Introduction to Linux Security Modules

Draft

Rui Gonçalo

pg18378@alunos.uminho.pt

October 8, 2013

Abstract

"If you cannot explain it simply, you do not understand it well enough."

1 Introduction

In 2001, Peter Loscocco and Stephen Smalley wrote an article introducing the *Security-Enhanced Linux (SELinux)* [1]. The main reason that led to the development of such mechanism was the flawed assumption that adequate security should reside in applications, leaving the role of the operating system behind [2]. They supported the idea that secure applications require secure operating systems. A strong concept related to operating systems security is *access control policy*. In a simple manner, this term specifies which of the operations associated with an object are authorized to perform. Linux kernel inherited from the UNIX security model the *Discretionary Access Control (DAC)* that allows the owner of an object to set the security policy for that object (the control of access is based on the discretion of the owner). However, this model of access control brings some advantages. For instance, every program executed by a certain user receives all of the privileges associated with that user. Therefore it is able to change the permissions of all user's objects, creating potential security threats. In this sense, a *Mandatory Access Control (MAC)* was purposed to protect the system against vulnerabilities left by other access control models. In MAC the operating system constrains the ability of a subject to perform an operation on an object, depending on the security attributes. Whenever a subject attempts to access an object, an authorization rule enforced by the operating system kernel checks these security attributes in order to allow or deny the access.

At the Linux Kernel 2.5 Summit, the National Security Agency (NSA), based on the security issues previously mentioned, presented their work on SELinux, a security mechanism of a flexible access control architecture in the Linux kernel. NSA reiterated the need for such support in the mainstream Linux kernel. Other projects were presented to enforce access policies, namely Domain and Type Enforcement (DTE), Linux Intrusion Detection System (LIDS) and

POSIX.1e capabilities. Given these projects, Linus Torvalds decided to provide a general framework for security policy, named Linux Security Modules (LSM). This framework allow many different access control models to be implemented as loadable kernel modules. Linus enforced that LSM should be truly generic, where using a different security model was a question of loading a different kernel module. The framework should also be conceptually simple, minimally invasive and efficient. At last, the mechanism should be able to support the POSIX.1e capabilities logic as an optional security module [3].

This security framework has motivated developers and gave them freedom to build their own LSM according to how they consider kernel objects should be accessed. SELinux[1] was originally made by the NSA and has been in the mainstream kernel since version 2.6 (December 2003). This module gives executables the minimum privileges required to complete their tasks...

AppArmor (Application Armor)[2] was originally developed by *Immunix*, which was a commercial operating system acquired by *Novell* in 2005. Novell laid off AppArmor programmers in 2007, but they continued the work. Since 2009, *Canonical* contributes to the project. This module has been in the mainstream Linux kernel since version 2.6.36 (October 2010). While SELinux is based on applying labels to files, AppArmor operates with file paths. Many Linux administrators claim that AppArmor is the easiest security module to configure. Yet, others state that it is the worst security mechanism compared to alternatives.

TODO:
Talk about the types of LSM!
They are not actually kernel modules anymore!

2 Design

The basic abstraction of the LSM interface is to intercede in the access to internal kernel objects. Security modules should answer a simple question "May a subject `S` perform a kernel operation `OP` on an internal kernel object `OBJ`?". The mechanism that allow modules to execute this task lies in *hook* functions that are placed in the kernel code, as shown in Figure 1.

Immediately before the kernel access the object, represented as *inode* in Figure 1, the hook makes a call to a function that the LSM module must provide. The module, based on policy rules, allow the access, or deny it, forcing an error code return.

[1] http://selinuxproject.org
[2] http://wiki.apparmor.net

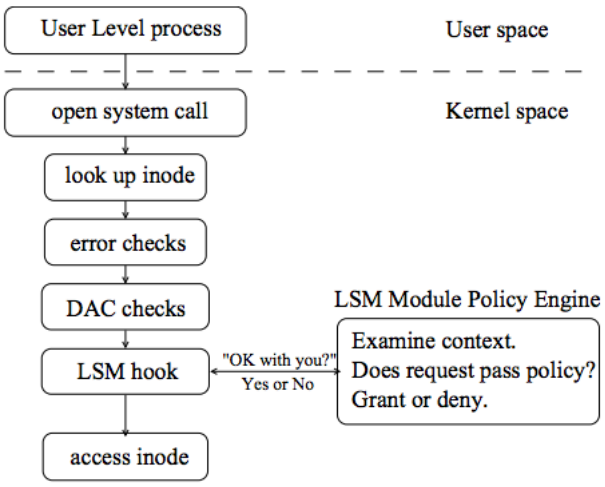

Figure 1: LSM hook functions architecture

3 Implementation

The LSM framework comprises a few files in the kernel. Figure 2 highlights the relevant files that implement the security mechanism.

