Provable Security Under Virtualization

Aditya Pendyala  aditya.pendyala@iiitb.net
Nabi Saheb G MD  gmdnabi.saheb@iiitb.net
Reehan Ahmed Khan IL  reehanahmedkhan.il@iiitb.net
Rishabh Namdeo  rishabh.namdeo@iiitb.net
Shweta M  Shweta.M@iiitb.net

Technical Report IIITB-OS-2010-3C
April 2010
Abstract

Virtualization is the mechanism that provides users a way to use many Operating Systems (OS) without need to reboot the system to shift one to another. If an adversary compromises any guest then virtual monitor (XEN in this context) has no reject any request from that OS. There is no formal proof that this virtual monitor provides security from them.

We use virtualized system infrastructure of Xen hypervisor to prove that the system is secure. Any request should be sent by an hypercall to XEN. So we consider here proving the security of these hypercalls. The proof is done using the concept of Random Oracle Model. Only the hypercall security issue of the hypervisor have been dealt with. The hypercall system is proved secure under control data attacks through cipher text attack, either known message text or chosen message attacks.

Online Repositories


© 2010 Aditya Pendyala, Nabi Saheb G MD, Reehan Ahmed Khan II, Rishabh Namdeo, Shweta M. This material is available under the Creative Commons Attribution-Non-commercial-Share Alike License. See http://creativecommons.org/licenses/by-nc-sa/3.0/ for details.
Acknowledgement

We express our gratitude to Prof. Shrisha Rao for his guidance and advice. We would like to thank him for constant encouragement, support and valuable suggestions throughout the project. We take this opportunity to thank institute, team members for their support throughout the project.
**Contents**

1 Introduction 4
2 Technical Description 4
   2.1 Provable Security 4
   2.2 Virtual Machine Monitor 5
   2.3 Hypercall 6
   2.4 Control Data Attacks on Hypervisor 7
3 Xen Hypercalls 7
   3.1 List of Hypercalls 7
   3.2 Hypercalls Control Flow 10
   3.3 Hypercalls at Implementation Level 11
4 Approach 13
5 Implementation 14
   5.1 Problem Statement 14
   5.2 Proof 15
6 Conclusion and Future Work 17
7 References 18
1. Introduction

In the last few years, virtualization technology has been becoming increasingly popular. Virtualization infrastructure definitely gives an advantage of cost effectiveness by allowing different operating systems to run on same hardware [6]. Virtualization platforms such as Xen and VMWare are widely used by both research and industry communities for a variety of applications. With the increased usage and dependence on this technology, its security issues become more and more relevant.

Virtualization merits such as isolation and security are usually taken for granted. There is a tendency to assume that given a much narrower low-level interface compared to that of conventional operating systems, virtualization platforms are inherently more secure. Although such an assumption seems plausible, recent studies show that the virtualization interface can be vulnerable to randomized attacks and is potentially exposed to other attacks [1].

Main Features

This project makes use of the Random Oracle Model (ROM) to prove the security of Xen hypervisor. In this project, only the hypercall security issues of the hypervisor have been dealt with. The project aims to formally prove Xen secure against Known Cipher-text attack using ROM.

2. Technical Description

2.1 Provable Security

In cryptography, a system has provable security if its security requirements can be stated formally in an adversarial model, as opposed to heuristically, with clear assumptions that the adversary has access to the system as well as enough computational resources. The proof of security (called a 'reduction') is that these security requirements are met provided the assumptions about the adversary’s access to the system are satisfied and some clearly stated assumptions about the hardness of certain computational tasks hold.
Some proofs of the security are in given theoretical models such as the random oracle model, where real cryptographic hash functions are represented by an idealization. 'Exact security' or 'concrete security' is the name given to provable security reductions where one quantifies security by computing precise bounds on computational effort, rather than an asymptotic bound which is guaranteed to hold for 'sufficiently large' values of the security parameter.

