Message Filters for hardening the Linux kernel

Suneetha N
Dept. of Computer Science
Indian Institute of Technology Madras
Chennai -36
suneetha@cs.iitm.ernet.in

D Janakiram
Dept. of Computer Science
Indian Institute of Technology Madras
Chennai -36
djram@iitm.ac.in

ABSTRACT
Various mechanisms for hardening the Linux kernel (for example, enforcing system call policies, device driver failure recovery, protection against exploitation of bugs in code) are proposed in the literature. The main problem with these mechanisms is that, they require changes in the kernel code leading to the possibility of introducing new bugs and hence increasing the testing time. We propose a message filter model as extension to object oriented wrappers for the Linux kernel, to dynamically provide various filtering capabilities to the kernel. This model works as a comprehensive framework for specifying system call policies, handling device driver faults, protecting the kernel against exploits of bugs in code etc, without modifying the existing kernel code. This considerably reduces the possibility of creating new bugs in the kernel code. We have integrated policies for system call interception and device driver failure handling, into the Linux kernel (2.6.9), using message filter model. Our experiments show that the overhead due to filter objects is very low, making it a useful mechanism for providing filtering capabilities to the Linux kernel.

Categories and Subject Descriptors
D.4.6 [Security and Protection]:

General Terms
Security, Reliability

Keywords
Operating systems, Linux kernel, Security, Device drivers, Message filters, Object Orientation

1. INTRODUCTION
Linux is a widely used open source operating system. It uses traditional access control lists to grant access to system resources by distinguishing normal users from root or privileged users. The applications run either in user mode or kernel mode. The switch from the user mode to the kernel mode is through system call interface, with the privileged system calls being accessible only to the root user. The kernel space can be viewed as consisting of two parts viz. core kernel comprising of memory, process, timer, interrupt subsystems and the loadable kernel modules such as device drivers, file systems etc.

Various attempts have been made in literature to harden the Linux kernel. Confinement of the application interaction with the kernel, through specified policies on system call interaction, has been addressed in [17], [12], [6]. They make the kernel secure by disallowing kernel interactions from untrusted applications. To prevent device driver failures from crashing the kernel, NOOKS architecture provides shadow device driver approach which assumes the role of failed device driver, during recovery, making the kernel more reliable [14]. These enhancements are addressed in two ways.

1. Loadable kernel modules:
   The extensions are provided as kernel modules which are inserted into the kernel at runtime. The main disadvantage of this approach is that, new functionality on top of the base kernel functionality can be added but the functionality of existing kernel cannot be modified.

2. Modifying the kernel code:
   The existing kernel code is modified to add new functionality. This is advantageous because existing functionality of the kernel can be modified, but it may introduce new bugs into the kernel code. Also, it creates the need for testing the new code and other parts of the kernel dependent on this, thus increasing the testing time. Also, modifying the kernel code requires recompilation of the kernel.

We propose a novel object oriented message filter model for the Linux kernel to dynamically provide various filtering capabilities to the kernel, transparent to user applications. The message filter model checks and filters the requests inside kernel space before they are serviced by the kernel. Our
message filters are developed as extensions to object oriented wrappers developed for the Linux kernel [7]. The message filter model is useful in intercepting system calls to build an intrusion detection system, handling device driver faults etc. Although the additional mechanisms (e.g. security related modules and C++ device drivers with filters) are loaded as kernel modules, the presence of object oriented wrappers helps to change the behavior of modules, dynamically through filter objects. These filters are linked dynamically to the wrapper objects. The additional functionality (e.g. security checks) to be performed is added as filter objects to the kernel.

The rest of the paper is organized as follows. Section 2 presents the design of message filter model and the issues involved in building it. Section 3 gives the implementation details. In Section 4, we discuss the applications of message filter model for hardening the Linux kernel. The advantages and performance of the model are presented in sections 5 and 6 and we conclude the paper in section 7.

2. DESIGN

We propose a filtering mechanism based on object oriented wrappers for the Linux kernel, to filter messages from faulty and malicious applications, from reaching the kernel.

