Fast and Secure Append-Only Storage with Infinite Capacity

Yongge Wang and Yuliang Zheng
Department of Software and Information Systems
University of North Carolina at Charlotte
{yonwang, yzehng}@uncc.edu

August 27, 2003

Abstract

Computer forensic analysis, intrusion detection and disaster recovery are all dependent on the existence of trustworthy log files. Current storage systems for such log files are generally prone to modification attacks, especially by an intruder who wishes to wipe out the trail he leaves during a successful break-in. In light of recent advances in storage capacity and sharp drop in prices of storage devices, as well as the demand for trustworthy storage systems, it is timely to design and develop fast storage systems that practically have no limit in capacity and admit "secure append-only" operations (namely data can only be appended to a storage device; once appended it can no longer be modified, and can be read out by authorized users only.) This paper reports some preliminary findings in our research into building a secure append-only storage system. It discusses a possible secure append-only storage architecture that could be used to detect and prevent deletion or modification by inside/outside attackers. A specific implementation of the architecture based on block device drivers is also presented.

1 Introduction

Today's cyber-infrastructure has numerous vulnerabilities, and it is unlikely that this situation will change significantly in the short term. Thus it is very important to design detection and prevention mechanisms within the present infrastructure to enhance its security.

Security measures require the existence of trustworthy log data about a system to be examined. Two well-known examples are computer forensics and intrusion detection both of which rely on inputs from log files about system and user activities. A major challenge is presented by sophisticated attackers who could delete all the system logs after a successful intrusion. An inside attacker may also easily delete all the trails that she/he has left. It is becoming increasingly evident that there is a need to develop storage systems that are able to provide unalterable, permanent records on computer system and user activities for the entire life span of a system. Specifically, the benefits of creating a complete, unalterable and permanent record of all activities on a computer system (infrastructure) include at least the following:

- Advanced intrusion detection systems can use these records to detect attacks as they happen.
- Computer forensic analysis could be carried out based on these records; the property of unalterability may also serve as court evidence in legal proceedings.
- The very presence of a secure and unalterable append-only log storage itself may act as a quite effective deterrence to potential inside and outside attackers. When a potential attacker knows that her/his activities will be recorded there permanently and he/she has no way to delete them, she/he may hesitate to launch an attack in the first place.

A major goal of this paper is to design a secure append-only storage system to fight against attackers who may have root privileges to a computer system and who may try to delete log files containing information about the attacks. Such type of attackers include both inside and outside attackers.

In what follows we discuss major requirements of storage systems for the creation of a permanent, immutable record on computer system activities, which is followed by the description of a secure append-only storage architecture that meets the requirements.

2 Requirements and related works

A storage system for recording computer system activities should satisfy the following requirements:

- Efficiency. It can record a huge amount of data on activities in real time and well before an attacker has time to delete them.
- Append-only. Even a sophisticated attacker should not be able to change any information that has already been written to a storage device.
- Permanency. The storage device should be able to provide permanent evidence for computer system and user activities.
- Unlimited capacity in practice. Each append-only storage device should be able to record activities for a relatively long period of time (e.g., a week), and an administrator can easily (and not so frequently) install a new storage device when the old one is full.
- The device should be cheap enough so that large scale deployment is economically viable. Furthermore, it is preferable that the secure append-only storage system could be built from COTS components.
- Security. The devices may contain critical and confidential information about computer activities. Thus the data stored on the device should be protected with strong cryptographic techniques in such a way that data stored can be readily accessible to authorized users only.

The "Advanced Packet Vault (APV)" project at University of Michigan designed append-only storage devices based on CD-ROM technologies. Basically, their prototype writes captured network packets to long-term CD-ROM storage using encryption for later analysis. Though the APV provides a simple solution for intrusion detection purpose and meets some of the requirements that we have mentioned above, it is not a satisfactory solution. In particular, the APV has the following shortcomings:

- It is very slow to write due to the use of CD-ROM devices.
- The capacity of CD-ROM discs is limited. Even for a moderately large installation, one may need to replace discs in a very short time period.
- In the APV architecture, the system receives network (encrypted) packets and assembles them on magnetic disk for subsequent writing to CD-ROM. If the machine is under attack, then the attacker may have sufficient time to delete the assembled information on the magnetic disk before it is written to the CD-ROM. Though it is possible to install the APV system in a hardened OS that has no Internet services and an outside attacker may have no chance to get the root access for the APV system, an inside attacker who has local access to the APV system could still delete the data before they are written to the CD-ROM.

