Data Hiding in Journaling File Systems

By

Knut Eckstein and Marko Jahnke

From the proceedings of

The Digital Forensic Research Conference

DFRWS 2005 USA

New Orleans, LA (Aug 17th - 19th)

DFRWS is dedicated to the sharing of knowledge and ideas about digital forensics research. Ever since it organized the first open workshop devoted to digital forensics in 2001, DFRWS continues to bring academics and practitioners together in an informal environment.

As a non-profit, volunteer organization, DFRWS sponsors technical working groups, annual conferences and challenges to help drive the direction of research and development.

http:/dfrws.org

Data Hiding in Journaling File Systems

Knut Eckstein[†], Marko Jahnke[‡]

Abstract: Data hiding is one technique by which system perpetrators store information while reducing the risk of being detected by system administrators. The first major section of this article structures and compares existing data hiding methods for UNIX file systems in terms of usability and countermeasures. It discusses variant techniques related to advanced file systems. The second section proposes a new technique that stores substantial amounts of data inside journaling file systems in a robust fashion with low detectability, which is demonstrated by means of a proof-of-concept implementation for the ext3 journaling file system.

Index Terms: Data hiding, UNIX, journaling file system

I. INTRODUCTION

Having successfully penetrated a computer system, attackers often want to continue (ab)using the system without being detected by the system owner or administrator. Detection avoidance is accomplished using several techniques, amongst them

- Trojanizing system binaries or kernel, often referred to as introducing "backdoors" or "rootkits"
- Purging of system audit and process accounting logs
- Data hiding

This paper focuses on data hiding which is often used to hide incriminating data such as sniffer or password collector log files or contraband such as pirated copies of programs or media files. Rootkits offer their own data hiding techniques, usually by means of keeping a list of "invisible" files. While their substantial capabilities go beyond hiding files, their usability compared to other hiding techniques is constrained by higher complexity and limited platform availability.

Known hiding techniques for UNIX-based computer systems are discussed in section II. Each technique will be qualified in terms of usability from an attacker's and potential countermeasures from a system administrator's point of view.

For reasons of space, the countermeasures discussion focuses on "live" countermeasures performed on a running system, as opposed to forensic off-line analysis and detection techniques. Also, the attacker's main intent while using the hiding methods described is to hide from casual inspection during normal system operation, not from a forensic analysis[1] applied to a file system image.
Section III introduces and demonstrates a new hiding technique that pertains specifically to journaling file systems. Section IV contains a summary and conclusions.

II. KNOWN HIDING TECHNIQUES

This section is organized into subsections following the high-level abstraction layers for digital data in forensic examinations [2].

- Media management layer
- File system layer
- Application layer

Each layer will be briefly introduced at the beginning of a subsection.

A. Media Management Layer

At the media management layer, digital storage media can be subdivided into multiple parts. These are referred to as partitions, slices or volumes. Subdivision is recorded in partition tables or similar media management data structures.

1) Using unused media areas

The "standard" data hiding technique at this layer is the usage of an area that is marked as not in use according to the partition table. Additionally, small unused areas exist on a storage medium between media management data structures that can also be used for data hiding [4]. For example, in the MS-DOS partitioning scheme used by the Windows OS family and a number of x86 UNIX implementations, the sectors immediately following a boot sector in the 1st track until the start of the first disk partition in the 2nd track offers 62 sectors or 31KB. Also, a media management data structure may not occupy the entire sector length of 512 bytes, thus offering a very small additional hiding area.

Some storage technologies offer specific areas for data hiding outside the generic partition-based scheme discussed here, for example the Host Protected Area (HPA) that is part of the IDE disk drive specification.

a) Usability for the Attacker

To hide data on a standard mass storage medium, an attacker has to reduce the size of one of the partitions on the

[†] NC3A, P.O. Box 174, 2501CD The Hague, The Netherlands, *knut.eckstein@nc3a.nato.int*
[‡] FGAN/FKIE, 53343 Wachtberg, Germany, *jahnke@fgan.de*

[1] Barring steganograhpy and encryption, all hiding techniques discussed here can be discovered by a sector level search of the storage media for key words or file header sequences.

medium, assuming that the current partition layout makes full use of the physical storage space.

In order to be able to reduce the partition size, the file system contained inside the partition first has to be shrunk to the desired new size. This technique requires administrator privileges to operate both the partition and file system related tools as well as access to the raw disk device in order to access the newly created, "unused" space.

