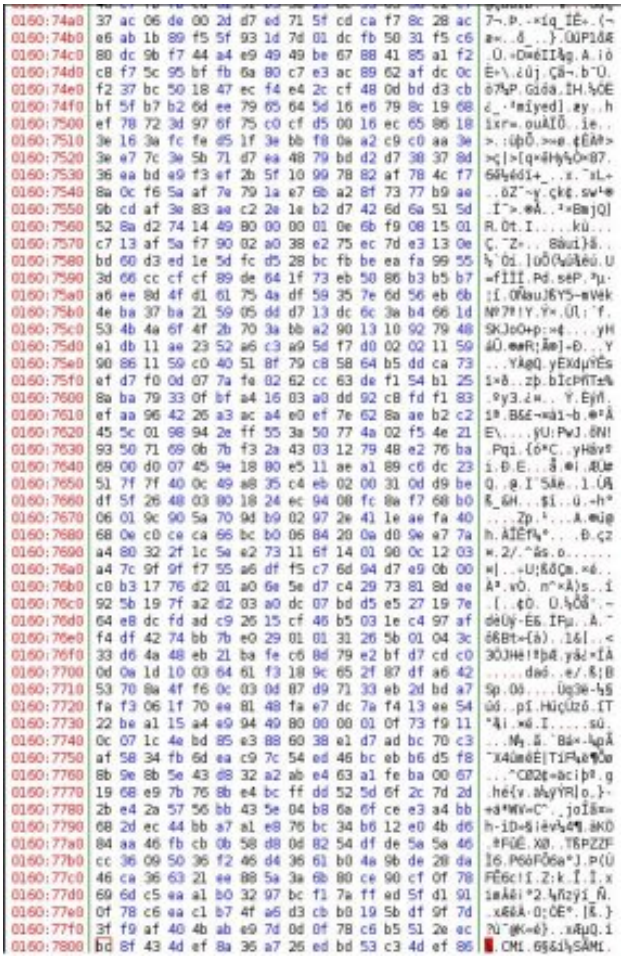


Abbildung 4.1: Stelle des Dateiendes vor dem Kopieren

holt, wobei sich immer das selbe Ergebnis abzeichnete: Der ungenützte Bereich des letzten Datenblockes wird mit Nullbytes aufgefüllt. Jedoch bleiben gänzlich unbenützte Datenblöcke, die der Datei folgen, unverändert. Das ist nicht weiter verwunderlich, da diese mit der Datei selbst (anders wie bei FAT32, wo die Einteilung in Cluster erfolgt) nichts mehr zu tun haben.

Ergebnis

Das Ext2 und Ext3 Dateisystem kopieren (soweit festgestellt werden konnte) keine zufälligen Speicherinhalte aus dem System RAM auf den Datenträger. Der letzte (meist nicht ganz belegte) einer Datei zugeordnete Datenblock wird mit 0-Bytes aufgefüllt. Nicht (mehr) verwendete Datenblöcke werden nicht gelöscht, was ein Wiederherstellen von Dateien ermöglicht. Jedoch werden in den *I-Nodes*, die eine gelöschte Datei einst repräsentiert haben, die Pointer auf die ehemaligen Datenblöcke zurückgesetzt. Dieser Umstand erschwert das Wiederherstellen von Dateien.