The above analysis shows that while the APV presents an interesting experiment, it is still open to find a better solution to the permanent record problem for network activities.

File systems for special purposes have been implemented in several operating systems. For example, the Linux Intrusion Detection System (LIDS) group has implemented the LIDS system [9] which could achieve the following goals:

- Read-Only Files/Directory. Read only files means that they do not allow any user including root users to modify it.
- Append-Only Files/Directory. Append-only files means that one can only append bytes to the end of the file, no users (including root users) can do any other operations on the files.
- Exception-Files/Directory. The files is not to protected. In some case, one wants to protect the whole directory but also want some specified file to be unprotected, so one can defined the files as exception and the directory as read-only.
- Protection-mounting/unounting. When some filesystems are mounted after system boot up, one can disable any user, including root, from unmounting the filesystems.

These implementations are done by adding special flags to the inode of protected files/directories and are based on Virtual Files Systems (VFS) or File Systems (FS). Since no protection is done at the device driver level, any privileged user (e.g., a root user) can access the hard disk directly via device drivers (e.g., use `dd` command) and delete these protected files/directories. Thus this solution does satisfies all the requirements we discussed earlier.

3 Recommended solutions

In follows we propose a secure append-only storage architecture using modified magnetic disks. Magnetic disks are re-writable, and generally does not satisfy the requirements of append-only. In our solution, we propose to achieve append-only for magnetic disks using one of the following mechanisms.

1. Modify block-drivers for magnetic disks and make these disks append-only. As soon as data is written on a specific block on the magnetic disk, no one can modify that block. Special blocks on the magnetic disks will be reserved to keep track of which blocks have been written. This solution only requires changes in the operating systems and could be easily implemented.
2. Modify firmware inside the magnetic disk controllers and make these disks append-only. Similarly, certain area on the disks will be reserved to keep information on which blocks have been written.

With the second solution, since one needs to modify the firmware in the disk controllers to achieve the goal, it may not be possible in all situations (e.g., manufacturers may be reluctant to provide source codes for the firmware). However, if manufacturers could convert their magnetic disks into append-only storage systems by installing append-only firmware in the controllers of disks, we will be able to have an enhanced level of security.

Cryptographic techniques will be used to guarantee that appropriate security policies are automatically enforced in append-only storage systems. The architecture of our scheme is described in Figure 1.

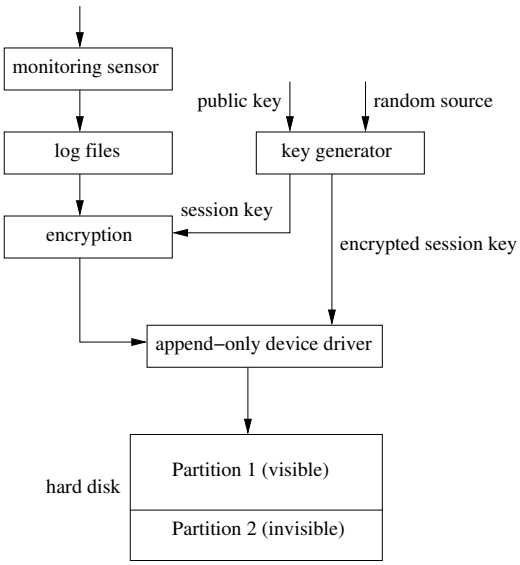

Figure 1: Architecture of append-only Storage Systems

4 Append-only storage systems based on block device drivers

For the purpose of network monitoring, the BSD Packet Filter (BPF) [6] could be used (with appropriate modification) to capture desired packets and write them to a file in the main memory. To monitor host activities, log messages generated by the kernel could be directly written to a file in the main memory. When encryption is desired, these files will be encrypted before being written to the main memory. The file size should be chosen appropriately to ensure that (1) enough data could be collected in any specified time period for the file so that the file could be promptly written to the append-only storage system; (2) files are large enough to avoid extensive disk fragmentation.

