USENIX Association

Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference

Boston, Massachusetts, USA
June 25–30, 2001

LOMAC: MAC You Can Live With

Timothy Fraser NAI Labs
tfraser@nai.com 3060 Washington Road
Glenwood, MD 21738

Abstract

LOMAC is a security enhancement for Linux kernels. LOMAC demonstrates that it is possible to apply Mandatory Access Control techniques to standard Linux kernels already deployed in the field, and to do so in a manner that is simple, compatible, and largely invisible to the traditional Linux user. The LOMAC Loadable Kernel Module protects the integrity of critical system processes and files from viruses, worms, Trojan horses, and malicious remote users. It is compatible with standard Linux 2.2 kernels and applications, and seeks to provide useful protection without site-specific configuration. LOMAC is designed to be a form of MAC that typical users can live with.

1 Introduction

Over the last 25 years, many projects have demonstrated useful Mandatory Access Control (MAC) features on UNIX systems. Two early examples include KSOS [18] and UCLA Secure UNIX [23]. More recent examples include DTE [3], and Security-Enhanced Linux [17]. However, despite their success, these demonstrations have not prompted widespread adoption of MAC in mainstream UNIX kernels.

One likely explanation for this lack of widespread adoption may be overall cost of use: In these demonstrations, the new MAC features came at the cost of incompatibility with existing kernel and application software, increased administrative overhead, or a disruption of traditional usage patterns. Among typical users, the overall cost of adopting the new MAC features outweighed the perceived benefits, discouraging widespread mainstream adoption.

The LOMAC project is an attempt to bring simple but useful MAC integrity protection to Linux in a form that:

- is applicable to standard kernels,
- is compatible with existing applications,
- requires no site-specific configuration, and
- is largely invisible to traditional users.

In short, LOMAC aims to provide a form of MAC that typical users can live with [19]. LOMAC implements a form of Low Water-Mark MAC integrity protection [5] in a Loadable Kernel Module (LKM). Administrators can load the LOMAC LKM into standard, off-the-CD-ROM Linux 2.2 kernels, including both kernels distributed in binary form and kernels built from standard sources. Once loaded, the LOMAC LKM protects the integrity of critical system processes and files from viruses, worms, Trojan horses, and malicious remote users. Because of its compatible design, LOMAC can be used to provide integrity protection for presently-deployed systems based on standard Linux kernels with little impact on their normal operation.

Several theoretical aspects of the LOMAC project have been discussed in a previous paper [9]. These aspects include LOMAC's application of Low Water-Mark model, the UNIX compatibility benefits of models like Low Water-Mark over many better-known models, and some of the drawbacks of LOMAC's LKM-based implementation with regard to the reference monitor approach [1]. This paper, on the other hand, will focus on the details of LOMAC's implementation, paying particular attention to the techniques required to enhance standard Linux kernels without patching their source, and to manage security attributes without kernel and filesystem support.

The discussion begins with section 2, which describes the integrity protection provided by LOMAC. This is followed by a detailed examination of LOMAC's architecture and implementation in section 3, focusing on LOMAC's use of interposition and implicit attribute mapping to maintain compatibility with standard Linux kernels. Section 4 explains how LOMAC applies its

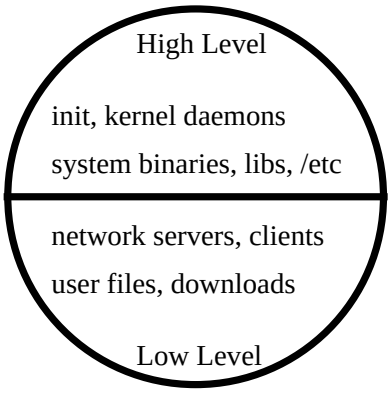

Figure 1: LOMAC's 2-level partitioning of a system.

protection mechanism in a manner that encourages application compatibility and avoids administrative overhead. Section 5 presents the results of some performance benchmarks, and discusses potential optimizations. Section 6 addresses usability concerns and lists some future directions for LOMAC, including strategies to overcome some of its present shortcomings and an upcoming port to FreeBSD. Section 7 follows with a summary of related efforts to enhance the security of Linux kernels. Finally, section 8 presents some conclusions.

2 Protection

LOMAC provides protection by dividing a system into two integrity levels: high and low. The diagram in figure 1 illustrates this division. The high level contains critical system components that must be protected, such as the init process, kernel daemons, system binaries, libraries and configuration files. The low level contains the remaining components, such as client and server processes that read from the network, local user processes and their files. Once LOMAC assigns a file to one level or the other, its level never changes. This is not so for processes: LOMAC can "demote" high-level processes by reducing their levels to low during run-time. LOMAC never increases the level of a process. Section 4 describes how LOMAC decides which files and processes belong in which part; this section summarizes how LOMAC uses this division to provide protection.

When LOMAC is running, a process's level determines how much power it has to modify other parts of the system. Given the above division of the system into two levels, LOMAC provides integrity protection with two main mechanisms. First, LOMAC prevents low-level processes from modifying (writing, truncating, deleting) high-level files or signalling high-level processes. Since non-administrative users, their network clients, and all network servers run at the low level, these restrictions protect the high-level part of the system from direct attacks by malicious remote users and compromised servers.

Second, LOMAC ensures that (potentially dangerous) data does not flow from low-level files to high-level files. A process could attempt to cause such a flow by reading from a low-level file (as data or as program text) and subsequently writing to a high-level file. LOMAC prevents such flows through demotion: whenever a high-level process reads from a low-level file, LOMAC reduces the process's level to low. Once at the low integrity level, LOMAC's first mechanism prevents the process from modifying high-level files, as described above. This combination of mechanisms prevents indirect attacks by viruses, worms and Trojan horses.

LOMAC cannot distinguish between a program that has read low-integrity data but is still running properly and one that has read low-integrity data and has been compromised. However, LOMAC *can* ensure that processes which read potentially dangerous low-level data during run-time are demoted to the low integrity level. Once at this low level, LOMAC's other mechanisms prevent them from harming high-integrity processes or files.