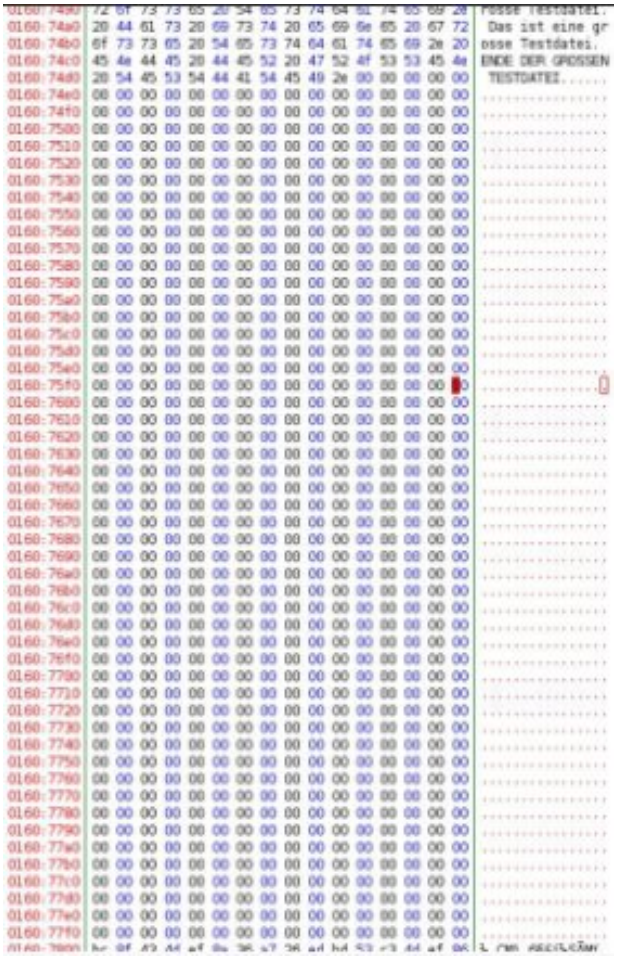


Abbildung 4.2: Stelle des Dateiendes nach dem Kopieren

Bei einer Formatierung werden Datenblöcke ebenfalls nicht zurückgesetzt.

Ext2

Dateigröße	RAM Slack	Drive Slack
Kleine Dateien	nein	JA
Große Dateien	nein	JA

4.2 FAT32

FAT32 ist zwar ein Windows-Dateisystem, wird aber aus Gründen des Datenaustausches zwischen Microsoft Windows und Linux oft auch unter Linux eingesetzt. Aus diesem Grund gibt es eine eigene Implementierung des FAT32 Datesystems im Linux Kernel, die hier untersucht werden soll.

FAT32 arbeitet mit einer Adressierung von 28 Bit (eigentlich müsste es deshalb FAT28 heissen, FAT32 klingt jedoch besser), wodurch es 268.435.456 Cluster adressieren kann. Ein Cluster besteht aus einer fixen Anzahl von Sektoren und ist (abhängig von der Partitionsgröße) in der Regel 4 KB groß. Ein Sektor bildet somit die kleinste Speichereinheit und hat ebenfalls eine fixe Größe (in der Regel 512 Bytes). Eine Datei wird somit Sektor für Sektor auf den Datenträger geschrieben. Die Granularität für das Dateisystem selbst bildet jedoch der Cluster, also eine auch nur 1 Byte große Datei belegt genau 4kB (Clustergröße) an Festplattenspeicher.
Das Herzstück des FAT32 Dateisystems ist die sogenannte *File Allocation Table*, die dem Dateisystem auch seinen Namen gibt. Das ist eine Tabelle, die in den Hauptspeicher geladen wird, in der vermerkt wird, welcher Cluster auf einen anderen Cluster folgt. Im Verzeichniseintrag wird zu einem Dateinamen jeweils der erste Cluster dieser Datei gespeichert. Beim Zugriff auf eine Datei wird der Index des jeweils nächsten Clusters von der *File Allocation Table* ermittelt. Ist ein Cluster der letzte Cluster einer Datei, so ist in der *File Allocation Table* (je nach Betriebssystem) ein bestimmter Wert (meist -1) vermerkt. Die Dateiatribute werden bei diesem Dateisystem (anders wie bei *I-Node* basierten Dateisystemen) direkt in den Verzeichniseinträgen gespeichert.
Die Tatsache dass dieses Dateisystem sowohl Sektoren als auch Cluster fixer Länge verwendet, lässt darauf hoffen, in puncto *File Slack* fündig zu werden.

Bei kleinen Dateien kommt es bei FAT32 zu einer viel höheren Bildung von File Slack, da eine Datei aus einer Menge von 4 kB Clustern besteht. Bei Ext2/Ext3 besteht eine Datei meist aus 1 kB Datenblöcken.

Der Testablauf

Für den Test wurde eine Testpartition von 64 MB verwendet, wobei die ersten 32 MB der Testpartition der für den Ext2/Ext3 Test verwendeten Partition entsprachen. Die Testpartition wurde unter Windows mit dem FAT32 Dateisystem formatiert. Anschließend wurde ein Image dieser frisch formatierten FAT32 Partition erstellt und in einem Hexeditor untersucht. Auffallend ist, dass Windows einige Einträge in den Boot-Sektor der Partition schreibt, was bei Ext2 nicht der Fall war. Beim Formatieren mit FAT32 werden auch keine Daten auf der Festplatte gelöscht, sondern nur die erforderlichen Verwaltungsstrukturen angelegt. So sind die Inhalte der letzten Testdateien im Hexeditor noch klar ersichtlich. Aus diesem Grund wurde bei diesem Test eine neue Testdatei verwendet, um feststellen zu können, was nun neu auf die Festplatte geschrieben wurde bzw. was noch an Datenschrott von den vorhergehenden Versuchen übrig ist.
Der Testablauf entsprach im Wesentlichen dem der Ext2/Ext3 Tests. Eine Testdatei wurde auf die Partition kopiert, ein Image erstellt und dieses mit dem vorherigen Image des FAT32 Dateisystems verglichen. Das Ergebnis war ähnlich des Ergebnisses der vorigen Tests bei Ext2/Ext3. Nach mehrmaliger Wiederholung stand fest, dass die FAT32 Implementierung des Linux Kernels die letzten Inhalte eines Sektors mit Nullen auffüllt und die restlichen Sektoren, die zum Auffüllen eines Clusters benötigt werden, unverändert belässt.

