SECURE, AUDITED PROCESSING OF DIGITAL EVIDENCE: FILESYSTEM SUPPORT FOR DIGITAL EVIDENCE BAGS

Golden G. Richard III and Vassil Roussev
Department of Computer Science, University of New Orleans
New Orleans, Louisiana 70148, USA
golden,vassil@cs.uno.edu

Abstract Traditional digital forensics methods capture, preserve, and analyze digital 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 of a digital investigation, from identification and seizure of evidence through duplication and investigation is essentially ad hoc, recorded in separate log files or in an investigator's case notebook. Auditing performed in this fashion is bound to be incomplete, because different tools provide widely disparate amounts of auditing information. 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 of their actions (e.g., *dd* or the command line tools of the Sleuth Kit) and some that generate their own audit logs (e.g., FTK). At the end, an investigator is left to piece together these bits of audit trail to create a comprehensive view of what occurred during the investigation.

Digital Evidence Bags (DEBs) are a recently proposed mechanism for bundling digital evidence, associated metadata, and audit logs into a single structure. DEBs categorize the digital evidence they contain and provide a mechanism for associating an audit log that details the investigative processes that have been applied throughout an investigation. 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 filesystem support for DEBs, which has a number of benefits over ad hoc modification of digital evidence bags. The first is that some of the advantages of DEBs can be realized even for current generation tools which are DEB-unaware, since a DEB-enabled filesystem can transparently offer the contents of a digital bag 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 updating a DEB, both for introducing and removing items and for updating the audit log, needs to be certified only once. Finally, a stan-

dard API for accessing DEBs will greatly reduce the effort in adding DEB support to both existing and future applications. Native filesystem support for a digital evidence container is an example of what we believe is an urgent need in the digital forensics community: digital forensics-aware operating systems components, which can increase the consistency, security and performance of digital forensics investigations.

Keywords: Digital forensics, operating systems internals, filesystems, digital evidence bags

1. Introduction

Currently, the majority of digital forensics tools operate over standard operating systems components-for example, standard filesystems and caching mechanisms. But there are compelling performance, consistency, and security reasons for making operating systems components digital forensics-aware. These include performance (e.g., better data distribution and clustering mechanisms, particularly for distributed digital forensics [7]), security (e.g., protection of digital evidence from unauthorized access or tampering), and consistency. In this paper, we consider both the advantages and design challenges of forensics-aware filesystems. Specifically, we consider how auditing of digital evidence is currently handled and how an enhanced filesystem can make this process much more automated and more accurate.

Evidence bags and seals are a standard item in traditional crime scene investigation. Bags and seals allow evidence to be preserved and categorized 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. Further, 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 allow tracking the movement of the bag, noting the chain of custodians who have undertaken the bag's care.

Traditional digital forensics methods capture, preserve, and analyze digital 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 of a digital investigation, from identification and seizure of evidence through duplication and investigation is essentially ad hoc, recorded in separate log files or in an investigator's case notebook. For example, *dd* provides no direct method for capturing information such as when the imaging operation took place, who performed the operation, or an integrity

check. This information must be recorded separately and in the case of the integrity check, additional commands (such as *md5sum* or a similar cryptographic hashing command) must be executed and the output recorded manually. While enhanced versions of *dd* exist, such as *dcfldd*, adding integrity checks to each application is tedious and error-prone. In addition, the integrity problem is aggravated further 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 which 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 of their actions (e.g., *dd* or the command line tools of the Sleuth Kit [1]) and some that generate their own audit logs (e.g., FTK [2]). At the end, an investigator is left to piece together these bits of audit trail to create a comprehensive view of what occurred during the investigation. Further, failure to record a bit of information, such as the MD5 hash generated by *md5sum* for a large disk image, could potentially result in a huge amount of lost time if the operation must be repeated.

