Automated and Safe Vulnerability Assessment

Fanglu Guo Yang Yu Tzi-cker Chiueh
Computer Science Department, Stony Brook University, NY 11794
Rether Networks Inc., Centereah, NY 11720
{fanglu, yyu, chiueh}@cs.sunysb.edu

Abstract

As the number of system vulnerabilities multiplies in recent years, vulnerability assessment has emerged as a powerful system security administration tool that can identify vulnerabilities in existing systems before they are exploited. Although there are many commercial vulnerability assessment tools in the market, none of them can formally guarantee that the assessment process never compromises the computer systems being tested. This paper proposes a featherweight virtual machine (FVM) technology to address the safety issue associated with vulnerability testing. Compared with other virtual machine technologies, FVM is designed to facilitate sharing between virtual machines but still provides strong protection between them. The FVM technology allows a vulnerability assessment tool to test an exact replica of a production-mode network service, including both hardware and system software components, while guaranteeing that the production-mode network service is fully isolated from the testing process. In addition to safety, the vulnerability assessment support system described in this paper can also automate the entire process of vulnerability testing and thus for the first time makes it feasible to run vulnerability testing autonomously and frequently. Experiments on a Windows-based prototype show that Nessus assessment results against an FVM virtual machine are identical to those against a real machine. Furthermore, modifications to the file system and registry state made by vulnerability assessment runs are completely isolated from the host machine. Finally, the performance impact of vulnerability assessment runs on production network services is as low as 3%.

1. Introduction

Hundreds of new vulnerabilities are being discovered annually. Dozens of new patches are being released monthly. As attack tools become more user-friendly and automated, more script kiddies can use them to randomly scan the Internet for victims with unpatched vulnerabilities. To make things worse, while a system administrator needs to patch every possible hole in her systems, an attacker only needs to locate one to break in.

Given the escalating threats of malicious cyber-attacks, modern enterprises employ multiple lines of defense to protect themselves. First, content-aware intrusion prevention systems (IPS), including firewalls, try to filter out network packets containing attack payloads, including virus, worms, and spyware/adware. Then, file system scanning tools further eliminate those attack programs that somewhat evade the IPS deployed at the enterprise's network entry point. Finally, vulnerability assessment tools determine if existing systems contain certain known vulnerabilities by sending them a series of probe packets and checking if they fall victim to these packets. Among these defenses, only vulnerability assessment allows the system administrator to proactively find the security holes before any attacks try to exploit them.

However, there is a well-known problem associated with vulnerability assessment that prevents it from being used as extensively as IPS/firewall or anti-malware file scanning tools: *safety*. Although there are many commercial vulnerability scanning tools, none of them are truly safe. In one study [1], all scanners studied caused adverse effects on the network servers being tested. One scanner crashed at least five servers during an assessment run. According to Nessus documentation [2], every existing network-based vulnerability scanner comes with the risk of crashing the systems/services being tested, or even worse leaving permanent damaging side effects. In some sense, these results are not surprising at all because vulnerability testing packets should behave like real attack packets in order to expose the vulnerabilities. The only difference is that vulnerability testing packets are not supposed to intentionally cause damage to the tested network servers, even though they might do so accidentally. There are several reasons why this accidental damage could happen in practice. First, some protocol implementations do not handle errors very well, so any unexpected inputs may crash them. Second, if a vulnerability is related to memory errors, e.g., buffer overflow vulnerability, a scanner would send enough data to overflow the buffer, and the overflow could result in unpredictable program execution, including a program crash, or some unde-

sirable modifications to the system state. As a result, vulnerability testing is done once per month or per quarter, even though new vulnerabilities appear every day.

In this paper, we propose a vulnerability assessment support engine (*Vase*) that can ensure the safety of the vulnerability testing process. Unlike Nessus, *Vase* itself cannot send out probe packets that can check if existing network servers contain certain vulnerabilities. Instead, *Vase* is designed to prepare the network servers that are going to be tested by tools such as Nessus, such that the vulnerability testing runs do not leave any permanent side effect on the tested servers. Furthermore the tested targets are identical to the network servers in every aspect.

Vase has two components: Feather-weight Virtual Machine (FVM) and network application duplicator. FVM is a virtual machine technology that creates virtual machine at the operating system layer. An FVM virtual machine is an execution environment on the Microsoft Windows platform. In the execution environment, applications have an illusion of accessing the operating system exclusively. Thus to applications, each FVM virtual machine looks as real as the native host machine. The network application duplicator is a tool that can prepare an FVM virtual machine and duplicate all network applications from host machine to FVM virtual machine. Thus safe vulnerability assessment can be automated.

