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
CERIAS TR–99–11

29 October 1999

Abstract

One difficulty encountered by intrusion and misuse detection systems is a lack of application level audit data. Frequently, applications used are written by third parties and may be distributed only in a binary format. In this paper we present a technique to 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 interest. We created a library that detects some types of unsafe programming practices, and discovered two unreported race conditions in common applications.

A prototype interposition library that dynamically detects and prevents some forms of buffer overflow attacks is also introduced. This second prototype library was able to successfully detect and prevent several buffer overflow attacks against privileged programs.

1 Motivation

Researchers in Intrusion detection have stated (Kumar [1], Lunt [2], Price [3]) that there is a desire or need by software developers in the intrusion detection community for an increase in the amount of application level audit data available for their use. Frequently, applications report audit information only when their programmers insert specific instructions into the source code. Many of the the programs that generate such data (e.g. tcp_wrappers, login, NFR, COPS) are already designed as part of an intrusion or misuse detection system (see Mukherjee, Heberlein, and Levitt [4] for further definitions). We wanted to generate more application level audit data to meet this need, preferably with legacy applications and operating systems, including those not designed with auditing in mind.

We selected a technique called *library interposition* (explained in Section 2) that allows us to control the interactions between a dynamically linked executable and the shared objects used. We elected to work in the well known and understood[1] problem space of *unsafe* UNIX *programming practices* (AUS [5], Garfinkel and Spafford [6]).

We designed and built a prototype interposition library that would report on some of the known unsafe practices frequently encountered in software. Our results (discussed in Section 6) show that our prototype does generate the desired information for the selected problem space and consequently helps us achieve our

[1] Unfortunately, this problem space is the source of many of the new attacks on existing systems despite the amount of attention that has been given to it.

primary goal of increasing the amount of application level audit data. A second prototype was designed and built to demonstrate how this technique could be used to help enforce a security policy.

2 Interposition

Interposition is the "process of placing a new or different library function between the application and its reference to a library function" [7]. 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[2], refer to Curry [7], Sun [8]. Curry [7] 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.

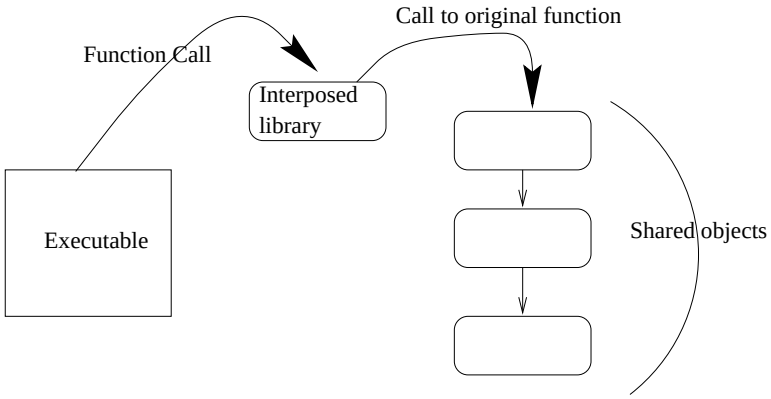

Figure 1: During the execution of an interposed program, undefined function calls are first referenced in the shared objects specified by `LD_PRELOAD` and then other objects are checked.

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 or shared objects. This allows us to create a prototype library that generates the new audit data 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.

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

This is useful because many commercial operating system and application vendors do not make the source code to their products available.

One interposing library can interpose on functions defined 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.

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. This can be used to keep track of the number of times a particular function was used (e.g. `crypt`), recording file access requests for detecting race conditions, or 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.

Taken together, the examination, recording, altering, and storage abilities allows us to monitor and enforce a policy on the usage of dynamically linked functions.

2.2 Why not static analysis?

Static analysis is a valuable tool for detecting problems before compilation [9], 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 external 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 interposing 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 enough data at the time of the occurrence, it may be possible to track down the source of a problem (crash, intrusion attempt, etc.) after its occurrence.

Static analysis is a valuable tool, but it fails to meet our goal of increasing the amount of application level data, and it can present spurious warnings about the use of poor programming techniques in our specific problem domain space.

