SOFTWARE – PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2010; **40**:989–1005
Published online 2 July 2010 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/spe.983

Audlib: a configurable, high-fidelity application audit mechanism

Benjamin A. Kuperman[1,*,†] and Eugene H. Spafford[2]

[1]*Computer Science, Oberlin College, Oberlin, OH 44074, U.S.A.*
[2]*CERIAS, Purdue University, West Lafayette, IN 47907, U.S.A.*

SUMMARY

In this paper, we introduce Audlib, an extendable tool for generating security-relevant information on Unix systems. Audlib is a wrapper environment that generates application level audit information from existing executable programs. Audlib is not a detection system, instead it is designed to supplement existing audit systems and work transparently with them. Audlib records information that is not presently available from existing kernel-level audit sources. Here, we describe the design of the Audlib framework and the information it provides. We compare auditing the actions of a web server with Audlib to existing kernel audit sources and show that we have 2–4 times the throughput of Linux `auditd` and less than half the performance overhead of Solaris BSM while collecting detailed information about the server's execution. Although Audlib is focused on recording security information, this technique can be used to collect data for a wide variety of purposes including profiling, dependency analysis, and debugging. Copyright © 2010 John Wiley & Sons, Ltd.

Received 13 October 2009; Revised 13 April 2010; Accepted 14 April 2010

KEY WORDS: audit systems; computer security monitoring; attack detection; intrusion detection; misuse detection

1. INTRODUCTION

AUDLIB is an extensible tool for capturing security-relevant information on Unix systems. AUDLIB is a wrapper environment that generates application level audit information from existing executable files. It can be applied to specific programs, or used to instrument an entire system. It is designed to supplement existing audit systems and work transparently with them. AUDLIB collects information at the application level and is therefore able to obtain information not normally provided by existing kernel level audit sources (e.g. [1, 2]).

AUDLIB is not a detection system. Instead, it is a tool designed to improve both the quality of audit information produced on a system and the information generated to be used by various forms of detection systems. We are augmenting the information produced rather than introducing a new detection technique. This will allow for more accurate detection and a reduction in the number of false alarms generated. We have included a few proof-of-concept detectors to demonstrate how the audit records can be used for detection purposes.

The remainder of the paper is structured as follows: Section 2 describes the overall design of AUDLIB, Section 3 discusses the past and related work, Section 4 covers our various experimental results, Section 5 describes the use of this technique outside of security, and Section 6 contains our conclusions and plans for future work.

*Correspondence to: Benjamin A. Kuperman, Computer Science, Oberlin College, Oberlin, OH 44074, U.S.A.
†E-mail: Benjamin.Kuperman@oberlin.edu

2. AUDLIB DESIGN

AUDLIB is designed as a 'wrapper' for existing programs on Unix-like operating systems without requiring either recompilation of programs or the addition of new features to the kernel. It is intended to not interfere with the operations of the operating system or the programs themselves. Audit information is collected from the application level allowing AUDLIB to report information not currently available from the existing kernel-based audit systems. By collecting information from the application level, we can provide better and more accurate information than existing audit sources. It is also desirable that the overhead of using AUDLIB is not too great—at least compared with existing audit systems. A small change in the performance of an application is almost always justified when there is a need for additional security auditing.

2.1. Interposition

Based on past experience [3], we knew that it was possible to meet our goals through the use of a technique known as *library interposition*. Interposition is the 'process of placing a new or different library function between the application and its reference to a library function' [4]. This technique allows a programmer to intercept function calls to code located in shared libraries by directing the run-time dynamic linker to first attempt to reference a function definition in a specified set of libraries before consulting the normal library search path.

Essentially, programmers need to create their own library containing functions that have the same name (and signature) as the library calls they wish to intercept. Then, by either setting the environment variable `LD_PRELOAD` or modifying `/etc/ld.so.preload`, the dynamic linker `ld.so` will examine the specified library first before searching the default library search path (Figure 1).

To make our audit library transparent to the operation of the system, we collect the information we need, log it, and then tell the dynamic linker to continue searching and to invoke the next version of this library call it finds. Our library then waits for that call to return, logs additional information if needed, and passes the return value back up the call stack. If we have only a single interposing library, then the next version of the call will be the actual one. However, if we have more than one interposing library, it will go through all of those in sequence before calling the normal version. This allows our library to be able to compose multiple audit libraries into a single system. More information about library interposition and the construction of interposable libraries can be found in [5–7].

2.2. Goals of detection/data recorded

The current version of AUDLIB consists of three stand-alone interposable libraries, each focused on producing information relevant to a particular class of security event. These three categories are attacks, intrusions, and computer misuse. In this section, we briefly explain each goal of detection

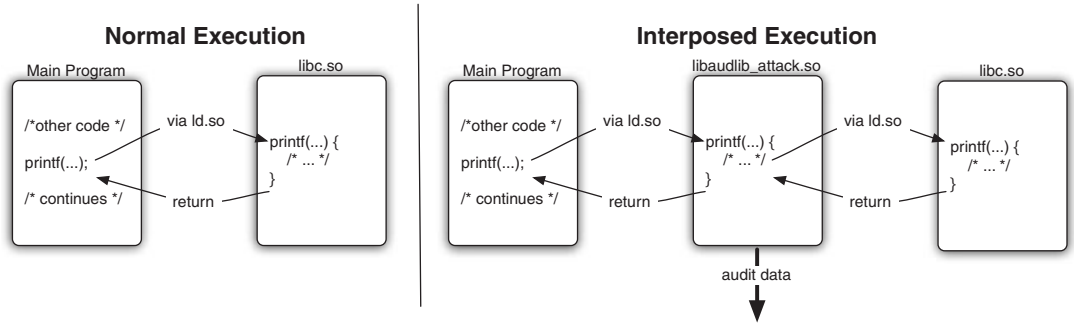

Figure 1. The execution of a call to `printf()` in a dynamically linked executable. The left side shows the normal execution sequence and the right shows how a transparently interposed library such as AUDLIB changes the control flow.

DOI: 10.1002/spe

and list the events we monitor. We note that other libraries could be constructed for additional classes of information.

For each audit record, AUDLIB records a header containing the following:

- The size of the record.
- The library and call that generated it.
- The timestamp when the record was created.
- The timestamp when the event was recorded.
- The PID (process id) and parent PID.
- Additional information specified below.