3 Implementation

There are two main problems in implementing kernel-resident MAC: gaining supervisory control over kernel operations, and mapping security attributes to files. There are a range of potential solutions to these problems, each embodying a different tradeoff between features such as generality and efficiency, and costs such as incompatibility with existing software and the need for configuration. LOMAC has chosen low cost solutions in both cases. LOMAC uses interposition at the kernel's system call interface [10, 11, 20] to gain supervisory control. LOMAC uses implicit attribute mapping [3] to map security attributes to files. These choices may not be as supportive of generality and efficiency as alternate approaches involving direct modifications of the kernel source. However, they allow LOMAC to operate on standard Linux kernels already deployed in the field - an essential part of LOMAC's approach to encouraging adoption.

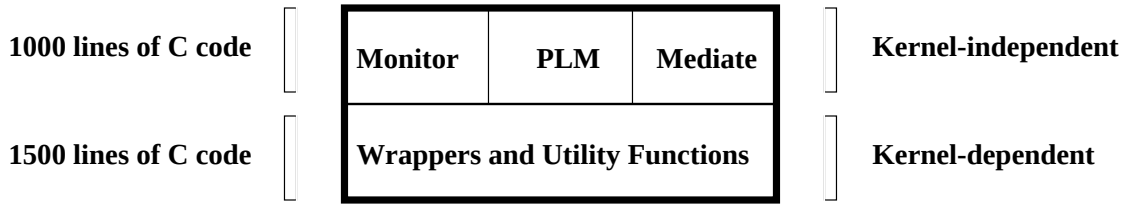

Figure 2: LOMAC Loadable Kernel Module Architecture

Figure 2 shows the architecture of the LOMAC LKM. The diagram shows a horizontal split between upper and lower halves. The upper half implements high-level LOMAC functionality in a kernel-independent manner, and consists of approximately 1000 lines of C code (counting only those lines containing semicolons or braces). The lower half implements a kernel-specific interface to the Linux 2.2 series of kernels, and consists of approximately 1500 lines of C code. An alternate Linux 2.0 interface was supported in the past; alternate Linux 2.4 and FreeBSD interfaces are expected in the future.

3.1 Gaining control

In order to provide protection, LOMAC must gain supervisory control over kernel operations - that is, LOMAC must be able to make access control decisions as described in section 2, and compel the kernel to enforce them. LOMAC achieves this control by interposing itself between processes and the kernel at the kernel's system call interface. LOMAC's kernel interface contains a series of functions called "wrappers," due to their similarity to Generic Software Wrappers [10]. Ultimately, there will be one such wrapper for each security-relevant Linux system call; some wrappers have not yet been implemented in the present version of LOMAC. Each wrapper takes the same parameters as its corresponding system call. At initialization time, LOMAC traverses the kernel's system call vector, which is essentially an array of function pointers through which the kernel provides services to user processes. LOMAC replaces the addresses of security-relevant system calls with the addresses of the corresponding wrappers. Once done, calls made through the system call vector will call the wrappers, rather than the kernel's corresponding system call functions.

Wrappers follow the algorithm shown in figure 3. First, LOMAC performs mediation: it decides whether to allow or deny the calling process's request for service. It bases this decision on a comparison of the calling process's level and the levels of the arguments, as described in section 2. If LOMAC decides to deny, it returns an appropriate error code to the caller. Otherwise, LOMAC proceeds to the next step, where it calls the kernel's original system call function to provide the actual service. Finally, LOMAC monitors the completion of the kernel's original system call, updating its data structures to reflect changes in the system state. This is where LOMAC demotes processes, and marks the in-memory data structures representing open files (`dentry` structures) with the appropriate levels for future reference.

Viewed from a high level of abstraction, this interposition-based wrapper algorithm is not overly complex. However, implementing it in a manner that avoids Time-Of-Check, Time-Of-Use (TOCTOU) errors requires care [11, 26]. Early versions of LOMAC had many TOCTOU errors: Wrappers would copy user-space pathname arguments into kernel-space, and make mediation decisions based on these copies. After positive decisions, the kernel's original system call functions would copy the pathnames into kernel-space a second time, and operate on this second copy. The potential existed for a user process to make a system call with an allowable pathname and change it to a non-allowable pathname after LOMAC had made its mediation decision, but before it called the kernel's original system call function. This ability to change pathnames between the

```
wrapper( arguments ){
  Mediate: decide to allow
  or deny the operation;

  call kernel's original
  system call function;

  Monitor: update LOMAC's state
  on successful completion;
}
```

Figure 3: Wrapper Algorithm

```
01: int wrap_open( const char *filename, int flags, int mode ) {
02:   char *k_filename_s, *k_canabspath_s;
03:   struct dentry *p_dentry, *p_dir_dentry;
04:   struct file *p_file;
05:   int ret_val;
06:
07:   if( IS_ERR( ( k_filename_s = getname( filename ) ) ) ) {
08:     return( PTR_ERR( k_filename_s ) );
09:   }
10:   if( !( k_canabspath_s = (char *)__get_free_page( GFP_KERNEL ) ) ) {
11:     ret_val = -ENOMEM;
12:     goto out_putname;
13:   }
14:   if( ( ret_val = make_canabspath( k_filename_s, k_canabspath_s,
15:     &p_dir_dentry, &p_dentry ) ) ) {
16:     goto out_dputs;
17:   }
18:
19:   if( ( flags & O_TRUNC ) ||
20:       ( ( flags & O_CREAT ) && ( !p_dentry ) ) ) {
21:     if( ! ( p_dentry && WRITE_EXEMPT( p_dentry ) ) ) {
22:       if( !( mediate_subject_object("open",current,p_dir_dentry) ) ) {
23:         ret_val = -EACCES;
24:         goto out_dputs;
25:       }
26:       if( !( mediate_subject_path("open",current,k_canabspath_s) ) ) {
27:         ret_val = -EACCES;
28:         goto out_dputs;
29:       }
30:     } /* if this is not an exempt case */
31:   } /* if we should mediate */
32:
33:   TURN_ARG_CHECKS_OFF;
34:   ret_val = ((int (*)(const char *, int, int))orig_open)
35:     ( k_canabspath_s, flags, mode );
36:   TURN_ARG_CHECKS_ON;
37:   if( ret_val >= 0 ) {
38:     p_file = fget( ret_val );
39:     monitor_open( current, p_file->f_dentry );
40:     fput( p_file );
41:   }
42:
43:  out_dputs:
44:   if( p_dir_dentry ) { dput( p_dir_dentry );  }
45:   if( p_dentry ) { dput( p_dentry );  }
46:   free_page( (unsigned long)k_canabspath_s );
47:  out_putname:
48:   putname( k_filename_s );
49:   return( ret_val );
50: } /* wrap_open() */
```