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` [10]. As these record only kernel calls, they miss many of the functions of interest such as those involved in buffer overflow attacks. These tools give limited access to other data (e.g. environment variables) during run-time. Additionally, these tools fail to meet our goal of increasing **application level** audit data because they only record kernel level functions. However, they can be used to verify some of the problems that the interposed library might indicate (especially race conditions), and are very valuable as general debugging tools.

3 Related Work

3.1 System Library Interposition

Timothy W. Curry of Sun Microsystems used library interposition to generate a run time analysis of the X11 library functions [7]. 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[3]. Curry [7] 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. Curry [7] was invaluable for our understanding and implementing library interposition.

Other published work has focused on the related topic of system (kernel) call interposition, as discussed in the following subsections.

3.2 Janus

The *Janus* software was developed to address concerns regarding programs and helper applications that process unauthenticated network data, as well as the untrusted nature of the calling applications themselves. Goldberg, Wagner, Thomas, and Brewer [10] 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 Goldberg et al. [10], *Janus* is constructed around a framework that calls dynamic modules that implement various aspects of the security configuration by filtering the relevant system calls. Their design uses *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) may be insufficient for restricting applications because it can be avoided by static linking. (Jain and Sekar [11] describe some recent advances based on the *Janus* prototype.)

3.3 Interposition Toolkit

Jones [12] 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 Jones [12] was that of system or kernel calls. However, these concepts are extendible 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 Jones [12] included code reuse, toolkit development, low performance cost relative to the cost of the system call interception mechanism and the operations being emulated.

[3] At the time that these libraries were built, the SLI toolset was not available for our use.

```
1  /*************************************
2   * Logging the use of certain functions *
3   *************************************/
4  char *strcpy(char *dst, const char *src) {
5     char *(*fptr)(char *,const char *); /* pointer to the real function */
6     char *retval;                       /* the return value of the call */
7
8     AUDIT_CALL_START;
9
10    AUDIT_LOOKUP_COMMAND(char *(*)(char *,const char *),"strcpy",fptr,NULL);
11
12    AUDIT_USAGE_WARNING("strcpy");
13
14    retval=((*fptr)(dst,src));
15
16    return(retval);
17 }
```

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.

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.

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

```
#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.

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?

What follows is a brief description of the unsafe programming practices from Garfinkel and Spafford [6] and AUS [5] we elected to audit. Our prototype was designed to report on usage and not attempt to enforce any particular policy. Additional design considerations are described where appropriate.

- The effect of vulnerabilities often differ depending on the privilege level at which the process is running. Therefore, we set the reporting level to be of increasingly higher priority as functions were being called as
  - As a regular user (`EUID`$\neq 0$, `EUID` $=$ `UID`)
  - As a non-root, 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 include any 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* by Aslam, Krsul, and Spafford [13]. 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 (Stevens [14]). 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 (from Bishop and Dilger [9]):

  - 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
- The command line arguments specified by the user
- Process ID (`PID`)
- Time of logging

6 Results

Our interposing library was tested on a RedHat 5.2 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 the SUID/SGID programs that were installed. We then switched to the root account and performed similar tasks.

No instances of SUID/SGID 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 occurrences of `sprintf`, `strcpy` and `strcat` were noticed in the normal file utilities. These could have been avoided because Linux has `snprintf` as a standard I/O routine, as well as `strncpy` and `strncat` which 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 (Bishop and Dilger [9], Bishop [15]). The following subsections describe

our findings in greater detail. The race conditions are illustrated using system call traces because our library simply logs that a potential race condition occurred, but does not indicate where in the application code this took place[4].

6.1 Vim

The first race condition (or *synchronization error* [13]) occurs in Vim version 5.1 and the relevant system call trace 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 system call trace that shows the two race conditions that were discovered in the Vim 5.1 binary that is shipped with Redhat 5.2. 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

[4]It is possible to create an interposing library that performs an identical trace at the application layer, but this was outside the scope of the current project.

```
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 system call trace that shows the race condition in GNU cp 3.16 The race condition occurs during the sequence `open("./passwd")` (line 6) → `chmod("./passwd")` (line 14).

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 system call trace is shown in Figure 5.

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**.