2.2 Virtual Machine Monitor (VMM)

The VMM or hypervisor can fully mediate all interactions between guest operating systems and underlying hardware, thus allowing strong isolation between virtual machines and supporting the multiplexing of many guests on a single hardware platform. This property is valuable for security.

Xen VMM

The Xen VMM or hypervisor is designed to achieve security. According to its design goal, Xen should be a relatively simple program compared to the conventional OS kernel, with a very narrow, stable and well-defined interface to the software running above it. For instance, Xen has only 35 hypercalls compared to 324 system-calls of the current Linux 2.6.34 kernel. Unlike traditional operating systems, which must support file systems, network stacks etc.; VMM only needs to present relatively simple abstractions, such as a virtual CPU and memory. As a result of these properties, Xen (and other VMMs in general) have been widely considered as architecture for building secure operating systems.

Enhancing security for a virtualization is a practical need to make this technology more robust. For instance, a more secured Xen is beneficial for a variety of services built on top of Xen and would improve both its reliability and usability. There are several aspects regarding to security of Xen. Among these the security of the Xen hypercall interface is critically important because of its role as a sole means to communicate to the VMM. Secure Xen hypercalls would prevent certain kinds of attacks against the virtualization environment; thus enhance its security overall [1].
2.3 Hypercall

Guest Operating Systems communicate with Xen and make resource requests via the XEN API or hypercalls. The hypercall interface serves as the sole mechanism for cross-layer interaction between VMs and Xen. The role of hypercalls in the Xen VMM is similar to the role of system calls in an OS. Basically, a hypercall is a software trap from a domain or virtual machine to Xen, just as a syscall is a software trap from an application to the kernel. A hypercall transfers control to a more privileged state in the VMM. Domains use hypercalls to request high privileged operations like low-level memory requests.

Xen Hypercall Security Issues

A guest kernel on Xen cannot directly modify memory data structures such as page tables, global descriptor tables or local descriptor tables. These operations can be requested through hypercalls. If attacker could tamper with hypercalls, she potentially can cause some damages for example, elevation of access rights to pages not intended for this guest so she can access memory pages of other guests.

Hypercalls being similar to native syscalls can potentially have equivalent flaws. From a security viewpoint, the hypercall interface is a convenient vehicle for an attacker to carry attacks. Compromising a guest OS by an attacker is necessary to launch malicious hypercalls, leading to serious security problems to the entire virtualization system.

A typical scenario of hypercall attack could happen in the following steps:

1. Attacker compromising one of the applications on a guest OS. This is possible even if the guest started in a clean state because guest applications communicate with the outside world; they could be infected with malware;

2. On success of the first step, the attacker would escalate his privilege by common syscall attack methods;
3. When the attacker can get inside the guest kernel (equivalent to the escalation to the ring-1 privilege in the x86 protection ring architecture), he can launch attacks to the hypervisor via hypercalls.

### 2.4 Control Data Attacks on Hypervisor

This is a type of attack where the attacker targets control data - data which are loaded into the program counter during program execution such as return addresses and jump pointers as these are transition points where a change of control flow is possible. Mostly these types of attacks try to alter the target program’s control data in order to execute injected code or out-of-context library code.

One of the form of control data attack that can be launched with the hypercall is to get the control transferred to malicious code on the occurrence of system call in a guest OS. This code could generate a different type of system call as opposed to the original one and pass different set of arguments to Xen kernel so as to get control of resources the current guest OS not intended for.

The key here is the appropriate authentication of the hypercall. If the malicious code is able to generate the same set of authentication that a genuine hypercall can do then it is clear that the hypervisor has been compromised and the adversary can easily launch attacks on the system taking full control and misusing the resources.

### 3. Xen Hypercalls

#### 3.1 List of Hypercalls

Following is the location of all hypercalls: The Hypercalls appear according to their order in xen.h.