Figure 4: C source for LOMAC v1.1.0's wrapper for sys_open (run-time assertions and most comments removed).

time of LOMAC's check, and the time the kernel used the pathname gave user processes the opportunity to defeat LOMAC's protection.

Figure 4 illustrates the solution to the TOCTOU problem: copy pathname arguments into kernel-space at the beginning of the wrapper, and invoke the kernel's original system call with the address of this copy, rather than the address of the original user-space buffer. The figure contains the C source for LOMAC's open system call (`sys_open`) wrapper. The source shows the additional buffer-copying, as well as the unusual toggling of the Linux kernel's sense of the kernel-/user-space boundary required to make the its original system calls accept these copies.

In its first 18 lines, the wrapper examines its arguments, gathering the information it needs in later steps. Line 7 copies the filename argument into kernel-space to avoid TOCTOU errors. All subsequent operations are on this copy, rather than the user-space original. LOMAC determines the levels of files based on their absolute canonical-form pathnames using an algorithm discussed in the next subsection. Line 14 prepares the filename for its level determination by converting it into this form.

The nested `if` statements in lines 19 through 21 ensure that LOMAC performs mediation only when there is the potential for a file creation or truncation. LOMAC does not mediate writes to files in the open wrapper. This mediation is handled by other wrappers corresponding to the Linux kernel's various write system calls. The `WRITE_EXEMPT` macro on line 21 exists to allow harmless truncates of device special files such as serial lines and terminals. Similar exemptions exist in the write system call wrappers. These exemptions allow low-level processes to perform I/O on these devices, while keeping the device special files themselves in the high-level part of the system.

Lines 22 through 32 perform the actual mediation. Before allowing the open, LOMAC makes checks both on the file and on its parent directory, as traditional UNIX does. Line 22 ensures that the calling process has sufficiently high integrity level to modify the contents of the named file's parent directory. Line 26 ensures that the calling process has a sufficiently high integrity level to create or truncate the named file. These checks are handled by functions in the kernel-independent part of the LOMAC LKM.

Lines 33 through 36 invoke the kernel's original system call function using the wrapper's kernel-space copy of the filename argument. When serving user processes, the kernel's system calls expect to copy their pathname arguments from user-space. Before copying, the system calls execute a check to ensure that the pathname buffer address is indeed on the user side of the kernel-/user-space boundary - a check that will normally fail on the wrapper's kernel-space pathname buffers. Fortunately, the kernel provides a mechanism to disable this check on a per-process basis. The macros on lines 33 and 36 toggle this check off for the duration of the original system call function. For safety, the canonical-absolute pathname conversion function on line 14 performs the safety checks that LOMAC turns off in the original system call.

Lines 37 through 50 conclude the wrapper. If the open system call succeeded in opening a file, lines 37 though 41 call LOMAC's kernel-independent open monitoring function to label the file's in-memory data structure (`dentry`) with the appropriate level. The various read and write wrappers will subsequently use this label when they mediate and monitor operations on the file.

As shown in figure 4, it takes a considerable amount of wrapper code to support mediation and monitoring in an interposition-based scheme. The extra buffer copy to avoid TOCTOU errors adds overhead. Similarly, many wrappers contain nested `if` statements like those in lines 19 through 22 to predict, based on the arguments, what operation the kernel will eventually perform. The read and write wrappers require more extensive logic, because these system calls must handle operations on a variety of objects (files, pipes, sockets), each of which requires different mediation and monitoring.

An alternative approach to gaining control might be to patch the kernel source, placing mediation and monitoring further down in the kernel, at the point closer to where it operates on objects. This move would reduce overhead by eliminating the extra TOCTOU buffer copies and the need to predict the kernel's behavior ahead of time. However, this patching strategy is not presently an option for LOMAC, which must avoid modifying kernel source in order to maintain compatibility with existing kernels.

3.2 Attribute Mapping

In addition to gaining supervisory control, LOMAC must also assign integrity levels to files in a manner that is persistent across reboots. LOMAC maintains a persistent mapping between levels and absolute canonical pathnames in its Path Level Map (PLM) module.

level	*flags*	*path*
high		`"/home/httpd"`
low	child-of	`"/home"`
high		`"/"`

Table 1: Three Path-Level Map Rules

Whenever the kernel opens a file, LOMAC labels its in-memory data structure (`dentry`) with the integrity level indicated by the PLM.

LOMAC's PLM implements a simple form of implicit attribute mapping [3]. Given an absolute canonical pathname, it consults a data structure similar to the abridged one shown in table 1. This data structure is an array of records, each a level, flag, path triplet. The records are sorted, longest path first. The basic algorithm is, given a target path, its level can be found by searching linearly though the list of records until a record is found whose path is a prefix of the target path. The level in this record is the proper level for the file named by the target path. For example, the level of "/home/httpd/html" is high, because it matches the record for prefix "/home/httpd". The attribute mapping is "implicit" because the appropriate level of a large number of files is implied by a small set of rules.

The child-of flag adds a slight bit of additional complexity. For example, the list of records uses the child-of flag in the record for /home. This record indicates that all children of /home are low by default. Because of the child-of flag, the record does not apply to /home itself, only its children.

