Towards a VMM-based Usage Control Framework for OS Kernel Integrity Protection

Min Xu
George Mason University
Fairfax, Virginia, USA
mxu@gmu.edu

Xuxian Jiang
George Mason University
Fairfax, Virginia, USA
xjiang@gmu.edu

Ravi Sandhu
Institute for Cyber-Security Research
Univ. of Texas at San Antonio, USA
ravi.sandhu@utsa.edu

Xinwen Zhang
Samsung Information Systems America
San Jose, California, USA
xinwen.z@samsung.com

ABSTRACT

Protecting kernel integrity is one of the fundamental security objectives in building a trustworthy operating system (OS). For this end, a variety of approaches and systems have been proposed and developed. However, access control models used in most of these systems are not expressive enough to capture important security requirements such as continuous policy enforcement and mutable process and object attributes. Even worse, most existing protection mechanisms in these systems reside in the same space as the running OS, which unfortunately can be disabled or subverted after an attacker successfully exploits kernel-level vulnerabilities (or features) to compromise the OS kernel. The increasing number of kernel-level rootkit attacks clearly demonstrates this threat.

In this paper we present a simple but effective usage control model UCON$_{\mathcal{KI}}$ with unique properties of decision continuity and attribute mutability for OS kernel integrity protection. Further, to enforce UCON$_{\mathcal{KI}}$ security policies, we propose a virtual machine monitor (VMM) based architecture that is isolated and protected from other untrusted processes inside a virtual machine (VM). We have implemented a proof-of-concept prototype in Linux to demonstrate the feasibility of our approach. Our experiments with 18 real-world kernel rootkits show that our approach is able to successfully detect and prevent all kernel integrity violations from them. Beyond kernel integrity protection, we also explore additional opportunities for general OS security, such as the confinement of process activities as well as the protection of system utility programs at the VMM level.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Access controls*; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection—*Unauthorized access*

General Terms

Security

Keywords

authorization, access control, usage control, UCON, operating system protection, kernel integrity, VMM, security architecture

1. INTRODUCTION

Despite efforts for decades, commodity operating systems (OS) today continue to be buggy, and vulnerabilities such as buffer overflows are frequently found. Exploiting a kernel-level vulnerability not only gives an attacker access to system resources, but also allows the attacker to modify the operating system kernel and important system utilities, hence compromising the integrity of the entire system. Due to the fact that all user applications rely on the integrity of kernel and core system utilities, compromising any part of the kernel leads to complete penetration of the entire system.

A variety of approaches such as [11, 16] have been proposed and deployed to enhance the security and trustworthiness of OS kernel. Unfortunately, most, if not all, existing protection mechanisms reside in the same space as the victim OS kernel. As a result, once an attacker takes control of the machine, the protection mechanism could be potentially disabled or subverted. For example, many security mechanisms use loadable kernel modules to define and enforce access control policies. With the very same OS feature or by exploiting a kernel-level vulnerability, an attacker can directly compromise the underlying access control mechanism so that security checks are disabled or defined security policies are changed [23, 24].

Typically, kernel integrity protection is achieved through access control mechanisms by confining a process's activities. Several access control models have been developed. Traditional Unix systems implement discretionary access control (DAC) policies with access control lists (ACL's), which however cannot prevent attacks such as Trojan horses [10]. Early mandatory access control (MAC) polices, such as the Bell-LaPadula secrecy policy [6] and the Biba integrity policy [7], define clear security goals, but are too restrictive for convenient use of applications. These approaches provide insufficient support for data and application integrity, separation of duty and least privilege requirements. Recent efforts at MAC systems (e.g., Flask [22] and SELinux [16]) use flexible and convenient access control models, but demonstrate that particular security goals are more difficult to meet. Flexible access control models typically result in more complex policies, so it is more difficult to determine if these policies have desired effects. For example, SELinux supports a variety of policies by using an extended Type Enforcement model [8] for most policy development. However, a typical SELinux system for Linux 2.6.18 has 29 different classes of objects, hundreds of possible operations, and over 50,000 policy rules. Understanding and configuring a policy that guarantees satisfaction of system security goals is nontrivial and difficult even for a security expert.

Another significant shortcoming of existing MAC models in many Unix and Linux systems is that they cannot satisfy increasing security requirements such as continuous authorization enforcement. In traditional access control models, once an authorization decision has been made, there is no further checks during the access process. However, it is desired in many systems where a decision can be revoked after granting if some conditions change during the access. A related aspect of this continuous security check requirement is the update of subject and object attributes. Subject and object attributes are factors to make decisions of access requests by a *reference monitor*. In *dynamic* systems, attribute values can be changed, not only because of system administrative purposes, but also, and more importantly, because of the side-effects of historical or ongoing accesses from subjects. Attribute changes, recursively, can result in further decision checks of a granted access or affect future access requests. Note that interactive security properties can be achieved with these two aspects. For example, in a Linux kernel, the system call table contains the addresses of sensitive system functions. Typically, a regular user-level process is not allowed to modify this table, while a legitimate installation of a system component (e.g., anti-virus software) may need to modify the system call table, thus the address space of the table needs to be updated. With traditional access control model such as DAC, any hijacked root process such as a rootkit can change the system call table. Existing MAC policies such as SELinux [16] and LOMAC [11] can prevent this but lack the capability to update object attributes (e.g., system call table function pointers) after a legitimate access. As another example, during a process's sensitive operation such as reading a sensitive data file, the process should be in a "good" running state such that whenever the process's integrity is changed, its access should be revoked immediately since a compromised process can maliciously release data to other entities. Access control models in existing operating systems only enforce integrity check before an access, while continuous security check during the access is not supported.

Aiming to overcome these problems and enhance the trustworthiness of security mechanism, we propose a novel framework with an access control model for policy specification and an enforcement architecture for kernel integrity protection. Specifically, a simple but effective usage control model (UCON) is used in our framework. The *decision continuity* and *attribute mutability* properties of UCON can provide flexible and fine-grained access control [17, 26]. To precisely specify kernel integrity policies, we develop an event-based logic model of UCON in this paper called UCON$_{\mathcal{KI}}$. In the enforcement architecture, our approach utilizes virtual machine monitor (VMM) technology to "vertically" control access to sensitive kernel objects in a single virtual machine (VM) running on top of the VMM (as opposed to "horizontal" control across multiple VMs running on the VMM). Access requests must go through the VMM layer. Our VMM-base approach maintains the lowest level accesses to the system and ensures that such accesses can not be compromised by internal processes of a VM. We implement a prototype system to show the feasibility of our framework. We demonstrate how our architecture and example policy can be used to detect and prevent real-world kernel rootkits. Beyond integrity protection in an OS kernel, we also explore extensions of our model and architecture to support more general security objectives for both OS and user space applications. Our extensions include extra control decision components such as conditions and obligations, authorization attribute management, and trusted policy enforcement.

