

# **Attacking a High Performance Computer Cluster**

**Miguel Torres, Rayford B. Vaughn, German Florez, Zhen Liu, Susan M. Bridges**  
**Mississippi State University**  
**Department of Computer Science**  
**Center For Computer Security Research**  
**Box 9636**  
**Mississippi State, MS 39762**  
**(662) 325-2756 fax: (662) 325-8997**  
**<mt98,vaughn,gf24,zliu,bridges>@cs.msstate.edu**

Abstract – This paper describes several scenarios that could potentially be used to implement attacks against a Linux high performance cluster, resulting in actions that corrupt information in the system (i.e. integrity issues), collect important information (i.e. confidentiality violations), and perform denial of service attacks (i.e. availability issues). These attacks were designed and tested for the MPI/PRO and C environments. We have implemented anomaly detection algorithms which are able to verify the correct execution of parallel programs. We also report representative results of our detection efforts that provide evidence of success. Technical level 4.

# Attacking a High Performance Computer Cluster

## Introduction<sup>1</sup>

Linux clusters have become widely used computational resources in sensitive environments. The workstations within these clusters exchange information not only through TCP/IP networks, but also through special high-speed switches that facilitate rapid communication. Different types of attacks and software failures in a single node may lead to an unstable state of the entire cluster system. The power of the computational resources afforded by these clusters combined with the sensitivity of the applications that they run make them attractive targets for intrusions. To enhance the security in a cluster environment, the system administrator can focus on prevention measures, detection of anomalies or recovery actions. As a prevention measure, the system designer could define well-controlled access rules to prevent the intrusion or abuse of the system from outside or inside users. For example, a job scheduler designer could develop rules to prevent an unauthorized user from submitting a job and consuming computational resources [5]. Operating system access control can also be used to enhance the security level of the entire cluster. However, the stricter the access control rules, the less usable and flexible the system. Thus, detection and recovery are necessary complements to prevention. We describe several scenarios that could potentially be used to mount attacks against a Linux high performance cluster, explain our implementation of these attacks in a cluster environment, present data sets that we have collected that capture both normal program behavior and program behavior with simulated attacks, and report results of anomaly detection experiments that detect the attacks. Research in intrusion detection systems requires collection of audit data that reflects the behavior of the system under both “normal” conditions and “attack” conditions. Such data sets allow development and testing of intrusion detection techniques. The provision of both “normal” and “anomalous” behavior allows the system to test both for the ability to detect specific anomalous behaviors as well as the ability to avoid alerts when no anomalous behavior is present. Such data sets exist for traditional host and network environments, but they do not exist for a high performance computing (HPC) cluster architecture.

We describe three attack scenarios that we refer to as a daemon process attack, an interposition library attack, and a device driver attack. These attacks are launched against a cluster architecture using the Linux operating system and result in actions that corrupt information in the system (i.e. integrity issues), collect important information (i.e. confidentiality violations) and perform a denial of service attack (i.e. availability issues). These attacks were designed and tested for the MPI/PRO and C environment and have been used to generate datasets containing sequences of function calls and sequences of system calls.

The remainder of this paper describes, in some detail, the attacks we created and an overview of the techniques we use to detect these specific attacks as well as other attacks that would be expected to result in anomalous behavior patterns. We also report representative results of our detection efforts that provide evidence of success.

## Creation of a Daemon Process

An UNIX-like daemon process is an application that does not need interaction with a terminal, like a web or ftp server. A daemon process can be launched by any other process (called parent), and it can be used to perform an attack since it might be able to steal resources from the system even when the parent process

---

<sup>1</sup> This work was partially sponsored by the National Science Foundation Grant# CCR-9988524, the Army Research Laboratory Grant# DAAD17-01-C-0011 and the Office of Naval Research Grant# N00014-01-1-0678.

has terminated its execution. In the MPI environment this kind of attack can be very effective since when a daemon process is created the MPI scheduler loses track of it, allowing it to perform almost any kind of CPU usage. To accomplish this type of attack, a Trojan horse is inserted into a trusted application and at some point of execution or under specific conditions the trusted application will launch the attack by creating another process that runs with the same permissions of the parent process.

## Construction of a Daemon

A daemon is defined as a non-interactive process running in the background that cannot maintain an association with a terminal, cannot write error messages on standard error, but can write files and perform any other operation allowed to a normal process [13].

