Operating System Auditing and Monitoring – Thesis Proposal

Wu Yongzheng*
School of Computing
National University of Singapore

Abstract

Auditing and monitoring is an important system facility to aid in monitoring correct system operation and as a means of detecting security problems. We present a monitoring architecture which allows one to examine the behavior of programs and the operating system to quickly identify the root causes of program error or performance bottlenecks. The system should be reliable, extensible and efficient. That is: (i) the system can be safely used by not only the super user but also any normal user; (ii) programmers can write different auditing programs for different tasks; (iii) auditing should not bring too much performance overhead. We have designed and implemented a prototype which allows one to write monitors to receive all the interesting events based on an event specification and a process specification. The monitor itself is simply an arbitrary process which receives events and deals with them. Although the prototype is currently neither complete nor optimized, we can already compare our system with some related existing systems. Preliminary experimental results show that the current implementation is 8 to 200 times faster than other monitoring systems.

*Under the supervision of A.P. Roland H.C. Yap

Contents

1 Introduction

Logging and auditing is an important system facility to aid in monitoring correct system operation and as a means of detecting security problems. Traditionally logging in Unix systems is *application based*. The application itself controls what is being logged through the system logging mechanism `syslog`. For example, security auditing would log messages generated by `login`, `su`, etc. Application logging tends to be at a high level of abstraction and entirely subject to what the particular application wishes to do. 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. For example, Solaris Basic Security Module[3] defines kernel auditing events which can serve to log certain system calls. Auditing becomes more and more important as operating systems and software application become more and more complicated. Advanced auditing mechanisms has been developed on modern operating systems. DTrace[4][5], the new auditing system in Solaris 10, is considered as one of the most significant new features in Solaris 10.

There is a wealth of work in the management of logs and their analysis and processing. However there is less work in flexible mechanisms for creating flexible monitors for auditing or logging at the kernel level. We are interested in the monitoring of actions which relate to the system calls executed by collections of processes, hence *process-based*. What we would like to monitor are the "interesting events" which are the result or side-effect of an action to the system from a system call. This is different from simply logging the system call themselves. The processes which are being monitored can be defined using a general specification of the processes of interest which tracks both children and brand new processes.

We develop an architecture and mechanism which allows one to write monitors which will receive all the interesting events which have been specified for a specified collection of processes. In this project, we are focus on the auditing and monitoring mechanism. A monitor is simply an arbitrary process which receives events and deals with them as it sees fit. It does not have to be a privileged process.

Consider the following motivating example. Suppose we want to monitor whether a web server has been attacked. This could be part of an intrusion detection system. Obviously the web server logs cannot be used for this purpose since they could be compromised. A traditional auditing facility like a disk based log would have a number of problems. Firstly, there may be confidentiality issues since by giving the system log to the IDS, the IDS may have access to information it shouldn't have (assuming it is not running as root). There could be denial of service attacks against a disk log to cause the file system to run out of space. The overhead of updating the log continually may be very high, as we shall see, the overhead of non-file based mechanisms is already rather high. Now suppose, we want to also add a network IDS to this scenario and have the monitor scan the network traffic. This will further strain the audit log even more! It should be clear by now that what is needed is a transparent monitoring

facility which does not affect what is being monitored together with the special needs for auditing and security.

Another example is that suppose we want to be able to verify whether file access by the ftp demon follows the correct security policy. One should be able to write a monitor which is separate from ftpd. The ftp monitor can then choose to create a log, do statistics, perform compression of the log, etc. Ideally given a sufficiently efficient mechanism, the monitor should be user space to give maximum flexibility.

Although we mostly talk about auditing and logging, our work is not limited to auditing. We can apply the the same mechanism to access control systems. The limitations of the UNIX security model have created much interest in alternate paradigms. We can implement fine grained sandbox mechanism to confine processes to have "least privilege". This will be further discussed in section 5.2.

In the following sections, we first compare some existing auditing and monitoring works. We then propose the design objectives of our auditing system. After that we introduce our current design and implementation, *logbox*[19], together with some preliminary experimental results. Finally, we point out some directions for the future work.

2 Related Work

2.1 Overview

We can classify the related works into 3 categories, *kernel mechanism*, *sandboxing system* and *tracing mechanism*. Kernel mechanisms such as `ptrace(2)`, `/proc` and the *Light-weight Auditing Framework* provide a general instrumentation method at the kernel level. It is possible to use these methods to implement our auditing system. Sandboxing systems such as *Janus*[14] and *Systrace*[2] are used for confining programs so that they no not harm the computer. Tracing mechanisms such as *SystemTap*[11] and *DTrace*[5][4] are used for observing program and kernel behavior. In contrast to sandboxing system, tracing mechanism does not affect the running program.

2.2 Traditional syslog

Syslog is the main logging tool on a UNIX systems. It centrally manages logging for your main system services. Syslog is discretionary in such a sense that it requires the program to actively call the `syslog(2)` system call in order to generate a log message. Syslog is implemented in two halves. One half is a set of standard C-library routines. Programs utilize these routines to send messages to the log daemon. The other is a daemon process `syslogd`. It is configured with the `/etc/syslog.conf` file. It writes log messages to various locations.

When writing kernel code, one can send log messages to syslog by calling `printk`[1] in kernel. This is realized through `klogd`, the kernel log daemon.

[1] `printk` is Linux kernel specific

`klogd` is a system daemon which intercepts and logs Linux kernel messages. It is usually configured to forward kernel messages to `syslogd`.

Traditional logging mechanism such as syslog is discretionary in nature, thus cannot be used for security purpose. In the security monitoring context, we should not trust the monitored program, thus the log message supplied by it should also not be trusted. For example, the application does not log or logs something to spoof the monitor.

2.3 ptrace

The `ptrace(2)` system call provides a means by which a parent process may observe and control the execution of another process, and examine and change its core image and registers. It is primarily used to implement breakpoint debugging and system call tracing. Most UNIX systems, such as Solaris, FreeBSD, and Linux implement this system call.

While being traced, the child will stop each time a signal is delivered, even if the signal is being ignored. The parent will be notified at its next `wait(2)` and may inspect and modify the child process while it is stopped. The parent then causes the child to continue, optionally ignoring the delivered signal (or even delivering a different signal instead).

There are many disadvantages on `ptrace` as an system auditing tool. Firstly, it is very inefficient. Since the child process will stop at each system call, there will be a high process switch cost on each system call. Further more, the process still has to stop at every system call even if we only want to audit one specific system call. Secondly, if the process is already been traced by some process. It is not possible for another process to trace it. This is not desirable as an auditing tool, because sometimes we need different processes to audit on different things. Lastly, `ptrace` will bring side-effect to the process been traced. For example, if a process is attached to with `PTRACE_ATTACH`, its original parent can no longer receive notification via wait when it stops, and there is no way for the new parent to effectively simulate this notification. This is also not desirable because we want the process behave the same as while not been traced.

`ptrace` suffers from many race condition problems[8]. One could write a simple C program as in Figure 1 to escape from `ptrace`ing. After several iterations, some child processes will not be traced by the tracing program.