The remainder of this paper is organized as follows. Section 2 provides the overview of our framework and threat assumptions. Section 3 presents the event-based logic model for UCON. Section 4 describes the proposed VMM-based architecture. Section 5 illustrates an implemented prototype according to our architecture and evaluates some experimental results. Extensions of our model and architecture are investigated in Section 6 for general OS security requirements. Section 7 presents related work on access control models and policy enforcement for OS protection. Section 8 summarizes this paper and presents our ongoing and future work.

2. FRAMEWORK OVERVIEW AND THREAT MODEL

2.1 System Framework Overview

The basic requirement of OS kernel integrity protection problem is to control accesses to sensitive kernel objects (e.g., the kernel text, the system call table, the interrupt descriptor table) in a real-time manner. That is, authorization decisions should be based on the real-time attribute values of requesting subjects and target objects. Furthermore, with the increasing complexity of security requirements, the control should be flexible and fine-grained, so as to reflect authorization changes with mutable security attributes. User identity and group membership are the main elements emphasized for authorization in previous work. However, in the OS kernel, running states of active processes and memory locations of sensitive objects are important authorization decision factors. For example, normal process cannot write to any place within the scope of system call table, while this address scope can be updated after authorized access.

Based on these requirements, an event-based logic model for usage control (UCON) is used in our framework, as it is attribute-based and has the unique properties of decision continuity and attribute mutability. In UCON, authorizations are predicates defined on subject and object attributes. A UCON policy can be specified in the following form: when an *Event* occurs, if some particular *Predicates* are true, then a set of *Actions* must be executed. An event can be an access request, or a subject/object attribute change. There are different types of actions, including allowing/denying an access request, revoking an ongoing access, and updating subject and/or object attribute values. Authorizations are enforced not only when a subject generates an access request event, but also during the entire ongoing stage of the accessing event, which is referred to as *decision continuity*. Subject and object attributes can be updated as the side-effect of usage; this is referred to as *attribute mutability*. Previous work has shown that decision continuity and attribute mutability can provide flexible and fine-grained access control [17, 26].

From enforcement point of view, a typical authorization system includes a policy decision point (PDP) and a policy enforcement point (PEP). In our enforcement architecture, both PDP and PEP are located at the VMM layer, as shown in Figure 1(b). For an access request event generated inside a VM, the PEP collects the subject and object attributes and submits them to the PDP along

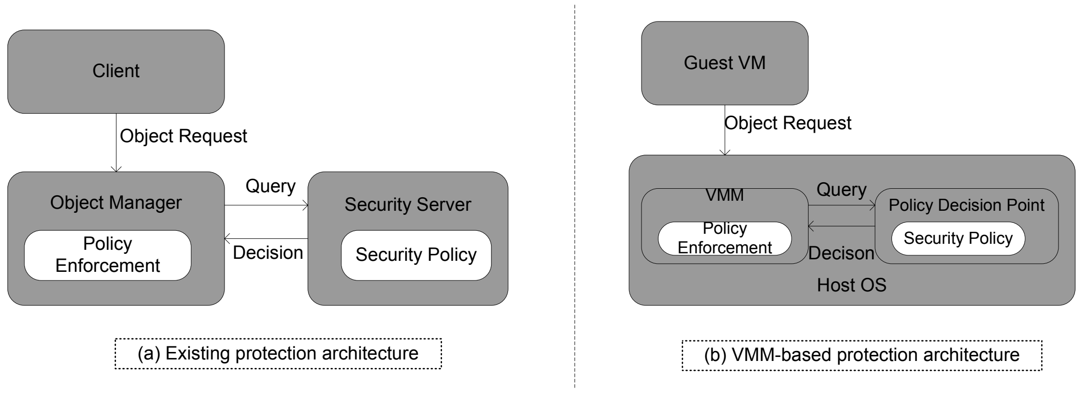


Figure 1: Existing protection architecture (within the same OS) vs. our VMM-based protection architecture

with the access request; The PDP makes the access control decision according to access control policies, which is forwarded back to the PEP and enforced by the PEP. For attribute acquisition, the PEP fetches the subject and object attributes from the VM's attribute repository. The objective is to protect kernel resources in a VM by utilizing the resource management capability of the underlying VMM.

The PEP updates the mutable subject and object attributes and reports to the attribute repository. An update of subject or object attribute triggers revaluation of the policy by the PDP according to an ongoing access event, and possibly revokes the access or updates attributes if necessary. This approach supports decision continuity and attribute mutability of UCON.

2.2 Threat Model

We assume that a large complex operating system can never be uncompromisable. Many architectures and frameworks have been proposed and deployed to enhance the security at the kernel level. As show in Figure 1(a), which is the typical architecture of existing protection mechanisms, the PEP and the security server (which is also called PDP) are located within the same space as kernel itself. The overall security relies on the correctness of some kernel portion to assure trustworthiness. A significant problem with this architecture is that, by exploiting a known or hidden vulnerability and gaining privileged permissions, sophisticated attackers can modify the OS kernel and disable or subvert the enforcement mechanism. For example, in a Linux system, a user may be able to modify the kernel by loading a kernel module or device driver [23, 24], accessing the special device file – /dev/kmem [3], or even launching DMA-based write operations from I/O devices.

As Figure 1(b) shows, security enforcement components in our architecture are located in the VMM. A VMM is a thin layer of software that emulates underlying hardware, and virtualizes all hardware resources in such a way that allows multiple virtual machines to run on top of it with efficient multiplexing [12]. VMM has traditionally been used for logical server partitioning, installation management, and cross-platform development and testing. Recently, virtual machines are widely used in servers as well as in desktop environments. We utilize the VMM to strictly isolate internal (untrusted) processes of virtual machines (VMs) so as to enforce security policies that aim to protect a VM's kernel integrity.

With the assumption of a trustworthy VMM, our architecture strongly isolates the PDP and PEP from the OS space of the protected VM running on top of the VMM, as shown in Figure 1(b). This gives the security enforcement components a high degree of attack resistance and allows them to continue to monitor and protect even if the VM has been corrupted.

3. USAGE CONTROL MODEL FOR KERNEL INTEGRITY PROTECTION

From a VM point of view, the VMM has ultimate control of the virtual hardware resources allocated to a VM. To achieve the integrity goal for the OS kernel in a VM, we need to confine accesses to sensitive system resources and critical kernel objects (e.g., the kernel text, the system call table, the interrupt descriptor table) according to pre-defined access control polices. The polices need to meet fine-grained, continuous, and mutable requirements.

We present a simple UCON model for OS kernel integrity, called UCON$_{\mathcal{KI}}$, which is a minimal form of UCON. Starting with a brief introduction of the concept of UCON$_{\mathcal{KI}}$ and its decision continuity and attribute mutability features, we present an event-based logic model to specify UCON$_{\mathcal{KI}}$ policies for OS kernel integrity protection.