Digital Evidence Bags (DEBs) [6] provide a new approach for storing and processing digital evidence obtained from a variety of sources and serve as a universal container for digital evidence, much as traditional evidence bags serve as 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 digital evidence in the bag as well as integrity checks. In addition to providing increased security for digital evidence, the audit log details the investigative processes that have been applied throughout an investigation. This is potentially useful from educational and evaluation standpoints, allowing novice investigators to see what steps were taken, which tools were used, and in which order 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. In this paper, we suggest that while adoption of a standard format for storage of digital evidence will offer radical improvements in the investigative process, system support for DEBs will be even more useful. This paper investigates native filesystem support for DEBs, in which the basic file type is a Digital Evidence Bag. This provides a number of benefits over ad hoc implementations of DEB support. The first is that some of the advantages of DEBs can be realized even for current generation tools

which are DEB-unaware, since a DEB-enabled filesystem can transparently offer the contents of a digital bag 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 introduction and removal of items from the bag and updates to the audit log and metadata, needs to be certified only once. Finally, a standard API for accessing DEBs will greatly reduce the effort in adding DEB support to both existing and future applications.

2. Basic DEB Structure

The basic structure of Digital Evidence Bags (DEBs) is outlined in [6]. A working group was discussed at DFRWS 2005 to provide a forum for further development of the format and as this paper is written, no final standard has been proposed. As such, we sketch the structure of a DEB at a high level only, discussing DEB components necessary for native file system support. The reader is referred to [6] for a possible concrete implementation. 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. This information includes a unique identifier for the DEB, the creation date and time, the name and organization of the creator of the DEB, 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 in detail, 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 [6]) tracks the operations performed against a DEB, including the date, time, affected blocks, application signature, of each operation as well as periodic hashes of the DEB contents. We note in Section 3.4 that secure auditing techniques can be used to protect the DEB contents against tampering.

3. Design Overview

The basic ideas in providing native filesystem support for DEBs are:

- to allow transparent import of DEBs into an enabled filesystem-that is, automatic conversion of DEBs copied into the filesystem into a native storage format.

- to allow transparent export of DEBs from an enabled filesystem to a "normal" filesystem-that is, automatic conversion of DEBs

stored in the enabled filesystem to a popular container format, such as XML.

- to allow provide efficient, secure, easy-to-use access to DEBs for both new, DEB-enabled applications as well as legacy applications.

The next section surveys some design choices for meeting these goals. The following sections define an API for DEB-enabled applications, discuss how native applications can transparently access DEBs, and discuss methods for securing the DEB audit log.

3.1 Design Choices

When stored outside the DEB-enabled filesystem, a format such as XML is a sensible choice for DEB files. In the draft specification outlined in [6], plain text was used for components of a DEB. A more efficient format is required, however, for native storage of DEBs. This is because unlike other compound file types, such as ZIP files or tarballs, DEBs will be updated quite frequently as units of evidence are introduced and the audit log is 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 filesystem should support not only efficient updates of the audit log, but also fast access to the digital evidence blobs.

We evaluated the capabilities of several filesystems before choosing a candidate for native DEB support, including ext2/3, reiserFS, and XFS. NTFS was eliminated from consideration because 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 these filesystems contain features that can potentially be used to efficiently support DEBs, such as extended attributes (EAs). Unfortunately, the extended attributes implementation in ext2/3 places substantial limits on the maximum size of EAs (one disk block), precluding their use to store larger components in a DEB. Similarly, XFS and stable versions of reiserFS have limitations on maximum sizes of EAs. In a future version of reiserFS, EAs will be stored as regular files in the filesystem, with the filename referring to the name of the EA and the file contents being the associated value of the extended attribute. We are adopting a similar strategy for storing DEB components, using symbolic links. This is discussed further below.

In the end, we believe that the best choice is to use a standard filesystem, such as ext3. The resource forks within a DEB can be stored as separate files by using symbolic links stored within 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 the proposed DEB API and to support 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.

To test our ideas, we are currently using a user-level filesystem, FUSE (File System in User Space) [3], for prototyping. In FUSE, system calls are redirected by a kernel-level FUSE component into a user-space application (written against the FUSE library). This has allowed us to rapidly build a proof-of-concept primarily in user-space, without the complexity of in-kernel hacking.

