Deletion poses a fundamental quandary to secure usability. On the one hand, users would like to be able to undo their mistakes—to undelete files after they have been deleted accidentally. On the other hand, users would like to know that sensitive data is actually removed from a disk when they delete it—removed so that it cannot be recovered by an adversary.

Anybody who has a paper shredder lives with this quandary. If you get one of those preapproved credit-card offers in the mail and you don’t need it, you can always just throw it into a recycling bin. If you change your mind and decide that the 0% introductory rate might help you finance a new laptop, you can always pull the offer out of the bin and fill it out. Of course, these preapproved offers can also be used by crooks in the commission of identity theft: if you are really sure that you don’t want to take out that new credit card, you’re better off shredding the offer and perhaps even the envelope in which it came. The best paper shredders make it easy for you to inspect the chad to make sure that the information is no longer intelligible. Some companies that care about their data security let their employees throw whatever documents they wish into recycling, but the paper is then shredded before it is given to a waste hauler.

Today’s computers use the metaphors of folders, files, recyclers, and paper shredders frequently to describe how information is stored and erased, but few actually work the
way that these metaphors imply. In fact, they work in a way that is perverse and truly anti-user: when a file is deleted—perhaps by putting it in the Windows Recycle Bin and then emptying the recycler—normal users can no longer recover the contents of their files, but specialists armed with special forensic tools frequently can. Simply put, the DELETE key lies.

Nobody set out to make delete a deceitful act, it just sort of happened. And it’s something that can be undone. If computers were programmed to simply overwrite data when that data was “deleted,” a process commonly called sanitization, this problem would not exist. But changing the behavior of a function that’s nearly 40 years old can be hard work.

This chapter addresses the question of sanitization and usability. My argument, based on an analysis of operating systems and the results of a data forensics investigation, is that the time has come to redesign the way that operating systems implement DELETE.

Introduction

Visibility is a powerful tool for aligning security and usability. All too often, hidden properties, functionality, or data storage that is part of a complex system can make it very difficult for a user to operate a system in a secure manner. Although it is possible, with significant effort, to teach users about hidden aspects of a system, an attractive alternative is to remove the opportunities for a system’s visible state to be inconsistent with its internal state.

Many of the specific security and privacy problems facing users of the World Wide Web today are a direct result of the mismatch between what is visible to the user and what is actually happening inside the computer. Much of the initial furor over web cookies in the popular press that accompanied the introduction of Netscape’s 2.0 browser centered upon the fact that cookies were a hidden tracking device that was not generally visible to web site visitors. The same resentment was played out several years later when web sites and email marketers started tracking page views with web bugs. (For more discussion of web cookies and web bugs, see Chapters 24 and 23, respectively.)

Web browsers contain much personal information that is hidden from the average user, including browser caches, history lists, and databases used for form completion. In many cases, information left behind by the browser has revealed information that the computer user would have preferred to remain hidden. Novice Internet users are rarely aware of the fact that their browser records such information. But even advanced users can easily forget to clear their browser’s history, empty the browser’s cache, and explicitly delete the “form filling” database after using a borrowed browser.

This chapter looks at another way that private information may be compromised by modern computer systems: the improper sanitization of disk space when the files on the disk are “deleted.” Although the need to sanitize magnetic media properly has been recognized for more than 30 years, as of this writing, operating system developers have still refused to make proper sanitization a standard part of any computer system that is
widely available today. In the first section of this chapter, I present the results of a study that demonstrates the need for clean deletion in mass-market operating systems such as Windows, Mac OS, and Linux. In the second section, I explore sanitization standards and academic studies, and discuss support for sanitization in today’s operating systems. In the final section, I present a plan for incorporating sanitization into today’s mass-market systems.

The Remembrance of Data Passed Study

In August 1998, I was chief technology officer of a computer security start-up. One of my jobs involved setting up a test bed of modem-equipped computers that would answer incoming phone calls and respond with a variety of different prompts. Instead of purchasing new computers for this somewhat mundane task, I bought 10 used machines at $20 each from a small-town computer store. Most of the computers had been sitting on a shelf for more than a year, and the store’s owner didn’t even know if they worked. My plan was to mix and match the components until I had five or six operational systems.

When I got the computers back to my house and started to inventory the parts, I discovered that the computer store had neglected to sanitize the hard drives prior to selling me the machines. Intrigued, I inventoried the drives and discovered the following:

• One of the larger machines, a ‘486-class system with a 40-gigabyte hard drive, had been the Novell file server for a small law firm. The computer had considerable client material on it, including contracts, wills, and billing records.

• A second computer had been used by an organization that delivered mental health services to community residents under contract to a state agency. The computer included a FileMaker Pro database that had the names, addresses, and diagnoses of several dozen individuals living in the community.

• A third machine had belonged to a writer who worked for a national magazine and also wrote novels. This machine contained many unpublished works, works-in-progress, and correspondence.

• A fourth machine had letters sent between a woman and her daughter in college. This computer also had a copy of Quicken, which the woman apparently used to manage her finances.

All of this information was visible once the computers were turned on; no special disk recovery software was needed at all. I called the store’s owner. Once he got over his shock and embarrassment, he asked me to wipe the systems as a favor. Apparently, he had meant to sanitize the machines before he sold them, but he had forgotten to do so.
Other Anecdotal Information

My experience with data left on disks that were subsequently sold on the secondary market is hardly unique. In recent years, there have been numerous reports of such cases, including:

- In April 1997, a woman in Pahrump, Nevada, purchased a used IBM PC and discovered records from 2,000 patients who had prescriptions filled at a Smitty’s Supermarkets pharmacy in Tempe, Arizona.¹
- In August 2001, more than 100 computers from the consulting firm Viant containing confidential client data were sold at auction by Dovebid following the closure of Viant’s San Francisco office.²
- In spring 2002, the Pennsylvania State Department of Labor and Industry sold computers containing “thousands of files of information about state employees.”³
- In August 2002, a Purdue student purchased a used Macintosh computer at the equipment exchange and discovered that the computer contained a FileMaker database with names and demographic information of 100 applicants to the Entomology Department.
- Also in August 2002, the United States Veterans Administration Medical Center in Indianapolis retired 139 computers. Some of these systems were donated to schools, others were sold on the open market, and at least three ended up in a thrift shop where they were purchased by a journalist. Examination of the computer hard drives revealed sensitive medical information, including the names of veterans with AIDS and mental health problems. Also found were 44 credit card numbers used by the Indianapolis facility.⁴
- In June 2004, the UK computer security firm Pointsec purchased 100 hard disks on eBay as part of a project on the “life cycle of a lost laptop.” Although all of the hard drives had “supposedly” been “wiped clean” or “reformatted,” the company was able to recover data from approximately 70 of the drives. The company also purchased laptops at auction that had been lost at airport terminals in England, Germany, Sweden, and the U.S. and verified that police did not sanitize the laptops prior to selling them. Reportedly, the laptop recovered from Sweden “contained sensitive information from a large food manufacturer. The data recovered included four Microsoft Access databases containing company and customer-related information and 15 Microsoft PowerPoint presentations containing highly sensitive company information.”⁵

While these cases are certainly notable, they represent a tiny fraction of the number of hard disks that are being repurposed, recycled, or otherwise resold on the secondary market.

