A Hypervisor Based Security Testbed

Dan Duchamp and Greg DeAngelis

Computer Science Department
Stevens Institute of Technology
Hoboken, NJ 07030 USA
djd@cs.stevens.edu, gdeangel@stevens.edu

ABSTRACT

We are developing an experimental testbed intended to help support security research. The testbed allows a network of unmodified hosts, running any of several of unmodified operating systems, to execute in a controlled and reproducible manner. The network is implemented on a hypervisor that is instrumented to observe and control security-relevant events. These events are securely logged to a relational database for later analysis.

1. INTRODUCTION

We are developing an experimental testbed intended to help support security research. The testbed allows a network of unmodified hosts to be subjected to security attacks in a controlled, observable, and reproducible manner using hypervisor technology.

The two most notable characteristics of our testbed are the use of a virtual machine monitor (VMM, aka hypervisor) as a tool for gathering information over an entire heterogeneous LAN, and a highly generalized infrastructure for the description, measurement, and analysis of experiments.

Unmodified guest host software (operating system and applications) runs on an instrumented virtual machine monitor. The VMM provides a powerful tool for observing and controlling the behavior of the guest software. Measurement points added to the hypervisor allow the experimenter to log the occurrence of key security-relevant events such as a system call or a buffer overflow. When our work is complete, the experimenter will be able not only to log but also to control guest host behavior via the measurement points; for example, the hypervisor could intercept and verify a system call before allowing it to proceed.

The VMM not only emulates x86 hardware, it also implements a virtual Ethernet. Emulating both hosts and network on the same hypervisor provides experimenters with a more complete view of how network-based attacks spread.

Furthermore, the idea of implementing a LAN and all its hosts on a single virtual machine might also prove to be a superior way to deploy an intrusion detection system (IDS) on a real world network that is small and highly vulnerable, such as a honeypot [13] or a DMZ.

Conventional approaches to intrusion detection are either host-based or network-based. A host-based IDS monitors computing activity on a host and raises an alarm when suspicious activity is observed. One weakness of a host-based IDS is that it places the IDS in the same domain of execution as the vulnerable machine, exposing the IDS to whatever happens when the machine is compromised. Another weakness is that intrusion detection logic is applied to only a single host. To protect an entire installation using host-based IDS software, each host must run the IDS and anomalous activity must be detectable on a per-host basis. Attacks that manifest across several hosts may not be detected by a host-based IDS approach. In contrast, a network IDS sniffs packets, reconstructs "flows" from the constituent packets, then applies intrusion detection logic to each flow. One weakness of a network-based IDS is that flow reconstruction can be very difficult, especially if—as is often the case—an attack intentionally includes nonsense packets specifically intended to confuse or trick the recipient. A classic example of an attack containing nonsense packets is when two TCP segments contain overlapping data; e.g., one segment contains bytes numbered 100-300, while the next segment contains bytes numbered 200-400. It is implementation-dependent whether TCP will accept bytes 200-300 from the first segment or the second. A network IDS must first reconstruct the flow that will be seen by the host under attack, then analyze the flow for anomalies or suspicious content. In the case of overlapping TCP segments, merely to reconstruct the flow the network-based IDS must be aware of the host's exact TCP state and even of the host's TCP implementation details. Host-based and network-based intrusion detection approaches each suffer from their own blind spot. A security testbed facility based on virtual machine (VM) technology need not have a blind spot.

Having a system-wide view permits capture and recording of a timestamped "movie" of activity in all parts of the system during an attack. Such movies can be an-

alyzed off-line afterward to discover attack signatures that could then serve as input to standard IDSs. Movies could also be stored and replayed later to reproduce an attack on demand.

The main components of our testbed are:

1. Hardware base

2. Infrastructure for experiment description and configuration

3. Support for measurement and logging during experiments

Each component is discussed in its own section below.

