

# A User-level Framework for Auditing and Monitoring

Wu Yongzheng, Roland H. C. Yap  
School of Computing  
National University of Singapore  
{wuyongzh, ryap}@comp.nus.edu.sg

## Abstract

*Logging and auditing is an important system facility for monitoring correct system operation and for detecting potential security problems. We present an architecture for implementing user-level auditing monitors which: (i) does not require superuser privileges; (ii) makes it simple to create user defined monitors which are transparent; and (iii) provides security guarantees such as mandatory and reliable monitoring while maintaining confidentiality of setuid processes. We avoid problems of self-referential monitoring. Monitor use policies can be specified to increase flexibility. We show that our framework can be tailored so that it is very efficient with low overhead on macro and micro benchmarks. This demonstrates that it is feasible to make use of arbitrary and programmable user-level monitors for system security and auditing applications.*

## 1. Introduction and Overview

Logging and auditing are important operating system facilities used to help monitor correct system operation and to detect potential security problems. In Unix systems, logging is traditionally *application based*. The application itself controls what is being logged through the system logging mechanism `syslog`, e.g. security audit log messages generated by `login`, `su`, etc. The drawback of application logging is since it is under the control of an application which may be compromised or malicious, no security guarantees are possible. More secure versions of Unix have finer grained auditing mechanisms to satisfy the Trusted Computer System Evaluation Criteria (TCSEC) or Common Criteria (CC) security requirements. The Solaris Basic Security Module [7] for example defines kernel auditing events which can serve to log certain system calls. Such auditing is typically system-wide on all processes and requires administrator privileges.

Traditional auditing mechanisms are designed mainly for system audit trail purposes. As such, they are not sufficient for the needs of more demanding security monitoring applications such as intrusion detection systems (IDS), determining correct application behavior, detecting improper system usage, etc. In this paper, we present an approach to auditing and monitoring which is sufficiently flexible for a variety of applications. We provide a kernel extension which enables easy programming of user level (as opposed to kernel level) monitors for observing the effects of system calls made by specified processes of interest. Our philosophy is to separate mechanism from policy. A kernel-level mechanism provides transparent, secure and efficient monitoring, while the core logic and functionality is encapsulated in a user-level monitor. Having a user-space monitor means that we do not have to worry about code safety issues unlike a kernel-level one. As user-level monitors do not have to be privileged, ordinary users can create/run their own monitoring tools. We show that general purpose user-level monitors are easy to write without requiring any knowledge of kernel programming. In the remainder of this paper, we will refer to monitoring as encompassing the concept of auditing and logging.

We provide a number of security guarantees: (i) the selected processes (which can include their children) cannot circumvent monitoring, we call this *mandatory monitoring*; (ii) none of the operations/events of interest from the set of monitored processes are missed, we call this *reliable monitoring*; and (iii) the monitor cannot escalate its privileges, only precisely the operations/events of processes at the same privilege level can be monitored. The mandatory and reliable properties are necessary to ensure that a monitor can be used for security purposes. The last property is important since the user-level monitors can be unprivileged. Finally, we also require that the monitor be *transparent* to the monitored processes — thus the act of being monitored has no side effects to the monitorees. We re-

mark that traditional Unix mechanisms such as `ptrace` and `proc` do not provide these guarantees.

A key objective is that the monitoring mechanism be efficient and scalable. By efficiency, we mean that fine-grained monitoring is possible with low overheads. Scalability means that the cost of monitoring should be dependent on how much is being monitored and the amount of information desired. The cost should be controllable by the monitor so that overhead is commensurate with need. In the end, we want to be able to have several fine-grained root-level and unprivileged monitors to be permanently running without paying too high a price. On one end of the spectrum, we allow for global monitors which log all interesting events across all processes to disk like an audit log; and on the other end, the monitor might only be concerned with writes to particular system files from particular processes and then perform sophisticated analysis.

Consider the following motivating example. Suppose we want to monitor whether a web server has been attacked, perhaps as part of an IDS. The web server logs cannot be used since either the server or the logs could be compromised. A traditional auditing facility like a disk based log would have a number of problems. Firstly, there may be confidentiality issues in giving the system log to the IDS, the IDS may gain access to confidential information (assuming it isn't running as root). Another question is what happens if the disk log causes the file system to run out of space? Add a network IDS to this scenario will further strain the audit log! One could use `ptrace` to monitor the web server but this can have a significant performance penalty and may not ensure mandatory or reliable auditing.

Our prototype implementation shows that it is possible to get all these desirable features in a user-space monitor without requiring special privileges. We demonstrate an efficient implementation which has low overhead even with user-space monitors.

## 2. Related Work and Approaches

A commonly used technique for monitoring is system call tracing or system call interposition to monitor the system calls made by a process. For portability, systems like Janus [4] or Alcatraz [5] use the Unix user-level mechanisms `ptrace` or `proc` to do system call tracing. This usage is problematic because it is not meant to be a secure monitoring mechanism, e.g. `ptrace` was meant to support debuggers. In the Solaris manual pages, `ptrace` is described as being “unique and arcane”. [3] discuss these kinds of problems and common pitfalls with user-level system call interposition are discussed in [3], such as: (i) race conditions between time

of check and time of use (TOCTOU), i.e. a buffer can be modified by another thread; (ii) non-inheritance of tracing, i.e. special `strace` hacks in Linux; and (iii) not transparent with respect to `setuid/setgid` executables and signals, i.e. `ptrace` and `proc` disable tracing on `setuid/setgid` executables. Because of their subtleties and intrinsic difficulties, `ptrace` or `proc` are not suitable for general purpose user-level monitoring although they may be useful in specific situations.

