Xen Network Flow Analysis for Intrusion Detection*

Reece Johnston†, Sun-il Kim†, David Coe‡, Letha Etzkorn†, Jeffrey Kulick‡, and Aleksandar Milenkovic‡
Computer Science† and Electrical and Computer Engineering‡ Departments
University of Alabama in Huntsville
{reece.johnston,sunil.kim,coed,etzkorl,kulickj,milenka}@uah.edu

ABSTRACT

Virtualization technology has become ubiquitous in the computing world. With it, a number of security concerns have been amplified as users run adjacently on a single host. In order to prevent attacks from both internal and external sources, the networking of such systems must be secured. Network intrusion detection systems (NIDSs) are an important tool for aiding this effort. These systems work by analyzing flow or packet information to determine malicious intent. However, it is difficult to implement a NIDS on a virtualized system due to their complexity. This is especially true for the Xen hypervisor: Xen has incredible heterogeneity when it comes to implementation, making a generic solution difficult. In this paper, we analyze the network data flow of a typical Xen implementation along with identifying features common to any implementation. We then explore the benefits of placing security checks along the data flow and promote a solution within the hypervisor itself.

CCS Concepts

•**Security and privacy** → **Intrusion detection systems; Virtualization and security;** *Network security;*

Keywords

Hypervisor, Xen

1. INTRODUCTION

Cloud computing has become ubiquitous in today's world due to its transparency and ease-of-use for a client and due to its profitability as a utility [3]. As with any computer system, security is a major concern, especially considering the nature of clouds: they run multiple users' applications adjacently on a single machine which incurs greater impact from a successful attack. Ultimately, to secure the cloud its constituent technologies must be secured, the most important being virtualization. We focus on the Xen hypervisor, an increasingly popular virtualization technology.

*This work was supported by the National Security Agency grant H98230-15-1-0268.

CISRC '16, April 05-07, 2016, Oak Ridge, TN, USA

DOI: http://dx.doi.org/10.1145/2897795.2897802

Unfortunately, the Xen hypervisor is a minimally documented piece of software, making security analysis difficult; it also utilizes a vast number of computing techniques all of which need securing. Fortunately, one technique that is paramount is networking, so we will look at Xen's architecture through the lens of network security. Specifically, we will inspect the inner workings of Xen with regard to network flow and analyze said flow for potential intrusion detection implementation.

The rest of the paper is organized as follows: first we discuss background information, followed by a trace of Xen's networking, then the resulting analysis, and finally a conclusion with a discussion of future work.

2. BACKGROUND

In this section, we discuss virtualization, and cover intrusion detection systems (IDSs) with a focus on clouds.

2.1 Virtualization

Virtualization is the process of creating and managing a virtual version of a resource or entity. Typically, the entity is an operating system (OS) or physical device (i.e. NIC). For this paper, we are primarily concerned with virtualization of physical devices. This type of virtualization is commonly handled via a virtual machine monitor (VMM/hypervisor) which lies between a collection of virtual machines (VMs) and the underlying hardware. This VMM provides all of the resource sharing and shared memory mechanisms of a traditional OS, but it does so for OSs that run atop it (the VMs). Due to its popularity, the Xen hypervisor, as introduced in [1], is of particular concern. Xen has a few particularities when compared to other VMMs; its use of privileged domains being most relevant to our discussion. Xen has various domains (VMs) that can be granted certain hardware access. These domains are referred to as Doms (ie. Dom0, Dom1, ..., DomU). Typically, Dom0 is treated as the single privileged domain handling all IO. However, stub domains are arising that handle specific types of IO for guest DomUs. Either way, there are still a number of security concerns with any privileged domain. Therefore, a number of security mechanisms should be utilized such as firewalling, intrusion detection, and virus scanning. For this paper, we will focus on intrusion detection.

