Breaking Forensics Software: Weaknesses in Critical Evidence Collection

Tim Newsham - <tim[at]isecpartners[dot]com>
Chris Palmer - <chris[at]isecpartners[dot]com>
Alex Stamos - <alex[at]isecpartners[dot]com>

iSEC Partners, Inc
115 Sansome Street, Suite 1005
San Francisco, CA 94104
http://www.isecpartners.com

July 1, 2007

Abstract

This article presents specific vulnerabilities in common forensics tools that were not previously known to the public. It discusses security analysis techniques for finding vulnerabilities in forensic software, and suggests additional security-specific acceptance criteria for consumers of these products and their forensic output. Traditional testing of forensics software has focused on robustness against data hiding techniques and accurate reproduction of evidence. This article argues that more security focused testing, such as that performed against security-sensitive commercial software, is warranted when dealing with such critical products.

Note: *Due to the deadline for submitting presentation materials to Black Hat and the ongoing nature of our conversation with Guidance Software, we are unable to present in this revision of the paper all the details of the defects in EnCase that we found. By the time you read this, this version of the paper will be out of date and the canonical version may have the defect details. Please see https://www.isecpartners.com/blackhat to find the most recent version of this paper as well as several of the tools we created during our research.*

1 Introduction

This article presents specific vulnerabilities in common forensics tools that were not previously known to the public, discusses techniques for finding vulnerabilities in forensic software, and recommends additional security-specific acceptance criteria buyers should apply. The primary contribution of this work is to take an adversarial look at forensics software and apply fuzzing and vulnerability assessment common to analysis of other products, such as operating systems or office suites, to forensic software.

Two popular software packages for performing forensic investigations on computer evidence, Guidance EnCase and Brian Carrier's The Sleuth Kit (TSK), are vulnerable to attack. The most common problems are "crashers", that is, damaged data files, storage volumes, and filesystems which cause the software to crash before the forensic analyst can interpret the evidence.

We performed some random and targeted fault injection testing against the two products and uncovered several bugs, including data hiding, crashes that result in denial of service, and infinite loops that cause the program to become unresponsive.

2 Prior Art

While blind fuzzing and targeted fault injection are not new techniques, there has not been much (public) research into the relatively small niche of forensics software.

There is not much on EnCase or TSK in the Common Vulnerabilities and Exposures database[1], for example. Searching on "encase" in the CVE search engine returns only one result (at the time of writing, 28 June 2007), http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2005-1578 ("EnCase Forensic Edition 4.18a does not support Device Configuration Overlays (DCO), which allows attackers to hide information without detection"). Searching CVE for "sleuth" (http://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=sleuth) and "sleuthkit" (http://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=sleuthkit) return 0 results, and a searching on "sleuth kit" returns results containing the word "kit" but not "sleuth".

3 Classes of Attacks Against Forensic Software

Forensic software is especially difficult to secure, yet must be especially robust against attack. It must acquire data from any type of device, in any format; it must parse, render, and search as many data formats as possible; and it must do all this with acceptable performance without sacrificing correctness or accuracy — in the presence both of malicious tampering and of accidental faults in the evidence.

In a forensic investigation, denial of service (DoS) vulnerabilities are no longer merely annoyances, but can impede the investigation, making evidence difficult or impossible to examine. "Crasher" bugs (often the result of the overflow of buffers on the stack or heap) are sometimes exploitable, leading to a situation in which a maliciously crafted evidence file might allow an attacker to frustrate analysis or possibly execute code on the investigator's workstation[2]. This may compromise the integrity of investigations performed on the machine (or allow a lawyer to argue the point).

3.1 Data Hiding

The purpose of forensic software is to discover and analyze evidence stored on digital media. If the software fails to detect information on the medium for some reason, an attacker could exploit the weakness to hide evidence. For example, a data acquisition tool is vulnerable to a data hiding attack if it fails to acquire the host protected area of a hard disk.