If, during a search through the record list, the the target path matches a record's path exactly, the flag field is checked. If the child-of flag is set, the match is ignored, and the search continues. Consequently, the level of "/home/httpd" is high because it exactly matches the record for prefix "/home/httpd", which has no child-of flag. The level of "/home/tfraser" is low because it matches the record for prefix "/home" with the child-of flag, and the level of "/home" is high because it skips the child-of "/home" record and matches the record for prefix "/".

The actual list of PLM records used by the present version of LOMAC contains 25 records. The PLM can map levels to files on any type of filesystem, including remote network filesystems. It requires no filesystem support for storing attributes on disk. Since the PLM's list of rules is completely static, it is trivially persistent across reboots, and is not susceptible to consistency problems if the filesystem is modified while LOMAC is not running.

The PLM does have two main drawbacks, however. First, it requires canonical absolute pathnames as input. Determining the canonical absolute form of a pathname in a system call wrapper adds overhead.

Second, the PLM can produce inconsistent integrity level results when queried on files named by multiple hard links: If the different hard link names correspond to different levels, the PLM will return whichever level corresponds to the hard link name specified in a query. LOMAC prevents the creation of such confusing hard links during its run-time; administrators must take care to avoid creating them before they load LOMAC. This problem does not extend to symbolic links. LOMAC calls the appropriate kernel functions to translate all paths into canonical (all symbolic links translated) absolute (relative to the root directory) form before examining them. Consequently, LOMAC handles symbolic links properly.

4 Application

In order to apply the protection scheme described in section 2, LOMAC must be able to determine the appropriate level for every process and file in the system. This section describes how LOMAC makes this determination. LOMAC's choice of solution impacts both application compatibility and the degree to which LOMAC remains invisible to users. It is also essential to LOMAC's ability to automatically assign the appropriate levels to users and network servers without site-specific configuration.

4.1 Dividing the Filesystem

Section 3.2 explained how LOMAC uses a small set of rules to determine which parts of the filesystem are at the high integrity level, and which are at the low level. These rules are presently set at compile-time. Although future versions of LOMAC may provide a more configurable rule set, the goal of the present implementation is to deliver a single generic configuration that provides at least some protection on a wide variety of systems.

The division described by the current rule set reflects the tension between two competing goals: providing

the maximum amount of protection, and maintaining the maximum amount of application compatibility. The first goal demands that all files be at the high level, where LOMAC will keep them safe from modification by low-level processes. However, the second goal demands that all files be at the low level, where LOMAC will never prevent low-level processes from modifying them. This second goal is important to compatibility - preventing file modifications can introduce incompatibilities by causing applications to fail.

LOMAC's present division is a compromise between these goals that emphasizes application compatibility. The division roughly parallels the traditional UNIX boundary between the portion of the filesystem owned by the root user (high), and the portion owned by local non-root users (low). This parallelism helps to reduce LOMAC's visibility to non-root users. For example, LOMAC tends to prevent the same operations as the traditional UNIX access control mechanisms: high-level files tend to be owned by the root user. Non-root user processes run at the low level. LOMAC prevents low-level processes from modifying high-level files. However, this behavior is often not surprising because the familiar UNIX access controls would also prevent these modifications as attempted non-root modifications of root-owned files. Only when a low-level process acquires root privileges does the difference become readily apparent - a low-level root process has greatly reduced powers in the presence of LOMAC.

4.2 Monitoring Processes

While file levels are static, process levels can decrease during run-time. In general, LOMAC assigns a new process the same level as the process who created it. At initialization time, LOMAC assigns the high integrity level to the first process (the idle/init process), which initializes the system by creating a new high-level process to handle various system tasks. These processes continue by creating more high-level children. As individual processes read from low-level files, LOMAC demotes them to the low integrity level. From that point on, all their children begin life at the low integrity level.

This demotion behavior allows LOMAC to automatically assign user sessions to the appropriate level. For example, with a console login, the init, getty, and login processes all run at a high level. Upon verifying a user's identity, login spawns a child which executes the user's shell. The shells of non-root users immediately read resource files from the low-level part of the system, causing LOMAC to demote them. From that point on, their children operate at a low level. LOMAC does not demote the root user's shell because the root user's home directory and its contents are at a high level. The root user's shell may therefore create high-level children, although LOMAC will demote them if they go on to read from the low-level part of the system. This automatic assignment of levels allows LOMAC to provide protection without being configured to recognize a site-specific list of users.

LOMAC also uses its demotion behavior to automatically confine programs that use the network to interact with (potentially malicious) remote entities. LOMAC treats all network interfaces as low-level files. As soon as a process reads from a network interface, LOMAC demotes it to the low integrity level. This scheme places network clients and servers at a safe, low level at the moment they first risk compromise - that is, as soon as they receive their first communication from the network. Furthermore, this scheme allows LOMAC to provide protection without being configured to recognize a site-specific list of potentially dangerous network-readers - LOMAC simply waits for a potentially dangerous network read operation and then makes the appropriate demotion.

4.3 Exceptions for Compatibility

LOMAC's protection scheme is specifically designed to prevent possibly malicious remote entities from using the network to command local processes to modify local `/etc` configuration files. Unfortunately, this scenario essentially describes the purpose of `pump`, the client-side DHCP agent: `pump` modifies local configuration files such as `/etc/resolv.conf` on behalf of remote DHCP servers. Similarly, LOMAC's protection scheme is specifically designed to prevent processes from transferring data from low-integrity to high-integrity files. Unfortunately, this is essentially what occurs as log messages travel from low-integrity processes to the high-integrity system log file through the system log daemon, `syslogd`.

In both these cases, LOMAC must make an exception to allow these critical programs to operate properly. To this end, LOMAC maintains a short list of "trusted" programs. LOMAC never demotes processes that are running trusted programs. Being free from demotion, as long as `pump` and `syslogd` begin running at a high level, they will remain at that level and operate properly. Since trust frees a program only from LOMAC's demo-

tion behavior, running a trusted program at the low integrity level does not provide any additional privileges. Still, the presence of trusted programs represents some risk. If a high-level process running a trusted program were compromised, LOMAC would not prevent it from harming the high-integrity part of the system.

