Generation of Application Level Audit Data via Library Interposition

Benjamin A. Kuperman and Eugene Spafford
COAST Laboratory
Purdue University
West Lafayette, IN 47907–1398
{kuperman,spaf}@cs.purdue.edu
COAST TR–98–17

Abstract

One difficulty encountered by intrusion and misuse detection systems is a lack of application level audit data. In this paper we present a technique to automatically generate application level audit data using library interposition. Interposition allows the generation of audit data without needing to recompile either the system libraries or the application of concern. We created a library that detects some types of unsafe programming practices, and discovered two unreported race conditions in some common applications.

1 Motivation

[Pri97] examines misuse detection systems and the operating system audit data used by them. Focus is placed on the host based component of the selected systems. The surveyed systems used a variety of data input sources including BSM[1] audit logs, UNICOS[2] audit data, Application specific security scanners, and "Standard" UNIX accounting data.

Summarizing the results, [Pri97] notes that “application level audit data is often insufficient.” In general, the application developer must insert commands into the software to generate audit data . Unfortunately, many programmers do not include the desired audit generation routines in their programs.

It is desirable to increase the amount of useful audit data being generated from existing applications and operating systems without requiring the recompilation of deployed software. This paper describes a technique to automate the collection of additional application level information by interposing or modifying the system shared libraries. We examine certain unsecure programming practices [AUS96, GS91] which can be detected by monitoring the function calls made to the system shared libraries.

2 Interposition

Before discussing our approach, we need to clarify what we mean by *interposition.* Interposition is the “process of placing a new or different library function between the application and its reference to a library function”

[1] Sun Microsystem’s Basic Security Module
[2] Cray’s proprietary version of UNIX

[TWC94]. This technique allows a programmer to intercept function calls to code located in shared libraries by directing the dynamic linker to first attempt to reference a function definition in a specified set of libraries before consulting the normal library search path. This is useful for testing new libraries or for inserting debugging code. For a description of the exact procedure used to create a shared object[3], refer to [Sun94, TWC94]. [TWC94] contains a concise description of the run time process involving shared libraries.

On Solaris and Linux, a shared object can be interposed by setting the `LD_PRELOAD` environment variable before the execution of a program that we want to be interposed. When a function call is made that is undefined in the application, the dynamic linker will first check for definitions of this function in the objects listed in the `LD_PRELOAD` variable, and then check along the usual library search path. Figure 1 shows the sequence of comparisons made when a library call is encountered.

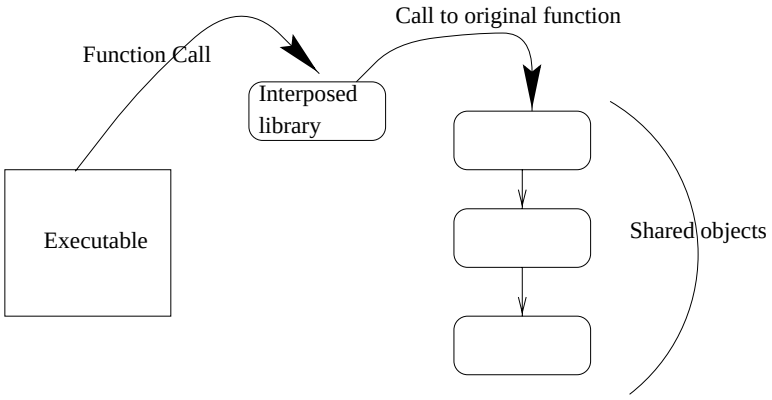

Figure 1. During the execution of an interposed program, undefined function calls are first referenced in the shared objects specified by `LD_PRELOAD` and then other objects are checked.

2.1 Why interposition?

Interposing on the system shared objects allows both the interception of certain function calls and the storage of state between such calls without requiring the recompilation of the executable. This allows the creation of a prototype to implement new audit data generation without changing the system shared libraries. Being a separate executable, an interposed library can be created without having access to the source code for the system shared libraries, or the executable on which interposition is to be performed. This can be useful because many commercial operating system and application vendors do not make the source code to their products available.