Each audited event includes function-specific information after this common header. There are over 100 audited events. We list these items in Appendix A and in this section use some samples from each category to illustrate the contents.

In addition to the auditing of function calls, each library contains initialization and finalization routines that are invoked on program start and exit. These are used to initialize and shut down communication channels needed for transmitting audit information, and they also serve as data collection points for information used over the duration of the program run.

2.2.1. Attacks. Attacks are attempts to exploit a vulnerability in a computer system. These include buffer overflow attacks, format string attacks, and race conditions. These are the actions that most current IDSes attempt to detect.

To allow for the detection of buffer overflow attacks, AUDLIB monitors all unbounded string copy functions (`strcpy()`, `gets()`, etc.) and reports:

- Where each buffer is located in memory (stack, heap, data, etc.) and if that location is set to be executable by the OS.
- The size, address, and payload of the source buffer.
- The size and address of the destination buffer.

For format string attacks, we record the size, location, and content of the format field of the `printf()` family. Note that neither the buffer information nor the format strings are available from typical kernel-level audit sources.

For race conditions, we record all symbolic file accesses (`access()`, `open()`, etc.) as well as operations that change the meaning of symbolic file names (`chdir()`, `chroot()`, etc.). This information is not available from kernel audit sources because they record the actual files accessed, not what the program requested. A successful attack would show two different files, but the program would be using the same symbolic names.

The initialization function also logs real, effective, and saved UID and GID for the process; the current working directory; the complete command line; and the complete set of environment strings.

2.2.2. Intrusion. For the detection of intrusions, we provide information for detection systems modeled after [8, 9], which use command sequences to determine whenever an impostor is masquerading as a legitimate user. In addition, we augment the audit record with information that can be used for basic anomalous behavior detection as proposed by Denning [10].

This information includes the real, effective, and saved UIDs and GIDs at both the start and end of the program; the login name of the user; the name of the executable as invoked by the user; the complete set of command line arguments; the `stat()` of the binary itself as well as any possible symbolic link used by the user to invoke the program; the amount of user and system CPU time used by the process; and the amount of user and system CPU time used by all children processes.

2.2.3. Computer misuse. The types of potential computer misuse activities that AUDLIB monitors includes file accesses that a kernel level audit system might be able to supply information about (such as the accessing of files belonging to another user). We include this information for

DOI: 10.1002/spe

completeness, anticipating that one might deploy AUDLIB to do all auditing. We audit all library functions that access or modify the file system from `access()` to `mknod()` to `truncate()`. This allows for a coarse-grained (file level) of monitoring.

In addition to those operations, we also log every `read()` or `write()` request made by the targeted program. Neither BSM nor the Linux audit daemon `auditd` enable logging of these actions in their default configurations: their documentation indicates that the performance cost is prohibitive. However, even with these logging options enabled, we outperform the current audit sources.

By collecting information on reads and writes at the application level, it is possible to construct monitoring systems that can perform detection based on which *portions* of a file were accessed, rather than simply assuming that anything (and thus, everything) within might have been accessed. It is possible to record the contents of reads and writes, and the audit trail can be augmented to also report file offsets, if desired. In Section 4.4 we briefly describe a system that uses the fine-grained functionality of our computer misuse monitoring system to track and log all changes made by a system administrator when handling a trouble ticket.

2.3. Configuration

AUDLIB uses a central configuration file to control the audit library logging behavior. An example of this file is shown in Figure 2. `Program_Name` can be either a full path (`/bin/ls`) or only the program name (`ps`) to match on any path. `Function_Name` is the interposed function name complete with library indicator A, I, or M; a function name of `all` can be used to refer to everything in the library. `M:all` would refer to all events in the **M**isuse library and `A:printf` is simply the `printf()` function in the **A**ttack library. It is also possible to specify the inclusion of items that are **C**ommon to all libraries with **C**.

The same audit configuration applies to all descendant processes unless they are covered by another configuration entry that takes precedence. When wrapping a program from the command line instead of `/etc/ld.so.preload`, a different configuration file can be specified for the execution of a program.

2.4. Audlib deployment

The interposing libraries that perform the audit data generation and collection are a set of shared objects (e.g. `libaudlib_attack.so`) that are typically installed in a system library location such as `/usr/local/lib` with the standard ownership and permissions—owned by **root** with

```
all {

    all = exit, perror

    C:audlib_openlog = perror, exit

}

/bin/ls {

    M:all = syslog

    C:exec = syslog, exit

}

ps {

    A:printf = perror, exit

}
```

Figure 2. The layout of the AUDLIB configuration file.

DOI: 10.1002/spe

read and execute permissions. There are three ways in which AUDLIB might be used to monitor a system:

1. Full system monitoring.
2. Targeted application monitoring.
3. Directly linked to an executable.

2.4.1. Full system monitoring. To use AUDLIB to monitor an entire system, the administrator needs to add entries for these libraries into `/etc/ld.so.preload`. AUDLIB will monitor every program, and the configuration file will be used to determine what information, if any, is to be collected from that process. The system can be configured to audit everything by default and then have exceptions for specific programs to not generate records, or have nothing audited by default and then only target particular programs.

In this mode, AUDLIB will be applied to the execution of all dynamically linked executables, including those that might be uploaded or created by users. This mode gives the greatest coverage and is the most difficult for an attacker to evade or disable.

2.4.2. Targeted application monitoring. AUDLIB can also be used to only monitor a subset of processes running on the system via the `aud_run` wrapper (or using `LD_PRELOAD`) when starting a program. In this mode, only the initial process and all of its descendant processes will be monitored. An administrator might use this to gather more security audit information about the web server (or FTP, SMTP, SSH, etc.), but not generate information from the rest of the system. Targeted monitoring can be applied to more than one service on the same system, without causing any interaction problems.

When used in this manner, it is possible for a program to disable auditing by unsetting the `LD_PRELOAD` environment variable and then calling `exec()`. AUDLIB will no longer monitor that subprocess or its descendants. Additionally, if a program is not run with `LD_PRELOAD` set, it will not be monitored, hence, programs run by users or outsiders will not be audited.

2.4.3. Directly linked to an executable. The final way in which AUDLIB might be used is somewhat non-standard and not recommended as a deployment choice. It is possible to directly link a compiled program against one of the AUDLIB shared libraries. This usage scenario will result in a program that cannot be run without being audited, but none of its subprocesses will be monitored if they execute executables not linked against AUDLIB. As with the targeted technique, programs run by users or outsiders will not be monitored.