Many Linux kernel exploits make use of `ptrace` bugs. The famous ones are "Linux Non-Readable File Ptrace Vulnerability" in 2000, "Linux Ptrace/Setuid Exec Vulnerability" and "Linux ptrace/execve Race Condition Vulnerability" in 2001.

2.4 Solaris /proc

Solaris `/proc`[16][2] is a file system that provides access to the state of each process in the system. `/proc` can be used for system call interposition.

[2]Different UNIX systems implement different `/proc` file systems. Solaris `/proc` is one of the most powerful implements.

```
#include <sys/types.h>
#include <sys/times.h>
#include <unistd.h>

main()
{
   struct tms timing;
   int i;

   getpid();
   if (fork() == 0) {
      getppid();
      if (fork() == 0) {
         times(&timing);
         _exit(1);
      }
      for (i=0; i < 10; getpid()) ;
   }
}
```

Figure 1: Simple C program to escape ptrace

The name of each entry in the `/proc` directory is a decimal number corresponding to a process-ID. These entries are themselves subdirectories. Access to process state is provided by additional files contained within each subdirectory. The owner of each `/proc` file and subdirectory is determined by the user-ID of the process.

There are many files under each `/proc/pid` directory as shown in Figure 2. Most files describe process state and can only be opened for reading. Address space files contain the image of the running process and can be opened for both reading and writing. An open for writing allows process control; a read-only open allows inspection but not control.

In general, more than one process can open the same `/proc` file at the same time. Exclusive open is an advisory mechanism provided to allow controlling processes to avoid collisions with each other. A process can obtain exclusive control of a target process, with respect to other cooperating processes, if it successfully opens any /proc file in the target process for writing while specifying `O_EXCL` in the `open(2)`. Such an open will fail if the target process is already open for writing.

Data may be transferred from or to any locations in the address space of the traced process by applying `lseek(2)` to position the as file at the virtual address of interest followed by `read(2)` or `write(2)` (or by using `pread(2)` or `pwrite(2)` for the combined operation). The address-map file `/proc/pid/map` can be read to determine the accessible areas (mappings) of the address space.

A tracing process writes specially-formatted commands to the ctl file to specify events of interest and actions to be applied to the tracing process. These

```
$ ls -l /proc/12599
total 3543
-rw------- 1 atp users 1794048 Mar 18 20:48 as
-r-------- 1 atp users 152     Mar 18 20:48 auxv
-r-------- 1 atp users 32      Mar 18 20:48 cred
--w------- 1 atp users 0       Mar 18 20:48 ctl
lr-x------ 1 atp users 0       Mar 18 20:48 cwd ->
dr-x------ 2 atp users 1056    Mar 18 20:48 fd
-r--r--r-- 1 atp users 120     Mar 18 20:48 lpsinfo
-r-------- 1 atp users 912     Mar 18 20:48 lstatus
-r--r--r-- 1 atp users 536     Mar 18 20:48 lusage
dr-xr-xr-x 3 atp users 48      Mar 18 20:48 lwp
-r-------- 1 atp users 1728    Mar 18 20:48 map
dr-x------ 2 atp users 544     Mar 18 20:48 object
-r-------- 1 atp users 2048    Mar 18 20:48 pagedata
-r--r--r-- 1 atp users 336     Mar 18 20:48 psinfo
-r-------- 1 atp users 1728    Mar 18 20:48 rmap
lr-x------ 1 atp users 0       Mar 18 20:48 root ->
-r-------- 1 atp users 1440    Mar 18 20:48 sigact
-r-------- 1 atp users 1232    Mar 18 20:48 status
-r--r--r-- 1 atp users 256     Mar 18 20:48 usage
-r-------- 1 atp users 0       Mar 18 20:48 watch
-r-------- 1 atp users 2736    Mar 18 20:48 xmap
```

Figure 2: List of the /proc/pid directory

commands appear similar to the arguments given to the `ptrace(2)` call: an identifier indicating the requested control operation and any argument required for that operation. We will describe some commands which are related to system auditing.

1. *PCSTOP*: When applied to the process control file, PCSTOP directs the process to stop and waits for it to stop.

2. *PCRUN*: Make a process runnable again after a stop.

3. *PCSTRACE*: Define a set of signals to be traced in the process. The receipt of one of these signals by a process causes the process to stop.

4. *PCKILL*: If applied to the process control file, a signal is sent to the process with semantics identical to those of `kill(2)`.

5. *PCSENTRY PCSEXIT*: These control operations instruct the process to stop on entry to or exit from specified system calls. When entry to a system call is being traced, a process stops after having begun the call to the system but before the system call arguments have been fetched from the process. When exit from a system call is being traced, a process stops on completion of the system call just prior to checking for signals and

returning to user level. At this point, all return values have been stored into the process' registers.

6. *PCWATCH*: Set or clear a watched area in the controlled process from a `prwatch` structure operand. A watch-point is triggered when a process in the traced process makes a memory reference that covers at least one byte of a watched area and the memory reference is as specified in `pr_wflags`. When a process triggers a watch-point, it incurs a watch-point trap.

Comparing to `ptrace(2)`, `/proc` gives a finer grained control over monitored system calls. We can specify only the interested system calls to intercept. This can give a great performance improvement if we want to monitor a small set of system calls. Another advantage over `ptrace(2)` is that a process can be traced by more than one tracing process using `/proc`. However, `/proc` still suffers from performance penalty caused by process switch on each traced system calls. It requires the tracing process' immediate response on each traced system calls. As a result, it can be used for process confining but not good for auditing.

As well as `ptrace(2)`, `/proc` will still bring side-effect to the traced process. When a traced process calls `setuid(2)`, the call will fail because the tracing process would have insufficient privileges to the `setuid`ed process.

2.5 Janus

Intuitively, Janus[14] can be thought of as a firewall that sits between an application and the operating system, regulating which system calls are allowed to pass. This is analogous to the way that a firewall regulates what packets are allowed to pass. Another way to think about Janus is as an extension of the OS reference monitor that runs at user level. Concretely, Janus consists of `mod_janus`, a kernel module that provides a mechanism for secure system call interposition, and `janus`, a user-level program that interprets a user-specified policy in order to decide which system calls to allow or deny.

We will look at the lifetime of a program being run under Janus:

1. At startup, `janus` reads in a policy file that specifies which files and network resources it will allow access to.

2. `janus` then forks, the child process relinquishes all of its resources (closes all of its descriptors, etc.), and the parent attaches to the child with the tracing interface provided via `mod_janus`. At the user level, this consists of attaching a file descriptor to the child process. `janus` then selects on this descriptor and waits to be notified of any interesting events.

3. The child execs the sandboxed application.

4. All accesses to new resources via `open`, `bind` etc. is first screened by Janus to decide whether to allow the application access to the descriptor for the resource.

5. The program continues to run under Janus's supervision until it voluntarily ends its execution or is explicitly killed by Janus for a policy violation. If a sandboxed process `fork`s, its new children will have new descriptors attached to them, and will be subjected to the security policy of their parents by the `janus` process.