Since an interposed library is self–contained, one library can interpose on functions contained in many separate system libraries without concern of which physical files the original definitions are contained in. This simplifies the act of porting an interposing library across operating system versions as well as different flavors of operating systems.

[3]Technically, we are referring to shared objects, but a more common term is shared libraries. For this paper, we will use the two terms interchangably.

When an interposing library intercepts a function call, it has the ability to examine, record, and alter the arguments being passed to the call. The interposed function can prevent the intended function from being called, or choose to call a different function altogether. Additionally, the interposing function can examine, record, and modify the return value before returning control back to the user program.

An interposing library offers a persistent memory area in which state can be stored across multiple function calls. The interposing library can collect state data that might not have been available before because the original functions resided in separate shared libraries. One possible use of this state area is the counting of the frequency of access to a particular library function such as `crypt`. Another possible use is the logging of the total amount of certain data usage (memory allocations, descriptors requested and not used, filenames passed) for the duration of a program execution.

2.2 Why not static analysis?

Static analysis is a valuable tool for detecting problems prior to compilation [BD96], but does not give us access to audit information during the execution of a program. Static analysis is limited in cases where input comes from sources outside the control of the program (keyboard, configuration files, etc.) By examining arguments at run time, library interposition can examine data that comes from outside sources.

Static analysis often requires access to the source code. The analyzer needs to be able to span across multiple files and expand all macros and definitions in advance of the analysis. While this is not impossible, it becomes a much larger task when one has to consider shared objects or system libraries in addition to the application code. As noted above, source code may not be available for the executables of concern. Additionally, all function calls made within system libraries also need to be traced.

The interposed library collects data at execution time, allowing response or logging at the precise moment when conditions of interest occur during the execution of a program. By logging sufficient data at the time of the occurence, it may be possible to track down the source of a problem (crash, intrusion attempt, etc.) after its occurence. Conceptually, an interposing library can be used as a flight recorder approach to audit data collection.

2.3 What about strace and truss?

Many current incarnations of UNIX provide tools that allow the tracing of kernel calls. Common names for these tools include `trace`, `strace` and `truss` [GWTB96]. As these record only kernel calls, they miss many of the functions of interest such as those involved in buffer overflows. These tools give limited access to other data (e.g. environment variables) during run-time. These tools can be used to verify some of the problems that the interposed library might indicate (especially race conditions).

3 Previous and Related Work: Other usage of library interposition

Timothy W. Curry of Sun Microsystems used library interposition to generate a run time analysis of the X11 library functions [TWC94]. His goal was “to get useful performance data without special request placed on either the application or libraries.” His research led to the development of the Shared Library Interposer (SLI) toolset. [TWC94] discusses many pitfalls and solutions useful to the development of interposed shared libraries, and notes that shared library interposition possesses utility extending beyond graphic applications.

Other work has focused on the related topic of system (kernel) call interposition.

The *Janus* software was developed to address concerns regarding programs and helper applications which process unauthenticated network data, as well as the untrusted nature of the calling applications themselves. [GWTB96] describes the attempts to reduce this threat by construction of a user-mode, secure environment space to run such applications. *Janus* uses the Solaris process tracing facility to "intercept and filter dangerous system calls."

As described in [GWTB96], *Janus* is constructed around a framework that calls dynamic modules which are used to implement various aspects of the security configuration by filtering the relevant system calls. Part of the design criteria described is the use of *sandboxing* to restrict an untrusted helper application to a limited environment within which it is unfettered.

They discuss some of the difficulties of their implementation and offer possible solutions such as building this type of functionality into each application, and adding user level functions to allow the restriction of the runtime environment (`chroot` and `setuid` require super-user privileges at present). They also note that a wrapper approach (such as library interposition) is insufficient for restricting applications because it can be avoided by static linking.

[Jon93] describes an interposition toolkit that was developed that "substantially increases the ease of interposing user code between clients and instances of the system interface by allowing such code to be written in terms of the high–level objects provided by this interface, rather than in terms of the intercepted system calls themselves" for the Mach 2.5 kernel on Intel 386/486 and VAX architectures.