3.1 UCON$_{\mathcal{KI}}$ Model for OS Integrity

3.1.1 Overview

DEFINITION 1. *A UCON$_{\mathcal{KI}}$ model has six components:*

- *Subjects (S): active processes and loadable kernel modules (LKMs);*
- *Objects (O): kernel memory spaces, disk devices, and registers;*
- *Subject attributes (ATT(S)): text hash values of subjects.*
- *Object attributes (ATT(O)): addresses, types, status of objects;*
- *Rights (R): generic actions such as read and write;*
- *Authorizations (A): functional predicates that have to be evaluated for usage decisions.*

Based on the above definition, UCON$_{\mathcal{KI}}$ only considers authorizations in the original UCON model [17, 26] (omitting UCON obligations and conditions). In a UCON$_{\mathcal{KI}}$ system, each entity (subject or object) is represented with a set of attributes. An event is an activity performed by a subject. An action is an activity performed by the security system. Predicates can be defined based on attributes as variables and constants. When an event occurs, an action is triggered if the corresponding predicates are evaluated to be true.

The virtual memory address of a kernel memory space is used as one of its attributes. We can also use the type of a kernel memory space as another attribute, such as the system call table, the kernel text, the interrupt descriptor table, and others. The text segment of each running process or a LKM is unique. So we can use the text's hash value of a subject as its attribute.

An authorization decision is determined by the requesting subject attributes, the target object attributes, and the requested right. An authorization check can be either performed before the requested right is exercised or while the right is exercised. For example, a process (subject) requests to write (right) to a kernel memory space (object). The policy is that the process's attribute (text hash value) is authenticated and the type (attribute) of the kernel memory can not be one of a restricted set (e.g., the system call table).

3.1.2 Continuity and Mutability

The most important properties of UCON$_{\mathcal{KI}}$ are continuity of usage decisions and mutability of subject and object attributes. Figure 2 shows a complete usage process consisting of three phases along time sequence: before usage, during usage, and after usage. In UCON$_{\mathcal{KI}}$, usage decision is not only evaluated before an access, but also during the usage process of an access. If any of the specified predicates does not hold during the usage process, the ongoing access is revoked by the system. That is, an access control in UCON$_{\mathcal{KI}}$ is not a one-time check and enforcement, but a continuous process. Mutability means that for a usage process, subject and/or object attributes can be updated as the side-effect of granting the access and processing the usage. Attribute updates can be performed before, during, or after a usage process. Due to the existence of concurrent accesses in a system, attribute updates may invoke cascading authorization checks, e.g., an attribute change in one usage involving a particular subject and object may affect another concurrent usage of the same subject or another subject's access to the same object.

Both decision continuity and attribute mutability concepts are based on a continuous ongoing accessing process. A *usage session* is defined as an accessing process initiated by a subject *s* to an object *o* with a generic right *r*, according to a UCON$_{\mathcal{KI}}$ policy. A usage session refers to a specific usage process with (s, o, r) that follows a particular policy, where the attribute predicates are evaluated based on s and/or o, and updates are performed on s's and/or o's attributes.

In UCON$_{\mathcal{KI}}$, continuous security check is event-based by attribute mutability: the updates of subject or object attributes. We give some example policies using our proposed policy description language in the next subsection.

3.2 Event-based Policy Model for UCON$_{\mathcal{KI}}$

Event Condition Action (ECA) languages are intuitive and powerful for policy description in programming reactive systems [4, 9, 15]. Originated from ECA, Event-Predicate-Action (EPA) is used in our framework for UCON$_{\mathcal{KI}}$. In a high level view, a policy is enforced with the following primitive activities: the VMM observes an *event* happening in a VM environment and reacts by performing corresponding *actions* to make an authorization decision and/or attribute updates if particular predicates are true.

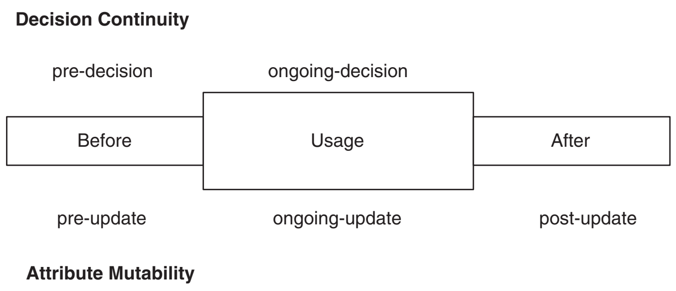


Figure 2: Continuity and mutability properties of UCON$_{\mathcal{KI}}$

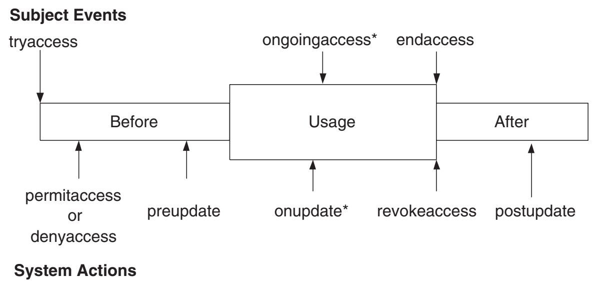


Figure 3: Subject events and system actions

A UCON$_{\mathcal{KI}}$ policy is defined as follows:

DEFINITION 2. *A UCON$_{\mathcal{KI}}$ policy is a well-typed policy rule of the form:*
$(e_1 \wedge ... \wedge e_i)$ ***causes*** $(act_1 \wedge ... \wedge act_j)$ ***if*** $(p_1 \wedge ... \wedge p_k)$
where e_1, ..., e_i *are events,* act_1, ..., act_j *are actions, and* p_1, ..., p_k *are predicates.*

A UCON$_{\mathcal{KI}}$ policy specifies that when events e_1, ..., e_i are happening, actions act_1, ..., act_j must be performed by the system if predicates p_1, ..., p_k are satisfied.

An event is an action that is being performed by an active subject. Figure 3 shows three events defined in our policy model: $tryaccess$, $ongoingaccess$ and $endaccess$. The semantics of these events are briefly explained below.

1. $tryaccess(s, o, r)$: subject s generates an access request to o with right r. For sake of simplicity, we refer (s,o,r) as an access request or an ongoing access when the context is clear.

2. $ongoingaccess(s, o, r)$: subject s executes an access to object o with right r. This event only happens when the access is in the ongoing usage phase.

3. $endaccess(s, o, r)$: subject s ends an access to object o with right r.

Note that $ongoingaccess(s, o, r)$ events can be repeated during a usage process, which may be periodically or triggered by other events or actions, e.g., other access requests or updates of subject/object attributes.

An action is performed by the security system. Figure 3 shows six different actions defined in our model, which are briefly explained below.

1. $permitaccess(s, o, r)$: grants an access request (s,o,r).

2. $denyaccess(s, o, r)$: rejects an access request (s,o,r).

3. $revokeaccess(s, o, r)$: revokes an ongoing access (s,o,r).

4. $preupdate(attribute)$: updates a subject or object attribute in the pre-usage phase.

5. $onupdate(attribute)$: updates a subject or object attribute during the ongoing usage phase.

6. $postupdate(attribute)$: updates a subject or object attribute in the post-usage phase.

Similar to $ongoingaccess(s, o, r)$, $onupdate(attribute)$ actions can be repeated periodically or triggered by other events or actions.

