A Reference for Secure Virtual Machine Communication

Daniel Moussatche

Intel Software de Argentina
Cordoba, Argentina
Daniel.Moussatche@intel.com

Abstract. Current implementations of information transfer in virtualized environments can produce security breach that could allow malware code to be transferred from one virtual machine (VM) to another. This paper proposes a framework for secure information transfer in virtualized environments via shared memory. An enforcement mechanism implemented in the virtual machine monitor (VMM) uses a policy engine to control the data traffic between VMs. This proposal includes basic functionalities for the sake of the presentation. Enhancements and further extensions are discussed at the end.

Keywords: Virtualization, Secure Virtual Machine, Security Policy.

1 Introduction

In a virtualized environment the necessity of information transmittal between virtual machines (VMs) usually requires to share information between different levels of secure VMs.

The concept of secure VM refers to the level of trust the user sets and the data or process the system in the VM is running, e.g. in one secure VM the user can be using the browser for checking the bank account (secure VM) and in other VM the user is opening an attachment from unknown source (unsecure VM).

By enabling the operations such as *drag and drop* or *copy/cut and paste*, information transfer operations can produce a security breach that allows malware code to be transferred from one VM to another. In this paper a framework for secure information transfer between VMs that enforces the security policies stored in a Policy Engine is proposed.

Sharing information in a virtualized environment, by drag & drop method or cut & paste, is a critical task. Currently, in virtualized environments such as Xen hypervisor [1] there exist several proposals for providing information sharing between VMs: sharing directories through the Network File System (NFS) [2] or mounting the same partition and exchanging information through files, as in the XenFS solution [3]. Another solution proposed is to provide communication from the VMs to a Virtual

Machine Monitor (VMM) by using shared memory tables with an access level, e.g. Xen provides creation of hypercalls to use this shared memory method [4].

One of the major problems of the solutions described above is that they are not enforcing any type of security policy configured by an administrator. In this proposal the administrator can provide a set of security policies, such as configure levels of security for different VMs and which VMs can communicate with other VMs. Moreover, this proposal will allow us to use OS-independent VMs. The keys of the implementation of this proposal are the enforcement agent in the VMM and the use of a policy engine to store the entire configuration of the system.

In order to present this proposal, let us suppose that there exists a set of VMs with a security policy that disallows the reception of information coming from a less secured VM. In this framework, if an application in a virtual machine VM1 is trying to send information to several VMs of the set, the VM1 has to obtain memory space from the *hypervisor* and then store the information in that space. Then, the hypervisor will evaluate the status of the transaction based on a security profile obtained from a *policy engine*. If the security profile does not allow the VM1 to transmit information to a higher security level VM, the hypervisor will detect this infraction and will enforce the profile by preventing the secure VM of receiving the information. For those VMs with lower security level, the hypervisor will transmit the information to them and it will wait for the completion event o timeout. After that, the hypervisor will inform to the origin of the message (the VM1) the result of the operation, allowing the VM1 to decide the retransmission of the original message for the failed ones.

2 Proposed Method

This section describes the proposal for VM secure communication by showing graphically and explaining the flow of information during each step of the process.

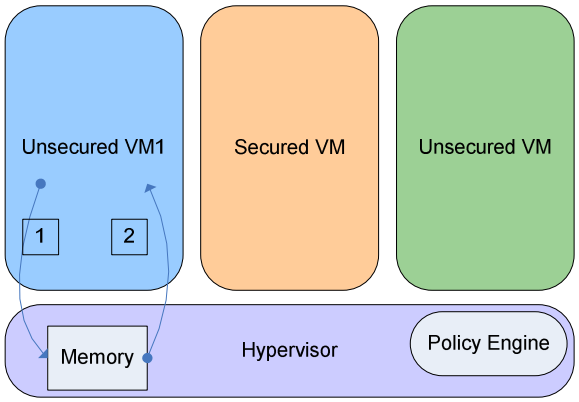

Figure 1: Memory Request

Figure 1 shows an unsecured virtual machine (VM1) requesting permission to transmit information to other VMs. In order to request permission, during the step (1) the VM1 will request to the VMM a suitable space of memory by using the following call and obtaining a handler.

```
char* VMM_Request_Shared_Memory(int nSize);
```

The VMM will check availability in the buffer dynamic allocation data base, and if a free slot exists then the VMM will return the handler to the reserved space in step (2). In case of unavailability or error, an error code will be returned.