Row		User time (sec)	Kernel time (sec)
A	No interposition	10.59	0.00
B	Interposition using syslog	10.91	0.02
C	Interposition dumping to a file	10.86	0.02
D	Interposition ignoring messages	10.84	0.02

Table 1: The average (mean) runtime results of running an example program which invoked `snprintf` 1,000,000 times. 10 runs were performed in multi-user mode on a RedHat Linux 5.2 machine. Timing results were collected using the shell's `time` command.

7 Runtime Performance

To measure runtime performance of library interposition, we selected two functions defined in the system libraries, only one of which was defined by our interposing library, `snprintf` and `sprintf`. Direct comparisons between the execution times of these two cannot be made because of their differing definitions, however, we can examine the relative costs imposed by interposition.

We created a small program that executed either `snprintf` or `sprintf` in a tight loop with 1,000,000 cycles[5]. In single user mode, RedHat Linux 5.2 performs disk access synchronously and charges the I/O wait time to the calling process. To avoid this problem, we created a new runlevel that had a minimal set of services enabled (kerneld, syslog, random, linuxconf, pcmcia, and keytable). No network devices were enabled, and no console access was permitted during our testing. We executed our program 10 times in this multi-user runlevel and recorded the results of the shell `time` command. We tested using three different techniques for logging to gain a measure of the overhead imposed by the logging as well as the overhead of the interposition itself. We had three interposable libraries, one recorded the audit data using `syslog`, one wrote all audit information to a file, and the final one simply discarded the audit information as it was generated.

Table 1 lists the measured results for the function `snprintf` which is not defined within our interposed library. Rows B, C, and D all reflect similar execution profiles except for the setup of the output channels which occurs in B and C. For the purposes of timing evaluations relative to a non-interposed executable, we will use row B as our representative as it has the highest measured time.

The combined runtime for the `snprintf` program is 10.59 seconds uninterposed, and 10.93 seconds interposed. If we amortize the 340 millisecond difference across the 1,000,000 invocations of `snprintf` we obtain a cost of 340 nanoseconds per function invocation, or an additional runtime of 3.2% for functions that are not defined within the interposing library[6].

Table 2 on the next page shows the measured execution times for a similar program that used `sprintf` (which is defined within our interposable libraries) tested under similar conditions. Row E is the execution of the application without interposition. Row F is the execution using our interposition library and logging audit data with the `syslog` function. Row G contains run times when the interposing library is recording audit data to a specified file. The interposing library in row H simply discards the audit data, and is included to allow us to separate the costs of interposition from the costs of logging.

The interposition costs can be measured by comparing row H with row E. The interposed executable executes 210 milliseconds *faster* than the non-interposed version. This is an 1.8% decrease in runtime. Generating audit data, but not logging it is interesting, but we would prefer to have it recorded. The run

[5]The timing resolution on RedHat 5.2 is too coarse to allow us to measure the timing of individual function calls [16].

[6]These percentages are relative to the execution time of the function, and would change based on the function invoked. They are supplied to give the reader an impression of the relative length of the increased execution time.

Row		User time (sec)	Kernel time (sec)
E	No interposition	11.41	0.00
F	Interposition using syslog	13.52	4.42
G	Interposition dumping to a file	13.48	4.27
H	Interposition ignoring messages	11.20	0.02

Table 2: The average (mean) runtime results of running an example program which invoked `sprintf` 1,000,000 times. 10 runs were performed in multi-user mode on a RedHat Linux 5.2 machine. Timing results were collected using the shell's `time` command.

times between rows F and G are similar, and since logging with `syslog` is preferred and it has the greatest runtime, we will make measurement comparisons using row F.

The combined runtime for the `sprintf` program is 11.41 seconds uninterposed, and 17.94 seconds interposed. There is an additional 6.53 seconds of execution time incurred when we interpose the executable and log the audit data with `syslog`. Per invocation of a function that is defined within our interposing library, the amortized increase is 6.53 microseconds. This is an approximately 57.2% longer than not interposing at all. Contrastingly, if no audit information is being recorded, our measured run times decreased. `sprintf` is one of the functions where we are concerned as to its usage and log every invocation of it. Other functions such as `fork` are only logged when the return value indicates failure, and consequently these functions would only incur the higher cost in exceptional circumstances.

In summary, interposition with our library increases function invocation time by 340 nanoseconds if the function is not defined within our library and 6.53 microseconds if the function is defined, and we are logging the data using `syslog`.

