CERIAS Tech Report 2001-41

An Embedded Sensor For Monitoring File Integrity

James P. Early
Center for Education and Research in
Information Assurance and Security
Purdue University, West Lafayette, IN 47907

An Embedded Sensor For Monitoring File Integrity

James P. Early

March 31, 2001

Abstract

This paper describes a method of monitoring file integrity (changes in file contents) using a collection of embedded sensors within the kernel. An embedded sensor is a small piece of code designed to monitor a specific condition and report to a central logging facility. In our case, we have built several such sensors into the 4.4 BSD kernel (OpenBSD V2.7) to monitor for changes in file contents. The sensors look for files which are marked with a specific system flag in the inode. When the sensors detect a file with this flag, they will report all changes to file contents made through the file system interface. This provides administrators with a valuable audit tool and supplies more reporting granularity than conventional file system integrity checkers (such as Tripwire ®).

Our technique relies on only two fundamental file system characteristics. First, the file system object must have a provision for storing file characteristics (i.e. flags) within the object. Secondly, the file system must present a block device interface to the operating system.

We show that system performance is not severely hampered by the presence of this monitoring mechanism given the select set of files that would be monitored in a conventional system and the beneficial audit data that results from monitoring.

1 Introduction

Maintaining integrity of files within a file system is a fundamental component of system security [11]. Nearly all modern computers use a file system as a means to maintain persistent data that is crucial to system performance. This data is often in the form of system executables and system configuration files. Files in these categories are logical targets of individuals intent on altering the system's performance or gaining access.

Recognizing the importance of file integrity in a system, several tools have been developed over the past ten years to monitor a collection of files for changes in file contents [3, 4, 14]. Among the most popular file integrity checkers currently in use is a tool called Tripwire®. Tripwire[1] is used to create a database of cryptographic hashes (message digests) which are called *file signatures* [7]. With this database in a secure location (usually a write

[1]Tripwire is a registered trademark of Tripwire, Inc.

protected medium), the Tripwire tool interrogates files listed in its database and compares a file's current signature to the one stored in the database. The signature properties make it highly unlikely that a malicious user could alter a file to produce a given signature value. Further, the space of available signature values is very large (2^{128} possible values for MD5 message digests) [12] making it highly unlikely that two different files would share the same signature. Therefore, Tripwire is able to confidently report a file modification when signatures do not match.

2 Motivation

The motivation for the work in this paper is the desire to provide enhancements to existing methods of monitoring file integrity.

Typically, a file integrity tool such as Tripwire is an application run on a periodic basis - perhaps once per day. This means that if a file were modified during the interval between runs, then again modified back to the original, the modification would go unnoticed. One such example might be a modification to the */etc/passwd* file on a UNIX system in order to create a temporary account, then restoring the original file. In cases such as this, it is desirable to have *temporal* integrity monitoring. We wish to know about modifications at the time they occur.

Next, suppose that we wish to have audit information about file modifications. Assuming a user writes to a file, we might want to know the user's id, process id, and the contents of the write request. Unfortunately, conventional integrity tools do not have access to this information because the information is readily available only to the kernel. If a file integrity mechanism were placed in the kernel, this information could be collected.

Finally, it is clear that some files on the system are modified regularly during normal operation. As a result, we need to frequently update our database of file signatures. If the database is stored on an read-only medium, we may need to create a new medium.

In essence, we seek more granularity in the information available about file content modifications. The need to provide temporal integrity monitoring, audit data about the modification, and the acknowledgment that some files on the system may change frequently has led us to develop our kernel-based file integrity monitor. This new mechanism provides the additional needed granularity.

3 Scope of Monitoring

The work described in this paper focuses on monitoring changes in file contents, i.e. data within a file. We do not address modifications to file characteristics such as permissions or file name, nor do we address the deletion of files or creation of new links to files. We impose this limitation because the file signatures described above are independent of this meta file data. Another reason for this decision is the existence of mechanisms to handle these issues [13, 10]. Finally, we envision the use of our mechanism on files where the integrity of the contents is paramount over issues such as file name, permissions, or whether the file has been deleted. Such files can govern system performance and therefore a modification of contents is the most important objective to a malicious user.

4 The Embedded Sensor Paradigm

The need to position our file integrity monitoring mechanism in the kernel necessitates a design that does not place an undue burden on operating system resources and performance. The following is a list of desirable properties of the monitoring mechanism.

