Tamper-Resistant, Application-Aware Blocking of Malicious Network Connections

Abhinav Srivastava and Jonathon Giffin

School of Computer Science, Georgia Institute of Technology
{abhinav,giffin}@cc.gatech.edu

Abstract. Application-level firewalls block traffic based on the process that is sending or receiving the network flow. They help detect bots, worms, and backdoors that send or receive malicious packets without the knowledge of users. Recent attacks show that these firewalls can be disabled by knowledgeable attackers. To counter this threat, we develop VMwall, a fine-grained tamper-resistant process-oriented firewall. VMwall's design blends the process knowledge of application-level firewalls with the isolation of traditional stand-alone firewalls. VMwall uses the Xen hypervisor to provide protection from malware, and it correlates TCP or UDP traffic with process information using virtual machine introspection. Experiments show that VMwall successfully blocks numerous real attacks—bots, worms, and backdoors—against a Linux system while allowing all legitimate network flows. VMwall is performant, imposing only a 0–1 millisecond delay on TCP connection establishment, less than a millisecond delay on UDP connections, and a 1–7% slowdown on network-bound applications. Our attack analysis argues that with the use of appropriate external protection of guest kernels, VMwall's introspection remains robust and helps identify malicious traffic.

Key words: Firewall, virtual machine introspection, attack prevention

1 Introduction

Application-level firewalls are an important component of a computer system's layered defenses. They filter inbound and outbound network packets based on an access policy that includes lists of processes allowed to make network connections. This fine-grained filtering is possible because application-level firewalls have a complete view of the system on which they execute. In contrast, network- or host-level firewalls provide coarse-grained filtering using ports and IP addresses. Application-level firewalls help detect and block malicious processes, such as bots, worms, backdoors, adware, and spyware, that try to send or receive network flows in violation of the fine-grained policies. To be successful, these firewalls must be fast, mediate all network traffic, and accurately identify executing processes.

The conventional design of application-level firewalls has a deficiency that may prevent filtering of malicious traffic. The architectures pass packet information from a kernel-level network tap up to a user-level firewall process that

executes alongside malicious software. The firewall is both performant and able to identify the processes attached to a network flow, but it is exposed to direct attack by any malicious software aware of the firewall. Baliga et al. [1] demonstrated the ease of such attacks by manipulating the netfilter framework inside the Linux kernel to remove the hooks to packet filtering functions. Similarly, attackers can disable the Windows Firewall by halting particular services normally running on the system. Once the firewall fails, then all network traffic will be unmediated and the malware can send and receive data at will.

An alternative design isolates firewalls from vulnerable systems to gain protection from direct attack. Virtual machines allow construction of firewall appliances that execute outside of operating systems under attack. Such firewalls dispense with application-level knowledge and filter inbound and outbound packets using coarse-grained rules over IP addresses and port numbers. Attacks can easily evade these firewalls by using allowed ports directly or via tunneling.

This paper leverages the benefits of both application-level firewalls and virtual machine isolation to develop tamper-resistant application-oriented firewalls. Such a firewall needs good visibility of the system so that it can correlate network flows with processes, but it also needs strong isolation from any user-level or kernel-level malware that may be present. We architect an application-level firewall resistant to direct attack from malicious software on the system. Our design isolates the application-level firewall in a trusted virtual machine (VM) and relies on the hypervisor to limit the attack surface between any untrusted VM running malware and the trusted VM. Our firewall, executing in the trusted VM, becomes an application-level firewall by using virtual machine introspection (VMI) [10] to identify the process in another VM that is connected to a suspicious network flow.

Our prototype implementation, VMwall, uses the Xen [2] hypervisor to remain isolated from malicious software. VMwall executes entirely within Xen's trusted virtual machine dom0; it operates with both paravirtualized and fully virtualized domains. A dom0 kernel component intercepts network connections to and from untrusted virtual machines. A user-space process performs introspection to correlate each flow to a sending or receiving process, and it then uses a predefined security policy to decide whether the connection should be allowed or blocked. Policies are straightforward whitelists of known software in the untrusted VM allowed to communicate over the network. To correlate network flows with processes, VMwall's user-space component maps the untrusted operating system's kernel memory into its own address space and uses programmed knowledge of kernel data structures to extract the identity of the process attached to the flow.

VMwall is effective at identifying and blocking malicious network connections without imposing significant performance degradation upon network traffic. Using a Linux system and a collection of known attacks that either send or receive network traffic, we show that VMwall identifies all malicious connections immediately when the first packet is sent or received. In particular, VMwall blocked 100% of the malicious connections when tested against bots, worms, and

backdoors, and it correctly allowed all legitimate network traffic. In our design, VMwall only performs introspection for the first packet of a new connection, so network performance remains high. Our tool adds only about 0–1 milliseconds of overhead to the first packet of a session. This is a latency cost to network connection creation that will not impact the subsequent data transfer of legitimate connections.

VMwall looks into the state of the untrusted operating system's memory to find the process bound to a network connection. The system monitors network flows, and it is not an intrusion detection system designed to detect an attack against the OS. Hence, an attacker may try to evade VMwall either by hijacking a process or by subverting the inspected kernel data structures. In Sect. 6.4, we study this problem, provide an in-depth security analysis of VMwall, and suggest appropriate measures to thwart these attacks.

We believe that our tamper-resistant application-oriented firewall represents an appropriate use of virtualization technology for improved system security. We feel that our paper provides the following contributions:

- Correlation between network flows and processes from outside the virtual machine (Sect. 4).
- VMwall, an implementation of a tamper-resistant application-oriented firewall (Sect. 5).
- Evidence that application-aware firewalls outside the untrusted virtual machine can block malicious network connections successfully while maintaining network performance (Sect. 6).