8 Recommended usage

The costs of interposition vary depending on the implementation chosen and the policy that is being applied. Our library attempts to perform its actions transparent to the underlying application. As noted in section 7, functions not defined in our library are slowed by 340 nanoseconds, and functions defined are slowed by 6.53 microseconds. If the function is not defined within our library, the delays due to the interposition itself would be imperceptible to a normal user operating in an interactive environment.

The bulk of the slowdown is caused from the mechanism by which the audit data is logged. Much of this occurs asynchronously and outside the user's perception, so the performance of applications that heavily utilize functions which are defined within the interposing library and are generating data that must be logged will be slightly less responsive to the user but should not be significantly slower. On a system that is heavily used and frequently close to maximum capacity, this technique may impose too much overhead for use on all program invocations, instead we recommend that this library be used as part of an invocation wrapper for certain applications of specific concern (such as a ftp daemon).

Note that the largest cost is incurred when data is logged. When no logging is needed, the impact is much lower. The more detailed the policy for logging is (i.e. more conditionals must be satisfied before logging takes place) the more suited this technique is to mandatory usage by all applications.

9 Example: Detecting Buffer Overflows

As described in Section 5, certain standard library functions are frequently involved in a type of vulnerability called a *buffer overflow*. We have created an interposing library that detects and prevents a subset of these attacks against setuid binaries. This detecting/preventing library is a concrete example of a policy that can be enforced using the interposition technique.

9.1 Background

Buffer overflow attacks have been discussed and analyzed in the past, yet modern software and operating systems continue to have programs that are vulnerable to this type of attack. A buffer overflow attack occurs when a program copies data into a fixed length buffer without checking the amount of data that is copied, thus overflowing a buffer. If this buffer resides on an executable stack, then the data overflow can modify instructions that will be executed by the computer. An attacker can use this technique to cause arbitrary code to be executed. If the program that is being attacked is running with raised privileges (as superuser or some group), then it is possible for an attacker to control the computer at that heightened level of privilege. There are other criteria that are necessary for such a failure to be exploitable, Krsul, Spafford, and Tripunitara [17] and Smith [18] discuss the necessary conditions.

"Mudge" [19] and "Aleph One" [20] have published tutorials for exploiting a buffer overflow vulnerability. Other approaches to preventing these type of attacks include static analysis, specialized boundary checking compilers (Cowan, Pu, Maier, Hinton, Bakke, Beattie, Grier, Wagle, and Zhang [21] and Cowan, Beattie, Day, Pu, Wagle, and Walthinsen [22]), and modifying the stack to be non-executable ("Solar Designer" [23], Dik [24], and Snarskii [25]).

9.2 Design of the library

For this prototype, we interposed three library functions that are common culprits in buffer overflow attacks (`strcpy`, `strcat`, and `getenv`). Our library examines the data that is to be copied into a buffer and looks for the following characteristics as a heuristic of a possible attack in progress:

- Sequences of NOP assembly instructions (Sparc and x86)
- Common subsequences of buffer overflow code (Sparc and x86)
- Subsequences of non-printable characters

The `strcpy` and `strcat` checks are only performed if the destination address is located on the stack (one of the necessary conditions). All of the checks are configurable at compile time, so only Sparc or x86 codes could be selected, and the characteristics of suspicious non-printable characters can be set (frequency and length). There is also the option of simply having flagged library calls fail or to have the interposing library terminate the application. In either case, when a suspicious string is discovered, an entry is made into a log.

9.3 Implementation

For our testing, we focused on setuid binaries present in Solaris 7 and RedHat Linux 5.2. Local buffer overflow attacks against these platforms were gathered from various online collections of exploit scripts [26–28] and the results of our testing are presented in Section 9.4.

In order to ensure the `LD_PRELOAD` environment variable was set to our interposing library, we wrapped the SUID or SGID programs as suggested by AUSCERT [29] (see Figure 6 on the following page). Line 3 shows how the interposing library is placed. There is no path specified because Solaris and Linux use a default library search path whenever an executable is running with enhanced privileges. Line 5 has the

```
int main(int argc, char *argv[]) {

    putenv("LD_PRELOAD=libbufferdetect.so"); /* set up the environment */

    execv(REAL_PROG,argv);
    perror("execv failed");
    exit(1);
}
```

Figure 6: A wrapper program for SUID or SGID binaries that sets the environment variable `LD_PRELOAD` to point to an interposing library that detects some buffer overflow attacks.