Many traditional UNIX file systems do not allow an on-line resize. In this case the attacker has to unmount, shrink the file system and remount in order to create an unused area. This may generates audit log entries which increase the risk of being detected. Advanced file systems often introduce on-line resizing in addition to other features such as journaling, thereby simplifying this hiding method for the attacker to some extent[2]. Regarding HPA-based attacks, HPA size changes may require a system reboot, depending on the disk model [4].

Available "hiding space" is expected to scale linearly with the disk size, assuming that partition or HPA related manipulations below a certain level, for example 5% of total disk size, will not be noticed by system administrators immediately. The method is "reboot safe" in that after a system restart all hidden data remains hidden and no additional measures have to be taken to prevent it from being overwritten or otherwise lost.

b) Countermeasures

Regular checking of partition size and IDE disk/HPA sizes will reliably detect this technique, keeping in mind that local copies of the relevant tools could be trojanized. Given that the attacker typically has obtained system administrator privileges and is thus able to purge local audit logs, auditing the use of disk partitioning tools such as `fdisk` or auditing access to raw disk devices must involve a remote logs server.[3] On highly sensitive systems, employing a single file system which is not online resizable will not entirely prevent the attack, but make it substantially more complicated and easier to detect. In case other legitimate programs do not require raw disk device access, BSD-style SecureLevels [14] could be used to block access to raw disk devices. With SELinux [8] entering mainstream Linux distributions, mandatory access control technology can now be used to strictly limit access to raw disk devices to a small number of legitimate tools, at the cost of additional system management complexity

2) Mounting on non-empty directories

A very simple but yet powerful method to hide information is to mount a filesystem onto an non-empty directory. The data to be hidden is stored in ordinary files or subdirectories in an appropriate directory. The attacker then mounts an existing filesystem – or a newly created one, for example using loopback mounting – onto this directory and therefore hides the data inside the mountpoint directory.

a) Usability for the Attacker

This hiding technique is extremely easy to use, no special tools nor deeper filesystem knowledge are required. Privileges required depend on the access control settings for mounting the target directory. As above, storage capacity scales with the available disk space.

b) Countermeasures

Detecting the use of existing partition mounts has to rely on auditing the (remote) system log for subsequent unmount and mount operations[4]. Newly created mounts can be detected easily by controlling the filesystem mount table at regular intervals using the `df` or `mount` command. This also needs to be done during filesystem backups. Avoiding user-mountable file systems can reliably prevent filesystems from being mounted by non-privileged attackers.

B. File system layer

The filesystem abstraction layer offers a large variety of data hiding techniques. The analysis in this subsection is structured along four categories[5] of file system data structures. These categories have been introduced [2] to analyse file systems following a generic model:

- File system
- Data unit / Content
- Metadata
- File name / Human Interface

1) File system category

Data structures in the file system category are partition level boot blocks, "superblocks" or other global data structures that describe the file system's general properties.

These data structures may not use an entire logical disk block. Similar to the previous section, this may lead to a number of very small data hiding opportunities [4].

2) Data unit category: Slack space

The data unit category refers to a file system organizing storage media sectors into individually adressable data units, often referred to as "logical disk blocks."

Slack space is defined as the unused part of a file's last data unit. For example, a file which is 10KB in size will require three 4KB data units for storage in a file system with 4KB block size. The last unit will only be used up to its first 2KB. The other two 2KB are unused and available to an attacker for hiding data in this file, which can be seen as a "host file."

[2] Among those file systems that do allow online resizing, some only allow growing but not shrinking

[3] In this case the attacker has to decide whether the "benefit" of (partially) disabling remote logging outweighs the risk of revealing his existence through this change in the system's audit log configuration.

[4] Often times, mount and unmount operations are not logged by default.

[5] These categories are not layers of abstraction, thus the term "sub layer" is avoided in this context.

a) Usability for the Attacker

The slack space can be accessed through publicly available tools, for example `bmap` [11]. This hiding technique requires administrator privileges to access the raw disk device.

On the downside, this method offers limited storage space, on average bounded by 1/2 * block size * number of files. Also, it cannot guarantee the hidden data against being overwritten, in case the "host file" is deleted or changes its size. The attacker could limit storage to files he expects to be fairly static. While this would reduce the chances of losing hidden data, it still does not provide any guarantees and it further limits the available storage.

b) Countermeasures

Auditing of raw disk device access is one potential countermeasure, which can be resource intensive if legitimate tools do access raw disk devices on a regular basis. In a highly sensitive environment, access to raw disk devices could be either be blocked altogether via the BSD secure level mechanism or regulated by the mandatory access control mechanisms offered by SELinux.

Attacker tools like `bmap` can also be used to audit or wipe the available slack space. Alternatively, a small 'slack space wiper' software could be created and run periodically. Such a program must be carefully crafted to avoid manipulating files which the OS kernel is writing to at the same time.