LOMAC also uses the trusted program mechanism to make some concessions to usability. Because it demotes network-reading programs, LOMAC effectively prevents remote administration. (A level-1 process cannot modify critical configuration files, even with the root identity.) Since remote administration is critical to some real-world operations, LOMAC trusts the Secure Shell daemon `sshd`. This arrangement grants administrators high-level user sessions via SSH, as follows:

LOMAC demotes untrusted remote login daemons such as `telnetd` and `rlogind` as soon as they read from the network, preventing them from forking off high-level children. However, because of LOMAC's trust, high-level processes running `sshd` can read from the network without being demoted, and fork off high-level processes to run local user shells. With the trusted `sshd` acting as an un-demotable bridge to the network interface, these local user shells escape demotion themselves by interacting with the network only indirectly, through high-level pseudoterminal devices.

LOMAC also provides a trusted file upgrader, `lup`. When run at a high integrity level, `lup` allows administrators to copy low-integrity files (such as downloaded software updates) to the high-integrity area of the system, presumably after manually verifying that they represent no threat to integrity. The `lup` program is effectively a limited version of `cp` with additional logging. Only its trust-enabled escape from demotion allows it to upgrade files. Consequently, running `lup` from a low integrity level will not permit a user to write a high-level file.

4.4 LOMAC and root

Although LOMAC's division of the system attempts to parallel the traditional UNIX root/non-root boundary for the sake of compatibility, LOMAC's protection mechanism does not depend on the Linux kernel's existing root-identity-based protection mechanism. LOMAC provides protection by observing requests for service made by processes at the kernel's system call interface, and denying those requests it identifies as threats to the integrity of the system. It is not aware of the Linux notion of user identity; consequently it does not allow the root user any special privileges. Conversely, LOMAC does not override, disable, or weaken the existing Linux protection mechanisms: When LOMAC is running, an operation will be allowed if and only if both LOMAC and the existing Linux protection mechanisms agree it should be allowed.

Since LOMAC's strategy of controlling the transfer of data is orthogonal to the traditional UNIX root-based mechanism, it is also orthogonal to efforts to increase the granularity of this root-based mechanism, such as Linux-privs [21].

5 Performance

Table 2 shows the results of three benchmarks comparing the performance of Linux kernels running with ("LOMAC v1.1.0") and without LOMAC ("No LOMAC"). The benchmarks tested version 1.1.0 of LOMAC with run-time assertions disabled. The first entry in the table measure the time to perform the "make" portion of the Linux 2.2.5 kernel build procedure on 450MHz Intel Pentium II-based RedHat 6.0 system. Each result is the average of 10 trials, discarding an initial uncounted trial to prime caches. Although this macro-benchmark tends to hide LOMAC's additional kernel overhead, it gives an impression of how a user might perceive LOMAC's performance on a real workload.

The second and third table entries show the latency and throughput performance of the Apache/1.3.9 web server running on a 133MHz Intel Pentium-based RedHat 6.1 system. This web server was connected via a 10Mbit crossover (uplink) Ethernet cable to a Sun Microsystems Ultra 5 workstation running Solaris 2.6. This workstation performed a series of 47 10-minute-long trials running the WebStone 2.5b4 web server benchmark using 32 test clients applying the standard WebStone static workload to the webserver to produce each result. The apparent small improvement in latency is spurious; the performance impact of LOMAC is much smaller than the variance in the WebStone benchmark's results.

The remaining table entries show the results of the BYTE UNIX benchmarks performed with the UnixBench 4.1.0 software on the same system used for the kernel-build benchmark. Each result is the average of 21 trials. The table omits the largely computational DhryStone and WhetStone components of the bench-

Kernel Build Elapsed Time (s)			
	mean	std. dev.	penalty
No LOMAC	269.61	0.03	-
LOMAC v1.1.0	278.05	0.03	3.1%
Webstone Latency (s)			
	mean	std. dev.	penalty
No LOMAC	0.569	0.003	-
LOMAC v1.1.0	0.567	0.003	-0.2%
Webstone Throughput (Mbit/s)			
	mean	std. dev.	penalty
No LOMAC	8.327	0.058	-
LOMAC v1.1.0	8.305	0.063	0.3%
UB Execl Throughput (loops/s)			
	mean	std. dev.	penalty
No LOMAC	642.4	23.7	-
LOMAC v1.1.0	537.0	21.7	16.4%
UB File Copy 256 Byte buffers (KByte/s)			
	mean	std. dev.	penalty
No LOMAC	34393	289	-
LOMAC v1.1.0	31131	222	9.5%
UB File Copy 1024 Byte buffers (KByte/s)			
	mean	std. dev.	penalty
No LOMAC	69672	385	-
LOMAC v1.1.0	66155	573	5.0%
UB File Copy 4096 Byte buffers (KByte/s)			
	mean	std. dev.	penalty
No LOMAC	81379	547	-
LOMAC v1.1.0	79078	775	2.8%
UB Pipe Throughput (loops/s)			
	mean	std. dev.	penalty
No LOMAC	263124	1679	-
LOMAC v1.1.0	234225	4289	11.0%
UB Pipe-based Context Switch (loops/s)			
	mean	std. dev.	penalty
No LOMAC	139917	1827	-
LOMAC v1.1.0	116993	1510	16.4%
UB Process Creation (loops/s)			
	mean	std. dev.	penalty
No LOMAC	3811	20	-
LOMAC v1.1.0	3830	24	-0.5%
UB System Call Overhead (loops/s)			
	mean	std. dev.	penalty
No LOMAC	249414	332	-
LOMAC v1.1.0	249356	303	0.2%
UB 8 Shell Script Load (loops/minute)			
	mean	std. dev.	penalty
No LOMAC	144.2	3.0	-
LOMAC v1.1.0	129.0	3.1	10.5%

Table 2: Benchmark Results

mark; the presence of LOMAC did not significantly affect these components. The apparent small improvement in process creation time is also spurious; the performance impact of LOMAC is smaller than the variance in the Process Creation portion of the UnixBench benchmark.