According to the market research firm Dataquest, nearly 150 million disk drives will be retired in 2002—up from 130 million in 2001. Dataquest estimates that 7 disk drives will be retired for every 10 drives that ship in the year 2002; this is up from a 3-for-10 rate of retirement in 1997. Thus, more and more drives are being retired every year!

But the term *retired* is something of a misnomer. As the experience at the VA Hospital demonstrates, many disk drives that are “retired” by one organization can appear elsewhere. Indeed, mainstream businesses are increasingly turning to used equipment in an effort to cut costs—the editors at *CIO Magazine* even ran a cover story giving their readers advice on finding the best deals.7

These anecdotal reports are interesting both because of their similarity to each other and because of their relative scarcity. Clearly, confidential information has been disclosed through computers sold on the secondary market more than a few times. Why, then, have there been so few reports of unintended disclosure?

In the initial publication detailing this study,8 Shelat and I proposed three possible hypotheses to answer this question:

• Disclosure of so-called “data passed” information, while it occurs from time to time, is nevertheless exceedingly rare.

• Confidential information might be disclosed so often on retired systems that such events are simply not newsworthy.

• While used equipment is awash with confidential information, nobody is looking for it—or at least, few people who are looking for this data are publicizing the fact.

This chapter argues that the third hypothesis is correct. Based on a combination of the information found on the drives and interviews conducted with some of the original data owners, it seems that most confidential information on these “retired” drives is erased but not overwritten. As a result, I believe that many repurposed drives contain significant amounts of personal or confidential information, but few of the drives’ current users are aware of this fact.


**Study Methodology**

Between January 1999 and January 2003, I purchased 235 used hard drives on the secondary market in an effort to determine what information they contained and what, if any, means were taken to clean the drives before they were discarded. Initially the drives were purchased at used computer stores such as WeirdStuff in Sunnyvale, California, and PC Recycle in Bellevue, Washington. The majority of drives were purchased as the result of winning bids on the eBay online auction web site. Most purchases consisted of between 3 and 5 drives; in no case were more than 20 drives at a time from the same vendor.

Modern hard disks store information in individually addressable blocks, with each block being 512 bytes in length. A 50-gigabyte disk thus has approximately 10 million blocks.

On receipt, each drive was cataloged and entered into a database. Each drive was then attached to a computer running the FreeBSD operating system and the contents were copied off, block for block, using the command:

```
dd if=/dev/ad2 of=NNN.img conv=noerror,sync
```

where /dev/ad2 is the raw device of the disk, noerror instructs that the dd command should continue copying data even if an error is encountered, and sync specifies that error-containing blocks should be written to the output stream as all zeros.

A filesystem is the piece of a computer’s operating system that controls the allocation of disk blocks to individual files. Popular filesystems are FAT32 (used by Windows 3.1, Windows 95, and Windows 98), NTFS (used by Windows NT, 2000, and XP), FFS (used by BSD Unix), and ext2fs (used by Linux). The following discussion is for the FAT32 filesystem, but it applies to all modern filesystems with only minor changes.

Once the images were created, they were mounted with FreeBSD’s “memory disk” driver. I then attempted to read the data in the image using FreeBSD’s native filesystem implementations for the FAT, NTFS, Novell, and Unix filesystems.

Of the 235 disks, 59 were dead on arrival, and the remaining 176 had data that could be read, for a total of 125 gigabytes of image files. Of these drives, 11 disks contained no data at all—that is, every block on these disks had been overwritten with ASCII NUL bytes. Another 22 disks appeared to have been overwritten completely and then formatted using the Windows FORMAT command. On these 22 disks, more than 99% of the blocks were blank. For the majority of the remaining disks, it appeared that little if anything had been done to remove the data of their previous owners.

Further examination appeared to contradict this conclusion. The remaining disks contained relatively large amounts of recoverable data. Nevertheless, a relatively small percentage of this data seemed to actually reside in files. There were only 168,459 files on
the 176 readable drives, accounting for just 38,296,9039 of the 190,681,765 non-zero disk blocks. Examining the files by file type, I found just 783 Microsoft Word files, 184 Microsoft Excel files, 30 Microsoft PowerPoint files, and just 11 Outlook PST files—numbers that seemed suspiciously low given that these were used disk drives.

Typical of the disks recovered was Disk #70, an IBM DALA 3540 that was purchased for $5 on eBay from a Massachusetts retail store. The disk contained 541 megabytes of data in 1,057,392 disk blocks (each disk block holds 512 bytes). Only 6% of the disk blocks were filled with ASCII NUL bytes; the rest contained data. Yet when the disk was mounted, just three files were observed—two of which were marked as “hidden” by the operating system:

- IO.SYS (hidden)
- MSDOS.SYS (hidden)
- COMMAND.COM

Where was the rest of the data?

**FORMAT Doesn’t Format**

Broadly speaking, modern disk drives have the ability to store two kinds of information. The majority of information stored by the device is *directly addressable user data*—these are the actual blocks that are written by the computer’s operating system onto the drive’s media in response to WRITE commands, and read back into the computer in response to READ commands. The second kind of information stored on the disk drive is *hidden data* that is used for the proper operation of the disk drive itself. This information includes the disk’s firmware and spare blocks that the drive will use when blocks containing directly addressable user data begin to fail.

When a manufacturer delivers a drive to the computer maker or end user, all blocks that will be used to hold directly addressable user data are filled with the ASCII NUL character—that is, the blocks are zeroed. (The hidden blocks generally are not zeroed, but they cannot be accessed by the computer’s operating system; for most practical purposes, these blocks do not exist.)