- Small code footprint in comparison to the surrounding kernel code
- Avoids excessive context switching
- Does not require additional kernel data structures

Zamboni [15] describes a paradigm called Embedded Sensors which employs small pieces of code strategically placed in operating systems or applications for use in intrusion detection systems. These sensors are designed to monitor for a specific condition and report to a central logging facility. Kershbaum [6] then used this technique to create sensors that look for certain classes of network-based attacks. This paradigm meets the objectives we've outlined above and provides an additional advantage in that the sensors are difficult to circumvent [13]. Thus, the embedded sensor design approach was selected as the model for our file integrity monitor.

The operating system chosen for the above embedded sensor implementations is OpenBSD version 2.7. This version of OpenBSD is based on the 4.4 BSD kernel. There were two primary reasons for this selection. The first reason is the fact that the source code is freely available. Secondly, the operating system employs design elements and concepts that are widely used in other commercial and freely-available UNIX variants. As such, we have also chosen to implement our file integrity monitor in OpenBSD.

5 File Marking

5.1 Design Considerations

We next consider the question of how to mark files that we wish to monitor. We stated in the previous section that one of our design objectives was to avoid the use of additional kernel data structures in an attempt to improve efficiency and reduce overhead of our sensor. Therefore, it would not be appropriate to create a list of files that the kernel would need to maintain. Also, we would like this marking to be persistent such that in the event of a system crash, the knowledge about the monitoring is recovered easily.

Another option might be to keep the list of files we wish to monitor in a separate file. This method has another disadvantage, namely that we would need to do frequent read operations (and therefore context switches) in order to check if a file being accessed was one requiring monitoring. This approach would likely have a severely negative impact on system performance.

The method we have chosen to employ places the marking in the file's physical disk structure. This structure is called the *inode* and is widely used by many UNIX file systems. The inode is identified by a unique number within a given file system. Thus, by marking the inode, we uniquely identify the physical file object we wish to monitor, regardless if this object is referenced by multiple file names. In addition, the marking is persistent.

The marking itself is designated as a single bit in the inode's flags field. This field is usually a 32 bit integer with each flag bit indicating the presence of a given attribute. We define a new flag bit to indicate file marking. This gives us a simple way for the sensor to determine if a given file is to be monitored.

5.2 Implementation

An inode is a structure containing a collection of pointers to physical disk blocks [1]. These disk blocks hold the data contents of the file. In addition to block pointers, inodes contain a flags field which describe characteristics of the file. In the BSD implementation, this field is a 32 bit value, separated logically into *owner* and *super-user* (i.e. root) set-able flags. The lower order 16 bits can be set by the file's owner (as defined in the file permissions) and the super-user, while the high order 16 bits are only set-able by the super-user.

Traditionally, inode flags have been used to enforce a particular security policy [10]. For example, one such existing inode flag is the *immutable* flag. Once set, this flag prevents the file from being modified in any way. Another flag is called *append*. This flag mandates that file writes may only append information - no information can be overwritten. There are super-user-only versions of these flags (also known system flags) in the 4.4 BSD implementation are called *SF_IMMUTABLE* and *SF_APPEND*. The SF_IMMUTABLE flag enforces an "immutable" policy, namely it prohibits all changes to the file. The SF_APPEND flag prohibits all writes to the file that are not appends of new data. In other words, we are only allowed to add new information to the end of the file. The system flags have the additional property that they cannot be cleared unless the security level of the system is reduced to zero - typically by placing the machine in single user mode.

We chose to locate our flag in the system flag area in order to prevent users from being able to clear the flag. We name our new inode flag *SF_WRMON* - the system flag for write monitoring. This new flag shares a number of properties with the immutable and append flags. First, as a system flag, it may only be set by a user with root privileges. In addition, once set, this flag cannot be cleared unless the system's security level in downgraded to zero.

In order to set this flag, we needed to modify the *chflags* routine. This routine is responsible for first checking that the user has appropriate permissions to be able to change the flags within the inode, then modifying the flags field. Our modification makes this routine aware of the new SF_WRMON flag.

We set this flag in the inodes of files or devices that we wish to monitor for write activity and other changes to file contents. Henceforth, we will refer to files with this flag set as *write-monitored files*.

6 Vnodes

This section describes an interface layer to files in the 4.4 BSD kernel called the *Vnode interface layer* [8, 10]. The purpose of the Vnode layer is to provide a single interface to multiple types of file systems. A new Vnode is created for each active file in the kernel. For the purposes of this paper, we will use the Vnodes as an in-kernel representation of the inodes. Thus, we can examine the flags within the Vnode to determine if the file is write-monitored.

Figure 1 shows how this interface is positioned in the calling sequence.