The high level steps needed to create and run a daemon process are as follows [13]:

- 1) Split the execution of a normal process using a *fork()* function to create a new process. This process then runs from the same point at which the fork was called and exits its parent process. It is also necessary to guarantee that the child process is not a group leader. Execution of the child process then continues after the parent process terminates.
- 2) Clear the file creation mask using system call *umask(0)*. This is done to ensure that any file created later will not have the permissions specified during the creation of the parent process.
- 3) Disassociate the child process from the user's terminal (when invoked from the user's terminal). This step ensures that signals destined for the terminal will not affect the running process. A new session and process group are then created containing only the child process using the system call *setsid()*.
- 4) Prevent the next open terminal from becoming a controlling terminal of the new child process. This is done by setting signal SIGHUP as ignored using the system call *signal()*.
- 5) Split the execution of the child process again using a *fork()* function to create the actual daemon process.
- 6) Change the directory from which the process was started to a well know directory (delete any reference it had to the starting directory, so that it can eventually be unmounted). Change the directory to the root directory "/", using the system call *chdir()*. This is important in order to avoid the use of the path that is used by the parent process, which could potentially be unmounted at the end of execution of the parent process.

Sample code of these steps is provided at Appendix A.

Once the daemon process is created, it can accomplish a variety of damaging functions or it can exploit machine resources. We describe two such functions that we have implemented in our work: daemon memory allocation attack and daemon file attack. Each is described in following sections.

## Daemon Memory Allocation Attack

This attack was designed to take over memory resources from the system. When the daemon process is launched, it repeatedly allocates memory through a pointer. This process consumes the memory that the system has allocated for the user and leads to a denial of service attack. In most of the cases this kind of attack finishes in a denial of service error but it can also be used in time-shared based systems, where a user can use this kind of attack to execute processes that are not allowed for him in his own time-share.

## Daemon File Attack

This attack was designed to take over storage, memory, and computational resources of the system. When the daemon process is launched, it repeatedly allocates memory through a pointer, copies data to a temporary file, and then closes the file. This can rapidly overtake the resources of the system, slowing down the machine, and resulting in a denial of service attack and a confidentiality issue. Similarly, an integrity attack can be launched.

## The Interposition Library Attack

Library interposition is a useful technique for tuning performance, collecting runtime statistics, function/parameter trace information, or debugging applications when used in a traditional sense. It can also become an attack technique. Interposition libraries can be used to generate attacks by intercepting function calls that an application makes to the stack of shared libraries. Once the function call is intercepted, the called function can be modified to add malevolent functionality as well as exercising the actual function that the application originally intended to call. This procedure leaves both the original application and the library intact [5]. Additional information on this procedure can be found in Curry [3] and Kuperman [9].

In a Linux system, the link editor uses the *LD\_PRELOAD* environment variable to search for the user's dynamic libraries. Using this feature, the operating system gives the user the option of interposing a new library. Interposition is "the process of placing a new or different library function between the application and its reference to a library function" [3]. Thus, the library interposition technique allows interception of the function calls without the modification or recompilation of the dynamically linked target program. By default the C compilers in LINUX use dynamic linking [5].

The most common library used in a Unix/Linux environment is Libc (the C library and operating system interface). The MPI library is an additional library that helps in developing applications for parallel programming and message passing applications. Both of these libraries are important in implementing the attacks described here. Figure 1 shows a schematic of this type of attack.

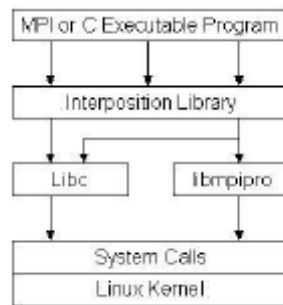


Figure 1 Interposition library

To implement this attack, it is necessary to create a special shared library that contains a set of variables and functions that will be shared among many applications. The function headers are defined identically to those of the functions that are going to be monitored and captured. The next step consists of achieving the interposing itself. This is done by placing a new (or different library) function between the application and its call to a library function. One way to accomplish this is by setting the *LD\_PRELOAD* environment variable to the path and name of the libraries that we are interposing. The dynamic linker will use the specified library before any other when it searches for a function in the system's shared libraries [3, 4, 5].

A problem that must be avoided in the use of interposition results when trying to find the function that the application originally called. When the new function makes the call to the real function to perform the actual call, this operation should not attempt to call the interposed function again. In Unix/Linux based systems this can be avoided by using the dynamically loaded (DL) libraries package of the operating system, which includes specific functions that help in the implementation of the interposition technique [3]. We describe some of these below.