Although existing virtual machine technologies such as VMware [3] do provide full isolation, they are not appropriate for vulnerability testing because it takes too long to clone a virtual machine from a physical machine, especially for systems with hundreds of gigabytes of active disk space. Moreover, any patching to or reconfiguration of the physical machine requires a full copying to synchronize the physical and virtual machines. Finally, using a separate physical machine to host the cloned virtual machine entails a fixed cost. Instead, *Vase* uses FVM to solve the problem of *quickly* cloning a physical machine to a virtual machine, and makes it possible to conduct vulnerability assessment in an automatic and safe way.

FVM supports several features that make it a good fit for safe vulnerability assessment. First, FVM allows applications to run natively on the host machine or in an FVM virtual machine. This allows production-mode network applications to run on the host machine directly with minimum performance penalties. Second, an FVM virtual machine can be cloned from the host machine in seconds rather than hours. Therefore, one can create an FVM virtual machine from the host machine on the spot and use it as the target of a vulnerability assessment run. This FVM virtual machine will have the same patches, configurations, and OS environment as the host machine. So the testing fidelity is as high as the test is run against the host machine directly. Third, an FVM virtual machine is fully isolated from the host machine. Any modifications to an FVM virtual machine's persistent state, such as files or registries, are contained within the virtual machine and guaranteed not to have any effects on the applications running on the host machine.

FVM is based on resource renaming at the system call interface. When an FVM virtual machine is started, by default its state is the same as that of the host machine. That's why creating an FVM virtual machine can be done instantaneously. When an FVM virtual machine updates its state, the target resource is copied to its own workspace. This keeps any updates within an FVM virtual machine from polluting the host machine. For example, suppose an application in one virtual machine (say VM1) tries to access a file `/a/b`. If it is a read-only access, FVM allows it to access /a/b directly. If it is write access, FVM will copy `/a/b` to `/vm1/a/b`, and transparently redirects subsequent file accesses from `/a/b` to /vm1/a/b. This way, updates in an FVM virtual machine are isolated from the host machine. To the best of our knowledge, FVM is the first virtual machine system in the Microsoft Windows platform that is built on system call interface.

The rest of the paper is organized as follows: Section 2 surveys previous research related to vulnerability assessment and virtual machine. Section 3 describes the technical challenges of *Vase* and their solutions. Section 4 reports the results of an evaluation of a fully working *Vase* prototype. Section 5 concludes the paper with a summary of its major contributions.

2. Related Work

Hardware abstraction layer virtualization. VMware [3] and Microsoft Virtual PC [4] have the virtualization interface at the hardware abstraction layer. They virtualize common PC hardware like processor, memory and peripheral I/O devices such that multiple operating system instances of different type can be installed on a single physical x86 CPU-based machine. The advantage of this approach is that common x86 operating systems can run on virtual machines without modification. But the performance of VMWare is less than 50% of the native host operating system in benchmark tests as reported in [5].

Some light-weight virtual machines on the hardware abstraction layer [6, 5, 7] virtualize only a subset of the hardware. Denali [6] is a virtual machine monitor (VMM) specifically for server applications. It is designed to scale the number of concurrent virtual machines by using *para-virtualization* techniques. Para-virtualization means modifying the virtual machine architecture to enhance scalability, performance, and simplicity. Thus operating systems need to be modified to fit for the new architecture. Denali [6] demonstrates a simple operating system, Ilwaco and its own simple web server. It is not clear if current server applications can be run on it without modification.

Xen [5] goes along the same line as Denali. It also uses para-virtualization to improve performance. But Xen is designed to support full multi-application operating systems and unmodified industry standard application binaries. Linux is ported to Xen architecture and the performance is

close to native Linux. Microsoft Windows on the other hand is more complex to be ported and there is no working system or performance data yet.

The User-Mode Linux (UML) [7] ports the Linux kernel to Linux itself. The UML kernel itself runs in the user space of the host Linux. When an application in the UML is started, the application process is created in both UML kernel and host Linux kernel. The system calls made by UML process are intercepted by ptrace mechanism and redirected to UML kernel. Unfortunately, UML performance is also 50% less than host Linux.

All of the above virtualization techniques try to simulate the hardware abstraction layer so as to create multiple instances of virtual machines for the guest operating system. From the point of view of safe vulnerability assessment, the advantage of these techniques is that they can provide better isolation because each virtual machine has its own operating system. Ironically, the disadvantage of these technologies is the side effect of their advantage: total independence. Because each virtual machine has its own operating system, it needs to copy the whole file system of the production machine to create a duplicate. With the performance gain from Xen, it seems that running production server applications in Xen virtual machine is feasible. The remaining issues are how to create duplicate quickly and support Microsoft Windows. On the other hand, FVM resolves all the issues now by virtualizing at operating system layer.