Note that we do not include cryptography as a data hiding attack because, while it may be uninterpretable, encrypted information is still visible.

Another type of data hiding attack is the "attack by tedium": where there exist asymmetries such that an attacker can obfuscate data more easily than the forensic investigator can unobfuscate it, the effect can

[1] http://cve.mitre.org/

[2] To be clear, we have not found any code execution vulnerabilities.

be similar to a steganographic attack. This method of attack is relatively weak since it relies on the usability affordances of a particular forensic software kit. A determined investigator could defeat the attack by using multiple software kits or scripting the toolkit for automated evidence analysis, for example.

3.2 Code Execution, Evidence Corruption

Code execution vulnerabilities result from particular implementation flaws such as stack and heap overflows. Programming errors in native code may allow an attacker to specify data that overwrite control flow information in the program and provide new code of the attacker's choice to be executed in the context of the vulnerable process (including accessing files and the network).

If an attacker succeeds in executing arbitrary code on the forensic workstation, such as by exploiting a buffer overflow in one of the data format parsers or renderers, he can — among a vast array of other things — cause the forensic image to become corrupted, hiding or destroying evidence. A subtle attack would be to cause the forensic toolkit not to alter the forensic image, but simply to ignore the evidence the attacker wishes to hide. There would be no obvious tip-off that an attack has occured, such as if the checksums of the evidence files changed, yet the toolkit would be instructed by the attacker never to reveal the incriminating evidence.

All the usual "mundane" attacks are of course also possible, such as planting spyware or other malware on the forensic workstation.

Bugs that allow an attacker to overwrite memory, even without arbitrary code execution, could also allow an attacker to spoil or hide evidence[3].

3.3 Denial of Service, Blocking Analysis

Denial of service vulnerabilities (when the program crashes or hangs) frustrate forensic analysis. If an attacker hides the incriminating evidence in a file that crashes the forensic software but does not crash the attacker's own file viewer (and we observed many instances of this type of bug), the analyst must perform extra work to discover the evidence. If the analyst is diligent, the attack is a mere service downgrade (the analyst loses the rich search and parse functionality of their forensic tool). If the analyst is overworked, on a tight deadline, or lazy, they might miss important evidence.

This might seem minor, but note that the sophisticated searching, parsing, key recovery, and file carving features are the entire reason forensic software toolkits exist as a distinct class of software. Forensic analysis can of course be done with nothing but a write-blocker and the standard Unix tools, but few professional analysts would be satisfied with such a limited toolbox. Until forensic toolkits become more robust, analysts may be in the dark without knowing it.

4 Techniques Used to Find Vulnerabilities

Because one of the most obvious attack surfaces on forensic software is the filesystem metadata parsing code and the rich data file parsing and rendering code, we attacked it by generating fuzzed filesystems and

[3]We do not know of any such defects.

data files. We also made attempts to hide data by making disks with many partitions, and deeply-nested archive files (TAR, ZIP, etc.).

4.1 Fuzzing Data Formats

We did not need sophisticated fuzzing to discover many of the vulnerabilities. We used simple random fuzzing with no special provision for the particulars of any given data format. The fuzzer is instantiated with a random seed and provides a set of mutator functions; when fed source data, one of the mutators is chosen and invoked on the source. iSEC has a library of mutators that can

- randomly choose a substring (of random length) of the source and replace it with a random string of the same size;
- replace a random number of single bytes in the source with random bytes;
- increment or decrement a random number of single bytes in the source;
- replace a randomly-selected NUL byte or sequence of two NULs in the source with a given value of the same size;
- replace the entire source with a random string of the same size;
- overwrite 32-bit values in the source with some other given 32-bit value, advancing the replacement position on successive calls;
- delete or insert a randomly-selected substring (of random size) from the source; and
- randomly choose any two mutators and perform both mutations on the source.