2.5. Additional tools

We supply additional tools that facilitate the use of AUDLIB.

- `audlogread`**:** There is a log parser called `audlogread` that is capable of parsing one or more AUDLIB audit records and re-displaying the log records. Some basic filtering can be performed. Output can be either in a detailed 'human readable' format or in a parseable format with one entry per line.
- `aud_run`**:** We include a script called `aud_run` that allows execution of a command sequence under AUDLIB monitoring. It also supports features for further AUDLIB development including the ability to test a program under different successive configurations.

Finally, we include a sample of proof-of-concept security monitoring scripts. These scripts take as input the parseable format of AUDLIB logs and demonstrate the detection of buffer overflows, race conditions, and format string attacks.

2.6. Extending the library

We are releasing AUDLIB under an open-source license and want to encourage community development and extension. It is a straightforward process to add monitoring for a new library call. As an example, the complete code for the attack library monitoring of the `strcpy()` function

DOI: 10.1002/spe

is shown in Figure 3 on the facing page. Using the macros and functions provided by AUDLIB, adding a new function only requires knowledge of the name and function signature. The log entry should contain relevant information, and we have included support for many standard C variable types. The sequence of tokens must also be recorded in the file used to compile `audlogread`.

The audit information collected and recorded by AUDLIB is not restricted to what is present in the current implementation. AUDLIB's audit data could be extended to support a wide variety of additional information. For example, a mobile device might query and record GPS information for certain events, checksums or digital signatures could be regenerated and checked before allowing certain events to proceed, the state of network connections could be monitored or recorded, etc. Furthermore, extending AUDLIB can be done without touching either the kernel or the monitored applications.

3. PAST WORK

3.1. Current audit systems

Many, if not most, modern computer audit data generation systems are descended from the US Department of Defense's documents on trusted systems (often referred to as the 'rainbow series'

```
char *strcpy(char *dst, const char *src) {              // function signature
    static char *(*fptr)(char *, const char *)=NULL;
                           // cached pointer to the next function definition
    int curent;                                         // used for logging

    /* lookup next function */
    AUDLIB_LOOKUP_COMMAND(char *(*)(char *, const char *),"strcpy",fptr,NULL);

    /* generate audit information */
    if (attack_initialized) {
        curent = log_newent("strcpy");        // create a new record

        /* log each argument */
        audlib_log_dst(curent, dst);          // log the dest
        audlib_log_str(curent, src);          // log the source

        log_commitent(curent);                // commit to disk
    }

    return((*fptr)(dst,src));                 // call the actual function
}
```

Figure 3. An example of an interposing function in an AUDLIB library.

DOI: 10.1002/spe

because of their bright covers). The *Orange Book* [11] describes the necessary characteristics for certifying the 'trust level' of an operating system. Together with the *Guide to Understanding Audit in Trusted Systems* (the *Tan Book* [12]), these documents describe the necessary characteristics for auditing and audit data generation in a trusted system[‡].

To be eligible for purchase and use by governmental organizations, commercial operating systems were augmented with audit systems based on the specifications and testing methodologies outlined in the Orange Book. One such system is the SunSHIELD Basic Security Module or BSM [1]. In GNU/Linux, there have been a number of projects to bring BSM-style auditing to the platform including [15–17]. Starting with the 2.6.× kernel, the Linux Audit System [18] is now a standard feature. In the BSD family, there is now OpenBSM [19], and a similar system exists in Apple's OS X [20].

These systems focus on collecting information at the kernel level. This is useful for verification/enforcement of a Bell-LaPadula style policy (for document confidentiality), but as has been pointed out by others [21, 22], there is a desire or need for more information from a higher level in the system, and that existing audit information is insufficient [23]. Skilled attackers can hide their behavior in these trails of system calls [24], hence, we need better audit information.

One approach has been to do a 'deep inspection' of network packets and their contents (e.g. [25]). However, this requires us to potentially monitor a large amount of network traffic and attempt to infer which portions are going to be received by what programs. Our AUDLIB technique moves such responsibilities to the host and allows us to examine only the actual data/behavior of a program as well as the ability to monitor incidents that occur entirely on the host itself.

3.2. Related work

One similar project to AUDLIB is Janus [26, 27]. Janus uses the Solaris debugging facility to intercept all system calls performed by an executable. These are then compared against a policy file to determine their acceptability (Janus is intended to sandbox applications). If found to be in violation, either the call is prevented, an alert is raised, or both. This is a system that performs an interception of program behavior similar to ours, however it is restricted to system calls, which means that the information available will be similar to those from existing audit systems.

Yongzheng and Yap's monitor framework [28] requires a kernel modification for data collection and reporting, and a user-level API to allow configuration. The 'user-level' component is the configuration interface; the information collected is still at the kernel level.

Another related project is libsafe [29, 30]. Similar to AUDLIB, libsafe uses library interposition to intercept library calls that might result in a stack buffer overflow or a format string attack. However, it is narrowly focused on these two types of attacks, and is focused on detection/prevention rather than the generation of new audit data.

Finally, there are the network sniffers such as `tcpdump/libpcap` [31] that promiscuously collect network traffic. When these systems are monitoring a high-traffic network, they cannot process or receive the packets fast enough and end up 'dropping' packets and cannot process them. Analysis engines using this type of data (such as Snort [32]) sometimes have a high false alarm rate because they must infer that a potentially malicious payload will reach a vulnerable program. Additionally, they cannot monitor application-to-application interactions on a single host.

There are a number of variations on these types of systems—using debugging or kernel resources to monitor sequences of system calls, prevention of targeted application behaviors, or collection of network traffic. However, we are not aware of any system other than AUDLIB that is designed to collect and record application level audit data for use in security monitoring.

[‡]While the Orange Book was officially retired in 2000, these requirements were ported to the Common Criteria framework as CAPP [13] and LSPP [14].