A predicate is a boolean expression built from variables and constants, including subject and object attributes. Similar to attributes, predicates are application-specific. Different types of predicates can be defined in a security system. For example, a unary predicate $p(a)$ is built with one attribute and any number of constants, where the attribute can be a subject or an object attribute; a binary predicate $p(a_1, a_2)$ is built with two attributes and any number of constants, where the attributes can be from a single entity or two different entities. Typically, a predicate can be defined with any number of attributes from a single entity, or two entities.

Now we use the EPA language to specify three example policies as follows.

Example 1 ***Pre-Authorization*** Consider a process (subject) that wants to write to a kernel memory space (object). The policy requires that the process's attribute (text hash value) is authenticated and the type (attribute) of the kernel memory cannot be one of the restricted set: the system call table (sys_call_table), the kernel text (k_text), and the interrupt descriptor table (idt). The policy is specified as follows:

$tryaccess(s, o, w)$ **causes** $permitaccess(s, o, w)$ **if** $\mathcal{H}(s) \in C \wedge o.type \notin \{sys\_call\_table, k\_text, idt\}$

where $\mathcal{H}(s)$ is the text's hash value of s and C is a set of approved text hash values. □

Example 2 ***Continuity*** Suppose a process (say s_1) is accessing the system call table of an OS kernel. When another process (say s_2) is accessing s_1 at the same time, s_1's integrity value is measured and the $revokeaccess(s_1, sys\_call\_table, r_1)$ action is triggered if the process's integrity value is changed. Here we use a process's text hash value as its integrity value. The policy is specified as follows:

$ongoingaccess(s_2, s_1, r_2) \wedge ongoingaccess(s_1, sys\_call\_table, r_1)$ **causes** $revokeaccess(s_1, sys\_call\_table, r_1)$ **if** $\mathcal{H}(s_1)' \neq \mathcal{H}(s_1)$

where $\mathcal{H}(s_1)$ and $\mathcal{H}(s_1)'$ are the hash values of s_1's text before and after the event, respectively. □

Example 3 ***Mutability*** A legitimate installation of anti-virus software modifies the system call table of an OS kernel. The system call table's address is updated after installation. The usage control policy is specified as follows:

$tryaccess(s, sys\_call\_table, write)$ **causes** $permitaccess(s, sys\_call\_table, write)$ **if** $p_1 \wedge ... \wedge p_i$

$endaccess(s, sys\_call\_table, write)$ **causes** $postupdate(sys\_call\_table.addr)$ **if** $true$

where s is the anti-virus software installation process. $p_1, ..., p_i$ are predicates that have to be satisfied in order to allow s to $write$ the system call table, and they are determined by other system or organization security policies. For example, a predicate can require that the anti-virus software must be signed by a trusted provider. The second rule states that after the access, the system call table's address (attribute $addr$) is updated. □

4. ARCHITECTURE

This section first introduces the overall VMM-based architecture of our framework, followed by the features to support attribute mutability and decision continuity. The goal of our architecture is to support the UCON$_{\mathcal{KI}}$ model introduced in previous section.

4.1 Architecture Overview

Figure 4 shows the architecture overview for VMM-based OS kernel integrity protection. Typically, the architecture includes three main components within the VMM: VM enforcer (VME), Attribute Repository (AR), and Policy Decision Point (PDP). VME is a module to intercept access request events from the guest VM, push/pull subject and object attributes, and enforce access executions. AR is used to store and push subject and object attributes. PDP delivers authorization decisions according to policies. To improve performance, the Access Vector Cache (AVC) component caches access decisions.

A usage session is initialized by a subject (e.g., a process from the guest VM) and works as follows. First the subject generates an access request event from the guest VM, and the request is intercepted by VME (step 1). After receiving the request, VME contacts AR and retrieves the subject attributes and object attributes (steps 2 and 3). Then VME queries AVC (step 4). If there is already a decision cached and the attributes of subject and object have not been changed since last access, the cached access decision is used (step 5) and the decision is forwarded to VME and enforced in the execution (step 8); otherwise VME pushes the request to PDP (step 6) along with the attributes. PDP makes the access control decision by evaluating polices with subject and object attributes (step 7). Finally, the decision is forwarded to VME and enforced in the VM execution environment (step 8).

As the side effect of making a usage decision, attribute updates are performed by VME according to the corresponding policy. VME gets the new attributes from the guest VM (step 9). New subject and object attributes are pushed back to AR (step 10). As a result, updated attributes can be shared between different usage sessions, and VME always checks AR for the latest attribute values when a new access request event is generated. Also after a usage session, VME pushes a decision cache to AVC (step 11).

4.2 Attribute Acquisition and Management

As UCON$_{\mathcal{KI}}$ is attribute-based, a critical requirement to correctly enforce access control policies is to get real-time attribute values when PDP makes authorization decisions. The VMM can observe low-level system states (e.g., memory pages and machine registers). By transparently inspecting a VM's internal system state, we can accurately derive the processes, kernel-level object (e.g., the system call table), and their attributes (text hash values of the running processes).

We assume that all the internal processes of a VM are untrusted. Typically, when a subject generates an access request, subject and object attributes are pulled from VM by VME, e.g., after integrity measurements. VME queries the values stored in AR to check whether they have been changed or not. When an update is performed, the new attribute value is pushed by VME to AR.

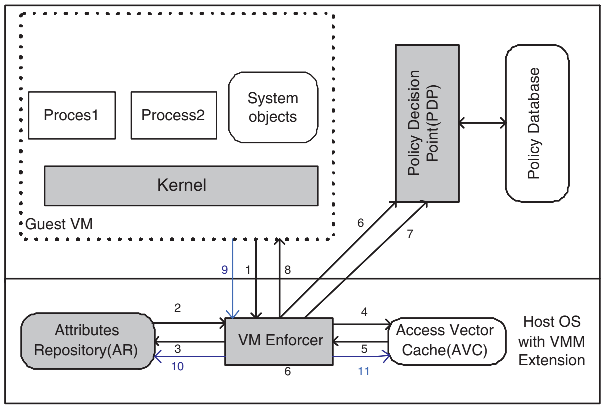


Figure 4: Our VMM-based OS Kernel Integrity Protection Architecture

4.3 Mutability and Continuity

4.3.1 Updates of Attributes

UCON$_{\mathcal{KI}}$ includes pre-updates, ongoing-updates and post-updates. As ongoing-updates are results of decision continuity, they are discussed in the following subsection.

According to our architecture, pre-updates are performed by VME when the access is granted according to a policy. As a policy decision is generated by PDP, attribute update requests are forwarded by PDP to VME along with the authorization decision. Updated attribute values are reported to AR by VME.

Post-updates can be triggered by a subject's *endaccess* event or the *revokeaccess* action of an ongoing usage session by the system. For the first type, VME reports an *endaccess* event after a subject ends the usage. VME performs the updates and the new attribute values to AR. For the second case, as the revocation of an ongoing access is the result of the decision continuity (described shortly) enforced by VME, the post-updates are performed by VME and reported to AR.