When fuzzing disk and volume images we did not use the mutators that change the size of the object, since filesystems are based on fixed-size blocks with many structures expected to be aligned at particular offsets. Perturbing a filesystem too drastically tends to cause implementations to reject it completely, with no chance of bug discovery.

We fuzzed otherwise normal data files (JPEGs, PDFs, MS Word documents, etc.), volumes and partitions, and disk device images. Each successive fuzzing target also includes the previous targets, but also fuzzes more data: fuzzing partitions affects filesystem metadata and file data; and fuzzing disks affects partition metadata, filesystem metadata, and file data). We then performed appropriate tasks for the object with the forensic application: viewing and acquiring disks and volumes, viewing and searching files, etc.

4.2 Manual, Targeted Manipulation of Data Formats

We also performed targeted, manual mangling of data formats, such as by creating directory loops in filesystems, creating loops in MBR partition tables, creating disk images with very many partitions, tweaking data objects in JPEG files that influence memory management, and so on.

MBR partition tables. We wrote code to identify all of the MBR records and performed fuzzing on only those blocks. The fuzzing was again simplistic. The reason we did this was to increase the amount of mutations in each test case (since we can only test one disk at a time, whereas we can test many

filesystems at a time, for example) without causing other issues (i.e. in the filesystem code) that might mask findings.

Directory loops. We manually edited ext2fs and NTFS filesystems so that a directory became a child subdirectory in its own directory, and then analyzed the resulting image.

Long file names. We generated filesystems with very long file names. These were created both inside a single directory and in a chain of deeply nested directories.

Large directories. We generated filesystems with a directory containing large numbers of files. These were filled with files having really short names, medium length names and long filenames.

Deeply nested directories. We generated filesystems with deeply nested directories. The directory names were very short and very long.

5 Defects Found in The Sleuth Kit

Brian Carrier's The Sleuth Kit (TSK) is a tool-oriented forensic software suite in the Unix tradition (although it does run on Windows in addition to Linux, Mac OS X, and other Unix variants). Individual programs with command-line interfaces each do one task — extracting inode metadata from a disk image, copying data referenced by an inode to a file — and to work together by reading and writing streams of text. It is implemented in C and tied together by a web interface implemented as Perl CGIs (Autopsy).

In contrast to EnCase, TSK almost completely relegates evidence display to third-party software. TSK consists of 23 separate single-purpose programs, but we found vulnerabilities in only a few:

fls lists the file and directory names in a give filesystem image, including deleted files;

fsstat displays the metadata for the filesystem, including inode numbers and mount times;

icat copies files in the disk image by inode number (Unix filesystems) or MFT entry number (NTFS), as discovered and chosen by the investigator using e.g. the istat tool; and

istat displays the metadata stored in an inode or MFT entry;

Simple fuzzing raised several issues, and the inherent programmability of Unix-type tools allowed us to easily isolate the particular spot in damaged files that was causing the problem. In general it appears that the implementation is sufficiently careful about keeping buffer writes in bounds, but it places too much trust in the data from the disk image when reading from buffers. Most issues involve out-of-bounds reads that can lead to incorrect data, or crashes. There were also some issues that may lead to denial of service.

5.1 Data Dereferenced After Free

A crash can occur when processing a corrupted ext2fs image because the error processing code dereferences data after it frees it. In ext2fs.c:

```
1230      for (n = 0; length > 0 && n < inode->direct_count; n++) {
1231          read_b = ext2fs_file_walk_direct(fs, buf, length,
1232              inode->direct_addr[n], flags, action, ptr);
1233
1234          if (read_b == -1) {
```

```
1235                free(buf);
1236                data_buf_free(buf[0]);
1237                return 1;
1238            }
```

If *read_b* is -1 (line 1234) then *buf* is freed (line 1235) before data in *buf*[0] is freed (line 1236). This leads to a crash on some systems.