The other serious drawback of `ptrace` or `proc` is that the overhead is considerable, incurring at least two context switches per traced system call. Our micro benchmarks show that this can lead to an order of magnitude slowdown on system call intensive programs. Another example is Alcatraz [5], their user-level system call interception mechanism gives an overhead of 77% on a `tar` benchmark. Thus, user-level mechanisms also have the drawback of high intrinsic overhead.

The obvious alternative is a kernel-level system call interposition, e.g. the later version of Janus used a kernel module, `mod_janus` [3]. We remark that while a kernel-level system call interposition is more efficient and can be transparent, one still has to be careful about race conditions. For example, Systrace [6] takes special care to address aliasing and atomicity.

Efficient user-level monitoring also however requires a flexible mechanism to reduce the overhead of communication and the number of context switches to the user-level monitor, so this is more than what is done in system call interposition. To further improve efficiency, we provide specifications which can be checked more efficiently than a general purpose argument analysis as in system call interposition.

Systems such as Janus and Systrace are intended for sandboxing or confinement. For example, Systrace uses a user-level policy daemon which implements the confinement policy. Efficiency is obtained by implementing some of the policy in the kernel and having a timely mechanism to invoke the user-level daemon for the other cases. Monitoring for auditing is different from confinement in that confinement uses synchronous events while auditing uses asynchronous events. Furthermore for auditing, we want to efficiently monitor large numbers of relevant events and have minimal system impact on those events which are not relevant. The scope of processes being monitored may be quite different. We can monitor a collection of processes which do not form a proper subtree in the process hierarchy sense, whereas a sandbox monitors processes which form a subtree. We also provide a choice between reliable monitoring (no events are missed) and unreliable monitoring (some events are allowed to be missed).

Systems for instrumenting and tracing the kernel

can also capture data similar to the ones which we are interested in for auditing. In particular, the new Dynamic Tracing (DTrace) facility [1] in Solaris 10, has been attracting much interest. DTrace allows almost all aspects of the kernel to be instrumented, with 30,000 probes in the kernel. DTrace uses a scripting language, D programs, which are run within the kernel to do monitoring. DTrace is extremely powerful because they run within the kernel and also because of the large number of instrumented probes. In-kernel also means that DTrace can be efficient.<sup>1</sup> To ensure safety of running in the kernel, DTrace programs are restricted and checked for safety. However, we believe that it is far safer to run monitors in user mode.

As kernel tracing mechanisms are meant for debugging and performance analysis, they differ from monitors for auditing. For example, although DTrace does integrate in privileges, it does not have the same security concerns. As kernel tracing is meant to minimize impact, DTrace does not provide reliable event monitoring and can drop events. Another difference with our framework is the auditing of file operations. Since logging access to files is a very important for many kinds of monitoring, we provide more powerful specifications on directories and subtrees. The DTrace implementation requires extensive changes to Solaris, in the words of the developers, “*it was a hell of a lot of work* – Bryan Cantrill, Solaris Kernel Development”. We feel our approach is less of a massive change to the kernel type solution, and hence simpler.

DTrace compiles D programs for a virtual machine which is emulated in the kernel. An early work is [12] which provides a high level kernel instrumentation targeted mainly at the system call interface using a Wrapper Definition Language (WSL). WSL scripts are translated into C programs and compiled into loadable kernel modules. Dynamic Probes (DProbes) [10] is similar to DTrace but for Linux. It uses the KProbes mechanism to implant probes and probe handlers. Rather than providing a high level language, probe handlers are written for a RPN-based assembly like language. SystemTap [11] is another Linux project which uses KProbes as the underlying kernel instrumenting mechanism. It uses an awk-like scripting language similar to D but uses the WSL approach of compiling into a kernel module.

### 3. The Monitor Framework

A monitor is a user-space process which audits the behavior of other processes. Monitors are described

---

<sup>1</sup>Our micro-benchmarks show that while DTrace has less impact than `proc`, the overhead is still significant.

by two specifications: (a) a process specification defining which processes to monitor; and (b) event specifications which define what operations to monitor from those processes. In what follows, we describe the design of our monitoring framework and portions of the API. The API is actually a user library which provides a convenient interface to the kernel monitoring interface. Due to lack of space, we cannot give all the underlying details but rather illustrate by example.