Das ist im Grunde ein zufriedenstellendes Ergebnis, das auch durch die Inspektion des Sourcecodes der FAT32 Implementierung im Linux Kernel (2.6.18) bestätigt wird. Der Sourcecode weist imemr wieder Methoden auf, in denen nicht benötigter Pufferplatz auf Null gesetzt wird.
Beispielsweise wird in der Datei *dir.c* in der Methode

```
static int fat_add_new_entries(struct inode *dir,
void *slots,
int nr_slots,
int *nr_cluster,
struct msdos_dir_entry **de,
struct buffer_head **bh,
loff_t *i_pos)
```

die für das Anlegen neuer Fat-Entries (Dateien) zuständig ist, mit der Zeile

```
memset(bhs[n]->b_data + copy, 0, sb->s_blocksize - copy)
```

darauf geachtet, dass der nicht benötigte Bereich des Puffers

```
(bhs[n]->b_data + copy)
```

auf 0 gesetzt wírd. Die Variable `copy` ist dabei die Anzahl der Bytes, die tatsächlich auf den Datenträger kopiert werden sollen. Auch beim Erstellen eines Verzeichnisses und anderen Funktionen wurde immer wieder Sourcecode gefunden, welcher nicht ganz gefüllte Puffer mit 0-Bytes auffüllt. Dieser Auffüllungsvorgang ist recht intelligent und performant gelöst. So wird nicht etwa der gesamte Puffer vor dem Füllen auf 0 gesetzt (was softwaretechnisch einfacher scheint), sondern es wird im Nachhinein ermittelt, wie viele Bytes nicht mehr benötigt werden und genau diese Bytes werden dann mit Nullbytes überschrieben. Das mag etwas komplizierter klingen, ist aber im Endeffekt performanter, als (egal wieviele Bytes in den Puffer geschrieben werden) vorher den gesamten Puffer mit Nullbytes zu initialisieren.

Ergebnis

Ähnlich wie bei Ext2/Ext3 wird auch bei der Linux Implementierung von FAT32 der letzte nicht mehr ganz benötigte Sektor mit Nullen aufgefüllt. Speicherinhalte des Arbeitsspeichers werden nicht auf das Dateisystem übertragen. Jedoch wird auch hier darauf verzichtet, nicht (mehr) verwendete Sektoren / Cluster zu bereinigen und auf Null zu setzen. Ganze nicht verwendete Sektoren werden einfach so belassen wie sie sind. Das erleichtert das Wiederherstellen von gelöschten Dateien ungemein.

FAT32 unter Linux

Dateigröße	RAM Slack	Drive Slack
Kleine Dateien	nein	JA
Große Dateien	nein	JA

4.3 ReiserFS

Das ReiserFS Dateisystem wurde ursprünglich von *Hans Reiser* mit dem Ziel entwickelt, eine performante Alternative zum gängigen *Ext2* Dateisystem zu sein. Außerdem soll dieses Dateisystem den Speicherplatz effizient nutzen und besser mit großen Verzeichnissen zurecht kommen, als andere Dateisysteme. Es basiert auf der von *Rudolf Bayer* entwickelten Datenstruktur des *B*-Baumes*, die verwendet wird, um die Verzeichnisse und Dateien zu verwalten. Wie andere Dateisysteme arbeitet auch das *ReiserFS* blockorientiert. Die Standardblockgröße unter Linux beträgt 4kB. Jedes Item (Datei, Verzeichnis, Meta-Block) im ReiserFS Dateisystem wird durch einen Schlüssel eindeutig identifiziert. Im Dateisystem gibt es Blöcke, die Teil des Verwaltungsbaums sind, und sogenannte *unformatierte Blöcke*. In den Blöcken, die dem Verwaltungsbaum zugeordnet sind, werden Schlüssel, Zeiger auf Kindknoten (andere Blöcke des Verwaltungsbaumes) und kleinere Daten gespeichert. Der Großteil der Daten wird in Blatt-Blöcken (Blöcke des Verwaltungsbaues, welche keine Kindknoten haben) oder unformatierten Datenblöcken gespeichert. Unformatierte Datenblöcke gehören nicht zum Verwaltungsbaum. Die letzten Blöcke von Dateien (*Tails*) oder sehr kleine Dateien werden entweder direkt im Verwaltungsbaum, oder, wenn sie für den Verwaltungsbaum zu groß sind, in den unformatierten Blöcken gespeichert. ReiserFS schafft es als einziges Dateisystem, mehrere kleinere Dateien in einen Datenblock zu speichern und nutzt so den zur Verfügung stehenden Platz optimal aus.

