Chapter 3

FILE SYSTEM SUPPORT FOR DIGITAL EVIDENCE BAGS

Golden Richard III and Vassil Roussev

Abstract Digital Evidence Bags (DEBs) are a mechanism for bundling digital evidence, associated metadata and audit logs into a single structure. DEB-compliant applications can update a DEB's audit log as evidence is introduced into the bag and as data in the bag is processed. This paper investigates native file system support for DEBs, which has a number of benefits over ad hoc modification of digital evidence bags. The paper also describes an API for DEB-enabled applications and methods for providing DEB access to legacy applications through a DEB-aware file system. The paper addresses an urgent need for digital-forensics-aware operating system components that can enhance the consistency, security and performance of investigations.

Keywords: Operating system internals, file systems, digital evidence bags

1. Introduction

Digital forensic tools typically utilize standard operating system components, e.g., file systems and caching mechanisms. However, there are compelling performance, consistency and security reasons for making operating system components "digital forensics aware." These include performance (e.g., better data distribution and clustering mechanisms, particularly for distributed digital forensics [2]), security (e.g., protection of digital evidence from unauthorized access or tampering), and consistency. This paper considers the advantages and design challenges of digital-forensics-aware file systems. Specifically, it examines how auditing of digital evidence is currently handled and how an enhanced file system can make this process more automated and more accurate.

Evidence bags and seals are a standard item in traditional crime scene investigations. Bags and seals allow evidence to be preserved and catego-

rized, and tamper-evident designs indicate if the evidence is still secure. Many types of evidence bags provide ample writing space so that notes can be written directly on the bag. Furthermore, the bag's seal may include information such as the name of the investigating officer, case identifiers, the suspect's name, a description of the item, and the date and time when the bag was sealed. Continuity sections on the bag permit tracking the movements of the bag and noting its chain of custody.

Traditional digital forensic methods capture, preserve and analyze evidence in standard electronic containers: images of seized hard drives (e.g., created using the Unix `dd` command) are stored in regular files and documents are typically processed "as is." Auditing a digital investigation, from identification and seizure of evidence through duplication and analysis is essentially ad hoc, with much of the information recorded in separate log files or in the investigator's case notebook.

For example, the popular `dd` utility provides no direct method for capturing information about when the imaging operation took place, who performed the operation, and the results of integrity checks. This information must be recorded separately and, in the case of an integrity check, additional commands (e.g., `md5sum` or a similar cryptographic hashing command) must be executed and the output recorded manually. While enhanced versions of `dd` exist, e.g., `dcfldd`, adding integrity checks to each application is tedious and error-prone. In addition, integrity problems are aggravated when large chunks of digital evidence must be split into pieces, for example, when a disk image is fragmented to fit on removable storage, and then reassembled for processing.

Ad hoc auditing is bound to be incomplete. Because different tools provide widely disparate amounts of auditing information, much of the information must be recorded manually by an investigator. Over the course of an investigation, a piece of digital evidence may be touched by many different tools, some of which generate no audit trail (e.g., `dd` and Sleuth Kit command line tools [5, 4]) and some that generate their own audit logs (e.g., FTK [1]). Ultimately, an investigator is left to piece together these bits of audit trail information to create a comprehensive view of what occurred during the investigation. The failure to record certain information, e.g., the MD5 hash generated by `md5sum` for a large disk image, could result in a huge amount of lost time if the operation must be repeated.

Digital Evidence Bags (DEBs) [8] are universal containers for digital evidence, much as traditional evidence bags are containers for other types of forensic evidence. DEBs bundle digital evidence, associated metadata and audit logs into a single structure, providing an audit trail of operations performed on the evidence as well as integrity checks. In

addition to providing increased security for digital evidence, the audit log details the processes applied to the evidence throughout an investigation. This is potentially useful from the educational and evaluation standpoints, allowing novice investigators to see what steps were taken, which tools were used, and the order in which they were used. DEB-compliant applications can update a DEB's audit log as evidence is introduced into the bag and as data in the bag is processed.

This paper suggests that while the adoption of a standard format for storing digital evidence could radically improve the investigative process, system support for DEBs would be even more useful. The paper investigates native file system support for DEBs in which the basic file type is a DEB. This approach provides several benefits over ad hoc DEB implementations. Some of the advantages of DEBs can be realized even for current DEB-unaware tools: a DEB-enabled file system can transparently offer a DEB's contents to such tools while automatically updating the DEB's metadata and audit log. Another advantage, even for DEB-enabled tools, is that the code for accessing a DEB, including introducing and removing items from the bag and updating the audit log and metadata, needs to be certified only once. Finally, a standard API for accessing DEBs greatly reduces the effort in involved adding DEB support to current and future applications.