Advanced file systems often introduce extent-based addressing schemes. These schemes no longer require addressing of individual data units in order to manage the data units allocated to a file. Instead, address and length of potentially very large data unit areas on disk are recorded. This means that smaller data unit sizes can be specified in order to reduce available slack space without adversely effecting data unit allocation management overhead.

3) Metadata category: Use reserved inodes

In the metadata category, data structures commonly referred to in UNIX file systems as "inodes" organize per-file metadata such as timestamps, ownership, access rights etc.

An attacker may use inodes which the operating system itself will not use, assuming that those inodes will also be ignored by file system checks, forensic and auditing tools. Best known examples are inodes numbers zero and one in the Berkeley Fast File System (FFS) and inode number one [12] in Linux ext2/3 file systems[6].

Recent analysis [5] has shown that advanced UNIX file systems can offer more opportunities in this regard. Particularly in cases where "multiple file systems per partition" functionality is present, which is mandated by the Distributed File System (DFS) specification [5]. Multiple file systems per partition are typically implemented through a meta file system, for example the "aggregate fileset" in JFS [1] and the "structural file system" in the Veritas file system [7]. Both the meta file system and additional file systems offer additional inodes for this data hiding scheme.

a) Usability for the Attacker

An attacker who wants to employ this hiding scheme has to program a nontrivial piece of software, which is going to create files using reserved inodes**.** Such software will require system administrator privileges to access the raw disk device. The number of files that can be created this way is limited depending on the specifics each file system, for example 2 for FFS and 13 for JFS for Linux [1]. The available storage space scales linearly, assuming – as above – that a certain percentage of discrepancy between `du` and `df` output would not be noticed.

b) Countermeasures

Comparing the output of `du` and `df` totals per file system can be one way of detecting this data hiding technique. Auditing or limiting access to raw disk devices as discussed above provides another countermeasure. The file system check software provided by the operating system may or may not complain about using reserved inodes. Forensics tools have been updated to check for reserved inode numbers, for example the `ils` command from "The Sleuthkit" (TSK) [4] or "The Coroner's Toolkit" (TCT) [7] can be run against a FFS or ext2/3 file system.

4) Metadata category: Extended file attributes

In recent years, UNIX file systems like FreeBSD's UFS2, UFS in Solaris 9 etc. have introduced additional file attributes for data storage, comparable to MacOS resource forks, NTFS alternate data streams or OS/2 extended attributes. These capabilities vary in maximum storage space and implementation. All offer a hiding opportunity for attacker in that they are ignored by traditional UNIX file integrity checkers, not listed during normal directory list or search operations and sometimes not even known to the system administrator. They are easy to use for the attacker and easy to detect for the system administrator using commands provided by the operating system. Hiding space scales linearly with available disk space.

5) File name category: "Special" filenames

In the file name category, a file system stores and processes data to assign human-recognizable names to files and directories, usually creating a hierarchical "directory tree."

Human perception can be deceived in the file name category through creating filenames consisting of space characters or spaces and dots, for example ".. " is a popular combination. Another technique creates "credible" file names in densely populated areas of the directory tree, like `/dev` or `/usr/man`, where the casual or untrained inspector would fail to recognize that the file was not a legitimate operating system component.

[6] Ext2/3 actually reserves inodes 1-11 without using all of them, but the tool introduced in [12] appears to only use inode one.

Advanced schemes use special characters like Carriage-Return inside a filename. These filenames are not meant to fool the human perception, but specifically designed to thwart standard auditing techniques by making commands like `find` or `xargs` employed by host based file checking routines skip the file to be hidden.

a) Usability for the Attacker

This hiding technique is very easy to use, no special tools are required. Privileges required depend on the access control settings of the target directory. Storage capacity scales linearly with available disk space.

b) Countermeasures

Detection can be performed easily by searching for filenames containing special characters. Alternatively, a host-based intrusion detection tool could monitor for file creation.

6) File name category: Removal of open files

For a long time, attackers have been known to hide both program and data files by removing them while the program in question was still running. Upon deletion of an open file the operating system immediately removes the filename from the directory but delays removal of metadata and content until all referring file descriptors have been closed. Thus the file becomes "invisible" for filename-level inspection.

a) Usability for the Attacker

No administrator level privileges are required for this technique. It is thus very easy to use but storage is highly ephemeral: Accessing these hidden files after program termination or from another program is difficult and requires forensic tools. Another drawback is that not all implementations of UNIX allow the removal of open files. HP-UX, for example, allows removal of open data files but not program files. Storage space scales linearly with disk size, again assuming that `df`/`du` discrepancies below a certain percentage will not be noted immediately.

b) Countermeasures

The `lsof` (LiSt Open Files) tool, which is available for most UNIX platforms, can be used to detect this hiding technique. Its output clearly marks deleted but still open files. On a trojanized system, `ils` can be used on supported file systems to provide a independent view without being influenced by tool-level or kernel-level distortions.