The focus of [Jon93] was that of system, or kernel calls. However, these concepts are extendable to interposition in some respects. The toolkit described was object oriented, and focused on the creation of *agents* to perform the interposition. The design was such that an agent could interact at different levels of the interface abstraction created by the toolkit.

Agents are described that emulate the 4.3 BSD system interface, change the apparent time of day, trace the execution of client processes, and simulate a "union" directory (separate directories appearing as merged). Major goals of [Jon93] included code reuse, toolkit development, low performance cost relative to the cost of the system call interception mechanism and the operations being emulated.

4 Implementation

Our interposing library was written in C and built with components of a structure similar to that shown in Figure 2. Items in all capitals are macros. Macros are used instead of function calls to reduce the impact of interposition by limiting the number of additional function calls that our object makes. A description of the interposing function for `strcpy` as shown in Figure 2 follows:

1. `AUDIT_CALL_START` (line 8) is placed at the beginning of every interposed function. This allows us to easily insert arbitrary initialization code into each function.

2. `AUDIT_LOOKUP_COMMAND` (line 10 in Figure 2, detail in Figure 3) performs the lookup of the pointer to the next definition of the function in the shared libraries using the `dlsym(3x)` command. By using the special flag `RTLD_NEXT` (Figure 3, line 2), we indicate that the next reference along the library search path used by the runtime loader will be returned. The function pointer is stored in `fptr` if a reference is found, or the error value is returned to the calling program.

3. Line 12 contains the commands that are executed before the function is called.

```
/*************************************
 * Logging the use of certain functions *
 *************************************/
char *strcpy(char *dst, const char *src) {
   char *(*fptr)(char *,const char *); /* pointer to the real function */
   char *retval;                       /* the return value of the call */

   AUDIT_CALL_START;

   AUDIT_LOOKUP_COMMAND(char *(*)(char *,const char *),"strcpy",fptr,NULL);

   AUDIT_USAGE_WARNING("strcpy");

   retval=((*fptr)(dst,src));

   return(retval);
}
```

Figure 2. An example of a function that is being used in the interposed library. Items in ALL CAPS represent macros that are defined elsewhere.

```
#define AUDIT_LOOKUP_COMMAND(t,n,p,e)
    p=(t)dlsym(RTLD_NEXT,n);
    if (p==NULL) {
        perror("looking up command");
        syslog(LOG_INFO,"could not find %s in library: %m",n);
        return(e);
    }
```

Figure 3. Instructions used in the macro to look up the next reference of a function in the remaining shared objects.

4. Although not required, we choose to execute the original function call, as-is and return the value to the user (line 14). Other possible actions include the examination, recording, or transformation of the arguments; the prevention of the actual execution of the library call; and the examination, recording, or transformation of the return value.

5. Additional code could be inserted before the result is returned (line 16), but this particular example has none inserted.

As our goal is the collection of additional audit data, the interposed library we created attempts to be transparent in its actions to the user. No attempt is made to stop known, dangerous practices, and in the event of a failure of our library (e.g. unable to find another reference to the same named function) the proper error value for that function is returned.

5 What were we looking for?

In this case, we had an idea of what information we wanted to generate from the system shared objects from looking at [GS91, AUS96]. We directed our efforts in this prototype toward the recording of certain functions that are commonly associated with security problems. The following lists briefly describe the items of concern and considerations used during the design phase.

- The effect of vulnerabilities often differ depending on the privilege level at which the process is running. Therefore, we set the priority to be increasingly more severe as functions were being called as
  - As a regular user (`EUID`$\neq 0$, `EUID` $=$ `UID`)
  - As a set user ID program (`EUID`$\neq 0$, `UID`$\neq$ `EUID`)
  - As a root process (`EUID`$= 0$)

  and use this to set our priority during our call to `syslog`.

