Stateful access control in Linux using the LSM

Thomas Howard Uphill

December 14, 2007

Abstract

Modern computing environments are subjected to a greater number of security problems than the operating systems were originally designed to withstand[7]. Only a small number of programs should be able to manipulate sensitive information[5]. Current operating system security policies are inadequate to address this problem. Maintaining the state of a system can be used to maintain order[8]. We propose that maintaining state can be used to not only reduce the complexity of a security policy but also achieve results not possible without maintaining state.

To implement our model we use the LSM framework in the Linux kernel. LSM is a mature security framework with several real world implementations[2]. We implement a simple system we call lsmlgi which maintains counters for the operations of read, write, execute and delete. We also create unique identities for processes and attach these ids to files created by a process.

We are able to test a visitor law which prevents a user of an account from deleting or modifying files created by another session of his/her account. We implement an Apache law which prevents an account from executing files that it has created (to be used on a webserver to prevent the webserver from running files it has created). In our budget law example we prevent a process running as a certain user from executing more than a set number of subprocesses.

Maintaining state in Linux is possible. In future work we would implement several of the LGI paradigms such as doAdd() which allows an arbitrary element to be added to the state of an agent (process)[8]. Our law system is very rigid, one of the strengths of a system like LGI is that laws can do almost anything and are not restricted to a simple yes or no answer when performing an operation. Of the LSM implementations studied, SELinux appears to be the most mature, future work should attempt to augment SELinux with dynamic state information.

1 Introduction

When UNIX was first developed in 1969, access control was not a great concern. The systems of the day were simple and there were very few users. Intrusion and attack were not yet a problem. Security in general was not a concern. However, due to increases in connectivity and data sharing, the assumption that existing security mechanisms are adequate is fading[9].

The simple access control provided by UNIX is a discretionary access control mechanism based on three 3-bit access vectors. Permissions to do anything on the system are broken into three groups. For each file there is an “owner”, a “group” and the catchall bucket of “other”. The “owner” is the owner of the file. The “group” is the group owner of the file, a group is a collection of users. Any user in the specified group is granted the permissions specified in the 3-bit group vector. “Other” is any user on the system, id est anyone. Other includes the owner and the group and therefore precludes preventing access for the owner or group while allowing access for “other”. Each of these three octets can have the permission to read, write or execute the file (as well as some special permissions not addressed here). This system was adequate in 1969 and was adopted by Linus Torvalds when he wrote Linux in 1991. Not much has changed in 20 years.

These permissions make it difficult to properly secure a system as there is not enough resolution to finely tune the access control on the system. For example, certain files on the system need to be accessible by more than one user or group but should not be accessible by everyone on the system (other).

To address the issue of fine grain access control, POSIX Access Control Lists (POSIX ACLs[1]) were created. An ACL is a list of permissions assigned to an object. Each permission in an ACL is known as an access control element (ACE). ACLs may contain multiple entries for multiple groups, in this way files may belong to more than one group. ACLs may also contain negative permissions. For example a file may belong to group X and be readable and writable by group X. An ACE can be applied to the file that says user Y cannot read or write the file. Even if Y belongs to group X, she cannot access the file. To support ACLs, Linux stores the ACL in an extended attribute on the filesystem (we mention this here as we make use of this as discussed in section 2.1).

POSIX ACLs increase the resolution of permissions but are still a form of discretionary access control (DAC). A more secure system of access control is mandatory access control (MAC). With a discretionary policy, the user is involved in the definition of the policy and assignment. Carelessness by any one user can result in a violation of the systems security policy[9]. Mandatory policy removes the user from the equation and states that all access is denied by default. The policy must allow operations to take place or else they are denied. However, even with a very restrictive mandatory policy, an application may misbehave within its sandbox[9].

A better system of assigning permissions is needed, a system of least privilege and separation of duty. One solution is to assign permissions to roles and then assign users to roles, this is the approach taken in [4].

