Centralized Security Policy Support for Virtual Machine

Nguyen Anh Quynh, Ruo Ando, and Yoshiyasu Takefuji – Keio University

ABSTRACT

For decades, researchers have pointed out that Mandatory Access Control (MAC) is an effective method to protect computer systems from being misused. Unfortunately, MAC is still not widely deployed because of its complexity. The problem is even worse in a virtual machine environment, because the current architecture is not designed to support MAC in a site-wide manner: machines with multiple virtual hosts needs to have multiple MAC security policies, and each of these policies must be updated and managed separately inside each virtual host.

In order to ease the burden on administrators when deploying security policies in a virtual environment, this paper proposes an architecture named *Virtual Mandatory Access Control* (*VMAC*) to centralize security policies, so that all policy management can easily be done from a central machine. VMAC securely centralizes the security logging information from all virtual hosts into a central machine so intrusion detection analysis on the logging data is straightforward.

To arrive at the architecture presented here, we have investigated various popular MAC schemes, and implemented several schemes with VMAC on the Xen Virtual Machine. This paper presents our experiences in the development process.

Introduction

In today's world, in which almost everything relies on a computer to work, information assurance becomes very important. Computer security is primarily done through the enforcement of security policy, which sets a context within which the notion of a secure system can be defined. If we consider a system to be a finite-state automaton with a set of states, a security policy is a statement that partitions the states into secure and non-secure states.

Basically, we can classify security polices into two classes. The first class, *discretionary security policy*, consists of any security policy which ordinary users can define, alter or assign. The second class, *mandatory security policy* is different: this kind of policy must be defined and controlled tightly by the system administrators, enforced at the operating system level.

Correspondingly, a security policy may use two types of access control: *Discretionary Access Control* (or *DAC* in short) or *Mandatory Access Control* (or *MAC* in short). It is also possible to combine them together, as usually seen in current practice.

- With DAC, a user can set an access control mechanism to allow or deny others access to a system object he owns. DAC bases access rights on the identity of the subject and the identity of the object involved, in which the owner of the object can specify which subjects can access the object, with particular rights.
- In contrast with DAC, MAC does not rely on the identity of the subject: when the access control is set on an object (typically by the system administrator), the owner of that object cannot modify the access right. In this case, the OS is responsible for enforcing the MAC, and inspects information associated with both the subject and object to determine whether to give the access right to the subject.

In the real world, DAC is widely used in all modern operating systems because it is very simple, easy to use, and easy to manage. However, DAC is the root of most security issues today. The reason is that with DAC, it is impossible to protect the system against malicious code (such as hostile mobile code or Trojan horses) because such code gains the security permissions of the user executing them. Exploiting these legitimate rights, the malware gains unauthorized access to the user's files and resources, and can then compromise their confidentiality and integrity. Even worse, it can change the DAC policy to abuse the system without the user's knowledge. This causes an arms race between anti-malware researchers and the bad guys, and the problem becomes more and more serious as time goes on [1, 2].

To address the problems of DAC, it is necessary to eliminate them and employ MAC instead [3]. In fact, for decades various researchers have demonstrated that mandatory security is essential for security of computer systems, so MAC should be used to enforce security policy instead of DAC [4]. With MAC, a malicious user (or malicious code running on behalf of a legitimate user) cannot misuse resources for a purpose other than what the system specifies. Because MAC can only be changed by the system

administrator, harmful code cannot modify the policy without approval from the system.

To realize the MAC capability, various schemes and implementations have been introduced in all kinds of operating systems, with the hope that all systems would eventually switch from DAC to MAC. Unfortunately, while MAC might efficiently stop or mitigate security damages, many proposed schemes proved hard to deploy and complicated to manage. MAC often becomes so painful that administrators choose to disable MAC and return to DAC instead, despite the fact that MAC would help to secure their machines [5]. Currently, researchers are working on making MAC easier to use and more widely accepted.

Recently virtual machine (VM) technology has emerged as one of the hottest topics in computer research. The principle of VM technology is to allow the creation of many virtual hosts, each running an instance of an operating system, all running at the same time on the same physical machine. Obviously VM technology can help to reduce both hardware and maintenance costs for organizations that need to use various machines for different services.

However, systems with virtual hosts face a major problem when deploying MAC: current schemes do not address MAC within a VM environment. A machine with multiple virtual hosts needs to have multiple security policies, and each of these policies must be managed separately inside each virtual host. Whenever the administrator wants to update the security policy of a specific host, he must login to that host and carries out the job there. As a result, managing machines with hundreds or more virtual hosts is very time-consuming and complicated, especially if the hosts employ different policies. Clearly, we need more efficient and flexible methods to solve this problem.