#### 3.1. The Monitor Process Specification

An arbitrary collection of processes, not necessarily related by parent-child relationships can be designated for mandatory monitoring. To allow for flexibility and dynamic process creation (including children), we use an API for constructing boolean expression in a functional lisp-like style which allows easy creation in C. The boolean expression is built from the following predicates using the following usual boolean operators, `AND`, `OR` and `NOT`:

1. *true/false*: For example, a global specification to monitor all processes is simply the boolean expression *true*.
2. *uid/euid/suid/fsuid* (user id): These predicates are true if and only if the user id of the process is same as the user id specified. Similar predicates are also used for group ids.
3. *pid* (process id): This predicate is true if and only if the pid of the process is same as the pid specified. This is used to include or exclude existing processes.
4. *childof*: This predicate is true if and only if the process specified by the pid is an ancestor of the current process. Note that we do not distinguish direct child processes and grandchild processes - so *childof* can specify a subtree in the process hierarchy. This can be used to include or exclude both existing processes and processes which are not yet created.
5. *executable*: This predicate is true when the executable of the process is the same as the given pathname. This can be used to include or exclude both existing processes and processes which are not yet created.

An example of the API (see also Section 3.4) is to monitor all processes owned by the user Bob except for process 1468 and its child processes.

```

proc_spec = lbox_AND(
    lbox_UID("bob"),
    lbox_NOT(
        lbox_CHILDOF(1468)));

```

Thus, the monitor can be targeted to observe only the activities of particular processes of interest, ignoring other processes. This helps to reduce monitoring overhead.

### 3.2. The Event Specification List

An event specification defines which behaviors of the monitored processes is of interest to the monitor. Suppose a monitor event expression is  $S$  and event  $e$  happens. Then  $S$  is triggered when  $e$  is an object which matches  $S$  and the operation is one which is compatible with  $S$ . The notion of matching and compatibility is specific to the type of object.

A monitor defines a list of all the events of interest. The event specification also defines the appropriate information to return when an event is triggered. One monitor might specify that a file event should consist of the inode, canonical pathname and operation type. Another monitor might only need the operation type, perhaps because the file is unambiguous and known to the monitor. This helps achieve scalability. In the previous example, it avoids the need for constructing a canonical pathname and reduces the event size.

Our events are specified on system resources or objects together with their associated operations. We feel this is more declarative than a system call approach and makes it easier to specify the events of interest. Due to lack of space, we will focus on file objects and briefly mention other important system objects. Common information which can be specified for all types of events are:

1. *time*: This gives the wall clock time of the event.
2. *pid*: This is the process which performs the operation.
3. *type*: The operation type which is specific to the object.

All event specifications can specify whether the event is asynchronous and can be buffered up, or if it is synchronous and needs to be delivered directly to the monitor. Synchronous events also flush any earlier events which have been buffered. This is analogous to the `PUSH` flag in the TCP packet and allows timely delivery of important events to the monitor.

An event can also specify that reliable monitoring is not needed — this means that the kernel can choose

to drop the event if the event buffer is currently full. This would mean that the monitored processes do not have to be suspended and thus higher throughput at the cost of losing some events.

#### File Objects:

A file event specification consists of:

1. *file pathname*: The pathname is translated into inode number and device number pair. For efficiency, this pair is used internally by the kernel as the identifier of the file.
2. *inclusion flag*: For directories, we can specify whether we are monitoring the directory itself, or all files in its subdirectories. For example, you can specify that all files under `/etc` are included by using the `SELF+SUBDIRECTORIES` flag. You can specify that only the `/etc` directory itself is included by using the `SELF` flag only.

Another flag, `IGNORE`, means that we are not monitoring this directory and its subdirectories. Thus `IGNORE` can be used for removing files in the existing file definition. This is useful because sometimes we want to audit file access *outside* some directory. For example, we are interested in the file access *outside* `/var/www` by the web server process. We can combine two file event specifications to achieve this.

- (a) `(/, SELF|SUBDIRECTORIES, R|W|X)`
- (b) `(/var/www, IGNORE, R|W|X)`

The way multiple file specifications are treated is that when more than one file event specifications matches the file access, the more specific event — the deeper file event specification takes precedence.

3. *operations*: It specifies which operations (read/write/execute) we are interested in. For example, with a read operation, a file read access event is generated when a process reads from the file but not for write operations. Note that the execute operation differs from the executable predicate in the process specification as the latter is used for process selection.

Operations on the file meta-data such as permissions and access times can be monitored. The same holds for directory operations such as file creation, deletion or renaming.

When a file is removed (i.e. its reference count becomes 0), all the corresponding file event specification are also removed. This implies that the number of

event specifications may decrease at run time. For example, suppose initially a monitor has 4 file event specifications, later there may be only 2 file event specifications. Removing the file event specification is necessary because an inode number is reusable (like a pid). When a process *A* creates a monitor with a read-only specification on the file `/tmp/a`. Later, another process *B* deletes file `/tmp/a` and creates another file `/tmp/b` which may have the same inode number of previous `/tmp/a`. If the corresponding file event specification is not removed, the monitor would get an incorrect event.

File events can specify the following information to be returned:

1. *inode number and device number*: This uniquely identifies a file in Unix systems.
2. *operation*: This is the type of operation. For example, read, execute or delete.
3. *data*: the data which is read or written — which allows the monitoring of actual I/O.
4. *canonical pathname*: A file can have many possible (absolute) pathnames because of hard links, symbolic links, the “.” component and different mount points. We thus return a canonical pathname not containing symbolic links with the same semantics as `realpath(3)`.

#### Other Objects:

Other than file objects, we can monitor events on other important system objects such as sockets and processes. Due to lack of space, we can only briefly summarize them. Socket specifications allow the monitoring of interprocess communication and networking. Process objects allow monitoring of process operations such as process creation, signals, `setuid` operations.