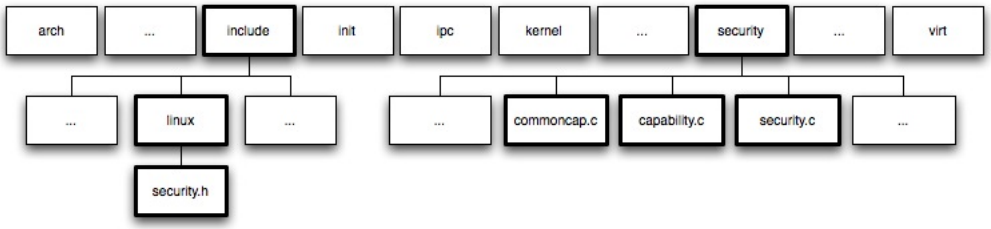

Figure 2: LSM framework files in the Linux kernel

3.1 Header file

The `include/linux/security.h` file contains the hook functions declarations. The source code may be divided into two parts, depending on the value of the conditional group `CONFIG_SECURITY` being true or false. In the first case, an extensive structure with pointers to all hook functions is declared. If false, only default functions are declared and the kernel loads the default security module. The code snippet in Listing 1, extracted from the Linux kernel v3.11, presents the initial function pointers in the structure `security_operations`.

```
struct security_operations {
  char name[SECURITY_NAME_MAX + 1];

  int (*ptrace_access_check) (struct task_struct *child, unsigned int
      mode);
  int (*ptrace_traceme) (struct task_struct *parent);
  int (*capget) (struct task_struct *target,
           kernel_cap_t *effective,
           kernel_cap_t *inheritable, kernel_cap_t *permitted);
  int (*capset) (struct cred *new,
           const struct cred *old,
           const kernel_cap_t *effective,
           const kernel_cap_t *inheritable,
           const kernel_cap_t *permitted);
  int (*capable) (const struct cred *cred, struct user_namespace *ns,
      int cap, int audit);
  int (*quotactl) (int cmds, int type, int id, struct super_block *sb);
  int (*quota_on) (struct dentry *dentry);
  int (*syslog) (int type);
  int (*settime) (const struct timespec *ts, const struct timezone *tz)
      ;
  int (*vm_enough_memory) (struct mm_struct *mm, long pages);
```

Listing 1: Security structure declaration (Linux kernel v3.11)

Along with the structure, the functions prototypes are declared, as shown in Listing 2. Some security hooks are declared depending on conditional groups:

- `CONFIG_SECURITY_PATH`, includes security hooks for pathname based access control;
- `CONFIG_SECURITY_NETWORK`, enables socket and network security hooks;
- `CONFIG_SECURITY_NETWORK_XFRM`, security hooks for XFRM framework, that implement per-packet access controls based on labels derived from IPSec policy;
- `CONFIG_KEYS`, provides support for retaining authentication tokens and access keys in the kernel;
- `CONFIG_AUDIT`, enables auditing infrastructure that can be used with another kernel subsystem.

```
int security_ptrace_access_check(struct task_struct *child, unsigned
    int mode);
int security_ptrace_traceme(struct task_struct *parent);
int security_capget(struct task_struct *target,
        kernel_cap_t *effective,
        kernel_cap_t *inheritable,
        kernel_cap_t *permitted);
int security_capset(struct cred *new, const struct cred *old,
        const kernel_cap_t *effective,
        const kernel_cap_t *inheritable,
        const kernel_cap_t *permitted);
```

```
int security_capable(const struct cred *cred, struct user_namespace *ns
    ,
      int cap);
int security_capable_noaudit(const struct cred *cred, struct
    user_namespace *ns,
           int cap);
int security_quotactl(int cmds, int type, int id, struct super_block *
    sb);
int security_quota_on(struct dentry *dentry);
int security_syslog(int type);
int security_settime(const struct timespec *ts, const struct timezone *
    tz);
int security_vm_enough_memory_mm(struct mm_struct *mm, long pages);
```

Listing 2: Security functions declaration (Linux kernel v3.11)

If the configurable option `CONFIG_SECURITY` is not selected, the default security module is loaded. This module only executes a few capabilities, being permissive in all other hooks, which means that allow access to all kernel internal objects. Listing 3 exhibits some of these hooks' source code.

```
static inline int security_capable(const struct cred *cred,
           struct user_namespace *ns, int cap)
{
  return cap_capable(cred, ns, cap, SECURITY_CAP_AUDIT);
}

static inline int security_capable_noaudit(const struct cred *cred,
             struct user_namespace *ns, int cap) {
  return cap_capable(cred, ns, cap, SECURITY_CAP_NOAUDIT);
}

static inline int security_quotactl(int cmds, int type, int id,
             struct super_block *sb)
{
  return 0;
}

static inline int security_quota_on(struct dentry *dentry)
{
  return 0;
}

static inline int security_syslog(int type)
{
  return 0;
}
```

Listing 3: Default security functions (Linux kernel v3.11)

The same process is kept to the other configurable options. Depending on their values, security hooks are either declared or coded with default instructions.