One challenge is to provide a transparent filter model so that the existing kernel and applications work the same way. Also, filtering conditions need to be chosen dynamically, based on the filtering requirements. It should be easier to add or delete the filtering conditions. Since the in-kernel code is huge and suffers from coupling [18] and cohesion [13] problems, it should be easier to provide better maintainable and extensible kernel extensions.

The object oriented wrappers for the Linux kernel abstract the major subsystems like memory, process, timer subsystems of the Linux kernel [7]. These wrappers can be used to develop better maintainable and extensible kernel modules (device drivers, security modules) in C++. They can store the state of variables such as values of critical parameters for the drivers. Message filter model [8] for object oriented systems proposes filter objects that are like conventional C++ objects but are intended for transparent interception of messages. The filter model proposes a filter relationship among classes that introduces filter objects between them to intercept the communication.

We propose the concept of message filters for the Linux kernel that can intercept messages before they reach the core kernel. Filter classes, extensions to wrappers, are used to specify the set of rules or preconditions (policies) that need to be checked before kernel processing. Thus they can be used to prevent malicious messages from reaching the kernel. Message filters can be applied at the wrapper end or at the device driver end based on requirement.

Design of the Linux kernel with wrappers and filters is as shown in figure 1. The communication with the kernel in case of modified kernel with filter model is along the dotted lines shown in the figure 1. Filters for system call wrappers is useful to intercept system calls for applying policies on them. The system call table is modified to hold the filter functions and hence any system call request from applications reaches the filters which decide the execution of the request.

The device drivers developed in C++ use wrappers for communication with the core kernel. The filter model helps in filtering messages from faulty device drivers by verifying the parameter values of critical parameters of the drivers. This is explained in detail in further sections.

This design provides a transparent way of intercepting messages for the kernel. The users and applications are not aware of the presence of filter model as there is no change in the user interface. Also, the design provides an easy, extensible and maintainable way of adding new policies.

3. IMPLEMENTATION DETAILS

We have implemented the model of message filters as wrappers with filtering capability for the Linux kernel (2.6.9). We provide a keyword “fclass” as language extension to C++ and a tool to parse the keyword and add filtering extensions to the class [10]. The filter classes are realized as decorator class implementation [5] of the C++ wrapper classes. The tool adds a function called policy for each decorator class, whose code determines the preconditions. This filter member function is associated with a particular member function of the wrapper class and intercepts the calls to this function.

3.1 Policy specification
Policy is a set of preconditions that need to be fulfilled before servicing the requested functionality. These rules are specified in policy function of decorator class. A sample policy is shown below.

```cpp
void decorator_derived1::func1()
{
    if(policy())
        //call the original function
    decorator_base::func1();
}

//A sample policy specification
int decorator_derived1::policy()
{
    if(proc_write_file()==1)
        //function which verifies the output
        //from user process
        return 1;
    else
        return 0;
}
```

Message filters can be used as an intrusion detection framework, system call policy engine or faulty driver isolation, based on the type of policy specified and the set of wrappers used. Section 4 talks of the applicability of the proposed framework for the Linux kernel.

4. APPLICATIONS

4.1 System call interception

System call interface is the only way user applications can interact with the kernel. Persistent changes to system are made only through system calls. An intruder may exploit the system call interface for accessing sensitive data or manipulate the kernel protection mechanisms that can compromise the system security even after the intruder logs out of the system.

The system can be protected from these attacks by authenticating user in a passive way (without direct user intervention). We propose a biometric based protection mechanism to prevent possible corruption of the system due to intruder intervention. The idea is to authenticate the user through passive biometric systems (webcam or fingerprint sensor attached to mouse) when the system calls are invoked on sensitive data. Classification of system calls based on threat levels is proposed in [1]. We intercept only the system calls of threat level 1 i.e. the systems calls whose exploitation can give complete control of the system to the intruder.