7) Metadata/File name category: Hide in deleted inodes plus trojan fsck

This is a complicated schema which has been observed and reported on a few occasions [12]. It is based on the method introduced in the previous subsection but addresses its ephemerality in an intriguing fashion.

The fundamental idea is to use a trojanized version of the file system checking program (`fsck`). At boot time the trojanized `fsck`/`rc` script combo "undeletes" and spawns the malicious software, for example a sniffer, then "removes" the sniffer executable and the sniffer log file again.

a) Usability for the Attacker

This is a complex procedure where the attacker has to make sure that the trojanized `fsck` works as predicted under all circumstances. Storage space offered by this method is identical to the previous method, since the hiding of the malicious data files is performed in exactly the same fashion.

b) Countermeasures

In one account, this method was discovered by the incident handler eventually noticing that the system involved would invoke `fsck` during each and every boot process. which then led to a closer investigation of the boot process. In the other account [12], an off-line analysis where the system was booted from a trusted CD, revealed the trojan `fsck` "unhidden" while the clean `fsck` was hidden[7] in an inode marked as deleted. For detecting the running process that uses the malicious executable and logfile, `lsof` or `ils` could be employed as in the previous section.

C. Application Layer

Hiding data at the application layer works with individual files ("host files") and is thus independent of both the operating system and the file system in use. It is briefly addressed in this section for completeness.

1) Obfuscated Loopback Filesystems

Loopback filesystems provide a widely used mechanism offered by a number of Unix type operating systems to create and mount a filesystem image within an arbitrary host file. Regularily created filesystem images can easily be identified by the `file` command due to their unique "magic number." A simple, but effective method to obfuscate the real purpose of the image file is to use the `offset` option of the loopback mount command: Just by concatenating a filesystem image to a (partial) file with a registred magic number (e.g. the beginning *n* bytes of a dynamic loadable library), the files real purpose cannot be determined reliably, while it is still possible to mount the image by using the apropriate offset flag of the loopback mount command (e.g. "`-o loopback, offset=`*n*").

2) Unused spaces in application file formats

Many file formats contain unused sections, for example the comment field in a jpeg image format. Space is quite limited and largely depends on the number of available application files. Administrator privileges are usually required. Creating lots of new "host files" will not go unnoticed. Still auditing can be cumbersome since detection is specific to every possible file format used. Alternatively, a timeline analysis of file modification and access times or

[7] In [12], additional measures are taken to hide the trojanized fsck program itself, but the persistence of this additional measures across multiple reboots remains unclear.

the logfile analysis of a host-based intrusion detection system (HIDS) monitoring file operations could reveal anomalies hinting at a "secondary use" of the application files in question. For example, a hiding tool which would simply split up a large piece of information and hide it in a large number of files, would create a large number of almost identical access or modification timestamps during read or write operations.

3) Steganography

Although steganography is very much a research field on its own, from the file system centric or "lower level" point of view of this analysis it is very similar to the previous section in that space is being offered largely depends on the number and size of "host files." Likewise, countermeasures are either very specific in terms of content/application specific steganalysis or could be based on anomaly analysis of file access timestamp patterns. A steganographic file system is described in [7].

D. Summary of known hiding techniques

Barring steganography, all hiding methods described up to this point either were designed to escape a casual human observer only, or they were designed to use unused areas, "holes" in data structures or relying on "nominal" deletions not being overwritten due to limited file system activity.

III. NEW SCHEME: DELIBERATE FS INCONSISTENCIES

The new data hiding scheme that is proposed in this paper works by introducing deliberate file system inconsistencies. It relies on a fundamental property of journaling file systems. Because the journal records all recent file system modifications, the time required for the file system consistency check at boot time can be drastically reduced: Instead of checking consistency amongst all categories of file system data, the modifications recorded in the journal are "replayed" to check whether all of them were executed successfully. If, due to a system crash etc. not all recorded modifications were performed, the journal allows for a very efficient "roll back" to a clean file system state.

That means that the file system consistency check during system start up no longer scales linearly with disk size but is reduced to a short check of the journal contents. Thereby system restart times are drastically reduced, particularly for data center systems with very large storage subsystems. Only in very exceptional circumstances of massive corruption, where rolling back the journal does not result in a consistent file system, will a full, traditional consistency check be performed across the whole file system.