2. Basic DEB Structure

The notion of a Digital Evidence Bag (DEB) was proposed by Turner [8]. This section discusses the DEB structure at a high level, focusing instead on the DEB components necessary for native file system support. Interested readers are referred to [8] for additional details about DEBs.

A DEB consists of a collection of objects. The first is the tag area, which is a set of name/value pairs containing metadata associated with the DEB. The information includes a unique identifier for the DEB, the creation date and time, the name and organization of the DEB creator, and a list of Evidence Units (EUs) stored in the DEB. Each EU provides a name for a distinct blob of digital evidence stored in the bag and the blob's associated index file. An index file describes one blob of digital evidence, e.g., detailing the files contained in the blob or the physical characteristics and model/serial number of an imaged disk device. Finally, an audit log (called a Tag Continuity Block in [8]) tracks the operations performed against a DEB, including the date, time, affected blocks, application signature of each operation as well as periodic hashes of DEB contents. Section 3.4 discusses how secure auditing techniques can be used to protect DEB contents from tampering.

3. Design Overview

Native file system support for DEBs must:

- Allow transparent import of DEBs into an enabled file system, i.e., support the automatic conversion of DEBs to a native storage format.
- Allow transparent export of DEBs from an enabled file system to a "normal" file system, i.e., support the automatic conversion of DEBs to a popular container format, e.g., XML.
- Provide convenient, efficient and secure access to DEBs for new DEB-enabled applications as well as legacy applications.

The next section surveys the main design choices for meeting these goals. The subsequent sections describe an API for DEB-enabled applications, discuss how native applications can transparently access DEBs, and evaluate methods for securing DEB audit logs.

3.1 Design Choices

A format such as XML is a sensible choice for DEB files when they are stored outside a DEB-enabled file system. The original DEB specification [8] used plain text for the components of a DEB. However, a more efficient format is required for the native storage of DEBs. This is because unlike other compound file types, e.g., ZIP files or tarballs, DEBs are updated quite frequently as evidence units are introduced and the audit logs are modified. It is also likely that some DEBs will be extremely large, so methods for in-place updates will be necessary for efficient access. In summary, a native storage format for a DEB-enabled file system should support efficient updates of the audit log as well as rapid access to the digital evidence blobs.

Several file systems were evaluated before choosing a candidate for native DEB support, including ext2/3, reiserFS and XFS. NTFS was eliminated because its source code is not available, although NTFS alternate data streams are an attractive mechanism for implementing DEB resource forks (e.g., blobs of digital evidence, the audit log and DEB metadata). Most of the evaluated file systems contain features, e.g., extended attributes (EAs), that can be used to efficiently support DEBs. Unfortunately, the EA implementation in ext2/3 places substantial limits on the size of EAs (one disk block), precluding their use to store larger components in a DEB. Similarly, XFS and stable versions of reiserFS limit the size of EAs. In a future version of reiserFS, EAs will be stored as regular files in the file system, with the file name referring to

the name of the EA and the file contents being the associated value of the extended attribute. We adopt a similar strategy using symbolic links to store DEB components.

The best choice may be to use a standard file system like ext3. The resource forks within a DEB can be stored as separate files using symbolic links stored in the first data block of a DEB file. Changes are also required at the inode level (to tag DEBs as a special type of file and to accommodate efficient storage of DEBs), to pathname handling (to allow transparent access to digital evidence blobs within a DEB using a convention like `DEBname.blobname`), and at the system call level (to support a DEB API and legacy applications). At the system call level, standard file I/O calls such as `read()` and `write()` must be modified to perform auditing functions in addition to accessing blocks of a blob stored within a DEB.

We are currently using a user-level file system, FUSE (File System in User Space) [7], to test our ideas. System calls in FUSE are redirected by a kernel-level FUSE component to a user-space application (written against the FUSE library). This has enabled us to rapidly build a proof-of-concept primarily in user-space, without the complexity of in-kernel hacking.