In an attempt to address this difficulty, this paper proposes a novel architecture named Virtual Mandatory Access Control (*VMAC*) for virtual hosts: instead of having security policies inside of each host, VMAC moves the security policies outside of the VMs, and puts them into a central machine. This strategy allows the administrator to manage the security policies of all the virtual hosts from an administrative virtual host. As a result, updating the MAC for virtual hosts becomes much easier, because it can be done from a central place. The architecture also gathers the security logging data from virtual hosts and saves them into the central machine, so the intrusion detection process based on security violation logging can also be done there. Obviously, this scheme makes it easier to correlate and manage intrusion evidence. To prove the concept of VMAC, we have implemented it on the Xen Virtual Machine [6, 7, 8], with three selected MAC schemes: AppArmor, LIDS and Trustees.

The rest of this paper comprises five sections: a summary of related works, the VMAC architecture, and implementation of VMAC on Xen, experiences employing selected MAC schemes, a few concerns with the proposed solution, and a conclusion with future plans.

Related Works

Our work is a combination of two ideas: MAC and operating system virtualization. The original meaning of an operating system virtual machine, also called a hardware virtual machine, is that of a number of different identical execution environments executing on a single computer. One of the most popular uses of virtual machines is to allow a user to run multiple full-featured operating systems at the same time on one physical machine. The host software providing this ability is often referred to as a virtual machine monitor (VMM) or hypervisor.

Recently, virtualization has emerged as one of the hottest research topics, garnering much attention from academia and industrial sectors. Various VM software systems are available, ranging from commercial solutions such as VMWare or Microsoft Virtual PC to free solutions like UML, OpenVZ, VServer and Xen.

Among these technologies, the Xen Virtual Machine Monitor is one of the most interesting solutions. Basically, Xen is a thin layer of software above the bare hardware. Xen exposes a VM abstraction that is only slightly different from the underlying hardware. Xen introduces a new architecture called *xen*, which is very similar to the x86 architecture. The VMs executing on Xen are modified (at the kernel level) to work with the *xen* architecture. With this Xen-aware kernel, the performance impact of the virtual hosts is amazingly low: only around 3% in some published experiments [9]. All accesses of virtual hosts to the hardware and peripherals must go through Xen, so that Xen can keep a close eye on those VMs and control all activities.

Formerly developed by researchers from Cambridge University, currently Xen is backed by various industrial players such as IBM, Intel, AMD, HP, RedHat and Novell. The whole Xen community is working very hard to push the code into the Linux kernel, so that it will be available for everyone to use.

Mandatory security has been a research topic for decades in the research community. Originally, the Orange Book defined mandatory security as the multilevel security policy of the Department of Defense [10]. However, in this paper we use a more general notion of mandatory security, in which we consider a security policy mandatory if the definition of the policy and the assignment of security attributes are tightly controlled by the system administrator. This is the most common use of this terminology nowadays [4].

To realize mandatory security, quite a few MAC schemes have been introduced. The most practical solutions have been implemented and are available in commodity operating systems, such as SELinux [11],

LIDS [12], Trustees [13] and AppArmor [14]. Of these, SELinux offers the most secure solution, and has been merged into the Linux kernel for a few years. Unfortunately, despite the fact that it is available on all Linux systems, SELinux is very hard to manage: it usually requires expert skill to develop SELinux policies. Consequently, it is often recommended that novice users disable SELinux, which eventually subjects their system to vulnerabilities. But a lot of people would rather have their systems run without complexity than be secure. This is a headache for SELinux developers: how to make the system secure, but still easy to use for everyone. We believe that this problem cannot be easily solved within the foreseeable future.

In contrast with SELinux, Trustees, AppArmor and LIDS are very easy to manage, even for those who are new to these schemes. With the aim to be a simple and easy-to-use MAC solution, AppArmor enforces its policies only on selected applications (contrary to SELinux, which enforces its policies in a system-wide manner). The policy of AppArmor is based on file paths and can be declared in a human-friendly way in its policy files.

Originally a project named Subdomain [15] developed by a company called Immunix, AppArmor has been acquired by Novell and published under the open-source GPL license. Its ideas have been warmly welcomed by those who dislike the complexity of SELinux. At the moment, AppArmor attracts a group of open source developers who are working to push the project into the Linux kernel as an alternative option to SELinux.

LIDS adopts a similar approach to AppArmor, but in a ''reverse'' way: while AppArmor puts policies on applications, in which a policy declares which files and which modes a specific application can access, LIDS declares which applications can access a specific file. Subsequently, while LIDS seems to be more focused on protecting the data in file systems, AppArmor pays more attention to keeping applications from damaging data in file systems. Another difference is that LIDS aims to provide system-wide protection, but AppArmor only tries to protect critical applications, mostly network services. That is one of the reasons AppArmor has received criticism from the security community: if an unprotected local application is broken, the attacker can leverage it to attack a protected one, and bring it down easily.

However, AppArmor is superior to LIDS in its capabilities in confining sub processes: a thread in an application can choose to change the policy (which is called a ''hat'' in AppArmor terminology). As a result, AppArmor can be used to enforce separate policies for one multi-threaded application such as the Apache web server: with Apache, PHP or Perl scripts can be run with different isolated policies from the overall Apache policy.