DOI: 10.1002/spe

4. EXPERIMENTAL RESULTS

In this section we will detail the various testing that was performed on AUDLIB. For an audit system such as AUDLIB to be useful, it must be transparent to the normal operation of the audited programs. In Section 4.1, we describe our examination of interaction effects. Additionally, the audit system should not unduly degrade the performance of the computer on which it is deployed. In Section 4.2, we describe the performance impact of the system and compare our results against those from the Linux audit system `auditd` [2, 18] and Solaris `BSM` [1]. Finally, we verified that the information generated can actually be used to detect attacks, intrusions, and computer misuses. In Section 4.3, we describe our basic tests of the utility of information. Specifically, we verify that AUDLIB information can be used to detect various security-related events.

4.1. Non-interference testing

Previous experience with writing and deploying interposable libraries has shown that faults may be caused by interactions between an application and an interposing library, as well as interactions between multiple libraries being interposed simultaneously. (For example, an early prototype of our system encountered difficulties when the bash shell was started, as it attempted to close all possible file descriptors including the ones we had opened for logging.) We selected a number of non-trivial applications and verified that their behavior was not negatively impacted and that appropriate audit data was generated.

Additionally, we wanted to ensure that there were no interactions between the three libraries that make up AUDLIB, and that the order in which the libraries were applied did not matter. We set up a script (`aud_run`) that would repeatedly run a program using all combinations and orderings of the AUDLIB libraries and tested with the maximum amount of logging enabled.

The programs we tested with AUDLIB included the following:

- Quake3
- Firefox
- Eclipse
- The GIMP
- Inkscape
- OpenOffice
- GCC (Linux kernel compile)
- Apache web server

We also configured our X sessions and all child processes to be audited by AUDLIB and performed several days of development in this state with no observed problems. While we cannot state definitively that there will be no negative interactions, considering the complexity of our test programs and the results of our longer-term deployment test, we believe that AUDLIB will not interfere with the normal operations of the vast majority of applications.

4.2. Performance testing

The next area of concern is the performance impact of enabling auditing. We ran our application benchmarks and then examined the specific sources of performance overhead by running a system benchmark against various configurations of auditing systems. These measurements were taken on Linux systems equipped with 3.20 GHz Intel Pentium 4 processors with 1GB RAM running a Linux 2.6 kernel. The machines were in a closed lab with limited system processes and no external network activity.

To generate upper bound figures for performance cost, AUDLIB was configured to record all monitored events from all programs without any filtering. The numbers below thus represent the *worst-case* performance scenario. Through configuration and filtering, these costs can be markedly reduced.

4.2.1. Linux—Quake3. Our first performance test was '`demo 4`' inside the `timedemo` system of Quake3. This is a pre-recorded sequence of moves through a level in the game. Once started, it operates without user interaction moving the character as fast as possible. The end result is the average number of frames per second (FPS) able to be rendered. While this score is heavily

DOI: 10.1002/spe

Table I. The average number of pages per second able to be served by an Apache web server on Linux under various auditing conditions.

	Baseline	Audlib	auditd (blocking)	auditd (non-block)
Pages/s	3464.58	2199.97	356.75*	1016.57*
% overhead		36.5	89.7	70.7

Shown here are the average response times of 5 sessions each requesting 10 000 pages, up to 10 concurrently. During the second iteration of 10k page requests, the Linux audit system would begin to dump messages to the console and freeze/crash the system.
*Only one iteration shown.

dependent on the graphics card, AUDLIB is monitoring the data being used in buffers within the program itself.

The unmonitored application rendered 204 FPS and under AUDLIB, it was 204.7 FPS. This slight performance *improvement* is an anomaly we have previously noticed and believe it may be caused by caching from the use of interposing libraries.

4.2.2. Linux—apache webserver. A more realistic test of the intended deployment of AUDLIB is the wrapping of a commonly attacked system server application. We installed the Apache web server package on our Linux system and activated it in the default configuration. We then used the supplied Apache benchmarking tool `ab` to take our measurements. This benchmark measures web server performance by making a sequence of requests as rapidly as possible and reports on the average number of page fetches performed per second. To avoid affecting results, `ab` was run on an identical machine, connected via a hub to the server. Only the web server process and its children were subjected to auditing.

We compared the performance of AUDLIB with all logging enabled to that of the Linux audit system (`auditd` in the charts). The Linux audit system can be configured to operate in a *blocking mode* where a process generating an auditable event is suspended until the kernel is ready to record that record or a *non-blocking mode* where any audit records not able to be handled by the kernel are simply dropped and the process continues. We tested against both configurations.

Our tests were up to 10 concurrent page requests with a request size of 10 000 pages. We ran each test 5 times and calculated the average of all the tests[§]. Table I on the next page shows the results and the percent overhead incurred by each audit system. We also measured the overhead of AUDLIB on the Apache benchmark for 400 000 and 800 000 page requests. In both configurations, the results were nearly identical to those above with AUDLIB having a consistent 36.4% overhead.

While AUDLIB has a significant overhead cost, it is at least half the overhead of the existing audit source in either the blocking or non-blocking modes. Additionally, AUDLIB is supplying information that can be used to look for buffer overflows and other attacks against the web server process itself. AUDLIB's performance cost is likely caused by the recording of all reads and writes—something Linux's audit system has disabled by default because of the volume of events.

4.2.3. Solaris—Apache webserver. An earlier version of AUDLIB was deployed to a Sun Microsystems SunFire v100 with 256MB of memory, using Solaris 9 with trusted mode enabled (necessary for enabling BSM auditing). The Apache server benchmark was run requesting 10 000 pages non-concurrently under various auditing conditions. Table II on the following page shows the results.

In this setting, we examined the performance of each individual library against that of the BSM audit system. Here we can see clearly that the largest overhead is coming from the misuse library.

[§]In the middle of the second set of 10 000 page requests, the Linux audit daemon would begin to print console messages about dropped audit records and then freeze the system. This occurred in both blocking and non-blocking modes.

DOI: 10.1002/spe

Table II. The average number of pages per second able to be served by an Apache web server running on Trusted Solaris 9 under various auditing conditions.

		Audlib			
	Baseline	Attack	Intrusion	Misuse	BSM
Pages/s	108.9	108.6	108.5	94.9	77.3
% overhead		0.3	0.4	12.8	29.0

Shown here are the average number of page fetches per second over 10 000 requests.

Table III. The baseline component and index score for the BYTEBench 4.1 benchmark.