2 Related Work

Prior research has contributed to the development of conventional host-based firewalls. Mogul et al. [21] developed a kernel-resident packet filter for UNIX that gave user processes flexibility in selecting legitimate packets. Venema [29] designed a utility to monitor and control incoming network traffic. These traditional firewalls performed filtering based on restrictions inherent in network topology and assumed that all parties inside the network were trusted. As part of the security architecture of the computer system, they resided in kernel-space and user-space, and hence were vulnerable to direct attack by malicious software.

Administration of firewalls can be cumbersome, and distributed firewalls have been proposed to ease the burden [3,15]. In distributed firewalls, an administrator manages security policies centrally but pushes enforcement of these policies out to the individual hosts. Although we have not implemented support for distributed management, we expect VMwall to easily fit into this scheme. VMwall policies dictate which processes can legitimately make use of network resources. In a managed environment where administrators are knowledgeable of the software running on the machines in the local network, preparing and distributing VMwall policies from a central location may be an appealing solution.

The recent support for virtual machines by commodity hardware has driven development of new security services deployed with the assistance of VMs [9,

27, 30]. Garfinkel et al. [11] showed the feasibility of implementing distributed network-level firewalls using virtual machines. In another work [10], they proposed an intrusion detection system design using virtual machine introspection of an untrusted VM. VMwall applies virtual machine introspection to a different problem, using it to correlate network flows with the local processes bound to those flows.

Other research used virtual machines for malware detection. Borders et al. [4] designed a system, Siren, that detected malware running within a virtual machine. Yin et al. [33] proposed a system to detect and analyze privacy-breaching malware using taint analysis. Jiang et al. [17] presented an out-of-the-box VMM-based malware detection system. Their proposed technique constructed the internal semantic views of a VM from an external vantage point. In another work [16], they proposed a monitoring tool that observes a virtual machine based honeypot's internal state from outside the honeypot. As a pleasant side-effect of malicious network flow detection and process correlation, VMwall can often identify processes in the untrusted system that comprise portions of an attack.

Previous research has developed protection strategies for different types of hardware-level resources in the virtualized environment. Xu et al. [32] proposed a VMM-based usage control model to protect the integrity of kernel memory. Ta-Min et al. [28] proposed a hypervisor based system that allowed applications to partition their system call interface into trusted and untrusted components. VMwall, in contrast, protects network resources from attack by malware that runs inside the untrusted virtual machine by blocking the illegitimate network connections attempts.

These previous hypervisor-based security applications generally take either a network-centric or host-centric view. Our work tries to correlate activity at both levels. VMwall monitors network connections but additionally peers into the state of the running, untrusted operating system to make its judgments about each connection's validity. Moreover, VMwall easily scales to collections of virtual machines on a single physical host. A single instance of VMwall can act as an application-level firewall for an entire network of VMs.

3 Overview

We begin with preliminaries. Section 3.1 explains our threat model, which assumes that attackers have the ability to execute the real-world attacks infecting widespread computer systems today. Section 3.2 provides a brief overview of Xen-based virtual machine architectures and methods allowing inspection of a running VM's state.

3.1 Threat Model

We assume that attackers have abilities commonly displayed by real-world attacks against commodity computer systems. Attackers can gain *superuser privilege from remote*. Attackers are external and have no physical access to the

attacked computers, but they may install malicious software on a victim system by exploiting a software vulnerability in an application or operating system or by enticing unsuspecting users to install the malware themselves. The software exploit or the user often executes with full system privileges, so the malware may perform administrative actions such as kernel module or driver installation. Hence, malicious code may execute at both user and kernel levels. For ease of explanation, we initially describe VMwall's architecture in Sect. 4 under the assumption that kernel data structure integrity is maintained. This assumption is not valid in our threat model, and Sect. 6.4 revisits this point to describe technical solutions ensuring that the assumption holds.

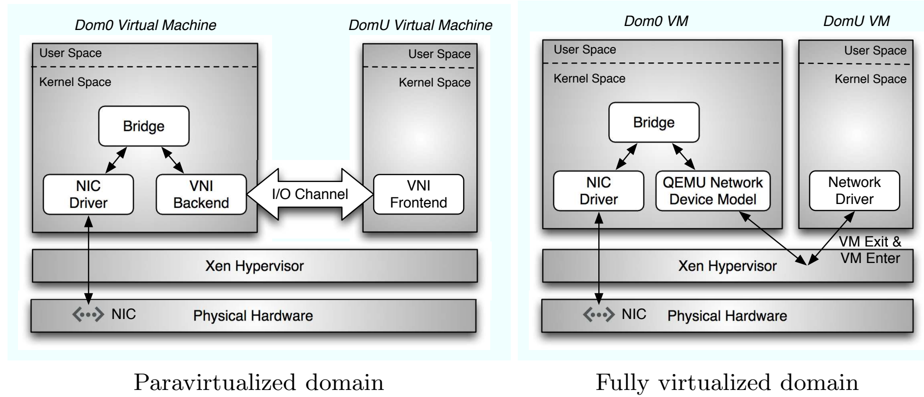
The installed malware may periodically make or receive network connections. Many examples exist. *Bots* make network connections to a command and control channel to advertise their presence and receive instruction, and they send bulk network traffic such as denial-of-service packets and email spam. *Spyware* programs collect information, such as keystrokes and mouse clicks, and then transmit the confidential data across a network to the attacker. *Worms* may generate network connections to scan the network in search of additional victims suitable for infection. *Backdoors* open holes in machines by listening for incoming connections from the attacker. One common feature of these different classes of attacks is their interest in the network.

In a typical system, malware can directly affect an application-level firewall's execution. The architecture of these malware instances frequently combines a user-level application performing network activity with a kernel-level module that hides the application from the view of host-level security software. The malicious application, likely running with full system privileges, may halt the execution of the firewall. Similarly, the malicious kernel component may alter the hooks used by an in-kernel module supporting the user-level firewall so that the firewall is simply never invoked as data passes to and from the network. Conventional application-level firewalls fail under these direct attacks. Our goal is to develop a system that withstands direct attack from malware at the application layer or the kernel layer.