Trustees is a MAC scheme inspired by NetWare. It allows the system administrator to attach ''trustees'' to any directory or file. All subdirectories and files in that directory will also inherit these trustees. Trustee rights can be overridden or extended in subdirectories. Subsequently, the access control enforced by Trustees is also based on the file path of the object.

Trustees is designed to lessen the requirement of maintaining too many ACLs in a system with a lot of files and directories. It is also hard and time-consuming to check whether all ACLs are correctly set. Deploying a system with Trustees, the administrator only needs to specify the full system rights in a small configuration file, for specific top directories.

All of the above projects are possible thanks to hooks provided by a security subsystem in Linux named the Linux Security Module (or LSM in short). LSM provides a lightweight and general-purpose access control framework for the mainstream Linux kernel. LSM enables many different access control models to be implemented as loadable kernel modules [16, 17]. Note that LSM does not provide any policy, but rather a generic framework in the form of interception points throughout the kernel. Just before the kernel would have accessed an internal object, a hook makes a call to a function provided by an LSM module. The module can either allow the access, or deny the access and force an error code to be returned.

VMAC Architecture

Goals and Approach

VMAC is designed with the aim of making it more comfortable for the administrator to handle MAC policies in a VM environment. VMAC has the following goals:

1. **An easy-to-manage security policy**: To address the difficulty of managing MAC policies in a VM environment, we propose to centralize the security policies of virtual hosts in a central machine: instead of having separate policy in each separate host, we put all the MAC policies of all the virtual hosts in an administrative VM called *Vi0*. The policies can be managed from inside *Vi0*. The reference-monitor in the kernels of these virtual hosts are modified,[1] so that MAC policies are downloaded from *Vi0*. In *Vi0*, the administrator can locally manage and update the policies of any VM at any time.

   Obviously, with this centralized scheme, the MAC policies become much easier to deploy in a VM environment, because everything can be done from inside *Vi0*.
2. **Aggregate and protect security logging data**: Traditionally, security logging has saved data inside each virtual host. Given machines with

[1] Note that the modification is limited to the security subsystem only.

multiple virtual hosts, managing these scattered logs becomes complicated. Moreover, if an attacker breaks into a virtual host, he might compromise the logging in order to cover his penetration activities. To fix this issue, VMAC is designed to centralize security logging data: instead of keeping the logging data in each separate virtual host, all the access control logging data are sent out to *Vi0*. Thanks to this tactic, all logging data can be saved in one place, so that it is straightforward to aggregate and analyze intrusion evidence of all the maintained hosts in a site-wide manner. We suppose that in our scheme, *Vi0* is securely protected[2] such that the attacker has no chance to exploit our immune data, even if he totally takes over his virtual box.

VMAC Design

The architecture of VMAC exploits the fact that VMs are able to share memory with each other. So each VM (called *ViU* from now on) that wishes to have its security policies centralized and managed in *Vi0* would allocate an area of memory and share it with *Vi0*. This memory is used to exchange information: when receiving a request from *ViU* for the security policy, *Vi0* puts the corresponding policy into shared memory, and notifies *ViU* to take it. Contrariwise, when *Vi0* wants to adjust the policies of a specific virtual host, it informs the *ViU* to get rid of the old policies and take the new policies it has put into shared memory. To mitigate the performance impact, *ViU* can cache the policies in its internal kernel memory, and only flush it out and update the cache when there are updated requests from *Vi0*. Because all information about the policies are exchanged between *Vi0* and *ViU* via the shared memory, the process incurs very little overhead, and our scheme works reliably. Note that the network stack is never used to exchange data between virtual hosts. This strategy avoids the traffic from being sniffed, which could happen if we used the network to forward data.

To send logging data to *Vi0*, all access control logging from the kernel of *ViU* is sent out to *Vi0*, and optionally sent to the traditional kernel log buffer as well. We employ the same shared memory area above for this job: the memory can be divided into two halves, with one half for exchanging security policies and the other for logging data.

VMAC consists of two main components: the first component in each *ViU*, called *VMACU*, and a daemon process running in *Vi0*, called *VMACd*. The overall architecture of VMAC is outlined in Figure 1.

The VMACU Component

The *VMACU* component in the kernel of each *ViU* is responsible for allocating kernel memory which it shares with *Vi0*. Instead of getting policies uploaded from user-space as in the legacy method, *VMACU* gets its policies updated from a daemon process named *VMACd*, which runs in *Vi0*. *VMACU* also uses this shared memory to send logging data to *VMACd*.

In order to send notifications between *VMACU* and *VMACd*, *VMACU* should employ an inter-host communication channel, and use this channel to inform the other host when it wants to get new data, or when asking the other host to get updated data put into shared memory.