		Audlib				Linux auditd	
Test	Baseline	Attack (%)	Intr. (%)	Misuse (%)	All 3 (%)	normal (%)	w.files (%)
Dhrystone	331.4	99.8	98.9	97.4	99.8	100.2	98.0
Whetstone	140.8	96.7	98.4	96.9	91.5	98.4	93.3
Execl throughput	434.8	37.1	32.5	40.7	21.8	12.3	12.5
File copy	876.3	107.2	106.3	24.2	24.2	1.6	1.6
Pipe throughput	257.3	101.6	101.6	11.7	11.6	1.1	1.1
Concurrent scripts	31.7	94.6	94.6	94.6	94.0	57.7	57.7
Index score	276.7	87.1	81.5	32.9	28.6	7.0	6.9

To the right is the percentage of the baseline that each configuration of audit system received. A higher value is better.

Once more, the performance overhead of AUDLIB is less than half of the overhead of the existing kernel level audit source.

4.2.4. Linux—BYTEBench. To get a better idea of what the possible source of performance overhead is, we tested various auditing configurations under a Unix benchmark suite. We used the BYTE Unix Benchmark version 4.1 [33] for our measurements. Table III shows the results.

If we first examine the index scores, we see that under AUDLIB the attack library has an overhead of 12.9%, the intrusion library is 18.5%, and the misuse library is again the largest at 67.1%. As one might expect, when you combine the three libraries, the overhead of 71.4% is worse than that any of individual library.

The Linux audit system was configured to monitor only the actions of the benchmark program and was supposed to record all audit records. To further examine the effect of the monitoring of reads and writes to files, we also ran measurements against `auditd` with those system calls being recorded. This change seemed to have only a minor impact on performance, but a significant change in the size of audit files generated. While the system did not crash this time, we saw in the `syslog` records that `auditd` was dropping a large number of records both with and without the filesystem monitoring. Looking at the index scores, we see that `auditd` has a 93% overhead—the throughput performance is only one quarter of the slowest AUDLIB configuration.

If we examine the sources of AUDLIB's overhead, we notice that the `execl` throughput is low in all 3 of the monitoring libraries. This test is a program `exec()`ing itself as rapidly as possible for 30 seconds. The larger the score of a test in the baseline, the more the index score is influenced by any performance change. The low scores make sense considering that the initialization routines from AUDLIB are called for each `exec()`, and these routines make a number of system call requests to collect the information reported as well as setup communication channels, and this causes a number of context switches. The initialization function is probably the most performance-intensive operation in AUDLIB. As each library is designed to operate on its own, combining the libraries triples the number of context switches required; this could be reduced with a redesign.

DOI: 10.1002/spe

The other areas that have a significant performance overhead for the misuse library are the file copy and pipe throughput tests. In both of these cases, there are a large number of reads and writes, and AUDLIB is logging each event along with information about the buffer involved. In the case of the pipe throughput test, we were auditing both sides of the connection resulting in twice as many events. We reiterate that these results are the worst-case performance (no filtering) under a pathological test case.

4.3. Detection testing

To verify that information generated by AUDLIB can be used for security monitoring, we needed to generate audit logs containing events that we wanted to detect and verify that a system with AUDLIB could detect them.

To generate the attack data, we created a number of sample attacks in the following manner:

- *Format string attacks:* To test that we were able to supply information needed to detect a format string attack, we constructed a small, vulnerable program and used a variation of one of the listed attacks described in [34] to exploit the vulnerability.
- *Buffer overflow attacks:* To create some buffer overflow attacks, we modified the sample vulnerable program and exploit from [35] and also tried a number of student submissions from an assignment in our Information Security course.
- *Return to libc:* A technique that is used to get around some buffer overflow detection methods is the `return-to-libc` attack [36]. To test this, we generated our own vulnerable program and implemented an exploit against the vulnerability.
- *Race conditions:* To test TOCTTOU attacks [37], we created a vulnerable program with an `access()–open()` race condition, installed it as a setuid program and then wrote a program to exploit that vulnerability.

We then created a set of scripts that would use `audlogread` to process AUDLIB logs and then perform some basic security monitoring. The various buffer overflow and format string attacks were detected by searching for patterns of shellcode, NOP sleds, and targeted payloads from the Snort rule set [32]. A simple additive score was calculated based on the size of the buffer, the payload in the buffer, the destination of the buffer (stack, heap, etc.), and if that memory was marked as executable. A simple race condition (TOCTTOU) detector was constructed by monitoring processes for access of the same symbolic pathname. Currently it is only looking for `access()–open()` sequences, but can easily be extended to look for more. The intrusion parser processes the logs and extracts information specified by Lane and Brodley [8], Maxion [9] and others to be useful in detecting masquerading users. Finally, our misuse parser is set up to report on any file permission changes or file deletions.

4.4. Audit information utility

As previously mentioned, AUDLIB contains information that is not present in typical kernel-level audit systems such as BSM. Audit records generated in the kernel contain information as resolved by the operating system while AUDLIB operates and records information at the application level. For example, AUDLIB records the symbolic filenames requested by a program while BSM records the information of the resolved pathnames. Also, AUDLIB is able to observe and record information about events that never reach the kernel such as string operations.

Portions of a pre-release version of AUDLIB were used as an information source in a system administration tool named *Trackle* [38]. It uses components of AUDLIB (mostly the computer misuse portion) to track the actions of a system administrator while they are correcting some problem so that these activities can be incorporated within the bug report itself, within the bug tracking system. All commands executed, including all arguments, are logged. All file changes made, even as side effects are recorded and are available as diffs within the bug/resolution report itself. While not strictly a security-focused application, the additional monitoring demonstrates the usefulness of AUDLIB's approach.

DOI: 10.1002/spe

4.5. Limitations of interposition

As with any security tool, it is important that users are aware of the limitations of AUDLIB. Because AUDLIB uses library interposition to intercept function calls, there are a few issues that we wish to clarify.

First, AUDLIB can only generate data from dynamically linked executables. Statically linked executables do not use the dynamic linker and therefore will not be fully interposed. Static linking incorporates the library's executable code directly into compiled binary, and AUDLIB would generate no audit record from those calls. However, on modern systems, these appear to occur infrequently. On our Linux distribution, of the 303 linked programs in `/sbin` or `/usr/sbin`, only 9 of them were statically linked. Most were copies of system tools and named in the fashion `fsck.static`. Out of the 1491 linked applications in `/usr/bin` only 2 were statically linked. If necessary, a static version of AUDLIB could be linked into any such programs.