Our system has requirements for correct execution. As with all requirements, an attacker who is able to violate any requirement is likely able to escape detection. Our two requirements of note center on basic expectations for the in-memory data structures used by the kernel that may be infected by an attack.

First, we expect to be able to find the head of linked data structures, often by extracting a kernel symbol value at boot time. An attacker could conceivably cause our firewall to inspect the incorrect kernel information by replicating the data structure elsewhere in kernel memory and by altering all code references to the original structure to instead refer to the new structure. Our firewall would then analyze stale data. It is not immediately clear that such an attack is plausible; moreover, our tool could periodically verify that code references to the data match the symbol value extracted at boot.

Second, we expect that attacks do not alter the ordering or length of fields in aggregate data structures. Our firewall is preprogrammed with type information



Paravirtualized domain   Fully virtualized domain

Fig. 1. Xen networking architecture.

about kernel structures, and an attack that alters the structure types would cause our system to read incorrect information from kernel memory. Successfully executing this attack without kernel recompilation appears to be complex, as all kernel code that accesses structure fields would need to be altered to use the attacker's structure layout. As a result, we believe that relying upon known structure definitions is not a limiting factor to our design.

3.2 Virtual Machine Introspection

Our design makes use of virtual machine technology to provide isolation between malicious code and our security software. We use Xen [2], an open source hypervisor that runs directly on the physical hardware of a computer. The virtual machines running atop Xen are of two types: unprivileged domains, called domU or guest domains, and a single fully-privileged domain, called dom0. We run normal, possibly vulnerable software in domU and deploy our application-level firewall in the isolated dom0.

Xen virtualizes the network input and output of the system. Dom0 is the device driver domain that runs the native network interface card driver software. Unprivileged virtual machines cannot directly access the physical network card, so Xen provides them with a virtualized network interface (VNI). The driver domain receives all the incoming and outgoing packets for all domU VMs executing on the physical system. Dom0 provides an Ethernet bridge connecting the physical network card to all virtual network devices provided by Xen to the domU VMs. (Xen offers other networking modes, such as network address translation, that are not used in our work and will not be considered further.) Dom0 uses its virtual bridge to multiplex and demultiplex packets between the physical network interface and each unprivileged virtual machine's VNI. Figure 1 shows the Xen networking architecture when the virtual machines' network interfaces are connected through a virtual Ethernet bridge. The guest VMs send and receive packets via either an I/O channel to dom0 or emulated virtual devices.

The strong isolation provided by a hypervisor between dom0 and the guest domains complicates the ability to correlate network flows with software executing in a guest domain. Yet, dom0 has complete access to the entire state of the guest operating systems running in untrusted virtual machines. *Virtual machine introspection* (VMI) [10] is a technique by which dom0 can determine execution properties of guest VMs by monitoring their runtime state, generally through direct memory inspection. VMI allows security software to remain protected from direct attack by malicious software executing in a guest VM while still able to observe critical system state.

Xen offers low-level APIs to allow dom0 to map arbitrary memory pages of domU as shared memory. XenAccess [31] is a dom0 userspace introspection library developed for Xen that builds onto the low-level functionality provided by Xen. VMwall uses XenAccess APIs to map raw memory pages of domU's kernel inside dom0. It then builds higher-level memory abstractions, such as aggregate structures and linked data types, from the contents of raw memory pages by using the known coding semantics of the guest operating system's kernel. Our application-level firewall inspects these meaningful, higher-level abstractions to determine how applications executing in the guest VM use network resources.

4 Tamper Resistant Architecture of VMwall

VMwall is our application-level firewall designed to resist the direct attacks possible in our threat model. The architecture of VMwall is driven by the following three goals:

- **Tamper Resistance:** VMwall should continue to function reliably and verify all network connections even if an attacker gains entry into the monitored system. In particular, the design should not rely on components installed in the monitored host as processes or kernel modules, as these have been points of direct attack in previous application-level firewalls.
- **Independence:** VMwall should work without any cooperation from the monitored system. In fact, the system may not be aware of the presence of the firewall.
- **Lightweight Verification:** Our intent is to use VMwall for online verification of network connections to real systems. The design should allow for efficient monitoring of network traffic and correlation to applications sending and receiving that traffic.

Our firewall design satisfies these goals by leveraging virtual machine isolation and virtual machine introspection. Its entire software runs within the privileged dom0 VM, and it hooks into Xen's virtual network interface to collect and filter all guest domains' network packets. Since the hypervisor provides strong isolation among the virtual machines, this design achieves the first goal of tamper-resistance.

In order to provide application-level firewalling, VMwall must identify the process that is sending or receiving packets inside domU. VMwall correlates

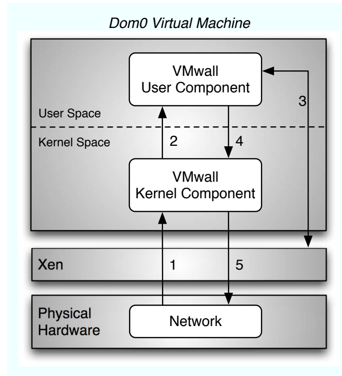


Fig. 2. VMwall's high-level architecture. (1) Packets inbound to and outbound from a guest domain are processed by dom0. (2) The VMwall kernel component intercepts the packets and passes them to a user-space component for analysis. (3) The user-space component uses virtual machine introspection to identify software in a guest domain processing the data. (4) The user-space component instructs the kernel component to either allow or block the connection. (5) Packets from allowed connections will be placed on the network.

packet and process information by directly inspecting the domU virtual machine's memory via virtual machine introspection. It looks into the kernel's memory and traverses the data structures to map process and network information. This achieves our second design goal of independence, as there are no components of VMwall inside domU. Our introspection procedure rapidly analyzes the kernel's data structures, satisfying the third goal of lightweight verification.