Operating system layer virtualization. The FreeBSD based *jail* utility [8] creates multiple virtual file systems, each having its own root and other system resources, using resource renaming technique. Even privileged users of a jail can only access resources within that jail. The *chroot()* system call and a few modified macros in the kernel are what it takes to perform the necessary name space separation.

The Linux VServer project [9] is a more advanced jail-like implementation for Linux. The VServer project modifies the process management, file system, networking, root capabilities and system V inter-process communication (IPC). On the host Linux, multiple vservers (virtual machines) can be created and run their own server applications. Interestingly, it provides a *unification* feature. The vservers can use hard links to duplicate the files on the host Linux to save disk space and quickly create the vserver file system. Towards the goal of safe vulnerability assessment, Linux VServer can almost do everything that FVM can do except two issues: Linux VServer doesn't implement copy-on-write and its technology is not applicable to Microsoft Windows. Actually it turns out that the FVM for Windows is much more complex. It involves more modules such as service, registry, objects, IPC (DDE, clipboard, COM), etc which are difficult to virtualize.

Similar to the Linux VServer project, Sphera [10], SWsoft [11], etc provide product to support so called "Virtual Dedicated Server" or "Virtual Private Server" feature. They virtualize a physical machine as if there are multiple stand-alone servers. Each virtual server can have its own server software, root access and files. It seems that SWsoft also support Microsoft Windows. Since few technical details are known, it is not clear if these products can be used for safe vulnerability assessment.

Microsoft Windows Terminal Services [12] avoids the resource conflict among multiple terminal sessions by name space virtualization. The *object manager* in the Windows executive uses a hierarchical naming convention for all the system resources and every terminal session gets a separate sub-tree in this hierarchy. All the resources associated with a particular session are named using the session identifier as its prefix. However, Windows terminal services stops short of virtualizing files, registry hives and network modules.

The Alcatraz project [13] provides a virtual execution environment for Linux applications through system call interposition. It is designed specifically to sandbox untrusted applications that may need to access privileged files. However, it does not support general virtualization that allows multiple virtual machines to co-exist on a single physical machine without interfering with one another. Nor does it support virtualization of other types of system resources than files.

Vulnerability assessment. There are dozens of vulnerability assessment tools [14, 15, 2, 16] in the market. As in the report [1], all 11 tested tools cause some adverse reactions on network servers. Thus it is not surprising that some tools keep safe vulnerability scanning in mind in its design. For example, Foundstone's FoundScan [14], Harris's STAT Scanner [15] and Nessus [2] provide users an option of "safe scan". But even the "safe scan" option may not always work. For instance, Foundstone's FoundScan is reported in [1] causing outage with NetWare even with "safe scan". In the case study reported in [17], the author is aware of the need of regular scan. But he is also aware of the potential to cause issues with scanning. So finally he chooses to scan the network only after midnight once a week.

Tenable [18] reports some techniques to improve the safety of vulnerability assessment. The first solution is to only check the network server's banner to get its version number. Vulnerabilities are reported solely based on server application's version. The technique may cause false positives because some software may be patched without changing the version number. The second solution is passive scanning. Vulnerability is checked by passively monitoring network application's traffic. The disadvantage of this technique is that it may report vulnerability only when attackers attack the applications. This probably is too late. The third technique is to log on a computer and audit its patch. But getting credentials of important computer systems may be a political battle that the security team needs to face.

3. Design and Implementation

The design goals of *Vase* are

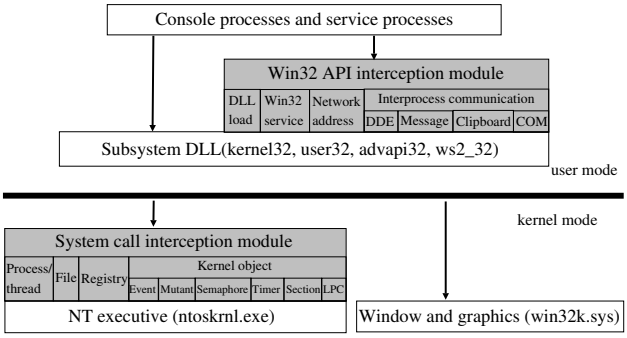

Figure 1. The architecture of FVM.

- *High Testing Fidelity:* The actual tested target should be as close as possible to the real network applications in every conceivable aspect.
- *Full State Isolation:* Vulnerability testing runs should not leave any side effects on the tested network server that could affect its subsequent operation.
- *Automation:* Vulnerability testing runs should be completely automated to the point that vulnerability scanning can be done as frequently as scanning file systems for malware.

3.1. System Overview

As stated before, *Vase* has two components: FVM and network application duplicator. FVM is the underlying technology for virtual machine which realizes the goals of high testing fidelity and full state isolation. Network application duplicator uses the FVM technology to realizes the goal of automating the process of vulnerability testing.