- A frequently encountered class of vulnerabilities are those that involve the overrunning of static buffers. Certain library calls write to user allocated buffers, and do not perform any sort of length or overrun checking. Therefore, the use of these function calls should be avoided in general and especially by anything running with privileges higher than a regular user. The use of the following library functions were logged for non-regular user processes:

  - gets
  - strcpy
  - strcat
  - sprintf
  - vsprintf
  - scanf
  - sscanf

- Certain function calls are generally assumed to succeed, although an individual user might be able to affect the results. However, because they almost always are assumed to work, errors/faults might result. These are considered **exceptional conditions** in [AKS96]. Consequently, we record when any of the following categories of library calls fail:

- malloc family
- fork family
- dup family

- Race conditions can occur in file access sequences, and these may be present because POSIX does not include some of the needed system calls to prevent them [Ste92]. These are vulnerabilities even in non-SUID files because an ordinary user might be able to orchestrate an attack to affect other users (including privileged users) especially in scripts. We looked for the following known race conditions [BD96]:
  - stat → open
  - open → chown
  - open → chmod
  - open → stat

- Some function calls are inherently dangerous for SUID programs because they rely on external environment variables. These variables can be set by an ordinary user. We logged the use of any of the following calls running above regular user status:
  - system
  - popen
  - execlp
  - execvp

To provide more context for the audit data, several attributes are captured and logged on the first instance of audit data being recorded for a process. This data included the following:

- Actual User ID (`UID`)
- Effective User ID (`EUID`)
- Actual Group ID (`GID`)
- Effective Group ID (`EGID`)
- Original login name (via `getlogin`)
- Name of executable as specified on the command line
- All of the command line arguments specified by the user
- Process ID (`PID`)
- Time of logging

6 Results

Our interposing library was tested on a Redhat 5.1 Linux machine. To gather a basic set of audit data, we logged in as a regular user and tried to perform basic user functions. We read and sent mail, created and deleted files, and ran all of the SUID/SGID programs installed by default. We then switched to root and performed similar tasks.

No instances of programs using library functions that rely on external environment variables were found. Several small programs were written that tested each class of our auditing goals and the reporting occurred as expected. A large number of occurences of `sprintf, strcpy` and `strcat` were noticed in the normal file utilities. This could have been avoided because Linux has `snprintf` as a standard I/O routine, and `strncpy` and `strncat` are standard POSIX functions.

We also discovered two unreported race conditions within the first set of tests and verified them on a Solaris workstation. There has been substantial work explaining how race conditions work, and presenting techniques for preventing them [Bis95, BD96]. The following subsections describe these findings in greater detail.

6.1 Vim

The first race condition (or **synchronization error** [AKS96]) occurs in Vim version 5.1 and the relevant strace output can be seen in Figure 4.

```
1  ioctl(0, SNDCTL_TMR_START, {B38400 opost isig icanon echo ...}) = 0
2  write(1, "\"/tmp/testing/foo\"", 18)    = 18
3  brk(0x80c2000)                          = 0x80c2000
4  stat("/tmp/testing/foo", {st_mode=0, st_size=0, ...}) = 0
5  open("/tmp/testing/foo", O_RDONLY)      = 4
6  stat("/tmp/testing/foo~", 0xbfffb664)   = -1 ENOENT (No such file or directory)
7  unlink("/tmp/testing/foo~")             = -1 ENOENT (No such file or directory)
8  open("/tmp/testing/foo~", O_WRONLY|O_CREAT, 0666) = 6
9  chmod("/tmp/testing/foo~", 0644)        = 0
10 fchown(6, 4294967295, 100)              = 0
11 read(4, "\33[0m\33[0mfoo\33[0m\n\33[0mfoo"..., 8192) = 46
12 write(6, "\33[0m\33[0mfoo\33[0m\n\33[0mfoo"..., 46) = 46
13 read(4, "", 8192)                       = 0
14 close(6)                                = 0
15 utime("/tmp/testing/foo~", [98/09/03-17:21:52, 98/09/03-17:21:00]) = 0
16 close(4)                                = 0
17 open("/tmp/testing/foo", O_WRONLY|O_CREAT|O_TRUNC, 0666) = 4
18 write(4, "\33[0m\33[0mfoo\33[0m\n\33[0mfoo"..., 46) = 46
19 close(4)                                = 0
20 chmod("/tmp/testing/foo", 0100644)      = 0
21 write(1, " 3 lines, 46 characters written", 31) = 31
22 unlink("/tmp/testing/foo~")             = 0
23 write(1, "\r\33[?1l\33>", 8)            = 8
24 write(1, "\33[2J\33[?47l\0338", 12)     = 12
25 write(1, "\33[J", 3)                    = 3
26 close(5)                                = 0
27 unlink("/tmp/testing/.foo.swp")         = 0
28 _exit(0)                                = ?
```