The high-level design of VMwall has two components: a kernel module and user agent, both in dom0 (Fig. 2). The VMwall kernel component enforces a per-packet policy given by the user agent and either allows or drops each packet. The user agent determines policy by performing introspection to extract information about processes executing in guest VMs and evaluating the legitimacy of those processes. Sections 4.1 and 4.2 present detailed information about the two components.

4.1 Kernel Component

VMwall's kernel component is a module loaded inside the dom0 Linux kernel. It intercepts all network packets to or from untrusted virtual machines and uses security policies to decide whether each packet should be allowed or dropped. Interception occurs by hooking into Xen's network bridge between the physical interface card and virtual network interface. When the kernel component intercepts a packet, it checks a rule table to see if a firewall rule already exists for the packet, as determined by the local endpoint IP address and port. If so, it takes the allow or block action specified in the rule. If there is no rule, then it invokes the VMwall user agent to analyze the packet and create a rule. The user agent performs introspection, generates a rule for the packet, and sends this rule back to the kernel module. The kernel module adds this new rule to its policy table and processes the packet. Further packets from the same connection are processed using the rule present in the kernel component without invoking the user agent and without performing introspection.

As kernel code, the kernel component cannot block and must take action on a packet before the user agent completes introspection. VMwall solves this

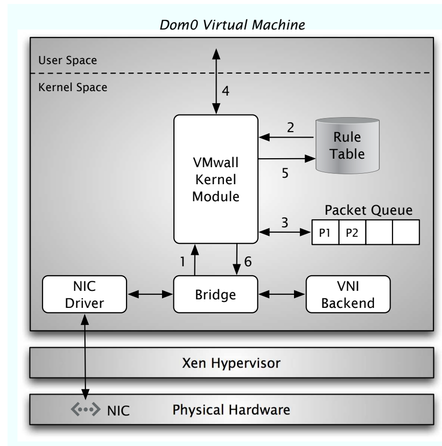


Fig. 3. VMwall's kernel module architecture. (1) Packets inbound to and outbound from a guest domain are intercepted and passed to the kernel module. (2) The module receives each packet and looks into its rule table to find the rule for the packet. (3) The kernel module queues the packet if there is no rule present. (4) VMwall sends an introspection request to the user agent and, after the agent completes, receives the dynamically generated rule for the packet. (5) The kernel module adds the rule into its rule table to process future packets from the same connection. (6) The kernel module decides based on the action of the rule either to accept the packet by reinjecting it into the network or to drop it from the queue.

problem for packets of unknown legitimacy by queuing the packets while waiting for the user agent's reply. When the user agent sends a reply, the module adds a rule for the connection. If the rule's action is to block the connection, then it drops all the packets that are queued. Otherwise, it re-injects all the packets into the network.

Figure 3 presents the kernel module's complete architecture. It illustrates the steps involved in processing the packet inside the kernel. It shows the queue architecture, where packets are stored inside the kernel during introspection.

4.2 User Agent

The VMwall user agent uses virtual machine introspection to correlate network packets and processes. It receives introspection requests from the kernel component containing network information such as source port, source IP address, destination port, destination IP address, and protocol. It first uses the packet's source (or destination) IP address to identify the VM that is sending (or receiving) the packet. When it finds the VM, it then tries to find the process that is bound to the source (or destination) port.

VMwall's user agent maps a network port to the domU process that is bound to the port, shown in Fig. 4. As needed, it maps domU kernel data structures into dom0 memory. Process and network information is likely not available in a single data structure but instead is scattered over many data structures. VMwall works in steps by first identifying the domU kernel data structures that store IP address and port information. Then, VMwall identifies the process handling this network connection by iterating over the list of running processes and checking each process to see if it is bound to the port. When it finds the process bound to the port, it extracts the process' identifier, its name, and the full path to its

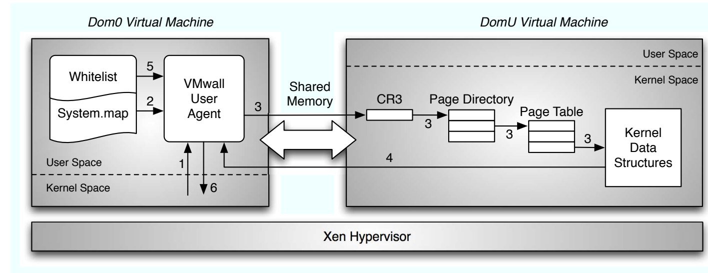


Fig. 4. VMwall's user agent architecture. (1) The VMwall user agent receives the introspection request. (2) The user agent reads the System.map file to extract the kernel virtual addresses corresponding to known kernel symbols. (3) The user agent uses Xen to map the domU kernel memory pages containing process and network data structures. (4) VMwall traverses the data structures to correlate network and process activity. (5) The agent searches for the recovered process name in the whitelist. (6) The user agent sends a filtering rule for the connection to the VMwall kernel module.

executable. If the user agent does not find any process bound to the port, it considers this to be an anomaly and will block the network connection.

VMwall uses information about the process to create a firewall rule enforceable by the kernel component. The user agent maintains a whitelist of processes that are allowed to make network connections. When the user agent extracts the name of a process corresponding to the network packet, it searches the whitelist for the same name. VMwall allows the connection if it finds a match and blocks the connection otherwise. It then generates a rule for this connection that it passes to the VMwall kernel component. This rule contains the network connection information and either an allow or block action. The kernel component then uses this rule to filter subsequent packets in this attempted connection.

5 Implementation

We have implemented a prototype of VMwall using the Xen hypervisor and a Linux guest operating system. VMwall supports both paravirtualized and fully-virtualized (HVM) Linux guest operating systems. Its implementation consists of two parts corresponding to the two pieces described in the previous section: the kernel module and the user agent. The following sections describe specific details affecting implementation of the two architectural components.