When a new FVM virtual machine is created, it inherits the state of the host machine by default. Therefore, when *Vase* duplicates a network application in an FVM virtual machine, the duplicate can access everything accessible to the original network application, including special registry entries and the configuration files. Therefore, *Vase* can achieve high testing fidelity with FVM.

FVM uses a copy-on-write technique to ensure that every file/registry update made by processes in an FVM virtual machine result in a copy of the target file/registry in the virtual machine's work space. Consequently, *Vase* can also achieve full isolation with FVM.

The network applications duplicator automates the process of enumerating network applications in host machine, starting a virtual machine, and starting network application duplicates in the virtual machine.

The key idea in FVM is resource renaming, which is implemented by intercepting system calls and rewriting their resource arguments, as in Figure 1. FVM performs two types of interception. To intercept at the system call interface, FVM modifies the System Service Dispatch Table (SSDT) to redirect a subset of system calls through FVM's virtualization logic, which is implemented as a kernel driver. To intercept at the library call interface, FVM uses a separate DLL (Dynamically Linked Library) to intercept Win32 API calls. FVM uses the second interception method for two reasons. First, some system calls are undocumented, so we have to intercept them at the library call level. Second, some Win32 API, such as `CreateService`, is not just a simple wrapper around a system call; rather it makes a series of system calls that cannot be easily recognized by the kernel-level system call interceptor.

FVM's virtualization logic is divided into the following modules:

Console process virtualization. Console process means the process created by the user who uses the console. This module intercepts process management calls to classify console process to its corresponding virtual machine.

Service process virtualization. Services are to Windows what daemons are to UNIX. A service is a background process which is started by the Service Control Manager (SCM). It runs independently of who, if anyone, is using the console. This module traces the services installed in FVM virtual machines and classifies the FVM service process to its corresponding virtual machine.

File virtualization. This module implements a copy-on-write policy to share host machine's file state while containing virtual machine's update to its own context. It handles file and directory operations.

Registry virtualization. Windows registry is almost like standard UNIX file system. The root of registry has five keys. Each key is like a directory. In the key, subkeys can be created like subdirectories. In the key, values can also be created. Values are like files. The value name is like file name and value data is like file content. This module handles key/value operations.

Object virtualization. Windows objects here refer to the base named kernel objects such as semaphores, mutexes and events. These objects usually share the same global name space and are mainly used for process/thread synchronization and notification. Object virtualization localizes object access and thus avoids the interference between the original instance on the host machine and the duplicate instance on the virtual machine.

Interprocess communication. In addition to synchronization objects mentioned above, we will discuss more interprocess communication (IPC) mechanisms to share data or facilitate process synchronization.

Network virtualization. A network server application often listens on a well-known port with a wildcard IP address (INADDR_ANY). This creates problems to start its duplicate in FVM virtual machine because the duplicate will try to bind to the same port too. This module renames the IP address in bind call to its virtual machine IP address. Thus the port can be reused and network applications feel that it exclusively owns the network resource.

3.2. Console Process Virtualization

There are two types of processes that can run on an FVM virtual machine: *console process* and *service process*. Regardless of the process type, when a process issues a system call to read or modify a system resource, FVM needs to answer the question of which virtual machine should serve as the context within which to interpret this system call. So FVM needs to maintain a mapping between the process ID and virtual machine ID.

This module handles the console process. When a virtual machine is started, FVM automatically starts its initial process, which is either a command shell or file browser, and tags it with the virtual machine ID. All the descendant processes of this initial process belong to the same virtual machine and thus are tagged with its virtual machine ID too.

FVM maintains a list of processes running on a virtual machine by registering a kernel callback routine that is notified whenever a process is created. The callback routine receives both the new process ID and its parent process ID. If the parent process is in a virtual machine, the new process is assigned to the same virtual machine. The callback routine registration is achieved by calling kernel function `PsSetCreateProcessNotifyRoutine`.

3.3. Service Process Virtualization

Service processes differ from console processes in that their parent process is the Service Control Manager (SCM), which runs on the host machine. Thus the parent process of all service processes is the SCM process. The idea used in console process virtualization is not applicable to this context.

A user application normally installs a service by adding a service name and its program image name into SCM's database. Later on, the SCM starts or stops the service process upon an application's request. Because SCM is responsible for managing all the system and user services, it cannot be duplicated on each virtual machine. To fully isolate one virtual machine from others, FVM needs to ensure that a service process is executed within the context of a virtual machine where the service is installed. The idea is to build another mapping between service and virtual machine at the service installation time and convert it to process-VM mapping at the run time.