When a disk is formatted with the FAT filesystem, the Windows FORMAT command scans the entire disk, reading every block to make sure that the block is functioning. The FORMAT command then writes down *boot blocks*, the disk’s root directory, and finally a *file allocation table* that is used to distinguish blocks that are in use by the filesystem from those that are not. This process typically takes between 10 and 20 minutes, owing to the time required to read every block on the drive.

Once the root directory is written out, any information that was previously on the disk is rendered inaccessible. The data is still on the disk, but it cannot be retrieved using

---

9 This figure takes into account the fact that a 1-byte file takes an entire block, and a 1,025-byte file takes three blocks.
Windows because the files and directories of the disk cannot be reached by starting at the disk’s now empty root directory. Thus, the Windows FORMAT command doesn’t really erase the contents of the disk: it actually reads the entire disk and writes a new root. (Overwriting the FAT does make it more difficult to reassemble files that have been fragmented—that is, written partially in one location and partially in one or more others. This tends to make it harder, although not impossible, to recover large files.)

The failure of the FORMAT command to zero or otherwise initialize a hard drive has an interesting history. The first version of DOS, MS-DOS 1.0, worked only with floppy disks. At the time, floppies were sold without any track or sector information on their magnetic surface and they needed to be “formatted” before they could be used. In the process of formatting the disk, any bad blocks were detected and noted in the disk’s FAT so that they would not be used accidentally. If a floppy disk containing data was formatted, the information that it contained would necessarily be overwritten. This process took a few minutes. Thus, the initial meaning of “format” to PC users in 1981 was “a process that initializes a piece of magnetic media, making it usable, and destroying any data that the media might contain in the process.”

With the introduction of DOS 2.0, the first version of DOS that directly supported hard-disk drives, FORMAT of hard disks was made nondestructive. Because hard drives were sold already initialized, it was only necessary for the FORMAT command to literally write a format of data structures into the disk’s logical blocks so that the disk could be used with the operating system. But the FORMAT command continued to scan the entire disk for bad blocks—a process that might take between 10 and 30 minutes.

Thus, the FORMAT command gave the impression that it was overwriting the entire disk because it took a long time and because the resulting disk appeared to contain no data. But, in fact, no such overwriting took place. Not only does the FORMAT command turn visible data into invisible data, but it furthermore does so in a manner that is misleading. Equally misleading is the warning that the command displays:

```
A:\>format c:
WARNING, ALL DATA ON NON-REMOVABLE DISK
DRIVE C: WILL BE LOST!
proceed with Format (Y/N)?y

Formatting 1,007,96M
100 percent completed.
Writing out file allocation table
Complete.
Calculating free space (this may take several minutes)... Complete.

Volume label (11 characters, ENTER for none)?
1,054,851,072 bytes total disk space
1,054,851,072 bytes available on disk

4,096 bytes in each allocation unit.
```
The DOS 2.0 FORMAT command could have overwritten the entire disk, but this would have doubled the amount of time that the command required to prepare a new hard drive because every block would have needed to be both written and then read. The program’s creators appear to have made a tradeoff here between usability and security—increasing one while decreasing the other. Unfortunately, it was an invisible, undocumented tradeoff.

Microsoft could have done things differently. For example, the program’s creators could have put in a command-line switch that would have forced the program to first overwrite each block with NULs before it was read back. Then, the program could have been modified so that it would display one of two different messages. The “ALL DATA ... WILL BE LOST” message could have been used when the disk was actually overwritten, and a different message could have been used for the less severe option.

One reason that Microsoft’s engineers may not have gone in this direction is that the hard drives that were sold in the 1980s generally came with their own separately packaged “disk utilities.” Invariably, one of the “utilities” was a program that performed a so-called “low-level format” on the physical disk. The details of what a “low-level format” actually did varied from manufacturer to manufacturer and from drive to drive, but it generally was viewed as destroying all of the user-addressable information that the disk might contain. Mueller’s 1991 book, Que’s Guide to Data Recovery, noted that the key difference between a low-level format and a high-level format was that “you can recover data—unformat—from a high-level format.” Nevertheless, such knowledge did not diffuse into the general computer-user population.

It is incredibly misleading for an operating system to give the impression that all of the information has been removed from a disk when, in fact, the information has merely been made inaccessible to users who have not obtained special data recovery tools. Such a situation is an invitation for mishap: given a freshly formatted hard disk, there is no way for a user to audit the disk and determine if it is, in fact, clean, or if it has a treasure-trove of hidden, confidential information.

Modern versions of the Windows FORMAT command also have the ability to “quick format” a disk, which omits the media scan step. In this case, the entire disk can be formatted in just a few seconds. When Microsoft created the “quick format” option, the company could have gone back and changed the behavior of FORMAT when the “quick”
option wasn’t selected. Ideally, a non-quick format would actually overwrite the data on the disk. This would have aligned once again the internal workings of the commands with the effects that are visible to the user. Unfortunately, Microsoft left the behavior of the command as it was.

**DELETE Doesn’t Delete**

Just as today’s FORMAT command doesn’t actually format disks, it turns out that commands for erasing individual files do not actually perform that function, either. Instead of overwriting the actual data, commands like DELETE and ERASE simply remove the entry in the file’s containing directory and return the file’s blocks to the free list. What happens after the file is deleted depends upon many factors, including the amount of free space on the disk and the system’s pattern of usage.

Once again, the usability problem is that the operating system gives the user the appearance that the data has been removed from the computer when, in fact, the data has merely been made inaccessible by ordinary means. The usability problem for end users is compounded by the fact that there is no mention of this behavior in the Microsoft documentation. For example, the Windows built-in help for the DELETE command simply states that DEL “deletes one or more files.”

As before, this systematic deception on the part of DEL and ERASE wasn’t exactly secret—a 1987 advertisement for the Mace Utilities appearing in *The New York Times* noted that the $59.95 program could “Unformat, Undelete, Diagnose & Remedy” and much more. But mention that files could be undeleted did not appear in a feature article until 1990, and then only in Peter Lewis’s “Executive Computer” column on the 11th page of the Business section of *The Times*.