In this section, we demonstrate the usage of message filters for intercepting system calls and specifying policies, to provide an intrusion detection system. Wrappers to system calls provide a clear abstraction for system call functions of the kernel. We use a hybrid system call interception model with the interception mechanism at the kernel level and a provision for policy verification either at kernel or user level. We use a user level policy verification, as the biometric verification code is huge and it uses opencv [2] user level library. Biometric verification is implemented as a policy of filters. We used Viola-Jones algorithm [16] provided by opencv, for face detection and eigen faces for face recognition algorithm [15] for face recognition.

4.1.1 Working:

Figure 2 shows the working of the biometric protection model.

1. When an intruder triggers an operation such as chmod(/etc/passwd,666) from the user space, a trap is issued to the kernel.
2. The function corresponding to chmod in the system call table is invoked, which in our case is the new chmod() (Filter function).
3. The policy is to trigger the user process to perform biometric verification (face recognition). The communication between the kernel module and the user process of biometric verification is through a /proc file created for this purpose, by the biometric module.
4. The user process which is continuously monitoring the /proc file reads the request from the kernel and performs the biometric verification which in our case is face recognition. Face detection is performed on the image and is verified against root user of the system. The outcome of the verification process is written back into the /proc file.
5. The kernel reads the input from the user space.
6. Based on the outcome, the system call processing is transferred to the wrapper to call the original kernel function (sys chmod).

4.1.2 Evaluation

1. The system call table is modified to hold a pointer to the filter function. 1 The system call interface for users is not change and hence the user applications are transparent to filtering.
2. Based on the sensitivity of the request, corresponding filter object (which holds the policy) is invoked dynamically.
3. To add new policies, new filter objects are defined for wrappers.
4. The overhead achieved through the redirection is minimal, as shown in table 2, for various critical system calls. The additional overhead would be the policy

1Since the kernel does not understand C++, a C function is created in extern “C” block which invokes the filter function. This C function is registered with the kernel.
execution time, which in this case is the biometric verification system as shown in table 3.

Since the application makes fewer critical system calls on sensitive data, the overhead incurred is considerably low. Thus the message filter model with object oriented wrappers for the Linux kernel enables dynamic and transparent intrusion detection system with no perceivable overhead.

A continuous biometric verification system is proposed in [9] which uses similar system call classification of [1]. We could provide on-demand biometric verification because the filter objects are associated dynamically with the system call wrapper. We reduce the overhead incurred by restricting the biometric verification only for sensitive data requests.

**4.2 Device driver failure handling**

Device drivers interact with the core kernel for handling interrupts, memory allocation etc. The faults in device drivers can crash the kernel [4].

The need for development of device drivers in object oriented fashion and a framework to support the same is proposed in [7]. The filter model proposed in this paper is useful for filtering the messages of faulty drivers from reaching the kernel. Wrappers help in storing the state of parameter values before passing them to the kernel. The tool [10] helps to develop the filter classes for C++ device drivers. The policy code added to the driver is to monitor the critical parameters for faulty values (specified by the developer). In case of a faulty device driver, if the value of a critical parameter is faulty, the message is filtered thus protecting the kernel from corruption by the faulty messages.

We have re-engineered ne2k_pci.c, which is the network driver for PCI NE2000 clones, into C++. The driver uses wrappers for communication with the core kernel. Struct net_device *dev and struct e8390_pkt_hdr *hdr were identified as critical parameters whose faulty values can crash the kernel. The concept of message filters is useful in this scenario to prevent the faulty values from corrupting the kernel. Message filters are added for the device driver using [10] tool. The policy is the set of possible values for these parameters and the kernel function is invoked after verifying the parameter values. This way the kernel can be protected from corruption by faulty device drivers.