5.1.1 Reproduction

```
$ patch.py NtfsPart.dsk Bad.dsk 7616332 \x01
$ icat Bad.dsk 56-128-3
```

5.2 Corrupted NTFS image Causes icat to Run Indefinitely

icat runs virtually forever when run on some altered NTFS images. It appears that a 64-bit value was read off the disk and used as a byte count. We observed the following in *gdb*:

```
#9  0x080892ef in ntfs_data_walk (ntfs=0x80ee0c0, inum=56, fs_data=0x80f0400,
    flags=0, action=0x80a45d0 <icat_action>, ptr=0x0) at ntfs.c:1639
1639                retval = action(fs, addr, buf, bufsize, myflags, ptr);
(gdb) p/x fsize
\$3 = 0xffffffffff8ecd42
(gdb) p/x fs_data->size
\$4 = 0x9342
(gdb) p/x fs_data->runlen
\$5 = 0xffffff0600009200
```

5.2.1 Reproduction

```
$ patch.py NtfsPart.dsk Bad.dsk 7616332 \x01
$ icat Bad.dsk 56-128-3
```

5.3 NTFS Image Causes icat to Crash

icat crashes while processing a file on a corrupted NTFS filesystem image. The crash occurs when dereferencing *fs_data_run* at line 1570 of ntfs.c:

```
1543        /* cycle through the number of runs we have */
1544        while (fs_data_run) {
1545
1546            /* We may get a FILLER entry at the beginning of the run
1547             * if we are processing a non-base file record because
1548             * this \$DATA attribute could not be the first in the bigger
1549             * attribute. Therefore, do not error if it starts at 0
1550             */
1551            if (fs_data_run->flags & FS_DATA_FILLER) {
      [... code elided ...]
1564                }
1565                else {
1566                    fs_data_run = fs_data_run->next;
1567                }
1568            }
1569
1570            addr = fs_data_run->addr;
```

The check at line 1544 ensures that *fs_data_run* is non-NULL, however line 1566 updates the variable without returning to the check (with a continue) or performing the check again. This leads to a NULL dereference at line 1570.

5.3.1 Reproduction

```
$ patch.py NtfsPart.dsk Bad.dsk 7567157 AA
$ icat Bad.dsk 8-128-1
```

5.4 NTFS Image Causes fls to Crash (1)

fls crashes while listing files from a corrupted image. The crash occurs when copying data in *fs_data_put_str*. The calling code is in ntfs.c:

```
1778          /* Add this resident stream to the fs_inode->attr list */
1779          fs_inode->attr =
1780          fs_data_put_str(fs_inode->attr, name, type,
1781          getu16(fs->endian, attr->id),
1782          (void *) ((uintptr_t) attr +
1783              getu16(fs->endian,
1784              attr->c.r.soff)), getu32(fs->endian,
1785              attr->c.r.ssize));
```

The read buffer is *attr* plus an arbitrary 16-bit offset and the read length is an arbitrary 32-bit length. No bounds checking is performed to ensure that the buffer contains as many bytes as are requested.

5.4.1 Reproduction

```
$ patch.py NtfsPart.dsk Bad.dsk 7652939 \x01
$ fls -rlp Bad.dsk
```

5.5 NTFS Image Causes fls to Crash (2)

fls crashes while listing files from a corrupted image. The crash occurs at line 208 of ntfs_dent.c while dereferencing *idxe*:

```
205          /* perform some sanity checks on index buffer head
206           * and advance by 4-bytes if invalid
207           */
208          if ((getu48(fs->endian, idxe->file_ref) > fs->last_inum) ||
209              (getu48(fs->endian, idxe->file_ref) < fs->first_inum) ||
210              (getu16(fs->endian, idxe->idxlen) <= getu16(fs->endian,
211                      idxe->strlen))
212              || (getu16(fs->endian, idxe->idxlen) % 4)
213              || (getu16(fs->endian, idxe->idxlen) > size)) {
214              idxe = (ntfs_idxentry *) ((uintptr_t) idxe + 4);
215              continue;
216          }
```