**A Taxonomy of Sanitized Recovered Data**

Now we have an explanation for what happened to the data on Disk #70: the disk was formatted with the Windows FORMAT command before it was resold. Indeed, running the Unix `strings(1)` command over the disk’s image file reveals many interesting things about the disk’s previous owner, including the fact that the disk had a copy of IBM AntiVirus Trial Edition installed (Example 15-1) and that the disk was used in some kind

---


of medical application (Example 15-2). Additional investigation revealed that this disk had been used in a computer that belonged to a mail-order pharmacy.

**Example 15-1. The contents of block #854420 from Disk #70**

Displaying block 854420

Notes to Users of IBM AntiVirus version 3.0 build 307...

This file contains important notes for all users of IBM AntiVirus—including a summary of highlights in this release and last-minute changes to the printed documentation. It is divided into these sections: Introduction, Highlights of release 300, Highlights of release 301, Highlights of release 302, Highlights of build 304, Highlights of build 305.

**Example 15-2. The contents of blocks 315782 and 315783 from Disk #70**

Displaying block 315782

*...

RUG TO DRUG INTERACTION @...DPC0...

C DUPLICATION @...CMBO...

S - C...

CMBO...

SUSPENDED LICENSE...

DIABETIC STRIP

DIABETIC STRIPS - B

GENERIC PROD. SUBST-REFILL...

GENERIC PROD. SUBST-NEW & REFILLS...

BENEFICIARY NOT ELIGIBLE PRIME...

DIURPARITION CLAIM...

PAID...

REQUIRES RECEIPT

DUPLICATION CLAIM...

FREE...

INCREASE

Displaying block 315783

09/30/1981 03:00 DUPLICAT.ION

@.SUSPENDED LICENSE...

DIABETIC STRIP

DIABETIC STRIPS - B

GENERIC PROD. SUBST-REFILL...

GENERIC PROD. SUBST-NEW & REFILLS...

BENEFICIARY NOT ELIGIBLE PRIME...

DIURPARITION CLAIM...

PAID...

REQUIRES RECEIPT

DIURPARITION CLAIM...

FREE...

INCREASE

In order to facilitate the discussion of sanitization tools and practices, Shelat and I created a sanitization taxonomy (see Table 15-1). Using this taxonomy to discuss Disk #70, we can say that the disk contained one Level 0 file (COMMAND.COM) and two Level 1 files (IO.SYS and MSDOS.SYS)—both files that were “hidden”—and approximately 508 MB of Level 3 data.
The combination of the taxonomy and the statistical analysis of the operational disks provides a simple answer to the questions posed earlier in this chapter. Although the disks that were purchased contained large amounts of personal information, most of this information consisted of Level 2 and Level 3 files: a casual examination of the disks showed disks that were either formatted or had the user files deleted, leaving only the program files. Most potential recipients of disks sold on the secondary market, lacking tools for accessing Level 2 and Level 3 information, probably never encounter the confidential information on disks that they purchase.

This answer was confirmed, in part, by a series of interviews conducted between December 2003 and October 2004 with the previous owners of 16 of the drives. In some cases (Drives #7, #11, #73, #74, #75, #77, #94, and #134), the organization had a procedure in place for sanitizing the drives, but that procedure was not sufficient to do the job. In other cases (Drives #21 and #44), there was no formal procedure in place. Many owners that were not sophisticated had trusted their reseller to perform the

---

**Table 15-1: A sanitization taxonomy**

<table>
<thead>
<tr>
<th>Level</th>
<th>Type of data</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Regular files</td>
<td>Information contained within the filesystem. Includes filenames, file attributes, and file contents. By definition, there has been no attempt to sanitize the information that is contained within Level 0 files. Level 0 also includes information that is written to the disk as part of any sanitization attempt. For example, if a copy of Windows 95 is installed on a hard drive in an attempt to sanitize the drive, then the files contained within the C:\WINDOWS directory would be considered Level 0 files. No special tools are required to retrieve Level 0 data.</td>
</tr>
<tr>
<td>1</td>
<td>Temporary files</td>
<td>Temporary files, including print spooler files, browser cache files, files for “helper” applications, and files in “recycle bins.” Most users either expect that these files will be deleted automatically in time or are not even aware that these files exist. Note that Level 1 files are a subset of Level 0 files. Experience has shown that it is useful to distinguish this subset, because many naive users will overlook Level 1 files when they are browsing a computer’s hard drive to see if it contains sensitive information. No special tools are required to retrieve Level 1 data, although special training is required so that the operator knows where to look.</td>
</tr>
<tr>
<td>2</td>
<td>Deleted files</td>
<td>When a file is deleted from a filesystem, most operating systems do not overwrite the blocks on the hard disk on which the file is written. Instead, they simply remove the reference to the file from the containing directory. The file’s blocks are then placed on the free list. These files can be recovered using traditional “undelete” tools such as Norton Utilities.</td>
</tr>
<tr>
<td>3</td>
<td>Retained data blocks</td>
<td>Data that can be recovered from a disk but that does not obviously belong to a named file. Level 3 data includes information in slack space, swap space for virtual memory, and Level 2 data that has been partially overwritten so that an entire file cannot be recovered. One common source of Level 3 data is disks that have been formatted with the Windows FORMAT command or the Unix newfs command. Even though these commands give the impression that they overwrite the entire hard drive, in fact they do not, and the vast majority of the information on a formatted disk can be recovered with Level 3 tools. Level 3 data can be recovered using advanced data recovery tools that can “unformat” a disk drive, and using special-purpose forensics tools.</td>
</tr>
<tr>
<td>4</td>
<td>Vendor-hidden data</td>
<td>This level consists of data blocks on the drive that can be accessed using only vendor-specific commands. This level includes the drive’s controlling program, blocks used for bad-block management, and the Host Protected Area (HPA) of modern hard drives.</td>
</tr>
<tr>
<td>5</td>
<td>Overwritten data</td>
<td>Many individuals maintain that information can be recovered from a hard drive even after it is overwritten. Level 5 is reserved for such information.</td>
</tr>
</tbody>
</table>
Sanitization process—a trust that was betrayed (Drives #54, #193, and #205). In the remaining cases of drives that were traced back to their owners, no determination could be made (Drives #6 and #128).