**4.3 Protection from exploitation of bugs in code to gain root shell**

Bugs in code are often exploited to gain root shell. Uid setting system calls are used by an intruder to escalate his privileges by performing successful buffer overflow attacks [11]. Uid setting requests can be from interactive or setuid or background processes. Interactive tasks are started by root user. Setuid tasks are requests from non root users for privilege escalation. Background processes are system services which have root privileges and started by the system. The filter model proposed in this paper can be used to monitor the setuid tasks and verify the authenticity of requests.

```c
//For setuid request
int decorator Derived::policy()
{
    if (!(current->euid) && (current->uid))
    //a setuid request from a non root user
    {
        if(check database())
```

---

**Figure 2: Intrusion detection system using filter model**

---
A database of programs allowed to make setuid requests can be maintained as in [1]. The above policy checks the database of valid programs to invoke “setuid” for a particular user and allows the execution accordingly. Requests as part of buffer overflow attacks will be denied as a request from an untrusted applications or the user is asked to re-enter his password.

5. ADVANTAGES

1. The message filter model can be used for real time monitoring. One direct application is to build an intrusion detection system that monitors and filters the interactions with the kernel.

2. Message filters are in the kernel space. Hence they are safe from manipulations by normal users.

3. It provides an ability to add and enforce variety of security policies.

4. The user applications are transparent to the modifications made by filters.

5. Since the model is developed in an object oriented way abstracting the core functionality of the kernel, adding extensible and maintainable kernel extensions is made easier.

6. PERFORMANCE

The prototype model is implemented for the Linux kernel (2.6.9). We have analyzed the overhead of filter model using HBench [3] benchmarking tool and “gettimeofday” system call to measure the overhead for one single system call.

Test Bed:
The experiments are conducted on an Intel Celeron 2GHz processor with a 512 MB RAM and 80 GB HDD under no external load.

Table 1: HBench Results

<table>
<thead>
<tr>
<th>Test</th>
<th>Original kernel (microseconds)</th>
<th>Modified kernel (microseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>lat_syscall_write</td>
<td>0.3855</td>
<td>0.3864</td>
</tr>
<tr>
<td>lat_syscall_sigaction</td>
<td>0.7053</td>
<td>0.7057</td>
</tr>
<tr>
<td>lat_syscall_gettimeofday</td>
<td>0.3853</td>
<td>0.3856</td>
</tr>
<tr>
<td>lat_syscall_getpid</td>
<td>0.0092</td>
<td>0.0104</td>
</tr>
<tr>
<td>lat_syscall_sbrk</td>
<td>0.7426</td>
<td>0.7456</td>
</tr>
<tr>
<td>lat_syscall_sbrk</td>
<td>0.0143</td>
<td>0.0145</td>
</tr>
</tbody>
</table>

return 1;
else
  return 0;
}

HBench tool measures performance of the kernel by performing various tests to measure read and write bandwidths and latencies of various system calls. The system call latencies measured using HBench, shown in table 1, do not show any variation as there are no modifications for the non critical system calls. The system calls classified as critical are not tested by the benchmarking tool. Hence we used “gettimeofday” system call to measure the time taken for the redirection through filters. The policy() function is replaced with a printk statement. The experiment is performed on the kernel with C++ support, with and without the redirection of critical system calls to wrappers and the time taken for some of the system calls is shown in table 2 and the overhead for biometric verification is hown in table 3. An overhead of 40 microseconds is observed for filter redirection. The readings are averaged over 100 executions of the system calls.

7. CONCLUSION

We have proposed a message filter model for the Linux kernel which can be used as a comprehensive protection framework for hardening the Linux kernel. It protects against intrusions, malicious interactions in code etc. depending on the policy specified. It is dynamic and transparent to the user. It is easier to extend and maintain new policies because of the object oriented interface. Also, the filter model is useful in operating systems with object oriented components (Mac OS has object oriented device drivers), to filter the messages between the kernel and components.

8. ACKNOWLEDGMENTS

We acknowledge the Department of Information Technology, Government of India for sponsoring the Linux project. We would also like to thank Mr. P. Nagaraju for implementing the message filters for Linux kernel. We thank Dr. Sukhendu Das for helping us in building intrusion detection system for the Linux kernel.
9. REFERENCES