3.2 An API for DEB-enabled Applications

In this section we present an API for enabled applications to create, access, and modify DEBs. The functions fall into three categories. The first group of functions allow digital evidence bags to be created and for "blobs" of digital evidence to be introduced into a DEB. 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 group of functions allows access to a DEB's tags. Recall that the tags record DEB metadata, such as the investigating agent's name and contact information. The third group of functions provide access to the DEB's audit log, so that applications can insert additional entries into the log to document investigative operations. Use of any of the functions automatically introduces entries into the DEB's audit log. Each function is described briefly below.

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

  Creates a new Digital Evidence Bag 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 creating the DEB. A variable number of tags, which document the investigator's name, contact information, 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 digital 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);*

  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 introduction 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 to introduce is contained in the file named *blobfilename*, instead of in a block of memory.

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

  Returns a file handle attached to the blob with name *blobname* contained in the DEB whose complete pathname is *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 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);*

  Releases the file handle *handle*, attached to a single blob of evidence in a DEB. The *comment* describes the close operation (from the close-ing application's perspective) in the DEB's audit log.

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

  Reads a block of *data* from the stream identified by *handle*. This handle must have been obtained from a call to OpenDEBBlob(). The length of the block to read is *len*. The *comment* argument

describes the read operation from the application's perspective. Returns the number of bytes read.

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

  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 write is *len*. The *comment* argument describes the write operation from the application's perspective. Returns the number of bytes written.

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

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

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

  Creates (or modifies) the tag identified by *tagname*, setting (or replacing) its value by *tagvalue* for the DEB identified by *filename*. The *applicationinfo* argument describes the application issuing the operation while the *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 *applicationinfo, char *comment);*

  Returns a file handle associated with the audit log for the DEB *filename*. The file handle's mode is read-only. This function's primary use is for reviewing the audit log. To modify the audit log, AppendDEBAuditLog() must be used.

- *void CloseDEBAuditLog(int handle, char *applicationinfo, char *comment);*

  Closes the audit log stream associated with *handle*.

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

> Appends a log entry auditentry to the audit log associated with the DEB identified by *filename*. A positive return value indicates a successful append operation.

3.3 Support for Non-DEB-enabled Applications

Native filesystem support for Digital Evidence Bags allows DEBs to be used even with non-enabled applications. Rather than using the API suggested in Section 3.2, legacy applications may simply use the standard C library open(), close(), read(), and write() operations (and their buffered counterparts) on digital evidence blobs contained within a DEB. The open() system call is modified to return a handle to a blob within a DEB and operations against the returned handle target the associated digital evidence blob, rather than the DEB itself. Further, hooks in the implementation of these system calls can identify the process name, process number, and affected blocks, allowing transparent update of the audit log in the DEB. This information is useful not only in identifying which legacy applications accessed the DEB, but also in auditing the correct behavior of a legacy application. For example, unauthorized write operations against a unit of digital evidence can be readily identified from the audit log. Further, the "thoroughness" of an application can be identified, by ensuring that it truly accesses all of the blocks comprising a blob of digital evidence.

We have developed a prototype system for native filesystem support for 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 filesystem. 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 the blobs of digital evidence in these special directories automatically results in updates of the audit log. For example, read access to a blob of digital evidence results in auditing of the application name (and process number), the time the access occurred, which portions of the blob were accessed, and optionally, a hash of the executable of the accessing application. Creation of new blobs of digital evidence results in similar audit log entries. Exporting a DEB from the DEB-enabled filesystem simply recreates the DEB structure from the data stored in the corresponding directory.

We ran a number of experiments to determine the overhead of automatically auditing access to the digital evidence blobs. Tables 1 and 2 present representative performance data for use of our DEB-enabled filesystem. Under Linux, with direct access to the DEB filesystem, over-

Scalpel v1.52 on ext3 FS, no legacy DEB support	3m12s
Scalpel v1.52 on ext3 FS, legacy DEB-enabled FS	3m29s

Table 1. Scalpel JPG file carve on 1GB disk image, 1.6GHz Pentium M T40p Thinkpad with 2GB RAM.