The problem all these systems suffer is that to achieve useful results, the policy must become enormous. As well, system administrators "must resort to complex policy modifications to resolve conflicts"[6]. A well designed security framework should be flexible, economic and have a simple policy[9]. No model can meet all the security needs of modern operating system. A successful policy must be flexible; there must be a clean separation of policy from enforcement[10]. We believe that maintaining state will reduce the complexity of the policy and still achieve the same level of security. Our increasing use of web protocols and network communication means that access control needs to be stateful in order to ensure proper *conversations* are taking place between agents[7]. It has been suggested that conversations can be modeled with a finite state transition system[7].

The next few sections introduce the key kernel concepts and background required to implement our model.

1.1 Linux Kernel and Files

The Linux kernel views everything as a file (keyboards, disks, printers, screens, etc). For example when a process wishes to update the screen with some information, it simply writes the information to the screen file and the kernel updates the screen accordingly.

Real files, things that live on a disk or other media are represented as inodes or information nodes in the filesystem.[2] An inode is a data structure that contains information about files, such as owner, group, type of file (directory, plain file or special file) or permissions and pointers to blocks of data.

Directories are files, which means they are actually inodes, the permission to write a file to a directory is granted by first checking that permission is given to update the directory inode. A directory is just a mapping between filenames and inodes[3].

In UNIX[4], files may be listed in several directories, the filename of a file is just a string in a directory. The multiple filenames are known as hard links to the inode (this is not to be confused with symbolic linking).

[1] The name POSIX is used by the POSIX security drafts, .1e and .2c, were never ratified.

[2] Although some things are are not real files also have inodes associated with them.

[3] This is why a move command on linux is so fast, move is just deleting an entry in one directory and inserting it in another; rename is equally quick.

[4] Actually in the ufs and ext2/ext3 filesystems

The Linux kernel does not maintain file information for the purposes of permissions, it only deals with inodes, since inodes are the ultimate target of filenames (and this removes any ambiguity). Our implementation only addresses inodes for this same issue.

1.2 Programs and threads

When a program is executed on Linux it becomes a process. Running processes may start subprocesses or enable several threads. Since threads share the same process information, concurrency is an issue when attempting to maintain the state of the task. We use semaphores to protect updates of the state. Processes are spawned by other processes, each process has a unique parent and may have several children.

One of the first things the kernel does when booting is to start a process called init, since it is the first true process on the system, it is assigned the process id (pid) of 1. All subsequent processes are children of init.

1.3 LSM

In order for Linux to remain secure in the changing environment, a stronger mechanism had to be built to prevent Linux from becoming a "fortress built upon sand."[9]. To address growing concerns over security, the Linux Security Modules (LSM) framework was created. LSM has been part of the Linux kernel since version 2.6. In 2001, the NSA presented its work on SELinux to Linus Torvalds[1] and expressed an interest in having SELinux incorporated into the mainstream kernel. Linus acknowledged that some sort of enhanced security was needed, but didn't want to choose the implementation. Linus opted to create a framework for Linux security modules (LSM).

A kernel module is a self contained piece of the kernel that can be dynamically inserted and removed from the running kernel[5]. Kernel modules have previously been used without the LSM for policy enforcement.

[5]The module framework also allows for modules to be compiled statically into the kernel. Modules that are statically compiled still behave as though they were dynamically loaded.

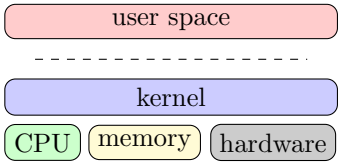

Figure 1: Separation of user space and kernel space

In Linux the kernel runs in a memory space that is separate from user programs. The kernel has access to the hardware and memory of the system and user processes must access hardware via the kernel as shown in figure 1. User processes access hardware and memory via system calls.