### 3.3. The Monitor Operational Model

In our framework, an auditing monitor is simply a user-level process which is the user-level monitor. It can be thought of as a *logging box* in an analogy to a sandbox, hence we abbreviate this as *lbox*. The `lbox_create(proc_spec, event_spec, n)` is used to designate the current process to be the lbox monitor which will receive the events generated from the processes defined by `proc_spec` which trigger the set of events defined in `event_spec`. One caveat is that we disallow self-referential monitoring, see section 4.3.

In our framework, asynchronous events are stored in a kernel buffer of size `n` defined at monitor creation time. Asynchronous events can only be received by the monitor when the buffer becomes full — this can

be thought of as flushing the buffered events. To ensure mandatory monitoring which guarantees that no events are lost, when the monitor buffer is full, the process generating the event is blocked until the monitor has emptied the buffer. Processes which are blocked because of the full buffer are also unblocked if the lbox is deleted explicitly by the monitor, `lbox_delete()`, and also implicitly when the monitor terminates. Synchronous events also cause the buffer to be flushed. The monitor can also choose to disable and then re-enable events from the event specification.

We provide a convenient API to read events one at a time, `lbox_next_event()` which abstracts out the low level details. This is simply a library function (in an analogy to `stdio`) which actually reads an event buffer up to size `n` which can contain one or more events. `lbox_next_event()` behaves like a blocking read, it blocks till either the kernel buffer has filled up and can now be read in one go or if it is flushed. A timeout can also be specified. Thus, assuming the default of asynchronous events, there are only a small number of context switches to the monitor which mostly sleeps until there are sufficient number of events. Using of synchronous and timeouts allows the monitor to have finer control and more timely event delivery but with more system overhead.

The monitor API described in this section is actually a user-space library which uses the monitoring kernel extension. Thus, this particular API is chosen to be one which is reasonably easy to write specifications in a declarative fashion in C. One could of course build a different user-space library. Even nicer would be a scripting language to make monitoring even easier, i.e. even a D-like scripting language.

### 3.4. An Extended Example

Figure 1 illustrates how a monitor is written in our system. For simplicity, we have used C-like pseudo code to hide some details and have not given the complete program. Error handling also has been omitted for the most part.

In part 1, two events are to be monitored. Any access to any files starting from the current working directory including its subdirectories (the `SUBDIRECTORIES` flag is used to mean this directory including any file with a path starting from here). We also monitor whether `bash` is being executed.

Part 2 defines the process specification which determines which processes are to be monitored by this monitor. Here we want to monitor process given by `pid` and all its children but not if it happens to be the process `inetd.pid`. Also monitor any processes which

```

/* initialize context. It will open the /proc/lbox file descriptor */
lbox_open();
/* PART 1: create events spec: current directory and executable, and information to gather */
lbox_addevent_file(event_spec, ".", SELF|SUBDIRECTORIES,
    F_R|F_W|F_X, I_INO|I_DEV|I_PID|I_ACC|I_PATH);
lbox_addevent_file(event_spec, "/bin/bash", SELF, F_X,
    I_INO|I_DEV|I_PID|I_ACC|I_PATH);

/* PART 2: create a process specification */
pid = (get root pid of process tree from argv)
proc_spec =
    lbox_OR(
        lbox_AND( lbox_OR( lbox_PROC(pid), lbox_CHILDOF(pid)),
            lbox_NOT(lbox_PROC(inetd_pid))),
        lbox_PROC(lbox_EUIDNAME("apache"))
    );

/* PART 3: now create the lbox */
lbox_create(proc_spec, event_spec, 4096);

/* PART 4: read and process events. */
for (;;) {
    /* lbox_next_event takes a timeout value, -1 means to block forever */
    event = lbox_next_event(-1);
    switch (event->type) {
    lbox_FILE_EVENT:
        event_file = (struct lbox_event_file *)event;
        printf("pid=%d, dev=%lu, ino=%lu, access=%d, path=%s\n",
            event_file->pid, event_file->dev,
            event_file->ino, event_file->access,
            event_file->path);
    }
}

```

**Figure 1. A Simple Monitor**

happen to have an effective user id which is the same as the user name “apache”. It should be obvious that this specification specifies some existing processes as well as potential future processes which are to be monitored.

Part 3 creates a lbox with the process and event specifications. A buffer of size 4096 is specified. Part 4, just collects matching events for processing by the monitor.

## 4. Security and Monitor Interactions

We now turn to security considerations. Earlier we have looked at mandatory and reliable (and non-reliable) auditing. There are two other important security components in our framework.

### 4.1. Confidentiality Considerations

As we allow non-privileged users to run their own lbox monitors, it is important to ensure system confi-

dentiality. We do not allow a non-root user to monitor processes owned by other users. A corollary of this constraint is that a non-root user  $u$  cannot monitor a setuid (setgid) executable which is setuid (setgid) to user  $w$  as long as the effective user (group) id  $w$  is different from  $u$ . The dual to this is that a monitor process belonging to  $w$  can monitor such a process from user  $u$  when its effective userid is  $u$ . This allows monitors to have separate privileges from the monitored processes.