LOMAC's performance is comparable to interposition-based general kernel extension mechanisms such as Generic Software Wrappers [10] and SLIC [11]. For example, the SLIC prototype reported performance penalties ranging from 0% to 5% on an emacs-building benchmark, depending on how many security extensions were loaded at the time. The Generic Software Wrappers prototype reported penalties ranging from 3.5% to 6.5% on a kernel-building benchmark, up to 1.4% for WebStone latency, and up to 3.3% for WebStone throughput, again depending on how many security extensions were loaded.

LOMAC has not yet been optimized for performance; there are several areas of its implementation that trade performance for simplicity in order to support the rapid development of new features. For example, when a process opens or executes a file, LOMAC consults the PLM to determine the file's level and the level of its parent directory. LOMAC saves these levels in memory for the benefit of its read and write mediation functions. However, LOMAC makes no attempt to skip the PLM lookup on subsequent opens, even for files and directories that already have their levels stored in memory. The PLM implementation is presently based on a simple but inefficient sequential search. Lookups on short, common directories such as "/bin" and "/usr/bin" require 25 string comparisons. This inefficiency is reflected in the high penalty shown by the UnixBench Execl Throughput benchmark. Considerable time could be saved by avoiding redundant PLM lookups, and by improving the PLM's search algorithm.

At a higher level, LOMAC might save time by not mediating the actions of high-level processes, since LOMAC always allows high-level processes to do as they wish. Similarly, LOMAC might save time by not considering low-level processes for demotion, since low-level processes are already running at the lowest integrity level. This optimization has the potential to reduce the overhead of read and write operations shown in the three UnixBench File Copy benchmarks. As LOMAC nears its goals for features, an increasing amount of development resources will be allocated to improving performance.

6 Discussion

Section 4 presents an analytical argument for the usability of LOMAC, describing how LOMAC is designed to be compatible with existing applications, and is largely invisible to non-root users. Although there have been no formal usability studies of LOMAC, there is some anecdotal evidence of its compatibility with traditional UNIX: In order to test LOMAC under normal usage conditions, NAI Labs' Chief Scientist runs LOMAC on his Linux workstation. However, he was forced to turn LOMAC off near the end of January, 2001 while the author fixed a serious bug. On the evening of 31 January 2001, the author completed the fix and re-installed LOMAC on the chief scientist's workstation. Significantly, he carelessly forgot to inform anyone of what he had done. LOMAC was not discovered until 11 days later, when the author mentioned the re-installation in casual conversation. Although the chief scientist's overall usage of the workstation during that period was light, the fact remains that LOMAC was sufficiently compatible with traditional UNIX to remain undetected by an highly experienced user until it was unwittingly revealed by its author.

There is much work yet to be done on LOMAC. With more development, LOMAC can overcome many of its present limitations. The remainder of this section summarizes some possible future directions.

Improved handling of "/tmp": In its present state, the PLM prevents the effective use of temporary files by high-level processes. Directories like "/tmp" must be able to contain files of different integrity levels where the appropriate level can be determined only by considering the level of the file's creator, not by considering its pathname. The PLM presently supports only low-level files in "/tmp", making it impossible to run temporary-file-dependent programs like emacs or gcc at a high level.

There are at least two ways in which the PLM might be extended to overcome this problem. The PLM might be extended to polyinstantiate "/tmp", providing separate temporary directories for each level in a manner that is transparent to processes. Alternately, the PLM might apply a new flag to "/tmp" indicating that the levels of files there should be based on the level of the creating process. Both of these solutions involve tradeoffs: A polyinstantiated "/tmp" may confuse users ("why can't my low-level process see that high-level temporary file?"). On the other hand, allowing files to inherit their creators' levels will add complexity - the present invariant that a file's level may always be determined by its pathname greatly simplifies many aspects of the LOMAC code.

Complete controls: LOMAC does not yet control all critical kernel operations. For example, even though LOMAC controls the kernel's read and write system calls, processes may still bypass LOMAC by modifying files via memory-mapping. Access to memory-mapped files is difficult for LOMAC to mediate because once a process maps a file, it may modify the file through memory operations that do not require system calls. To solve this problem, LOMAC might perform pessimistic read/write mediation at the time a file is memory-mapped, and revoke or downgrade dangerous mappings upon process demotion. Several other kernel abstractions also lack sufficient controls, including message queues, semaphores, and all forms of shared memory.

Port to Linux 2.4, FreeBSD, TrustedBSD: As was described in section 3, LOMAC's architecture includes a separate kernel-dependent interface. Although earlier versions of LOMAC had alternate interfaces for Linux 2.0 and 2.2, only the 2.2 interface is supported in the present version. The 2.2 interface does not support the 2.4 Linux kernel; a new interface will be required. Experience with these interfaces has shown that LOMAC tracks changes in the Linux kernel relatively easily: because it has so few dependencies on the kernel source, porting has been required only between major kernel revisions (2.x, not 2.2.x) so far.

An interposition-based port to FreeBSD is scheduled for the second half of 2001. As the TrustedBSD project begins to provide improved kernel support interfaces for LKMs like LOMAC, the author will port to these interfaces, as well. In addition to making LOMAC available to more users, these ports will provide an opportunity to reimplement LOMAC's kernel interfaces with the benefit of previous experience. These implementations may provide better performance and additional features, such as multiprocessor support.