3.2 API for DEB-Enabled Applications

This section describes an API for DEB-enabled applications to create, access and modify DEBs. The API's functions fall into three categories. The first category of functions facilitate the creation of DEBs and the introduction of "blobs" of digital evidence into DEBs. A blob is an arbitrary unit of digital evidence and might be a disk image, a single document or a compound file type. The second category allows access to a DEB's tags. Recall that the tags record DEB metadata, e.g., the investigating agent's name and contact information. The third category provides access to the DEB's audit log, so that applications can insert additional entries into the log to document investigative operations. Using any of the functions automatically introduces entries into the DEB's audit log. The functions are described briefly below.

- `int CreateDEB(char *filename, char *applicationinfo, char *comment, /* variable number of DEB tags */);`

  This function creates a new DEB whose complete pathname is `filename`. The `comment` field is a free-form string entered into the audit log to describe the creation event, while `applicationinfo` documents the application that created the DEB. A variable number of tags, which document the investigator's name, contact infor-

mation, and case characteristics are permitted. An initial entry is made in the DEB's audit log to document the creation event. This entry also contains a hash of the initial DEB contents, which at this stage are essentially metadata. The `AddDEBBlob()` function, described below, allows evidence to be introduced into the bag. A positive return value indicates successful creation of the bag.

- `int AddDEBBlob(char *filename, char *blobname, void *blob, char *applicationinfo, char *comment);`

  This function introduces a new piece of digital evidence, `blob`, named `blobname`, into the bag whose pathname is `filename`. The `blobname` must uniquely identify the piece of digital evidence in the DEB, otherwise an error is generated. The `comment` is introduced into the DEB's audit log to describe the digital evidence introduced, while `applicationinfo` documents the application itself. In addition, audit log entries are automatically written to document the cryptographic hash of the introduced evidence plus a hash of the entire bag contents after the introduction of the `blob` is completed. A positive return value indicates successful introduction of the `blob`.

- `int AddDEBBlobFile(char *filename, char *blobname, char *blobfilename, char *applicationinfo, char *comment);`

  This function performs the same operations as `AddDEBBlob()`, except that the digital evidence that is introduced is contained in the file `blobfilename`, instead of in a block of memory.

- `int OpenDEBBlob(char *filename, char *blobname, int mode, char *applicationinfo, char *comment);`

  This function returns a file handle attached to the blob `blobname` contained in the DEB identified by `filename`. The file handle is opened with read/write permissions described by `mode`, which has the same semantics as the mode parameter for the standard C `open()` function. The `applicationinfo` argument describes the application issuing the open command while the `comment` describes the open operation (from the opening application's perspective) in the DEB's audit log. A positive return value indicates success.

- `void CloseDEBBlob(int handle, char *comment);`

  This function releases the file handle, `handle`, attached to a single blob of evidence in a DEB. The `comment` describes the close operation (from the closing application's perspective) in the DEB's audit log.

- `unsigned long long ReadDEBBlobBlock(int handle, void *data, unsigned long long len, char *comment);`

  This function reads a block of `data` from the stream identified by `handle`. The `handle` must have been obtained from a call to `OpenDEBBlob()`. The length of the block to be read is `len`. The `comment` argument describes the read operation from the application's perspective. The function returns the number of bytes read.

- `unsigned long long WriteDEBBlobBlock(int handle, void *data, unsigned long long len, char *comment);`

  This function writes a block of `data` to the stream identified by `handle`. This `handle` must have been obtained from a call to `OpenDEBBlob()`. The length of the block to be written is `len`. The `comment` argument describes the write operation from the application's perspective. The function returns the number of bytes written.

- `char *GetDEBTagValue( char *filename, char *tagname, char *applicationinfo, char *comment);`

  This function returns a pointer to a string containing the value of the tag `tagname` associated with the DEB `filename`. The `applicationinfo` argument describes the application issuing the operation while `comment` describes the operation in further detail in the DEB's audit log. The function returns NULL if the tag's value cannot be returned.

- `int PutDEBTagValue(char *filename, char *tagname, char *applicationinfo, char *comment);`

  This function creates (or modifies) the tag `tagname`, setting (or replacing) its value by `tagvalue` for the DEB `filename`. The `applicationinfo` argument describes the application issuing the operation while `comment` describes the operation in further detail in the DEB's audit log. A positive return value indicates successful modification of the tag.

- `int OpenDEBAuditLog(char *filename, char *application info, char *comment);`

  This function returns a file handle associated with the audit log for the DEB `filename`. The file handle's mode is read-only. This primary use of the function is to review the audit log. To modify the audit log, the function `AppendDEBAuditLog()` must be invoked.