The *dlsym* function is the function that we use for implementation of the interposition technique. This function allows a process to obtain the address of a symbol defined within an object made accessible through a *dlopen* call. For the purpose of this study, we assume that the library we require is already open

(libc). The *dlsym* function will search for the named symbol in all objects that are loaded automatically as a result of loading the object referenced by its handle. Load ordering is used in *dlsym* operations upon the global symbol object [3, 6, 7]. This is important in order to assure that the functions that are being called correspond to the hacked (i.e. libc or MPI functions) or interposed functions.

To implement an MPI library attack, it is necessary to open the MPI/PRO library using the *dlopen* function. This makes the executable object file specified available to the calling program. The class of files eligible for this operation and the manner of their construction are specified by the implementation, although typically such files are executable objects such as shared libraries or programs. A successful call to *dlopen* returns a handle that the caller may use on subsequent calls to *dlsym* and *dlclose*. This is then used to inform the system that the object referenced by a handle returned from a previous *dlopen* invocation is no longer needed by the application [6]. It is important to note that the user who is running this attack has gained root privileges on the computer under attack.

Code demonstrating the basic idea of the interposition technique is included in appendix A. To create the interposition library, the code must be compiled to create a shared object using the following directives:

```
g++ sourceFile.c -o library.so -shared -ldl -D_GNU_SOURCE
```

## Attacking the libc Library

In order to attack the libc library, two libraries were created. The first one generates an attack to the *fopen* and *fclose* functions of the standard C library. These were implemented as follows:

- ✓ The *fopen* attack: This interposer function copies the file that the user intends to open to the directory */temp/tmp* with a randomly generated name *filexxxxxx*, where *x* represents an alphanumeric digit. Subsequently, the original open function is executed for the file that was requested by the user.
- ✓ The *fclose* attack: This interposer function copies the file that the user is closing into the directory */temp/tmp* with a randomly generated name *filexxxxxx*, where *x* represents an alphanumeric digit. It then executes the close function for the file that the user requested.

Other attacks designed for this experiment are intended to corrupt the information and the memory allocated to the user's processes. This leads to a denial of service attack. The attacked functions were: *fread* (reads data from a file), *fwrite* (writes data to a file), *malloc* (allocates memory to a pointer), and *memcpy* (copies a block of memory). This interposer attack changes the size of the data that is to be copied, moved or read by adding or subtracting an integer number from the size parameter of the called function. It then executes the actual function that the user requires. To do this we use a random number generator from the Random Number Generator library published by George Marsaglia and Arif Zaman at Florida State University [11].

Additional attacks can be performed by attacking not only Operating System functions, but also by user specific applications, which perform function calls to user generated libraries allowing the attacker to get access to sensitive information processed by the user library.

## Attacking the MPI Library

Attacks against the MPI library essentially use the same technique used against the libc library. A single library was created. Most of the attacks result in corruption of data and denial of service attacks; however, they can be modified to perform a Trojan horse or a back door attack, as well as many others.

The functions used in our demonstration were:

- ✓ *MPI\_Init* and *MPI\_Finalize*: these functions are responsible for starting and terminating MPI library usage. In this attack a daemon process is randomly generated in some of the processors

where the parallel application is being executed and it is left running in the background of computer memory after the parent program finishes its execution.

- ✓ *MPI\_Comm\_rank*: This MPI function returns the rank of a process in the communicator (a collection of processes that can send messages to each other) that contains it. The attack makes a call to the *MPI\_Comm\_size* function that returns the size of the communicator. Then the process generates a random number between 0 and the size of the communicator thus confusing the rank identities of the processes and sending the information to the wrong destination.
- ✓ *MPI\_Recv* and *MPI\_Send*: These MPI functions receive and send a memory block that corresponds to the structure of an MPI data type between processes. This interposer attack changes the actual information contained in the memory block that is transferred, resulting in the corruption of the information used by the MPI program and later in a denial of service attack.
- ✓ *MPI\_Reduce*: This MPI function is a collective communication operation, in which all the processes in a communicator contribute data that is combined using a binary operation. This attack changes the kind of operator used in the binary operation to another valid binary operator resulting in an integrity violation.