5.1 Extending ebtables

Our kernel module uses a modified ebtables packet filter to intercept all packets sent to or from a guest domain. Ebtables [7] is an open source utility that filters packets at an Ethernet bridge. VMwall supplements the existing coarse-grained firewall provided by ebtables. Whenever ebtables accepts packets based

on its coarse-grained rules, we hook the operation and invoke the VMwall kernel module for our additional application-level checks. We modified ebtables to implement this hook, which passes a reference to the packet to VMwall.

Ebtables does not provide the ability to queue packets. Were it present, queuing would enable filters present inside the kernel to store packets for future processing and reinjection back into the network. To allow the VMwall kernel module to queue packets currently under inspection by the user agent, we altered ebtables to incorporate packet queuing and packet reinjection features.

5.2 Accessing DomU Kernel Memory

VMwall uses the XenAccess introspection library [31] to accesses domU kernel memory from dom0. It maps domU memory pages containing kernel data structures into the virtual memory space of the user agent executing in the trusted VM. XenAccess provides APIs that map domU kernel memory pages identified either by explicit kernel virtual addresses or by exported kernel symbols. In Linux, the exported symbols are stored in the file named `System.map`. VMwall uses certain domU data structures that are exported by the kernel and hence mapped with the help of kernel symbols. Other data structures reachable by pointers from the known structures are mapped using kernel virtual addresses. The domU virtual machine presented in Fig. 4 shows the internal mechanism involved to map the memory page that contains the desired kernel data structure.

5.3 Parsing Kernel Data Structures

To identify processes using the network, VMwall must be able to parse high-level kernel data structures from the raw memory pages provided by XenAccess. Extracting kernel data structures from the mapped memory pages is a non-trivial task. For example, Linux maintains a doubly-linked list that stores the kernel's private data for all running processes. The head pointer of this list is stored in the exported kernel symbol `init_task`. If we want to extract the list of processes running inside domU, we can map the memory page of domU that contains the `init_task` symbol. However, VMwall must traverse the complete linked list and hence requires the offset to the `next` member in the process structure. We extract this information offline directly from the kernel source code and use these values in the user agent. This source code inspection is not the optimal way to identify offsets because the offset values often change with the kernel versions. However, there are other automatic ways to extract this information from the kernel binary if it was compiled with a debug option [18].

This provides VMwall with sufficient information to traverse kernel data structures. VMwall uses known field offsets to extract the virtual addresses of pointer field members from the mapped memory pages. It then maps domU memory pages by specifying the extracted virtual addresses. This process is performed recursively until VMwall traverses the data structures necessary to extract the process name corresponding to the source or destination port of a network communication. Figure 5 shows the list of the kernel data structures

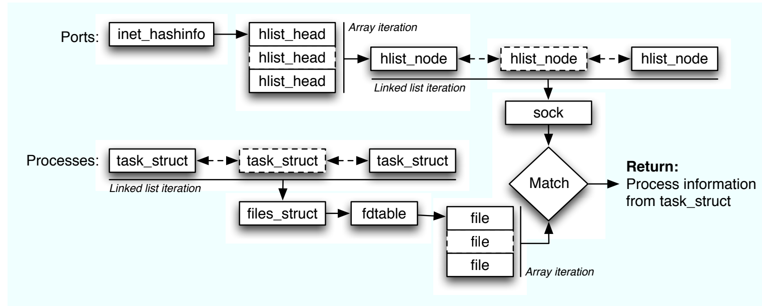


Fig. 5. DomU Linux kernel data structures traversed by the VMwall user agent during correlation of the process and TCP packet information.

traversed by the user agent to correlate a TCP packet and process information. First, it tries to obtain a reference to the socket bound to the port number specified in the packet. After acquiring this reference, it iterates over the list of processes to find the process owning the socket.

5.4 Policy Design and Rules

VMwall identifies legitimate connections via a whitelist-based policy listing processes allowed to send or receive data. Each process that wants to communicate over the network must be specified in the whitelist *a priori*. This whitelist resides inside dom0 and can only be updated by administrators in a manner similar to traditional application-level firewalls. The whitelist based design of VMwall introduces some usability issues because all applications that should be allowed to make network connections must be specified in the list. This limitation is not specific to VMwall and is inherent to the whitelist based products and solutions [6, 12].

VMwall's kernel module maintains its own rule table containing rules that are dynamically generated by the user agent after performing introspection. A rule contains source and destination port and IP address information, an action, and a timeout value used by the kernel module to expire and purge old rules for UDP connections. In the case of TCP connections, the kernel module purges TCP rules automatically whenever it processes a packet with the TCP `fin` or `rst` flag set. In an abnormal termination of a TCP connection, VMwall uses the timeout mechanism to purge the rules.

6 Evaluation

The basic requirement of an application-level firewall is to block connections to or from malicious software and allow connections to or from benign applications. We evaluated the ability of VMwall to filter out packets made by several different classes of attacks while allowing packets from known processes to pass

unimpeded. We tested VMwall against Linux-based backdoors, worms, and bots that attempt to use the network for malicious activity. Section 6.1 tests VMwall against attacks that receive inbound connections from attackers or connect out to remote systems. Section 6.2 tests legitimate software in the presence of VMwall. We measure VMwall's performance impact in Sect. 6.3, and lastly analyze its robustness to a knowledgeable attacker in Sect. 6.4.

6.1 Illegitimate Connections

We first tested attacks that receive inbound connections from remote attackers. These attacks are rootkits that install backdoor programs. The backdoors run as user processes, listen for connections on a port known to the attacker, and receive and execute requests sent by the attacker. We used the following backdoors:

- **Blackhole** runs a TCP server on port 12345 [22].
- **Gummo** runs a TCP server at port 31337 [22].
- **Bdoor** runs a backdoor daemon on port 8080 [22].
- **Ovas0n** runs a TCP server on port 29369 [22].
- **Cheetah** runs a TCP server at the attacker's specified port number [22].