Most modules resort to system call interposition to *overpower* the system call[11]. To exemplify system call interposition, assume a module wishes to affect the outcome of `open` calls. The module x will inform the kernel that `open` calls are now handled by x. The module must then perform the checks it wishes to do on incoming data and pass the results to the original system call of `open`, or perform the work of the `open` system call itself. System call interposition is a potentially dangerous mechanism because it requires that the security module directly interfere with the system calls. The module needs to know if the API for the system call has changed in any way. Also the interposition function must return appropriate return values to the calling program or risk crashing the program (or worse, crashing the kernel).

LSM uses a series of hooks in the kernel that allow a security module to either allow or reject an operation. In this way the LSM API can remain relatively unchanged while the system calls upon which it depends remain fluid. LSM asks the security module the question "May a subject S perform a kernel operation OP on an internal kernel object OBJ?"[11].

LSM is layered on top of the existing DAC system in Linux. The discretionary access control checks are made before the LSM is consulted, if the DAC checks are negative (indicating permission denied) then LSM is never consulted. This serves two purposes, it reduces the amount of time that LSM checks are made (which improves performance) and it greatly reduces

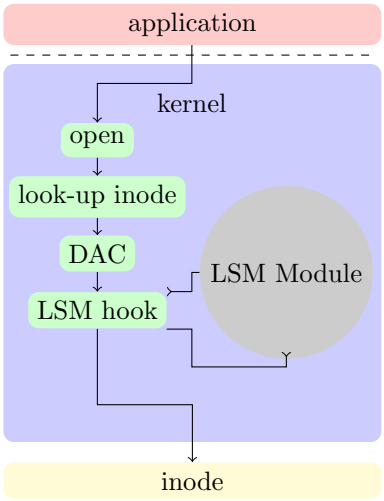

Figure 2: LSM Implementation

```
struct inode {
struct hlist_node i_hash;
struct list_head i_list;
...
void *i_security;
...
}
```

Figure 3: inode struct

the amount of modification to the kernel required to make the LSM hooks authoritative[11]. A diagram of the implementation of LSM is shown in figure 2.

LSM adds a security field to key kernel structures. As described in section 1.1, files on disk are inodes. An inode loaded into memory is held internally by the kernel in a struct inode structure as shown in figure 3. To support LSM, a security field has been added to the inode struct which is a pointer to type void (`void *i_security`). Using a void pointer allows the pointer to point to anything or nothing depending on what the security module decides.

Calls are made to LSM functions (hooks) when a new inode is created or when any inode is accessed by the kernel (read and write operations). As an example, when a new inode is created, a call is made to the lsm function `inode_alloc_security`, this creates an `inode_security_struct` and attaches it to the inode struct. An entry is placed in the kernels inode security cache at this time as well. A call is then made to `inode_create` which determines if permission should be granted to create the inode. LSM is compartmentalized in this way so that the operation of creating a security context is independent from the verification of permission. Similarly when a request is made to delete an inode, the permission to do so is determined before calling the function which removes the inode (`inode_free_security`).

A module identifies itself as an LSM module by calling the function register_security, it can unregister with unregister_security. LSM also provides a notion of stacking. A module may register itself with a primary LSM module as a secondary module[6].

LSM does not provide any locking for the security fields, the module must perform any required locking (for synchronization). Allocation and deallocation of the security structures is also the responsibility of the LSM module. Our module maintains an inode cache, the module must create and destroy entries in this cache as inodes are created, deleted or accessed. To persistently bind security attributes to files, a set of extended attributes are set on the inodes in the filesystem, a complex task[11].

1.4 SELinux

SELinux was developed by the National Security Agency (NSA) and released to the public on December 22, 2000[1]. SELinux is a mandatory access control (MAC) system for Linux. SELinux provides support for several types of MAC policies including role-based access control (RBAC) and multi-level security (MLS). An SELinux policy is composed of allow statements[7] of the form:

```
allow type target_type :  file { permission };
```

[6] We attempted to stack with our SELinux implementation and were unsuccessful.