To further examine how Janus screens system calls, let us consider the sequence of events that occurs when a sandboxed process attempts the call `open("foo")` as described in Figure 3.

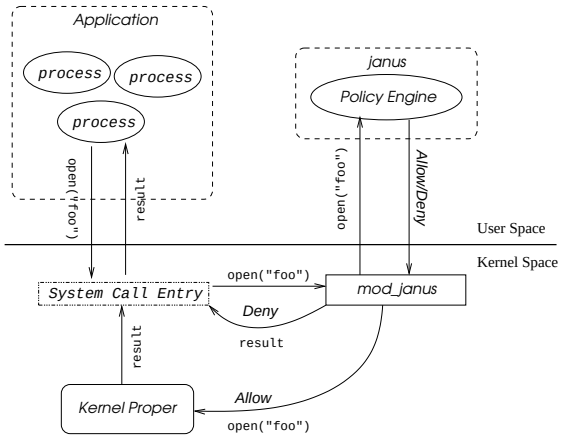

Figure 3: System Call Interposition in Janus

1. A sandboxed process makes a system call open("foo"); this traps into the kernel at the system call entry point.
2. A hook at the system call entry point redirects control to `mod_janus`, since open is a sensitive system call.
3. `mod_janus` notifies `janus` that a system call has been requested and puts the calling process to sleep.
4. `janus` wakes up and requests all relevant details about the call from `mod_janus`, which it uses to decide whether to allow or deny the call. It then notifies `mod_janus` of its decision.
5. If the call is allowed, control is returned to the kernel proper and system call execution is resumed as normal. If `janus` decides to deny the call, an error is returned to the calling process and the system call is aborted.

2.6 Sandbox

The purpose of the Sandbox[13] project is to design and implement a system call API that serves as a simple yet powerful mechanism for confining untrusted programs. The code is implemented as a patch for Linux kernel 2.4.21.

The sandbox system call API is conceptually similar to `chroot(2)` and BSD jails. However, it differs because it is has much greater flexibility and expressive power. Here are some of its key features:

1. Sandboxes may be safely created and manipulated by either trusted or untrusted users and programs.

2. Sandboxes may be dynamically reconfigured at runtime. Any changes made to the privileges associated with a given sandbox have an immediate effect on processes executing inside.

3. Privileges are specified in a fine-grained manner, and are grouped into components that serve as modular, reusable building blocks for construction of customized protection domains. Components group related privileges together according to different areas of system functionality (file access, device I/O, network access, signals, etc.).

4. A set-theoretic approach to privilege specification has been taken. Specifically, components are viewed as sets of privileges that may be manipulated using set-theoretic primitives. Given any two components x and y of the same type (i.e. two file system components), a new component may be constructed that represents any of the following:

   (a) the union of the sets of privileges represented by x and y

   (b) the intersection of the sets of privileges represented by x and y

   (c) the complement of the set of privileges represented by either x or y

   This allows construction of protection domains whose privileges satisfy assertions relative to each other given by arbitrary set-theoretic primitives.

5. Sandboxes may be nested hierarchically for simultaneous enforcement of security policies at different levels of granularity. Privileges at any level of a hierarchy may be manipulated independently. Sandboxes may be used as a means of simultaneously implementing both mandatory and discretionary access controls.

6. Sandboxes may be configured so that a sandboxed process will block when it attempts certain actions rather than being immediately denied the required privileges. When a process blocks in this manner, an event is generated and placed in an event queue associated with the sandbox. A process with authority over the sandbox may retrieve these events and decide individually which actions to allow. The set of attempted actions that trigger the blocking mechanism may be specified in a fine-grained manner using the set-theoretic primitives described above, and is dynamically reconfigurable at runtime.

2.7 Systrace

`Systrace`[2] is a process confinement facility which can also be used as an auditing tool. It can be used for both confining a privileged process and elevating an unprivileged process. Systrace enforces system call policies for applications by constraining the application's access to the system. The policy is generated interactively. Operations not covered by the policy raise an alarm, allowing an user to refine the currently configured policy. It has a GUI monitor program to manage policies and is able to modify policies at run time.

Systrace has been integrated into NetBSD and OpenBSD. With Systrace, a system administrator can say which system calls can be made by which programs and how those calls can be made. Proper use of Systrace can greatly reduce the risks inherent in running poorly-written or exploitable programs. Systrace policies can confine users in a manner completely independent of Unix permissions. You can even define the errors that the system calls return when access is denied, to allow programs to fail in a more proper manner. Proper use of Systrace requires a practical understanding of system calls, what programs must have to work properly, and how these things interact with security.

Systrace denies all actions that are not explicitly permitted and logs the rejection to syslog. If a program running under Systrace has a problem, you can find out what system call the program wants and decide if you want to add it to your policy, reconfigure the program, or live with the error.

Each Systrace policy file is in a file named after the full path of the program, replacing slashes with underscores. For example, the policy file for `/usr/sbin/named` is `usr_sbin_named`. The file starts with:

```
# Policy for named that uses named user and chroots to /var/named
# This policy works for the default configuration of named.
Policy: /usr/sbin/named, Emulation: native
```

The `Policy` statement gives the full path to the program this policy is for. You can't fool Systrace by giving the same name to a program elsewhere on the system. The remaining lines define a variety of system calls that the program may or may not use. For example:

```
native-accept: permit
native-bind: sockaddr match "inet-*:53" then permit
native-chdir: filename eq "/" then permit
native-chdir: filename eq "/namedb" then permit
native-chroot: filename eq "/var/named" then permit
native-fsread: filename eq "/" then permit
native-fsread: filename eq "/dev/arandom" then permit
native-fsread: filename eq "/etc/group" then permit
```

2.8 Light-weight Auditing Framework

The lightweight auditing framework is intended to be a way for the kernel to get various types of audit information out to user space without slowing things

down, especially when auditing is not being used. The framework is meant to serve as a complement to SELinux; it is already being shipped as a part of the Fedora Core 2 test 2 kernel.

There are two kernel-side components to the audit code. The first is a generic mechanism for creating audit records and communicating with user space. All of that communication is performed via netlink sockets; there are no new system calls added as part of the audit framework. Essentially, a user-space process creates a `NETLINK_AUDIT` socket, writes `audit_request` structures it, and reads back `audit_reply` structures in return.

The generic part of the audit mechanism can control whether auditing is enabled at all, perform rate limiting of messages, and handle a few other tasks. On the kernel side, it provides a `printk()-like` mechanism for sending messages to user space. This code also implements a user-specified policy on what happens if memory is not available for auditing; truly paranoid administrators can request that the kernel panic in such situations.

2.9 DProbes and SystemTap

DProbes[10] and SystemTap[11] are runtime instrumentation systems in Linux which can be used to diagnose complex performance or kernel debugging problems without kernel rebuilds or system reboots. They both use KProbes[9] infrastructure to dynamically instrument the kernel. KProbes allow programmers to insert break points and watch points (also referred to as probe points in general) into running programs. Probe points can be inserted anywhere in the running code. When a probe point is hit, the corresponding probe handler, which is provided by the programmer, is called. KProbes heavily depends on processor architecture specific features and uses slightly different mechanisms depending on the architecture on which it's being executed. KProbes has merged into the official Linux kernel and is available on the following architectures: ppc64, x86_64, sparc64 and i386.