FVM employs a DLL hooking mechanism [19] to intercept the `CreateService` call in the user level and passes all the necessary information to the kernel driver, including a service name and the name of the program image used for the service process. As usual, the DLL hook modifies the original service name by appending the virtual machine ID to it before calling `CreateService`. The system call interceptor extracts the program image name and the renamed service name to create an internal mapping. Later on, when SCM starts a service process using a program image name that matches one of the stored image names, FVM binds this service process to the virtual machine whose ID appears in the associated renamed service name, and all subsequent modifications made by this service process are automatically ascribed to the matching virtual machine. In addition, other service manipulation Win32 calls that take a service name as argument, such as `OpenService`, are also intercepted to ensure the consistency of SCM database and service states.

3.4. File Virtualization

File virtualization isolates any file updates on a virtual machine from the host machine by a copy-on-write approach. By default all the files are shared by virtual machines and the host machine. When processes on a virtual machine intends to write a file, FVM makes a copy of the file and all the subsequent access from the virtual machine will be directed to this copy. The full path of the new copy is constructed from the original file's path and the virtual machine ID.

FVM intercepts file-related system calls that take file names as input arguments and renames the file names. Such system calls include creating/opening files and reading file attributes. According to the read/write access type and complicated option flags, as well as the existence of the target file on both the host machine and the virtual machine, FVM decides whether the system calls should operate on the original file or a local copy on the virtual machine, or simply return an error status. FVM ensures that a file's local copy on a virtual machine has the same attributes and directory structure as the original file on the host machine.

System calls to read/write files take a file handle as input argument instead of a file name. Since the file handle is obtained through creating/opening file system calls, it already points to the right copy of the file and there is no need to intercept these handle-oriented system calls. One exception stays with deleting or renaming a file. In Win32 subsystem, deleting or renaming a file involves opening the file and setting special attributes on the returned file handle by `NtSetInformationFile` system call. Because FVM needs to keep track of the deleted and renamed files in a virtual machine to accurately reflect its file image state, FVM intercepts `NtSetInformationFile` when this system call is used to delete or rename a file. FVM recovers the file name from the input file handle and adds it into a delete name list of this virtual machine. The name list is written into an on-disk log file periodically. In the file renaming case, FVM also renames the new file name so that the new file is only visible on the virtual machine.

3.5. Registry Virtualization

Windows registry is the repository where Windows configurations are stored and must be virtualized to isolate any configuration updates on a virtual machine from the host

machine. To reduce implementation complexity, FVM embeds a virtual machine's registry entries in the host machine's registry using Windows' own registry subsystem. More concretely, FVM intercepts all registry-related system calls that use registry keys as arguments, and renames these registry key arguments by prepending a prefix to the path name of these keys. For each virtual machine, FVM creates a registry entry under the key `\HKEY_CURRENT_USER`. For example, for the virtual machine VM1, FVM creates the key `\HKEY_CURRENT_USER\VM1` as its root, and all the registry entries in VM1 will be stored under this key. Whenever any application from VM1 accesses a registry key, FVM adds the prefix `\HKEY_CURRENT_USER\VM1` to the path name of the requested registry key.

FVM utilizes the same copy-on-write approach as file virtualization to handle registry access in a virtual machine. Depending on whether the access is opened for read or write, FVM directs the intercepted registry-opening system call to operate on either the original registry key, or a new registry key copy under the virtual machine's registry root. If an application in a virtual machine creates a new registry key, it is always created under the virtual machine's registry root.

3.6. Object Virtualization

Interprocess synchronization on Windows is mostly implemented by kernel objects, such as event and timer. In addition, Windows provides many Windows-specific interprocess communication mechanisms for data sharing. We will talk about the Windows interprocess communication issues in the next subsection, and focus on the virtualization of generic kernel objects in this subsection.

The Windows kernel objects, e.g., mutex, shared memory, event, etc., normally share the same global name space, especially when these objects are created by a network application, which has at most one instance running simultaneously. This implementation may cause problems when doing vulnerability assessment on FVM that requires duplicated instances running on both the host machine and a virtual machine. If one instance on the host machine is accessing a named kernel object, another instance of the same application on a virtual machine may not be able to access the same object correctly. FVM resolves this problem by virtualizing kernel objects.

Kernel objects have a hierarchy similar to files and registries. There are many object directories in the system under the root object directory. Each object directory contains multiple kernel objects with similar types. Based on the same renaming approach as registry virtualization, a local object directory tree is created for each FVM virtual machine whenever the virtual machine is started. When a virtual machine is stopped, its local object directory is removed and all the opened object handles under it are closed. FVM intercepts the create/open system calls that access base kernel objects, including mutant, semaphore, event, section[1], timer, iocompletion, eventpair and symbolic link. FVM redirects access to these objects to a virtual machine's local object directory and thus guarantee that the application instances on the host machine and the virtual machine do not access the same kernel object. Similarly, special file object like named pipe and mailslot are also virtualized.