4.3.2 Continuous Enforcement of Policies

During a usage session, the policy is checked and the decision is enforced repeatedly if there are ongoing decision components in this policy. Since a decision component is built with attribute predicates, ongoing checks, in practice, are triggered by attribute changes during a usage session (e.g., the updates of mutable subject and object attributes as discussed above).

In our architecture, upon successful update of a subject or an object attribute, ongoing usage sessions involving the subject and the object are re-evaluated by querying PDP. When the predicates of the policy are still satisfied, the ongoing access is allowed to continue; otherwise, a *revokeaccess* action is generated by PDP and the decision is enforced by VME. As a UCON$_{\mathcal{KI}}$ policy associates exactly one subject and one object, an update event is involved with an ongoing usage session (s,o,r) only if the event updates an attribute of s or o.

5. IMPLEMENTATION AND EVALUATION

We have implemented a proof-of-concept prototype of the proposed architecture. In order to validate its practicality and effectiveness, we subject the prototype to a variety of kernel rootkit attacks that compromise OS kernel integrity. So far, we have tested with 18 real-world kernel rootkits that are publicly available and the experimental results are encouraging: our system is able to successfully detect and prevent all kernel integrity violations from these rootkits.

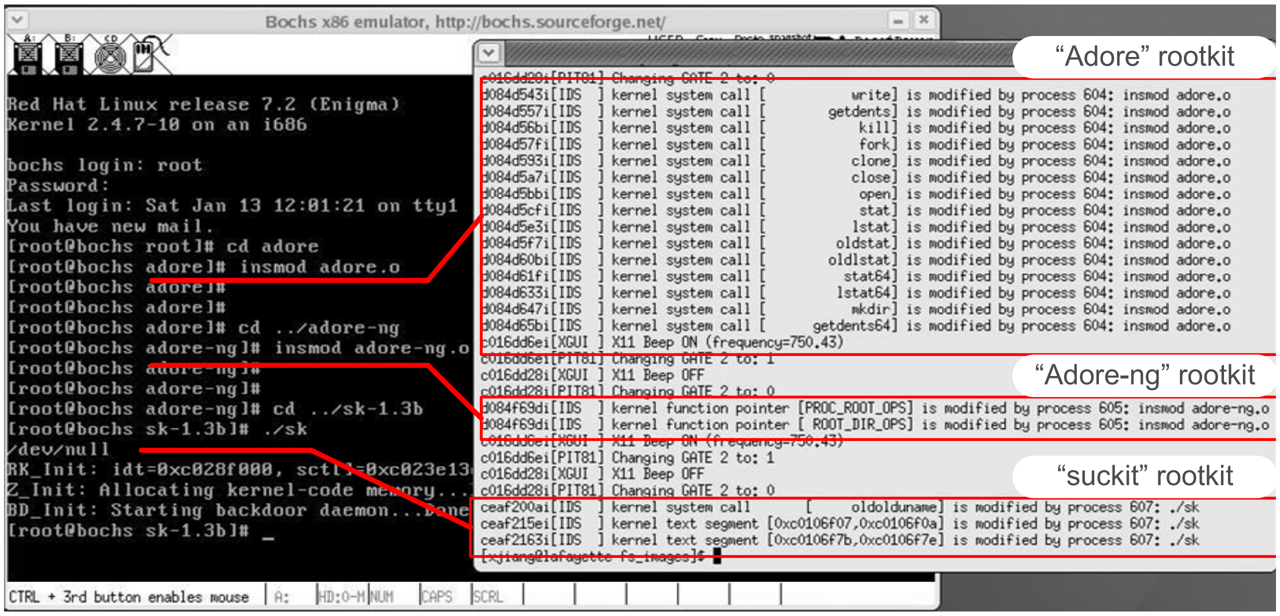
5.1 Prototype Implementation

The prototype is implemented by leveraging and extending an open-source emulator-based VMM platform called Bochs [1]. With the availability of its source code, we can conveniently prototype our system to validate the feasibility of the proposed approach. However, we point out that our approach is general and can also be applied to other VMM platforms, such as VMware [2] and Xen [5]. In the following, we describe three main extensions we made to the original Bochs system:

(1) From the VMM, we transparently inspect a VM's internal system state and accurately derive various internal model-related entities such as processes, kernel-level objects (e.g., the system call table), and their attributes (e.g., hash values of the running processes). Some entities are straightforward to identify. For example, the location of the system call table (*sys_call_table*) might be exported by the kernel and we can directly find its location from the */boot/System.map* file. However, there also exist some entities (e.g., running processes and the related hash values) that require additional efforts to obtain. The main reason is that the original VMM only observes low-level system states (e.g., memory pages and machine registers), which, unfortunately, are not straightforward to derive model-friendly entities with high-level semantic-rich information. To address that, we instantiate the general methodology known as "virtual machine introspection" (VMI) [12, 13] at the VMM layer. With the knowledge of the VM system internals, VMI aims to interpret the runtime VM state and infer internal entities (e.g., processes and files). For instance, our current prototype is able to extract all interested model-related information, including the entities such as internal processes and specific kernel-level objects as well as their attributes. Note that different types of policies may involve different entities and our current prototype mainly focuses on the integrity protection of important kernel objects, i.e., the system call table, the interrupt descriptor table, the kernel text segment, and various virtual file system (VFS)-related data structures.

(2) The second extension is to dynamically generate model-related events from the detected internal VM actions at the VMM level. Particularly, our current prototype continuously monitors those kernel-level memory-writing instructions and checks whether they are attempting to modify protected kernel objects. Note that each detected attempt leads to generating a model-related event. As an example, when an internal process is attempting to modify the system call table, an event is triggered to execute our policy enforcement engine. Based on the specified policy, our enforcement engine then validates whether the modification should be allowed or not.