[2]Thus *Vi0* might be considered to belong to the Trusted Computing Base (TCB), the core component required to enforce the system security policy.

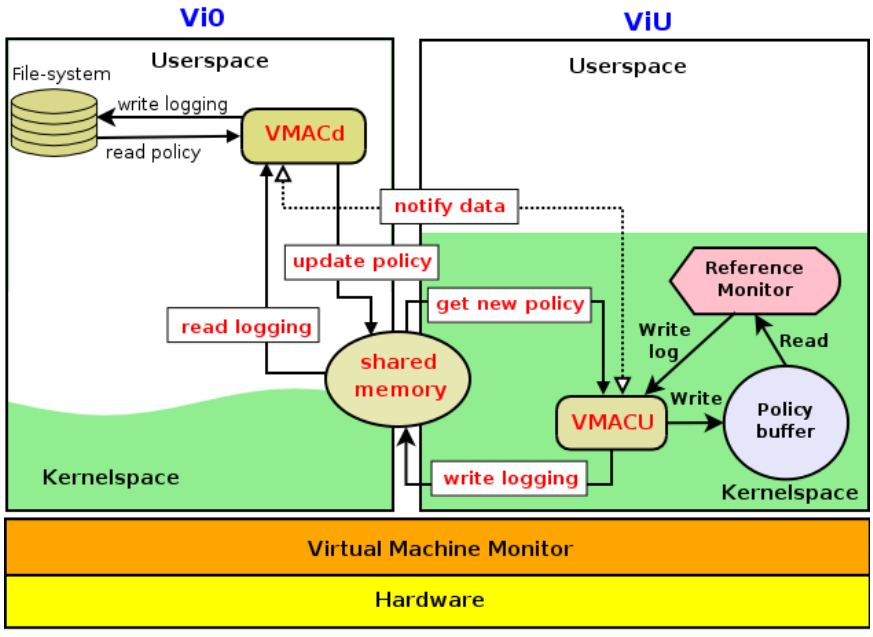

Figure 1: MAC architecture.

In our scheme, information needs to be exchanged between *VMACU* and *VMACd* in two ways: either from *Vi0* to *ViU*, in which case *VMACd* requests *ViU* to update the security policy; or from *ViU* to *Vi0*, in which case *VMACU* sends logging data out to *VMACd*. To address this problem, we employ the same trick widely used in current Xen code: a ring buffer with two halves. The first half is for sending data from *Vi0* to *ViU*, while the other half is for sending data the reverse way. Each half has two heads: one for writing data, and the other for reading data. Applying this format, the shared memory can be accessed by both sides at the same time to update and retrieve information without worrying about conflict.

When initializing, *VMACU* sends a request for security policy to *VMACd*, and gets the policy from shared memory. It then puts all the policy [3] into the policy buffer, and somehow calls the reference-monitor of the MAC scheme running in the system. The reference-monitor then is notified to pick up policies from the buffer, and analyzes them. From then on, the reference-monitor caches policy in memory and never asks for the same policy again. The security enforcer does its job by retrieving policy from cache, so there is no penalty paid for getting policy from *Vi0* at runtime.

To allow administrators to update the policy, at run-time *VMACd* can also ask *VMACU* to update its policy. In this case, *VMACd* does the similar job: it copies the policy into the shared memory, and then notifies the corresponding *VMACU* to come to pick up them. This requires the MAC scheme in *ViU* to get rid of the old policy and take the new policy put in the policy buffer.

VMACU also provides a mechanism to send security logging data to *Vi0*: it exports a function for the MAC scheme to use. Whenever the MAC wishes to log data, it calls the function with the logging data as a parameter. The function then puts the data into shared memory and notifies *VMACd* to pick them up.

The VMACd Component

VMACd is a daemon which listens for policy update requests for a specific host. *VMACd* puts the policy into the shared memory of the corresponding *ViU*. Then *VMACd* informs the related *ViU* about the new policy, so that the virtual host can retrieve the updated policy.

There are two kinds of request for updating security policy that *VMACd* must serve: one is from *VMACU*, and the other is from administrators in *Vi0*.

- At initializing time, *VMACU* sends request to get the security policy. To do that, *VMACU* directly notifies *VMACd* about its demand. In this case *VMACd* passively waits for the request from *VMACU*.
- At run-time, from inside *Vi0* the administrators can modify the policy, and then asks *VMACd* to send the updated policy to *VMACU*. In this case, VMACd actively informs *VMACU* about the updating.

In order to support both these kinds of request, we propose to let *VMACd* actively get the policy from *Vi0*. So whenever *VMACd* needs to serve the updating policy request, either to *VMACU* or administrators, it executes a *policy feeder* and gets the output of the command (this *policy feeder* tool is specific for each *ViU*). The output is regarded as the "raw policy," which is then put into the shared memory with *VMACU* as described above.

Figure 2 describes the connection between the *policy feeder*, *vmacd* and the shared memory with *ViU*.