An attacker can exploit this lack of consistency checking to hide large amounts of data by deliberately introducing inconsistencies between categories while keeping consistency in each individual category to avoid loss of hidden data due to overwriting. By directly modifying data structures on disk, the attacker bypasses the journaling mechanism altogether. The following paragraph illustrates one of several possible inconsistencies: The attacker allocates and uses data units. He does not create inodes to reference the data units, thereby deliberately introducing an inconsistency between the data unit and the metadata category.

In the first step he localizes a set of unallocated data units. These units are then marked as allocated and used for the purpose of hidden storage. Because of the allocation marking, the operating system cannot accidentally overwrite the hidden data. In this simple scheme the attacker has to record the addresses of the data units used for later reference. While this may sound like an onerous task, many journaling file systems offer extent-based allocation schemes where large areas of disk storage can be addressed contiguously. Thereby only a single address and length specification would in many cases be all that needs to be recorded by the attacker.

A. Proof of concept demonstration

In order to test the usability of this attack, a simple proof-of-concept code was implemented for the ext3 file system which is a popular default choice for many Linux installations. The listings below were created in a VMware virtual machine running the Suse 9.2 Linux distribution[8].

```
# dd if=/dev/zero of=/dev/sda1 bs=1k
dd: writing `/dev/sda1': No space left on device
17393+0 records in
17392+0 records out
# mkfs.ext3 /dev/sda1
[...]
Block size=1024 (log=0)
Fragment size=1024 (log=0)
4368 inodes, 17392 blocks
3 block groups
8192 blocks per group, 8192 fragments per group
1456 inodes per group
[...]
Creating journal (1024 blocks): done
[...]
```

Listing 1: Creation of sample ext3 file system

Listing 1 shows the creation of the test file system on a 16MB virtual disk (`/dev/sda1`). Note that the file system journal takes up 1024 logical disk blocks, whose size is 1K.

```
# mount /dev/sda1 /mnt
# df -h /mnt
Filesystem            Size  Used Avail Use% Mounted on
/dev/sda1              17M  1.1M   15M   7% /mnt
# du -sh /mnt
13K     /mnt
# perl -e "print 'A' x 4 x 1024 x 1024" >/mnt/4MB-of-As
# ll /mnt
total 4134
drwxr-xr-x   3 root root    1024 Sep 19 02:00 .
drwxr-xr-x  24 root root    4096 Sep 13 08:07 ..
-rw-r--r--   1 root root 4194304 Sep 19 02:00 4MB-of-As
drwx------   2 root root   12288 Sep 19 01:59 lost+found
# df -h /mnt
Filesystem            Size  Used Avail Use% Mounted on
/dev/sda1              17M  5.1M   11M  33% /mnt
# du -sh /mnt
4.1M    /mnt
```

Listing 2: Initial file system usage

[8] A small, secondary harddrive was chosen for simplification and readablility of listings, nonetheless this hiding method was also successfully tested on a 20GB ext3 system partition.

The attacker begins his work in listing 3, analyzing the file system for areas of low usage. For this purpose, he uses the `fsstat` tool from TSK.

```
# fsstat -f linux-ext3 /dev/sda1
FILE SYSTEM INFORMATION
--------------------------------------------
File System Type: EXT3FS
[…]
Number of Block Groups: 3
Inodes per group: 1456
Blocks per group: 8192
[…]
Group: 0:
  Inode Range: 1 - 1456
  Block Range: 1 - 8192
[…]
  Free Inodes: 1444 (99%)
  Free Blocks: 3379 (41%)
  Total Directories: 2
Group: 1:
  Inode Range: 1457 - 2912
  Block Range: 8193 - 16384
[…]
  Free Inodes: 1456 (100%)
  Free Blocks: 7477 (91%)
  Total Directories: 0
Group: 2:
  Inode Range: 2913 - 4368
  Block Range: 16385 - 17391
[…]
  Free Inodes: 1456 (100%)
  Free Blocks: 823 (10%)
  Total Directories: 0
```

