Kernel approach for Security

Open Source Developpers’ European Meeting

Philippe Biondi

Webmotion Inc.

2 février 2001
Aim

- Context
- Trustfulness
- Conclusion

Technical description

- Design
- Untamperability
- Unbypassability

Existing projects

- Openwall, Medusa, RSBAC, NSA SE Linux, LIDS

Conclusion

- GACI
Outline

- **Aim**
  - Context
  - Trustfulness
  - Conclusion

- **Technical description**
  - Design
  - Untamperability
  - Unbypassability

- **Existing projects**
  - Openwall, Medusa, RSBAC, NSA SE Linux, LIDS

- **Conclusion**
  - GACI
We are facing

- Fun/hack/defacing
- Tampering
- Resource stealing
- Data stealing
- Destroying
- DoS
- ...
- We must ensure
  - Confidentiality
  - Data integrity
  - Availability

- What we must do to ensure all of this:
  - We define a set of rules describing the way we handle, protect and distribute information.
  - This is called a security policy.
To enforce our security policy, we will use some security code

- Tripwire, AIDE, for data integrity
- SSH, SSL, IP-SEC, cryptography for confidentiality
- Password, secure badge, biometric access controls
- ...

Can we trust them?
The fortress built upon sand — D. Baker – *Proceedings of the New Security Paradigms Workshop*

- User space is untrusted and can take control of the kernel space (module insertion, `/dev/kmem`, …)
  - → kernel space is also untrusted:
The fortress built upon sand — D. Baker – Proceedings of the New Security Paradigms Workshop

- User space is untrusted and can take control of the kernel space (module insertion, /dev/kmem, ...)
  
  ⇒ kernel space is also untrusted:
Security must be built layer by layer.
Each layer is built with the hypothesis the underlayer is trusted.
It is not worth building security applications on untrusted layers

We need:

Why don’t we want user space to be trusted?
The mice and the cookies

- Facts:
  - We have some cookies in a house
  - We want to prevent the mice from eating the cookies
The mice and the cookies

- Solution 1: we protect the house
  - too many variables to cope with (lots of windows, holes, ...)
  - we can’t know all the holes to lock them.
  - we can’t be sure there weren’t any mice before we closed the holes

  This protection can’t be trusted.

- Solution 2: we put the cookies in a metal box
  - we can grasp the entire problem
  - if we trust the metal box, this solution has a good trusting level
  - the cookies don’t care whether mice can break into the house

  This protection can be trusted.
To enforce our security policy, we need to add code to

► protect the kernel and the code itself
  ⇒ trusted kernel space
  ⇒ untamperability

► protect other code/data involved in the security policy
  ⇒ mandatory controls
  ⇒ unbypassability
Aim

- Context
- Trustfulness
- Conclusion

Technical description

- Design
- Untamperability
- Unbypassability

Existing projects

- Openwall, Medusa, RSBAC, NSA SE Linux, LIDS

Conclusion

- GACI
So, we need to

- make the kernel space trusted
  - we protect the kernel and the code itself
    - we must block everything coming from user space

- protect other code/data involved in the security policy
  - we rely on the fact that we trust kernel space
  - we add controls on user space
    - make our code a mandatory way
Why should the last layer be the kernel space?
Because of the design of the CPU (PMMU),

- we have few entry points
  - untamperability
- we can force everything to go through kernel space
  - unbypassability
The kernel space is unreachable by user space code

The execution of some defined kernel code can be triggered
  ➤ system calls
  ➤ devices
  ➤ procfs
  ➤ hardware interruptions