(3) The third extension is the real-time capability of the VMM to enforce specified policies so that the runtime OS kernel integrity can be protected. For example, suppose one policy stipulates that the kernel text, the system call table, the interrupt descriptor table, and the jump table of the virtual file system should not be modified after the system boots up. Any tampering attempts are captured and the subsequent writing operations are denied. In our current prototype, we utilize Bochs to intercept every memory writing instruction to ensure that its execution does not violate this policy. Note that the instruction-level interception will introduce significant performance slowdown and it can be improved by leveraging certain hardware features (e.g., read-only memory pages) [12]. Figure 5 shows a number of key kernel-level objects (and their runtime addresses in a running RedHat 7.2) that are protected by our system.



```
Red Hat Linux release 7.2 (Enigma)
Kernel 2.4.7-10 on an i686

bochs login: root
Password:
Last login: Sat Jan 13 12:01:21 on tty1
You have new mail.
[root@bochs root]# cd adore
[root@bochs adore]# insmod adore.o
[root@bochs adore]#
[root@bochs adore]#
[root@bochs adore]# cd ../adore-ng
[root@bochs adore-ng]# insmod adore-ng.o
[root@bochs adore-ng]#
[root@bochs adore-ng]#
[root@bochs adore-ng]# cd ../sk-1.3b
[root@bochs sk-1.3b]# ./sk
/dev/null
RK_Init: idt=0xc028f000, sctl=0xc023e13
Z_Init: Allocating kernel-code memory...
BD_Init: Starting backdoor daemon...Done
[root@bochs sk-1.3b]# _
```

```
c016dd28i[PIT81] Changing GATE 2 to: 0
d084d543i[IDS  ] kernel system call [         write] is modified by process 604: insmod adore.o
d084d557i[IDS  ] kernel system call [      getdents] is modified by process 604: insmod adore.o
d084d56bi[IDS  ] kernel system call [          kill] is modified by process 604: insmod adore.o
d084d57fi[IDS  ] kernel system call [          fork] is modified by process 604: insmod adore.o
d084d593i[IDS  ] kernel system call [         clone] is modified by process 604: insmod adore.o
d084d5a7i[IDS  ] kernel system call [         close] is modified by process 604: insmod adore.o
d084d5bbi[IDS  ] kernel system call [          open] is modified by process 604: insmod adore.o
d084d5cfi[IDS  ] kernel system call [          stat] is modified by process 604: insmod adore.o
d084d5e3i[IDS  ] kernel system call [         lstat] is modified by process 604: insmod adore.o
d084d5f7i[IDS  ] kernel system call [       oldstat] is modified by process 604: insmod adore.o
d084d60bi[IDS  ] kernel system call [      oldlstat] is modified by process 604: insmod adore.o
d084d61fi[IDS  ] kernel system call [        stat64] is modified by process 604: insmod adore.o
d084d633i[IDS  ] kernel system call [       lstat64] is modified by process 604: insmod adore.o
d084d647i[IDS  ] kernel system call [         mkdir] is modified by process 604: insmod adore.o
d084d65bi[IDS  ] kernel system call [     getdents64] is modified by process 604: insmod adore.o
c016dd6ei[XGUI ] X11 Beep ON (frequency=750.43)
c016dd6ei[PIT81] Changing GATE 2 to: 1
c016dd28i[XGUI ] X11 Beep OFF
c016dd28i[PIT81] Changing GATE 2 to: 0
d084f69di[IDS  ] kernel function pointer [PROC_ROOT_OPS] is modified by process 605: insmod adore-ng.o
d084f69di[IDS  ] kernel function pointer [ ROOT_DIR_OPS] is modified by process 605: insmod adore-ng.o
c016dd6ei[XGUI ] X11 Beep ON (frequency=750.43)
c016dd6ei[PIT81] Changing GATE 2 to: 1
c016dd28i[XGUI ] X11 Beep OFF
c016dd28i[PIT81] Changing GATE 2 to: 0
ceaf200ai[IDS  ] kernel system call       [     oldolduname] is modified by process 607: ./sk
ceaf215ei[IDS  ] kernel text segment [0xc0106f07,0xc0106f0a] is modified by process 607: ./sk
ceaf2163i[IDS  ] kernel text segment [0xc0106f7b,0xc0106f7e] is modified by process 607: ./sk
[xjiang@lafayette fs_images]$
```

```
CTRL + 3rd button enables mouse | A: | HD:0-M | NUM | CAPS | SCRL
```

Figure 6: Detecting and preventing OS kernel integrity violations

.

5.2 Evaluation

We have evaluated our system with 18 real-world kernel-level rootkits (Table 1) that are publicly available. Note that all of these rootkits can successfully compromise the OS kernel integrity on vulnerable systems.

Rootkit	Target	Attacker Vector	Prevented
adore 0.53	system call table	LKM	√
knark	system call table	LKM	√
phide	system call table	LKM	√
rial	system call table	LKM	√
rkit 1.01	system call table	LKM	√
synapsys	system call table	LKM	√
override	system call table	LKM	√
maxty	system call table	LKM	√
kbdv3	system call table	LKM	√
all-root	system call table	LKM	√
suckit	kernel text	Raw memory access	√
suckit2priv	kernel text	Raw memory access	√
enyelkm	kernel text	LKM	√
phantasmagoria	kernel text	LKM	√
superkit	kernel text	Raw memory access	√
Phalanx	system call table	Raw memory access	√
mood-nt	system call table	Raw memory access	√
adore-ng	virtual file system	LKM	√

Table 1: 18 **Linux kernel rootkits used for our evaluation**

Among these rootkits, we choose three representative ones for detailed examinations: adore [23], adore-ng [24], and suckit [21]. The detailed examinations are helpful to understand how our system is able to effectively detect and prevent kernel integrity violations. As shown in Table 1, the adore rootkit makes use of the loadable kernel module (LKM) support in commodity Linux kernels to directly hijack a number of system call table entries. The adore-ng rootkit uses the same LKM support to subtly modify the jump table of the virtual file system. Note that the jump table contains a number of function pointers that are indirectly invoked when system monitoring tools such as *ps* and *ls* are executed to list files in the current directory or running processes. Once these function pointers are hijacked, certain files and processes can be chosen by attackers to be "invisible" from these system monitoring tools. As such, though the system call table is not modified, the same "hiding" goal can still be achieved. The third chosen rootkit – suckit takes advantage of the writable special device file – */dev/kmem* to directly modify OS kernel instructions. These instructions are craftily chosen to control the kernel-level execution flows so that system call results can be manipulated.

Figure 6 shows a screenshot of how our system detects the integrity violation attempts from these three representative kernel rootkits: The *left* xterm screen (with black background color) shows the inside of the VM while the *right* xterm screen (with white background color) shows the reported alerts from our system. Our system is able to capture all kernel-level write instructions and further infer the kernel objects that are being attacked. In the following, we describe detailed attacks from these three rootkits.

- The *adore* rootkit (executed by the command *insmod adore.o* as shown in the left window) hijacks 15 entries in the system call table and replaces them with its own implementations. A customized user-space program called *ava* can be used to send hiding instructions to the kernel-level attack code so that certain files or processes of attackers' choices can be hidden. As shown in the right window of Figure 6, when the command is executed to load the kernel module *adore.o*, our system immediately reports an alert with details showing what are those system call entries being modified. Moreover, our system further examines the VM state to identify the corresponding process ID as well as the responsible user command triggering these violation attempts. Such trace-back feature is desirable when there is a need to hold them accountable for launching the rootkit attack.

- The *adore-ng* rootkit subverts the jump table of the "Virtual

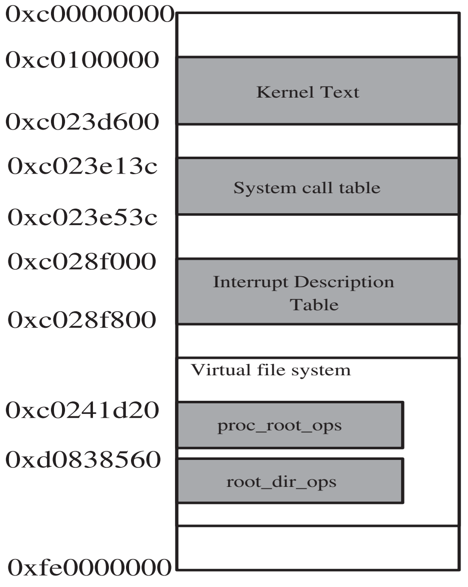


Figure 5: Protected key kernel objects that are commonly attacked by real-world kernel rootkits. Their runtime addresses are collected from a RedHat 7.2 Linux system (with kernel version 2.4.7-10).