- `void CloseDEBAuditLog(int handle, char *application info, char *comment);`

  This function closes the audit log stream associated with `handle`.

- `int AppendDEBAuditLog(char *filename, char *auditentry, char *applicationinfo, char *comment);`

  This function appends a log entry `auditentry` to the audit log associated with the DEB `filename`. A positive return value indicates a successful append operation.

3.3 Support for Non-DEB-Enabled Applications

Native file system support for DEBs enables them to be used even with non-enabled applications. Rather than using the API described in Section 3.2, legacy applications may use standard C library `open()`, `close()`, `read()` and `write()` operations (and their buffered counterparts) on digital evidence blobs in a DEB. The `open()` system call is modified to return a handle to a blob in a DEB; operations against the returned handle target the associated digital evidence blob rather than the DEB itself. Hooks in the implementation of these system calls can identify the process name, process number and affected blocks, facilitating transparent updates of the audit log in the DEB. This information is useful not only in identifying which legacy applications have accessed the DEB, but also in auditing the behavior of a legacy application. For example, unauthorized write operations can be readily identified from the audit log. Also, the "thoroughness" of an application can be identified, by ensuring that it truly accesses all the blocks in a blob of digital evidence.

We have developed a prototype system for native file system support of non-DEB enabled applications based on FUSE. In our prototype, user-level applications are used to import and export DEBs into and out of a special DEB-aware FUSE file system. An import operation essentially splits the DEB into component files and places these files in a special directory, along with the DEB audit log and other metadata. Legacy access to digital evidence blobs in these special directories automatically

results in audit log updates. For example, read access to a digital evidence blob causes the application name (and process number), access time, portions of the blob accessed, and optionally, a hash of the executable of the accessing application to be recorded. The creation of a new blob of digital evidence results in a similar audit log entry. Exporting a DEB from the DEB-enabled file system simply recreates the DEB structure from the data stored in the corresponding directory.

Table 1. Scalpel JPG file carving results (1 GB disk image).

Scalpel v1.52	**Time**
File carving on ext3 file system (no legacy DEB support)	3 min. 12 sec.
File carving on ext3 file system (legacy DEB-enabled FS)	3 min. 29 sec.

We ran several experiments to determine the overhead of automatically auditing access to digital evidence blobs. Table 1 presents the results obtained when Scalpel was used to carve JPEG files from an 1 GB disk image. The test was run under Linux on a 1.6 GHz Pentium M Thinkpad with 2 GB of RAM.

Table 2. FTK evidence processing results (8 GB disk image).

FTK v1.60	**Time**
Add Evidence step on Samba share (no legacy DEB support)	47 min. 56 sec.
Add Evidence step on Samba share (legacy DEB-enabled FS)	59 min. 04 sec.

Table 2 shows the results obtained for FTK v1.60's Add Evidence step on an 8 GB disk image. FTK was run on a 3 GHz Pentium 4 desktop with 2 GB of RAM. Access to the DEB-enabled file system was through Samba over a 100 Mb Ethernet connection. The Samba server was a 1.7 GHz Thinkpad with 512 MB of RAM.

Under Linux, with direct access to the DEB file system, the overhead is approximately 9%. Over a Samba mount, FTK showed about 23% overhead, but further investigation indicated that the Windows XP platform running FTK was issuing two parallel, non-overlapping sequences of read operations through Samba, even when application accesses were strictly sequential. Running the Scalpel file carver under Windows over Samba to access a DEB-enabled file system showed similar overhead (approximately 25%; this result is not shown in the tables). We plan to investigate this strange Samba behavior in the future, but note that other developers have seen similar behavior in Windows XP.

Naturally, there are limitations to providing automatic auditing of DEB-unaware applications. For one, the audit log is not as "tidy" as it might be if auditing were controlled by a compliant application using the DEB API. This is because audit log entries for reads (or writes) that serve a common purpose cannot be easily grouped; since our prototype does not have high-level application knowledge, it can only track low-level file operations. Another limitation is that access to the special DEB directories via a network share, e.g., Samba, obfuscates the name of the application touching a blob of digital evidence. For example, if a Windows application accesses DEB data through a Samba share, the audit log shows the Samba daemon under Linux (`smbd`) as the accessing application. Still, we believe our legacy application support is a good interim solution as legacy applications are modified to use common DEB formats or are replaced with DEB-compliant applications.