Related Work: Sanitization Standards, Software, and Practices

Confidential information that is stored on a computer system can be protected by numerous mechanisms, including operating system mechanisms, physical isolation of the computer system itself, and even the use of cryptography. The first two of these techniques fail when media is physically discarded: discarded media is, by definition, no longer contained within a protected perimeter. And once a disk or tape drive is removed from the computer on which it was created and is taken elsewhere, other operating systems are free to honor or ignore meta information such as file protection attributes. Only cryptographic measures survive when a disk is discarded. Yet experience has shown that many kinds of confidential information are not stored with encryption. Interestingly,
the risk that confidential information could be released inadvertently on discarded media has been recognized since the 1960s.13

**DoD 5220.22-M**

In the 1970s, each of the armed services adopted their own standards for sanitizing discarded media. These were combined into the unified Department of Defense standard 5220.22-M.14 According to the standard, nonremovable rigid disks may be “cleared” simply by “overwrit[ing] all addressable locations with a single character,” which is generally taken to mean writing an ASCII NUL onto all user-addressable locations. Sanitizing requires that the media be degaussed with a Type I or Type II degausser, that a three-pass sanitization process be implemented, or that the disk be physically destroyed—“disintegrate, incinerate, pulverize, shred, or melt.”

DoD’s three-pass procedure has been the subject of some debate. According to the standard, the procedure is: “Overwrite all locations with a character, its complement, then a random character and verify. THIS METHOD IS NOT APPROVED FOR SANITIZING MEDIA THAT CONTAINS TOP SECRET INFORMATION.” [Emphasis in original.] This is normally implemented with the following algorithm:

1. Seed a cryptographically secure random number generator with a random seed R1.
2. Write the output of the generator to the entire disk.
3. Reseed the random number generator with the same seed R1.
4. Write the complement15 of the generator’s output to the entire disk.
5. Seed the generator with a new random seed R2.
6. Write the generator’s output to the entire disk.
7. Reseed the random number generator with the seed R2.
8. Read the contents of the entire disk, verifying that the contents of each block match the output of the random number generator.

This process is designed to guard against a hard-disk drive that claims to be accepting write commands but that does not actually write data onto the disk.

There has been much debate as to why the standard’s authors have taken such great pains to specify that this standard is not approved for sanitizing media that contains “top secret” information. The implication is that organizations such as the National Security Agency have capabilities for recovering information that has been overwritten using this

---

15 That is, for each output byte \(d\), write the byte \(255-d\).
procedure. An alternative explanation is that the NSA does not possess this capability, and that the NSA doesn’t want the targets of its collection efforts to employ 5220-22-M to sanitize their media!

Peter Gutmann explored the issue of recovering overwritten information in his 1996 USENIX paper, “Secure Deletion of Data from Magnetic and Solid-State Memory.” Gutmann notes that it should be possible to recover information that has been overwritten once or twice—or possibly even more times—depending on the particular recording technology in question. For example, when a 1 is written over a 0, the resulting magnetic field sensed by the read head is going to be different from when a 1 is written over a 1. Gutmann then presents a variety of patterns for overwriting data that are designed to interact with various recording schemes used by different generations of disk drives. In total, 35 different patterns are presented, covering several generations of disk-drive technology.

Surprisingly, a few developers of sanitization tools have adopted all 35 patterns, writing one after the other. Gutmann sneered at this misreading of his paper, and added this postscript to the online version:

“In the time since this paper was published, some people have treated the 35-pass overwrite technique described in it more as a kind of voodoo incantation to banish evil spirits than the result of a technical analysis of drive encoding techniques. As a result, they advocate applying the voodoo to PRML and EPRML drives even though it will have no more effect than a simple scrubbing with random data. In fact, performing the full 35-pass overwrite is pointless for any drive since it targets a blend of scenarios involving all types of (normally used) encoding technology, which covers everything back to 30+-year-old MFM methods (if you don’t understand that statement, reread the paper). If you’re using a drive which uses encoding technology X, you only need to perform the passes specific to X, and you never need to perform all 35 passes. For any modern PRML/EPRML drive, a few passes of random scrubbing is the best you can do. As the paper says, ‘A good scrubbing with random data will do about as well as can be expected.’ This was true in 1996, and is still true now.”

It is readily apparent that overwriting provides sufficient sanitization for the vast majority of computer users. Yet overwriting has not been built into operating systems until very recently, and even current operating systems do not implement it in a uniform or even consistent manner. This creates profound usability problems: not only does the computer user need to understand that the deleted files aren’t actually deleted, but also the user must obtain and use specially written software to properly sanitize his media if he wishes to discard it without risk.

The remainder of this section discusses a variety of techniques that have been developed to allow end users to overwrite data—either selectively or across an entire drive.
CHAPTER FIFTEEN

Add-On Software

Although sanitization has not been built into operating systems until recently, third-party and add-on programs to perform some kind of overwriting have been available for

CRYPTOGRAPHIC APPROACHES

Cryptography provides one of the simplest and possibly the best manner way to handle the problem of data passed. If information is simply encrypted when it is written to a disk and decrypted when it is read back from the disk, then a disk can be sanitized simply by throwing away the cryptographic key in question.

One of the simplest ways to encrypt data on a disk is to use a cryptographic filesystem. Such filesystems can operate on a file-by-file basis, as is the case with Matt Blaze’s Cryptographic File System, and Microsoft’s Encrypted File System, or they can operate on individual blocks, as is the case with PGP:Disk.

Another approach is to use an active hardware device that sits between the computer and the hard disk. Such devices have been made by a variety of manufacturers for both tape drives and disk drives.

The cryptographic approach has also been applied to the problem of email sanitization. The system, developed in the 1990s by a company that was coyly known as Disappearing Ink, turned ordinary HTML email into JavaScript-enabled email that contained a small decryption engine and a block of encrypted data. Each message would have a certain date, D, after which it could not be read. When the engine was run by the receiving mail client, the engine would download the decryption key for day D from the Internet and, if the key was still available, decrypt the message. Disappearing Ink promised that on day D+1, it would delete the key for day D. The Disappearing Ink system couldn’t protect against printouts or malicious pastings of the message text into an ordinary word processor, but it did provide a system whereby cooperating email partners could assure that all copies of messages in their control (including copies on backup tapes) would be deleted after a certain date.

All of the schemes discussed in this section rely on the ability to delete a key properly. Deleting keys may be difficult in practice, a problem discussed by both Crescenzo et al. and Gutmann.