Improved confinement: LOMAC protects the integrity of high-level processes and files, but does not provide any protection for the low-level part of the system. For example, although LOMAC prevents a compromised low-level server from installing trapdoors and Trojan horses in the high-level part of

project	*patch*	*module*	*general wrappers*	*access control*	*intrusion detection*
Beattie MAC	x			x	
Generic Software Wrappers		x	x		
Immunix/SubDomain	x	x		x	x
Janus (Linux)		x		x	
Kernel Hypervisors		x	x		
LIDS	x			x	x
LOMAC		x		x	
Medusa DS9	x	x		x	
Pitbull LX		x		x	
RSBAC	x			x	
SAIC DTE	x			x	
SELinux	x			x	
VXE	x			x	
William&Mary DTE	x			x	

Table 3: A Comparison of Related Projects

the system, it does not prevent a compromised low-level server from harming the integrity of the low-level part of the system, perhaps by destroying low-integrity user files, or by sending kill signals to other low-level servers. This drawback was due to the manner in which the Low Water-Mark model divides a system "horizontally" into levels, separating only high from low. Lipner has suggested an enhancement that would add additional "vertical" divisions, separating one server from another within a given level [16]. The potential of this technique to improve LOMAC remains to be explored.

Configurable levels: Early versions of LOMAC supported configurations with more than two levels, and allowed administrators to assign different levels to each network interface. One useful three-level configuration worked well on a host with two network interfaces: The configuration placed system objects and processes at the highest level, most local user processes, most user files and an interface to an internal network at the middle level, and the remaining servers and an interface to the an external network at the lowest level. This three-level configuration provided integrity protection to a larger portion of the system than LOMAC's present two-level configuration by bringing some user resources into the upper two protected levels.

However, some user files had to remain at the unprotected lowest level where programs that read from the lowest-level network interface, such as E-mail agents and web browsers, could modify them. Consequently, the three-level configuration was more visible to non-root users than the two-level configuration, because it forced them to operate at multiple levels. For example, it forced users to run separate text editor processes for modifying lowest-level and middle-level files, and to choose the proper editor depending on the situation.

Because this complexity conflicts with LOMAC's emphasis on remaining largely invisible to the user, this functionality has not yet been carried forward into the present prototype. However, future versions of LOMAC might be extended to allow site-specific configurations with many levels, and offer the existing two-level configuration as a default. In a configuration with many levels, support for programs that are trusted only in restricted ranges of levels may also be useful [15].

7 Related work

There are a wide variety of projects aimed at improving the security of Linux kernels using interposition and/or MAC. Examples include Generic Software Wrappers [10], the recent Linux port of Janus [12], Kernel Hypervisors [20], LIDS [28], Malcolm Beattie's MAC [4], Medusa DS9 [29], Pitbull LX [2], RSBAC [22], SAIC DTE [24], Security-Enhanced Linux [17], Immunix/Subdomain [7], VXE [14], and William&Mary DTE [13].

Different projects emphasize different goals. Table 3 compares these projects according to several criteria. The first two criteria deal with implementation: Those

projects that modify the kernel source receive a mark in the *patch* column, those that use an LKM receive a mark in the *module* column. The last three criteria deal with features: Projects that seek to provide general support for kernel security extension through system call interposition receive a mark in the *general wrappers* column. Those that provide MAC functionality are marked in the *access control* column. Finally, those that provide or are bundled with other useful security functionality, such as intrusion detection, are marked in the *intrusion detection* column. The projects that have the most relevance to LOMAC's goal of encouraging adoption by decreasing the overall cost of use are the four that avoid modifying kernel source.

Of these projects, Generic Software Wrappers and Kernel Hypervisors seek to provide general support for kernel extensions. Conceivably, LOMAC could be implemented in the frameworks they provide. The remaining two, Pitbull LX and Janus, attempt only to implement a single form of MAC, as LOMAC does. Pitbull LX and Janus provide protection by confining potentially dangerous applications according to the principle of Least Privilege [25]. They lessen their impact on UNIX compatibility by confining only certain applications, rather than applying their controls to every process on the system.

Each of these four LKM-based approaches has the potential to provide a very small overall cost of use, particularly if they were distributed in a form that lessened administrative overhead and did not overly disrupt typical usage patterns,

In the wake of the 2001 Linux Kernel Summit, several organizations have begun efforts to improve the Linux kernel's support for security enhancements like LOMAC. The TrustedBSD project [27] is also developing similar improvements for the FreeBSD kernel. The aspect of these efforts that is most relevant to LOMAC is their plan to provide a new means of gaining supervisory control over kernel operations. The Linux efforts are concentrating on placing "hooks" at strategic points inside the kernel. These hooks will transfer control to security modules like LOMAC, allowing them to make access control decisions. It is reasonable to expect a future version of LOMAC based on these hooks to perform better than the present one; placing the hooks inside the kernel has the potential to eliminate the need for much of the operation-prediction and buffer-copying overhead imposed by interposition at the system call interface.

Fortunately, LOMAC's architecture has strong separation between the interposition-based interface and the rest of the LOMAC LKM. When such hooks become standard kernel features, this separation will allow LOMAC to discard its present interposition-based interface and make use of them.

8 Conclusions

LOMAC's present implementation shows that it is possible to apply Mandatory Access Control techniques to standard off-the-CD-ROM Linux kernels. LOMAC uses interposition at the system call interface to gain supervisory control over kernel operations, and implicit attribute mapping to mark files with persistent labels.

By confining network-reading applications to the low integrity level, LOMAC prevents compromised servers, worms, and malicious remote users from harming the integrity of the high-level part of the system, even when they have root privilege. By demoting processes that read or execute low-integrity data, LOMAC ensures that network-imported virus and Trojan horse programs will be similarly confined, even if they are initially read or executed by root-privileged high-level processes. Due to this confinement, such malicious programs cannot copy themselves or be copied by others into the high-integrity part of the system.

Furthermore, LOMAC's protection scheme requires no support from applications. LOMAC's access control functionality is automatic: Applications do not need to request that it be applied. It is also transparent: LOMAC interposes itself at the kernel's system call interface, requiring only the standard parameters, and returning only the standard error codes. LOMAC does not require users or applications to explicitly choose roles [8] or domains [6]. Because of the automatic and transparent nature of its protection mechanism, LOMAC can operate with existing applications, even those distributed in binary-only form.