DProbes and SystemTap are built on top of KProbes and allow programmers to manipulate probe points using scripting languages. The difference between DProbes and SystemTap is that the DProbes scripting language uses Reverse Polish Notation (RPN), while SystemTap is similar to `awk`[12] and DTrace[5]'s `D` language. SystemTap uses a simplified C-like syntax, lacking types, declarations, and most indirection, but adding associative arrays and simplified string processing.

2.9.1 Architecture of KProbes

Figure 4 describes the architecture of KProbes. On the x86, KProbes makes use of the exception handling mechanisms and modifies the standard breakpoint, debug and a few other exception handlers for its own purpose. Most of the handling of the probes is done in the context of the breakpoint and the debug exception handlers which make up the KProbes architecture dependent layer. The KProbes architecture independent layer is the KProbes manager which is

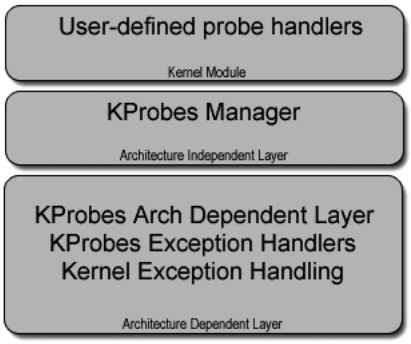

Figure 4: KProbes Architecture

used to register and unregister probes. Users provide probe handlers in kernel modules which register probes through the KProbes manager.

2.9.2 KProbes Manager

The KProbes Manager is responsible for registering and unregistering KProbes. Each probe is described by the `struct kprobe` structure and stored in a hash table hashed by the address at which the probe is placed. Access to this hash table is serialized by the spinlock `kprobe_lock`. This spinlock is locked before a new probe is registered, an existing probe is unregistered or when a probe is hit. This prevents these operations from executing simultaneously on a SMP machine.

2.9.3 Execution flow

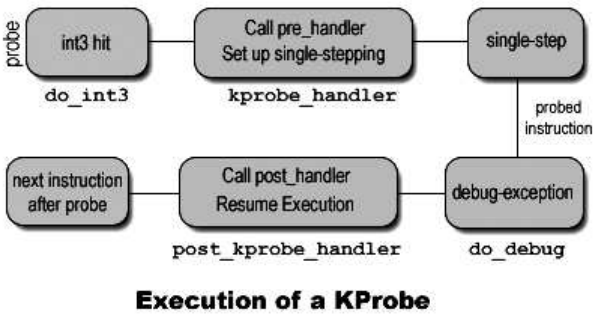

Figure 5: Kprobe Execution Diagram

Figure 5 describes execution flow when a KProbe is hit. The steps involved in handling a probe are architecture dependent. After the probes are registered,

the addresses at which they are active contain the breakpoint instruction (`int3` on x86). As soon as execution reaches a probed address the `int3` instruction is executed, causing the control to reach the breakpoint handler `do_int3()` in `arch/i386/kernel/traps.c`. `do_int3()` is called through an interrupt gate therefore interrupts are disabled when control reaches there. This handler notifies KProbes that a breakpoint occurred; KProbes checks if the breakpoint was set by the registration function of KProbes. If no probe is present at the address at which the probe was hit it simply returns 0. Otherwise the registered probe function is called.

There could be several possible problems which could occur when a probe is handled by KProbes. The first possibility is that several probes are handled in parallel on a SMP system. This problem is taken care of by the spinlock `kprobe_lock` which serializes the probe handling across processors. Another problem occurs if a probe is placed inside KProbes code, causing KProbes to enter probe handling code recursively. This problem is taken care of in `kprobe_handler()` by checking if KProbes is already running on the current CPU. In this case the recursing probe is disabled silently and control returns back to the previous probe handling code.

KProbes cannot be used directly for these purposes. In the raw form a user can write a kernel module implementing the probe handlers. However higher level tools are necessary for making it more convenient to use. Such tools could contain standard probe handlers implementing the desired features or they could contain a means to produce probe-handlers given simple descriptions of them in a scripting language. This is where DProbes comes in.

2.9.4 DProbes

Dynamic Probes[10] (DProbes) is a tool that can be used to insert software probes, dynamically into executing code modules. When a probe is fired, a user written probe-handler is executed. The probe-handler is a program written in an assembly-like language, based on the Reverse Polish Notation (RPN). Instructions are provided to enable the probe-handler to access the hardware registers, system data structures and memory. The RPN DProbes code is compiled into a kernel module[3] and run directly in kernel. The data may be collected and written to klog or traced using the Linux Trace Toolkit, depending on options available and/or specified.

2.9.5 SystemTap

SystemTap is an extension over DProbes. Instead of RPN, SystemTap employs a much more advanced awk-like language. We can have procedures, global and local varibles. The following simple example prints a message when any process reads a file.

[3] A linux kernel module is a dynamic library file which can be loaded by the kernel and run in kernel space.

```
probe kernel.syscall("read") {
  trace(string($tsk->pid) . " read " .
    string($size) . " bytes")
}
```

2.10 Solaris DTrace

`DTrace`[5] is a new tracing facility in Solaris 10. It is considered as one of the main new features in Solaris 10. It is a facility for dynamic instrumentation of production systems. DTrace features the ability to dynamically instrument both user-level and kernel-level software in a unified and absolutely safe fashion. `DTrace` allows almost all aspects of the kernel to be instrumented – there are about 30,000 instrumented probes in the kernel. DTrace uses a scripting language, D programs, which are run within the kernel to do monitoring. D programs resemble `awk` because a pattern matching style is used. DTrace is extremely powerful because of the large amount of instrumentation in the kernel. Running D within the kernel also means that monitoring can be done efficiently.

You can dynamically modify the operating system kernel or user processes to record additional data by placing `probes` at some locations of interest. A probe is a location or activity to which `DTrace` can bind a request to perform a set of actions, like recording a stack trace, a timestamp, or the argument to a function.

The core of DTrace (including all instrumentation, probe processing and buffering) resides in the kernel. Processes become DTrace consumers by initiating communication with the in-kernel DTrace component via the DTrace library. While any program may be a DTrace consumer, `dtrace(1M)` is the canonical DTrace consumer: it allows generalized access to all DTrace facilities.

2.10.1 An example

Before we go into detailed description, let's first look at a simple D program.

```
syscall::read:entry, syscall::write:entry
/pid == 12345/
{
    printf("%s(%d, 0x%x, %4d)\n", probefunc, arg0, arg1, arg2);
}
```

This program will print a line if the process 12345 reads or writes a file. For example, the output will be something like this.

```
write(2, 0x8089e48, 1)
read(63, 0x8090a38, 1024)
read(63, 0x8090a38, 1024)
write(2, 0x8089e48, 52)
```

The program places a probe at the entry point of `read(2)` and `write(2)`. The probe only fires when the process ID is 12345. Each time the probe fires, the system call name and the 3 formal parameters will be printed.