Listing 3: File system reconnaissance

In correspondence with the diagnostic output during file system generation in listing 1, `fsstat` reports three block groups in total. The analysis by group shows that the 4MB file created in listing 2 has been stored in group zero, leaving group one mostly empty. The last block group, number two, is less attractive for hiding data since it is very small, just a fraction of the regular group size.

```
# dstat -f linux-ext3 /dev/sda1 9400
Block: 9400
Not Allocated
Group: 1
# dls -e -f linux-ext3 /dev/sda1 9400-16384 | md5sum
2c2092ed51b9d00f787d3a41cf6c564b  -
# ../bin/dwrt -f linux-ext3 /dev/sda1 9400-16384
Block: 9400
Group: 1
[...]
Block: 16383
Group: 1
Block: 16384
Group: 1
# dls -e -f linux-ext3 /dev/sda1 9400-16384 | md5sum
43a851c93c8d0a768e88e337b8b485fa  -
# df -h /mnt
Filesystem            Size  Used Avail Use% Mounted on
/dev/sda1              17M  5.1M   11M  33% /mnt
# du -sh /mnt
4.1M    /mnt
# dls -e -f linux-ext3 /dev/sda1 11111 | xxd -c 4
0000000: 4945 4545  IEEE
0000004: 4945 4545  IEEE
0000008: 4945 4545  IEEE
000000c: 4945 4545  IEEE
[...]
```

Listing 4: Data hiding in progress

In listing 4, the attacker probes the start of the area of unallocated blocks within group one. According to `fsstat`, the block range of group one is 8193 – 16384. Since block and inode allocation bitmaps etc. take up some space at the beginning of each block group, his first guess is block 9400. Using `dstat` from TSK he can confirm that this block is indeed free.

Given the overall free ratio of group one, the attacker can now safely assume that all further blocks in this group up to and including block 16384 are not allocated. For diagnostic purposes, the md5 checksum over these blocks is created using `dls`. Then the proof-of-concept tool `dwrt` is used to fill the logical disk block sequence 9400-16384 with the repeated ASCII sequence 'IEEE'. Should `dwrt` encounter an allocated block within the specified sequence, an error condition would have been raised. The truncated dwrt screen output confirms successful hiding of 6MB of data.

Since the original content of these blocks was a sequence of zero bytes (viz zeroizing in listing 1), the repeated calculation of the md5 checksum gives a different result.

Most notably, `df` and `du` now report exactly the same disk usage figures as in listing 2. That means there is no straightforward way for the system administrator to notice the hidden data. The final command in listing 4 executes a hexdump of block 11111, arbitrarily chosen from the hiding sequence, to confirm that the expected ASCII strings can indeed be found on disk.

One sitution where the hidden data becomes apparent is when the disk fills up entirely, as shown in listing 5.

```
# dd if=/dev/zero of=/mnt/fill-up bs=1k
dd: writing `/mnt/fill-up': No space left on device
4675+0 records in
4674+0 records out
# df -h /mnt
Filesystem            Size  Used Avail Use% Mounted on
/dev/sda1              17M  9.7M  6.0M  62% /mnt
# du -sh /mnt
8.7M    /mnt
```

Listing 5: File system filling up

Instead of 11MB as expected from the output of `df`, only about 5MB can be written to the file "fill-up". According to `df`, the disk still has 6MB free, yet the file system is completely filled up, as reported by the I/O error condition in listing 5.

To show that this hiding technique is system-crash-proof, the demonstration now continues with power cycling the virtual machine.

```
# fsck.ext3 /dev/sda1
e2fsck 1.34 (25-Jul-2003)
/dev/sda1: recovering journal
/dev/sda1: clean, 13/4368 files, 10591/17392 blocks
# dls -e -f linuxt-ext3 /dev/sda1 9400-16384 | md5sum
43a851c93c8d0a768e88e337b8b485fa  -
```

Listing 6: File system check after power cycle

Listing 6 shows the test file system being checked. As predicted, `fsck.ext3` replays the journal and is able to recover from the inconsistencies introduced through the power cycle. Thus no full consistency check is performed and the hidden data remains undetected. The md5

checksum computed at the end of listing 6 confirms that the hidden data is still available in its entirety.