---


Add-On Software

Although sanitization has not been built into operating systems until recently, third-party and add-on programs to perform some kind of overwriting have been available for
personal computers for more than two decades. Programs have emerged to target all manner of data on a hard drive:

**Existing files (Level 0 and Level 1 data)**

Programs such as Norton Disk Doctor, PGP, and the Linux shred command have the ability to delete specific files securely. These features typically are implemented by opening the disk file through the operating system, seeking to the beginning of the file, and repeatedly issuing the `write()` system call.

**Free blocks (Level 2 and Level 3 data)**

A program can sanitize the contents of blocks on the free list using two approaches. The easiest technique is for the program to open a file for writing and write to the file until the file fills the entire disk. The operating system necessarily allocates most or all of the disk’s available free space to the new file. A problem with this approach is that the computer necessarily spends some time with no free disk space available; this can impact normal operations. A second problem is that it does not overwrite files that are small enough to be stored in the Windows MFT or in the Unix inode. Nevertheless, Microsoft’s `CIPHER.EXE`, distributed with the Windows operating system, uses this technique to sanitize blocks on the free list.

A second technique is to have a program that understands the computer’s filesystem open the raw disk device, interrogate the list of free blocks, and systematically overwrite the contents of each block on the free list. The advantage of this technique is that it does not compromise the computer’s “free space,” but this technique requires much closer integration with the computer’s operating system. This approach should not be employed on a disk that is actually in use by the filesystem.

**Slack space (Level 3 data)**

Although disk drives typically write data in 512-byte blocks, many operating systems allocate storage in clusters that consist of two, four, eight, or more blocks. If a file consists of only a few bytes, it is not uncommon for an operating system to write only the first block of the cluster and leave the rest with their previous contents: this is both a performance optimization, because it takes less time to write one block than four, and a memory optimization, because the data written to the disk needs to come from somewhere. The unused blocks at the end of the cluster are frequently referred to as *slack space*. They can contain the contents of older files that have long since been deleted.

**The entire disk (Level 1, 2, and 3)**

A number of programs simply overwrite the contents of the entire disk. These programs typically are run from bootable floppy disks or CD-ROMs and are typified by DBAN (Dirk’s Boot and Nuke).

Add-on programs are not without their risks: a poorly written program that reads and writes directly to the disk without going through the operating system risks corrupting the entire volume. Programs that go through the operating system run the risk that there may be a mismatch between the primitives that the operating system offers to programs
and the way those abstractions are implemented on the actual disk surface. For example, a user who types `pgp -w filename` to erase the contents of `filename` on a journaling filesystem may be surprised to learn that data that “overwrites” may actually be written to another location of the disk. That’s because journaling systems write modifications to an expanding journal, instead of modifying data that is stored in place, to protect the filesystem against inadvertent corruption that might occur if the computer fails during a disk write.

Another risk of add-on programs is that they may not actually perform the way that they claim to. Although users would certainly notice and complain about a program that was advertised to sanitize slack space but that actually overwrote Level 0 files, a program that overwrote only the first few bytes of a file being “wiped” would not be so easy to detect. Indeed, for several years, the Linux `wipe` command did just that.

But the biggest risk of relying on add-on programs, quite simply, is that users do not know that they exist. Driving this point home, a proactive white paper written by Guidance Software, evaluating the Windows `CIPHER.EXE` program, concludes:

**Results:** All unallocated space was filled with random values (which greatly affected file compression in the evidence file); however, the cipher tool affected only the unallocated clusters and a very small portion of the MFT [Master File Table]; 10–15 records were overwritten in the MFT, and the majority of the records marked for deletion went untouched). The utility does not affect other items of evidentiary interest on the typical NTFS partition, such as: file slack, registry files, the pagefile, and file shortcuts.

In terms of its anticipated end-user adoption, the cipher feature is a burdensome command-line utility that is difficult to find and operate. Notably, the cipher function is available on the Professional version, but not included in the Home version of XP and Windows 2000. Despite some speculation, the function is not set by default or even selected for repeated execution on an ongoing basis. The cipher must be executed from a command line each time the user wants to employ it. There is very little documentation supporting this feature, which is largely intended for programmers and system administrators for use in limited circumstances.16

Once again, usability is compromised because of an underlying failure in the operating system: not only does the user need to know that he must obtain a special sanitization program, but he also needs to be sure that the program works as advertised. Unlike a word processor or a spreadsheet, it is remarkably difficult for a user to test a sanitization program to determine if it is functioning properly.

Operating System Modifications

An alternative to third-party sanitization tools is to have sanitization technology embedded directly into the computer’s applications and operating system. Arguably this is where they belong, on the grounds of both usability and correctness. Certainly usability is enhanced when the operating system provides for proper sanitization of media that potentially has confidential information. Correctness is also furthered, because an implementation of sanitization within the operating system should take into account any implementation peculiarities (such as journaling filesystems) that might be hidden to the application program but that are relevant to the sanitization process.

For example, version 0.4 of the Linux ext2fs filesystem introduced the concept of the per-file scans attribute. Set with the chattr command, any file with the s attribute set was overwritten with ASCII NULs when the file was unlinked. Remy Card’s implementation was quite simple—only four places in the ext2fs source code needed to be modified to handle the secure deletion attribution.17

Card’s implementation was certainly open to criticism: blocks were overwritten only once, not three times or more. And because the write was only scheduled, there was a chance that a block queued for being overwritten might not actually get overwritten at all—the system might crash, or the blocks might be reallocated to another file and the confidential information might end up in the slack space at the end of a file. In any event, the code for “secure deletion” was removed from ext2fs in the Linux 2.2 kernel, apparently for performance reasons.

It’s interesting to note that even though the Linux 2.2 kernel dropped the actual secure deletion, the s attribute can still be set by the chattr command. Doing so, however, no longer does anything! Adding to the confusion is the fact that the chattr command’s documentation still discusses the s, c (automatic compression), and u (allow undeletion) attributes, but notes at the bottom:

BUGS AND LIMITATIONS
As of Linux 2.2, the ‘c’, ‘s’, and ‘u’ attribute are not honored by the kernel filesystem code. These attributes will be implemented in a future ext2 fs version.

Here, then, is another example of the disconnect between usability and security caused by poor programming and visibility, rather than an inherent disconnect between the two. Does the phrase “not honored by the kernel filesystem code” mean that secure deletion happens somewhere else, or that it doesn’t happen at all? The only way for an expert user to know is to inspect both the source code and a running system. The only way for a novice user to know is to ask an expert.