Once installed on a vulnerable system, attacks such as worms and bots may attempt to make outbound connections without prompting from a remote attacker. We tested VMwall with the following pieces of malware that generate outbound traffic:

- **Apache-ssl** is a variant of the Slapper worm that self-propagates by opening TCP connections for port scanning [23].
- **Apache-linux** is a worm that exploits vulnerable Apache servers and spawns a shell on port 30464 [23].
- **BackDoor-Rev.b** is a tool that is be used by a worm to make network connections to arbitrary Internet addresses and ports [20].
- **Q8** is an IRC-based bot that opens TCP connections to contact an IRC server to receive commands from the botmaster [14].
- **Kaiten** is a bot that opens TCP connections to contact an IRC server [24].
- **Coromputer Dunno** is an IRC-based bot providing basic functionalities such as port scanning [13].

VMwall successfully blocked all illegitimate connections attempted by malware instances. In all cases, both sending and receiving, VMwall intercepted the first SYN packet of each connection and passed it to the userspace component. Since these malicious processes were not in the whitelist, the VMwall user space component informed the VMwall kernel component to block these malicious connections. As we used VMwall in packet queuing mode, no malicious packets were ever passed through VMwall.

Name	*Connection Type*	*Result*
rcp	Outbound	Allowed
rsh	Outbound	Allowed
yum	Outbound	Allowed
rlogin	Outbound	Allowed
ssh	Outbound	Allowed
scp	Outbound	Allowed
wget	Outbound	Allowed
tcp client	Outbound	Allowed
thttpd	Inbound	Allowed
tcp server	Inbound	Allowed
sshd	Inbound	Allowed

Table 1. Results of executing legitimate software in the presence of VMwall. "Allowed" indicates that the network connections to or from the processes were passed as though a firewall was not present.

6.2 Legitimate Connections

We also evaluated VMwall's ability to allow legitimate connections made by processes running inside domU. We selected a few network applications and added their name to VMwall's whitelist. We then ran these applications inside domU. Table 1 shows the list of processes that we tested, the type of connections used by the processes, and the effect of VMwall upon those connections. To be correct, all connections should be allowed.

VMwall allowed all connections made by these applications. The `yum` application, a package manager for Fedora Core Linux, had runtime behavior of interest. In our test, we updated domU with the `yum update` command. During the package update, `yum` created many child processes with the same name `yum` and these child processes made network connections. VMwall successfully validated all the connections via introspection and allowed their network connections.

6.3 Performance Evaluation

A firewall verifying all packets traversing a network may impact the performance of applications relying on timely delivery of those packets. We investigated the performance impact of VMwall as perceived by network applications running inside the untrusted virtual machine. We performed experiments both with and without VMwall running inside dom0. All experiments were conducted on a machine with an Intel Core 2 Duo T7500 processor at 2.20 GHz with 2 GB RAM. Both dom0 and domU virtual machines ran 32 bit Fedora Core 5 Linux. DomU had 512 MB of physical memory, and dom0 had the remaining 1.5 GB. The versions of Xen and XenAccess were 3.0.4 and 0.3, respectively. We performed our experiments using both TCP and UDP connections. All reported results show the median time taken from five measurements. We measured microbenchmarks with the Linux `gettimeofday` system call and longer executions with the `time` command-line utility.

VMwall's performance depends on the introspection time taken by the user component. Since network packets are queued inside the kernel during introspection, the introspection time is critical for the performance of the complete

Configuration	*TCP Introspection Time*	*UDP Introspection Time*
Inbound Connection to domU	251	438
Outbound Connection from domU	1080	445

Table 2. Introspection time (μs) taken by VMwall to perform correlation of network flow with the process executing inside domU.

system. We measured the introspection time both for incoming and outgoing connections to and from domU. Table 2 shows the results of experiments measuring introspection time.

It is evident that the introspection time for incoming TCP connections is very small. Strangely, the introspection time for outgoing TCP connections is notably higher. The reason for this difference lies in the way that the Linux kernel stores information for TCP connections. It maintains TCP connection information for listening and established connections in two different tables. TCP sockets in a listening state reside in a table of size 32, whereas the established sockets are stored in a table of size 65536. Since the newly established TCP sockets can be placed at any index inside the table, the introspection routine that iterates on this table from dom0 must search half of the table on average.

We also measured the introspection time for UDP data streams. Table 2 shows the result for UDP inbound and outbound packets. In this case, the introspection time for inbound and outbound data varies little. The Linux kernel keeps the information for UDP streams in a single table of size 128, which is why the introspection time is similar in both cases.

To measure VMwall's performance overhead on network applications that run inside domU, we performed experiments with two different metrics for both inbound and outbound connections. In the first experiment, we measured VMwall's impact on network I/O by transferring a 175 MB video file over the virtual network via `wget`. Our second experiment measured the time necessary to establish a TCP connection or transfer UDP data round-trip as perceived by software in domU.

We first transferred the video file from dom0 to domU and back again with VMwall running inside dom0. Table 3 shows the result of our experiments. The median overhead imposed by VMwall is less than 7% when transferring from dom0 to domU, and less than 1% when executing the reverse transfer.

Direction	*Without VMwall*	*With VMwall*	*Overhead*
File Transfer from Dom0 to DomU	1.105	1.179	7%
File Transfer from DomU to Dom0	1.133	1.140	1%

Table 3. Time (seconds) to transfer a 175 MB file between dom0 and domU, with and without VMwall.

Direction	*Without VMwall*	*With VMwall*	*Overhead*
Connection from Dom0 to DomU	197	465	268
Connection from DomU to Dom0	143	1266	1123

Table 4. Single TCP connection setup time (μs) measured both with and without VMwall inside dom0.