2. HARDWARE BASE

Our current hardware base is shown in Figure 1. Experiments run on five real machines, each of which has two dual-core Xeon 5060 (Dempsey) processors with Intel Virtualization Technology (IVT) extensions [8, 6] for virtualization and 8GB of RAM. IVT provides `VMentry` and `VMexit` state transitions that manage where interrupts and system calls are directed. Certain instructions must be trapped within a guest host and forwarded to the hypervisor. Access to I/O devices is handled as described below.

The virtualization software is KVM [10, 11], a loadable module for Linux versions 2.6.20 and later. KVM leverages IVT to implement a virtual machine monitor with much less code than traditional hypervisors like VMWare [15], Xen [2], etc. KVM expands the traditional two modes of a UNIX process (user and system) to three: user, system, and guest. When guest software runs in such a 3-state process, I/O instructions execute in user mode at the Linux privilege level of the user who owns the guest host process. Non-I/O code runs in guest mode, and system mode is used only to transition among modes and for special instructions.

Linux/KVM can host any unmodified x86-based operating system. Therefore, our testbed supports all operating systems of interest to us: Windows, Linux, and NetBSD. Support for heterogeneity allows the execution of more varied and realistic experiments.

Because Linux exposes dual dual-core Xeon processors with hyperthreading enabled as 8 virtual processors, each hypervisor is configured to support 8 guest hosts by default. So the overall facility supports a minimum of 40 guest hosts. However, the hypervisors can be configured to support more than 8 guest hosts each, so larger virtual networks are possible. Memory is the resource most likely to limit the number of virtual hosts that can be supported.

Each real machine has 3 Ethernet ports that are all attached to a Cisco Catalyst 3750 switch. The switch implements three types of network. A "management" network connects the 5 real machines to a control server (to be described below). The management network is used exclusively to control the physical machines in limited ways such as rebooting them and initializing their disks. A second "control" network is used exclusively for logging. Events logged by each virtual machine are transmitted over the control network to a database server (to be described below).

The third type of network implemented by the switch is the "data" network. Guest hosts use only the data network for their traffic. To form networks of guest hosts, each guest operating system is given a virtual network interface. Each virtual network interface is connected to a software bridge running on the physical machine. The software bridge also connects to its machine's physical Ethernet adapter for the data network. A combination of the settings on the software bridges on each machine and the VLAN settings of the Catalyst switch allow guest hosts on the same or different physical machines to be grouped arbitrarily into virtual networks. Traffic for the virtual network passes through either one or two hypervisor(s) and therefore can be monitored.[1]

Besides the five physical machines that support guest hosts for experiments, two physical machines provide support functions. One machine is used as a database. During experiments, it collects records about security-relevant events produced by measurement points within KVM and transmitted over the control network. After queueing, these records are stored in a MySQL database. The database machine also acts as a disk image server. We initialize the disk images of the physical machines when a complete physical reset is necessary.

The final machine shown in Figure 1 is the control host. Users connect to this machine to configure the system in preparation for an experiment, configure and run experiments, and perform post-experiment analysis of events logged in the database.

The database server and control host are connected to only the control and management networks, keeping them physically isolated from the dangerous data network.

All this hardware is separated from the campus network and Internet by a Cisco ASA 5510 firewall. The firewall allows connections between the outside and the control host so that users can enter the testbed, set up and run their experiments.

Experiments running on the data network can also be granted access to the outside when appropriate. Not every experiment will involve dangerous code; the testbed can be used for non-security experiments as well as for security experiments. The firewall supports 100

[1]Traffic between guest hosts passes through one hypervisor if the two guest hosts are running on the same physical machine; otherwise, the two guest hosts are running on different physical machines so their traffic passes through two hypervisors.