In contrast, `ptrace` takes a different approach to maintaining security. Setuid and setgid programs are not allowed to be traced by `ptrace`. Thus, `ptrace` cannot be used for monitoring while our mechanism can still allow monitoring while maintaining confidentiality.

Consider a process  $p$  with original userid  $u$  is being monitored in a lbox where the monitor has userid  $u$ . Suppose  $p$  changes its effective userid to  $w$ , then confidentiality prevents the events in  $p$  to be monitored. To ensure mandatory monitoring, once  $p$  (or its children, if the lbox contains the children) returns its userid back

to  $u$ , the monitor can now receive monitored events.

## 4.2. The Lbox Policy Daemon

It may be the case that for non-root users, one would like to impose further restrictions on monitoring. This might be to disallow certain kinds of events from being monitored or certain processes from being monitors. It could limit monitor usage, e.g. a monitor quota. Monitor creation policy can be policed by an *lbox daemon* which is simply another user-space process. This is an extension of the same philosophy of user-level monitors but now for the construction of arbitrarily complex monitor access policies.

When a process tries to create a lbox, the kernel passes the lbox specification to the lbox daemon, and at the same time suspending the process. The lbox daemon then decides whether or not to allow or deny the request. Furthermore, the lbox daemon is able to modify the lbox specification. The suspended process can then be woken up, if the lbox is successful, it now becomes a lbox monitor with possibly a modified specification. The lbox daemon can also be turned off nor does it need to be present, in which case, a default policy is applied.

## 4.3. Cascading Monitors

It may be the case that one of the processes being monitor is itself a monitor, we call this cascaded monitors. Cascading auditing gives the maximum flexibility to create collections of independent monitors as well as an ecology of monitors. For example, the system administrator can have a global monitor to audit any processes (except itself) which tries to execute `/sbin/su`. A user might run a monitor to log all writes from some of his processes to a particular file.

A collection of monitors can be viewed as a directed acyclic graph (DAG) where the edges represent the monitoring relationship. Suppose that instead of a DAG, we have a cycle, thus a circular monitoring chain. The self-referencing monitoring will now lead to an ever increasing cascading chain of monitored events which will not terminate. For example, suppose monitors A and B are watching each other, then event  $a_1$  triggers monitor B which will generate event  $b_1$  for monitor A which will trigger yet another event  $a_2$  and so on. Thus, we require as a system safety condition no self-referential monitoring. It is interesting to note that DTrace does not prevent this, so for example, a D program which traces all the `write` I/O will also monitor the DTrace process itself. This appears to cause Solaris to freeze in the kernel.

Rather than disallowing the creation of a monitor which may lead to a cycle in the monitor DAG, we instead remove the monitors which would otherwise create a cycle from the process specification. This makes sense since a new monitor may not wish to care about monitors in the system. Furthermore, it can be easier to write a process specification which is a little looser than to write one which guarantees no cycles. For example, global monitoring using the process specification, `TRUE`, should strictly also exclude the monitor pid.

The following example shows a snapshot of a system where a web server is running and user Alice is using `vim`. The left part of Figure 2 shows the process tree induced by the parent-child relationship while the right shows the monitor DAG induced by the monitoring relationship. The following sequence describes the creation of the snapshot given in Figure 2:

1. The master web server process 4010 is created.
2. Process 4010 creates 3 worker processes: 4011, 4012, 4013.
3. The root user is logged in through ssh, creating shell(14560).
4. Root first creates a global monitor 14940 to audit file access on `/etc/passwd` for all processes (except 14940).
5. Root then creates a monitor 14941 to audit the web server processes.
6. Alice logs in and a shell 15343 is created.
7. Alice creates a monitor 15349 to audit her `vim`(15350) editor because she want to know which files are accessed by `vim`(15350).

Here, two monitors 15349 and 15340 are monitoring the `vim`(15350) process. Suppose the `vim` (15350) process accesses the `/etc/passwd` file, two file events will be generated to the two monitors, 15349 and 15340. If `vim` (15350) accesses another file, then only monitor 15349 will receive the event.

## 5. Using Monitors

We now give some simple examples of using the framework. These examples are only meant to exemplify the ease of creating a customized auditing monitor and are not meant to be full blown applications.