To discover the inconsistency introduced by the attacker to hide his data, `fsck.ext3` needs to run with the `-f` flag to indicate a full, time consuming consistency check. This check is equivalent to the consistency check of a regular, non-journaling file system. In listing 7, the output of this command clearly shows the type and area of inconsistency as introduced by the attacker. As soon as the inconsistency is removed, the 7MB are available again to the regular users of the file system. The hidden data remains on disk until is finally overwritten during the creation of the file named fill-2.

```
# fsck.ext3 -f /dev/sda1
e2fsck 1.34 (25-Jul-2003)
Pass 1: Checking inodes, blocks, and sizes
Pass 2: Checking directory structure
Pass 3: Checking directory connectivity
Pass 4: Checking reference counts
Pass 5: Checking group summary information
Block bitmap differences:  -(9400--16384)
Fix<y>? yes

/dev/sda1: ***** FILE SYSTEM WAS MODIFIED *****
/dev/sda1: 13/4368 files (15.4% non-contiguous),
10407/17392 blocks
# mount /dev/sda1 /mnt
# dstat -f linux-ext3 /dev/sda1 9400
Block: 9400
Not Allocated
Group: 1
# dls -e -f linux-ext3 /dev/sda1 9400-16384 | md5sum
43a851c93c8d0a768e88e337b8b485fa  -
# df -h /mnt
Filesystem            Size  Used Avail Use% Mounted on
/dev/sda1              17M  9.7M  6.0M  62% /mnt
# du -sh /mnt
8.7M    /mnt
# dd if=/dev/zero of=/mnt/fill-2 bs=1k
dd: writing `/mnt/fill-2': No space left on device
6957+0 records in
6956+0 records out
# df -h /mnt
Filesystem            Size  Used Avail Use% Mounted on
/dev/sda1              17M   17M     0 100% /mnt
# du -sh /mnt
16M     /mnt
# dls -e -f linux-ext3 /dev/sda1 9400-16384 | md5sum
79bff29eb565da721cf5c30a880b2379  -
```

Listing 7: Full file system consistency check

Since the attacker is directly manipulating file system data structures on disk, he should carefully choose areas with little or no file system activity. Otherwise the file system might realize that the metadata changes performed directly on disk conflict with the same metadata buffered by the file system driver. Listing 8 shows an error message captured during testing, which hints at such a conflict where both the file system and the attack tool write-accessed block 16002.

B. Usability for the Attacker

This attack requires in-depth knowledge about the layout of the target file system. It provides the attacker with a long-lived, "crash-proof" hiding scheme while avoiding the risk of accidental overwrites.

```
block=16002, b_blocknr=16000
b_state=0x00000019, b_size=1024
buffer layer error at fs/buffer.c:502
Call Trace:
 [<c0156bef>] __find_get_block_slow+0x8f/0x170
 [<c0156f28>] __find_get_block+0xb8/0x170
 [<c4af8947>] ext3_new_block+0x227/0x520 [ext3]
 [<c4afc17d>] ext3_alloc_block+0xd/0x20 [ext3]
 [<c4afc54d>] ext3_get_block_handle+0x22d/0x920 [ext3]
 [<c4ac92ec>] do_get_write_access+0x33c/0x520 [jbd]
 [<c4afb5a5>] ext3_mark_iloc_dirty+0x1b5/0x3b0 [ext3]
 [<c4afcc7e>] ext3_get_block+0x3e/0x80 [ext3]
 [<c0158f7f>] __block_prepare_write+0x1ef/0x470
 [<c0159216>] block_prepare_write+0x16/0x30
 [<c4afcc40>] ext3_get_block+0x0/0x80 [ext3]
 [<c4afd6ee>] ext3_prepare_write+0x3e/0xd0 [ext3]
 [<c4afcc40>] ext3_get_block+0x0/0x80 [ext3]
[…]
```

Listing 8: File system driver error message

C. Countermeasures

Unless access to raw disk devices is being audited or limited, the chances of this scheme being detected are very low: Occasionally, messages like in listing 8 might occur, depending on how successful the attacker is in choosing "quiet" regions on disk But those messages do not easily lead to suspecting an attack.

Another way of detection or potential loss of hidden data would be a system crash with substantial disk corruption, resulting in a full consistency check. Such a check would immediately notice that the data units occupied by the attacker are not being referenced by any inodes (see listing 7 in the previous section). Normal system operation would resume and the attacker would have a chance of "recovering" his hidden information which is still there but now at risk of being overwritten (a very defiant attacker would simply proceed to put his data unit allocation markings back in place).

To check consistency manually on a running system, an administrator could perform read-only file-system checks (`fsck -n …`) on mounted file systems. These read-only checks typically skip the journal-replay phase and perform a full file system analysis. The difficulty with this check is to differenciate between inconsistencies introduced by the attacker and "natural" inconsistencies occuring due to journaled file system changes not yet being committed to disk.

Another means of detection could be, as with other methods, the administrator comparing the output of the disk usage and disk free commands. As the proof-of-concept exploit demonstrates below, the attacker may choose not to update the summary information from which `df` appears to compute its reports. That means while `du` only summarizes usage of regular, visible files, `df` would in a similar fashion rely only on the adjustments that regular file creation or deletion operations would perform on the summary information. Therefore, unless the file system performs additional "house-keeping" updates of its summary information tables, `df` and `du` output would

appear to be perfectly in accordance[9]. Only if the attacker would boldly occupy so much space on disk that a regular file creation operation would result in an "out of disk space" error condition, the discrepancy between the free disk space reported by `df` and the disk being totally filled up would be recognizable.