7 Sensor Location

7.1 Design Considerations

The governing principle for placement of our sensors within the kernel was to position them such that the variables being monitored were in the scope of the sensors. By this we mean that we could access variables without the need to iterate through various kernel data structures or make additional function calls. Our system flag indicating a write-monitored file is available through-

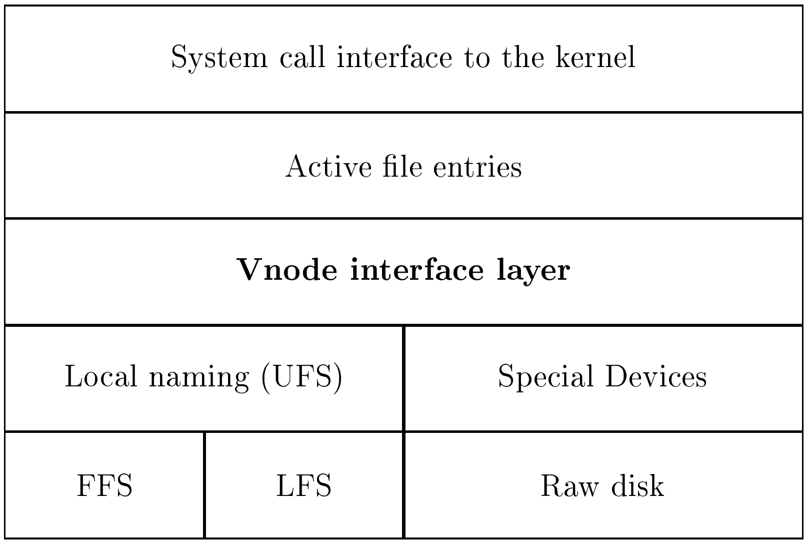

Figure 1: Position of Vnode interface layer

out in the Vnode interface layer. Logically, then, this led us to position sensors at the boundaries of the Vnode interface.

At the upper boundary of the Vnode interface, we have visibility to process information (process Id and user ID) and to the information passed in by the user. At the lower boundary, we make use of the logical blocks representing the underlying file system [10]. These logical blocks are characteristic of *block devices* which transfer information between the kernel and the physical disk hardware in units of blocks. The kernel makes use of these blocks to improve the performance of I/O operations through caching. The contents of the logical blocks represent the contents of the physical disk blocks (or they soon will), so our monitoring mechanism can compare the information in the user's request to the current disk contents to determine what was changed.

An exception to the above is the so called *raw device* [10, 9, 1]. Raw devices do no make use of the buffer cache within the kernel as do block devices. Therefore, we do not have the benefit of being able to compare the user's request with the information in the logical blocks.

In the 4.4 BSD kernel, file contents are modified through two primary mechanisms - the *open* system call and the *write* system call. The *open* system call can modify file contents if the programmer calls a library routine [5] such as

```
fp = fopen(myfile, "rw");
```

The above statement opens the file named by *myfile* for both reading and writing and places the read index at the beginning of the file. However, a side-effect of this statement is that the file will be truncated to length zero if it exists. Even if the file were immediately closed without any write activity, the file's contents have been altered. Thus, we place a sensor in the *open* system call to alert us if a write-monitored file has been truncated.

The *write* system call is the major source of activity which causes changes to file contents, and, as such, is the location where our sensors will be the most crucial. When a write call is made on a write-monitored file, we wish to know the context of the call. We would like to know the user and process id responsible for the write request. Further, we would like to know what are the contents of the buffer the user has passed in with the request. Finally, we would like to know exactly what file contents were modified as a result of this request. As noted above, this last item is not possible if the user is writing to a raw device. However, the first two items are possible for all write requests.

At this point, we must acknowledge that all the information we may like to collect is not readily available at all points within the Vnode interface. Further, the difference in functionality between calls to block devices and raw devices must also be considered. With this in mind, we implemented a multi-layered sensor approach.

7.2 Implementation

Our sensors were be placed following traditional checks made by the kernel to determine if a file operation can be performed. For example, we position our sensors after the kernel has checked permissions, and the immutable and append flags. This is done to avoid unnecessary write-monitoring checks if the operation can not be performed.

The reader will encounter numerous calls to a routine called *esp_log* throughout the following code. This is the routine defined by Zamboni [15] to report to the embedded sensor logging facility. Our sensors were implemented on an OpenBSD system that already had this logging mechanism in place for other types of sensors, so we chose to use the same mechanism.

Our first sensor is placed in the *open* call at the interface between Active File and Vnode layers. At this interface, we have visibility to the Vnode and thus can check for the presence of the SF_WRMON flag. We also can check for the presence of the I/O flag O_TRUNC signaling that the file was truncated to length zero. The sensor was placed in the *vn_open* routine within the *vn_vnops.c* file.