2.2 Intrusion Detection Systems

IDSs are used to detect incoming attacks using either signature-based detection (SD) or anomaly-based detection (AD). These work by analyzing data and then stopping or allowing an action based on the analysis. The methods for

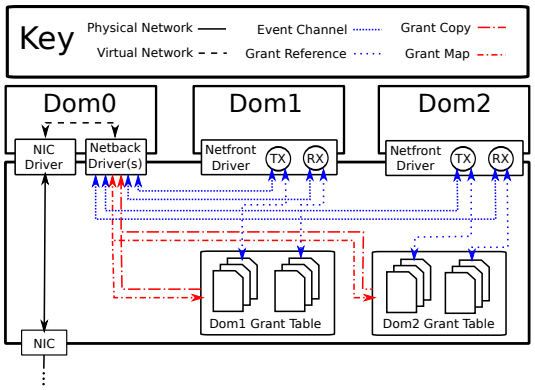

Figure 1: Typical Xen Network Data Flow

analysis can be statistics-based, pattern-based, rule-based, etc; and a large body of research literature covers many such methods. However, most focus on supporting a single system or application. Recently, there have been some advances towards IDSs designed for virtualized environments [5].

The cloud IDSs proposed are typically network-based systems (NIDS) or host-based systems (HIDS) [4]. NIDSs work by inspecting network flow and packet content, and it is worth noting that packet content may be encrypted making certain implementations impossible. One example of a NIDS in the cloud, SnortFlow, targets OpenFlow-based virtual networks via SD techniques; meaning, it sits on a network management VM [8]. However, a NIDS need not be loaded onto a VM since it can be on a cluster controller [6]. As for HIDS, they inspect local behaviour through various introspection or monitoring techniques. An important HIDS example, Collabra, is situated within the VMM but utilizes multiple hosts for attack detection. Moreover, it targets hypercalls and their parameters which is similar to the model we will be promoting. However, their implementation is more a HIDS than a NIDS since it does not focus on network data [2]. With this, the various flavors of cloud IDSs have been addressed, and notably, none of the NIDS lie within the VMM layer.

3. XEN NETWORK DATA FLOW

In this section, the Xen hypervisor is examined for NIDS deployment. Specifically, the network data flow of a Xen system is explored at the VM layer and at the VMM layer.

3.1 Xen Network Architecture

When dealing with Xen, a number of significant issues arise: it is in rapid flux, has wildly heterogeneous implementations, and exposes various public interfaces that may not be used by a given VM.

Figure 1 shows a typical Xen network data flow. The VM layer (comprised of Dom0, Dom1, and Dom2) sits on top of a VMM layer (Xen 4.5) which sits atop a physical layer. In each layer, the networking components are shown. Specifically, each DomU depicts the network drivers, their data flows (grant operations), and their control flows (event channels). However, the distinction between the VM type (HVM/full or PVM/paravirtualized) is not depicted since similar mechanisms will be used, especially when a HVM uses PV drivers. Also, the netback driver(s) are not expounded upon, but there is likely an instance per DomU. Then, the VMM layer depicts the grant tables of each DomU which are key to shared memory in a Xen system. These

```
570  while (xenvif_rx_ring_slots_available(queue)
571         && (skb = xenvif_rx_dequeue(queue)) != NULL) {
572      queue->last_rx_time = jiffies;
573
574          XENVIF_RX_CB(skb)->meta_slots_used =
                 xenvif_gop_skb(skb, &npo, queue);
     . . .
585  gnttab_batch_copy(queue->grant_copy_op, npo.copy_prod);
```

Listing 1: linux/drivers/net/xen-netback/netback.c

```
248  ref = get_free_entries(1);
249  if (unlikely(ref < 0))
250      return -ENOSPC;
251
252  gnttab_grant_foreign_access_ref(ref, domid, frame, readonly);
     . . .
770  if (HYPERVISOR_grant_table_op(GNTTABOP_copy, batch,
         count))
```

Listing 2: linux/drivers/xen/grant-table.c

function via the references shown which serve as capabilities for foreign DomUs. This completes the architecture explanation, so we can move to the data flows.

3.2 Xen VM Layer Network Flow

For network flow, there are a few possibilities: external to VM, VM to external, or VM to VM. Each of these has a slightly different flow and so each shall be discussed. However, it is important to keep in mind that some specifics of the flows can change with implementation, but the VMM level will remain consistent. Therefore, after detailing the VM level flows by examining Linux 4.4-rc8 code, we will switch to discuss the flows at the VMM level.