[7] allow statements are only one of several statements possible in SELinux policy, they are however, the only type of concern in our context.

This is an example of an allow statement, with MAC systems anything not allowed is denied by default. This line would allow a process running with type `type` to do `permission` on a file with type `target_type`. Each file on the system has an selinux security context assigned to it. The security context of the file `/etc/passwd` on our system is `system_u:object_r:etc_t`. The security context is a concatenation of the form `user:role:type`. An SELinux user is similar to a UNIX user. Role is a concept based on role-based access control, it connects a series of users with a collection of programs that they should be allowed to execute[5]. This means for an application to read the /etc/passwd file, its type must have the right to read a file of type etc_t. There are potentially hundreds if not thousands of different types on the system and each type must have rules to access the files it needs to access. This results in several thousand rules in the policy. On our RedHat Enterprise Linux 5® system the default "targeted" SELinux policy has 82460 allow lines. Clearly this is a very complex policy that is difficult to debug and difficult to audit. Our system will attempt to use state to achieve similar goals with more compact policies.

The rest of the paper is organized as follows: in section 2 we introduce our model and its implementation. In section 3 we present examples of Laws and the results achieved on our system. We present further work in section 4 and conclude in section 5.

2 LSMLGI

Our LSM module, lsmlgi, uses the LSM framework to maintain the state of a running process on the system. We maintain the state by incrementing counters of the major file operations being performed by the process. When a process is born we assign unique identifiers to the process. Currently we track read, write, create, delete and execute operations. When a process writes a file, we increment the write counter (likewise for read, delete and execute operations). We update the parent of the process as well so that subprocesses cannot run unrestricted with respect to their parents.

Law Governed Interaction (LGI) is a system of access control that regulates the interaction of agents in a distributed system via laws[8]. The processes running on a system can be thought of as autonomous agents running in a closed system. We attempt to regulate the interactions between these agents (processes) using a simple law system. The main tenant of LGI is that the state of the agents may be updated by any interaction (message), and that the law is not restricted in the actions it may take. The law is not restricted to a simple yes or no answer and may do anything it pleases. With LSM we can only say no to an operation, but we can update the state via our security struct in any way we please. Our initial implementation is restricted in action to a simple yes or no answer and is therefore not as powerful as a system like LGI. We suggest in section 4 that further work should allow our module to take more actions.

2.1 Implementation

A note on running in the kernel: kernel programs do not have direct access to user level data. There is a separation of user space and kernel space. The kernel only knows about uids and gids. To translate usernames into uids on behalf of the kernel, we use a user space helper application (`lawloader`). Our system requires that a law be loaded dynamically; we load the law by using a user space program to send the law in compiled form to the kernel module using the proc filesystem. The proc filesystem (procfs) is a window to kernel space. It stores information about running processes and modules. We create a procfs "file" called `/proc/lsmlgi`. We load a law into lsmlgi by writing to /proc/lsmlgi. When a new law is received on /proc/lsmlgi, the law is compiled and then replaces the current law on the system as depicted in figure 4

When a new law is loaded, we populate a list of uids that the module will consider. Any uid's that are not listed will bypass the module. This allows us to ignore the root account and test scenarios while developing the system. Eventually all accounts will be monitored, but for now we only consider specific accounts.

At each critical lsm hook function we perform the

```
sid: 1197282269
law:
law initialized: 5 rules
      48:-1:6:2:14:2
      500:-1:6:6:12:-20
      500:-1:7:2:15:2
      48:-1:6:2:13:2
      500:-1:7:2:15:2
```

Figure 4: Contents of `/proc/lsmlgi`

necessary memory check requirements then call the function `lsmlgi_check_law`, `lsmlgi_check_law` has access to the currently running tasks' security structure and makes the decision to allow or deny the access based on the current state. If the access is permitted we then call lsmlgi_update_parent to update the state on the current process and recursively to update the state of the parent[8]. It is worth noting that we do not update the state of process id 1. The process with id 1 is `init`, `init` is the master process on the system, it is the first process started on the system. All processes are descendants of `init`[9]. The process of loading our module, initializing a law and then acting upon the law is shown in figure 5.