The complete sensor reports when a write-monitored file has been opened for writing. It reports the device and inode numbers that uniquely identify the file, the user id and process id that initiated the request, and the file name used in the request. Our sensor looks like this:

```
{
  int tmperror;

  /* need to get the attributes again */
  tmperror = VOP_GETATTR(vp, &va, cred, p);

  if (!tmperror && (va.va_flags & SF_WRMON)) {
    /* This is a write monitored file */
    if (fmode & FWRITE) {
      /* opening for write */
      esp_log("vn_open: fs=%d; inode=%d; %s (pid = %d) opened write \
monitored file\n for writing\n", va.va_fsid, va.va_fileid,
               p->p_pgrp->pg_session->s_login, p->p_pid);
               esp_log("vn_open: File opened as %s\n", ndp->ni_dirp);
    }

    if (fmode & O_TRUNC) {
      /* Notify the file was truncated to zero length */
      esp_log("vn_open: File truncated to length zero\n");
    }
  }
}
```

Our next sensor is placed in the *write* system call, again at the interface between the Active file and Vnode layers. Here, we have access to the contents of the user's write buffer in addition to the process information as seen above. The sensor was placed in the *vn_write* routine, also within the *vn_vnops.c* file. Another I/O flag available at this location is the O_APPEND. This flag indicates that the user's request is an append operation. We refer to this sensor as the *upper level* write sensor because it is positioned at the upper level of the Vnode interface. Here is an excerpt.

```
  if (va.va_flags & SF_WRMON) {
    wrmon_flag = 1;

    esp_log("vn_write: %s (pid = %d) ",
            p->p_pgrp->pg_session->s_login, p->p_pid);

    if (fp->f_flag & O_APPEND) {
      esp_log("appending to a write monitored file\n");
    } else {
      esp_log("writing to a write monitored file\n");
    }

    esp_log("vn_write: fs = %d; inode = %d; requested offset = %d\n",
            va.va_fsid, va.va_fileid, *poff);
    esp_log("vn_write: buffer:\n");

    /* dump the buffer - 16 bytes per line */
    for(i = 0; buf_len - i >= 16; i+=16) {
      esp_log(" %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x
%02x %02x %02x %02x %02x  '%c%c%c%c%c%c%c%c%c%c%c%c%c%c%c%c'\n", ...);

  ...

  /* pass the "truncated" flag forward */
  if (fp->f_flag & O_TRUNC) {
      ioflag |= O_TRUNC;
  }
```

Our final sensor is placed at the interface between the Vnode layer and the Local Naming (UFS) layer. The reason for this placement is that we have access to the Vnode information, user request, and the logical blocks in the kernels buffer cache. We refer to this sensor as the *lower level* write sensor because it is placed at the lower level of the Vnode interface. This sensor was placed in the *WRITE* routine within the *ufs_readwrite.c* file.It is noteworthy that this sensor is placed above the interfaces to the individual file systems. This means that the sensor can operate in a file-system independent environment[2].

It is at this location that we can determine the exact nature of the file

[2]assuming that the file system is based on a block device

content modification requested by the user. To do this, we compare the contents of the user's buffer with the contents of the logical blocks beginning at the requested file offset. Here is an excerpt from the sensor.

```
if (ip->i_ffs_flags & SF_WRMON) {
  /* only compare buffers if we are not appending or
     the file was not truncated
  */

  if (!(ioflag & (IO_APPEND | O_TRUNC))) {
    s_buf = uio->uio_iov->iov_base;
    t_buf = (caddr_t)bp->b_data;

  for (i = 0; i<xfersize; i++) {
    /* Compare bytes and report changes */
    j = i % 8;
    out_buf[j]    = t_buf[i+blkoffset];
    out_buf[j+8]  = s_buf[i];

    if (s_buf[i] != t_buf[i+blkoffset]) {
      mod_flag = 1;
    }

    if (mod_flag) {
      if ((7 == j) || (i == (xfersize - 1))) {
        /* we've found a mismatch within a block of
           the 8 bytes or the last few bytes of transfer
        */
        esp_log("WRITE:%d:%u:%d: Offset %ld:\n",
                ip->i_dev, ip->i_number, p->p_pid,
                (i + uio->uio_offset - j));
        ...

        esp_log(" Old:%02x %02x %02x %02x %02x %02x %02x %02x  \
                  New:%02x %02x %02x %02x %02x %02x %02x %02x  \
                  '%c%c%c%c%c%c%c%c' '%c%c%c%c%c%c%c%c'\n", ...);
    ...
```

In the above sensor code, *s_buf* and *t_buf* refer to the user buffer (source) and logical block buffer (target), respectively. *out_buf* is an output buffer used to hold the values passed to the logging message.