Figure 4. A section of the truss output that shows the two race conditions that were discovered in the Vim 5.1 binary that is shipped with Redhat 5.1. The first occurs on lines 8 and 9 between the `open("/tmp/testing/foo~")` → `chmod("/tmp/testing/foo~")`**, and the second (which appears to be more dangerous) occurs on lines 17 and 20 with** `open("/tmp/testing/foo")` → `chmod("/tmp/testing/foo")`

The first race condition occurs between lines 8 and 9. Here the program is creating the temporary file that will be used as a backup of the existing file during editing. The second race condition occurs on lines 17 and 20 where the internal buffer is written out to the file. The second instance appears to be more dangerous because of the file write that takes place between the `open` and the `chmod`. A malicious user could move the file that is being

written during the write, and replace it with a symbolic link. The `chmod` would then follow the symbolic link to affect whatever file was referenced.

6.2 GNU cp

The second binary with synchronization errors was **cp** which is included in a set of file utilities that is widely deployed (GNU fileutils 3.16). Again the relevant strace output is shown in Figure 5.

```
1  getuid()                                        = 658 [658]
2  umask(0)                                        = 07
3  lstat("./passwd", 0xEFFFF6E8)                   Err#2 ENOENT
4  stat("/etc/passwd", 0xEFFFF4D0)                 = 0
5  open("/etc/passwd", O_RDONLY)                   = 3
6  open("./passwd", O_WRONLY|O_CREAT|O_TRUNC, 0600) = 4
7  fstat(4, 0xEFFFF348)                            = 0
8  fstat(3, 0xEFFFF348)                            = 0
9  read(3, " r o o t : x : 0 : 0 : S".., 8192)     = 175
10 write(4, " r o o t : x : 0 : 0 : S".., 175)     = 175
11 read(3, 0xEFFFD2E0, 8192)                       = 0
12 close(4)                                        = 0
13 close(3)                                        = 0
14 chmod("./passwd", 0100640)                      = 0
15 lseek(0, 0, SEEK_CUR)                           = 22865
16 _exit(0)
```

Figure 5. A section of the truss output that shows the race condition in GNU cp 3.16 The race condition occurs during the sequence `open("./passwd")` **(line 6)** $\rightarrow$ `chmod("./passwd")` **(line 14).**

The file that is created by the copy command is opened on line 6. The original file is completely read in, and completely written out before the permissions of the file are changed (line 14).

A possible attack using this vulnerability is as follows:

1. Attacker locates a SUID script that uses **cp**.
2. Attacker causes the **cp** to be reading from a slow device (e.g. floppy disk) and writing to a file under attacker control.
3. While the read/write cycle is taking place (lines 9–11), the attacker:
   (a) Moves the destination file to a different name.
   (b) Creates a symbolic link with the original destination file name pointing to a file to attack (e.g. /etc/shadow).
4. Waits until the `chmod` changes permission on the linked file (line 14).

A similar approach might work to exploit the synchronization error in **vim**.

It appears that adding this sort of auditing capability into the system shared objects will generate useful auditing information. If the changes are made to the system libraries, the overall impact should be minimized. Otherwise, the techniques described in [TWC94, GWTB96, Jon93, Sun94, Sun95] and others can be used to minimize the impact of an interposing library. As indicated in [GWTB96], an interposing library can be avoided by deliberate action on the part of the application programmer. Our implementation is intended to generate audit data, not provide a guarantee of security. One of the most important results is our ability to create this audit data without modifying either the system libraries or the executable programs.