This code is executed in a loop while walking a table and advancing *idxe*. The variable is initially in range, but is eventually moved out of range because the caller passes in a large size (line 735) which is an arbitrary 32-bit integer taken from the filesystem image. No bounds checking is performed to ensure that the value is in range.

```
734      retval = ntfs_dent_idxentry(ntfs, dinfo, list_seen, idxe,
735          getu32(fs->endian,
736              idxelist->buf_off) -
737          getu32(fs->endian, idxelist->begin_off),
738          getu32(fs->endian,
739              idxelist->end_off) -
740          getu32(fs->endian, idxelist->begin_off), flags, action, ptr);
```

5.5.1 Reproduction

```
$ patch.py NtfsPart.dsk Bad.dsk 7653298 d
$ fls -rlp Bad.dsk
```

5.6 NTFS Image Causes fsstat to Crash

fstat crashes while processing a corrupted filesystem image. The crash happens when dereferencing *sid* in line 2714 of ntfs.c:

```
2703      unsigned int owner_offset =
2704          getu32(fs->endian, sds->self_rel_sec_desc.owner);
2705      ntfs_sid *sid =
2706          (ntfs_sid *) ((uint8_t *) & sds->self_rel_sec_desc +
                          owner_offset);
2707
2708      // "1-"
2709      sid_str_len += 2;
2710      //tsk_fprintf(stderr, "Revision: %i\n", sid->revision);
2711
2712      // This check helps not process invalid data, which was noticed
             while testing
2713      // a failing harddrive
2714      if (sid->revision == 1) {
```

This variable is computed with an arbitrary 32-bit offset (line 2704) from an existing buffer (line 2706) without any bounds checking.

5.6.1 Reproduction

```
$ patch.py NtfsPart.dsk Bad.dsk 10667177 \x9b
$ fstat Bad.dsk
```

5.7 NTFS Image Causes fsstat to Crash

fstat crashes while processing a corrupted filesystem image. The crash happens while copying data from the *sds* variable (line 2831) in ntfs.c. Care is taken to ensure that no out of bounds reads occur with checks at line 2817. However, these checks are based on the offset computed in terms of 32-bit values (line 2809) not in terms of bytes. The check assumes that the *current_offset* is actually a byte count, which would be four times larger.

```
2808      while (total_bytes_processed < sds_buffer->size) {
2809          current_offset =
2810              (uintptr_t *) sds - (uintptr_t *) sds_buffer->buffer;
2811
2812          offset = getu32(fs->endian, sds->ent_size);
2813          if (offset % 16) {
2814              offset = ((offset / 16) + 1) * 16;
2815          }
2816
2817          if ((offset != 0) && (offset < (sds_buffer->size
                                              current_offset))
2818              && (getu64(fs->endian, sds->file_off) <
                                       sds_buffer->size)) {
2819
2820              NTFS_SDS_ENTRY *sds_entry;
2821
2822              if ((sds_entry =
2823                       (NTFS_SDS_ENTRY *)
                         mymalloc(sizeof(NTFS_SDS_ENTRY))) ==
2824                  NULL) {
2825                  return 1;
```

```
2826            }
2827            if ((sds_entry->data = (uint8_t *)
                                        mymalloc(offset)) == NULL) {
2828                free(sds_entry);
2829                return 1;
2830            }
2831            memcpy(sds_entry->data, sds, offset);
```

5.7.1 Reproduction

```
$ patch.py NtfsPart.dsk Bad.dsk 10666450 \x01
$ fsstat Bad.dsk
```

6 Defects Found in Guidance EnCase

EnCase from Guidance Software is a very different beast from TSK. It is Windows-only (with a Linux remote device acquisition component), features a complex GUI, and incorporates features for browsing, searching and displaying devices, filesystems, and data files. For programmability, it incorporates its own programming language, Enscript, that resembles C++ and Java.