Figure 7.2 provides a graphical depiction of where the individual sensors are located in relation to the various abstraction layers.

8 Modified Routines

This section describes the routines that were modified to accommodate the sensors. In addition to the routines mentioned in the previous section, some others were modified to be conscious of the new SF_WRMON inode flag. Here is a summary of what was done for each routine.

vn_open Added a sensor to report an open for writing call on a write-monitored file. Also, notifies if a file was truncated to length of zero.

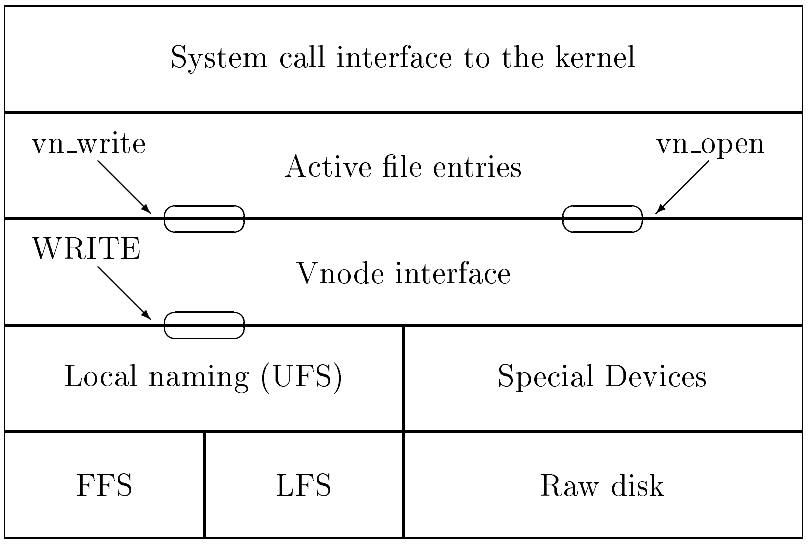

Figure 2: Placement of embedded sensors

vn_write Added a sensor to report the context of a write call on a write-monitored file. The context includes the user id, process id, requested file offset, and the contents of the user buffer passed in with the request.

WRITE Added sensor to compare and report differences between the contents of the user buffer and the contents of the logical block.

chflags Added additional checks for the presence of the new SF_WRMON flag in the inode. This prevents this flag from being cleared while the system is in secure mode.

sys_open Modified the *f_flag* field such that the O_TRUNC flag was stored in the active file structure. This permits an output optimization wherein we do not compare the user buffer with the logical block buffer if the file is truncated.

In total, roughly two hundred lines of code (including comments) were necessary to add these sensors and their supporting variable declarations to the kernel. This total represents a small percentage of the total amount of code used in each calling sequence. The size of the kernel was not effected by the addition of this code.

Also, through judicious placement of the sensors, we were able to effectively use needed variables without the need for excessive context switching or additional kernel structures. The only required function calls were those to the *esp_log* logging mechanism and a single call to the routine *VOP_GETATTR* [3].

9 Use on Directories

We have used the marking and sensors described here on directory inodes. However, further work must be done to develop the semantics and actions of the sensors in this context. Also, several sensors described by Zamboni [15] must be made aware of the new SF_WRMON flag. We are pursuing this as future work. Thus, the performance analysis we present is confided to regular files.

[3]Used to copy the inode flags into the Vnode

10 System Performance

10.1 Output Messages

In this section, we will examine the output messages from each sensor. Each message is designed to provide a summary of modification activity taking place at the level where the sensor is operating.

Our first example shows an append to a write monitored file.

```
vn_open: fs=4; inode=312304; user1 (pid = 15468) opened write monitored
 file for writing
vn_open: File opened as passwd
vn_write: user1 (pid = 15468) appending to a write monitored file
vn_write: fs = 4; inode = 312304; requested offset = 0
vn_write: buffer:
 62 61 64 67 75 79 3a 2a 3a 31 30 30 34 3a 31 30  'badguy:*:1004:10'
 30 34 3a 42 61 64 20 47 75 79 3a 2f 68 6f 6d 65  '04:Bad Guy:/home'
 2f 62 61 64 67 75 79 3a 2f 75 73 72 2f 6c 6f 63  '/badguy:/usr/loc'
 61 6c 2f 62 69 6e 2f 62 61 73 68 0a 00 00 00     'al/bin/bash....'
vn_write: request size: 60
vn_write: bytes written: 60
```

The first sensor in *vn_open* alerts us that user1 has opened a write-monitored file for writing. The file is identified by the file system number (fs) and the inode number. The file name used to open the file was "passwd". Had user1 truncated the file as described above, we would have seen an additional output line notifying us of the truncation.