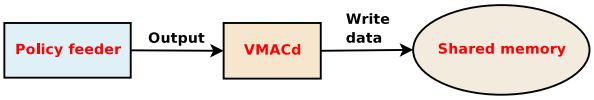

Figure 2: *vmacd* gets the raw policy from the output of the *policy feeder*, then puts these data into the shared memory with *ViU*.

Besides serving the policy update requests, *VMACd* also has the job of gathering logging data sent by *VMACU*. *VMACd* patiently waits for notification of new data from *VMACU*, then retrieves the logging data in the corresponding shared memory area and saves the collected data into separate logs for each corresponding virtual host.

VMAC on Xen

To realize the VMAC architecture described here, we have implemented on Linux with the Xen Virtual Machine. The reason we chose Xen as the virtual machine monitor is that Xen is open source, already stable and available with Linux. Note that the implementation presented here can be similarly applied to other kinds of VMs and MAC schemes, however.

Running on top of Xen, a VM is called a *Xen domain*, or domain for short. A special privileged domain called Domain0 (or Dom0 for short) always runs. Dom0 manages other domains called *User Domains* (or DomU for short), and performs tasks such as start, shutdown, reboot, save, restore and migration between physical machines. Dom0 plays the role of *Vi0*, while DomU plays the role of *ViU* in the VMAC architecture.

Because all domains run on the same machine, they can share physical resources such as memory, hardware interrupts, and peripherals. Note that all such sharing must be permitted by Xen or it is disallowed.

The following sections describe in detail the implementation of VMAC's components in Xen.

The VMACU Component

By default, when initializing *VMACU* allocates a single page of memory (which is equivalent to 4KB in the IA32 platform) and uses this area as shared

[3] The policy is represented in a raw format, and VMAC does not need to understand them

memory with *VMACd*. Then the hardware address of the shared memory is sent to *VMACd* via *XenStore*, using the *XenBus* functions [18]. This shared memory employs the ring buffer format with two halves: one for exchanging security policies, the other for logging data, as explained earlier.

In order to send notifications between *VMACU* and *VMACd*, *VMACU* registers a Xen channel event, and informs *VMACd* about this channel via XenStore. *VMACU* and *VMACd* then register a virtual interrupt (called a *VIRQ* in Xen terminology) and use this interrupt whenever they want to notify each other about data they put into shared memory. Information about this interrupt is also sent to *VMACd* via XenBus interface.

To save policies from shared memory, *VMACU* allocates a memory area and uses it as its policy buffer. After gathering policies from *VMACd*, *VMACU* calls the registered reference-monitor to retrieve the updated policies from this buffer.

Finally, to send logging data to *VMACd*, *VMACU* exports a function named *vmac_send_log()* for MAC schemes to use it. This function simply puts the logging data, which is represented by the function's parameters, to the shared memory, and then notifies *VMACd* to pick up them.

The VMACd Component

VMACd is implemented in Dom0's user-space as a daemon named *vmacd*. *vmacd* has two primary jobs: first, it must serve the policy update request on demand; second, it has to save security logging data.

Update Security Policy

One of *vmacd*'s jobs is to listen for the initializing policy request from *ViU* . Upon the request, *vmacd* executes a special *policy feeder* tool, which is specific for that DomU, and gets the output from the command. The output is considered ‘‘raw policy‘‘, and put into the shared memory with the DomU. Then *vmacd* informs the related *ViU* about the new policy, so it can retrieve the policy and update it in its cache.

vmacd also can be used as tool by administrators to update policy from Dom0 at run-time. When running with a special *--update* option on a DomU, *vmacd* does the similar things we described above: *vmacd* runs the *policy feeder* specific for that DomU,[4] then gets the output as ‘‘raw policy’’ and put them into the shared memory with DomU. The final step is to notify the related *ViU* about the new policy.

To designate the *policy feeder* tool, the pathname of it must be put in a configuration file, which is specific for each DomU.

Save Logging Data

Besides the job of updating policy, *vmacd* also has to gather the logs sent from *VMACU*. *vmacd* waits for notification from *VMACU*, then retrieves the logging data in the corresponding shared memory, saving the collected data into separate logs for the corresponding domain.

To retrieve data delivered from *VMACU* and save them to logs, *vmacd* uses two separate threads: the main thread is used to get the data, and a *worker* thread is used to save the data to the file-system. The main thread silently waits for notification of new data from *VMACU*. When it detects that new data has arrived, it reads the data out from the corresponding shared memory, then copies them to a *host-buffer* [5] allocated by *vmacd* when initializing. While the shared memory size between *vmacd* and *VMACU* should be limited (because it is the memory allocated by DomU's kernel, and kernel memory is a limited resource), this user-space memory can be much bigger.[6] After collecting data from shared memory, the main thread wakes the *worker* up to do its job. Figure 3 outlines the diagram of the whole process.