## Device Drivers and System Calls

Linux, like most operating systems, interact with hardware devices via modularized software components called device drivers. A device driver hides the specifics of a hardware device's communication protocols from the operating system and allows the system to interact with the device through a standard interface. In Linux, device drivers are a part of the kernel and may either be linked statically into the kernel or loaded on demand as kernel modules. Device drivers run as part of the kernel and are not directly accessible to user processes [2]. Writing a device driver involves writing a part of the Linux kernel. This means that a device driver runs with kernel permissions and privileges. Actions could include writing to memory, reformatting a hard drive, or possibly damaging a monitor or video card.

A system call is implemented in the Linux kernel. When a program executes a system call, arguments are packaged and handed to the kernel, which assumes execution of the program until the call completes. This means that a device driver can use and manipulate the system calls because the device driver is also part of the Linux kernel [2]. A listing of system calls for a specific version of the Linux kernel can be found at */usr/include/asm/unistd.h*. Some of these are for internal use by the system and others are used only in implementing specialized library functions. The system call number is an index in an array of a kernel structure called *sys\_call\_table[]*. This structure maps the system call numbers to the needed service function, and it can be accessed from a device driver. Figure 2 shows the schematic of this attack. The basic structure of a device driver is:

---

```
#define MODULE
#include <Linux/module.h>
int init_module(void)
{
    printk("<1>Hello World\n");
    return 0;
}
void cleanup_module(void)
{
    printk("<1>Bye, Bye");
}

```

---

When the *init\_module* function is called the first time, the module is loaded in kernel space and prepared for later invocation of the module's functions. The *cleanup* function is invoked immediately after the module is unloaded and it must undo whatever *init\_module* did so that the module can be unloaded safely.

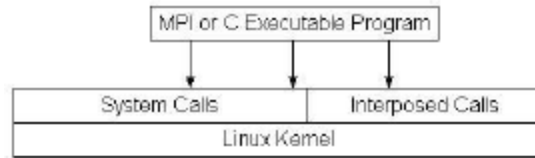


Figure 2 Device driver attack

The fact that the system call table exists as a global variable in the kernel space allows the developer (i.e. of a device driver) to modify it and replace some kernel functions with others that are implemented in the code of the module. The basic code for this attack can be found in appendix A. An example of a modified kernel function that could contain malicious functionality is also shown.

The general high-level approach for intercepting a system call is outlined below:

- ✓ Find a specific system call entry of interest in `sys_call_table[]` (look at `include/sys/syscall.h`).
- ✓ Save the old entry of `sys_call_table[X]` as a function pointer (where  $X$  stands for the system call number that is going to be intercepted).
- ✓ Save the address of the new (modified) system call defined by setting `sys_call_table[X]` to the needed function address.

It may also be useful to save the old system call function pointer because it may be needed in the modified function when emulating the original call.

Programming a Linux module is a complicated task and the programmer must be careful in the use of memory because a missing pointer can lead to a system crash. Most of the usual C functions are not used in kernel mode. The kernel mode has its own functions such as `printk` (same as `printf`) and `kmalloc` (same as `malloc`). The purpose of this attack was to hack the `sys_open` function that implements the open file function in kernel space and to make a copy of each opened file.

A major draw back of this kind of attack in developing kernel-resident software requires super user privileges. Thus it would be difficult for a normal user to develop and deploy its own extensions. It also depends on the configuration that the system administrator has on the operating system, and how the original system call table is generated for the attacked system.

## Detection of cluster attacks

Our longer goal is the design and building of a prototype intelligent intrusion detection system (IIDS) that demonstrates the effectiveness of artificial intelligence approaches to intrusion detection within a high performance computing (HPC) architecture, with the belief that such an environment represents a different security paradigm. The techniques used to implement this architecture include neural networks, data mining with fuzzy logic, and stochastic models.

An intrusion detection system in a high-speed environment faces the problem of large data volume due to the speed of the communication and the local computations. In addition, the generation of useful information may require resources that the cluster could need to complete the execution of other tasks. Thus, the correct representation of the data that should be analyzed by an intrusion detection system is a complex problem.

Although networking is the most critical component in a high-performance computer, monitoring the communication among workstations does not provide a complete description of the system's behavior. Software agents in each node are needed [4].

In order to detect the anomalies produced by the implementation of the daemon and interposer attacks, the behavior of parallel programs was analyzed using two mechanisms for data capture in each one of the hosts of the cluster: trapping of system calls and trapping of function calls. As explained before, system calls are the interface between user space and kernel space, and are used to execute hardware or OS/specific operations. Function calls are high-level implementations in user mode, which can execute one, several, or no system calls. Both methods allow the abstraction of very different patterns from the trace of an MPI/C program.