Few entry points, opened by the kernel side
  ➤ /dev/mem, /dev/kmem
  ➤ /dev/port, ioperm and iopl
  ➤ insmod and rmmod
  ➤ reboot and halt
Because of protected mode mechanisms, kernel coders don’t do buffer overflows programming faults.

```
linux/drivers/char/rtc.c

static int rtc_ioctl(struct inode *inode, struct file *file, unsigned int cmd, unsigned long arg)
{
    unsigned long flags;
    struct rtc_time wtime;

    switch (cmd) {
        [...]
        case RTC_ALM_SET: /* Store a time into the alarm */
        {
            unsigned char hrs, min, sec;
            struct rtc_time alm_tm;

            if (copy_from_user(&alm_tm, (struct rtc_time*)arg, sizeof(struct rtc_time)))
                return -EFAULT;
```
/dev/mem, /dev/kmem and /dev/port protection:

```c
static int open_port(struct inode * inode,
                     struct file * filp)
{
    return capable(CAP_SYS_RAWIO) ? 0 : -EPERM;
}
```
Module insertion control:

```c
asmlinkage unsigned long
sys_create_module(const char *name_user, size_t size)
{
    char *name;
    long namelen, error;
    struct module *mod;

    if (!capable(CAP_SYS_MODULE))
        return -EPERM;

    [...]
```
Reboot/halt can’t be forbidden:

- UPS must be able to shutdown
- Reboot is mostly user space stuff, the kernel just reboot the CPU
- No difference with a runlevel change

⇒ We need to guarantee a safe boot sequence, which is a huge problem
Boot sequence

- POST: Console vulnerable
- Boot loader: Console vulnerable / rely on boot disk
- Kernel: Rely on boot disk (kernel image)

- booting process (init, rc scripts, daemons, ...)
- working state
What must we protect?

- What is in memory
  - Processes
  - Kernel configuration (firewall rules, etc.)

- What is on disks or tapes
  - Files
  - Metadata (filesystems, partition tables, boot loaders, ...)

- Hardware
  - EPROMs, configurable hardware, ...
User space can’t access these items without asking the kernel

► system calls are a place of choice for controlling accesses
We’ll use a modular architecture to control syscalls: there will be

- An enforcer component

- A decider component
  
  - Lots of access control policies (DAC, MAC, ACL, RBAC, IBAC, ...)

![Diagram of modular architecture with components: app, syscall, enforcer component, decider component.]
How to add the enforcer code to the syscalls?

- Syscall interception
- Syscall modification

System call anatomy:

- User space
- Kernel space
- User space

Diagram:

- App
- Dispatching code
- System calls: open(), chmod(), kill(), execve(), syscall()
Syscall interception example: Medusa DS9

```
linux/arch/i386/kernel/entry.S

[...]
GET_CURRENT(%ebx)
cmpl $(NR_syscalls),%eax
jae badsys

#ifndef CONFIG_MEDUSA_SYSCALL
    /* cannot change: eax=syscall, ebx=current */
btl %eax, med_syscall(%ebx)
jnc 1f
pushl %ebx
pushl %eax
call SYMBOL_NAME(medusa_syscall_watch)
cmpl $1, %eax
popl %eax
popl %ebx
jc 3f
jne 2f
1:
#endif

testb $0x20, flags(%ebx)      # PF_TRACESYS
jne tracesys
[...]
```
Syscall interception advantages

- general system
- low cost patch

Drawbacks

- kind of duplication of every syscall
- need to know and interpret parameters for each different syscall
- architecture dependent
Syscall modification example: LIDS

```c
linux/fs/open.c

asmlinkage long sys_utime(char * filename, struct utimbuf * times)
{
    int error;
    struct nameidata nd;
    struct inode * inode;
    struct iattr newattrs;

    error = user_path_walk(filename, &nd);
    if (error)
        goto out;
    inode = nd.dentry->d_inode;

    error = -EROFS;
    if (IS_RDONLY(inode))
        goto dput_and_out;

    #ifdef CONFIG_LIDS
    if(lids_load && lids_local_load) {
        if (lids_check_base(nd.dentry,LIDS_WRITE)) {
            lids_security_alert("Try to change utime of ",filename);
            goto dput_and_out;
        }
    }
    #endif

    /* Don’t worry, the checks are done in inode_change_ok() */
    newattrs.ia_valid = ATTR_CTIME | ATTR_MTIME | ATTR_ATIME;
    if (times) {
        ...
    }
}"
```
Syscall modification advantages

- Syscall parameters already interpreted and checked
- Great tuning power. We can alter the part of the syscall we want.