Some time later, this same user appends a line to the end of the file adding a new user id for “badguy”. Recall that once the inode flag has been set on a write-monitored file, it cannot be cleared unless the system is brought down to security level 0. Thus, even if user1 had root privileges, he would not be able to perform this action without being detected. This user’s write request was 60 bytes in length, and all 60 bytes were written.

We observe that there is no message from the lower level *WRITE* sensor. The reason for this is the presence of an output optimization. This sensor does not report in the event of an append or truncation. In these two cases, all comparisons with logical blocks will not match because we are writing new data. The buffer reported by the *vn_open* sensor provides the extent of the modification.

In this next example, we will look at a modification that involves an overwriting of existing data.

```
vn_open: fs=4; inode=312304; user1 (pid = 569) opened write monitored file
 for writing
vn_open: File opened as passwd
vn_write: user1 (pid = 569) writing to a write monitored file
vn_write: fs = 4; inode = 312304; requested offset = 30
vn_write: buffer:
 2f 62 69 6e 2f 73 68 20 00 00 00 00 00 00 00     '/bin/sh .......'
vn_write: request size: 8
WRITE:4:312304:569: Offset 30:
 Old:2f 62 69 6e 2f 6b 73 68  New:2f 62 69 6e 2f 73 68 20  '/bin/ksh' '/bin/sh '
vn_write: bytes written: 8
```

As in the previous example, we get notification from the *vn_open* and *vn_write* sensors indicating the user has opened and subsequently written to a write-monitored file. The additional data in this example comes from the lower level *WRITE* routine. The modification data is preceded by a tuple of the file system, inode, process id number, and the write offset. Here, the we are notified that the string "/bin/ksh" was changed to "/bin/sh". In fact, this particular message was generated as a result of the root user's shell being changed.

We noted earlier that raw devices do not employ the kernel copy of information between the user's buffer and logical blocks. Therefore, the lower level *WRITE* sensor will not be triggered for a raw device that is write-monitored. However, the sensors in *vn_open* and *vn_write* will function as they do for block devices. This means that we will still get the contents of the user's write request and the requested write offset. Thus, it is still a very useful auditing tool.

10.2 Overhead Measurements

10.2.1 Measurement Procedures

The additional work involved in checking for the presence of the write-monitor flag in the inode, reporting the user's write request, and comparisons of this data with the current file contents creates increased overhead for the kernel in executing the *open* and *write* system calls. This section is devoted to quantifying and analyzing this overhead.

Our first step was to establish a benchmark for testing. We chose the Bonnie [2] file system benchmarking tool because of its wide use and its freely available source code. Bonnie operates by creating a file of specified size, then performing a series of sequential and random read and write operations. These operations are further subdivided into character and block sizes. The availability of the source code meant that we could modify Bonnie's functionality allowing us to specify an existing file for testing rather than having Bonnie create one. As a result, were able to instruct Bonnie to perform it's tests on different files with and without the write-monitor flag set.

The next step involved creating a collection of kernels with varying degrees of active sensors. The following is a list of the kernels and their attributes.

Kernel 1 This kernel contains the base code for embedded sensors, namely the *esp_log* mechanism, but contains no sensor code for write-monitored files. This represents our baseline.

Kernel 2 This kernel contains all the sensor code for write-monitored files, but does not make any calls to the *esp_log* routine. This allows us to gauge the overhead imposed by the logging mechanism itself.

Kernel 3 This kernel contains only the *vn_open* sensor.

Kernel 4 This kernel contains only the high level *vn_write* sensor.

Kernel 5 This kernel contains only the low level *WRITE* sensor.

Kernel 6 All sensors are active in this kernel.

With each kernel active, we ran Bonnie several times using test files that were write-monitored and those that were not. We then averaged the results to determine the mean for each type of test file. After some initial trials, we chose a test file size of 10 Mb. This value was chosen because it provided a good compromise between a reasonably short running time and low variability among the individual test results. Table 1 lists the average file system bandwidth measurements in megabits per second for each kernel, first with the write-monitor flag cleared, then with this flag set.