Abbildung 4.3: Struktur von internen Verwaltungsknoten [4]

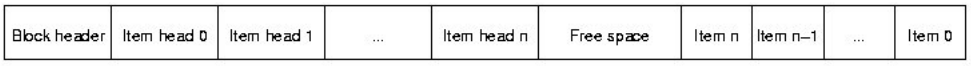


Abbildung 4.4: Struktur von Leaf-Knoten [4]

Interne Verwaltungsknoten bestehen aus einem Blockheader, Schlüssel, Pointer auf Kindknoten und freien Speicher, welcher für Dateifragmente oder kleine Dateien verwendet werden kann. Im Blockheader sind die Anzahl von Items und die Größe des freien Speicherplatzes in diesem Block gespeichert.
In Blattknoten befinden sich ebenfalls ein Blockheader, Itemheaders, Items und freier Speicherplatz, der für zu speichernde Daten verwendet werden kann.

Das Dateisystem beginnt auf dem Datenträger bei dem Offset von 64 kB mit dem *Superblock*, um anderen Anwendungen genug Platz für Bootcode zu lassen. In diesem Block werden wichtige Informationen zur Partition (z.B.: Blockgröße, Blocknummern der Root- und Journalblöcke) gespeichert. Auch das ReiserFS verfügt über ein Journal, auch wenn dieses anfänglich nur dazu verwendet wurde, die Operationen auf die Metadaten (nicht aber auf die Nutzdaten) zu protokollieren. Direkt nach dem Superblock befinden sich ein Bitmap-Block, der die freien Blöcke verwaltet. Die Anzahl der freien Blöcke, die durch einen solchen Bitmap-Block verwaltet werden können, hängt direkt von der Blockgröße ab. Kann ein Bitmap-Block zum Beispiel n Blöcke abbilden, so befindet sich solange an jeder n-ten Blockstelle des Dateisystems (ausgehend vom ersten Bitmap-Block) ein neuer Bitmap Block, bis alle vorhandenen Datenblöcke abgebildet wurden. Ein Einser im Bitmap-Block bedeutet dabei, dass der entsprechende Datenblock belegt ist.

Ergebnis

Das ReiserFS Dateisystem ist zwar blockorientiert aufgebaut, jedoch werden die Dateidaten bytegranular auf den Datenträger geschrieben. Wenn man den selben Testablauf wie bei den vorherigen Dateisystemen verwendet, stellt man fest, dass Dateifragmente, die ganze Blöcke benötigen, auch in ganzen Blöcken gespeichert werden. Nicht benötigte Bytes in einem Block werden auch bei diesem Dateisystem auf Null gesetzt. Kleine Dateien und Dateifragmente werden jedoch, wie weiter oben schon erwähnt wurde, entweder direkt in der Verwaltungsstruktur oder in unformatierten Datenblöcken gespeichert. Freie Plätze werden auch hier auf Null gesetzt. Ähnlich wie bei Ext2/Ext3 bzw. FAT32 wird auch bei ReiserFS beim Löschen von Dateien der eigentliche Inhalt nur vom Datenträger entfernt, wenn er wieder durch andere Dateien überschrieben wird.