Since data written to append-only storage systems cannot to be deleted, it is possible to design more efficient file systems (e.g., more concise inode layout). However, this method may not be preferred due to the following reasons:

- New file systems need new kernel level codes development and it may increase the cost of deployment.
- New file systems may not be scalable from one operating system to another operating system.

In the following, we will present our architecture which is independent of file systems (e.g., NTFS, FAT, FAT32, ext2fs) that are used in the operating system level. The only changes to an existing system is done in the block device driver level.

4.1 Log file encryption and key management

Log files generally contain sensitive information about user activities, and should always be protected. After the log files are generated and written to the main memory, it should be encrypted before being sent to the append-only storage systems. Since these files are to be kept as permanent records, key management is a challenging issue. With different assumptions about the attackers, one could design different key management schemes. When designing such a scheme, one should be aware that if the attacker manages to get root privilege to the computer system, the attacker will be able to examine the content of the main memory and monitor the operation of the CPU. Thus the attacker may be able to get the encryption key.

Our scheme consists of public key cryptosystems and symmetric key cryptosystems. Public key cryptosystem is used for key management and symmetric key cryptosystem is used for log file encryption. The system administrators should first generate a public key and private key pair for a given public key cryptosystem. The private key should be kept in a secure place (threshold secret sharing schemes could be used to split the private key so that a certain number of persons is required to recover the private key). The computer system holds the public key. From time to time, the computer system generates a random session key for the given symmetric key cryptosystem, and encrypts the log files with this random session key. At the same time, the computer system encrypts the random session key, together with validity period and other auxiliary information, and write this encrypted session key to the append-only storage system. In order to read the data on the append-only storage system, system administrators need to recover the private key for the public key cryptosystem first. The recovered private key can then be used to decrypt the random session key for the symmetric key cryptosystem. The data on the append-only storage system could be decrypted using this session key.

If the attacker gets root privilege at some time point, then the attacker may be able to find out the random session key used at that time point and the public key. From these information, the attacker could only read the log files encrypted with this session key (if the attacker can get these files). The attacker should not be able to recover the session keys that have been used to encrypt previous log files and the session keys that will be used to encrypt future log files.

If one has enough budget for tamper-resistant hardware, one can certainly install tamper-resistant hardware to hold the session keys and to encrypt log files. Thus the attacker will not even be able to get the session key after getting root privileges.

4.2 Hard disk basics

A hard disk could be viewed as a continuous sequence of sectors, the smallest physical storage units on disks. For most disks, a sector is 512 bytes in size. Each disk sector has a factory tack-positioning label. Sector identification data is written to the area immediately before the contents of the sector and identifies the starting address of the sector.

When a disk is formatted, the most important information about the disk is written to the first sector (physical location: cylinder 0, side 0, and sector 1), which is normally called Master Boot Record (MBR).

The Master Boot Record contains the partition table for the disk and a small amount of executable code. For Intel x86-based computers, the executable code examines the partition table, and identifies the system partition (the location of the system boot sector). The MBR then loads the system boot sector into the memory and transfers the execution to the execution code in the system boot sector.

MBR consists of 446 bytes of the first sector of a disk (including the disk signature at the end of the MBR code) and 64 bytes of partition table. The partition table conforms to a standard layout that is independent of the operating system. Each partition table entry is 16 bytes long, making a maximum of four entries available. In another word, partition table 1 starts at the 446-th byte, partition table 2 at the 462-th byte, partition table 3 at the 478-th byte, and partition table 4 at the 494-th byte. The last two bytes of the first sector are a signature for the sector and are always 0x55AA.

An entry in the partition table consists of the following fields: boot indicator (one bytes, indicating whether the partition is the system partition. For example, on x86 based systems, 00 standards for non-bootable partition and 80 for system partition), starting head (one byte), starting sector (6 bits), starting cylinder (10 bits), system ID (one byte), ending head (one byte), ending sector (6 bits), ending cylinder (10 bits), relative sector (four bytes), and total sectors (four bytes).

4.3 Block devices and block device drivers

In most Operating Systems, hard disks are accessed as block devices via the block device drivers. A block device driver accesses the underlying device in multiple of the block size and usually allows random access. An important use of block devices is to support filesystems and swap files where the access is in multiple block units. I/O operations on files are first cached in the kernel's buffer cache before the device is invoked. Different operating systems may have different mechanisms for block devices. In the following, our discussion will be based on Linux operating systems.