We defined the following three security structs, `bprm_security_struct`, `task_security_struct` and `inode_security_struct`. We added the fields of sid, tsid and fsid to these structs. We initially use the time in seconds on the system as a unique identifier. To the `task_security_struct` we added the additional counters of read, write, delete and execute.

We use sid (security id) to contain the id of the running lsmlgi implementation, that is, it is a unique key that identifies the running system, when the system is rebooted, a new sid will be created. The task sid is tsid and fsid is the file sid.

When a task is created, we create the security struct for the task using task_alloc_security and create a new tsid for the task. When a task spawns a subprocess, it receives the task sid of its parent. To differentiate between parents and children, the fsid is created uniquely for each task and is independent of lineage. An example context is shown in figure 6.

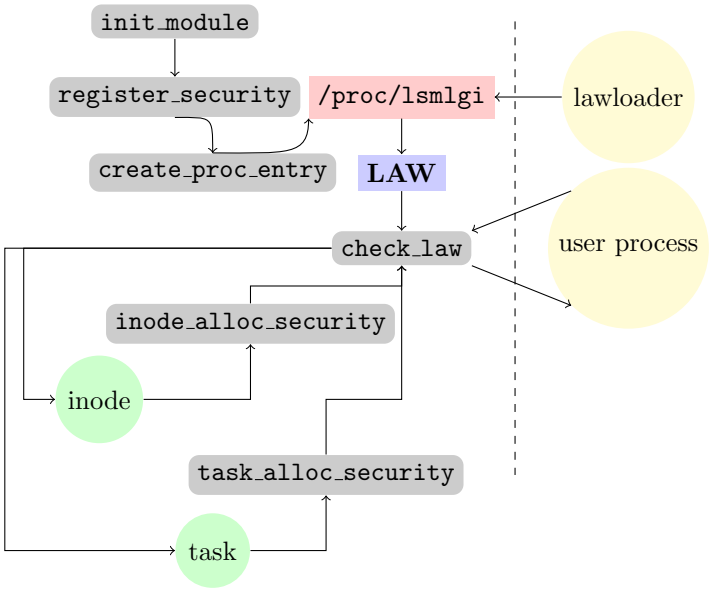

Figure 5: Implementation of our module

```
security.lsmlgi=‘‘1197221584:1197221793:1197221939:\000’’
```

Figure 6: Security Context

The read, write, execute and delete counters of the task structure are updated when each of the respective operations is performed by a task or its children. For example if a child task deletes a file, the child's task structure is updated and the child's parent task structure is also updated. In this way we are able to accurately track a process that spawns helper processes[10]. The current state of the process is exposed to the user in the proc filesystem. For a process with pid 42, the file `/proc/42/attr/current` contains the current state of the process as shown in figure 7

[8] and the grandparents, and the great grandparents, and the great great grandparents, and ... We recurse until we reach pid =1 (init)

[9] We do not update the state on init due to time constraints, init is an important system process, preventing init from operating properly would significantly increase debugging time.

[10] such as a shell

```
sid=1197282269
tsid=1197282275
fsid=0
read=28
write=0
del=0
exec=1
```

Figure 7: Process State information

When a task creates a file, the inode security structure is initialized. We also create an entry in the kernel inode cache table to identify the inode's security to the kernel. This is required if we wish to set a security struct on the inode, in the interest of efficiency when dealing with inodes, the kernel requires that these structures be placed in a cache. We then set the sid of the `inode_security_struct` to the sid of the running system. Next we set the tsid to the tsid of the task creating the file and finally we set the fsid to a new unique sid. We call the combination of `sid:tsid:fsid` is our security context, it is written to the filesystem in the extended attributes of the file. The security context is written to the extended attributes to ensure that the security context is persistent across reboots or if the inode is cleared from memory and/or it needs to be reloaded. The security context of the inode may be updated or overwritten while the inode is in memory, we are currently not updating the context, but this could be used in future work to maintain the tsid of the last task to update the inode.