Our second metric evaluated the impact of VMwall upon connection or data stream setup time as perceived by applications executing in domU. For processes using TCP, we measured both the inbound and outbound TCP connection setup time. For software using UDP, we measured the time to transfer a small block of data to a process in the other domain and to have the block echoed back.

We created a simple TCP client-server program to measure TCP connection times. The client program measured the time required to connect to the server, shown in Table 4. Inbound connections completed quickly, exhibiting median overhead of only 268 μs. Outbound connections setup from domU to dom0 had a greater median overhead of 1123 μs, due directly to the fact that the introspection time for outbound connections is also high. Though VMwall's connection setup overhead may look high as a percentage, the actual overhead remains slight. Moreover, the introspection cost occurring at connection setup is a one-time cost that gets amortized across the duration of the connection.

We lastly measured the time required to transmit a small block of data and receive an echo reply to evaluate UDP stream setup cost. We wrote a simple UDP echo client and server and measured the round-trip time required for the echo reply. Note that only the first UDP packet required introspection; the echo reply was rapidly handled by a rule in the VMwall kernel module created when processing the first packet. We again have both inbound and outbound measurements, shown in Table 5. The cost of VMwall is small, incurring slowdowns of 381 μs and 577 μs, respectively.

VMwall currently partially optimizes its performance, and additional improvements are clearly possible. VMwall performs introspection once per connection so that further packets from the same connection are allowed or blocked based on the in-kernel rule table. VMwall's performance could be improved in future work by introducing a caching mechanism to the introspection operation. The VMwall introspection routine traverses the guest OS data structures to perform correlation. In order to traverse a data structure, the memory page that contains the data structure needs to be mapped, which is a costly operation. One possible improvement would be to support caching mechanisms inside VMwall's user agent to cache frequently used memory pages to avoid costly memory mapping operations each time.

6.4 Security Analysis

VMwall relies on particular data structures maintained by the domU kernel. An attacker who fully controls domU could violate the integrity of these data struc-

Direction	*Without VMwall*	*With VMwall*	*Overhead*
Inbound Initiated	434	815	381
Outbound Initiated	271	848	577

Table 5. Single UDP echo-reply stream setup time (μs) with and without VMwall. In an inbound-initiated echo, dom0 sent data to domU and domU echoed the data back to dom0. An outbound-initiated echo is the reverse.

tures in an attempt to bypass VMwall's introspection. To counter such attacks, we rely on previous work in kernel integrity protection. Petroni et al. [26] proposed a framework for detecting attacks against dynamic kernel data structures such as `task_struct`. Their monitoring system executed outside the monitored kernel and detected any semantic integrity violation against the kernel's dynamic data. The system protected the integrity of the data structures with an external monitor that enforced high-level integrity policies. In another work, Loscocco et al. [19] introduced a system that used virtualization technology to monitor a Linux kernel's operational integrity. These types of techniques ensure that the kernel data structures read by VMwall remain valid.

Attackers can also try to cloak their malware by appearing to be whitelisted software. An attacker can guess processes that are in VMwall's whitelist by observing the incoming and outgoing traffic from the host and determining themselves what processes legally communicate over the network. They can then rename their malicious binary to the name of a process in the whitelist. VMwall counters this problem by extracting the full path to the process on the guest machine. Attackers could then replace the complete program binary with a trojaned version to evade the full path verification. VMwall itself has no defenses against this attack, but previous research has already addressed this problem with disk monitoring utilities that protect critical files [8, 25].

An attacker could hijack a process by exploiting a vulnerability, and they could then change its in-memory image. To address this problem, VMwall userspace process can perform checksumming of the in-memory image of the process through introspection and compare it with previously stored hash value. However, this process is time consuming and may affect the connection setup time for an application.

An attacker could also hijack a connection after it has been established and verified by VMwall as legitimate. They could take control of the process bound to the port via a software exploit, or they could use a malicious kernel module to alter packet data before sending it to the virtual network interface. VMwall can counter certain instances of connection hijacking by timing out entries in its kernel rule table periodically. Subtle hijacking may require deep packet inspection within VMwall.

VMwall's kernel module internally maintains a small buffer to keep a copy of a packet while performing introspection. An attacker may try to launch a denial of service (DoS) attack, such as a `SYN` flood [5], against VMwall by saturating its internal buffer. VMwall remains robust to such attempted attacks because

its buffer is independent of connection status. As soon as VMwall resolves the process name bound to a connection, it removes the packet from the buffer and does not wait for a TCP handshake to complete.

7 Conclusions and Future Work

We set out to design an application-oriented firewall resistant to the direct attacks that bring down these security utilities today. Our system, VMwall, remains protected from attack by leveraging virtual machine isolation. Although it is a distinct virtual machine, it can recover process-level information of the vulnerable system by using virtual machine introspection to correlate network flows with processes bound to those flows. We have shown the efficacy of VMwall by blocking backdoor, bot, and worm traffic emanating from the monitored system. Our malicious connection detection operates with reasonable overheads upon system performance.

Our current implementation operates for guest Linux kernels. VMwall could be made to work with Microsoft Windows operating systems if it can be programmed with knowledge of the data structures used by the Windows kernel. Since VMwall depends on the guest operating system's data structures to perform network and process correlation, it currently cannot be used for Windows-based guest systems. Recently, XenAccess started providing the ability to map Windows kernel memory into dom0 in the same way as done for Linux. If we have a means to identify and map Windows kernel data structures, then network and process correlation becomes possible.

Acknowledgment of Support and Disclaimer. This material is based upon work supported by the Defense Advanced Research Projects Agency and the United States Air Force under contract number FA8750-06-C-0182. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Projects Agency and the United States Air Force.