The hypercall table itself is in `xen/arch/x86/x86_32/entry.S` (ENTRY(hypercall_table)).

```plaintext
HYPERVISOR_set_trap_table => do_set_trap_table() (file xen/arch/x86/traps.c)
HYPERVISOR_mmu_update => do_mmu_update() (file xen/arch/x86/mm.c)
HYPERVISOR_set_gdt => do_set_gdt() (file xen/arch/x86/mm.c)
HYPERVISOR_stack_switch => do_stack_switch() (file xen/arch/x86/x86_32/mm.c)
```
HYPERVISOR_set_callbacks => do_set_callbacks() (file xen/arch/x86/x86_32/traps.c)
HYPERVISOR_fpu_taskswitch => do_fpu_taskswitch(int set) (file xen/arch/x86/traps.c)
HYPERVISOR_sched_op_compat => do_sched_op_compat() (file xen/common/schedule.c)
HYPERVISOR_dom0_op => do_dom0_op() (file xen/common/dom0_ops.c)
HYPERVISOR_set_debugreg => do_set_debugreg() (file xen/arch/x86/traps.c)
HYPERVISOR_get_debugreg => do_get_debugreg() (file xen/arch/x86/traps.c)
HYPERVISOR_update_descriptor => do_update_descriptor() (file xen/arch/x86/mm.c)
HYPERVISOR_memory_op => do_memory_op() (file xen/common/memory.c)
HYPERVISOR_multicall => do_multicall() (file xen/common/multicall.c)
HYPERVISOR_update_va_mapping => do_update_va_mapping() (file xen/arch/x86/mm.c)
HYPERVISOR_set_timer_op => do_set_timer_op() (file xen/common/schedule.c)
HYPERVISOR_event_channel_op => do_event_channel_op() (file xen/common/event_channel.c)
HYPERVISOR_xen_version => do_xen_version() (file xen/common/kernel.c)
HYPERVISOR_console_io => do_console_io() (file xen/drivers/char/console.c)
HYPERVISOR_physdev_op => do_physdev_op() (file xen/arch/x86/physdev.c)
HYPERVISOR_grant_table_op => do_grant_table_op() (file xen/common/grant_table.c)
HYPERVISOR_vm_assist => do_vm_assist() (file xen/common/kernel.c)

HYPERVISOR_update_va_mapping__otherdomain =>
do_update_va_mapping__otherdomain() (file xen/arch/x86/mm.c)
HYPERVISOR_iret => do_iret() (file xen/arch/x86/x86_32/traps.c) /* x86/32 only */
HYPERVISOR_vcpu_op => do_vcpu_op() (file xen/common/domain.c)
HYPERVISOR_set_segment_base => do_set_segment_base (file xen/arch/x86/x86_64/mm.c) /* x86/64 only */
HYPERVISOR_mmuext_op => do_mmuext_op() (file xen/arch/x86/mm.c)
HYPERVISOR_acm_op => do_acm_op() (file xen/common/acm_ops.c)
HYPERVISOR_nmi_op => do_nmi_op() (file xen/common/kernel.c)
HYPERVISOR_sched_op => do_sched_op() (file xen/common/schedule.c)

FILES IN SOURCE CODE HAVING THESE FUNCTION DEFINITIONS AND CALLS

These are 15 .C files in which the above 29 hypercalls are called....

1. xen/arch/x86/traps.c
2. xen/arch/x86/mm.c
3. xen/common/schedule.c
4. xen/common/dom0_ops.c
5. xen/common/memory.c
6. xen/common/multicall.c
7. xen/common/event_channel.c
8. xen/common/kernel.c
9. xen/drivers/char/console.c
10. xen/arch/x86/physdev.c
11. xen/common/grant_table.c
12. xen/arch/x86/x86_32/traps.c
13. xen/common/domain.c
14. xen/arch/x86/x86_32/mm.c
15. xen/common/acm_ops.c
3.2 Hypercalls Control Flow

The above figure depicts the control flow in Xen for a request sent to Xen kernel. If a request comes to kernel then kernel.c module is invoked which then validates the version after which the control flows to guest module. Only if it is valid access, respective module is invoked to serve the request.
Figure-2 shows a do_multicall hypercall trace. It can be found in xen/common directory with name multicall. This will manage multiple calls from guest to kernel and vice-versa.