File System" by replacing the directory listing handler routines with its own ones. Such replacement allows it to manipulate the information about the *root* file system as well as the */proc* pseudo-file system to achieve the file-hiding or process-hiding purposes. As shown in Figure 6, once the *adore-ng.o* module is loaded, our system reports what are those kernel objects being compromised and which process is the culprit process (i.e., the process with PID 605: *insmod adore-ng.o*).

- The *suckit* rootkit installs itself through */dev/kmem* interface to subvert the OS kernel where the LKM support might not be available. As reported from our system, the *suckit* not only modifies one system call entry – *oldolduname*, but also overwrites two instructions inside the kernel text. Further analysis shows that the *oldolduname* system call entry is hijacked to allocate a kernel memory space to store its own kernel-level attack code. The two instructions are overwritten to hijack the system call flow, which can be later misused to hide attack files or processes.

6. EXTENSIONS AND DISCUSSIONS

Our current model and the prototype system focus on OS kernel integrity protection. In this section, we explore other aspects of OS security that are beyond kernel integrity and examine possible extensions of UCON$_{\mathcal{KI}}$ and policy enforcement architecture for these purposes.

6.1 OS Security Requirements

To provide a high quality OS security protection, access control techniques must be able to support various real-word security requirements. From the model point of view, the access control model needs to support flexible and application-specific polices such as fine-grained least privilege and dynamic separation of duty. We explore possible extensions of UCON$_{\mathcal{KI}}$ to support flexible security policies in next subsection. From policy enforcement point of view, the architecture must ensure that performance overhead of the usage control is minimal and the OS must operate transparently to applications and users except when access failures occur. We discuss how to extend our VMM-based architecture to meet these requirements at the end of this section.

6.2 UCON$_{\mathcal{KI}}$ Extensions

6.2.1 Attributes Management

We can specify most OS security policy using UCON$_{\mathcal{KI}}$ by introducing additional subject and object attributes. Flexible policies can be defined, such as role-based access control (RBAC), dynamic separation of duty, Chinese Wall, and polices with low or high water mark properties, which have been conceptually and formally studied in previous work [17, 26]. In an OS, an object can be a file, a running process, a library, a kernel memory space, a disk device, or a register on which particular actions (requiring appropriate rights) can be performed by an active subject. A subject can be an active process, or a LKM that can generate access requests. Besides the text hash value, subject attributes can also include role, group, and domain name. Additionally, application-specific attributes can be defined depending on the target object. Specifically, when there is a system resource that can only be used by a single process at a time, a subject attribute specifying the conflict-of-interest groups could be defined. Object attributes include address, type, ownership, content, inclusive/exclusive accesses, access history and status, etc. For example, we can express Chinese Wall Policy using UCON$_{\mathcal{KI}}$ as follows.

Example 4 ***Chinese Wall Policy*** Consider a system with a set of conflict object classes $C = c_1, c_2..., c_n$. An object attribute *class* indicates which class it belongs to. A subject attribute is defined as $ac = c_{s_1}, c_{s_2}, ..., c_{s_m}$, where $s_1, s_2, ...s_m$ are integers from 1 to n, to record the classes that a subject has accessed. Another subject attribute is $ao = o_1, o_2, ...o_k$, which records the objects that the subject has accessed. If a subject has accessed an object, the Chinese Wall policy is:

$tryaccess(s, o, r)$ **causes** $permitaccess(s, o, r)$ **if** $o \in s.ao$

For an access request for an object that is not in the subject's ao, the policy is defined as the following:

$tryaccess(s, o, read)$ **causes** $permitaccess(s, o, read)$ **if**
$(o \notin s.ao) \wedge (o.class \notin s.ac))$
$permitaccess(s, o, read)$ **causes** $preupdate(s.ac)$
$\wedge\, preupdate(s.ao)$ **if** $true$
$preupdate(s.ac) : s.ac' = s.ac \cup o.class$
$preupdate(s.ao) : s.ao' = s.ao \cup o$ □

6.2.2 Conditions

In a modern OS, a subject's permission depends on the state of a system. To address this requirement, we need to add conditions as decision components in UCON$_{\mathcal{KI}}$. As conditions are environmental restrictions, system attributes are introduced. Specifically, a condition is a predicate built on system attributes to specify the restriction that has to be satisfied before or during a usage process. Although system attributes are not updated in UCON, they change due to dynamic system environment. For example, the system attribute *status* changes from *boot* to *normal* till *shutdown*. As a result, this change may affect the decision of a subsequent access request or an ongoing access.

6.2.3 *Obligations*

In many cases, in order to grant a subject's requested access, an action, which is not like any action or event defined in our UCON$_{\mathcal{KI}}$ model, need to be performed by the subject or another subject. We need to introduce the obligation component to UCON$_{\mathcal{KI}}$. An obligation is an action that must be performed by a subject before or during an access. For example, when an anti-virus application wants to update virus data, an obligation is defined that before the update, it has to check the availability of the latest virus definition by opening a dedicated network port. Without this action, the update request is denied.

6.3 Policy Enforcement

Currently, our VMM-based architecture has the capability to enforce policies for kernel integrity in a VM. To support user level policy enforcement, we need to extend this architecture with non-trivial extra capability of VMM. One limitation of our prototype is its inability of intercepting user-level semantic information of a guest OS. In order to enforce the user level security policies, one way is to develop tools for VMM to minimize the semantic gap.

Another possible solution is to insert a "trusted" kernel module into the OS kernel in the VM. The trusted module redirects the user level access requests to the VME in our architecture and enforces usage decisions. Our current prototype can be extended to not only ensure the kernel integrity, but also protect this particular security module. As part of our future work, we will investigate possible security risks and examine its feasibility.

7. RELATED WORK

A logic policy model of UCON has been proposed in previous work [26, 27]. In this model, activities performed by subjects and security system are called "actions" without distinctions. In this paper, we consider the activities performed by a subject as "events" and the reactive activities performed by a security system as "actions", which is more intuitive and precise. Another difference in our approach is that we use event-predicate-action (EPA) in our logic specification, and we focus on a higher level access control policies, while previous model focuses on atomic actions and temporal properties during a single usage process.

Traditional MAC polices, such as the Bell-LaPadula secrecy policy [6] and the Biba integrity policy [7], define clear security goals, but are too restrictive for convenient use of applications. These approaches are very coarse-grained for least privilege requirements. A different form of MAC known as Type Enforcement (TE) [8] offers several advantages over the traditional models. In TE, security policies can be easily customized, and users and individual programs can be easily limited to least privilege through the definition of domains and domain transitions. However, TE also has its limitations. Since the security policy logic is defined through labels and there are no implicit relationships among labels, it would be cumbersome to express a complex Bell-LaPadula or BiBa lattice using TE. TE also does not directly address dynamic security policy requirements, which are often needed in real-world environments. UCON can provide flexible, fine-grained, and dynamic access control. Additional comparison details can be found in previous work [17, 26].