3 Law Examples

In this section we present examples of simple laws that could be used on a real system to affect a result that is not possible without maintaining state on the system. Our law language is very simple, it consists of a user or group designation followed by a user or group name. We then specify the operation we are interested in controlling (read, write, delete or execute). Next we specify the rule, that is the combination of current task state versus inode state or current task state versus current task state that we wish to block. The rule can be any one of the identifiers (sid, tsid, fsid) or any of the counters (read, write, delete, execute) or a constant. An example law is shown in figure 8.

```
group apache exec { tsid == tsid }
user thomas exec { exec > 20 }
user thomas del { tsid != tsid }
```

Figure 8: Example Law

```
user visitor del { tsid != tsid }
user visitor write { tsid != tsid }
```

Figure 9: Visitor law

3.1 Visitor Account

At our institution there are numerous visitors each day that require some sort of short term account. Most users are using the system to access their home institution accounts. We provide a visitor account for the users to share, but to allow the users to work properly, we cannot restrict their ability to write files in the home directory. Since the account is shared, any user of the account can delete or modify the files of any other user of the account. We implement a visitor law that restricts the delete and write functions of a task to only those files that the task created as shown in figure 9 From the visitors point of view, they can only modify or delete the files that they created in this session. Any files that they leave on the system after logging out will be write and delete locked the next time they log into the system.

3.2 Web Server

A web server may create files while running to track remote users or to log traffic. We consider that the web server should only be able to execute files that were created before it started running, that is, it can-

```
user apache exec { tsid == tsid }
```

Figure 10: Web Server law

```
user thomas exec { exec > 20 }
```

Figure 11: Budget Law

not run any scripts or programs placed on the system after it was started. These files could potentially be maliciously created through some unknown exploit of the web server. We implement this law as shown in figure 10. This law states that files created by this task (process) cannot be executed by this task. This prevents the web server from writing a file and then immediately executing it.

3.3 Budget

In this example we track the number of subprocesses launched by a user or group. This example was inspired by the database query budget suggested in [3]. When the number reaches a critical number, we deny the user or group from starting any further tasks (the user can, however, simply logout and login again to renew her budget) In future the state could be maintained for a user, independent of process information. This would require another structure in the module but would be straight-forward to implement. We did not implement such a system due to time constraints. Our budget example law is shown in figure 11.

4 Further Work

Our module is a very simple implementation of a stateful system. Future work should expand the law language to allow variables to be set dynamically in the security structure. Many applications have well defined "conversations"[7] that could be used in a stateful system to deny any non-conversational actions.

SELinux has support for controlling network operations. Controlling network operations in a stateful manner is very useful, but currently this is handled by another package (iptables). Implementing stateful network control in the same package as access control has the potential to streamline the security policy and make policy validation a simpler task.

SELinux is a mature implementation of the LSM, several user space programs exist to validate policy and to digest error logs for system administrators[5]. Future work should include a mechanism to inform the user at a decision point why an action was blocked. In SELinux this is taken care of by the sealert (SE-Alert) applet which displays human readable interpretations of why access was denied. SELinux is a mature security policy enforcement system, there would be great benefit to making the state transition system of SELinux dynamic. That is, to allow processes to transition dynamically depending on their current state.

More examples of why state is useful to maintain will need to be developed if a real world system is to be created. Adding support for network operations should increase acceptance. Perhaps the most promising method of implementation may be to augment the existing SELinux system with some state saving capabilities.