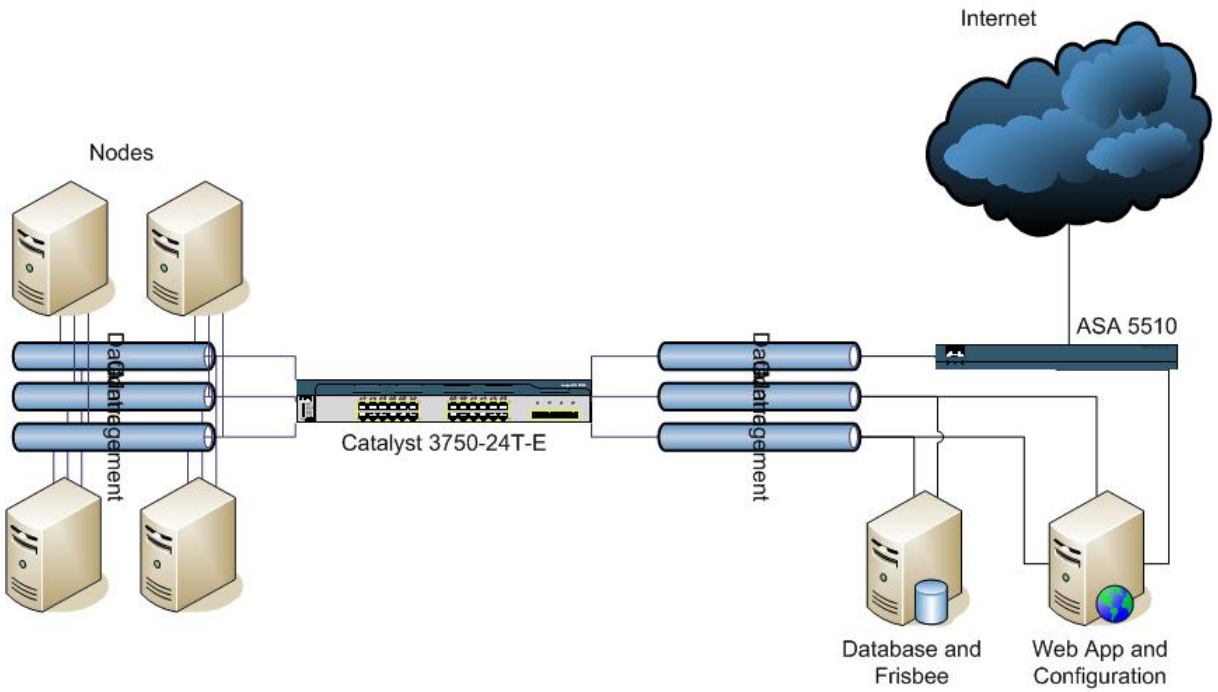

Figure 1: The hardware base includes five physical machines dedicated to experimental software. Each machine runs the Linux/KVM hypervisor and is capable of supporting several guest hosts. Each of these machines is connected to three networks, two of which are reserved for control and management. A Catalyst switch helps to connect guest hosts together on VLANs. Two support machines are dedicated for configuration and data storage, respectively. The entire facility lies behind a firewall.

VLANs. Therefore, if an experiment requires access to the outside—and the experiment is deemed trustworthy—then the firewall can join the VLAN allocated to a virtual network of guest hosts. In this way, guest hosts can access the outside through the firewall. The firewall configuration is part of the definition of an experiment.

3. SYSTEM AND EXPERIMENT CONFIGURATION

To set up and run an experiment, a user connects to the control host and runs a web application, cleverly named "WebApp." WebApp allows the user to set up, initiate, monitor, and post-analyze the experiment. The setup process is suggested by Figure 2 while experiment execution is suggested by Figure 3.

The first step is to configure the software on the subset of virtual nodes necessary for the experiment. For this purpose, we are developing a tool for virtual node configuration that is a "generalized package manager" (GPM) that can interoperate with popular Linux package managers such as RPM, portage, and apt. The GPM is not a single cross-OS package manager like Autopackage [1]. Instead, it is much simpler—it translates the user's specifications into calls to the native package manager of the underlying Linux distribution. GPM

