Why secure delete doesn't work and no sensitive data should ever be stored unencrypted on disk

Brad House

Main Street Softworks, Inc.
brad at monetra.com

The most common and recommended method of securely deleting a file (as per mil specs) requires a seven round rotation of writing 0's then 1's to the same disk location(s) where the file data was stored to ensure the data is not recoverable. Most implementations of secure delete routines use the standard file access methods available such as [f]open/read/write/close which rely on one very important assumption, that the filesystem overwrites data in place and that the address on disk is consistent at a hardware level. Historically, this has been true, but with modern filesystem and hardware designs, these assumptions cannot be guaranteed.

Examples of scenarios that the most common secure delete practices will _not_ be effective on are modern filesystems including log-structured or journaled file systems (such as JFS, ReiserFS, XFS, Ext3, etc), file systems that write redundant data and carry on in the event of write failures, filesystems that make snapshots, filesystems that cache in temporary locations (such as NFS3), and compressed file systems. To understand why simply requesting the OS to overwrite portions of a file will not result in a secure delete on modern filesystems requires some basic understanding of the way data is stored on disk. A very simplistic view will be outlined here.

A file is stored on disk using two primary components. One component is the file metadata which stores things like the file name, size, location(s) (blocks) on the physical media (disk), etc. The other component is the physical data blocks which hold the data (meat) of the file. A file can spread multiple data blocks, but those data blocks do not necessarily have to be contiguous. Data blocks are typically a predetermined size and it is up to the filesystem to keep track of which blocks make up a file. A modern journaling filesystem's primary goal is data integrity. In the event of a power or system failure, worst case scenario, the last few modifications would be lost. The result of a power or system failure should never end with data corruption or partially written data. To obtain this primary goal, a filesystem will maintain a 'journal' of actions being performed which have not been committed to disk. This journal exists in a fixed place on the disk and has a reserved pre-determined size and simply contains the 'metadata' changes which are taking place. In the event of a failure, a filesystem will use the journal to 'undo' any operations (again, only on the metadata) that had been performed. Most Journaling filesystems have a pessimistic outlook on their operating environment and will always expect an error to occur, and this is where the primary roadblock comes in to play with regards to a secure delete process. During file update operations, such as the act of attempting to overwrite the contents of a file, the filesystem will write the updates to a different physical location (block or blocks) on the disk, and simply update the file metadata to reference the new blocks and mark the old blocks as unused. It does not overwrite the data of the old blocks, that potentially sensitive data resides on the disk until the blocks are used for storage of another file. If the filesystem were to overwrite the original blocks containing data and in the middle a power or system failure were to occur, the file would be in an undefined state (meaning it wouldn't have the contents prior to the intended operation or after, it would have partially written, corrupt data).

Some file systems may provide routines to securely delete a file for you, but most will not. Each filesystem has a different on-disk layout, so a programmer wishing to securely delete a file must have a complete and deep understanding of the on-disk layout of each filesystem the intended application runs. Using that understanding, the programmer must be able to read the metadata of the intended file in order to locate the physical blocks used to store the file contents for the purpose of overwriting that physical block. To add to that, the application would be required to be designed to run at the highest privileges possible as it would require direct, raw access to the disk in order to carry out the secure delete operations (which in itself could pose a security risk). Though third party tools do exist for some select few operating systems and file systems, they have typically received no formal review to guarantee they do what they advertise, to complicate matters further, most are not designed to be used as a library or to be integrated into a 3rd party application, implying the need for end-user intervention when required.

The latest generation of storage media such as Compact Flash and Solid State Drives (SSD) have a limited lifespan per physical block and employ tactics to work around the shortcomings to the detriment of the secure delete process. Typical flash-based media can only handle as many as one million writes per block, though some are limited by as few as ten thousand writes per block. To compensate for this shortcoming, manufacturers have employed a technology known as 'write leveling'. Write leveling is implemented as an address abstraction layer where all free physical blocks are held in a pool, and a chain of physical blocks are assigned to every virtual block (most commonly round-robin) when a new virtual block is allocated (at write time). With write leveling hardware, it is impossible to determine where on the actual media the data has been written. Using the mil specs example, each of the seven rounds of writes could theoretically be written to a different section of the physical media (even though the software requested what it thought to be an exact physical address), leaving the initial data to be cleared (as well as each of the subsequent writes of ones and zeros) fully intact. The only way to ensure that sensitive data is not retrievable past its intended lifespan is to store that data securely to begin with by using a strong encryption algorithm which has had sufficient peer review.

Proof of Concept

Main Street has developed proof of concept code and test scenarios that can be made available to any validated PA QSA accredited entity, by request. If you are interested in learning more simply send an email to info@monetra.com and an associate will be in touch.

References

http://www.fsl.cs.sunysb.edu/docs/secdel/secdel.html

http://www.usenix.org/publications/library/proceedings/sec96/full_papers/gutmann/index.html

http://ieeeia.org/sisw/2005/PreProceedings/07.pdf