	Character Writes		Block Writes	
Kernel	No WRMON	WRMON	No WRMON	WRMON
1	29.1	-	14.6	-
2	29.1	10.0	14.5	12.5
3	29.0	27.4	14.6	16.3
4	29.1	1.2	14.3	1.2
5	29.1	0.42	14.6	0.42
6	29.0	0.28	14.4	0.28

Table 1: Bandwidth measurements for character and block writes. All measurements are in megabits per second (Mb/s)

10.2.2 Analysis

In this section, we will analyze the results in Table 1. The first thing that is apparent is the nearly constant performance of these kernels when they are operating on files that are not being monitored for write activity. Regardless of the which sensors are active, the bandwidth measurements for character and block writes remain largely unchanged. If we assume that the general case involves writes to files that are not write monitored, the overhead of the sensors on file system performance as a whole is likely to be quite small.

We next compare the results from tests using our base kernel (kernel 1) and our kernel with no calls to *esp_log*, kernel 2. By comparing these results, we hope to determine the overhead of the logging mechanism. The bandwidth is reduced by a factor of 3 for character writes, but by a significantly smaller factor (1.16) for block writes. We explain this difference by observing that there is a constant overhead involved in retrieving information, such as the inode flags. For small writes, the overhead is dominated by this constant amount of work. For large writes, however, the overhead is dominated by the reporting of the user buffer and logical block comparisons. But, since we are not making calls to the logging mechanism, the result is a smaller overhead factor for block writes.

We now investigate three more kernels, each with one sensor active, in order to determine which sensor dominates the overall performance. Comparing kernels 1 and 3 (only the *vn_open* sensor), it is clear that the *vn_open* sensor contributes very little to the combined overhead. This should be clear when we consider that the work done by this sensor is not a function of the user's buffer. As we observed above, the performance impact of this sensor is larger on small writes as compared to large writes. Again, this is caused by the fact that the sensor's work is constant.

Looking at kernels 1 and 4, we see a significant impact for both types of writes. The sensor active in this kernel is *vn_write*. It is the first sensor whose work is a function of the size of the user's buffer request and, in turn, a potentially large number of calls to the *esp_log* mechanism. However, we again observe that the relative impact differs for character and block writes. In this case, it is caused by the structure of the output message. A one character write will result in one call to to *esp_log*. In the case of block writes, calls to *esp_log* write 16 bytes at once.

A comparison between kernels 1 and 5 shows further reductions in bandwidth. Kernel 5 contains the low level *WRITE* sensor. Like *vn_write*, the work of this sensor is governed by the size of the user's buffer. This determines the size of the comparison with logical blocks as well as the resulting calls to *esp_log*. As above, the relative impact is different for character and block writes. The structure of the output is also the cause here. In the case of this sensor, the output line is optimized for comparisons of up to eight bytes.

The cumulative effect of all the sensors can be seen when we compare kernels 1 and 6. The result is roughly a reduction in bandwidth by a factor of 100 for character writes and by a factor of 50 for block writes. If all of the files on the system were being write-monitored, this would be a significant impact, indeed. However, it is important to keep in mind one of our original objectives, and that was that we wished to identify and tag a select group of files that are crucial for system performance, but may change during normal operation. The overhead for the individual file may be high, but due to the fact that the vast majority of files will not be monitored, the impact on the overall system is much lower. For example, if 1 out of 100 write requests were made to a write monitored file, the overall performance would be decreased by slightly less than 1%.

At this point, it may occur to the reader that the low level *WRITE* sensor may be eliminated if we do not care to know the exact contents that were modified. This is a possibility. Within the write system call, the upper level sensor has added importance in that it is the only one that can monitor write activity to raw devices. Thus, in choosing one write sensor, the upper level *vn_write* provides the widest monitoring coverage. The user can chose the reduced granularity of only the upper sensor in order to achieve higher bandwidth.

11 Resistance to Tampering

In this section, we will look at our mechanism's resistance to tampering. As we noted in our discussion of the embedded sensor paradigm, sensors placed in the kernel are difficult to disable. Thus, the logical point of attack is the inode flags field containing the SF_WRMON flag.

Let us consider the situation of a local user wishing to disable the monitoring on a particular file without detection. The user must clear the

SF_WRMON flag in the file's inode in order to achieve the first part of his objective. This must be done by either writing to the raw disk device or by using the system's chflags routine. We now describe these two attack scenarios in more detail.

In our first scenario, the user has decided to attempt to clear the SF_WRMON flag by writing to the raw disk. However, access to the raw device is privileged meaning that the user must first gain root privileges. Assuming he is able to gain these privileges, he must now determine the exact offset into the raw disk that points to the SF_WRMON flag. If he is able to do this, he can then issue his write command. However, recall that SF_WRMON flag is also intended to be set in the inode for the raw disk device. Therefore, a record of this user's write request will be logged and he will have failed in his attempt to avoid detection.