For external to VM, data moves from the NIC to some driver. Then, utilizing a virtual network (commonly, Linux bridge utilities and/or Open vSwitch), the data is connected to the appropriate netback driver. Essentially, this works like a physical network, either through routing or switching-like behaviour. Also, virtual interfaces (VIFs) are created which function like standard network interfaces. From here, the netback driver queues the packet and waits for available space in the associated DomU's RX buffer. The availability of this buffer is indicated by the netfront driver via the depicted RX event channel. In the Linux source, the netback and netfront drivers handle these channels via a Xen ring. These rings rely on Xen's event and grant mechanisms which are discussed in the next subsection. Looking at Listing 1, we see some code segments from the xenvif_rx_action function of the netback driver. In which, grant copy operations are constructed via a xenvif_gop_skb call that uses RING_COPY_REQUESTS to acquire available slots (and grant references) from the RX ring. Then, further down, a batch gnttab_batch_copy operation is performed using these constructed operations. In this batch operation, we finally arrive at the hypercall responsible for moving packet data. Specifically, this is the HYPERVISOR_grant_table_op hypercall seen in Listing 2. From here, execution switches over to the VMM to perform a grant copy operation. The details of which are discussed in Section 3.3. Once execution returns to the netback driver, the packet data will be in the grant entries referenced in the RX buffer, so netback proceeds to do response checking, fragment pushing, and cleanup. With this completed, the netfront driver can access the data, so once that DomU begins execution and netfront gets CPU time, the response messages are read and the packet data is retrieved. This exact functionality lies in the xennet_poll function of "linux/drivers/net/xen-netfront.c"; which is not covered since a NIDS should not be used past this point.

In the VM to external flow, the same mechanisms are

```
1011  case EVTCHNOP_alloc_unbound: {
1012      struct evtchn_alloc_unbound alloc_unbound;
1013      if ( copy_from_guest(&alloc_unbound, arg, 1) != 0 )
1014          return -EFAULT;
1015      rc = evtchn_alloc_unbound(&alloc_unbound);
      ...
1021  case EVTCHNOP_bind_interdomain: {
1022      struct evtchn_bind_interdomain bind_interdomain;
1023      if ( copy_from_guest(&bind_interdomain, arg, 1) != 0 )
1024          return -EFAULT;
1025      rc = evtchn_bind_interdomain(&bind_interdomain);
      ...
1069  case EVTCHNOP_send: {
1070      struct evtchn_send send;
1071      if ( copy_from_guest(&send, arg, 1) != 0 )
1072          return -EFAULT;
1073      rc = evtchn_send(current->domain, send.port);
```

Listing 3: xen-4.5/xen/common/event_channel.c

used, but the roles are reversed between the netfront and netback drivers: the DomU netfront driver now puts the packet data into the grant entries referenced in the TX ring; then, the netback driver copies that data directly since the entries were made available to its domain during the TX ring setup. This can be seen in the setup_netfront function of "linux/drivers/net/xen-netfront.c" which calls xenbus_grant_ring in turn invoking gnttab_grant_foreign_access. Looking in Listing 2, this grant operation requests a free grant entry via get_free_entries. Then, it updates that entry to allow foreign access by calling gnttab_grant_foreign_access_ref. Notably, this does not invoke a hypercall as it is working in its own memory space. As for the marshaling, this occurs in the xennet_start_xmit of the netfront driver. After loading the marshaled data into the TX ring, the netfront driver uses a ring request or notify_remote_via_irq call to message the netback driver. From here, execution picks back up at the netback driver. Specifically, the xenvif_tx_action function processes the requests: it performs a gnttab_batch_copy operation which in turn makes the grant copy hypercall(s). Once the data is copied, it's pushed to the virtual network which in turn routes it out the NIC interface.

For VM to VM communication, portions of the flows for external to VM and VM to external are followed. Specifically, the VM to VM flow can be viewed as identical to a VM to external until the virtual network is reached. Then, the arbiter of this virtual network simply redirects the traffic to the VIF associated with the destination DomU. From this point on, the flow mimics an external to VM flow.

3.3 Xen VMM Layer Network Flow

Network flows in the VMM layer rely on hypercalls to invoke event channel and grant operations. As such, this subsection will introduce hypercalls, then cover how the event channel and grant operations function. These mechanisms are used in the Section 3.2's various flows which can be referenced in relation to the hypercalls covered here.

All of the requests to the VMM layer hinge around a hypercall being invoked by a VM. This can be thought of as a syscall or software trap. However in a hypercall, the return path is an event channel which works as an asynchronous queue of notifications. These in turn tell an event-callback handler what actions to take. The specific hypercalls of concern revolve around event channel operations and grant operations, as these are pertinent to Xen networking.