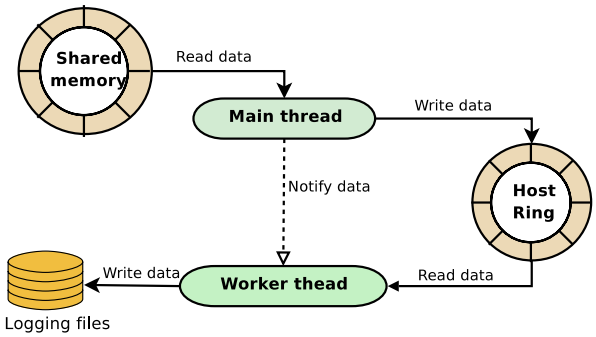

Figure 3: The *vmacd* workflow.

Regarding the *worker* thread, this thread simply waits to be woken up by the main thread (via the *pthread_cond_signal()* function), and reads the data from the *host-buffer*. The logging data is then extracted out, and saved to the logging files on file-system, separately for each domain.

Deploying MAC Schemes

The VMAC scheme presented is intended to serve as a generic framework for all sorts of MAC schemes. However, to make a specific MAC scheme work with the VMAC architecture, there is some work that must be done:

1. **Reference-monitor**: We must make the reference-monitor work with the VMAC architecture, and get the policy from *VMACU*, instead of from user-space as in the traditional method. Though *VMACU* already makes security policies available via the policy buffer, the

[4] this is the same *policy feeder* tool used when *vmacd* serves the request from *ViU*, explained above

[5] This *host-buffer* is of the same *ring buffer* style described earlier, but we do not need two halves in this case, because data only goes one way from main thread.

[6] In fact, we set aside 32KB for this *host-buffer* area

reference-monitor must be informed to retrieve and update the policy from the policy buffer of *VMACU*. This requires a trivial modification on the reference-monitor system of MAC schemes.

2. **Logging data**: Usually MAC schemes send the logging data[7] to the kernel buffer,[8] and the logging data are collected by certain *syslogd* processes in user-space. We must modify the MAC schemes so they deliver these data to the shared memory between *Vi0* and *ViU* instead. In most cases, we can do this simply by replacing the original report function[9] with the VMACU's exported function *vmac_send_log()*, which puts logging data into the shared memory.
3. **Policy feeder tool**: Usually, MAC schemes keep security policy in kernel memory after loading it from user-space. Each MAC scheme employs different way to load and update the security policy. As we have seen, in the VMAC architecture, the *vmacd* running within *Vi0* needs to get the ''raw policy'' from the *policy feeder*. Fortunately, we can modify the available policy-loading and -updating tools of each MAC scheme to do this job. Typically these tools send the compiled policy directly to its kernel. We must patch them, so that they do their job via *vmacd* instead. Specifically, each original tool must be modified to play the role of the *policy feeder*, so it delivers the ''raw policy'' to output for *vmacd* to pick up. After that, *vmacd* transmits these special data to the corresponding *VMACU*.

   Note that *vmacd* does not need to understand the semantic of the ''raw policy.'' It just sends the data as it is, and let the MAC scheme in *ViU* do anything it wants with the received data.

To ease the burden of maintaining patches in the future, we strived to isolate and minimize our modifications to the original source code of MAC schemes. This section describes our experiences when we ported three MAC schemes – AppArmor, LIDS and Trustees – to our VMAC on Xen.

AppArmor

AppArmor gets its security policy via the Linux *sysfs* interfaces: loading policy is done via */sys/kernel/security/.load*, updating policy via */sys/kernel/security/.replace*, and removing via */sys/kernel/security/.remove*. We have modified the kernel-space part of AppArmor such that all these processes are done via the policy buffer of *VMACU* instead. As a result, the original code handling the sysfs interfaces can be removed.

Regarding the security errors and reports made by the kernel component of AppArmor: originally they are sent to the internal kernel buffer with the *printk()* function. We modified the code so that all the messages are sent to the shared memory using the *VMACU*'s exported function *vmac_send_log()*.

At the user-space level, the tool named *subdomain_parser* originally transmitted policy data directly to kernel via these sysfs interfaces. We patched this tool so that it plays the role of the *policy feeder* described in the architecture of *VMACd* component above. Specifically, the modified *subdomain_parser* sends the compiled policy to standard output for *vmacd* to pick up.

All other parts of AppArmor are unchanged for VMAC. In total, the modifications to AppArmor are quite minimal: we replaced only 166 lines of code in the kernel-space part, and replaced around 50 lines of *subdomain_parser* with about 100 lines of new code. These modifications are pretty small when compared with the totality of the AppArmor code – version 2.0 contains around 20000 lines of code.

LIDS

LIDS employs a controversial way to load and update security policies from user-space: in contrast to the ''standard method'' we have commonly seen in other kernel projects, LIDS does not use an interface with user-space to load and update policy at all. Instead, LIDS exploits the system call *sys-read* to read data from user-space to the kernel. This method works quite well, and obviates LIDS from having a user-space tool to interact with kernel.[10] However, this method is strongly criticized by some Linux kernel developers because it violates kernel policy by failing to work in the file-system namespace environment [19].