Similarly, while AUDLIB generates information about unbounded buffer operations that use standard library functions, it is not able to report on operations that do not use a library routine. If a program has implemented its own version of `strcpy()`, AUDLIB cannot report on its use.

Finally, when used in the dynamic mode (via the environment variable `LD_PRELOAD`), it is possible for an application to disable auditing on child processes by changing the value of that variable before executing. For instance, `sudo` clears most environment variables before running the requested command. Using `/etc/ld.so.preload` is not affected by such evasion techniques.

4.6. Attacks on AUDLIB itself

Interposed libraries have the capability to dramatically change or subvert the behavior of a program as they essentially take control of a program for the duration of any interposed call. As such, it is worth considering the possibility of a malicious actor subverting AUDLIB itself. If installed as described in Section 2.4, then AUDLIB is subject to the same protections as `libc`, and any modifications that might be made to AUDLIB could have been made to any of the shared objects or even the program itself.

Another avenue of attack that a malicious user might take is to modify the library search path to cause their own library to be found when our interposing library asks the linker/loader to find the next copy of a given function. However, their version of the function will either run under their uid or be ignored. Privileged programs (such as those with the setuid bit set) ignore user-settable library search paths and only follow the system default set. It is our belief that neither case introduces any additional vulnerability or threat to the system.

5. APPLICABILITY OF THIS TECHNIQUE OUTSIDE OF SECURITY

While we have used library interposition to collect security-relevant information, this technique can be applied to a variety of purposes of interest to software developers and system administrators. We will briefly discuss some of those applications here. Some of these topics are covered in greater detail in [5–7]; however, we want to give readers a sense of the wide range of possible uses for this technique.

Interposition is useful for profiling the behavior of applications and libraries. By transparently interposing a function, a developer can generate a count of the number of times it is called. By instrumenting the interposing call with timestamps, it is possible to determine the total length of time spent in a function, and analyze the distribution of service times incurred. The arguments are also available for examination. Nakhimovsky [5] describes using interposition to create a histogram of the size of memory blocks requested by `malloc()` and other dynamic memory allocation requests.

Another area that is amenable to the use of interposition is that of dependency analysis. Interposition can uncover hidden dependencies within a program, some of which might come from

DOI: 10.1002/spe

library calls indirectly accessed by the libraries needed by the program. By interposing calls such as `getenv()` and `putenv()`, one can determine what environment variables are being used by a program or are being set before the spawning of a subprogram.

Interposition is also useful in debugging. Data fields can be examined as they are passed from function to function, which allows changes to be monitored as execution continues. In addition to allowing monitoring of program activity, an interposed library can also dynamically change the behavior of a program. This can allow a developer to introduce faults into a program. This can be useful for expanding the range of code covered by test cases, testing and debugging of error-handling routines, or examining the robustness of systems through the introduction of data corruption.

6. CONCLUSIONS AND FUTURE WORK

In this paper we have presented a new auditing tool called AUDLIB that is able to generate application-level audit data. Although the general idea of wrapping/interposition has been around for a number of years, to the best of our knowledge, this is the first implementation of an extendable, application level audit data source for use in security monitoring.

We have described how this system is able to collect information from existing systems and operate in a dynamic manner. AUDLIB can be used to monitor an entire system or be targeted to only a few security-relevant processes. We have explained the design behind the various libraries and described the information recorded.

We have explained that the application-level information provided contains information not available from the kernel or network levels such as format strings, actual buffer content information, and intended file names. AUDLIB does not interfere with the operations of other audit sources and can be used to augment the overall system audit trail. Furthermore, the system can be extended in a number of ways without requiring changes to either the kernel or applications being monitored.

We have measured the performance impact of AUDLIB and compared it with that of the existing audit systems on Linux and Solaris. It is reasonable to expect that a small change in the performance for a particular application is justified when there is need for extra auditing, and is not likely to be noticeable by the user. Additional processor speed and caching can often make the difference even less noticeable. AUDLIB's throughput performance was 2–4 times that of the Linux audit system and had half the overhead of Solaris BSM.

Looking forward, we hope to continue improving AUDLIB especially in the areas of configuration and filtering. With feedback from the security community, we plan on incorporating additional audit information into the system and port the software to similar systems. Additionally, AUDLIB or a similar system could be employed to gather additional information from honeypots or VM systems as logical extensions of them.

While we have used library interposition to collect security-relevant information, this technique can be applied to a variety of purposes. It allows an administrator to transparently monitor the activities of a program, and optionally, an interposed library can also dynamically change its behavior. Furthermore, this technique allows multiple systems to be composed together or used individually, all without requiring changes to the monitored program or the underlying operating system.

More generally, the design of AUDLIB is general, and it should be possible to extend it to other systems that use some form of dynamic libraries. Development of some standardized formats for reporting might even occur, thus allowing some comparison among different versions of software ported to multiple systems.

7. AVAILABILITY

Audlib will be available for download from `http://www.cs.oberlin.edu/~kuperman/research/audlib.html` and the CERIAS archive at `ftp://ftp.cerias.purdue.edu/`; it is being released under an open-source license.

DOI: 10.1002/spe

APPENDIX A: INTERPOSED FUNCTIONS

In this appendix, we briefly list the various functions that are interposed by the components of AUDLIB. Each of the components have their own individual initialization and finalization routines. Further, they monitor attempts to close the file descriptor being used for logging and report failure back to the calling program. For logging purposes, they watch for `fork()` calls as that will change their process id. For more information on the general format of the data collected, see Section 2.2.