Artificial neural networks (using datasets built upon system call traces), Hidden Markov Models and Sequence Matching (using datasets containing function calls) were implemented to detect the high-performance attacks in a Linux cluster system. The experiments were conducted on an 8-node cluster at Mississippi State University.

## Detection with system calls

We collected the system calls issued by an MPI program using a loadable kernel module (LKM). This technique allows expansion of the functionality of the Linux kernel in real-time. An LKM is designed to not be bypassed, does not require kernel modifications, and is not platform specific.

Artificial neural networks (ANN) are weighted directed graphs in which the vertices are small units of computation (artificial neurons) and the edges represent the weighted connections between neuron outputs and inputs. Knowledge is acquired by the network through a learning process (in our experiments we use the Backpropagation algorithm). A neural network derives its computing power through its parallel-distributed structure and its ability to learn and generalize. Generalization refers to the neural network that produces reasonable outputs for inputs not encountered during training (learning). This property is useful for building intrusion detection systems because it means that a system can detect unknown attacks with little increase in the false positive rate.

The output of the neural network is a flag that indicates whether or not a sequence of system calls is classified as anomalous. We have used different online and offline detection methods to evaluate the overall status of an MPI program. In offline detection, we already have all the system calls that a program has used during one execution process. The data we use to detect the intrusion is a trace of system calls. But in online detection we only know the calls that the program has issued before the current system call; i.e. we do not know what calls will be executed next. We have developed four methods to classify a trace as anomalous or normal (the first one listed is used in offline mode and the next three in online mode):

- ✓ Offline: We extract the sequences from the traces without any duplicates and classify these sequences. A threshold is defined for firing an alarm.
- ✓ Total A: We count the anomalous sequences within the trace. If the number is greater than a threshold, an alarm is fired.
- ✓ Cont A: The number of consecutive anomalous sequences is counted.
- ✓ Burst Counter: A variable is defined that remembers the total number of anomalous sequences seen so far. After a normal sequence is found, we decrease this variable by a small factor. The advantage of using this algorithm is that it allows occasional anomalous behavior, but is quite sensitive to clusters of anomalies.

To determine the number of anomalies that a program must have in order to be considered anomalous, we must define a threshold often known as a cut-off point. For example, if a program generates 50 anomalies and the cut-off point is 70, the program is not considered anomalous by the IDS. By iterating the cut-off point we can plot a ROC curve that is widely used for test analysis. The x-axis in a ROC graph contains the false positive rate and the y-axis contains the true positive rate. The false positive rate indicates the number of MPI programs (without attacks) that were classified as anomalous. In contrast, the true positive rate indicates the number of MPI programs containing the daemon and interposition attacks described above as Trojan horses that were correctly classified as anomalous.

Figure 3 shows the ROC analysis of an LU-factorization program. LU-factorization is an improvement of the Gaussian elimination method for solving systems of linear equations. This is a typical example of a real-world cluster application. The training data set contained 25 traces of runs of the LU-factorization program where parameters of the program were varied for different runs. A total of 25 of the traces were for “normal” traces and 14 of the traces were for “attack” runs. 12 traces were used for training and 33 traces were used for testing. Note that both the *Cont A* and *Burst Counter* methods detect no false positives for any threshold value.

## Detection with Function Calls

The second approach that we have developed for detecting anomalies in a high-performance environment can be seen as a distributed host-based anomaly detector and it is implemented using two modules: The *profiler* and the *analyzer*. The profiler collects data in real time from a MPI application in each host, and creates a database that is used to train the detection algorithm (training is done in offline fashion). The analyzer executes the detection algorithm in real time and produces an alarm when the behavior of the program deviates from the behavior extracted from the database. Both the Profiler and the Analyzer collect function calls issued by a parallel application written in C/MPI using library interposition. We were interested in the name of the function as well as its relative order in the program trace.