In the other scenario, our attacker attempts to disable monitoring by using the *chflags* routine. Recall that our SF_WRMON flag is a system flag that can only be cleared by the root user. Thus, our attacker must first gain root privileges. Assuming he is able to do so, let us further assume that the system is operating at a security level above zero. Recall that our SF_WRMON flag cannot be cleared while the security level is above 0, so the user would have to bring the system down to single user mode. This is unlikely to escape notice. Thus, the attacker is again unsuccessful in his attempt to avoid detection.

In essence, the inherent difficulty of gaining the required permissions and access necessary to circumvent the sensor is what makes this mechanism highly resistant to tampering.

12 Conclusion

We have presented an effective file integrity monitoring mechanism that provides system administrators with enhanced understanding and auditing capabilities. We have demonstrated that the Embedded Sensor paradigm applied to this solution provides a method that requires relatively little code in comparison to the surrounding kernel code. Furthermore, this paradigm allows us to implement a solution that does not add to the complexity of the kernel (in terms of additional data structures), nor to the kernel's size.

The introduction of the new inode flag SF_WRMON adds new semantics to such flags to include monitoring of events that effect the file. This also provides a simple and efficient means to mark files for monitoring. The marking is persistent in the event of system crashes. In addition, the properties of these flags make them difficult to remove once they have been set.

We have presented a collection of three sensors which, when combined, give as us a means to monitor the vast majority of events that can result in changes to file contents. Our sensors detect appends, truncations, and direct modification of file data. They perform this function for both block and raw devices, and do so in a file system independent way.

The underlying concept relies on only two mechanisms - a way to store (and ideally, protect) flags in the on-disk file structure and the presence of logical blocks representing, in the kernel, the file's current contents. As such, this monitoring concept can be extended to any file system that meets these criteria.

Finally, we presented our results measuring the overhead imposed by the use of the sensors. We showed that, although the impact on a single file can be large, the impact on the system as a whole is much smaller. This is due to the fact that the sensors have negligible impact when operating on files that are not being monitored. The added auditing data combined with low overall system impact enhances the current state of file integrity tools.

References

[1] Maurice J. Bach. *The Design of the UNIX Operating System.* Prentice-Hall, Upper Saddle River, NJ 07458, USA, 1986.

[2] T. Bray. Bonnie file system benchmark. http://www.textuality.com/bonnie/.

[3] Daniel Farmer and Eugene H. Spafford. The COPS security checker system. In *Proceedings of the Summer Conference*, pages 165–190, Berkley, CA, 1990. USENIX Association.

[4] David K. Hess, David R. Safford, and Douglas Lee Schales. The TAMU security package: An ongoing response to internet intruders in an academic environment. Technical report, Texas A&M University, 1993.

[5] Brian W. Kernighan and Dennis M. Richie. *The C Programming Language.* Software Series. Prentice Hall, second edition, 1988.

[6] Florian Kerschbaum, Eugene H. Spafford, and Diego Zamboni. Using embedded sensors for detecting network attacks. In Deborah Frincke and Dimitris Gritzalis, editors, *Proceedings of the 1st ACM Workshop on Intrusion Detection Systems.* The Association of Computing Machinery SIGSAC, November 2000.

[7] Gene H. Kim and Eugene H. Spafford. The design and implementation of Tripwire: a file system integrity checker. In *Proceedings of the 2nd*

ACM Conference on Computer and Communications Security, pages 18–29. The Association of Computing Machinery, 1994.

[8] S. R. Kleiman. Vnodes: An architecture for multiple file system types in Sun UNIX. In *USENIX Conference Proceedings (Atlanta, GA)*, pages 238–247. USENIX Association, Summer 1986.

[9] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System.* Addison-Wesley, 1989.

[10] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System.* Addison-Wesley, 1996.

[11] Charles P. Pfleeger. *Security in Computing.* Prentice Hall, second edition, 1996.

[12] R. L. Rivest. RFC 1321: The MD5 message-digest algorithm. Technical report, Internet Activities Board, April 1992.

[13] Eugene H. Spafford and Diego Zamboni. Design and implementation issues for embedded sensors in intrusion detection. In *Third International Workshop on Recent Advances in Intrusion Detection.* Recent Advances in Intrusion Detection (RAID2000), October 2000.

[14] David Vincenzetti and Massimo Cotrozzi. ATP – anti tampering program. In *Proceedings of the Fourth USENIX Security Symposium*, pages 79–89, Santa Clara, CA, 1993.

[15] Diego Zamboni. Doing intrusion detection using embedded sensors. Technical Report 2000-21, CERIAS - Purdue Univeristy, West Lafayette, IN, October 2000.