### 3.3 The Hypercall at Implementation level

A hypercall in a virtualized environment is like the system call in a non-virtualized OS environment. Any system call is trapped by the hypervisor and converted to a hypercall. The hypercall mechanism occurs in the steps as given in the below high-level view (Figure 1).

It has to be noted that the code to pass the arguments to the hypervisor is done by pushing the system call number and the arguments on to the stack and/or registers before the control is passed over to the hypervisor code. This code to generate the required arguments lies in the guest domain.

For the subroutine or hypercall handler of the hypercall__HYPERVISOR__xxx, usually is do_xxx. For example, the handler of HYPERVISOR sched op should be do_sched op. The do xxx routines are entries of the hypercall table defined in xen\arch\x86\x86 {32,64}\entry.S.
Whenever a request comes from a guest(1) it will be encrypted using in AES(2) where two files rjindeal.c and aes.c (3) are used to encrypt. This will goto XEN kernel where it will be decrypted(4) and again encrypted to host operating system(5). The host then sends this to access the corresponding resource(6). This is shown in Figure-3.
4. Approach

Figure-4 shows method begins by identifying the model that simulates Random Oracle in XEN and BlackBox. Identify the types of attacks that it may suffer from because of adversaries access to BlackBox and calculate the probability of its success.

**Oracle:** A Random Generator or a cryptographic algorithm

**Black Boxes:** Public Oracle (can be accessed by adversaries also)

**Adversary:** Malicious entity.
5. Implementation

5.1 Problem Statement

In a Xen hypervisor which traps all hypercalls, with the hypercalls being AES encrypted and AES being unbroken as of now then adversary cannot launch a known cipher text attack on Xen.

Assumptions:

We are assuming that AES is unbroken and is secure. The underlying security model used for proving Xen system secure is Random oracle Model (ROM).

Theorem Statement:

In order to prove that the system to be KCA (Known Ciphered Attack) secure we should prove that the probability of adversary attacks is having a very negligible value i.e.

\[
\Pr \left( k \leftarrow G \left( 1^n \right) ; c \leftarrow E_R \left( m, k \right), m \leftarrow f(c) \right) \text{ is negligible then Xen can be called provable secure.}
\]

Where,

- \( k \leftarrow G \left( 1^n \right) \) denotes the key generated by key generation algorithm, \( G \) and it is of \( n \) bits.
- \( c \leftarrow E_R \left( m, k \right) \) denotes the cipher text deduced from encrypting the message \( m \) using key \( k \).
- \( m \leftarrow f(c) \) denotes the message adversary gets by applying some function on \( c \), cipher text.

Notations:

**A. Algorithm**

1. **Encryption Algorithm (E):** It takes an input message \( m \in M \) and a key \( k \) and generates ciphered message \( c = E \left( m, k \right) \).

2. **Decryption Algorithm (D):** It outputs the plain text message \( m' \) on input a ciphered text message \( c \) and key \( k \), i.e. \( m' = D \left( c, k \right) \).

3. **Key Generation Algorithm (G):** It generates in polynomial time a random key \( k \) on input a string consisting of \( n \) bits. Here \( n \) can be referred as security parameter.
An adversary trying to detect cryptographic communication is faced to solve the cryptographic decision problem.

4. **Cryptographic Decision Problem**: Given $c$ determine if there exist a $k \in \{0, 1\}^*$ in the range of $G$ and a message $m$ such that $D(c, k) = m$.