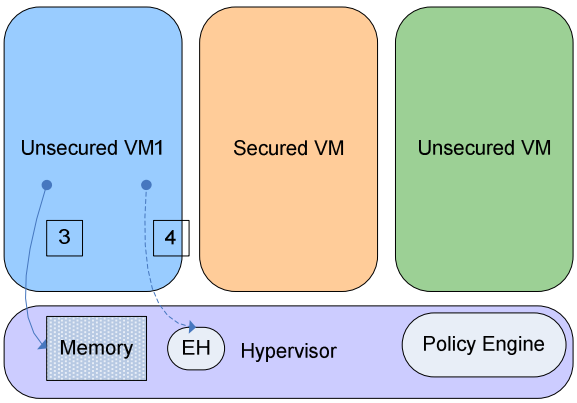

Figure 2: Memory storage

Once the VM1 has the location of the reserved memory, it will use a header structure to store the information in memory. The following structure is proposed to be used in step (3).

```
typedef struct {
          int Cmd;
          VM_List Destination[MAX_VM_COUNT];
          int Length;
          char *Body;
};
```

The fields of this structure will contain the following information:

- **Cmd**: Establishes the command that the VM tries to perform. The possible commands are *Execute*, *Read*, and *Write*.
- **Destination**: List used to declare multiple destination of the information stored in memory.
- **Length**: Bytes length of the information to be transmitted.
- **Body**: The pointer to beginning of the information to be transmitted.

Once the information to be transmitted is stored in the memory, during the step (4) the VM1 will send an event to the VMM indicating that the information was stored and is ready to be analyzed. The proposed event call has the following format.

```
int VMM_Event_Information_Stored(pointer* pHandler);
```

As a result of this call, the VM1 will receive a handler to the reserved memory by the VMM.

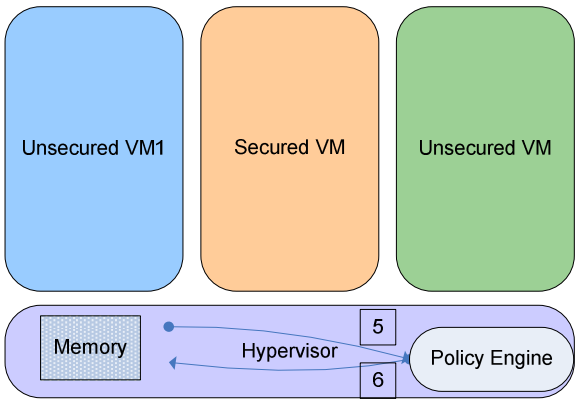

Figure 3: Hypervisor query the PE for security profiles

Figure 3 shows how the VMM deal with the information provided by the VM1. In step (5) the VMM will read the information contained in the struct without the payload and will contact the Policy Engine to validate the information. During step (6) the Policy Engine will provide to the VMM the security policy applied to the source (VM1). Then, the VMM will check the message destinations (field **Destination** of the structure) and will check that information against the security policy.

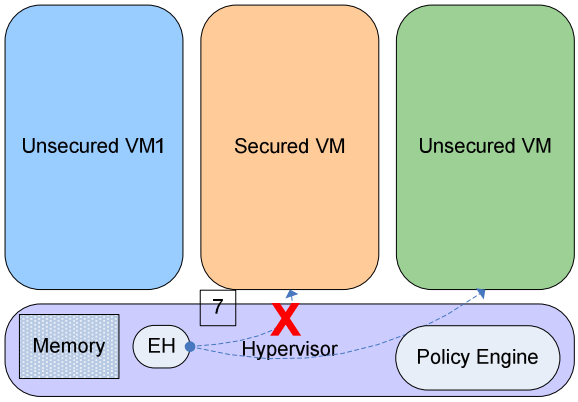

Figure 4: Security profile enforcement

As shown in the Figure 4, if any of the VMs mentioned in the **Destination** field is not allowed in the security profile, then the VMM will take ownership of the information and will eliminate the challenged destinations from the structure. This procedure will allow sending the information in step (7) only to those authorized VMs. Once the packet have been checked, cleaned of unauthorized destinations, and approved, the VMM will send the following event to the authorized destinations allowing them to read the memory.

```
int VM_Event_New_Data(pointer* pHandler);
```

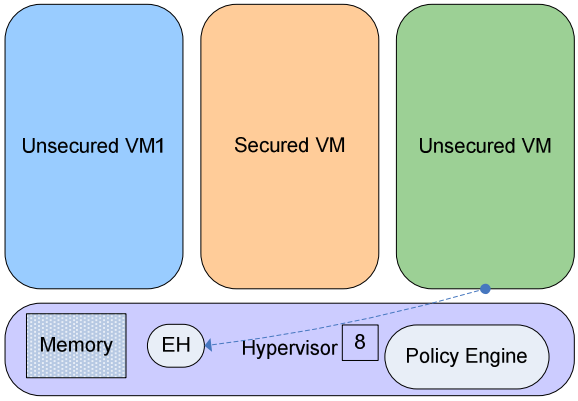

Figure 5: End of Read event sent to the VMM

As shown in Figure 5, the destination VMs will start the reading procedure after they receive the mentioned event. Once the read is complete, the involved VMs will send a event in step (8) to the VMM indicating the completion of the procedure.