2.10.2 Providers and Probes

The DTrace framework itself performs no instrumentation of the system; that task is delegated to instrumentation *providers*. Providers are loadable kernel modules that communicate with the DTrace kernel module using a well-defined API. When they are instructed to do so by the DTrace framework, instrumentation providers determine points that they can potentially instrument. For every point of instrumentation, providers call back into the DTrace framework to create a *probe*. To create a probe the provider specifies the module name and function name of the instrumentation point, plus a semantic name for the probe. Each probe is thus uniquely identified by a 4-tuple:

<provider, module, function, name>

Probe creation does not instrument the system: it simply identifies a potential for instrumentation to the DTrace framework. When a provider creates a probe, DTrace returns a *probe identifier* to the provider.

Probes are advertised to consumers, who can enable them by specifying any (or all) elements of the 4-tuple. When a probe is enabled, an enabling control block (ECB) is created and associated with the probe. ECB contains instruction on how the system should response when probe fires.

When a probe fires and control is transferred to the DTrace framework, interrupts are disabled on the current CPU, and DTrace performs the activities specified by each ECB on the probe's ECB chain. Interrupts are then re-enabled and control returns to the provider. The provider itself need not handle any multiplexing of consumers on a single probe. (all multiplexing is handled by the framework's ECB abstraction)

2.10.3 Buffers

Each DTrace consumer has a set of in-kernel per-CPU buffers allocated on its behalf and referred to by its consumer state. Before processing an ECB, the per-CPU buffer is checked for sufficient space; if there is not sufficient space for the ECB's data recording actions, a per-buffer drop count is incremented and processing advances to the next ECB. This means that kernel event may get lost if the tracing program does not read event fast.

It is up to consumers to minimize drop counts by reading buffers periodically. Buffers are read out of the kernel using a mechanism that both maintains data integrity and assures that probe processing remains wait-free. This is done by having two per-CPU buffers: an active buffer and an inactive buffer. This is similar to the double-buffering concept in display device.

2.10.4 D Intermediate Format

Actions and predicates are specified in a virtual machine instruction set that is emulated in the kernel at probe firing time. The instruction set, "D Intermediate Format" or DIF, is a small RISC instruction set designed for simple emulation and on-the-fly code generation.

As DIF is emulated in the context of a firing probe, it is a design constraint that DIF emulation be absolutely safe. To assure basic sanity, opcodes, reserved bits, registers, string references and variable references are checked for validity as the DIF is loaded into the kernel. To prevent DIF from inducing an infinite loop in probe context, only forward branches are permitted. This means that we cannot have loops in DIF code.

2.10.5 D Language

DTrace users can specify arbitrary predicates and actions using the high-level D programming language. D is a C-like language that supports all ANSI C operators and allows access to the kernel's native types and global variables. D includes support for several kinds of user-defined variables, including global, clause-local, and thread-local variables and associative arrays. D programs are compiled into DIF by a compiler implemented in the DTrace library; the DIF is then bundled into an in-memory object file representation and sent to the in-kernel DTrace framework for validation and probe enabling. The `dtrace(1M)` command provides a generic front-end to the D compiler and DTrace, but other layered tools can be built on top of the compiler library as well.

2.10.6 Variables

In term of variable scopes, there are three types of variables, Global variables, clause-local and thread-local variables. D variables can be declared using C declaration syntax in the outer scope of the program, or they can be implicitly defined by assignment statements. Global variables are accessed directly using the variable names. Clause-local and thread-local variables are accessed using the reserved prefixes this-> and self-> respectively. The prefixes serve to both separate the variable namespaces and to facilitate their use in assignment statements without the need for prior declaration. Clause-local variables access storage that is re-used across the execution of D program clauses, and are used like C automatic variables. Thread-local variables associate a single variable name with separate storage for each operating system thread, including interrupt threads.

2.10.7 Aggregating Data

When instrumenting the system to answer performance related questions, it is often useful to think not in terms of data gathered by individual probes, but rather how that data can be aggregated to answer a specific question. For example, if one wished to know the number of system calls by user ID, one would not necessarily care about the datum collected at each system call (one simply wants to see a table of user IDs and system calls). Historically, this question has been answered by gathering data at each system call, and post processing the data using a tool like `awk(1)` or `perl(1)`. However, in DTrace the aggregating of data is a first-class operation, performed at the source.

2.10.8 Comparing to SystemTap

DTrace and SystemTap can both be used for system and program auditing. Dynamic instructions can be executed when events are fired. They both employ awk-like pattern matching language. DTrace supports awk-like language whereas DProbes supports assembly-like language. However, there are some differences between them.

1. DTrace is for Solaris[4] while SystemTap is for Linux.
2. DTrace script is interpreted while SystemTap script is compiled in to native executable code. This makes SystemTap more efficient and powerful but less safe.
3. The `D` language is much more restricted than SystemTap. The `D` language does not support procedure declarations or a general purpose looping construct. This avoids a number of safety issues in scripts including infinite loops and infinite recursion.
4. SystemTap will support kernel debugging features that DTrace does not, including ability to write arbitrary locations in kernel memory and ability to invoke arbitrary kernel subroutines.
5. SystemTap does not have *thread-local variables* which are core to the DTrace architecture. The following `D` example shows how to use a thread-local variable to output the amount of time that a thread spends in a `read(2)` system call. If we change `t` to global variable, the time may be reported incorrectly.

```
syscall::read:entry {
    self->t = timestamp;
}
syscall::read:return /self->t/ {
    printf("%d/%d spent %d nsecs in read\n",
        pid, tid, timestamp - self->t);
}
```

2.11 Linux Trace Toolkit

The Linux Trace Toolkit[15] is a suite of tools designed to extract program execution details from the Linux operating system and interpret them. Specifically, it enables its user to extract processor utilization and allocation information for a certain period of time. It is then possible to perform various calculations on this data and dump this in a text file. The list of probed events can also be included in this. The integrated environment, based on the GTK+ library, can also plot these results and perform specific searches.

[4]DTrace has been ported to FreeBSD recently

LTT is composed of independent software modules. Each module has been designed in order to facilitate extension while minimizing performance overhead. Figure 6 represents the architecture used. Note that, for reasons of simplicity, only the architectural parts of the Linux kernel relating to LTT's functionality are presented. The arrows indicate the flow of information through the different modules making up LTT. Basically, events are forwarded to the trace module via the kernel trace facility. The trace module, visible in user space as an entry in the `/dev` directory, then logs the events in its buffer. Finally, the trace daemon reads from the trace module device and commits the recorded events into a user-provided file.

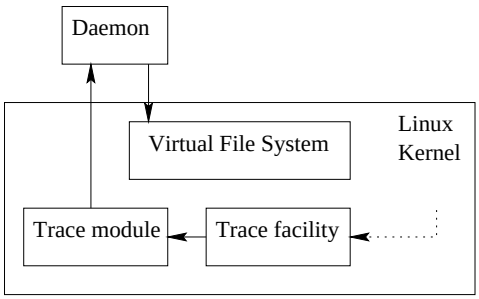

Figure 6: LTT architecture

3 Status of our current work

We have developed a user-level monitoring framework *logbox*[19]. Logbox is a framework 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.