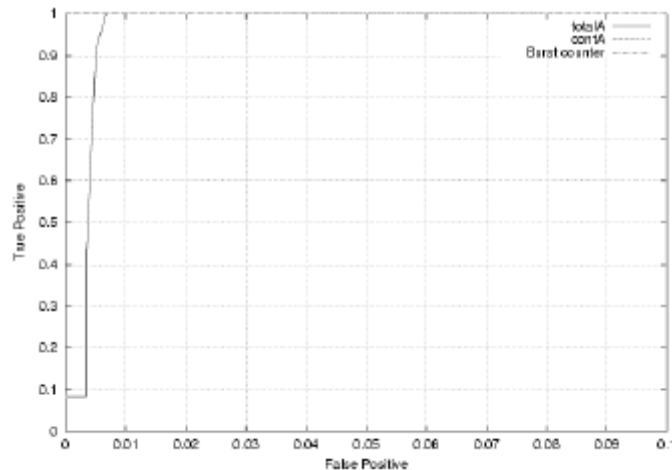


Figure 3 ROC curve for anomaly detection using system calls for LU Factorization

We have implemented two detection algorithms that have been previously reported to be effective methods for anomaly detection, the Hidden Markov Model and the exact sequence matching. Hidden Markov Models is a doubly stochastic process widely used for modeling sequences of events. An HMM contains probabilities of transition among states and for each state there exist a probability of producing a symbol (i.e. a function call). An HMM for a MPI program is trained with the Baum-Welch algorithm using the function calls database produced by the Profiler when the application has been executed under normal conditions. The detection of anomalies consist of comparing the HMM with the sequence of function calls issued by the parallel application in real time. If the model cannot produce the stream of function calls, an alarm is fired. In contrast, a deterministic algorithm is also used. This method is known as exact sequence matching. An MPI program trace is divided in small subsequences of a fixed length. The sequences produced by a program containing normal behavior using the Profiler tool are stored in a database implemented efficiently as a sorted tree. The analyzer detects an anomaly if a function call subsequence issued by the MPI application cannot be found in the tree.

Figure 4 demonstrates the use of the sequence matching to differentiate normal behavior among processors. The x-axis corresponds to time and the y-axis is the number of anomalies produced by the IDS. As we can

see at the beginning and at the end of an MPI application, the system is generating some false positives. This graphic shows that our system also has the potential to be used as a performance-monitoring tool.

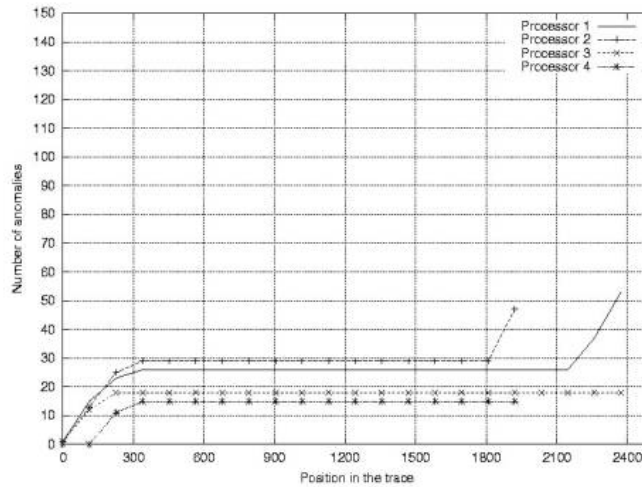


Figure 4 Analysis of false positives with the sequence matching algorithm

We can improve the accuracy of the IDS by training both the HMM and the sequence matching with function calls from several executions of the same MPI program. Figure 5 shows the impact of the cut-off point in the true positive rate of the HMM, and sequence matching algorithms for the LU-factorization program. The false positive rate in this example was 0. The training set consisted of 25 executions of the LU-factorization program, and the test set contains 250 normal executions and 138 executions with the Trojan horse attacks.

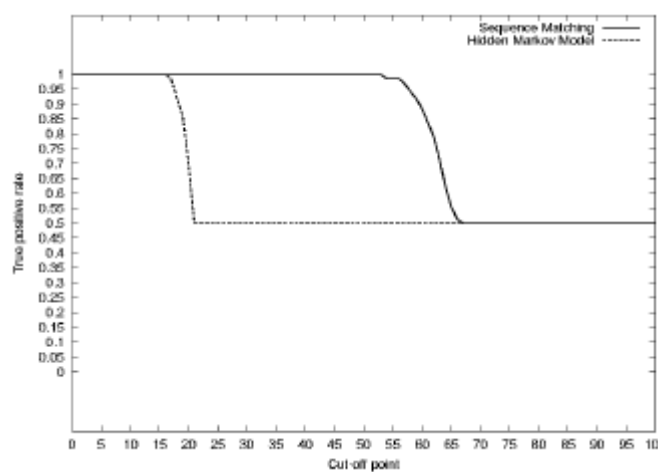


Figure 5 True positive analysis of LU factorization

## An attack database

We are creating an attack database that will be used for testing our anomaly detection algorithms that will be available for the research community. The creation of synthetic data for a cluster of workstations is not an easy task because it involves knowledge of MPI/C, operating systems, networking, and process monitoring. The first step to create such a dataset consists on the definition of normal patterns in the communication and the computer resources in the cluster. This can be achieved by collecting huge amounts