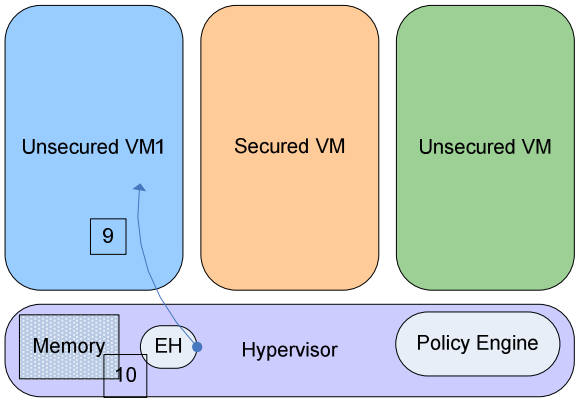

Figure 6: Send result and free memory

Finally, Figure 6 shows that in step (9) the VMM will inform to the VM1 the result of the reading with the following call.

```
int VMM_Event_Read_Completed(pointer* pHandler,
                             struct result pResult);
```

In case of error the VMM will report which VMs did not read the information. Therefore, the VM1 can decide to retry the transmission or not.

Once all the process is complete, the VMM will free the used memory (step 10) and removed the from the allocated table entry with the following call.

```
int VMM_Free_Allocation(pointer *pHandler);
```

3 Analysis

This section analyzes each step of the proposed method in detail and provides insight about the design decisions taken.

The trigger of the process is the request for moving information from one VM to another, e.g. a VM is shutting down and want to alert all the VMs in the system that is going down. The VM generates a call to the VMM requesting a place to store temporarily the information until the target VM(s) receive the information. The information is temporarily stored in order to allow the VMM to verify the security credentials of the destination VM(s). The event request generated by the source VM is received by the VMM's Event Handler. This event handler first validates the security

of the requester VM in order to impede to virtual machines with lower security profile to reserve memory space pretending to be a higher security profile VM.

Once the requester VM is validated, the VMM uses the memory allocation table to check if there is enough available memory. If the space exists, then the VMM returns a handle to this area to the VM and marks the sector in the memory allocation table as reserved. Otherwise, an error message is generated and the requester VM is informed. If the operation was successful, the VM receives the handler to the reserved memory and starts storing the information with a proprietary structure that, as specified in the previous section, includes fields for the command, the destination, the size of the information to be transmitted, and the information. The VMM validates the contents of this structure before allowing the destination VMs to access the information.

During the process of copying the information to the reserved memory the VMM will control the length of the copy, preventing the VM to overflow the reserved allocated memory or just try to copy to a different memory address.

The first field contains the type of command the source VM want the destination VM(s) to execute. By using the command "Read", the VM will ask to the destination VM to read the information as a face of a *Drag and Drop* command. The command "Execute" is used to indicate to the destination VM(s) to execute the file that is been transferred. The last option is the command "Write", allowing the source VM to request to the destination VM(s) to send certain requested information back to the source based on some information sent via the shared memory.

Another key field in the structure is the destination information. This field is implemented as a list of VM indexes, allowing the source VM to send the same message to several destinations VMs in one operation. This feature reduces the performance overhead by preventing the same message to be processed one time for each destination VM. The previous sections do not cover how the list of existing VMs is updated in each VM. One of the possible solutions to analyze can be the publication of an event to alert the VMs about the creation or destruction of a VM in the system. Each VM will be responsible to store and maintain an update list of VMs indexes.

The rest of the field in the structure are related to the information transferred, such as length of the information to transmit (the reported size must not includes the header structure length), this length will allow the destination VM to know exactly the portion of the memory to read; and the pointer to actual data to transmit.

Once the storage of the transfer data in the reserved memory is completed, the source VM generates an event with the memory handler as a parameter. This event is received by the VMM's Event Handler which checks the source of the event and evaluates if the event agrees with the reserved memory handler. Notice that only the owner of the reserved space can trigger an event related to that space, triggering an alert if another VM tries to use the same memory location. If the event is validated and the correct matching is made by the Event Handler then the Security Policy validation process begins.

The VMM reads the information header stored in the reserved memory and uses it to query the Policy Engine for a profile of the source VM. If there is no stored profile for the source VM, the VMM allows the transmission of the information to the destination VMs. Otherwise, if the Policy Engine detects a profile for the source VM, it sends the profile to the VMM. The security profile will contain the configured type of access of the different types of VM, this profiles will contain which are the allowed

commands to tranmit and the allowed destination the source VM can send to. This type of profile is a basic description of the capabilities of this implementation but can be enhanced in several ways.