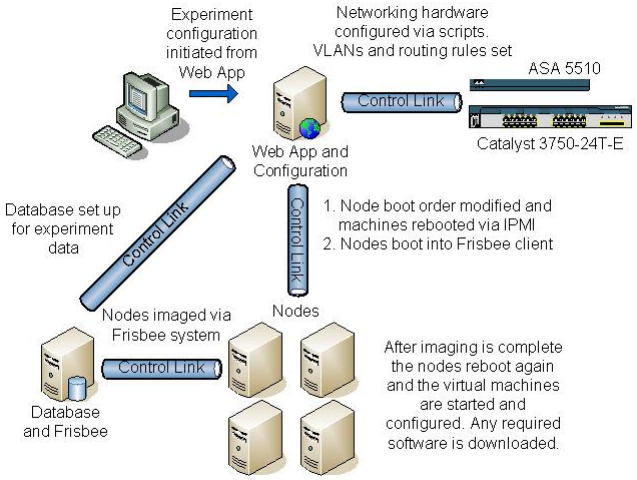

Figure 2: The experiment is configured by a user employing a modified version of Emulab's Netbuild client to specify network connections and our own Generalized Package Manager to specify guest software. The experiment description is compiled into XML and stored in the database. If necessary, the physical machines are given a clean disk image.

allows a user to load and configure both the kernel and application software for any Linux-based virtual node. In the future, we hope to configure Windows guest hosts in the same way if a package manager for Windows becomes available (e.g., WPM [12]).

The second step in setting up an experiment is to specify the mapping of virtual hosts to virtual LANs. For this step, we use a modified version of Emulab's "Netbuild" client in which nodes, links between them, and link properties are specified visually. A significant difference, however, is that our experiment configuration is compiled into XML and the XML is then loaded into the database. The reason for doing so is to create a machine readable permanent record of the experimental setup that can be associated with the data it produces, which will also be stored in the database. Once in the database, an experiment and its data can be analyzed and/or archived.

There are two ways to prepare virtual host software in advance of an experiment. One way is to reconfigure with the package manager as described above. However, if there is reason to suspect that either or both of the guest host or the hypervisor has been compromised or damaged, a second and more complete preparation can be performed using IPMI [5]. This preparation initializes an entire physical machine, not just a virtual host.

Our physical machines are each equipped with an IPMI card. IPMI allows a machine to be remotely controlled even if it is not yet running an operating system.

To initiate a clean-slate experiment, a physical machine is booted using IPMI commands sent over the management network. The physical machine boots up into the disk image client which then loads a virgin disk image containing the hypervisor operating system (i.e., Linux with KVM extensions). After receiving a complete clean disk image, the physical machine reboots, this time coming up running the Linux/KVM hypervisor.

Once KVM is running, virtual hosts are booted in the order specified as part of experiment configuration. Virtual hosts start application software following their usual procedures; e.g., `/etc/rc`, `cron` and the like. The sequencing of application startup, creation of background traffic, placement of innocent and malicious software, etc.—all this is the responsibility of the experimenter.

We plan to eventually add a hypervisor checkpoint facility so that an experimenter can pause a guest host and save its state. Being able to run a guest host from a prior saved state can benefit experiments that require a lengthy and/or complicated setup of the state that immediately precedes an attack.

4. MEASUREMENT AND LOGGING

Measurement points exist inside the hypervisor to report the occurrence of relevant low level events. Since KVM is implemented as a loadable module, we are developing a library of modules, each of which provides a different set of measurement points. The experimenter can select from among these modules, or he can write his own. Part of the experiment description is a specification of which measurement points should be activated, how often they should report, and what is the format of the log record that describes the event.

Logged events are queued within the hypervisor before being transmitted to the database server over the control network. Once at the database server, they are again queued there before being entered into the MySQL database. The purpose of queueing at both ends is to provide buffering to absorb variations in load.

One consequence of compiling the experiment description into XML is that the database holds an XML description of the format of log records produced by each type of monitored event. This allows XSLT to translate an XML description of a group of logged variables into typed headers of functions in a variety of languages.

For example, this XML description

```
<target name="target1">
    <value name="param1" type="integer" />
    <value name="param2" type="float" />
    <value name="param3" type="double" />
</target>
```

is translated into

```
void target1(int param1, float param2,
                         double param3) {
    // C or Java implementation
}
```

for C or Java and

```
def target1(param1, param2, param3):
    // Python implementation
```

for Python.