Drawbacks

- Each of the syscall must be altered (near 200 syscalls)
Aim
  ► Context
  ► Trustfulness
  ► Conclusion

Technical description
  ► Design
  ► Untamperability
  ► Unbypassability

Existing projects
  ► Openwall, Medusa, RSBAC, NSA SE Linux, LIDS

Conclusion
  ► GACI
Collection of security-related features for the Linux kernel.

- Non-executable user stack area
- Restricted links in `/tmp`
- Restricted FIFOs in `/tmp`
- Restricted `/proc`
- Special handling of fd 0, 1, and 2
- Enforce `RLIMIT_NPROC` on `execve`
Medusa DS9

Authors: Marek Zelem Milan Pikula Martin Ockajak
Extending the standard Linux (Unix) security architecture with a user-space authorization server.

- **layer 1**
  - Hooks in the original kernel code

- **layer 2**
  - kernel space code
  - called from hooks.
  - do basic permission checks
  - check for cached permissions
  - call the communication layer if necessary

- **layer 3**
  - communication layer
  - communicate with a user space daemon
- User space daemon
  - decider component

- Miscellaneous
  - syscall interception
  - can force code to be executed after a syscall
Authors: Amon Ott, Simone Fischer-Hübner, Morton Swimmer
Rule Set Based Access Control

- It is based on the Generalized Framework for Access Control (GFAC)
- All security relevant system calls are extended by security enforcement code.
- Different access control policies implemented as kernel modules
  - MAC, ACL, RC (role control), FC (Functional Control), MS (Malware Scan), ...
NSA Security Enhanced Linux

- It is based on the Flask architecture
  (Flexible architecture security kernel)

- Enforcer / decider components

- Pays a lot of attention to the change of the access control policy
  (revocation)
LIDS

Authors: Xie Huangang, Philippe Biondi
Linux Intrusion Detection System

- Self-protection
- Files protection
- Processes protection
- Online administration
- Special features
  - Dedicated mailer in the kernel
  - Scan detector in the kernel
Self-protection

- Modules insertion/deletion, /dev/mem, ..., ioperm/iopl filtered
- Boot process protected
  - Can forbid the execution of non-protected programs (not flawless)
- Sealing mechanism
  - fsck or insmod can run when booting
  - no human intervention is needed to seal the protection
  - after the seal, we are in the working state. Everything is locked
Files protection

- MAC-like approach:
  ```bash
  lidsadm -A -s /usr/sbin/httpd -o /home/httpd -j
  ```
  READ

- Files identified by VFS device/inode ⇒ works on every fs
Processes protection

- Rely on the linux capabilities bounding set
  - Hardware protection
  - Processes privacy (ptrace, promiscuous mode, ... can be forbidden)
  - Network administration locked
- Daemons can be made unkillable
- Processes can be made invisible
Online administration

- LIDS can be disabled globally
- LIDS can be reconfigured on the fly
- LIDS can be totally disabled only for a shell and its children
Special features

- Mailer in the kernel
  - Can make a network connection (TCP or UDP)
  - Can send a scriptable session (mail, syslog, ...)
  - Does not rely on anything in user space

- Scan detector in the kernel
  - kind-of interrupt driven ⇒ no load at all
  - does not need the promiscuous mode
  - works on every interface
LIDS general architecture

Boot stuff

Kernel image

lidsadm

LIDS AC data

init, rc, daemons

working stuff

syslog

procfs stuff

init code

AC data

decider component

scan detector

syscalls

enforcer component

Logging stuff

Kernel Mailer
Outline

- Aim
  - Context
  - Trustfulness
  - Conclusion

- Technical description
  - Design
  - Untamperability
  - Unbypassability

- Existing projects
  - Openwall, Medusa, RSBAC, NSA SE Linux, LIDS

- Conclusion
  - GACI
General Access Control Interface

- Very young project, at the very beginning
- Aims to be the security interface for Linux 2.5
- Gathers coders from Medusa, RSBAC and LIDS