The VMM then checks the policy profile against the command field and the profile's destination list against the destination field.

If an infraction to the security policy in the command field is detected, then the complete transaction is canceled and an error result is sent to the source VM. Otherwise, the VMM checks the destination list in search of the permission the source VM has to send messages to the destination VMs. If any infraction is detected, the transaction is not canceled, but the disallowed destination VM is marked in the original destination list of the structure. However, the challenged destination VM is not completely removed and an error report is generated.

Once the command and destination list are validated, the VMM Event Handler sends a new data event to the destinations VM requested by the original message. Those events are received by the destination VMs and they start the process of reading the information. The event provides the VMs with the location and size of the data to be read. However, the VMM prevents the destination VM from knowing which other VMs will receive the message by masking the destination list field of the structure. In practice, all the information in that field is set to 0. Once the destination VM accesses the transmitted information, the processing of such information (by executing the requested command) starts.

After the execution of the command each destination VM is responsible for sending a complete-read event to the VMM. The VMM maintains a list of destination VMs that still have to send a read-complete event before the memory can be freed. Once the complete list of destination VMs is unchecked, the VMM produces the transaction-complete event and sends the result of the transaction to the source VM. However, since a destination VM could fail to send a complete-read event, the VMM has a time out scheduled. If the VMM times out, the list of failing destination VMs is kept. This list can be the same list used for VMs that do not passed the security profile evaluation.

Once all the VMs complete the information transmittal or the VMM timed out, the VMM performs a transaction report to the source VM. This report contains the entire destination list with their transaction status, including information about VMs that timed out or failed the security profiling.

Finally, the VMM can release the memory and update the memory allocation table.

4 Conclusions and Future Work

This paper proposes a solution for today's main problem in virtualized environments: the enforcement of security policies. The proposed method of shared memory provides not only a controllable transfer environment via configurable policies but the way for broadcasting information across virtualized platforms. Previous efforts, such as XenFS, or NFS, allow communication through virtualized environments, but none of them targets the configured security enforcement and the multi routing of information across the virtualized platform. This method provides a multilevel

solution to the data transfer across VMs, targeting the possible spread of malware across share file system like XenFS or NFS.

In this paper, several features of the proposal were assumed or used without explicit definitions, e.g. the Policy Engine or the VM update mechanism. These transparent modules that provide services to the proposal should be explained in more detail, but that explanations go beyond the scope of this document.

Obviously, the gains in security functionality come to a cost in performance, making this method not suitable for transfer large amounts of information or stress environment. However, there exists the possibility of enhancing the method by using a chunk support mechanism.

A security enhancement can be implemented during the transmition of the data by analyzing the data transferred in search of known threat code signature, preventing the system to transmit and spread the threat over the rest of the VMs.

Further work can be performed in order to improve this proposal. One straight improvement can be to allow to the Security Profile to configure which VMs can reserve shared memory space. This feature will allow the administrator to prevent some insecure VMs from using the feature of transfer information via memory shared.

Another important security add-in can be the use of an authentication mechanism during the memory reservation request. During the VM creation phase, the VMM can provide to the VM with a unique signature to be used later as authentication mechanism[5].

As mentioned before, the proposal is presented in this paper with basic functionalities in order to simplify the explanation. Other functionalities can be added, such as the implementation of the CRC[6] in the transfer header or the support of chunk transmission. These performance enhancements could be extended with a transmittion request's queue in case the memory allocation table is completed and the VMM do not want to refuse the transactions and queue them for later execution.

The proposal is working in the VMM layer which allows the control of the network before reaching the VMs. By using this advantage, the VMM could use this memory reservation mechanism for sending information over the network and present a secure network information transfer transparent to the VMs.

Acknowledgments. I would like to thank Ricardo Medel (Intel Corp.) for helpful discussions in the preparation of this work; and the peer reviewers for their useful comments that improved this paper.

References

1. Williams, D. E.: Virtualization with Xen(tm): Including Xenenterprise, Xenserver, and Xenexpress. Syngress (2007)
2. Callaghan, B.: NFS Illustrated. Addison-Wesley (1999)
3. Williamson, M.: Xen wiki: XenFS. http://wiki.xensource.com/xenwiki/XenFS
4. Zhao. X., Borders, K., Prakash, A.: Towards Protecting Sensitive Files in a Compromised System. In: Proc. of the 3rd International IEEE Security in Storage Workshop (2005)
5. Kizza, J.M.: Computer Network Security. Springer (2005)
6. Buchanan, W.J.: Advanced Data Communications and Networks. CRC Press (1997)