The function implementation is responsible for sending a message to the local queueing daemon. Messages are tagged with the time, target name, and experiment name so that the message can be unwrapped and inserted into the "`ExperimentName.TargetName`" table in the relational database.

Presently, events are only logged. We plan to allow future modules to invoke policy modules so as to control the guest host.

The most difficult and least mature part of this work is how to relate low level events that the hypervisor can easily observe to high level actions that would be relevant to intrusion detection. Mapping low level events (such as system calls) and facts (such as the contents of a region of memory) to a security-relevant high level

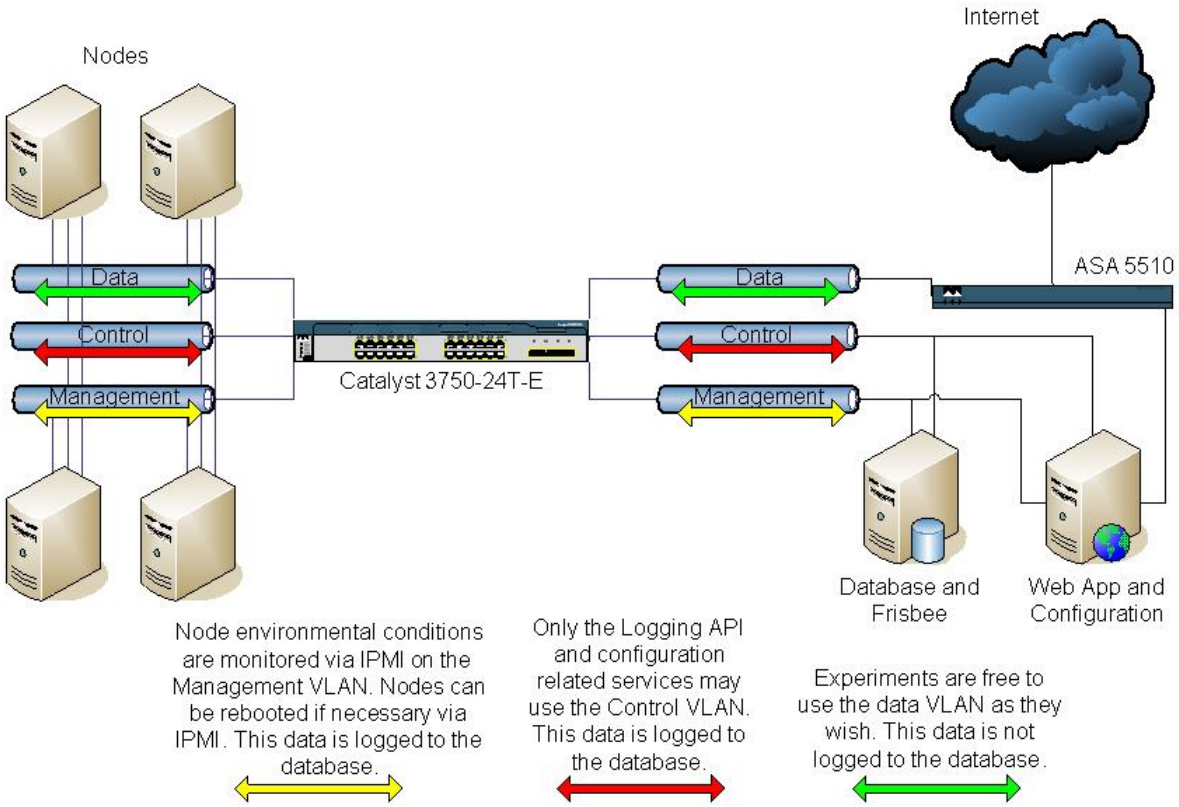

Figure 3: The data network is reserved for experiments. Several experiments can be in progress simultaneously by configuring the Catalyst switch to assign separate VLANs to separate groups of virtual hosts. The control network is reserved for log records produced by measurement points within the hypervisor and sent to the database server. The management network is reserved for IPMI-based reset, initialization, and monitoring of the physical hardware.

conclusion (such as that a buffer overflow occurred) requires non-trivial knowledge about how the guest operating system and guest applications use memory. This knowledge, encoded in data structures of the guest operating system, is complex and is subject to corruption when the guest operating system is compromised during an attack.