D. *Variants of the new hiding scheme*

Instead of just occupying data blocks, an attack variant could include the use of inodes in a similar fashion. This would further decrease detectability, as a file system check would only report a single inode being marked as used but no directory entry pointing to it, instead of hundreds or more disk blocks being marked as occupied but no inode pointers referring to them.

The technique introduced in this section could also be applied to traditional, non-journaling file systems. In that case the attacker would risk detection whenever a file system check was run. File system checks carried out in regular intervals could be avoided by manipulating the mount count and last-checked timestamp, but any crash or malfunction leaving the file system in an unclean state would lead to detection. Additionally, the attacker could employ a trojanized version of the file system integrity checker, similar to section II.B.7).

IV. SUMMARY AND CONCLUSIONS

In contrast to standard hiding methods which are either complex to use, easy to detect, limited in storage capacity or offer a rather volatile storage capacity, the new scheme avoids most disadvantages. In addition, it is not only "reboot-proof" but even "system crash-proof."

Fundamentally, it exploits the fact that journaling file systems have – in almost all cases – replaced a full-scale consistency check by a much faster journal recovery, which only audits recent modifications recorded in the journal. This allows for insertion of inconsistencies between file system data structures of different categories. These inconsistencies are designed to hide information and to preserve it from being overwritten.

System administrators of sensitive systems should be aware of the security implications of file system technology choices and perform detective measures accordingly.

Forensic analysis tools should include specialized file system consistency checkers, which ideally perform more rigorous checks than the default checking software provided by the respective operating system.

[9] The amount of housekeeping is highly file system dependent. The tests run with ext3 so far would suggest that very little housekeeping is being done by this particular file system. On file systems which were known to perform a lot of housekeeping, the attacker could actually choose to duly update summary information. Thereby he would trade the risk of "housekeeping hickups" showing in the system logs with the risk of `df`/`du` discrepancies being noticed.

V. ACKNOWLEDGEMENTS

The authors would like to thank Brian Carrier, John Collura, Ian Davies, and Andreas Thümmel for reviewing the manuscript and providing insightful comments.

VI. REFERENCES

[1] Best, S., Kleikamp, D., "JFS Layout", *IBM developerWorks*, May 2000, Available at: www.ibm.com/developerworks/opensource/jfs, viewed Jan 2005

[2] Carrier, B. "An Investigator's Guide to File System Internals", *FIRST Conference on Computer Security, Incident Handling & Response*, June 2002, Available at: www.first.org/events/progconf/2002/d1-02-carrier-slides.pdf, viewed January 2005

[3] Carrier, B. "Defining Digital Forensic Examination and Analysis Tools using Abstraction Layers", *International Journal of Digital Evidence*, Volume 1, Issue 4, Winter 2003.

[4] Carrier, B. "File System Forensic Analysis", *Addison-Wesley*, 2005.

[5] Chutani, S. et al, "The Episode File System", *Proceedings of the USENIX Winter 1992 Technical Conference*, p 43-60, Available at: citeseer.ist.psu.edu/chutani92episode.html, viewed March 2005

[6] Eckstein, K. "Forensics for Advanced UNIX File Systems", *Proceedings of the 5th IEEE Information Assurance Workshop*, 2004

[7] Farmer, D. and Venema, W. "Forensic Discovery, *Addison-Wesley*, 2004

[8] Locosmo, P. and Smalley, S. "Integrating Flexible Support for Security Policies into the Linux Operating System", *Proceedings of the 2001 USENIX Annual Technical Conference*, June 2001

[9] McDonald, A. and Kuhn, M. "StegFS: A Steganographic File System for Linux", *IH'99, LNCS 1768*, pp. 463-477, Springer Verlag, 2000, available at www.cl.cam.ac.uk/~mgk25/ih99-stegfs.pdf

[10] Pate, S. D. "UNIX Files Systems, Evolution, Design and Implementation", *Wiley Publishing*, 2003

[11] Pelcher, B. "Hidden Data is Evidence too", *SANS GIAC GCFA practical*, 2004, Available at www.giac.org/practical/GCFA/Bob_Pelcher_GCFA.pdf, viewed March 2005

[12] Ruiu, D. "Active Forensics: Tracking that Intruder", Jan 2001, available at staff.washington.edu/dittrich/misc/active-forensics.txt, viewed March 2005

[13] The Grugq, "Defeating Forensic Analysis on Unix", *Phrack Magazine*, Volume 10, Issue 59, July 2002, Available at www.phrack.org, viewed March 2005

[14] The FreeBSD Documentation Project "The FreeBSD Handbook", available at http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/, , viewed March 2005