Exploit name	Effect before wrapping	Effect after wrapping
ex_lpset	Segmentation Fault	Detected and Terminated
ex_sdtcm_convert	Segmentation Fault	Detected and Terminated
ex_LC_MESSAGES_rsh	Root shell	Detected and Terminated
ex_LC_MESSAGES_passwd	Illegal instruction	Detected and Terminated
ex_dtprintinfo	Root shell	Root shell
smashcap_linux	Root shell	Root shell
crontab_linux	Root shell	Detected and Terminated
w00w00_vixie	Root shell	Detected and Terminated

Table 3: Results from wrapping SUID and SGID binaries with a prototype interposition library that heuristically detects possible buffer overflow attacks. “Segmentation faults”and “Illegal instructions” often indicate that the exploit script was using an incorrect target address.

constant `REAL_PROG` defined at compile time to point to the original executable which has been stripped of any SUID or SGID flags and moved to a different filename. The wrapper is placed in the original location of the SUID or SGID program with the permissions for the file set to be the same as the original unwrapped executable.

9.4 Buffer Overflow Detection Results

We compiled our test library to detect both x86 and Sparc attacks, and indicated that it should terminate any offending application upon such detection. For each attack attempt, we first attempted the attack against an unpatched system, and then wrapped the affected binary and tried the attack again. The results are summarized in Table 3. Our prototype detector is only testing three of the many possible avenues for buffer overflow attacks, and as can be seen by our failure to detect ex_dtprintinfo (which involves a PATH attack) and smashcap_linux (the overflow is in the `tgetent` function) attacks, more work is needed. However, we were able to implement our detector on multiple platforms without requiring changes in the previously compiled, vendor supplied binaries. From the performance results presented in Section 7 on page 10 the overhead induced is relatively low.

10 Future Work

There is much more work that can be performed in this area. Our first interposing library was a prototype that was designed for us to demonstrate the usefulness of the technique. Based on our success there are several branches our future work can proceed along.

In non-security related areas, there are a few areas that could be explored further.

1. The most significant cost was created by the logging mechanism itself. It would be interesting to further explore the costs associated with logging audit data work to develop techniques to minimize the cost of submitting audit data into a logging system.

2. The decrease in measured run time that occurred when interposing but not logging `sprintf` has not been adequately explained. A careful, detailed analysis of this phenomenon would be interesting to see if it is an artifact of this implementation or indicative of a technique that may have larger application.

3. Develop a technique by which a policy can be specified external to the interposing library itself. The current prototype requires the policy to be hard coded into the executable. Some sort of specification language and parser would be needed to permit a configuration file to control the behavior of the library.

Specifically addressing this particular implementation in the security domain, there are some enhancements that can be made to our interposing library.

1. Explore what other data or states should be audited and recorded to further assist in intrusion and misuse detection. Our current prototype is focused on a subset of insecure programming practices, and we can continue to expand the domain covered.

2. Add additional function definitions to detect IFS and PATH environment variable attacks. These attacks involve a malicious user setting an environment variable that affects how the shell processes a command line.

3. Create specialized interposition libraries to use in conjunction with known executables that provide system and network services (sendmail, fingerd, nfsd, etc.). By using a known program, it is possible to create a much more specialized policy to implement, and to prevent unexpected behavior from occurring.

4. Integrate our audit data with existing intrusion and misuse detection systems such as the AAFID architecture [30]. As a stand alone data source, this technique is quite successful. It is a natural step to implement data readers to handle the new information in existing system frameworks.

Our second library prototype also suggests several areas for further research and implementation.

1. Extend the interposing library to cover as many “dangerous” library functions as possible. The current subset being examined gives us reasonable coverage, but further improvements should be possible.

2. Refine the library and wrapper combination for easy deployment on diverse architectural platforms. One goal of such improvements is to make a tool that is useful to the general computing public, and make it available for use.