3.1 Design Objectives

3.1.1 Flexible

Our goal is to have a general auditing frameworks which can then be used for:

1. traditional audit trail (e.g. this could be to fulfill a mandatory requirement);
2. system monitoring (e.g. this could check whether selected process run selected programs);
3. security monitoring (e.g. intrusion detection which checks for unusual operations).

Our architecture provides a kernel-based event delivery mechanism to a user-space monitor and as such it allows for monitors to perform arbitrary processing. As the monitor is in user-space, unlike traditional auditing mechanisms, we allow for non-privileged users to run their own arbitrary monitors. Some reasons for users to run their own monitors may be as an additional security measure such as wanting to audit untrusted software.

3.1.2 Secure and Reliable

Apart from a flexible mechanism, we would like some security and reliability guarantees. Firstly, for a collection of processes being monitored, they should not have any way of escaping from being monitored. Thus their child processes are also monitored. So this gives a form of *mandatory monitoring*. System such as `syslog` cannot be used for mandatory monitoring. Secondly, it should not be the case that a process can run in a way which can circumvent the monitoring of an event of interest. For example, if writing to a particular file is supposed to be the monitored event of interest, then the monitor should receive the event whenever the underlying file is written to irrespective of pathname and current working directory manipulations. So this gives a form of *reliable monitoring*. Kernel mechanism such as `ptrace(2)` and `/proc` can be easily escaped thus it is not reliable. It is possible to overflow the buffer in DTrace [5][4] thus it is also not reliable. Lastly, to avoid security problems, we prevent a user from monitoring process owned by other users with the exception of root. There is also a monitor policy demon which allows for more fine-grained control over what users can monitor, e.g. a user might only be able to monitor certain events or be restricted to only 2 monitors in total.

3.1.3 Efficient

Performance is very important for monitors which are in user-space. In order to minimize the number of task switches caused by our auditing system, the monitor is not activated after each monitored system event caused by the monitored process. Instead, the event is buffered in the event buffer inside the kernel. The monitor process can call `lbox_poll()` (a monitor library routine) to receive the events. The monitored process will be blocked at the next monitored system event if the buffer is full. Thanks to buffers, auditing system without immediate response can gain much performance improvement.

3.2 An Example

To give an overview of how our auditing system works, we give a concrete example. In this example, we implement a simple monitor using our framework.

Figure 7 illustrates an example on how to use our auditing system *logbox*[5] to implement a monitor to audit the *apache* web server. We want to see whether the web server executes a shell, for example, because of been exploited. For

[5] *logbox* is the code name of our current implementation

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. Rather than discuss the actual monitor API, we have used a library for monitoring which gives a simpler way of using the monitoring infrastructure.

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 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 current processes as well as future processes.

Part 3 creates a logbox with the process and event specifications. A buffer of size 4096 is specified and it has no timeout. It returns when there is one matching event. The rest of part 4, just collects events for processing by the monitor.

The monitor can also choose to disable and then re-enable events in the approved specification. The logbox can be destroyed either explicitly with `lbox_delete()` or implicitly when the monitor terminates. A variety of system events associated with objects can be monitored such as files, sockets, processes, etc. The library routines use the process specification to construct an expression which can be evaluated by the kernel to determine whether an existing or new process is to be monitored by this monitor.

3.3 The Monitor Specification

A monitor is a process which audits the behavior of other processes. Monitors are described by two specifications: (a) the process specification defines the processes to monitor; and (b) the event specification which defines the operations to monitor.

3.3.1 Process Specification

The processes to be monitored by a particular monitor is specified by means a boolean expression. The boolean expression is constructed from the usual boolean operators, `and`, `or` and `not`, over the following:

1. *true/false*: For example, a specification which is simply *true* means all processes.

2. *uid/euid/suid/fsuid*: These predicates are true if and only if the user id of the process is same as the user id specified.