Previous work—most notably Livewire [4]—tackled this problem by implementing an interface between the hypervisor and an out-of-hypervisor intrusion detection module that had detailed knowledge about the guest operating system. The Livewire interface allowed the external IDS module to give commands to the hypervisor to inspect and monitor guest host state.

In contrast, we are designing a callout mechanism that would permit an out-of-hypervisor IDS (i.e., Linux process that is not supporting a guest host) to register callout criteria with the hypervisor, in much the same way that processes can register packet filters with the Berkeley Packet Filter (BPF) [7]. Whenever a callout's criteria are met the hypervisor will pause the guest host and transfer control to the external IDS so that it can inspect the guest host's memory. This approach isolates guest-host-specific knowledge in the external IDS, leaving the hypervisor largely ignorant of any of the guest host's implementation details.

Until such time as the callout-based control mechanism is working, users can log (only) events such as system call, interrupt service, and I/O instructions (especially packet transmit/receive).

5. SUMMARY

We are developing an experimental testbed intended to help support security research. The testbed allows a network of unmodified hosts to execute on a hypervisor that is instrumented to observe and eventually to control security-relevant events occurring across the network. As much as possible, experimental parameters are compiled into XML and recorded in order to ease repeatability, control, and analysis.

6. ACKNOWLEDGEMENT

This work was supported in part by the U.S. Army Armament Research, Development and Engineering Cen-

ter (ARDEC) under contract W15QKN-05-D-0011, Task Order 12.

7. REFERENCES

[1] Autopackage development team. *Autopackage: Easy Linux Software Installation.* http://autopackage.org

[2] P. Barham et al. *Xen and the Art of Virtualization.* in Proc. 19th ACM Symp. on Operating Systems Principles (SOSP '03), October 2003.

[3] T. Benzel et al. *Experience with DETER: A Testbed for Security Research.* in Proc. 2006 IEEE TridentCom, March 2006

[4] T. Garfinkel and M. Rosenblum *A Virtual Machine Introspection Based Architecture for Intrusion Detection.* in Proc. Network and Distributed Systems Security Symposium, February 2003.

[5] Intel Corp. *Intelligent Platform Management Interface.* http://www.intel.com/design/servers/ipmi

[6] Intel Corp. *Intel Trusted Execution Technology.* http://download.intel.com/technology/security/downloads/31516803.pdf

[7] S McCanne and V. Jacobson *The BSD Packet Filter: A New Architecture for User-level Packet Capture.* in Proc. Winter USENIX Technical Conf., pp. 259–270, 1993.

[8] G. Neiger et al. *Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization.* Intel Technology Journal 10(3), August 2006. http://developer.intel.com/technology/itj/index.htm

[9] M. Ott et al. *ORBIT Testbed Software Architecture: Supporting Experiments as a Service.* in Proc. 2005 IEEE TridentCom, Feb. 2005

[10] Qumranet Corp. *KVM Wiki.* http://www.qumranet.com/kvmwiki

[11] Qumranet Corp. *KVM: Kernel-based Virtualization Driver.* http://www.qumranet.com/wp/kvm_wp.pdf

[12] E. Ropple *WPM—Windows Package Manager.* http://blacken.superbusnet.com/oss/wpm

[13] K. Seifried *Honeypotting with VMWare—basics* http://www.seifried.org/security/ids/20020107-honeypot-vmware-basics.html

[14] M. Singh et al. *ORBIT Measurements Framework and Library (OML): Motivations, Design, Implementation and Features.* in Proc. 2005 IEEE TridentCom, Feb. 2005

[15] J. Sugerman et al. *Virtualizing I/O Devices on VMWare Workstation's Hosted Virtual Machine Monitor.* in Proc. USENIX 2001.

[16] B. White et al. *An Integrated Experimental Environment for Distributed Systems and Networks.* in Proc. 5th USENIX Symp. on Operating Systems Design and Implementation (OSDI '02), October 2002.