We thank the anonymous reviewers for their comments that improved the quality of the paper. We thank Steve Dawson of SRI International for his assistance with this project. Portions of this work were performed while Abhinav Srivastava was at SRI International.

References

[1] A. Baliga, P. Kamat, and L. Iftode. Lurking in the shadows: Identifying systemic threats to kernel data. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2007.

[2] 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. In *19th ACM Symposium on Operating Systems Principles (SOSP)*, Bolton Landing, NY, Oct. 2003.

[3] S. Bellovin. Distributed firewalls. *;login;*, Nov. 1999.

[4] K. Borders, X. Zhao, and A. Prakash. Siren: Catching evasive malware. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2005.

[5] CERT. TCP SYN Flooding and IP Spoofing Attacks. CERT Advisory CS-1996-21. `http://www.cert.org/advisories/CA-1996-21.html`. Last accessed 4 Apr 2008.

[6] Check Point. ZoneAlarm. `http://www.zonealarm.com/store/content/home.jsp`. Last accessed 4 Apr 2008.

[7] Community Developers. Ebtables. `http://ebtables.sourceforge.net/`. Last accessed 1 Nov 2007.

[8] Community Developers. Tripwire. `http://sourceforge.net/projects/tripwire/`. Last accessed 1 Nov 2007.

[9] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *ACM Symposium on Operating Systems Principles (SOSP)*, Bolton Landing, NY, Oct. 2003.

[10] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2003.

[11] T. Garfinkel, M. Rosenblum, and D. Boneh. Flexible OS support and applications for trusted computing. In *9th Hot Topics in Operating Systems (HOTOS)*, Lihue, HI, May 2003.

[12] Gerald Oskoboiny. Whitelist-based spam filtering. `http://impressive.net/people/gerald/2000/12/spam-filtering.html`. Last accessed 4 Apr 2008.

[13] Grok. Coromputer Dunno. `http://lists.grok.org.uk/pipermail/full-disclosure/attachments/20070911/87396911/attachment-0001.txt`. Last accessed 4 Apr 2008.

[14] Honeynet Project. Q8. `http://www.honeynet.org/papers/bots/`. Last accessed 4 Apr 2008.

[15] S. Ioannidis, A. Keromytis, S. Bellovin, and J. Smith. Implementing a distributed firewall. In *ACM Conference on Computer and Communications Security (CCS)*, Athens, Greece, Nov. 2000.

[16] X. Jiang and X. Wang. Out-of-the-box monitoring of VM-based high-interaction honeypots. In *10th International Symposium on Recent Advances in Intrusion Detection (RAID)*, Surfers Paradise, Australia, Sept. 2007.

[17] X. Jiang, X. Wang, and D. Xu. Stealthy malware detection through VMM-based 'out-of-the-box' semantic view. In *14th ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, Nov. 2007.

[18] LKCD Project. LKCD - Linux Kernel Crash Dump. `http://lkcd.sourceforge.net/`. Last accessed 4 Apr 2008.

[19] P. A. Loscocco, P. W. Wilson, J. A. Pendergrass, and C. D. McDonell. Linux kernel integrity measurement using contextual inspection. In *2nd ACM Workshop on Scalable Trusted Computing (STC)*, Alexandria, VA, Nov. 2007.

[20] McAfee. BackDoor-Rev.b. `http://vil.nai.com/vil/Content/v_136510.htm`. Last accessed 4 Apr 2008.

[21] J. Mogul, R. Rashid, and M. Accetta. The packet filter: An efficient mechanism for user-level network code. In *ACM Symposium on Operating Systems Principles (SOSP)*, Austin, TX, Nov. 1987.

[22] Packet Storm. `http://packetstormsecurity.org/UNIX/penetration/rootkits/{bdoor.c,blackhole.c,cheetah.c,server.c,ovas0n.c}`. Last accessed 4 Apr 2008.

[23] Packet Storm. `http://packetstormsecurity.org/0209-exploits/{apache-ssl-bug.c,apache-linux.txt}`. Last accessed 4 Apr 2008.

[24] Packet Storm. Kaiten. http://packetstormsecurity.org/irc/kaiten.c. Last accessed 4 Apr 2008.

[25] B. D. Payne, M. Carbone, and W. Lee. Secure and flexible monitoring of virtual machines. In *23rd Annual Computer Security Applications Conference (ACSAC)*, Miami, FL, Dec. 2007.

[26] N. L. Petroni, Jr., T. Fraser, A. Walters, and W. A. Arbaugh. An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. In *15th USENIX Security Symposium*, Vancouver, BC, Canada, Aug. 2006.

[27] N. L. Petroni, Jr. and M. Hicks. Automated detection of persistent kernel control-flow attacks. In *14th ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, Nov. 2007.

[28] R. Ta-Min, L. Litty, and D. Lie. Splitting interfaces: Making trust between applications and operating systems configurable. In *Symposium on Operating System Design and Implementation (OSDI)*, Seattle, WA, Oct. 2006.

[29] W. Venema. TCP wrapper: Network monitoring, access control and booby traps. In *USENIX UNIX Security Symposium*, Baltimore, MD, Sept. 1992.

[30] A. Whitaker, R. S. Cox, M. Shaw, and S. D. Gribble. Constructing services with interposable virtual hardware. In *1st Symposium on Networked Systems Design and Implementation (NSDI)*, San Francisco, CA, Mar. 2004.

[31] XenAccess Project. XenAccess Library. http://xenaccess.sourceforge.net/. Last accessed 4 Apr 2008.

[32] M. Xu, X. Jiang, R. Sandhu, and X. Zhang. Towards a VMM-based usage control framework for OS kernel integrity protection. In *12th ACM Symposium on Access Control Models and Technologies (SACMAT)*, Sophia Antipolis, France, June 2007.

[33] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *ACM Conference on Computer and Communications Security (CCS)*, Arlington, VA, Oct. 2007.