3.7. Interprocess Communication on VMs

For the isolation purpose, FVM requires that a process running on one virtual machine not communicate with processes running on the host machine or other virtual machines through interprocess communication (IPC), unless it has to talk with a Windows system service on the host machine that cannot be virtualized, or it intends to use the IPC to talk to another physical machine.

Common IPC mechanisms include shared memory, named pipe, mailslot, LPC (Local Procedure Calls). Through object virtualization, IPCs based on shared memory (*section object*), named pipe and mailslot are automatically confined. Similarly, Windows LPC, which internally uses *port object* for high-speed message passing, is also virtualized.

There are still some Windows-specific IPC mechanisms like DDE, COM, window message, clipboard, etc. These IPC mechanisms are mainly used for interactive applications. For example, COM (Component Object Model) applications may query the *Running Object Table* (ROT) to determine whether there is already an instance running. DDE (Dynamic Data Exchange) allow new instance to communicate with existing instances by broadcasting Window messages to desktop windows. Although network applications normally are not involved with these mechanisms, we are still interested in confining them so we can provide a better isolation environment for all the Windows applications in future.

3.8. Network Virtualization

A network application starts by creating a socket and making a `bind` call to specify the local IP address and local port number for the socket. In supporting vulnerability testing, *Vase* creates a duplicate of the target network application to be tested in an FVM virtual machine. The duplicate will bind to the same IP address and port number. Without network interface virtualization, this is not possible because the operating system simply does not allow two processes to bind themselves to the same IP address and port number pair.

FVM virtualizes an network interface by assigning each FVM virtual machine a distinct IP address. FVM first uses the IP aliasing mechanism to assign multiple IP addresses to the network interface. When FVM starts a virtual machine, it adds the virtual machine's IP address to the host

[1]Refer to shared memory, also called file mapping object in Win32 subsystem

machine's physical interface as an IP alias. When a virtual machine is stopped, its IP address is removed from the physical interface accordingly.

When a network application running in an FVM virtual machine makes a `bind` call, FVM intercepts it and changes the local IP address in the call to the virtual machine's IP address transparently. That is, the network application actually binds to its virtual machine's IP address. Consequently, processes in one virtual machine can neither receive traffic destined to other virtual machines nor spoof another machine's IP address when sending traffic. Even if a target application hard-codes its bind IP address in a configuration file, FVM still can bind the application's duplicate to the IP address of its virtual machine.

3.9. Network Application Duplicator

To automate a vulnerability test run, *Vase* needs to be able to create a virtual machine from a physical machine, and start all network applications currently running on that physical machine in the new virtual machine in exactly the same way as they were started originally.

There are two types of network applications on the Microsoft Windows. The first type is console applications such as eDonkey. This type of applications can be started as normal command line programs. The second type is Windows service. Most server applications such as Apache and IIS (Internet Information Service) fall into this category. Different from console applications, service applications cannot be started from the command line. They need to be installed through the SCM (Service Control Manager), which is the one that actually starts service applications.

There are four phases in the duplicator. In the first phase, the duplicator enumerates all the network applications running on a target network server. The duplicator uses a tool called `fport` from Foundstone [14] to find out all processes that are listening on a network port, including the absolute pathname of the executable file name behind each of these processes. However, `fport` cannot tell whether a process is from a console application or from a service application. The duplicator queries SCM's database using `EnumServicesStatus` and `QueryServiceConfig` to identify the pathname of each service's executable file. If an executable returned by `fport` is found in SCM's database, it is treated as a service application; otherwise it is a console application. After this enumeration phase, the duplicator obtains a complete list of network applications that are currently running on the host machine and how to start them.

In the second phase, the duplicator first creates a new FVM virtual machine through a new function `StartVM` from FVM. Then the duplicator sets itself as the first process in the new created FVM virtual machine. Finally it installs and starts all the network applications identified in the first phase. Because the duplicator process runs in the new virtual machine, the applications it starts will also run in the same virtual machine.

In the third phase, the duplicator invokes a vulnerability assessment tool such as Nessus against the FVM virtual machine which runs all the duplicated network applications. These vulnerability assessment tools produce a report as the end result.

The final phase is cleanup. After a vulnerability scan run is completed, the duplicator will clean up the FVM virtual machine by stopping the console applications, stopping and uninstalling service applications, and finally terminating the FVM virtual machine.

3.10. Implementation Lessons

Although the idea of virtualization through resource renaming is conceptually simple, it is surprisingly difficult to implement on the Windows platform. First of all, there are so many name spaces in the Windows operating system. As summarized in Table 1, different Windows subsystems use different name spaces and involve many functions. There is no systematic way to enumerate all of them. Whenever some name spaces are not handled properly, the degree of isolation that FVM provides suffers. Secondly, even for a given name space, it is not always possible to apply resource renaming in a consistent way. For example, most kernel objects should be renamed but some of them should be left alone because some system services cannot be duplicated.