As with TSK, EnCase showed numerous defects with relatively simple fuzzing techniques, although we also created targeted, domain-specific faults in test data, such as carefully crafted partition tables and NTFS directory structures.[4]

We tested EnCase versions 6.2 and 6.5.

6.1 Note

Due to the deadline for submitting presentation materials to Black Hat and the ongoing nature of our conversation with Guidance Software, we are unable to present in this revision of the paper all the details of the defects in EnCase that we found. By the time you read this, this version of the paper may be out of date and the canonical version may have the defect details. Please see https://www.isecpartners.com/blackhat/ to find the most recent version of this paper. We apologize for the inconvenience.

6.2 Disk Image With Corrupted MBR Partition Table Cannot Be Acquired

EnCase cannot properly acquire disks with certain corrupted MBR partition tables. When running linen on a system with a disk with a carefully crafted partition table (including many partition table entries), linen won't start up properly. If linen is started prior to corrupting the image, it will start up, but EnCase will hang indefinitely while acquiring the image. (It is possible to cancel out of the linen import.)

If a disk image is made and transferred to the EnCase workstation and acquired as a raw disk image, EnCase will hang indefinitely while attempting to acquire the image. There is no way to cancel out of this process — the GUI becomes unresponsive. We have not identified the root cause of this issue, but it appears

[4]We noticed instances where EnCase's remote acquisition tool for Linux, linen, could not process a corrupted disk image if linen was started up after the filesystem was corrupted. Since our primary goal was to test EnCase, not linen, we worked around these issues by running linen prior to corrupting a disk image without further analyzing the issues in linen.

to be due to the overly large values in the 29th partition table entry. We were unable to reproduce this issue in similar situations with a small number of partitions.

6.3 Corrupted NTFS Filesystem Crashes EnCase During Acquisition

EnCase crashes while acquiring certain corrupted NTFS partitions. The crash occurs when EnCase processes FILE records that contain a larger-than-expected offset to update sequence value, causing it to read past the end of a buffer, resulting in a read access violation. Here is an example FILE record that causes the crash.

6.4 Corrupted Microsoft Exchange Database Crashes EnCase During Search and Analysis

EnCase crashes while searching/analyzing a filesystem containing a corrupted Microsoft Exchange database, as seen in Figure **??**. The crash occurs during the searching phase of an acquisition in which all Search, Hash and Signature Analysis options were enabled. The crash appears to be a read access violation with a bad value in *eax* that is dereferenced, but the exact value in *eax* appears to change every time (or at least very often). We have not determined the cause of or full implications of this problem.

6.5 Corrupted NTFS Filesystem Causes Memory Allocation Error

EnCase reports memory allocation errors when acquiring corrupted NTFS images. The size of memory being allocated is under the control of the attacker. iSEC has not found any ill effects caused by this error condition other than an error being displayed and corrupted records not being displayed.

6.6 EnCase and Linux Interpret NTFS Filesystems Differently

EnCase and Linux appear to use different NTFS metadata when parsing directory structures. We created an NTFS image with a directory loop in it by modifying an NTFS filesystem and replacing a directory entry for a file with a reference to the directory's parent directory. When mounting this directory in Linux[5], the modification was as expected and a directory loop was present. When importing this image into EnCase, the loop was not present and the original file was still present in the directory — but EnCase did not make other files in the directory visible.

This difference in behavior can be used by an attacker to hide data on a disk. An NTFS image can be constructed that has one interpretation on Linux and another in EnCase.

We manually edited an NTFS image to create a directory loop. This directory loop was visible in Linux (when using the NTFS-3g driver) but to our surprise, EnCase did not see the edits we made. Instead it displayed the unedited file. This indicates that EnCase and Linux give different interpretation to NTFS images, probably by using different parts of the redundant information stored in the filesystem. An attacker could abuse this inconsistency to hide data that could only be viewed in Linux and not in EnCase.