To adapt LIDS to VMAC, we modified its kernel part: we simply replaced the code that reads the security policies from the file-system [11] with the code to read the policies from the policy buffer. Besides, same as in AppArmor's case, we patched LIDS so that all the security messages are sent to the shared memory using the *VMACU*'s exported function *vmac_send_log()* instead of to the kernel buffer. In all, the kernel code of LIDS was changed by less than 150 lines of code.

Because LIDS user-space tools do not interact with the kernel, we had to write a small *policy feeder* that sends the compiled policy to output, which is then picked up by *vmacd*. This tool, named *lids-vmac*, is very small: only around 200 lines of C code.

Trustees

The latest stable version of Trustees is 3.0. To get security policies from user-space, Trustees creates a special virtual the kernel part of Trustees to pick up policies from the policy buffer of *vmacd*; the *trusteesfs* code can then be disabled. We also made Trustees to

[7] Primarily, security violation reports.

[8] In the Linux kernel, reporting can be done generally with the available *printk()* function.

[9] Most are just the *printk()* function in Linux case.

[10] In fact, the LIDS user-space tool *lidsconf* only focuses on managing and compiling policies.

[11] LIDS keeps all policies under the */etc/lids/* directory.

use the exported function *vmac_send_log()* to send error reports to shared memory instead of to its kernel buffer. The total code changes are around 150 lines.

As for the user-space tool *settrustees*, we modified it to be the *policy feeder*, so that it writes policies[12] to the standard output for *vmacd* to pick up, instead of to *trusteesfs*. This change affects around only 50 lines of *settrustees* code.

Discussions

We have implemented the VMAC architecture on Xen with three MAC schemes: AppArmor, LIDS and Trustees. From Xen's Dom0, the administrator is able to manage and update the policies on virtual hosts at runtime. Each DomU might be configured to run with any scheme, independent from other domains.[13] As for the performance impact of VMAC, because our tool only modifies and adds in support for managing policy from outside, and does not change anything in the security enforcer component of the MAC schemes, VMAC causes no performance penalty on the system.

In all three ports of the mentioned MAC schemes, we have kept the interface and supported code to allow policies to be updated from the user-space of the DomU. That means the administrator could manage the security policies either from inside the DomU, *or* from the central Dom0. But this ''dual-headed'' solution can be confusing, and might cause some conflicts if the policy is updated from Dom0 at the same time. Therefore, we recommend disabling the ability to manage and update the policies from inside DomU. We provide such a compilation option to achieve this for in all three implemented schemes. This tactic has another advantage: if the DomU's kernel access is prohibited,[14] if an attacker successfully breaks into the DomU, he cannot access or modify the security policies in the kernel. As a result, the whole system is more secure and tamper resistant to a root-level attack.

Conclusions and Future Work

This paper proposes an architecture named VMAC to centralize the security policies for VMs, so that managing MAC policies in this environment becomes easier and more flexible. VMAC also centralizes the security logs, so collecting and analyzing information to detect security violations becomes more straightforward. VMAC has no performance impact to the system, as the security enforcer remains unmodified inside the virtual host's kernel.

To prove the concept, we implemented VMAC on top of the Xen Virtual Machine and ported three popular MAC schemes – AppArmor, LIDS and Trustees – to our system. The whole system is flexible, easy to manage and can help ease the hard job put on MAC administrators.

As of this writing, Xen is being pushed into the Linux kernel, and the process is expected to be done sometime in 2006. Once Xen is merged, it will become widely used and a practical solution for everyone. We expect that our solution will then be useful for many people.

For the time being, we are making extensive investigations of other MAC techniques, such as FLASK (which SELinux is based on), RSBAC [20] and Grsecurity [21]. In the future, we plan to go on to support these other schemes if there are requests to do so.

At the moment, we have implemented *VMACU* only for Linux-based VMs. We plan to provide support for other operating systems, such as FreeBSD and NetBSD once these ports work more stably on Xen.

Acknowledgments

We are grateful for the comments we received from Trey Harris, our shepherd, as well as suggestions from the anonymous reviewers who greatly shaped our paper.

Author Biographies

Nguyen Anh Quynh is a Ph.D. student of Graduate School of Media and Governance, Keio University, Japan. His research interest includes Computer Scurity, Operating System, Virtualization technology and Computer Forensics.

Ruo Ando received his Ph.D. degree in Graduate School of Media and Governance, Keio University, Japan in 2006. He is a permanent researcher of National Institute of Information and Communication Technology in Japan since 2006. His research interests include secure operating system, software verification, security testbed and emulation, and automated reasoning.