Others have reimplemented secure deletion for Linux, but these modifications have not been incorporated into the mainstream Linux kernel. Bauer and Priyantha\(^\text{18}\) describe a modification to the Linux operating system to support secure deletion using the very same attribute bits developed by Card. Instead of implementing secure deletion directly in the `unlink()` system call, the implementation uses a scheduled kernel process. Unfortunately, this code was not integrated into the mainstream Linux kernel, and it can no longer be integrated because the Linux kernel has continued to evolve.

Following the publication of the “Remembrance of Data Passed” paper, Apple Computer added a Secure Empty Trash function to the Finder component of its Mac OS. According to interviews conducted with Apple’s security group on January 12, 2003, the security group had long wanted to put a secure delete-file function in the operating system’s interface: such efforts had been deemed by Apple’s management to be not a priority until the “Remembrance” paper was published.

Like the Windows Recycle Bin, the Macintosh Trash is actually a special directory that is managed by the graphical file manager (which Apple calls the Finger). Dragging a file to the Trash moves the file to this special directory. Emptying the Trash causes all of the files in the directory to be unlinked with the `unlink()` system call. Choosing the Secure Empty Trash feature, added by Apple in Mac OS 10.3, causes the contents of the files to be overwritten before they are unlinked.

The Mac OS 10.3 interface is subtle in the way that it informs the user of the difference between Empty Trash and Secure Empty Trash. Choosing either option from the Finder menu causes an alert panel asking for confirmation to appear. The wording on the Empty Trash panel states:

\[\text{Are you sure you want to remove the items in the Trash permanently?} \]
\[\text{You cannot undo this action.}\]

In contrast, the wording on the Secure Empty Trash panel states:

\[\text{Are you sure you want to remove the items in the Trash permanently using Secure Empty Trash?}\]
\[\text{If you choose Secure Empty Trash, you cannot recover the files.}\]

What is the difference between an operation that cannot be undone and one that cannot be recovered from? The answer is found in the Finder’s Help system when one types in Secure Empty Trash.

Even after emptying the trash, deleted files may still be recovered using special data-recovery software. To delete files so that they cannot be recovered, choose Finder → Secure Empty Trash. Files deleted in this way are overwritten completely by meaningless data. This may take some time, depending on the size of the file. You may want to use Secure Empty Trash if you sell or give away your computer.

Although this author finds it personally rewarding that a paper published in January 2003 would result in a significant modification to an operating system used by tens of millions of people in less than a year’s time, there is much to critique in Apple’s initial implementation of Secure Empty Trash from a usability perspective:

• Because Secure Empty Trash is such a slow procedure, it seems that it would be advantageous to be able to specify files to be erased securely on a file-by-file basis. However, there is no way to make such distinctions.

• Likewise, there is no way to securely erase a specific file but leave the other files in the Trash untouched. This poses an inconvenience to users who habitually keep hundreds or thousands of files in their Trash directories and who wish to securely delete a single file from time to time. (There is a rather straightforward, albeit annoying, workaround for this problem: simply move all of the files that are in the Trash into a second, temporary directory, leaving behind the files to be sanitized. Choose Secure Empty Trash. Finally, move the files from the temporary directory back into the Trash.)

• If the user inadvertently chooses Empty Trash... rather than Secure Empty Trash, there is no way to go back and securely overwrite the disk blocks once the files have been unlinked from the Trash directory.

• Implementing Secure Empty Trash in the Finder, rather than in the operating system’s kernel, means that there is no way to securely delete files that are deleted by programs other than the Finder (e.g., using rm or Emacs). Likewise, the contents of temporary files that are created and then deleted will not be sanitized.

• There is no way to remove from a disk the information that was contained in files that have been “overwritten” using the Save As... feature of many document-based applications. That’s because Save As is implemented by a four-step process that involves saving the requested file to a temporary file, renaming the file to be overwritten to a temporary name, renaming the new file to the final destination filename, and finally unlinking the file being “overwritten.” In this way, no data is lost if the computer fails during the procedure. Unfortunately, the interface promises that the new file will “overwrite” the original file, as demonstrated in Figure 15-1.

• Finally, by the time you are thinking of selling or giving away your computer, it’s too late! The files have already been unlinked and can no longer be “securely emptied!”

Apple’s implementation of Secure Empty Trash has the look of a quick hack—it adds functionality without properly integrating that functionality into the user experience. Indeed, the very user interface is inconsistent: the menu item Empty Trash... appears
with a trailing ellipsis, indicating that choosing the command will bring up another
window, while the menu item Secure Empty Trash appears without an ellipsis.
Meanwhile, the context-sensitive menu on the Trash Can icon was not modified between
Mac OS 10.2 and Mac OS 10.3 to reflect the new functionality: a user who only empties
the trash by using the context-sensitive menu might miss the new functionality in the
operating system. Both of these user interface errors are shown in Figure 15-2.

FIGURE 15-1. This pop-up window is displayed when an existing file is specified using the Save As? command; Mac
OS 10.3 promises that the new file will "overwrite [the] current contents" of the file being replaced; in fact, Mac OS does
not overwrite the file: it saves the new file to a temporary location, renames the original file to a temporary filename,
renames the new file to the original filename, and then uses the unlink() system call to delete the original file
with a trailing ellipsis, indicating that choosing the command will bring up another
window, while the menu item Secure Empty Trash appears without an ellipsis.

FIGURE 15-2. Modifications to the Mac OS 10.3 user interface to support the Secure Empty Trash command were
inconsistent and applied haphazardly.

While Apple should be commended for making an attempt to address the issue of file-by-
file sanitization, the company has not lived up to its reputation of interface
thoughtfulness, consistency, and user experience. Apple needs to rethink its
implementation of file sanitization and come up with a new interface that both is easier
to use and promotes improved security. Thoughts on such an interface are presented in
the next section.
Moving Forward: A Plan for Clean Computing

Although the need for proper sanitization of magnetic recording media has long been recognized as a serious issue for computer security practitioners, the problem has traditionally been addressed through the use of add-on software or physical destruction of the media itself. Only recently has the question of sanitization been addressed by computer operating system vendors themselves, and in the cases that we have considered, both Microsoft and Apple have addressed it poorly.