For event channels, the HYPERVISOR_event_channel_op hypercall performs these significant EVTCHNOP_* operations: alloc_unbound, bind_interdomain, and send. The do_event_channel_op function shown in Listing 3 handles calling these operations. The EVTCHNOP_alloc_unbound operation sets up a new event channel via the invoked evtchn_

```
2643  case GNTTABOP_map_grant_ref:
2644  {
2645      XEN_GUEST_HANDLE_PARAM(gnttab_map_grant_ref_t
          ) map =
2646          guest_handle_cast(uop, gnttab_map_grant_ref_t);
2647      if ( unlikely(!guest_handle_okay(map, count)) )
2648          goto out;
2649      rc = gnttab_map_grant_ref(map, count);
      ...
2709  case GNTTABOP_copy:
2710  {
2711      XEN_GUEST_HANDLE_PARAM(gnttab_copy_t) copy =
2712          guest_handle_cast(uop, gnttab_copy_t);
2713      if ( unlikely(!guest_handle_okay(copy, count)) )
2714          goto out;
2715      rc = gnttab_copy(copy, count);
```

Listing 4: xen-4.5/xen/common/grant_table.c

alloc_unbound function. In the drivers, this operation is called by netfront to setup event channels for the TX and RX ring buffers. The next operation, EVTCHNOP_bind_interdomain, is used to connect to an existing event channel via the bind_interdomain invocation. In which, the current domain's provided port is bound to the remote one. In the Linux drivers, this operation is used to connect a netback driver to a netfront driver. Lastly, EVTCHNOP_send operations send a notification on the event channel by flagging a bit. In essence, these event channels behave like hardware interrupts, and notably, this means that they are not directly responsible for any substantive data transmission: they merely notify, so for the Linux drivers, they notify when a ring has data to process. As for data transmission, we must look at the grant operations.

The grant hypercall, HYPERVISOR_grant_table_op has a few important GNTTAB_* operations: map_grant_ref, transfer, and copy. The provided listing, Listing 4, shows how these are called. In it, the code switches on what operation is passed in. For GNTTAB_map_grant_ref, the gnttab_map_grant_ref function is called. This function maps the referenced grant entry into the current domain's memory which allows that domain to perform read or write operations (depending on flags). This operation is used in the netback driver to access outgoing packets from a DomU. The next operation, GNTTABOP_transfer, while rarely used, is a valid methodology: it allows for the ownership of a grant entry to be changed to another domain. Meaning, a network driver could pass grant entries back and forth by switching their ownership. However, this incurs frequent TLB invalidation which is highly inefficient. The last operation, GNTTAB_copy, is the most common operation. In fact, the linux implementation discussed in Section 3.2 relies heavily upon it. The specifics of the copy operation are handled by the gnttab_copy invocation. In this function, the appropriate bounds and error checking is performed before memcpy'ing the entry into a local page. Thus, duplicating the page into a completely controlled space. Meaning, both reads and writes can be done. This operation was heavily used in the Linux netback driver to copy data to and from a DomU via the grant references in the RX and TX buffers.

4. XEN NIDS ANALYSIS

Looking at the network flow, a few areas arise for NIDS placement: at the virtual network, within the netback driver, within the netfront driver(s), and within the hypercall operations. We evaluate each approach with regards to the traffic they can capture, the level of security, and the compatibility to any possible Xen implementation. A generalized summary of this evaluation can be found in Table 1.

For NIDS deployment, the virtual network is a logical

Table 1: NIDS Locale Evaluation

Locale	Traffic	Security	Compatibility
Virtual Network	External to VM, VM to External	Moderate	Moderate
Netback Driver	External to VM, VM to External	Moderate	Moderate
Netfront Driver	All	Low	Low
Hypercall Operations	All Except Passthrough	High	High

choice as it sits between the netback drivers. Thus, when using specific netback implementations, a NIDS is guaranteed to capture all network traffic. However, special drivers can do direct copies between DomUs, removing the need to push packets onto the virtual network [7]. Meaning, VM to VM traffic is not guaranteed to be captured. Moreover, the NIDS would have to be built into a specific virtual network technology to be able to stop malicious packets rather than just identify them. As such, there is no guaranteed compatibility for any netback driver nor any virtual network technology when placing a NIDS at this locale. Fortunately, these two components are chosen by an administrator, so a NIDS at the virtual network is moderately compatible. As for security, it is dependent on the security of the domain the NIDS is running in. In most cases, this means Dom0 which is sandboxed from the guest domains. However, it runs on a significantly sized OS, which is inherently more susceptible to compromise. As such, a virtual network NIDS is only moderately secure since it runs in a moderately secure domain. Ultimately, the virtual network is a good choice for NIDS placement, especially considering the ease of implementing at this layer. However, convenience is not our concern; thus, we should look to another locale.