Yoshiyasu Takefuji is a tenured professor on faculty of environmental information at Keio University since April, 1992 and was on tenured faculty of Electrical Engineering at Case Western Reserve University since 1988. Before joining Case, he taught at the University of South Florida for two years and the University of South Carolina for three years. He received his BS (1978), MS (1980), and Ph.D. (1983) in Electrical Engineering from Keio University.

His research interests focus on neural computing, security, electronic toys. He received the National Science Foundation Research Initiation Award in 1989, the distinct service award from IEEE Trans. on Neural Networks in 1992, the TEPCO research award in 1993, the Takayanagi research award in 1995, the Kanagawa Academy of Science and Technology research award in 1993, the best courseware award

[12] Trustees puts all its policies in the file */etc/trustees.conf*.

[13] Unfortunately at the moment LSM does not support stacking module, so each virtual domain can adopt only one kind of scheme at a time.

[14] In Linux, this can be done by removing the ability to load the kernel module, together with eliminating the */proc/{mem,kmem,port}* interfaces.

from Asia multimedia forum in 1999, the best paper award of Information Processing Society of Japan in 1980, special research award from the US air force office of scientific research in 2003, chairman award from JICA in 2004. He has authored 25 books including neural network parallel computing in 1992, and has published more than 200 papers.

Bibliography

[1] *McAfee Avert Labs: Rootkits, Part 1 of 3: The Growing Threat*, 2006, http://www.mcafee.com/us/local_content/white_papers/threat_center/wp_newappleofmalwareseye_en.pdf .

[2] Naraine, Ryan, *Microsoft: Stealth Rootkits Are Bombarding XP SP2 Boxes,* 2005, http://www.eweek.com/article2/0,1895,1896605,00.asp .

[3] Loscocco, P. A., S. D. Smalley, P. A. Muckelbauer, R. C. Taylor, S. J. Turner, J. F. Farrell, ''The inevitability of failure: The flawed assumption of security in modern computing environments,'' *Proceedings of the 21st National Information System Security Conference*, 1998.

[4] Bishop, M., *Computer Security: Art and Science*, Addison-Wesley Professional, 2002.

[5] *Fedora Core Mailing List*: ''Keeping SELinux on,'' 2006, http://lwn.net/Articles/173812/ .

[6] Dragovic, B., K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, R. Neugebauer, ''Xen and the art of virtualization,'' *Proceedings of the ACM Symposium on Operating Systems Principles*, 2003.

[7] Pratt, I., K. Fraser, S. Hand, C. Limpach, A. Warfield, D. Magenheimer, J. Nakajima, A. Mallick, ''Xen 3.0 and the art of virtualization,'' *Proceedings of the 2005 Ottawa Linux Symposium*, Ottawa, Canada, 2005.

[8] Xen project, *Xen virtual machine monitor*, 2006, http://www.cl.cam.ac.uk/Research/SRG/netos/xen/ .

[9] Clark, B., T. Deshane, E. Dow, S. Evanchik, M. Finlayson, J. Herne, J. N. Matthews, ''Xen and the art of repeated research,'' *Proceedings of the Usenix annual technical conference, Freenix track*, pp. 135-144, 2004.

[10] *DOD 5200.28-STD: Department of defense trusted computer system evaluation criteria*, 1985.

[11] Smalley, S., C. Vance, W. Salamon, *Implementing SELinux as a Linux Security Module*p Nai labs report, NAI Labs, 2005.

[12] LIDS team, *Linux Intrusion Detection System*, 2005, http://www.lids.org .

[13] Ruder, Andrew, *Trustees ACL*, 2006, http://trustees.aeruder.net/ .

[14] AppArmor team, *AppArmor project*, 2006, http://en.opensuse.org/Apparmor .

[15] Cowan, C., S. Beattie, G. Kroah-Hartman, C. Pu, P. Wagle, V. Gligor, ''SubDomain: Parsimonious server security,'' *14th USENIX Systems Administration Conference (LISA 2000)*, New Orleans, LA, 2000.

[16] Wright, C., C. Cowan, J. Morris, S. Smalley, G. Kroah-Hartman, ''Linux Security Modules: General Security Support for the Linux Kernel,'' *Proceedings of the 11th Usenix Security Symposium*, 2002.

[17] Smalley, S., T. Fraser, C. Vance, *Linux Security Modules: General Security Hooks for Linux*, 2003, http://lsm.immunix.org/docs/overview/linuxsecuritymodule.html .

[18] Xen project, *Xen interface manual*, 2006, http://www.cl.cam.ac.uk/Research/SRG/netos/xen/readmes/interface/interface.html .

[19] Kroah-Hartman, Greg, *Driving Me Nuts – Things You Never Should Do in the Kernel*, 2005, http://www.linuxjournal.com/article/8110 .

[20] RSBAC Team, *RSBAC: Extending Linux Security Beyond the Limits*, 2006, http://www.rsbac.org/ .

[21] Spengler, Brad, *Grsecurity*, 2006, http://www.grsecurity.net/ .