3. Examine the feasibility of incorporating this functionality into the standard library system.

11 Conclusions

We have constructed an interposition library that satisfied our goal of increasing the amount of application level audit data without needing to modify either applications or system libraries. We have demonstrated that this technique imposes a relatively small overhead, and that recording the audit data imposes a much larger cost. Our library was able to detect many unsafe programming practices in a recent release of an operating system, as well as discover two previously unreported race conditions. A second prototype interposable library was presented which demonstrated how a policy might be enforced using this technique to help prevent local buffer overflow attacks from succeeding. Library interposition holds much promise for increasing the amount of useful information available to intrusion and misuse detection systems, and gives us another valuable tool for audit data generation.

References

[1] Sandeep Kumar. *A Pattern Matching Approach to Misuse Intrusion Detection*. PhD thesis, Purdue University, Department of Computer Sciences, 1995.

[2] Teresa F. Lunt. Detecting intruders in computer systems. In *Proceedings of the Sixth Annual Symposium and Technical Displays on Physical and Electronic Security*, 1990.

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

[4] Biswanath Mukherjee, Todd L. Heberlein, and Karl N. Levitt. Network intrusion detection. *IEEE Network*, 8(3):26–41, May/June 1994.

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

[6] Simson Garfinkel and Gene Spafford. *Practical UNIX Security*. O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, USA, 1991. ISBN 0-937175-72-2.

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

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

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

[10] 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. URL `http://www.cs.berkeley.edu/~daw/janus/`.

[11] Kapil Jain and R. Sekar. User-level infrastructure for system call interposition: A platform for intrusion detection and confinement. (to appear), 1999.

[12] 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.

[13] 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.

[14] W. Richard Stevens. *Advanced Programming in the UNIX Environment*. Addison-Wesley, Reading, MA, USA, 1992. ISBN 0-201-56317-7.

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

[16] Linux manual. *Linux Programmer's Manual: getitimer, setitimer*, August 1993.

[17] Ivan Krsul, Eugene Spafford, and Mahesh Tripunitara. Computer vulnerability analysis. Technical Report COAST TR98-07, COAST Laboratory, Purdue University, West Lafayette, IN, May 1998. URL ftp://coast.cs.purdue.edu/pub/COAST/papers/ivan-krsul/krsul9807.ps.

[18] Nathan P. Smith. Stack smashing vulnerabilities in the unix operating system. Technical report, Computer Science Department, Southern Connecticut State University, 1997. URL http://reality.sgi.com/nate/machines/security/nate-buffer.ps.

[19] "mudge". How to write buffer overflows. L0pht Advisory, October 1995. URL http://www.l0pht.com/advisories/bufero.html.

[20] "Aleph One". Smashing the stack for fun and profit. *Phrack Magazine*, 7(49):File 14 of 16, Fall 1997. URL http://www.phrack.com/.

[21] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Conference*, San Antonio, TX, January 1998. USENIX. URL http://www.cse.ogi.edu/DISC/projects/immunix/stackguard_usenix98.ps.gz.

[22] Crispin Cowan, Steve Beattie, Ryan Finnin Day, Calton Pu, Perry Wagle, and Erik Walthinsen. Protecting systems from stack smashing attacks with stackguard. In *Proceedings of the 5th Linux Expo*, Raleigh, NC, May 1999. URL http://www.cse.ogi.edu/DISC/projects/immunix/lexpo.ps.gz.

[23] "Solar Designer". Non-executable user stack, 1997. URL http://www.false.com/security/linux-stack/.

[24] Casper Dik. Non-executable stack for solaris. Posting to comp.security.unix, 1997.

[25] Alexander Snarskii. Freebsd stack integrity patch, 1997. URL ftp://ftp.lucky.net/pub/unix/local/libc-letter/.

[26] Rootshell, 1999. URL http://www.rootshell.com/. A searchable online collection of exploits and related news.

[27] Securiteam, 1999. URL http://www.securiteam.com/. An self-described "security portal" with an online collection of exploits and vulnerability reports.

[28] Securityfocus, 1999. URL http://www.securityfocus.com/. An online vulnerability database and official Bugtraq archive.

[29] AUSCERT. Auscert overflow_wrapper.c, 1997. URL ftp://ftp.auscert.org.au/pub/auscert/tools/overflow_wrapper.c.

[30] 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.