of data from the set of jobs that are executed on a normal basis on the cluster. In our dataset, those jobs are created by executing few parallel algorithms with several input parameters or initial conditions. The parallel algorithms selected are LU factorization (as an example of a typical scientific application), and a communication benchmark based on LLCbench from the University of Tennessee [REFE] (providing a wide variety of communication patterns). More complex algorithms and simulations will be included too. The second step consists of the creation of anomalies. Such anomalies are generated by executing some of the attacks described in this paper when authorized jobs are being executed.

Currently, the dataset contains more than 2000 traces for system and function calls, where a trace is defined as the dataset generated by one job in one node, and each job consists of one or more processes.

Some of the questions that we are planning to address with this dataset are: can we detect all the anomalies using host-based detectors, do we need network-based detectors, which stream (function or system calls) must be used, which is more efficient, which algorithm performs the best detection rate no matter which stream is being monitored, which attacks can be detected, and can we profile a work session in a cluster instead of profiling a single process?

## Summary and Conclusions

Clusters of workstations have become increasingly important computational resources and represent attractive targets for attack. They are widely used for intensive scientific computation in sensitive environments. In this paper, we describe several attack scenarios that can potentially be mounted against a Linux cluster. We have described our implementations of some of these attacks and their impact on the confidentiality, availability and integrity on the cluster environment. We have collected data sets for programs under attack and under normal conditions using both interposition libraries and LKM techniques for capturing function call sequences and system call sequences respectively. We have also implemented anomaly detection modules that demonstrate that the data streams collected can be successfully used to detect the type of attacks described with low false positive and low negative rate.

## References

- [1] S. M. Bridges, G. Florez, Z. Liu, "A Comparison Of Input Representations In Neural Networks: A Case Study In Intrusion Detection," *Proceedings: International Joint Conference on Neural Networks (IJCNN)*, Honolulu, Hawaii, 2002.
- [2] J. Corbet, A. Rubini, "Linux Device Drivers," O'Reilly, <http://www.oreilly.com/catalog/linuxdrive2/chapter/bookindexpdf.html> (current oct. 2002).
- [3] T. W. Curry, "Profiling and Tracing Dynamic Library Usage Via Interposition," *Proceedings: USENIX 1994 Summer Conference*, 1994.
- [4] G. Florez, A Trusted Environment for MPI, Dept. Computer Science, Mississippi State University, Masters Thesis, December 2002.
- [5] G. Florez, Z. Liu, S. M. Bridges, "Detecting Intrusions and Software Failures in a Cluster Environment," *Paper Under Review at 28th Annual USENIX Technical Conference - USENIX'2003*, San Antonio, Texas, 2003.
- [6] <http://www.ddj.com/documents/s=865/ddj0165m>, "Building Library Interposers for Fun and Profit," (current August 2002).
- [7] <http://www.linuxdocs.org/HOWTOs/Program-Library-HOWTO/dl-libraries.html>, "Dynamically Loaded (DL) Libraries"(current July 2002).

- [8] [http://packetstorm.decepticons.org/docs/hack/LKM\\_HACKING.html](http://packetstorm.decepticons.org/docs/hack/LKM_HACKING.html), “(nearly) Complete Linux Loadable Kernel Modules” (current oct. 2002).
- [9] B. Kuperman, E. Spafford, “Generation of Application Level Audit Data Via Library Interposition,” Technical report TR 99-11, COAST Laboratory, Purdue University, 1998.
- [10] Linux Journal, <http://www.linuxjournal.com/article.php?sid=4378>, “Kernel Korner: Loadable Kernel Module Programming and System Call Interception” (current oct. 2002).
- [11] G. Marsaglia, A. Zaman, “Toward a Universal Random Number Generator,” Technical Report FSU-SCRI-87-50, Florida State University, 1987.
- [12] D. Rusling <http://www.tldp.org/LDP/tlk/tlk.html>, “The Linux Kernel” (current oct. 2002).
- [13] Unix C Programming Frequently asked Questions, [http://www.geocities.com/capitalware/unix\\_faq\\_toc.html](http://www.geocities.com/capitalware/unix_faq_toc.html) (current July 2002).