A.1. Detection of attacks

The initialization routine records the following information:

- Real user id
- Effective user id
- Saved user id
- Real group id
- Effective group id
- Saved group id
- The current working directory
- The number of command line arguments
- The full command line
- The current environment

Calls with information useful for the detection of buffer overflow attacks:

- `fscanf()`
- `gets()`
- `getwd()`
- `realpath()`
- `scanf()`
- `sprintf()`
- `sscanf()`
- `strcat()`
- `strcpy()`
- `vfscanf()`
- `vscanf()`
- `vsprintf()`
- `vsscanf()`
- `wcpcpy()`
- `wcscat()`
- `wcscpy()`

Calls with information useful for the detection of format string attacks:

- `fprintf()`
- `printf()`
- `snprintf()`
- `sprintf()`
- `vfprintf()`
- `vprintf()`
- `vsnprintf()`
- `vsprintf()`

Calls with information useful for the detection of TOCTTOU attacks:

- `access()`
- `acl()`
- `chdir()`
- `chmod()`
- `chown()`
- `chroot()`
- `creat()`
- `creat64()`
- `execl()`
- `execle()`
- `execlp()`
- `execv()`
- `execve()`
- `execvp()`
- `faccessat()`
- `fchmodat()`
- `fchownat()`
- `fstatat()`
- `fstatat64()`
- `futimesat()`
- `getcwd()`
- `getwd()`
- `lchown()`
- `link()`
- `linkat()`
- `lstat()`
- `lstat64()`
- `mknod()`
- `mknodat()`
- `mount()`
- `open()`
- `open64()`
- `openat()`
- `openat64()`
- `rename()`
- `renameat()`
- `stat()`
- `stat64()`
- `statvfs()`
- `statvfs64()`
- `symlink()`
- `symlinkat()`
- `umount()`
- `umount2()`
- `unlink()`
- `unlinkat()`
- `utime()`
- `utimes()`

A.2. Detection of intrusions

The initialization routine records the following information:

- The current username
- Real user id
- Effective user id
- Saved user id
- Real group id
- Effective group id
- Saved group id
- The full command line as specified by the user
- The results of `stat()` on the executable
- The results of `lstat()` on the executable

DOI: 10.1002/spe

The finalization routine records the following information:

- The current username
- Real user id
- Effective user id
- Saved user id
- Real group id
- Effective group id
- Saved group id
- The full command line as specified by the user
- The results of `stat()` on the executable
- The results of `lstat()` on the executable
- The amount of time spent executing user instructions
- The amount of time spent executing by the operating system on behalf of the process
- The sum of the time spent executing user instructions by child processes
- The sum of the time spent executing by the operating system on behalf of all child processes

Calls with information useful for recording what programs are executed:

- `execl()`
- `execle()`
- `execlp()`
- `execv()`
- `execve()`
- `execvp()`

A.3. Detection of computer misuse

The initialization routine records the following information:

- Real user id
- Effective user id
- Saved user id
- Real group id
- Effective group id
- Saved group id
- The results of `stat()` on the executable
- The results of `lstat()` on the executable

The finalization routine records the following information:

- Real user id
- Effective user id
- Saved user id
- Real group id
- Effective group id
- Saved group id
- The amount of time spent executing user instructions
- The amount of time spent executing by the operating system on behalf of the process
- The sum of the time spent executing user instructions by child processes
- The sum of the time spent executing by the operating system on behalf of all child processes

Calls with information useful for recording what files and file contents were accessed:

- `open()`
- `openat()`
- `open64()`
- `openat64()`
- `close()`
- `creat()`
- `creat64()`
- `chmod()`
- `fchmod()`
- `fchmodat()`
- `chown()`
- `fchownat()`
- `lchown()`
- `fstatat()`
- `fstatat64()`
- `stat64()`
- `lstat()`
- `lstat64()`
- `fstat()`
- `unlink()`
- `unlinkat()`
- `truncate()`
- `ftruncate()`
- `fopen()`
- `fopen64()`
- `fclose()`
- `rename()`
- `renameat()`
- `tmpfile()`
- `mktemp()`
- `mkstemp()`
- `access()`
- `faccessat()`
- `acl()`
- `execv()`
- `execve()`
- `execvp()`
- `facl()`
- `link()`
- `mknodat()`
- `statvfs()`
- `statvfs64()`
- `utime()`
- `utimes()`
- `futimesat()`
- `dup2()`
- `dup()`
- `write()`
- `pwrite()`
- `read()`
- `pread()`

DOI: 10.1002/spe

- `fchown()`
- `symlink()`
- `linkat()`
- `stat()`
- `symlinkat()`
- `mknod()`

Additional information regarding the content and specific format of the generated audit records can be found in Section 2.2 and the documentation for AUDLIB.

ACKNOWLEDGEMENTS

Our thanks to the following for their assistance with portions of the project:

Undergraduate students: Mustafa Paksoy—Swarthmore College class of 2007—Summer 2005; Nick Hatt and Axis Sivitz—Oberlin College class of 2008—Summer 2007; Thomas Ramfjord—Oberlin College class of 2011—Summer 2009.

Sysadmins: Adam Hammer—Purdue University; Jeff Knerr—Swarthmore College; Nate Daniels—Oberlin College.

Our thanks to the Booth Ferris Foundation, The National Science Foundation, and partners of CERIAS at Purdue University for their financial support of this project. This material is based upon work supported by the National Science Foundation under Grant No. EIA-9903545.