The policy enforcement mechanisms in the systems, like SELinux [16], Systrace [19] and Nooks [25], reside in the same space as the running OS. Once an intruder takes control of the machine, the protection mechanism can be potentially disabled or subverted. An advantage of our mechanism over these systems is that an attacker can not comprise our protection mechanism from a guest VM.

Co-processor based systems, such as the system introduced by Zhang et al. [28] and Copilot [14, 18], monitor operating system by polling kernel memory periodically. Since they use separate co-processor such as PCI card, and are independently connected to monitor stations, they can reliably detect intrusions and cannot easily be compromised by an attacker with root privilege. While these approaches can perform very efficient detection, they cannot perform any prevention as they cannot interpose the host's executions. Also the view of the monitor is limited to memory. Our VMM based approach not only can detect intrusions but prevent malicious activities by intercepting events in real time. Note that our architecture is derived from a general methodology known as "virtual machine introspection" (VMI) [12]. We point out that VMI is mainly used for intrusion detection purposes, while our architecture is mainly for access control policy enforcement.

The Secure Hypervisor (sHype) [20] project aims to support controlled sharing of resources between VMs on a platform, such as memory, CPU cycles, and network bandwidth. While sHype protects sharing of resources between VMs in a "horizontal" way, our architecture "vertically" controls access to sensitive kernel resources in a single VM running on top of the VMM.

8. CONCLUSION AND FUTURE WORK

We present a security framework to protect OS kernel integrity which includes a policy model and an enforcement architecture. To meet flexible and fine-grained authorization requirements, a simple UCON$_{\mathcal{KI}}$ model is proposed. We define an event-predicate-action logic model to specify policies for kernel integrity purposes. Our proposed architecture can support attribute mutuality and decision continuity by leveraging the capabilities of strong isolation and resource monitoring of VMMs. We implement a proof-of-concept prototype to demonstrate the effectiveness of our approach. Experiments show that our system can successfully detect and prevent a variety of real-world rootkit attacks.

Based on this framework, we are extending our model and architecture to support general OS security objectives, including the protection of OS resources and services, and user level applications. With the strong expressive power of UCON and increasing capabilities of VMMs, our goal is to build a general framework for OS security.

9. REFERENCES

[1] Bochs ia-32 emulator project, http://bochs.sourceforge.net.

[2] VMware. *http://www.vmware.com/*.

[3] Linux on-the-fly kernel patching without lkm. Phrack Magazine, 2001.

[4] J. J. Alfres, F. Banti, and A. Brogi. An event-condition-action logic programming language. In *CILC 2006, Convegno Italiano di Logica Computazionale*, Bari, Italy, 2006.

[5] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, R. Neugebauer A. Ho, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. *Proc. of the 19th ACM Symposium on Operating Systems Principles*, October 2003.

[6] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations and model. *Mitre Corp. Report No.M74-244, Bedford, Massachusetts*, 1975.

[7] K. J. Biba. Integrity consideration for secure computer system. Technical report, Mitre Corp. Report TR-3153, Bedford, Mass., 1977.

[8] W. E. Boebert and R. Y. Kain. A practical alternative to hierarchical integrity policies. In *Proceedings of the 8th National Computer Security Conference*, 1985.

[9] J. Chomicki and J.Lobo. Monitors for history-based policies. In *2nd International Workshop on Polices for Distributed System and Networks*, Bristol, U.K., 2001.

[10] Department of Defense National Computer Security Center. *Department of Defense Trusted Computer Systems Evaluation Criteria*, December 1985. DoD 5200.28-STD.

[11] T. Fraser. Lomac: Low water-mark integrity protection for cots environment. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, California, 2000. IEEE.

[12] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture of intrusion detection. In *Network and Distributed System Security Symposium*, San Diego, California, 2003.

[13] Ashlesha Joshi, Samuel T. King, George W. Dunlap, and Peter M. Chen. Detecting Past and Present Intrusions through Vulnerability-specific Predicates. *Proc. of the 2005 Symposium on Operating Systems Principles (SOSP)*, October 2005.

[14] Nick L. Petrono Jr., Timothy Fraser, Jesus Molina, and William A. Arbaugh. Copilot-a coprocessor-based kernel runtime integrity monitor. In *the 13th USENIX Security Symposium*, San Diego, California, 2004.

[15] J. Lobo, R. Bhatia, and S. Naqvi. A policy description language. In *Proc. of AAAI*, Orlando, FL, July 1999.

[16] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the linux operating system. In *USENIX Annual Technical Conference, June 25-30*, Boston, Massachusetts, 2001.

[17] J. Park and R. Sandhu. The UCON$_{\mathrm{abc}}$ usage control model. *ACM Transactions on Information and Systems Security*, 7(1), February 2004.

[18] N. L. Petroni, T. Fraser, A. Walters, and W. A. Arbaugh. An Architecture for Specification-Based Detection of Semantic Integrity Violations in Kernel Dynamic Data. In *Proceedings for the 15th USENIX Security Symposium*, Vancouver, B.C., Canada, July 2006.

[19] Niels Provos. Improving host security with system call policies. Technical report, Center for Information Technology Integration, University of Michigan, 2002.

[20] Reiner Sailer, Trent Jaeger, Enriquillo Valdez, Ramon Caceres, Ronald Perez, Stefan Berger, John Linwood Griffin, and Leendert van Doorn. Building a mac-based security architecture for the xen open-source hypervisor. In *Proceedings of the 2005 Annual Computer Security Applications Conference*, pages 276–285, December 2005.

[21] sd. Linux on-the-fly kernel patching without LKM. *Phrack, 11(58):article 7 of 15*, December 2001.

[22] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Anderson, and J. Lepreau. The flask security architecture: System support for diverse security policies. In *Proceedings of the Eighth USENIX Security Symposium*, pages 123–139, 1999.

[23] Stealth. adore. http://spider.scorpions.net/stealth.

[24] Stealth. adore-ng. http://stealth.7350.org/rootkits/.

[25] M. Swift, B. Bersahd, and H. Levy. Improving the reliability of commodity operating system. *ACM Trans. Computer Systems*, 23:77–110, 2005.

[26] X. Zhang, F. Parisi-Presicce, R. Sandhu, and J. Park. Formal model and policy specification of usage control. *ACM Transactions on Information and Systems Security*, 8(4), 2005.

[27] X. Zhang, J. Park, F. Parisi-Presicce, and R. Sandhu. A logical specification for usage control. In *the 9th ACM Symp. on Access Control Models and Tech.*, Yorktown Heights, USA, 2004.

[28] X. Zhang, L. van Doon, T. Jaeger, R. Perez, and R. Sailer. Secure coprocessor-based intrusion detection. In *the Tenth ACM SIGOPS European Workshop*, Saint Emilion, France, September 2002.