5 Conclusion

The need for enhanced security policy and access control in modern operating systems is apparent. The LSM framework for Linux is a free and open API. Writing LSM modules has become an accessible way for researchers to work on access control projects. This implementation was completed in a little over 8 weeks by a non-kernel programmer.

Maintaining state allows our module to achieve interesting results with minimal policy. Simple policies are trivial to validate. Most security policy implementations rely on enormous and complex policies to affect similar results. Perhaps the greatest benefit of maintaining state is simplicity.

Stateful access control can achieve very different results than non-stateful access control. Our budget example is simply not possible without maintaining

state. The visitor account example may be possible without maintaining state but would require a very complicated non-stateful policy.

6 Acknowledgments

We would like to thank the NSA for releasing the source code for SELinux to the public. Much of our work was achieved by reading the SELinux implementation and adapting it to our model. We would also like to thank Chris Wright and Stephen Smalley for their work on LSM, making the framework available and providing documentation made this project possible for someone unfamiliar with kernel coding.

We would also like to thank Joško Plazonič for assistance with coding and debugging the module. Vinod Ganapathy for his suggestion of [5]. And finally Naftaly Minsky for the inspiration for the project with his work on LGI.

7 Resources

Source code for this module is available online at:

`http://ramblings.narrabilis.com/wp/linux/stateful-access-control-using-lsm/`

Slides from a talk about this project are available at:

`http://ramblings.narrabilis.com/lsmlgi/lsmlgi-presentation.pdf`

References

[1] National Security Agency. National security agency shares security enhancements to linux. http://www.nsa.gov/releases/relea00027.cfm.

[2] National Security Agency. Selinux. `http://www.nsa.gov`.

[3] Xuhui Ao and Naftaly H. Minsky. On the role of roles: from role-based to role-sensitive access control. In *SACMAT '04: Proceedings of the ninth ACM symposium on Access control models and technologies*, pages 51–60, New York, NY, USA, 2004. ACM.

[4] David F. Ferraiolo. An argument for the role-based access control model. In *SACMAT '01: Proceedings of the sixth ACM symposium on Access control models and technologies*, pages 142–143, New York, NY, USA, 2001. ACM.

[5] Joshua D. Guttman, Amy L. Herzog, John D. Ramsdell, and Clement W. Skorupka. Verifying information flow goals in security-enhanced linux. In *WITS '03: Workshop on Issues in the Theory of Security*, 2003.

[6] Trent Jaeger, Antony Edwards, and Xiaolan Zhang. Managing access control policies using access control spaces. In *SACMAT '02: Proceedings of the seventh ACM symposium on Access control models and technologies*, pages 3–12, New York, NY, USA, 2002. ACM.

[7] Massimo Mecella, Mourad Ouzzani, Federica Paci, and Elisa Bertino. Access control enforcement for conversation-based web services. In *WWW '06: Proceedings of the 15th international conference on World Wide Web*, pages 257–266, New York, NY, USA, 2006. ACM.

[8] Naftaly Minsky. *Law Governed Interaction (LGI): A Distributed Coordination and Control Mechanism.* Department of Computer Science, Rutgers University, 0.9.2 edition, 2005.

[9] Patrick A. Muckelbauer Ruth C. Taylor S. Jeff Turner John F. Farrell Peter A. Loscocco, Stephen D. Smalley. The inevitability of failure: The flawed assumption of security in modern computing environments. Technical report, National Security Agency, 1998.

[10] Stephen Smalley. Which operating system access control technique will provide the greatest overall benefit to users? In *SACMAT '01: Proceedings of the sixth ACM symposium on Access control models and technologies*, pages 147–148, New York, NY, USA, 2001. ACM.

[11] C. Wright, C. Cowan, J. Morris, S. Smalley, and G. Kroah-Hartman. Linux security modules: General security support for the linux kernel. 11th UseNIX Security Symposium, San Francisco, CA, 2002. UseNIX.