## Appendix A

### Daemon Process Attack

---

```
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <fcntl.h>
#include <errno.h>
#include <stdio.h>
#define SIZE__ 1000

void main(){
    int pid;
    void *pointerMemory;
    /* The first Child fork */
    pid = fork();
    if (pid < 0) {
        exit(1); /* error encountered,
no child has been created! */
    }
    if (pid != 0) {
        exit(0); /* this is the parent,
and hence should be terminated */
    }
    /* make the process a group leader,
session leader, and lose control */
    setsid();
    /* close STDOUT, STDIN, STDERR */
    close(STDIN_FILENO);
    close(STDOUT_FILENO);
    close(STDERR_FILENO);
    /* close STDOUT, STDIN, STDERR */
    /* ignore SIGHUP that will be sent to a
child of the process */
    signal(SIGHUP, SIG_IGN);
    umask(0); /* lose file creation mode
mask inherited by parent */
    chdir("/"); /* change to working dir */
    pid = fork();
    if (pid < 0) {
        exit(1); /* fork() failed, no
child process was created! */
    }
    if (pid != 0) {
        exit(0); /* this is the parent,
hence should exit */
    }
    /* this is the child process of the child
process of the actual calling process */
    /* and can safely be called a grandchild
of the original process */
    signal(SIGPIPE, SIG_IGN);
    /* ignore SIGPIPE, for reading, writing
to non-opened pipes
```

---

---

```
every program using pipes should
ignore this signal for
being on the safe side */
/* this is the main daemon process, also
the grand child process */
while(1){
    sleep(DURATION_SEC);
    PointerMemory=(char*)malloc(SIZE__
);
    }
}
```

---

### Implementation of an interposition library

---

```
#include <dlfcn.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>
#include <unistd.h>
#include <time.h>
#include <string.h>
#include "signal.h"
#include <sys/utsname.h>
#define TRUE 1
#define FALSE 0
// Indicator to enable or disable the profiling
static int DoProfile=TRUE;
FILE *fopen(const char *filename, const char
*mode){
    // Data referent to the real function
    // The Header of the function that we are
looking for
    typedef FILE>(*function_type) (const
char *filename, const char *mode);
    // Pointer to the function
    static function_type function=NULL;
    static char* function_name="fopen";
    // The value returned by the real
function
    FILE *retval;
    // Search for the next function that fits
the especification of function_name
    if (!function){
        function = (function_type)
dlsym(RTLD_NEXT,function_name);
    }
    // If the profiling is active
    if (DoProfile){
        // By doing this we assure that
the function next call would not be profiled
        DoProfile=FALSE;
```

---

---

```

        // Here we can insert our code
        to do something before the execution of the real
        function
        // Executes the real function
        retval =
        (*function)(filename,mode);
        // Here we can insert our code
        to do something after the execution of the real
        function
        DoProfile=TRUE;
    }
    else //do not profile, only execute
        retval =
        ((*function)(filename,mode));

    return (retval);
}

```

---



---

```

        sys_call_table[SYS_open]=orig_open;
        /*set sys_oepn syscall to the orignal one*/
    }

```

---

## Device driver attack

---

```

#define __KERNEL__
#define MODULE
#include <linux/module.h>
#include <linux/kernel.h>
#include <asm/unistd.h>
#include <sys/syscall.h>
#include <linux/string.h>
#include <linux/fs.h>

/*sys_call_table is exported, so we
can access it*/
extern void* sys_call_table[];

/*the original syscall*/
int (*orig_open)(const char *filename,int flags,
int mode);

int hacked_open(const char *filename,int flags,
int mode){
    /* Hacked function must go here*/

    /* Returns the original system call*/
    return orig_open(filename,flags,mode);
}

/*module setup*/
int init_module(void){
    orig_open=sys_call_table[SYS_open];
    sys_call_table[SYS_ope
n;

    printk("<I>Hello World\n");
    return 0;
}

/*module shutdown*/
void cleanup_module(void){
    printk("<I>Bye, Bye");
}

```

---