3. *pid*: 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.

```
/* open the /proc/lbox file descriptor */
lbox_open(); // implicitly used by library

/* PART 1: create events spec: current directory and exec.
   Also need to specify the information to be retrieved. */
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 logbox */
lbox_create(proc_spec, event_spec, 4096);

/* PART 4: read and process events. */
for (;;) {
   /* -1 is the timeout value which means to block for ever */
   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 7: A Simple Monitor

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. Both cases are treated as child processes. This can be used to include or exclude both existing processes and processes which are not yet created.

5. *executable program*: This predicate is true if and only if the executable program of the process is the same as the program specified. This can be used to include or exclude both existing processes and processes which are not yet created.

By using boolean expression, process specification can be very flexible. For example, one can specify all process owned by the user Bob except process 1468 and its child processes. Using the library this can be written as

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

3.3.2 Event Specification List

An event specification defines the monitor behaviors of the monitored processes which are of interest to the monitor. Suppose a monitor event expression is S and event E happens. Then S is triggered when E is compatible with the monitor expression S.

An event will be generated and information associated with the event will be passed to the monitor. For, example the inode number, canonical pathname, and operation type will be passed to the monitor for a file event. However, in order to maximize performance and suit different application, the monitor can choose not to receive the unwanted information. For example, the monitor can specify not to report the canonical pathname. This reduces the time for building the canonical pathname and also reduces the event size.

The time and pid are the common information for all types events.

1. *time*: Time is expressed in system jiffies. It can be translated into human readable time format by the library we provide.

2. *pid*: This is the process which performs the operation.

Currently we have only implemented three types of events which cover a large class of interesting events, namely those on files, sockets and processes.

Different event classes can be supplied different parameters. Here we only describe the parameters for the *file* event class.

1. *file pathname*: The pathname will be translated into inode number and device number pair. This pair is used as the identifier of file and stored in the system.

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.

   Since this is a two bit flag, 0 (defined as IGNORE) conveniently means that we are not monitoring this directory and its subdirectories. Thus 0 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 specification work is that when more than one file event specifications match 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, if read+execute is issued, a file read access event is generated when a process reads from the file. However, no event is generated when the process writes to the file.

   We can also specify delete (unlink) operation on a file. It is useful when you want to keep a specific file on a writable directory. We can also specify creation of file, deletion of file operations on a directory. It only makes sense when the file is a directory and writable. Other operations involving modification of meta-data such as creation time and access right can also be specified as operations.

When a file is removed (i.e. its reference count becomes 0), all the corresponding file event specification is removed also. This implies that the number of event specifications may decrease at run time. For example, 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 saying that the file `/tmp/a` is read-only. 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.

In return, besides the common information, the following information can be included.

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. *canonical pathname*: A file can have many possible (absolute) pathnames because of hard links, symbolic links, the ".." directory and different mounting points. The canonical pathname we provide is a normalized pathname not containing symbolic links.

In addition to the previous different event parameters, all event specifications have a common parameter which is the flush parameter. The flush parameter tells the kernel whether the event buffer needs to be flushed when the event is generated. This is analogous to the `PUSH` flag in the TCP packet and allows timely delivery of important events to the monitor.

An event can be specified as non-mandatory which is used to mean that the kernel can choose to drop the event if the event buffer is full. This can be useful for not so important events where we want to allow the monitored processes to continue execution.

3.4 Preliminary Experimental Results

Our current prototype is implemented in Linux and only supports files and network events. This is already rather useful and we are working on extending the classes of events. In this section, we want to show that the architecture is quite efficient. To do this, we use a simple micro-benchmark which gives an indication of worse case overheads. The program performs 1000000 open(2) and close(2) system calls with a monitor watching for all the file access.

The Linux kernel used here is 2.6.10. The machine is a Pentium IV 3.0GHz PC with 1G memory. We can compare the auditing framework in each of the following scenarios:

1. *clean kernel*: It is the clean stock kernel. The logbox module is not in the kernel.

2. *proc miss*: The logbox module is loaded in the kernel. However, no monitor is monitoring the busy-open process.

3. *file miss 1/2*: A monitor is monitoring the busy-open process. However, the monitor is monitoring a different file. Thus no event is generated. The difference between the two scenarios is that whether directory traversal is needed. In *file miss 1*, there is no file spec which has SUBDIRECTORIES flag set, whereas in *file miss 2*, there is a file spec which has SUBDIRECTORIES flag set, thus we need to traverse all the way up to the root directory to make sure that the action does not match the file spec.

4. *1 dir level*: The monitor is monitoring a regular file. The process is accessing the regular file which is being monitored.

environment	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

5. *2/3 dir levels*: The monitor is monitoring a directory which is 2 or 3 levels higher than the file which is accessed by the busy-open process. For example, the monitoring the directory /a and its subdirectories. The the busy-open process is accessing the /a/2/3.txt file. This micro-benchmark is used to examine the time used in directory traversing.

6. *no buff*: 1000000 events are generated. However, no buffering is used - every event requires a transition to the user-space monitor. Thus, the busy-open process is suspended until the monitor has read the event.

7. *ptrace*: Here, we compare our system with the traditional `ptrace()` mechanism.

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. However, *file miss 1* has smaller overhead ($\sim 9\%$) because directory traversal is not needed. When the monitor is invoked without any buffering of events (no buff), overhead jumps substantially to 337%. It is interesting to note that this is still much smaller than `ptrace` which has 28669% overhead or about seven times slower than the no buffering case. Adding a buffer drops the overhead to 15%.

We compare our performance results with other auditing systems in Solaris. We use the same benchmark program and the same hardware. Table 2 shows that truss which uses the `/proc` facility brings 2510% overhead to the `open()` system call and DTrace brings 119% overhead to the system call. In the DTrace

[6]The D program is "syscall::open:entry { @[execname]=count(); }"

environment	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[6]	8.41 ± 0.02	1.08 ± 0.01	7.33 ± 0.01

Table 2: Open micro-benchmark on Solaris 10 (SunOS 5.10 i386)

environment	real	user	sys
clean kernel	0.27 ± 0.01	0.01 ± 0.01	0.20 ± 0.01
monitored	0.28 ± 0.01	0.01 ± 0.01	0.20 ± 0.01

Table 3: Connect micro-benchmark

test, the kernel does not return any information to the user space monitor (because it is an aggregation D program), whereas our system returns information to the 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.

Since the open micro-benchmark reuses the same file over and over again, the open itself becomes quite fast. We use another micro-benchmark on socket operations to show the effect of a more expensive system call. The benchmark creates a TCP socket and connects to a local port. This is repeated for 4000 times and the results are in Table 3. We can see that in an expensive system call, the overhead is not noticeable.

Besides the micro-benchmark, we use a macro-benchmark which is the well-known uncompress kernel experiment. The "untar linux kernel source" script is used to illustrate monitoring of an application which has to do significant work. The script will untar a compressed linux source tarball (linux-2.6.10.tar.bz2), tar the source tree again, finally remove the tarball and the source tree.

The results given in Table 4 show that the overhead on the macro-benchmark is so small that it is hidden by the experimental error. The point of the macro-benchmark is that micro-benchmarks show worst case performance while macro-benchmarks give a better indication of typical performance. In this case, the benchmark spends enough time doing work in user space and requires quite a lot of I/O so that overheads of monitoring essentially disappear.

environment	real	user	sys
clean kernel	209.04 ± 0.72	203.23 ± 0.82	5.28 ± 0.15
directory monitored	209.44 ± 0.47	203.32 ± 0.59	5.56 ± 0.08
ptrace	209.48 ± 0.35	203.64 ± 0.45	5.42 ± 0.11

Table 4: Untar benchmark

4 Applications

To show the usefulness of a general auditing framework, we describe some more applications. As we will see, they can be easily implemented by making use of the user space library. Note that one can write even easier to use monitors in the style of DTrace simply by writing a scripting application to use the auditing framework, i.e. an awk-like interpreter.

1. Testing a possible malicious program

   Sometimes we need to run a program of which the source cannot be accessed. We are not sure whether this program is stealing our sensitive data, for example, our PGP keys. We can use our auditing system to monitor the suspicious program. We can specify the interested event to be the file read event on `/home/user/.gnupg/secring.gpg` When the program reads the file, the monitor will report the action to the user.