Another option is the netback driver, or specifically, in the netback driver after the skb seen in Listing 1 has been filled. Like the virtual network, a NIDS in the netback driver is not guaranteed to capture all network traffic. This is because DomU to DomU copying can avoid passing through the netback driver. An implementation like the one described in [7] covers such a process. Also, a netback NIDS can work with any existing virtual network technology but requires the use of a custom netback driver; therefore, an IDS implemented in a netback driver has moderate compatibility. As for security, a netback NIDS is moderately secure due to the exact reasoning given for a virtual network NIDS: it runs in a moderately secure domain. Again, we have not reached an ideal locale, so let us consider yet another one.

Implementing a NIDS within a netfront driver is not an ideal choice for one glaringly obvious reason: it runs in the guest domain. This makes it relatively insecure since it is within a user controlled domain. Also, this makes it relatively incompatible since it requires a special driver be created for every possible VM OS. However, a NIDS within a netfront driver does net one benefit that other locales do not: it captures packets that have direct passthrough to the physical layer. Granted, this means that the driver is less a netfront driver and more a network driver, but the logic still holds. Ultimately, this option is mentioned for completeness and to address the passthrough concern. Considering all of this, it should not be used (at least on its own); thus, we will evaluate the final locale.

Lastly, a NIDS can be implemented within the hypervisor by utilizing the aforementioned hypercalls. Specifically, the operations seen in Listings 3 and 4 can be targeted to capture packet data as it moves through Xen's shared memory mechanisms: the event channels and grant entries that are mapped to a netfront's TX and RX rings can be flagged (via IRQs and grant references respectively), and then any grant entries used by the flagged resources can be monitored via a NIDS. Such a system nets significant benefits in every factor of our evaluation: captures all traffic since networking runs on these shared memory mechanisms, remains compatible with VM software since the Xen interfaces have not been altered, and provides high security since it lies in the smallest portion of the trusted computing base, the hypervisor. Ultimately, a VMM-situated NIDS is ideal because it supports any Xen configuration, unlike the other options.

5. CONCLUSION AND FUTURE WORK

There needs to be significant effort towards securing cloud technologies, particularly virtualization. In this paper, we focused on the Xen VMM due to its prominence in the cloud world. For securing Xen, we analyzed its networking architecture and found that a NIDS system could be implemented within the VMM. Such an implementation provides significant benefits: captures all traffic, is highly compatible, and is highly secure. At present, we have implemented a grant entry filtering system that identifies network data within the VMM, and we are in the process of implementing an anomaly-based NIDS that analyzes grant entry byte-frequencies. Going forward, we plan to refine this system by having the training phase tune to each VM image's traffic. With this approach, a robust, efficient, and securely isolated NIDS will be constructed within Xen.

6. REFERENCES

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177, Oct. 2003.

[2] S. Bharadwaja, W. Sun, M. Niamat, and F. Shen. Collabra: a xen hypervisor based collaborative intrusion detection system. In *IEEE Information technology: New generations (ITNG)*, 2011.

[3] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic. Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Elsevier FGCS*, 25(6), 2009.

[4] M. Laureano, C. Maziero, and E. Jamhour. Intrusion detection in virtual machine environments. In *IEEE Euromicro Conference*, 2004.

[5] H.-J. Liao, C.-H. R. Lin, Y.-C. Lin, and K.-Y. Tung. Intrusion detection system: A comprehensive review. *Journal of Network and Computer Applications*, 36(1):16–24, 2013.

[6] C. Mazzariello, R. Bifulco, and R. Canonico. Integrating a network ids into an open source cloud computing environment. In *IEEE Information Assurance and Security*, 2010.

[7] J. Wang, K.-L. Wright, and K. Gopalan. Xenloop: a transparent high performance inter-vm network loopback. In *ACM international symposium on High performance distributed computing*, 2008.

[8] T. Xing, D. Huang, L. Xu, C.-J. Chung, and P. Khatkar. Snortflow: A openflow-based intrusion prevention system in cloud environment. In *IEEE Research and Educational Experiment Workshop*, 2013.