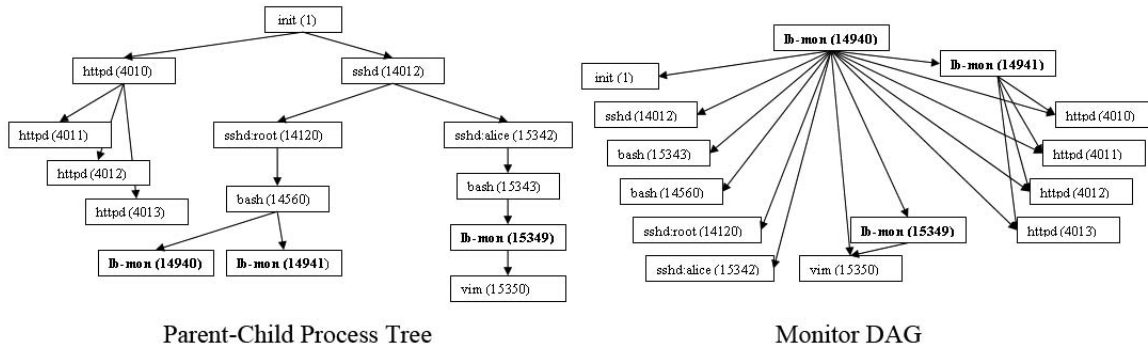


Figure 2. A Tree of Cascaded Monitors

### 5.1. Testing a possibly untrusted program

This example checks to see if a process is not stealing your PGP keys. The monitor process first creates the lbox and then a child process which executes the program to be audited.

```
proc_spec = lbox_CHILDOF(getpid());
lbox_addevent_file(event_spec,
    "/home/alice/.gnupg/secring.gpg",
    SELF, F_R, I_ALL);
// .... create the lbox
if (fork()) {
    // .... do the event processing
} else execl(...);
```

### 5.2. Monitoring the web server

In this example, we want to see whether the web server is working correctly. More precisely, we want to make sure it is only accessing files inside `/var/www`.

The process specification can be described as follows:

```
proc_spec = lbox_CHILDOF(apach_master_pid);
```

The event specification is an example of monitoring files *outside* some directory. We first make all files to be monitored, then we make all files under `/var/www` *not* to be monitored. Note that the order of the two specifications is not important.

```
lbox_add_event_file(&event_spec1, "/",
    SELF|SUBDIRECTORIES, F_R|F_W, I_ALL);
lbox_add_event_file(&event_spec2,
    "/var/www", IGNORE, F_R|F_W, I_ALL);
```

### 5.3. Global Monitoring

Sometimes you need to monitor all actions on a single process; and other times you need to monitor a single action on all processes. In this example, we want

to see if any process is sending packets to the network `137.132.0.0/255.255.0.0`. The process specification is simply,

```
proc_spec = lbox_TRUE();
```

The event specification is the following network event,

```
lbox_net_connect_ipv4(&event_spec,
    "137.132.0.0/255.255.0.0");
```

### 5.4. Other Applications

LAFS [8] is a logging and auditing file system where accesses not conforming to a policy are logged. The policy specification includes a time interval, userid specification, application and operation type. A LAFS-like monitor could easily be written. Note that it is harder to use DTrace to do this LAFS-style tracing because of its system call orientation. Here we can make use of inheritance in the directory structure.

In [2] a monitoring application is described to report when applications tasks did not run as they were supposed to, e.g. due to some failure. Again, it is easy to write a monitor which can detect whether tasks ran as scheduled.

## 6. Experimental Evaluation

Our prototype Linux implementation makes use of LSM [9] and version 2.6.10 of the kernel, and thus is convenient to install as it consists of loadable kernel modules. The interface to the kernel is done through `ioctl` to a pseudo file system in `/proc/lbox`. The lbox API library in Section 3 gives a more convenient interface than using `ioctl` system calls directly. Due to lack of space, we do not go into further details of the kernel implementation. The PC used here is a Pentium IV 3.0GHz PC with 1G memory.

We want to demonstrate that the framework and prototype system leads to rather efficient monitoring.

Although our prototype is not yet fully optimized, the results do show that the overheads of monitoring can be very low.

We first use a simple micro-benchmark which gives an indication of worse case overheads. The program performs 1000000 `open(2)` and `close(2)` calls with a monitor watching for the file access. We compare the auditing framework in each of the following scenarios:

1. *clean kernel*: A clean stock kernel.
2. *proc miss*: The `lbox` module is loaded in the kernel but no monitors are present.
3. *file miss 1/2*: A monitor is monitoring the file open but on a different file specification. Thus no event is generated. We compare two scenarios. *File miss 1*, uses a directory specification without the `SUBDIRECTORIES` property, thus no directory traversal is needed. *File miss 2* has a file specification which is some non-matching directory with the `SUBDIRECTORIES` set. This requires traversing up ancestor directories up to the root.
4. *0 dir level*: The monitor is monitoring a regular file. The benchmark is accessing the same regular file.
5. *1/2 dir levels*: The monitor is monitoring a directory which is 1 or 2 levels deep containing the file which is opened. For example, the monitor uses the file specification `/1` with `SUBDIRECTORIES` set. The 2 dir level benchmark opens the file `/1/2/foo`. This test investigates the time for directory traversal.
6. *no buff*: The file event is synchronous which means 1000000 context switches to the monitor are needed. The benchmark process suspends until the monitor has read the event.
7. *ptrace*: The comparison is with `strace` which uses the traditional `ptrace` mechanism in Linux.

The open micro-benchmark results are given in Table 1 with all times in seconds. The average and standard deviation is given over 10 runs. As the timings are only measured with the Unix user/system and real-time mechanism, they are only approximate and are meant to give an indication of the overheads of monitoring under the different scenarios.