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` directory. We can specify the interested event to be the file read and write event on files *outside* the `/var/www` directory. When the web server accesses files outside the `/var/www` directory, the monitor will report the action.

3. Monitoring the whole system

   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 packet to the network `137.132.0.0/255.255.0.0`. Our interested event is the socket connect event with the destination network address specified as the above address.

4. Program debugger

   Our auditing system can be used the same way strace is used. It can be used by the programmer to observe program behavior. Unlike strace and Systrace [2], our auditing system does not stop the monitored process on each system call. Events are buffered in the kernel before transfered to the monitor. This results in very little impact on the process being debugged.

5 Future Works

This system can be further extended in the follow direction. Although they are further plans, some of them already have design prototypes.

5.1 Kernel Inspection

We can extend *logbox* so that event handler can be executed not only in user space, but also in kernel space. This will bring us two advantages. Firstly, handling event in kernel will save us the time caused by context switch. Secondly, handling event in kernel provides the programmer kernel data structure and procedure accessing. However, we should make sure programmers access the kernel in a "secure" manner. I.e. programmers should not crash the system, modify the system where should not or reveal any information which should not. We do this by defining a language which can be verified to be secure.

In DTrace, the `D` language is very restricted. Program written in `D` cannot have loop and is usually not reusable. We can overcome this by allowing certain kind of loops whose number of iterations is bounded. For example, the `foreach` statement which appears in many scripting language can usually be bounded. Although some kernel data structure such as thread, file and process can be accessed, it is difficult to export new data structures in DTrace. We propose a systematic way of exporting kernel data structure and procedure. Programmers not only can retrieve information from the kernel but also modify the kernel as long as in a "secure" manner. This gives us the ability to send feedback to kernel. We will further describe this in the following subsection.

5.2 Access Control

We can apply the same idea to implement access control systems. The standard UNIX security model provides a basic level of protection against system penetration. However, this model alone is insufficient for security-critical applications. The security of a standard UNIX system depends on many assumptions. File permissions must be set correctly on a number of programs and configuration files. Network-oriented services must be configured to deny access to sensitive resources. Furthermore, system programs must not contain security holes. To maintain security, one must constantly monitor sites such as CERT and SecurityFocus, install new patches, and hope that holes are patched before an attacker discovers them. Since potentially vulnerable system programs often execute with root privileges, attacks against them often lead to total system compromise. The typical UNIX system is therefore characterized by many potential weaknesses and is only as secure as its weakest point.

The limitations of the UNIX security model have created much interest in alternate paradigms. This has drawn attention to a wide variety of access control systems. Examples are: Janus[14], MAPbox[21], Sandbox[22], Systrace[2], LIDS[23] and BSD jail[24]/chroot. We can extend *logbox* to implement sandbox systems. A sandbox is an execution environment in which processes are limited to do a set of operations. Sandboxes are attractive because they provide a centralized means of creating security policies tailored to individual programs and confining the programs so that the policies are enforced. They therefore provide great potential for simplifying system administration, preventing exploitation of security holes in system programs, and safely executing potentially

malicious code. Using sandbox, we can restrict a program to do only those operations it needs to do. This conforms to the "Least Privilege Principle".

The same mechanism used in *logbox* can be used to develop access control systems such as Janus[14] and sandbox[13]. When certain event happens, the programmer not only can notice it, but can also send feedback to kernel. For example, when a process try to access certain confidential file, we not only can log the access, but also can tell kernel to deny the access. This gives us the ability to build extensible access control system. We do this by defining "restricted event" whose handler will return the decision to the kernel. For example, the event of accessing any file in `/etc/` can be defined as restricted and the event handler can examine the file and user ID to determine whether to allow the access.

5.3 Investigating Windows

Although the current prototype is implemented on Linux, it does not mean it only works on Linux. The idea can be applied on Microsoft Windows also. Note that the difference between Windows and Linux is not just different system calls and different kernel data structures. They are on different paradigms and employ different kernel architectures. Linux employs monolithic kernel while Windows employs hybrid one. This makes monitoring Windows more complicated, because the functionality of kernel in Windows is splited into kernel and service processes while in Linux the kernel does everything. Registry plays an important role in software configuration in Windows. Access on registry must be specially monitored. We use inode to identify file objects, but this must be changed for Windows. Lacking of documentation and source code makes auditing more useful in Windows, because programmers are more curious to know what is going on there. However, not much work has been done except some trivial logging tools[17][18]. We have developed *ResMon*[20][7], a software diagnostic tool. System administrators can use *ResMon* to investigate program behavior and find out root cause of software problems. We do this by monitoring and analyzing resource (file, registry, network and IPC) usage.

5.4 Efficiency

We can further improve the logging efficiency by introducing event compression. The logged event is stored in a buffer in the kernel and then transfered to the user space monitor. If we compress the logged event in the buffer, there will be less data transfered thus improving the efficiency.

5.5 New Trace Mechanisms

Our current implementation use the LSM [7] framework in Linux 2.6.x as kernel instrumentation mechanism. We have developed three types of events, file,

[7]It has been accepted but the conference has not held yet.

network socket and process. We can develop more types of events such as X11 related operations (creation of windows), kernel module operations (install new device, insert module) and file system operations (mount/unmount).

5.6 Useful Applications

We can use our auditing system to develop real useful applications such as network IDS system. Normal network IDS system only deal with packet data. Using our auditing system, we can associate operating system status with packet. For example, we are able to know which user is trying to access `http://www.yahoo.com`.

6 Conclusion

In this project, we propose a general system auditing framework. This project, as far as we are aware, is the first to consider the requirements and uses of a general auditing framework for arbitrary monitoring of processes. In particular, it highlights, the following:

- What kinds of events are useful? One would need more than just system call tracing.
- Confidentiality requirements.
- Security considerations

Because of these requirements and the mandatory auditing requirements, an auditing framework for monitoring has different objectives from one for sandboxing (i.e. Janus, Systrace, etc.) and also general kernel instrumentation and tracing architectures (i.e. DProbes, DTrace, etc.).

References

[1] J. Finke, "Process Monitor: Detecting Events That Didn't Happen", Usenix LISA, 145–154, 2002.

[2] Niels Provos, "Improving Host Security with System Call Policies", USENIX Security Symposium, 2003.

[3] Sun Microsystems, "System Administration Guide: Security Services", part IV: Auditing and Device Management.

[4] Sun Microsystems, "Solaris Dynamic Tracing Guide".

[5] Bryan M. Cantrill, Michael W. Shapiro and Adam H. Leventhal, "Dynamic Instrumentation of Production Systems", Usenix 2004.

[6] C. Wee, "LAFS: A Logging and Auditing File System", Annual Computer Security Applications Conf, 231–240, 1995.

[7] 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, 2002.

[8] Tal Garfinkel, "Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools", Network and Distributed Systems Security Symp, 2003.

[9] Prasanna Panchamukhi, "Kernel debugging with Kprobes".

[10] Richard Moore, "A Universal Dynamic Trace for Linux and other Operating Systems", USENIX Annual Technical Conference, 2001.

[11] Vara Prasad, "Locating System Problems Using Dynamic Instrumentation", Linux Symposium, 2005.

[12] Alfred V. Aho, Brian K. Kernighan, and Peter J. Weinberger, "The AWK Programming Language". Addison-Wesley, 1988.

[13] David S. Peterson, "A Flexible Containment Mechanism for Executing Untrusted Code", Master's thesis, 1996.

[14] T. Garfinkel and D. Wagner, Janus: A practical tool for application sandboxing. 1999.

[15] Karim Yaghmour and Michel R. Dagenais, "Measuring and Characterizing System Behavior Using Kernel-Level Event Logging", Usenix Annual Technical Conference, 2000.

[16] R. Faulkner and R. Gomes, "The Process File System and Process Model in UNIX System V", USENIX Conference, 1991.

[17] Filemon, `http://www.sysinternals.com/Utilities/Filemon.html`

[18] Regmon, `http://www.sysinternals.com/Utilities/Regmon.html`

[19] W Yongzheng and RHC Yap, "A User-level Framework for Auditing and Monitoring", Annual Computer Security Applications Conference, 95–105, 2005.

[20] Rajiv Ramnath, Sufatrio, Roland Yap and Wu Yongzheng, "ResMon: A Tool for Discovering Software Dependencies, Configuration and Requirements in Windows", Large Installation System Administration Conference, 2006.

[21] A Acharya and M Raje, "MAPbox: Using Parameterized Behavior Classes to Conne Untrusted Applications", USENIX Security Symposium, 2000.

[22] DS Peterson, M Bishop and R Pandey, "A Flexible Containment Mechanism for Executing Untrusted Code", USENIX Security Symposium, 2002.

[23] "Linux Intrusion Detection System", `http://www.lids.org`.

[24] "FreeBSD jail manual page", `http://www.freebsd.org/cgi/man.cgi?query=jail&sektion=2`.