[5]iSEC used the NTFS-3g Linux driver: http://www.ntfs-3g.org/.

6.7 EnCase Crashes When Viewing Certain Deeply Nested Directories

We created NTFS images with very deeply nested directories and observed that EnCase would crash in different ways after the image was acquired when performing the Expand All action, or when manually expanding the subdirectory views in the file browsing GUI. Some of these crashes were caused when the program used a return address on the stack that had been overwritten. The values being written to the stack were small integers. While were able to manipulate the value of these integers to some degree, we were unable to exploit this flaw for arbitrary code execution.

7 Conclusion

We performed focused, shallow, and narrow testing of EnCase and The Sleuth Kit, yet immediately found security flaws with simple attack tenchniques. We believe these vulnerabilities exist for several reasons:

1. Forensic software vendors are not paranoid enough. Vendors must operate under the assumption that their software is under concerted attack. After all, the software is often used to examine evidenced seized from suspected computer criminals and from computers suspected to have been compromised by an attacker — that is, the evidence has been under the control of someone capable and motivated to frustrate an investgation against them, or to attack again.

2. Vendors do not take advantage of the protections for native code that platforms provide, such as stack overflow protection, memory page protection (e.g. ensuring that the write bit is unset whenever the execute bit is set on a page), safe exception handling (specific to Microsoft C), et c. EnCase in particular is not designed to be run by a low-privilege user, ensuring that any successful code-execution attack runs with maximum privilege on the forensic workstation.

   The use of managed code eliminates many types of attack altogether.

3. Forensic software customers use insufficient acceptance criteria when evaluating software packages. Criteria typically address only functional correctness during evidence acquisition (*not* analysis) when no attacker is present,[6] yet forensic investigations are adversarial. Therefore, customers should pressure vendors to observe the practices in (2) and to perform negative testing against the product (discussed further below).

4. The software and methods for testing the quality of forensic software should be public. Carrier notes[7] that sufficient public testing tools, results, and methodologies either don't exist or are not public. Making these public will help customers know what they are getting and where they may be vulnerable, and may even raise the standard of testing and improve the quality of the software.

[6] see http://www.cftt.nist.gov/ and [1], specifically lines 43 – 46 ("The two critical measurable attributes of the digital source acquisition process are accuracy and completeness. Accuracy is a qualitative measure to determine if each bit of the acquisition is equal to the corresponding bit of the source. Completeness is a quantitative measure to determine if each accessible bit of the source is acquired.") and 86 – 172. Although the NIST document focuses strictly on the acquisition of evidence, it is not enough to standardize only acquisition. Most forensic toolkits also include functionality for evidence analysis, and it is at the analysis stage where security, not just accuracy and completeness, is crucial.

[7] http://dftt.sourceforge.net/: "To fill the gap between extensive tests from NIST and no public tests, I have been developing small test cases."

7.1 Future Work

We have only scratched the broad attack surface of the products we investigated. We fuzzed and manipulated only some of the most common data types and only in simple ways — other data formats and more sophisticated attacks are likely to bring more defects to the surface.

7.2 Acknowledgements

We thank the vendors, Guidance Software and Brian Carrier, for their fast and helpful responses to our issue reports. Thanks also go to Jesse Burns for his help in debugging software on Windows, and for original authorship of the mutation functions we used in fuzzing.

References

[1] http://www.cftt.nist.gov/DA-ATP-pc-01.pdf. 11

[2] http://dftt.sourceforge.net/.

[3] http://www.seccuris.com/documents/papers/Seccuris-Antiforensics.pdf.

[4] http://www.blackhat.com/presentations/bh-usa-05/bh-us-05-foster-liu-update.pdf.

[5] http://metasploit.com/projects/antiforensics/.

[6] http://www.simson.net/clips/academic/2007.ICIW.AntiForensics.pdf.

[7] B. Carrier. File system forensic analysis. Addison Wesley, 2005.