LOMAC is designed to be compatible with existing software, largely invisible to traditional Linux users, and applicable without site-specific configuration. In short, it is designed to be a form of MAC that typical users can live with.

9 Acknowledgements

The author would like to thank Lee Badger for his long-standing support for the LOMAC project, and for suggesting the title of this paper.

LOMAC is Free software available for download under the GNU General Public License from `ftp://ftp.tislabs.com/pub/lomac`.

References

[1] J. P. Anderson. Computer Security Technology Planning Study. Technical Report ESD-TR-73-51, USAF Electronic Systems Division, Hanscom Air Force Base, Bedford, Massachusetts, October 1972.

[2] Argus Systems Inc. Pitbull LX. http://www.argus-systems.com/products/white_paper/lx/.

[3] L. Badger, D. F. Sterne, D. L. Sherman, and K. M. Walker. A Domain and Type Enforcement UNIX Prototype. *USENIX Computing Systems*, 9(1):47–83, Winter 1996.

[4] M. Beattie. MAC. http://users.ox.ac.uk/ mbeattie/linux.

[5] K. J. Biba. Integrity Considerations for Secure Computer Systems. Technical Report ESD-TR-76-372, USAF Electronic Systems Division, Hanscom Air Force Base, Bedford, Massachusetts, April 1977.

[6] W. E. Boebert and R. Y. Kain. A Practical Alternative to Hierarchical Integrity Policies. In *Proceedings of the 8th National Computer Security Conference*, pages 18–27, Gaithersburg, Maryland, September 1985.

[7] C. Cowan, S. Beattie, G. Kroah-Hartman, C. Pu, P. Wagle, and V. Gligor. SubDomain: Parsimonious Server Security. In *Proceedings of the 14th USENIX Systems Administration Conference (LISA 2000)*, New Orleans, LA, December 2000.

[8] D. Ferraiolo and R. Kuhn. Role-Based Access Controls. In *Proceedings of the 15th National Computer Security Conference*, pages 554–563, Baltimore, Maryland, October 1992.

[9] T. Fraser. LOMAC: Low Water-Mark Integrity Protection for COTS Environments. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, pages 230–245, Berkeley, California, May 2000.

[10] T. Fraser, L. Badger, and M. Feldman. Hardening COTS Software with Generic Software Wrappers. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, pages 2–16, Berkeley, California, May 1999.

[11] D. P. Ghormley, D. Petrou, S. H. Rodrigues, and T. E. Anderson. SLIC: An Extensibility System for Commodity Operating Systems. In *Proceedings of the USENIX 1998 Annual Technical Conference*, New Orleans, Louisiana, June 1998.

[12] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer. A Secure Environment for Untrusted Helper Applications. In *Proceedings of the 6th USENIX Security Symposium*, pages 1–13, San Jose, California, July 1996.

[13] S. Hallyn and P. Kearns. Domain and Type Enforcement for Linux. In *Proceedings of the 4th Annual Linux Showcase & Conference (ALS 2000)*, Atlanta, Georgia, October 2000.

[14] U. InteS, Odessa. VXE. http://www.intes.odessa.ua/vxe.

[15] T. M. P. Lee. Using Mandatory Integrity to Enforce "Commercial" Security. In *Proceedings of the 1988 IEEE Symposium on Security and Privacy*, pages 140–146, Oakland, California, April 1988.

[16] S. B. Lipner. Non-Discretionary Controls for Commercial Applications. In *Proceedings of the 1982 IEEE Symposium on Security and Privacy*, pages 2–10, Oakland, California, April 1982.

[17] P. A. Loscocco and S. D. Smalley. Integrating Flexible Support for Security Policies into the Linux Operating System. In *Proceedings of the FREENIX Track: USENIX Annual Technical Conference*, June 2001.

[18] E. J. McCauley and P. J. Drongowski. KSOS – The Design of a Secure Operating System. In *Proceedings of the National Computer Conference, Vol. 48, AFIPS Press*, pages 345–353, Montvale, New Jersey, 1979.

[19] M. D. McIlroy and J. A. Reeds. Multilevel security with fewer fetters. In *Proceedings of the USENIX UNIX Security Workshop*, pages 24–31, August 1988.

[20] T. Mitchem, R. Lu, and R. O'Brien. Using Kernel Hypervisors to Secure Applications. In *Proceedings of the 13th Annual Computer Security Applications Conference*, San Diego, California, December 1997.

[21] A. G. Morgan. linux-privs. http://www.kernel.org/pub/linux/libs/security/linux-privs/old/doc.

[22] A. Ott. Regel-basierte Zugriffskontrolle nach dem Generalized Framework for Access Control-Ansatz am Beispiel Linux. Master's thesis, Universitat Hamburg, Fachbereich Informatik, 1997.

[23] G. J. Popek, M. Kampe, C. S. Kline, A. Stoughton, M. Urban, and E. J. Walton. UCLA Secure UNIX. In *Proceedings of the National Computer Conference, Vol. 48, AFIPS Press*, pages 355–364, Montvale, New Jersey, 1979.

[24] SAIC. SAIC DTE. http://research-cistw.saic.com/cace/dte.html.

[25] J. H. Saltzer and M. D. Schroder. The Protection of Information in Computer Systems. In *Proceedings of the IEEE Vol. 63(9)*, pages 1278–1308, September 1975.

[26] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. The Flask Security Architecture: System Support for Diverse Security Policies. In *Proceedings of the 8th USENIX Security Symposium*, pages 123–139, Washington, DC, August 1999.

[27] R. Watson. TrustedBSD: Adding Trusted Operating System Features to FreeBSD. In *Proceedings of the FREENIX Track: USENIX Annual Technical Conference*, June 2001.

[28] H. Xie and P. Biondi. LIDS. http://www.lids.org.

[29] M. Zelem, M. Pikula, and M. Ockajak. Medusa DS9. http://medusa.fornax.sk.