7 Future Work

This area is far from being fully explored. Our interposing library was intended to be a prototype to allow us to verify the usefulness of such techniques. Our results indicate that this approach is useful. There are several steps we can take from this point.

- Consider how data will be logged. Currently, `syslog` is being used, however, since the control routines are user controlled, additional calls to `openlog` or `closelog` will redirect our messages.
- Find what additional data or states can be audited to gain useful information.
- Audit known IFS and PATH environment variable attacks.
- Consider longer term race condition testing (i.e. maybe on session level or across `exec`s).
- Look into making the library more user configurable rather than being hard coded.
- Create interposing libraries designed to work with a specific, known executable and utilize such knowledge for anomaly detection or prevention, and possibly to "sandbox" its execution. For example, restrict which directories sendmail might be able to write to, read from, or exec files in. This might be similar to the work in [GWTB96, LS98, FHS97].
- Expand our testing methodology to determine effects on system performance and our ability to detect known vulnerabilities or attacks.
- Integrate with the AAFID [BGFI+98] architecture.
- In general, find ways to incorporate this as a part of a larger security framework.

References

[AKS96] Taimur Aslam, Ivan Krsul, and Eugene H. Spafford. Use of a taxonomy of security faults. Technical Report TR-96-051, COAST Laboratory, Department of Computer Sciences, Purdue University, West Lafayette, IN 47907-1398, September 1996.

[AUS96] AUSCERT. *A Lab engineers check list for writing secure Unix code*, rev.3c edition, May 1996.

[BD96] Matt Bishop and Michelle Dilger. Checking for race conditions in file accesses. *Computing Systems*, 9(2):131–152, Spring 1996.

[BGFI+98] Jai Sundar Balasubramaniyan, Jose Omar Garcia-Fernandez, David Isacoff, Eugene Spafford, and Diego Zamboni. An architecture for intrusion detection using autonomous agents. Technical Report 98-05, COAST Laboratory, Purdue University, West Lafayette, IN 47907-1398, May 1998.

[Bis95] Matt Bishop. Race conditions, files, and security flaws; or the tortise and the hare redux. CSE-95-8, September 1995.

[FHS97] Stephanie Forrest, Steven A. Hofmeyr, and Anil Somayaji. Computer Immunology. *Communications of the ACM*, 40(10):88–96, October 1997.

[GS91] Simson Garfinkel and Gene Spafford. *Practical UNIX Security*. O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, USA, 1991.

[GWTB96] Ian Goldberg, David Wagner, Randi Thomas, and Eric Brewer. A secure environment for untrusted helper applications (confining the wily hacker). In *Sixth USENIX Security Symposium, Focusing on Applications of Cryptography*, San Jose, California, July 1996.

[Jon93] Michael B. Jones. Interposition agents: Transparently interposing user code at the system interface. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, Asheville, NC, December 1993.

[LS98] Wenke Lee and Salvatore J. Stolfo. Data mining approaches for intrusion detection. In *Proceedings of the 7th USENIX Security Symposium*, San Antonio, Texas, January 1998.

[Pri97] Katherine M. Price. Host-based misuse detection and conventional operating systems' audit data collection. Master's thesis, Purdue University, December 1997.

[Ste92] W. Richard Stevens. *Advanced Programming in the UNIX Environment*. Addison-Wesley, Reading, MA, USA, 1992.

[Sun94] Sun Microsystems. *Linker and Libraries (Solaris 2.4 Software Developer AnswerBook)*, 1994. SunOS 5.5.1.

[Sun95] Sun Microsystems. *C 4.0 User's Guide (Sun ANSI C Compiler-Specific Information)*, 1995. SunOS 5.5.1.

[TWC94] Sun Microsystems Timothy W. Curry. Profiling and tracing dynamic library usage via interposition. In *USENIX Summer 1994 Technical Conference*, pages 267–278, Boston, MA, Summer 1994.