REFERENCES

1. Sun Microsystems, Inc., Palo Alto, CA, U.S.A. *SunSHIELD Basic Security Module Guide*, 2000. Available at: http://docs.sun.com/app/docs/doc/806-1789 [March 2004].
2. Grubb S. auditd(8): The Linux audit daemon. Red Hat, February 2007.
3. Kuperman BA, Spafford E. Generation of application level data via library interposition. *Technical Report CERIAS TR 1999-11*, COAST Laboratory, Purdue University, West Lafayette, IN, October 1999. Available at: https://www.cerias.purdue.edu/assets/pdf/bibtex_archive/99-11.pdf [June 2008].
4. Curry TW. Profiling and tracing dynamic library usage via interposition. *USENIX Summer 1994 Technical Conference*, Boston, MA, 1994; 267–278.
5. Nakhimovsky G. Building library interposers for fun and profit. *Unix Insider*, July 2001. Available at: http://developers.sun.com/solaris/articles/lib_interposers.html [March 2004].
6. Jones MB. Interposition agents: Transparently interposing user code at the system interface. *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, Asheville, NC, 1993.
7. Sun Microsystems, Inc., Mountain View, CA. *Shared Library Interposer* (*SLI*) *User's Guide*. SLI version 1.2 edn., January 1996.
8. Lane T, Brodley CE. Temporal sequence learning and data reduction for anomaly detection. *ACM Transactions on Information and System Security* (*TISSEC*) 1999; **2**(3):295–331. DOI: http://doi.acm.org/10.1145/322510.322526.
9. Maxion RA. Masquerade detection using enriched command lines. *Proceedings of the 2003 International Conference on Dependable Systems* (*DSN-03*), San Francisco, CA, 2003; 5–14. DOI: http://doi.ieeecomputersociety.org/10.1109/DSN.2003.1209911.
10. Denning DE. An intrusion-detection model. *IEEE Transactions on Software Engineering* 1987; **SE-13**(2):222–232.
11. US Department of Defense. Trusted computer systems evaluation criteria. *Technical Report DoD 5200.28-STD*, DoD Computer Security Center, Fort Meade, MD, December 1985.
12. US Department of Defense. A guide to understanding audit in trusted systems. *Technical Report NCSC-TG-001*, *Version 2*, National Computer Security Center, Fort George G. Meade, MD, June 1988.
13. National Security Agency. Controlled Access Protection Profile, October 1999. Available at: http://www.radium.ncsc.mil/tpep/library/protection_profiles/CAPP-1.d.pdf, version 1.d [March 2004].
14. National Security Agency. Labeled Security Protection Profile, October 1999. Available at: http://www.radium.ncsc.mil/tpep/library/protection_profiles/LSPP-1.b.pdf, version 1.b [March 2004].
15. The Linux Basic Security Module Project, 2003. Available at: http://linuxbsm.sourceforge.net/ [June 2008].
16. InterSect Alliance. SNARE for Linux, Available at: http://www.intersectalliance.com/projects/SnareLinux/ [June 2008].
17. Wolfe W, Godinez J. Secure Auditing for Linux, February 2003. Available at: http://secureaudit.sourceforge.net/ [June 2008].
18. Grubb S. Linux Audit System, 2008. Available at: http://people.redhat.com/sgrubb/audit/ [June 2008].
19. Watson R. OpenBSM: Open Source Basic Security Module (BSM) Audit Implementation. Available at: http://www.openbsm.org/ [June 2008].
20. Apple Computer, Inc. *Common Criteria Configuration and Administration Guide*. Version 1.0.1 edn., April 2005. Available at: http://www.apple.com/support/security/commoncriteria/ [June 2008].
21. Lunt TF. Detecting intruders in computer systems. *Proceedings of the 1993 Conference on Auditing and Computer Technology*, 1993.
22. Price KE. Host-based misuse detection and conventional operating systems' audit data collection. *Master's Thesis*, Purdue University, West Lafayette, IN, December 1997.

23. Barse EL, Jonsson E. Extracting attack manifestations to determine log data requirements for intrusion detection. *Proceedings of the 20th Annual Computer Security Applications Conference* (*ACSAC'04*), Tucson, AZ, 2004; 158–167. DOI: http://doi.ieeecomputersociety.org/10.1109/CSAC.2004.20. Available at: http://www.acsac.org/2004/abstracts/127.html.
24. Wagner D, Soto P. Mimicry attacks on host-based intrusion detection systems. *Proceedings of the 9th ACM Conference on Computer and Communications Security* (*CCS 2002*). ACM: New York, NY, 2002; 255–264. DOI: http://doi.acm.org/10.1145/586110.586145.
25. Wang K, Stolfo SJ. Anomalous payload-based network intrusion detection. *Recent Advances in Intrusion Detection* (*RAID2004*) (*Lecture Notes in Computer Science*, vol. 3324). Springer: Sophia Antipolis, France, 2004; 203–222. DOI: 10.1007/b100714.
26. Goldberg I, Wagner D, Thomas R, Brewer E. A secure environment for untrusted helper applications (confining the Wily Hacker). *Sixth USENIX Security Symposium*, *Focusing on Applications of Cryptography*, San Jose, CA, 1996. Available at: http://www.cs.berkeley.edu/~daw/janus/ [June 2008].
27. Wagner DA. Janus: An approach for confinement of untrusted applications. *Technical Report UCB/CSD-99-1056*, EECS Department, University of California, Berkeley, 1999. Available at: http://www.eecs.berkeley.edu/Pubs/TechRpts/1999/5271.html [June 2008].
28. Wu Y, Yap RHC. A user-level framework for auditing and monitoring. *Proceedings of the 21st Annual Computer Security Applications Conference* (*ACSAC'05*), Tucson, AZ, 2005; 95–105. DOI: http://doi.ieeecomputersociety.org/10.1109/CSAC.2005.8. Available at: http://www.acsac.org/2005/abstracts/132.html [June 2008].
29. Tsai T, Singh N. Libsafe 2.0: Detection of format string vulnerability exploits. *Technical Report ALR-2001-019*, Avaya Labs, Avaya Inc., Basking Ridge, NJ, August 2001. Available at: http://www.research.avayalabs.com/techreport/ALR-2001-019-paper.pdf [March 2004].
30. Tsai TK, Singh N. Libsafe: Transparent system-wide protection against buffer overflow attacks. *DSN '02*: *Proceedings of the 2002 International Conference on Dependable Systems and Networks*. IEEE Computer Society: Washington, DC, U.S.A., 2002; 541.
31. tcpdump/libpcap. Available at: http://www.tcpdump.org/ [June 2008].
32. Roesch M. The Snort Developers. Snort®. Available at: http://snort.org/ [June 2008].
33. Niemi DC. BYTE Unix Benchmarks, Version 4.1, July 1999. Available at: http://www.tux.org/pub/tux/niemi/unixbench/ [June 2008].
34. Scut. Exploiting format string vulnerabilities. *Technical Report*, Team Teso, September 2001. Available at: http://teso.scene.at/articles/formatstring [March 2004].
35. Aleph One. Smashing the stack for fun and profit. *Phrack Magazine*, *Fall* 1997; **7**(49):File 14 to 16.
36. Solar Designer. Getting Around Non-Executable Stack (and Fix). Posting to BugTraq mailing list, August 1997. Available at: http://seclists.org/bugtraq/1997/Aug/0063.html [June 2008].
37. Bishop M, Dilger M. Checking for race conditions in file accesses. *Computing Systems Spring* 1996; **9**(2):131–152.
38. Crosta DS, Singleton MJ, Kuperman BA. Fighting institutional memory loss: The trackle integrated issue and solution tracking system. *Proceedings of LISA '06*: *20th Large Installation System Administration Conference*. USENIX Association: Washington, DC, 2006; 287–298. Available at: https://db.usenix.org/events/lisa06/tech/crosta.html [June 2008].

DOI: 10.1002/spe