ReiserFS

Dateigröße	RAM Slack	Drive Slack
Kleine Dateien	nein	JA
Große Dateien	nein	JA

Kapitel 5

Zusammenfassung

5.1 Vorgehen beim Testen von Dateisystemen auf ihre File Slack Eigenschaften

Um ein Dateisystem auf seine File Slack Eigenschaften testen zu können benötigt man zumindest eine Testpartition, die mit dem entsprechenden Dateisystem formatiert wurde. Diese Partition sollte nicht allzu groß gewählt werden, um die Verarbeitungszeit von benötigten Tools und die Größe von Images zu minimieren. Nun sollte man ein Shell-Script erstellen, welches Dateien verschiedener Größe auf das Test-Dateisystem kopiert und wieder löscht. Dadurch kann man das DAteisystem künstlich altern lassen, da man das Füllen und Leeren des Dateisystemes simuliert. Nun werden Testdateien mit bekannten Inhalten auf die Testpartition kopiert. Anschließend sollte man entweder ein Image der Partition erstellen, oder diese mit einem Tool wie beispielsweise *Autopsy* betrachten. Hat man das Image erstellt, so kann man mit Hilfe eines Hexeditors nach den Inhalten der Testdateien suchen. Praktisch sind dabei Hexeditoren, bei denen man eine Blockgröße und Offset einstellen kann, sodass die einzelnen Blöcke gut erkennbar sind. Wurde der Inhalt einer Testdateien gefunden, so wird mit der Analyse der Daten begonnen. Wurde der restliche Block auf Null gesetzt? Wurde vielleicht nur der Sektor, nicht aber der ganze Cluster auf Null gesetzt? Wie sieht es mit nicht verwendeten Datenblöcken aus? Befinden sich in ihnen noch Daten? Schwierig wird es, wenn festgestellt wird, dass sich tatsächlich *komische* Inhalte nach dem Ende einer Datei auf dem Datenträger befinden. Nun gilt es herauszufinden, woher diese kommen. Ist es wirklich File Slack oder nur das Ende einer anderen Datei? Ist es vielleicht Teil eines Bitmaps oder einer Dateisystemverwaltungsstruktur? Oder befinden sich etwa wirklich Arbeitsspeicherinhalte auf der Festplatte? Es ist von Vorteil, wenn man einen Hexeditor verwendet, der auch RAM anzeigen und durchsuchen kann. So kann nach im Dateisystem gefundenen Mustern auch im Arbeitsspeicher suchen. Jedoch ist es ohne exakte Kenntisse der Funktionsweise und des Sourcecodes des Dateisystems nicht einfach mit Sicherheit zu bestimmen, woher diese seltsamen Daten stammen.

5.2 Resumee - File Slack unter Linux

Vergleich der Dateisysteme

Dateisystem	RAM Slack	Drive Slack
Ext2	nein	JA
Ext3	nein	JA
FAT32 Implementierung von Linux	nein	JA
ReiserFS 3.6	nein	JA

Die durchgeführten Tests zeigen, dass heutige Dateisysteme unter Linux (Kernel 2.6.18) keine Arbeitsspeicherinhalte auf einen Datenträger schreiben. Daten werden jedoch beim Löschvorgang nicht vollständig entfernt. Auch nach einer Formatierung bleibt der Großteil der Dateninhalte auf dem Datenträger gespeichert. Sicherheitstechnisch stellt *File Slack* sicherlich einen Nachteil dar, da unter Umständen Daten auf einen Datenträger gespeichert bleiben, die ein Benutzer vernichten will. Andererseits kann *File Slack* auch von Vorteil sein, wenn es gilt verlorene Daten wiederherzustellen. Inwiefern also *File Slack* ein Nachteil oder ein Vorteil von Dateisystemen ist, bleibt jedem selbst überlassen.

Die in den Literaturquellen angegebenen Weblinks wurdem am 20. Oktober 2006 auf ihre Aktualität überprüft. Alle Tests und Feststellungen dieser Arbeit beziehen sich auf die Version 2.6.18 des Linux Kernels. Mit sehr hoher Wahrscheinlichkeit gilt das Ergebnis dieser Arbeit aber auch für die Nachfolgerversionen der getesteten Dateisysteme, da auf Grund besserer und billigerer Hardware Sicherheitseinschränkungen zu Gunsten der Performanz immer unwichtiger werden.

Literaturverzeichnis

[1] Automated image and restore. http://air-imager.sourceforge.net.

[2] Autopsy. http://www.sleuthkit.org/autopsy/index.php.

[3] The sleuth toolkit. http://www.sleuthkit.org/sleuthkit/index.php.

[4] Florian Buchholz. The structure of the reiser file system, 2006. http://homes.cerias.purdue.edu/ florian/reiser/reiserfs.php.

[5] Scott Courtney. An in-depth look at reiserfs, 2001. http://linuxplanet.com/linuxplanet/tutorials/2926/4/.

[6] Armor Forensics. File slack defined, 2005. http://www.forensics-intl.com/def6.html.

[7] Constantinos A. Loizides. *Analyse und Simulation von Fragmentierungseffekten beim ReiserFS Dateisystem.* ibidem, 2005.

[8] Andrew S. Tanenbaum. *Moderne Betriebssysteme.* Pearson Studium, 2003.

[9] Wikipedia. File slack. http://de.wikipedia.org/wiki/File-Slack.