4. Evaluation

We implemented a *Vase* prototype and evaluated it from the following angles: performance overhead, fidelity, isolation and start-up delay. Three computers are used in the following experiments, all with the same hardware configuration: an AMD Athlon XP 2000+ CPU with 512 MBytes memory and a 7200 RPM hard disk. The first computer is used as a server. It runs Microsoft Windows 2000 with service pack 4, Internet Information Service (IIS) 5.0, Microsoft SQL server 2000, MySQL server 4.1, and Apache web server 2.0.54. The second computer runs Redhat Linux 9.0 and Nessus 2.2.4, and serves as a vulnerability assessment scanner machine. The third computer is only used for performance test. It runs Redhat Linux 9.0, Apache benchmark, which is used to benchmark Apache web server, and Super Smack 1.3 [20], which is used to benchmark MySQL.

4.1. Performance Overhead

Because *Vase* runs a duplicate copy of the target production-mode network application in an FVM virtual machine but on the same physical machine, there may be some performance impacts on the original network application. This experiment intends to quantify this performance cost. We test each target network application in four scenarios:

Table 1. Functions intercepted for virtualization. Functions with name Nt* are intercepted in kernel. All others are intercepted in user level.

Modules	Functions
Console process	NtResumeThread
Service process	CreateService, OpenService, StartServiceCtrlDispatcher, RegisterServiceCtrlHandler(Ex), NtResumeThread
File	NtCreateFile, NtOpenFile, NtQueryAttributesFile, NtQueryFullAttributesFile, NtQueryDirectoryFile, NtQueryInformationFile, NtSetInformationFile, NtDeleteFile, NtClose
Registry	NtCreateKey, NtOpenKey, NtQueryKey, NtDeleteKey
Object	NtCreateMutant, NtOpenMutant, NtCreateSemaphore, NtOpenSemaphore, NtCreateEvent, NtOpenEvent, NtCreateTimer, NtOpenTimer, NtCreateIoCompletion, NtOpenIoCompletion, NtCreateEventPair, NtOpenEventPair, NtCreateSection, NtOpenSection, NtCreatePort, NtCreateWaitablePort, NtConnectPort, NtSecureConnectPort
IPC	SendMessage, SendMessageTimeout, SendNotifyMessage, SendMessageCallback, PostMessage, PostThreadMessage, DdeCreateStringHandle, GlobalAddAtom, NtCreateNamedPipeFile, NtCreateMailslotFile, NtCreateFile, NtOpenFile
Network	bind

1. We measure the performance of the network application being tested when it runs directly on the host machine. There is no FVM interception overhead.

2. We measure the performance of the network application when it runs directly on the host machine and a duplicate of itself runs in an FVM virtual machine on the same physical machine. This tests the performance impact of FVM's system call interception on the target network application.

3. We measure the performance of the network application when it runs directly on the host machine, and a duplicate of itself runs in an FVM virtual machine on the same physical machine and is being tested by Nessus. This measures the additional performance cost to the target network application due to the interactions between the duplicate and Nessus.

4. We measure the performance of the duplicate running in an FVM virtual machine. This produces the performance overhead introduced by FVM.

Three applications are used: Apache web server, MySQL and WinRAR. We use Apache bench to benchmark Apache web server. Apache bench uses 5 concurrent connections and sends 5000 requests for the index.html in the document root directory. The performance metric is the number of requests served per second. We used Super Smack to benchmark MySQL. We cannot test Microsoft SQL Server because no free tools are available and the license agreement of Microsoft SQL Server prohibits disclosing benchmark results to third parties. Super Smack uses 10 concurrent connections and 10000 requests for select operation (the `select-key.smack` scripts) and update operation (the `update-select.smack` scripts). The performance metric is the number of select or update operations served per second. Finally we tested WinRAR 3.42 to evaluate the performance impact of FVM on non-network applications. The performance metric for WinRAR is the time (seconds) required to extract a software package, in this case the Apache web server 2.0.54 package. Each reported performance number is an average of ten runs.

Figure 2 shows the normalized performance of each test application under Scenario (2), (3), and (4), with respect to that under Scenario (1). That is, 95% means that the application's performance is 95% of that when it runs alone on the host machine directly. The result shows that just running a duplicate in an FVM machine or even performing a vulnerability testing against such a duplicate has little effects on the performance of the production-mode network application. The performance degradation is less than 3%. This means that *Vase*'s approach of "duplicate and test" actually allows a target application to continue its service while it is being tested. As for FVM's virtualization overhead, it is negligible for MySQL, and is about 9% for Apache and WinRAR. From a thorough examination of the detailed system call traces, we found that both Apache and WinRAR invoke a large number of file and directory operations. Because of the copy-on-write policy, FVM always checks the virtual machine's files/directories before accessing the host machine's. The overhead of these additional checks becomes significant for applications that require intensive file/directory operations.