The kernel inspection implementation (*proc miss*) has negligible overhead ( $\sim 2\%$ ) over the clean kernel. The *file miss 2* test has ( $\sim 14\%$ ) overhead, this is because that the system needs to travel to the root directory to make sure the file is not monitored. The *file*

scenario	real	user	sys
clean kernel	1.99 ± 0.01	0.21 ± 0.01	1.77 ± 0.02
proc miss	2.04 ± 0.02	0.21 ± 0.01	1.82 ± 0.03
file miss 1	2.17 ± 0.01	0.22 ± 0.01	1.95 ± 0.02
file miss 2	2.27 ± 0.01	0.22 ± 0.01	2.04 ± 0.02
1 dir level	2.29 ± 0.01	0.21 ± 0.01	2.03 ± 0.02
2 dir levels	2.52 ± 0.01	0.21 ± 0.01	2.23 ± 0.03
3 dir levels	2.56 ± 0.01	0.20 ± 0.02	2.31 ± 0.02
no buff	8.70 ± 0.04	0.99 ± 0.08	4.57 ± 0.16
ptrace	59.04 ± 0.15	12.90 ± 0.32	46.11 ± 0.39

**Table 1. Open micro-benchmark**

*miss 1* scenario has a smaller overhead ( $\sim 9\%$ ) because directory traversal is not needed. When monitoring is synchronous, events are not buffered (*no buff*), overhead jumps substantially to 337%. It is interesting to note that this is still much smaller than `ptrace` which has 2866% overhead or seven times slower than the *no buffering* case. Using asynchronous events with buffering (0 dir level) drops the overhead to 15%.

One point of comparison with other systems on Linux would be with Systrace [6]. A pure comparison is not valid since in our system any event which must be monitored is eventually passed to the monitor. Systrace on the other as it does access control first can choose between a kernel-level policy or the user-level policy daemon. The kernel-level policy should entail less work for Systrace simply because the decision is handled only within the kernel, while the user-level policy is more expensive simply due to the increased number of context switches. The objectives of the two systems are also different. With those caveats in mind, the Systrace open micro-benchmark shows a 6.25% overhead over no monitoring. Our overhead for *file miss 1* is a little more at 9%. This makes sense since for auditing, an event has to cross in two directions: first inwards when it is recorded; and later back to user-space when it is sent to the monitor. Our implementation is not yet optimized and we expect also to be able to reduce the overheads further.

Our directory file specifications can be compared against the pathname normalization of Systrace. The “0-2 dir levels” scenario shows that the overhead of an inherited file specification is small, the initial overhead of checking a directory is reasonable at 26% and then a rather small overhead for the next component. The Systrace results show that each directory component in their specification adds about 1665% additional overhead over the original `open()` system call. This is because of the filename normalization done in user-space which is significantly more expensive.

<sup>2</sup>The D program is “`syscall::open:entry { @[execname] =`

scenario	real	user	sys
direct run	3.84 ± 0.01	1.04 ± 0.01	2.80 ± 0.01
truss	100.32 ± 0.85	10.12 ± 0.05	56.71 ± 0.80
DTrace <sup>2</sup>	8.41 ± 0.02	1.08 ± 0.01	7.33 ± 0.01

**Table 2. Open micro-benchmark on Solaris 10**

Table 2 shows the test on Solaris for the same hardware with the same benchmark program. While the baseline for Solaris is slower than Linux, it is reasonable to make relative comparisons. `Truss` which uses the `proc` adds 2510% overhead to the `open()` system call and DTrace is significantly more efficient with 119% overhead. The use of `proc` allows only the `open()` system call to be traced rather than all system calls. Even so, we see that the overhead is considerably higher than in Linux where `ptrace` traces every system call. In the DTrace test, we have allowed DTrace more advantage as the D program does not return any information to user space, whereas in our system, all the relevant `open()`'s are being returned to the user space monitor. The absolute running time in the DTrace test is 3.67 times of that in our system. The overhead added to the original system call in DTrace is 7.9 times of that in our system. This is slightly surprising since DTrace uses dynamic instrumentation (which is hardware dependent) while we use a simpler static instrumentation which is more expensive but portable. Perhaps it is due to Linux having less overheads than Solaris.

The `open` micro-benchmark gives a measure of worst-case overhead since the file is totally cached in memory and in that case, the Linux `open` code is heavily optimized. It is also useful to look at applications (macro-benchmarks which do more than just system calls) which may have moderate to moderately heavy system call usage. Obviously, there is little point benchmarking CPU bound applications which only have low system call use. In the macro-benchmarks, we also look at the cost of monitoring I/O (reading and writing), this is meant to simulate a monitor which might want to examine precisely what an application has done. We have chosen three application benchmarks which are realistic applications

- `apache web server`: this is chosen because web server performance is important. For apache, we measure the average number of requests served per second using the apache `ab` benchmark and I/O monitoring is for the HTTP requests (reading) and HTTP responses (writing).
- `make bash`: this builds the bash shell.

---

count(); }”

- `install mozilla`: this installs mozilla. The Linux and Solaris differ. The Linux one has a custom install program while the Solaris version is simply `untar`. This means the results are not comparable except in a relative to each operating system alone.

The last two benchmarks, `bash` and `mozilla`, have also been chosen because these applications have been used in Alcatraz. For I/O monitoring, we have monitored all the I/O from the application. The results given in Table 3. The results for `strace` and `truss` are only meant to illustrate the additional overhead of I/O since both programs do re-process the data and thus do have additional overhead but serves to bound the cost of `ptrace` and `proc`. We see that in all cases, `lbox` monitoring has very low overheads, below 2% without I/O and less than 4% with complete I/O monitoring of the applications. The `mozilla` install cannot be directly compared since the actual install is different. Even though these are macro-benchmarks, overhead for `ptrace` without I/O is significant, at least 15%. With I/O monitoring, the overhead shoots up. Alcatraz [5] which uses `ptrace` has rather high overheads for their system call interception, 43% for a `make` and 79% for `mozilla`. Their isolation overhead which is a measure of write I/O adds a further 20%. Our overheads are much smaller (< 4%) but we don't do isolation.