Some researchers have told me that operating system developers did not deploy sanitizing file deletion in the 1970s and 1980s so that accidentally deleted files could be recovered using special tools. I have looked and have found no evidence to support this claim. Indeed, had recoverability of accidentally deleted data been a goal, companies like Microsoft and Apple would have distributed such tools themselves—either as part of their operating system offerings or as aftermarket additions. (DOS 5.0 included an UNFORMAT command-line utility that could, in some cases, recover a volume that had been formatted inadvertently. But this looks like an after-the-fact attempt to make use of a DOS 2.0 idiosyncrasy, rather than an intentional design attribute that was put into DOS 2.0 and realized only when DOS 5.0 finally shipped.)

Instead, it seems that the lack of a sanitizing delete-file is a historical accident. The first multiuser computer systems did not need a sanitizing delete because of the ways in which they were operated. The developers of the Compatible Time Sharing System (CTSS) at MIT in the 1960s did not consider the problem of sanitizing disk blocks after deleting files because the computer system frequently ran with disks that were full or nearly full: blocks that were freed were quickly overwritten with new data.\(^\text{19}\) The CTSS disk drives were rented from IBM and, thus, were never offered on the secondary market. Indeed, the real data security problem was not that data on a disk returned for service might be sent accidentally to another IBM customer—the real problem was trying to keep data on the disks that were having head crashes every few days! In any event, while there was a general belief that CTSS should prevent one user from crashing a program being run by another user, overall the system did not have strong internal disclosure controls: the developers did not believe that CTSS was secure enough to store sensitive information.

Internal security between users was a design goal of the Multics operating system. But Multics did not have files as we know them today. Instead, it had named memory segments that could be created, placed in a hierarchical filesystem, and extended as needed. When a process created a new segment, the disk blocks for that segment were cleared (that is, overwritten with NULs) \textit{when the segment was allocated}. Thus, there was no need to overwrite the blocks when the segment was erased. Clearing on allocation doesn’t solve the data remanence problem, but that problem applies to both deleted and

undeleted information. If a Multics disk was decommissioned, the system’s operators would have understood the importance of erasing all of the information from the device.

Unix was developed in a research environment in which security was not a priority: when Unix first transitioned into the commercial world, it ran on large systems that were run by trained system operators who, presumably, were aware of the sanitization issue. Although Unix provided no tools for sanitizing disks, the \texttt{dd} command did an excellent job, as previously noted.

In the world of PC operating systems, an overwriting delete would have caused significant performance degradation (similar to Mac OS 10.3’s Secure File Delete) on any operating system that did not have asynchronous access to the filesystem and disk device. Windows did not have such capabilities until 32-bit clean disk drivers were available under Windows 95 and NT. Apple did not have such capabilities until it migrated to Mac OS X.

Today, matters have changed. Unlike the systems of the 1960s and 1970s, most of today’s desktop computers spend most of their time idle. Likewise, many—perhaps most—of today’s desktop computers are not managed by professional IT staffs that are aware of the sanitization problem. What is needed, then, is some straightforward way to add a sanitizing delete-file function.

Although a simple approach would be to resurrect Bauer and Priyantha’s Linux implementation, I believe that this is only half of the story. Although it’s good for overwriting to happen asynchronously with other activities, bringing truth to the phrase “\texttt{rm} is forever” is not a particularly user-friendly way to move forward. Many usability experts have noted that humans make frequent mistakes; simply adding a warning box saying something to the effect that “\texttt{rm} deletes all of your files” not a particularly strong barrier to improper use.

An alternative construct would be to remove the Empty Trash and Secure Empty Trash commands in the Macintosh user interface and replace them with a new command: Shred Trash. Executing this command would perform much the same function as the Secure Empty Trash, except that the overwriting would be performed asynchronously. There are two advantages to this approach:

• The term \textit{Shredder} is superior to Secure Empty Trash because most people know what a paper shredder does; most people do not know what it means to securely empty trash. Thus, less initial user education would be required.

• Because the shredding would be performed asynchronously, there would be no perceived penalty for using the feature.

The second change required is to allow the user to create rules that specify conditions under which the contents of the Trash directory are shredded automatically. Typical rules might include:
• Shred any file that has been in the Trash more than 30 days.
• Shred all files in the Trash when the user clicks the “Shred all files now” button.
• Shred all files in the trash at 7:00 a.m. every day.
• If a file is selected and the user chooses the Shred command from the File menu:
  — If the file is not in the Trash directory, move the file to the Trash directory and
    schedule for it to be shredded in 5 minutes.
  — If the file is in the Trash directory, shred it immediately.

These rules give users a chance to change their minds (as recommended by Cooper\(^\text{20}\)), but nevertheless provide for the possibility of immediate shredding, should such an action be necessary. Figure 15-3 shows a hypothetical user interface to implement this rule set.

![Hypothetical design of a Shredder rules interface](Image)

The third change required is to change the semantics of the POSIX `unlink()` system call, so that calling `unlink()` on a file has the effect of moving that file to the user’s Trash, instead of actually releasing the blocks. Such a move, while drastic, would overcome the problems that arise when a file is deleted by an application other than the Finder. This approach would also handle the problem discussed earlier with the Save As command.

A naïve implementation of this third recommendation would degrade system performance dramatically. This is because many small files are constantly being created and deleted on a typical computer system. But temporary files still need to be sanitized. Moving them to a shredder—perhaps with a tag that marks them for “immediate” shredding—assures that this will happen. The files can then be shredded in the background when the computer is not otherwise in use. For servers and computers that are constantly in use, the computer’s filesystem can simply use disk blocks in the shredder and marked for shredding as a convenient source of new blocks when they are needed for writing out new files. This would effectively sanitize the blocks in the same manner as blocks were sanitized on CTSS back in the 1960s.

\(^{20}\) Alan Cooper, “The Inmates are Running The Asylum: Why High-Tech Products Drive Us Crazy and How to Restore the Sanity” (Sams Publishing: 1999).
Acknowledgments

The credit card numbers in the forensics study were found using a program written by Abhi Shelat. David Clark and Ron Rivest provided much support, suggestions, and guidance on the original forensics study, as did Brian Carrier, Peter Gutmann, Rich Mahn, Eric Thompson, and Wietse Venema. Matthew Geiger and Lorrie Cranor provided useful comments on an earlier draft of this chapter.

About the Author

Simson L. Garfinkel has been working in the field of computer security since 1986. He recently completed his dissertation on patterns and principles for aligning usability and security at MIT and is now a postdoctoral fellow at Harvard University’s Center for Research on Computation and Society.