**B. Attack**

**Known Cipher Attack**:

Adversary deduces original message from ciphered text message by applying some function on cipher-text.

**C. Oracle**

A Random Oracle $R$ is a map from $\{0,1\}^* \to \{0,1\}^\omega$ chosen by selecting each bit of $R(x)$ uniformly and independently for every $x \in \{0,1\}^*$ where $\{0,1\}^\omega$ denotes the space of infinite binary strings. Oracle is a random generator. It maps the seed $x \in \{0, 1\}^*$ to infinitely long random sequence.

**D. Black Boxes**

1. **Encryption Black Box** ($E_R$): It takes an input message $m \in M$ and a key $k$ and generates ciphered message $c = E(m, k)$.

2. **Decryption Black Box** ($D_k$): It outputs the plain text message $m'$ on input a ciphered text message $c$ and key $k$, i.e. $m' = D(c, k)$.

3. **Message Generating Black Box** ($P$): It returns a message $m$.

A formalized proof for the above discussion based on proof by contradiction as in [5] is given below

**5.2 Proof**

**Step I** Firstly, we need to find out the Oracle of system. AES is the basal cryptographic algorithm.

**Step II** Based on the above analysis and the description of the system we have
\( \Pr (k \leftarrow G (1^n); c \leftarrow E_R (m, k), m \leftarrow f(c)) \)

Reduced to

\( \Pr (k \leftarrow G (1^n); c \leftarrow E_{AES} (m, k), m \leftarrow f(c)) \)

This is nothing but reducing the security of system to security of Oracle (AES).

**Step III** We are going to prove this by contradiction.

- Let \( A \) be an adversary who is going to attack the system.
- Let an event \( A_1 \) be that an adversary gets message \( m \) by performing some operation on cipher text.
- Advantage of \( A \) is denoted by \( \lambda (n) \) which is nothing but the value we get by removing the probability of adversary randomly guessing correct result i.e. \( \frac{1}{2} \) from the probability of adversary succeeding.
- Probability of \( A \) succeeding can be given as
  \[
  \Pr (A \text{ succeeds} | A_1) \Pr (A_1) + \Pr (A \text{ succeeds} | A_1^c) \Pr (A_1^c)
  \]
  This is equal to \( \frac{1}{2} + \lambda (n) \).

But,

\[
\Pr (A \text{ succeeds} | A_1) \Pr (A_1) + \Pr (A \text{ succeeds} | A_1^c) \Pr (A_1^c) \leq \frac{1}{2} + \Pr (A_1)
\]

From above 2 statements we can infer that \( \Pr (A_1) \geq \lambda (n) \). But \( \lambda (n) \) is not negligible. Hence \( \Pr (A_1) \) which is nothing but probability of adversary getting the message \( m \) is not negligible. In precise we can say that adversary breaks AES with not negligible probability. This is contrary to the fact that AES cannot be broken. So the advantage of adversary is negligible. Hence, the \( \Pr (A_1) \) is negligible. Hence the probability of adversary succeeding with its attacks is very less.

Hence the system is secure.
6. Conclusion and Future Work

The proof is only under the assumption that the AES is secure. Further a more tight reduction to more secure mathematical model would give more assurance to the security of the system.

A cipher text attack through control data attacks are not the only type of attacks that a hypercall can suffer. Further investigation is necessary to identify all the forms of attacks on the hypercall of Xen so as to locate any security issues.

Furthermore a hypercall attack is just one of the forms of attack that can occur on Xen. There can be attacks on other hypervisor resources too like attacks on VMM configuration information, virtual storage file system, per-guest configuration data etc. These have to be addressed and formalized in to a provable security model.
7. References


Available at: http://www.anandtech.com/printarticle.aspx?i=3263
[Accessed: January 26, 2010]

Available at: http://www.research.ibm.com/secure_systems_department/projects/hypervisor/
[Accessed: January 23, 2010]