3.4 Secure Audit Logs

To further strengthen DEB auditing capabilities, anti-tampering facilities can be introduced for DEB contents, especially the audit log. Our goal is not to prevent tampering of the audit log and DEB contents, but rather, to solve the slightly easier problem of detecting tampering. In general, secure auditing facilities require a trusted component. This component can be a WORM drive to which audit log entries are appended, or a secure server that is physically inaccessible to an attacker. In the following, we discuss some design choices.

Schneier and Kelsey [3] presented a scheme for secure auditing, which involves an untrusted machine U (e.g., a machine used in a digital investigation) that shares a secret A_0 with a trusted machine T. To append a new log entry D_j, U computes $K_j = hash(A_j), C = E_k(D_j), Y_j = hash(Y_{j-1}; C)$, and $Z_j = MAC_{A_j}(Y_j)$. Y_j is the jth entry in a hash chain, where $Y_1 = 0$ and MAC is a keyed hash function. Then, $[C, Y_j, Z_j]$ is written to the log. The shared secret is then recomputed: $A_{j+1} = hash(A_j)$, and A_j is destroyed. This scheme is tailored to disallow log entries created before a compromise at time t from being read by an attacker. The idea is that the attacker is then left to delete the entire log (which will be noticed when communication is established in the future between U and T) or leave the log alone (and not know if a log entry has recorded his unauthorized access). The scheme is useful if access to previous log entries by applications running on U is not required. Note that T can verify that the audit log on U is correct because it possesses A_0 and can "replay" the entire log.

Snodgrass *et al.* [6] have proposed a technique that allows read access to an audit log while preventing widespread tampering of the log. The scheme uses a trusted notary service, which accepts a digital document, computes a hash of the document and a secure timestamp and then stores and returns a notary ID. This notary ID is stored with the log entry. To determine if the audit log is consistent, a trusted party can verify that the notary IDs (and associated timestamps) on the notary service match those in the audit log. Omissions, additions and deletions can all be identified. This basic scheme has the drawback of requiring significant communication with the notary service, but audit log entries can be combined and submitted as a single document to the notary to reduce overhead (at the expense of a coarser level of log validation). The Snodgrass approach is particularly attractive for DEB audit logs as only limited storage is required on the trusted server. For each audit log entry, a hash is computed for the text of the log entry, this hash is submitted to the notary service, and the notary ID received is then stored in the DEB's audit log. Note that the DEB's audit log is readable by any application, which is useful for creating reports, evaluating an investigation, or performing tool evaluation.

4. Conclusions

Digital Evidence Bags (DEBs) mimic traditional evidence bags by providing a standard container for arbitrary digital evidence, with an integrated audit log and metadata that describes the evidence and the forensic processes applied to the evidence. Digital-forensics-aware operating system components – as provided by native file system support for DEBs – can significantly improve the performance and consistency of forensic investigations. The power of DEBs is increased substantially by providing a standard API and native file system support, because new applications (specifically written to support DEBs) and native applications (which use standard Unix system calls for I/O) can take advantage of automatic auditing of forensic operations.

Our system is a work in progress. However, once the initial implementation is stable, we expect to undertake a thorough performance study and determine whether user-level file system enhancements offer sufficient performance, or whether modifications to an existing file system, such as ext3, are actually necessary.

References

[1] AccessData Corporation, Forensic Toolkit (FTK) (www.accessdata.com).

[2] V. Roussev and G. Richard III, Breaking the performance wall: The case for distributed digital forensics, *Proceedings of the Fourth Digital Forensics Research Workshop*, 2004.

[3] B. Schneier and J. Kelsey, Secure audit logs to support computer forensics, *ACM Transactions on Information and System Security*, vol. 2(2), pp. 159-176, 1999.

[4] Sleuthkit.org, Autopsy (www.sleuthkit.org).

[5] Sleuthkit.org, Sleuth Kit (www.sleuthkit.org).

[6] R. Snodgrass, S. Yao and C. Collberg, Tamper detection in audit logs, *Proceedings of the Thirtieth International Conference on Very Large Databases*, pp. 504-515, 2004.

[7] SourceForge.net, FUSE: Filesystem in user space (fuse.sourceforge .net).

[8] P. Turner, Unification of digital evidence from disparate sources (digital evidence bags), *Proceedings of the Fifth Annual Digital Forensics Research Workshop*, 2005.