It is a little surprising that our user-level auditing framework can out perform DTrace which is an in-kernel tracing mechanism. This is very encouraging since further optimizations are possible.

## 7. Conclusion

We show a user-level auditing framework suitable for general purpose auditing and security monitoring. We achieve transparent auditing at a fine-grained level of applications while providing guarantees on the security of the auditing process. We are careful to avoid problems with privilege escalation and access to information beyond the user's privileges. Furthermore, we avoid problems associated with denial of service which can be caused by self-referential monitoring or tracing.

As our monitors are user-level, the auditing is expressed in terms of operations on operating system objects and resources. This makes it easy to write a custom monitor since the semantics is close to that of user code rather than having to understand kernel internals.

A user-level monitor is desirable given it is more safe than an in-kernel one. The question is whether a user-level monitor is sufficiently efficient. Our framework is designed so that the cost of monitoring is commensurate with the amount of events and information needed.

	web server (request/sec)	make bash (sec)	install mozilla (sec)
Linux clean	8903.1 ± 21.8	34.3 ± 0.2	1.8 ± 0.1
lbox no I/O, file miss	8773.1 ± 19.4(1.5%)	34.4 ± 0.2(0.17%)	1.8 ± 0.0(0%)
lbox no I/O, file hit	8762.0 ± 25.6(1.6%)	34.4 ± 0.2(0.17%)	1.8 ± 0.0(0.56%)
lbox with I/O	8728.9 ± 18.4(2.0%)	34.4 ± 0.2(0.29%)	1.7 ± 0.1(3.9%)
strace no I/O	3577.6 ± 8.5(148.9%)	39.6 ± 0.1(15.3%)	2.2 ± 0.1(21.1%)
strace with I/O	1981.4 ± 6.9(349.3%)	100.7 ± 0.1(293.2%)	23.2 ± 0.1(1189.4%)
Solaris clean	6889.1 ± 42.6	43.8 ± 0.1	1.6 ± 0.1
DTrace no I/O	6776.2 ± 53.6(1.7%)	44.4 ± 0.1(1.4%)	1.6 ± 0.1(0%)
DTrace with I/O	6326.9 ± 52.4(8.9%)	44.5 ± 0.0(1.6%)	1.6 ± 0.0(0.6%)
truss no I/O	1382.0 ± 2.0(398.5%)	70.9 ± 0.0(61.8%)	7.7 ± 0.0(372.2%)
truss with I/O	1126.7 ± 4.4(511.5%)	74.6 ± 1.5(70.2%)	13.4 ± 0.13(724.7%)

**Table 3. Macro-benchmarks**

Our experiments are very encouraging showing that the overhead is comparable to in-kernel mechanisms such as DTrace.

Future work includes further system optimizations to make the framework even more scalable and efficient. We would like to increase the expressiveness of the in-kernel mechanisms without incurring more cost and also avoiding executing monitor code in the kernel.

## References

- [1] B.M. Cantrill, M.W. Shapiro and A.H. Leventhal, “Dynamic Instrumentation of Production Systems”, USENIX Annual Technical Conference, 15–28, 2004.
- [2] J. Finke, “Process Monitor: Detecting Events That Didn’t Happen”, USENIX Large Installation Systems Administration Conference, 2002. 145–154, 2002.
- [3] T. Garfinkel, “Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools”, Network and Distributed Systems Security Symposium, 163–176, 2003.
- [4] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer, “A Secure Environment for Untrusted Helper Applications”, USENIX Security Symposium, 1–14, 1996.
- [5] Z. Liang, V.N. Venkatakrishnan, and R. Sekar, “Isolated Program Execution: An Application Transparent Approach for Executing Untrusted Programs”, Annual Computer Security Applications Conference, 182–191, 2003.
- [6] N. Provos, “Improving Host Security with System Call Policies”, USENIX Security Symposium, 257–272, 2003.
- [7] Sun Microsystems, “System Administration Guide: Security Services”, part IV: Auditing and Device Management.
- [8] C. Wee, “LAFS: A Logging and Auditing File System”, Annual Computer Security Applications Conference, 231–240, 1995.
- [9] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman, “Linux Security Modules: General Security Support for the Linux Kernel”, USENIX Security Symposium, 17–31, 2002.
- [10] S. Bhattacharya, “Dynamic Probes - Debugging by Stealth”, Linux.Conf.Au, 2003.
- [11] V. Prasad, W. Cohen, F. Eigler, M. Hunt, J. Keniston, B. Chen, “Locating System Problems Using Dynamic Instrumentation”, Linux Symposium, vol. 2, 57–72, 2005.
- [12] T. Fraser, L. Badger, M. Feldman, “Hardening COTS Software with Generic Software Wrappers”, IEEE Symposium on Security and Privacy, 2–16, 1999.