Block device drivers are the media between operating systems and physical devices. Block device drivers present physical devices as continuous sequences of data bytes to the operating system.

A block device driver is a collection of routines that get called as various operations are performed on the devices controlled by the driver. The list of functions could include: `open()`, `release()`, `ioctl()`, `init()`, and `request()`.

I/O operations are always expensive. Thus block device drivers generally do not provide `read()` and `write()` functions directly. When a user process calls either of the `read()` or `write()` system calls related to the block device, the operating system will automatically handle it to the cache mechanisms.

When it is essential to write data to or read data from the block device, the buffer cache will add the I/O request to a queue of such requests for the corresponding device and then arrange for the `request()` function in the block device driver to be called to deal with the queue of requests.

The `request()` function reads each of the pending I/O requests in turn from the request queue and perform the physical read or write operations specified. For Linux systems, each I/O request is stored in a structure **struct request** defined in `/usr/include/linux/blkdev.h`. The general layout of a `request()` function in a device driver without an interrupt service routine looks like this:

```
static void do_request (void) {
    loop:
    INIT_REQUEST;  /* make sure there is at least one request */
    if (MINOR(CURRENT->dev) > MY_MINOR_MAX) {
        end_request(0);
        goto loop;
    }
    if (CURRENT->cmd == READ) {
        end_request(do_read());
        goto loop;
    }
    if (CURRENT->cmd == WRITE) {
        end_request(do_write());
        goto loop;
    }
    end_request(0);
    goto loop;
}
```

where `CURRENT` is a pointer to the **struct request** at the head of the request queue. The request structure layout is as follows:

```
struct request {
    int dev;   /* physical device for this request */
    int cmd;   /* command to perform (READ or WRITE) */
```

```
    int errors;
    unsigned long sector;   /* sector number to start */
    unsigned long nr_sector; /* number of sectors to read or write */
    unsigned long current_nr_sector;
    char *buffer; /* kernel memory buffer for data read or written */
    struct semaphore *sem;
    struct buffer_head *bh;
    struct buffer_head *bhtail;
    struct request *next;
}
```

4.4 Achieving append-only storage systems via device drivers

Since all operating system accesses to hard disks are through the hard disk device drivers, it is possible to re-write the block device drivers so that files written to the disks cannot be deleted even by root users.

As we have mentioned in the previous section, a disk could be partitioned into a few partitions. In our scheme, we will use hard disks with two partitions (pre-partitioned on other computers). We will also assume that all bits on partition two is set to 0's at the beginning. Partition one will be used to store files and partition two will be used to record which sectors on partition one has been written before. In order to achieve that goal that any sector on partition one is written once, one can replace the `READ` and `WRITE` parts of the `do_request` function in the block device drivers with the following pseudo code.

```
    if (CURRENT->cmd == READ) {
        if (CURRENT->sector ==1) {
            ''read sector one into CURRENT->buffer
              and replace bytes between 462 and 510
              with 0x00'';
        } else {
            end_request(do_read());
        }
        goto loop;
    }
    if (CURRENT->cmd == WRITE) {
        ''check the bits between CURRENT->sector and
          (CURRENT->sector) + (CURRENT->nr_sector) on the
          partition two, if all these bits are 0's, then
          set first_time = 1, otherwise set first_time = 0'';
        if (first_time == 0) {
            goto loop;
        } else {
            ''write all 1's to the bits  between CURRENT->sector
              and (CURRENT->sector) + (CURRENT->nr_sector) on the
              partition two'';
            end_request(do_write());
            goto loop;
        }
    }
```

The pseudo code "read sector one into CURRENT–>buffer and replace bytes between 462 and 510 with

0x00" in the READ pars of the code will filter out all information about partitions two, three, and four. In another word, any users (including root) will only see partition one on the disk.

In the WRITE parts of the code, the pseudo code "check the bits between CURRENT–>sector and (CURRENT–>sector) + (CURRENT–>nr_sector) on the partition two, if all these bits are 0's, then set first_time = 1, otherwise set first_time = 0" is used to determine whether the sectors requested for written have been written before. If some of these sectors have been written before (that is, some corresponding bits on the partition two are 1's), then the written request is ignored. Otherwise, corresponding bits on the partition two are set to 1's and execution is transferred to the code to finish the written request.