3.2 Linux capabilities

Linux capabilities were designed to provide a solution to the UNIX-style user privilege set composed by privilege users (`root`) and non-privilege users `regular`

user. The first type has permission to execute every operation and the former can only execute a few set of operations. Thus, processes run either with all permissions or with very restrictive permissions. Unfortunately, most of the time processes do not need all privileges and this exposure raises serious risk when a process gets compromised [4].

In the scope of LSM, a set of functions called common capabilities were developed to give the security framework a default behavior in the case no other LSM is loaded. These functions are plugged in the kernel to overcome the privilege problem mentioned above. In security/commoncap.c we can see the source code of these functions.

If no LSM is loaded, there must be a default function hook that does not execute any operation and let the process access kernel internal objects. The file security/capability.c have all hook functions with the default code. If the return type is void, functions have no operations, otherwise is int and functions just return 0, which is the value to turn the hook permissive. Listing 4 shows some of these hook functions.

```
static int cap_syslog(int type)
{
  return 0;
}

static int cap_quotactl(int cmds, int type, int id, struct super_block
    *sb)
{
  return 0;
}

static int cap_quota_on(struct dentry *dentry)
{
  return 0;
}

static int cap_bprm_check_security(struct linux_binprm *bprm)
{
  return 0;
}

static void cap_bprm_committing_creds(struct linux_binprm *bprm)
{
}
```

Listing 4: Capability functions (Linux kernel v3.11)

These functions are called in the structure security_operations if the respective hook functions are not declared. Listing 5 presents the code snippet of the function security_fixup_ops.

```
#define set_to_cap_if_null(ops, function)       \
  do {                  \
    if (!ops->function) {         \
      ops->function = cap_##function;     \
```

```
      pr_debug("Had to override the " #function \
         " security operation with the default.\n");\
      }           \
  } while (0)

void __init security_fixup_ops(struct security_operations *ops)
{
  set_to_cap_if_null(ops, ptrace_access_check);
  set_to_cap_if_null(ops, ptrace_traceme);
  set_to_cap_if_null(ops, capget);
  set_to_cap_if_null(ops, capset);
  set_to_cap_if_null(ops, capable);
  set_to_cap_if_null(ops, quotactl);
  set_to_cap_if_null(ops, quota_on);
  set_to_cap_if_null(ops, syslog);
  set_to_cap_if_null(ops, settime);
  set_to_cap_if_null(ops, vm_enough_memory);
(...)
}
```

Listing 5: `security_fixup_ops` function (Linux kernel v3.11)

3.3 Framework initialization

The header file mentioned in subsection 3.1 declares some functions in charge of getting the LSM loaded, as shown in Listing 6.

```
/* prototypes */
extern int security_init(void);
extern int security_module_enable(struct security_operations *ops);
extern int register_security(struct security_operations *ops);
extern void __init security_fixup_ops(struct security_operations *ops);
```

Listing 6: Framework initialization functions (Linux kernel v3.11)

These functions are implemented in `security/security.c`. The first function being executed is `security_init`, which source code is present in Listing 7.

```
/* Boot-time LSM user choice */
static __initdata char chosen_lsm[SECURITY_NAME_MAX + 1] =
  CONFIG_DEFAULT_SECURITY;

static struct security_operations *security_ops;
static struct security_operations default_security_ops = {
  .name = "default",
};

(...)

int __init security_init(void)
{
  printk(KERN_INFO "Security Framework initialized\n");

  security_fixup_ops(&default_security_ops);
  security_ops = &default_security_ops;
```

```
  do_security_initcalls();

  return 0;
}
```

Listing 7: `security_init` function (Linux kernel v3.11)

At first, the default module is loaded with the available routines cited in subsection 3.2, by `security_fixup_ops(default_security_ops)`. Then `security_init()` updates the kernel's security structure `security_ops` with the initialized earlier and makes a call to `do_security_initcalls()` that implements a loop presented in Listing 8.

```
static void __init do_security_initcalls(void)
{
  initcall_t *call;
  call = __security_initcall_start;
  while (call < __security_initcall_end) {
    (*call) ();
    call++;
  }
}
```