FTK v1.60 Add Evidence on Samba share, no legacy DEB support	47m56s
FTK v1.60 Add Evidence on Samba share, legacy DEB-enabled FS	59m4s

Table 2. FTK Add Evidence processing for 8GB disk image. FTK running on 3GHz Pentium 4 desktop with 15K SCSI disks, 2GB RAM. 8GB disk image served over 100Mb Ethernet by 1.7GHz Pentium 4 A31p Thinkpad with / 512MB RAM.

head is approximately 9%. Over a Samba mount, FTK showed about 23% overhead, but further investigation indicated that the Windows XP box 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 filesystem showed similar overhead (approximately 25%; this result is not shown in the tables). We will investigate this strange Samba behavior in the future, but note that other developers have seen similar behavior in Windows XP.

Naturally, there are limitations in providing automatic auditing of DEB-unaware applications. For one thing, the audit log is not as 'tidy' as it might be if auditing were controlled by a compliant application using our DEB API. This is because audit log entries for reads (or writes) which 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, for example, via 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 will show *smbd* (the Samba daemon under Linux) as the accessing application. Still, we believe our legacy application support is useful as an interim solution, as legacy applications are either modified to use common DEB formats or replaced with DEB-compliant applications.

3.4 Secure Audit Logs

To further strengthen the auditing capabilities for Digital Evidence Bags, anti-tampering facilities can be introduced for the DEB contents, particularly the audit log. Our goal is not to prevent tampering with the audit log or contents of a DEB, but rather to solve a slightly easier problem, to make tampering detectable. In general, secure auditing facilities require a trusted component. This component can be either a WORM (Write Once Read Many) drive, to which audit log entries are appended, or a secure server, physically inaccessible to an attacker. We discuss a few design choices. Schneier and Kelsey [5] discusses one scheme for secure auditing, which involves an untrusted machine U (e.g., a machine used in a digital investigation) which 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 specifically 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 thus not know whether an entry in the log makes note of his unauthorized access). This scheme is useful if access to previous log entries by applications running on U aren't needed. 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 [4] have proposed a technique that allows read access to the log while preventing widespread tampering with the audit log. The scheme makes use of a trusted notary service, which accepts digital documents, computes a hash function over the document and a secure timestamp and then stores and returns a notary ID. This notary ID is then 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 noticed. This basic scheme has the drawback of requiring a significant amount of communication with the notary service, but audit log entries can be combined and submitted as a single document to the notary in order to reduce communication (at the expense of a coarser level of log validation). For DEB audit logs, the Snodgrass approach is particularly attractive, since only a limited amount of storage is required on the trusted server. For each audit

log entry, a hash is computed over the text of the log entry, this hash is submitted to the notary service, and the notary ID returned 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 and Future Work

Digital forensics-aware operating system components have the potential to significantly improve the investigative process, enhancing a number of factors, from performance to consistency. In this paper, we explored one example, examining how introducing native support for Digital Evidence Bags improves auditing in an investigation. 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 investigators. The power of DEBs is increased substantially by providing both a standard API and native filesystem support, because both new applications (specifically written to support DEBs) and native applications (which use the standard Unix system calls for I/O) can take advantage of automatic auditing of forensic operations.

Our system is a work in progress and implementation is not yet complete. Once the initial implementation is stable, we expect to undertake a more thorough performance study and determine whether user level filesystem enhancements offer sufficient performance, or whether modifications to an existing filesystem, such as ext3, are actually necessary.

References

[1] Sleuthkit and Autopsy, http://www.sleuthkit.org.

[2] Forensics Toolkit (FTK), http://www.accessdata.com.

[3] "FUSE: Filesystem In User Space," http://fuse.sourceforge.net/.

[4] R. Snodgrass, S. S. Yao, C. Colberg, "Tamper Detection in Audit Logs," *Proceedings of the 30th VLDB Conference*, Toronto, 2004.

[5] B. Schneier, J, Kelsey, "Secure Audit Logs to Support Computer Forensics," *Proceedings of ACM Transactions on Information and System Security*, vol 2, no 2, May 1999.

[6] P. Turner, "Unification of Digital Evidence from Disparate Sources (Digital Evidence Bags)," *Proceedings of the 5th Annual Digital Forensics Research Workshop* (DFRWS 2005).

[7] V. Roussev, G. G. Richard III, "Breaking the Performance Wall: The Case for Distributed Digital Forensics," *Proceedings of the 2004 Digital Forensics Research Workshop* (DFRWS 2004)