In our above example, one bit on partition two is used to represent the status of one sector on partition one. Assuming that the sector size is 512 bytes (or 4096 bits), the capacity of partition two should be at least the $\frac{1}{4096}$th capacity of the partition one.

4.5 Loop devices

A lookback device [2] in Linux is a virtual device that can be used like any other media device. A loopback filesystem associates a file on another filesystem as a complete device. The loopback devices can then be formatted and mounted as any normal block devices. To do this, the device called `/dev/loop`i is associated with the file and this new virtual devices is then mounted. The concept of loopback devices has been extensively used to achieve different goals. For example, it is used to encrypt block devices at sector level [8]. It has also been used in fibre channel Host Bus Adaptor (HBA) cards technologies. The advantage of loopback devices is that in order to achieve these goals, one does not need to hack the kernel codes. Loopback device technologies could also be used to produce append-only storage systems. However, this kind of append-only storage systems are not secure enough since any users with appropriate privilege could use low level commands such as `dd` to write the physical block device directly, thus bypassing the append-only protection mechanisms in the loopback device drivers.

Append-only storage systems based on the modification of block device drivers are more secure since even root users cannot write a sector two times.

4.6 Achieving append-only storage systems via firmware

In the previous sections, we discussed mechanisms to get append-only storage systems with device drivers. In order to achieve more security, one can embed these technologies into the controller firmware in the hard disks. For example, the firmware in the controller could make a reserved partition invisible to the users. This partition could be used to represent which sector in the visible partition has been written once. The firmware will block any endeavor to write a sector if it has already been written once.

5 Security considerations

In this paper, we introduced block device driver-based technologies to turn a normal magnetic disk into a append-only storage system. This system has the potential to be secure against not only outsider attacks but also sophisticated insider attacks. For example, an Internet attacker cannot delete the log files that he/she has left in the computer system since even if she/he has got the root privilege, she/he still cannot write a sector on the hard disk twice unless she/he could figure out the address in the main memory for the `do_request` function and modify the corresponding codes. It is generally extremely difficult to figure out these information for a specific running machine.

For dedicated attackers, they may monitor the operation of CPU and find out the location of code for the corresponding device driver in the main memory. By modifying the code for device drivers in the main memory, the attackers may be able to access the hard disk at a lower level and delete all data there. An

insider or an Internet attacker who has got the root privilege could certainly also install a new kernel onto the machine and reboot the machine. Then she/he will have a normal device driver and will be able to delete all data on the hard disk. We will assume that these kind of attacks will be hard to achieve. In cases that we have to address this kind of attacks, we recommend append-only storage systems based on firmware.

An inside attacker could also walk into the server room, take the disk out, install it on another machine, and delete all data on it. For this kind of attacks, one may require that the disk be secured with deadlocks and at least two persons need to be present to take the disk out.

References

[1] C. Antonelli, M. Undy, and P. Honeyman. The packet vault: secure storage of network data. In: *Proceedings of the USENIX Workshop on Intrusion Detection and Network Monitoring.* April 9–12, 1999.

[2] A. Bishop. The loopback root filesystem HOWTO. `http://www.linux.org/`.

[3] R. Card, T. Ts'o, and S. Tweedie. Design and implementation of the second extended filesystem. In: *Proceedings of the First Dutch International Symposium on Linux*. `http://e2fsprogs.sourceforge.net/ext2intro.html`

[4] CITI. Projects: Advanced Packet Vault. http://www.citi.umich.edu/projects/apv

[5] G. Malan, D. Watson, F. Jahanian, and P. Howell. Transport and application protocol scrubbing. In: *Proceedings of the IEEE Infocom 2000 Conference*, Tel Aviv, Isreal, 2000.

[6] S. McCanne and V. Jacobson. The BSD packet filter: a new architecture for user-level packet capture. In: *Proc. of Winter USENIX Conf.*, pages 259–269, San Diego, 1993.

[7] T. Ptacek and T. Newsham. Insertion, deletion, and denial of service: eluding network intrusion detection. 1998.

[8] R. Rhea. Loopback encrypted filesystem HOWTO. `http://www.linux.org/`.

[9] H. Xie. LIDS Hacking HOWTO. Linux Intrusion Detection Systems, `http://www.lids.org/lids-howto/lids-hacking-howto.html`