Listing 8: `do_security_initcalls` function (Linux kernel v3.11)

The callbacks `__security_initcall_start` and `__security_initcall_end` are declared in `include/linux/init.h` and the code snippet is shown in Listing 9.

```
/*
 * Used for initialization calls..
 */
typedef int (*initcall_t)(void);
typedef void (*exitcall_t)(void);

extern initcall_t __con_initcall_start[], __con_initcall_end[];
extern initcall_t __security_initcall_start[], __security_initcall_end
    [];
```

Listing 9: `init` callbacks (Linux kernel v3.11)

3.4 LSM registration

The kernel only runs one LSM, even though there are some available by the time this article was written. Therefore, there must be a way to register the desired module. This is achieved through the execution of `register_security(struct security_operations *ops)`, exhibited in Listing 10

```
int __init register_security(struct security_operations *ops)
{
  if (verify(ops)) {
    printk(KERN_DEBUG "%s could not verify "
           "security_operations structure.\n", __func__);
    return -EINVAL;
  }
```

```
  if (security_ops != &default_security_ops)
    return -EAGAIN;

  security_ops = ops;

  return 0;
}
```

Listing 10: `register_security` function (Linux kernel v3.11)

Some rudimentary check is done on the structure `ops` by `verify(struct security_operations *ops)`. If there is already a security module registered with the kernel, an error will be returned. Otherwise, the structure `security_ops` gets the hook functions in the structure passed by parameter, `ops`, and return success.

There is other important function related to the LSM registration, that is `security_module_enable`. Each LSM must pass this method before registering its own operations to avoid security registration races. This method may also be used to check if the LSM is currently loaded during kernel initialization. Listing 11 presents this function.

```
int __init security_module_enable(struct security_operations *ops)
{
  return !strcmp(ops->name, chosen_lsm);
}
```

Listing 11: `register_security` function (Linux kernel v3.11)

At last, the security functions declarations previously mentioned in subsection 3.1, are implemented by returning the function callback present in the structure `security_operations`. A code snippet is available at Listing 12.

```
int security_socket_create(int family, int type, int protocol, int kern
    )
{
  return security_ops->socket_create(family, type, protocol, kern);
}

int security_socket_post_create(struct socket *sock, int family,
        int type, int protocol, int kern)
{
  return security_ops->socket_post_create(sock, family, type,
           protocol, kern);
}

int security_socket_bind(struct socket *sock, struct sockaddr *address,
     int addrlen)
{
  return security_ops->socket_bind(sock, address, addrlen);
}

int security_socket_connect(struct socket *sock, struct sockaddr *
    address, int addrlen)
{
  return security_ops->socket_connect(sock, address, addrlen);
```

```
}
```

Listing 12: `register_security` function (Linux kernel v3.11)

3.5 Security functions in the kernel

Security functions presented in the previous subsection are called depending on each objective. For instance, the `socket_create` hook is part of the socket implementation, in `net/socket.c`. Note the code snippet in Listing 13.

```
int sock_create_lite(int family, int type, int protocol, struct socket
    **res)
{
  int err;
  struct socket *sock = NULL;

  err = security_socket_create(family, type, protocol, 1);
  if (err)
    goto out;
(...)
}

int __sock_create(struct net *net, int family, int type, int protocol,
       struct socket **res, int kern)
{
  int err;
  struct socket *sock;
  const struct net_proto_family *pf;
(...)
  err = security_socket_create(family, type, protocol, kern);
  if (err)
    return err;
(...)
}
```

Listing 13: `socket_create` hook in socket implementation (Linux kernel v3.11)

This hook is simply a flag where the returned value is checked and if it is different from `0`, the kernel blocks the socket creation. That is the reason why the default capability functions always return `0`.

References

[1] P. Loscocco and S. Smalley, "Meeting Critical Security Objectives with Security-Enhanced Linux," in *Proceedings of the 2001 Ottawa Linux Symposium*, July 2001.

[2] P. A. Loscocco, S. D. Smalley, P. A. Muckelbauer, R. C. Taylor, S. J. Turner, and J. F. Farrell, "The Inevitability of Failure: The Flawed Assumption of Security in Modern Computing Environments," in *Proceedings of the* 21^{st} *National Information Systems Security Conference*, pp. 303–314, October 1998.

[3] C. Wright, C. Cowan, J. Morris, S. Smalley, and G. Kroah-Hartman, "Linux Security Modules: General Security Support for the Linux Kernel," in *Proceedings of the* 11^{th} *USENIX Security Symposium*, August 2002.

[4] Wikipedia, "Capability-based security," September 2013.