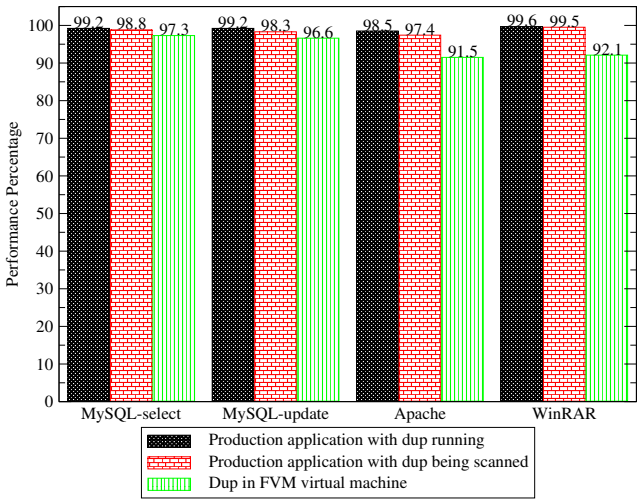

Figure 2. The performance impact of virtualization and vulnerability assessment.

Table 2. The vulnerabilities found by Nessus during the fidelity test. Applications that do not have vulnerabilities are not listed.

Applications	Security Holes	Security Warnings
IIS FTP	1	1
IIS SMTP	1	0
IIS Web	4	8
Apache Web	0	1
Telnet	0	1
KaZaA	1	0

4.2. Fidelity

To demonstrate that scanning a duplicate of a target application running in an FVM virtual machine is as real as scanning the application running on the host machine directly, we tested Windows's telnet service, Microsoft SQL server, IIS SMTP service, IIS FTP service, IIS Web service, MySQL, Apache Web server, eDonkey 2000 and KaZaA. We run all the network applications on the server machine and then apply Nessus to each of them to produce a vulnerability report. The test result is in Table 2. Among the test applications, IIS Web service appears to be the most vulnerable. There are totally 7 security holes found and 4 of them are in IIS Web service. Among the 11 security warnings, IIS Web service accounts for 8 of them.

Then we run all the applications in an FVM virtual machine and apply Nessus to them in the same way. The vulnerability reports produced are the same as is shown in Table 2. This experiments provides an initial evidence that as far as assessing the vulnerability of a network application is concerned, running the application in an FVM virtual machine is as good as running it on a physical machine.

4.3. Isolation

To demonstrate that *Vase* indeed isolates the side effects produced by vulnerability assessment from the host machine, we record the file system and registry state before a vulnerability test, and compare it with the file system and registry state after the test. First we create a disk image of the whole file system of the target server. Then we start vulnerability assessment runs against network applications running in an FVM virtual machine on the target server. After the test is completed, we create a second disk image of the target server. Finally, we use `Beyond Compare`, which is a Windows tool that can recursively compare all the subdirectories and files under the root directory, to compare the two disk images.

The comparison report shows that all modifications to the file system are indeed contained within the home directory of the FVM virtual machine. Suppose the FVM virtual machine has a home directory of `C:\fvm0`. Then the only file system modifications due to the tests occur under `C:\fvm0`. For example, IIS Web server's log file is changed during vulnerability assessment, and it is stored in `C:\fvm0\C\WINNT\system32\LogFiles\W3SVC1`. The corresponding file in the host machine `C:\WINNT\system32\LogFiles\W3SVC1` is not affected.

By taking a closer look at the FVM virtual machine's home directory, we found vulnerability testing runs indeed leave many side effects to the file system. Without FVM, these side effects would have affected the host machine. For example, IIS and Apache Web server changes their access logs, which record all the HTTP requests from Nessus; IIS FTP server logs records all of Nessus's file accesses, including "make directory" and file transfer operations; Microsoft SQL server and MySQL changes their database access log; eDonkey generates some .met files.

To detect changes to the registry state, we export the registry using the program `regedit.exe` before and after a vulnerability testing run, and then use `UltraEdit` to compare the two exported registry files. We found that Nessus does not seem to result in registry state changes. Among the test applications, the registry state changes only in the case of KaZaA. However, we believe this change is due to KaZaA itself rather than due to Nessus, because the key name is `kazaaNet` and `Seed`. In any case, *Vase* is able to successfully confine this registry change to the FVM virtual machine in which KaZaA runs.

4.4. Start-up Delay

One of the major goals of *Vase* is to automate the vulnerability testing process so that one can run vulnerability testing as frequently as virus scanning. In this subsection, we evaluate the time *Vase* takes to prepare a network server

for vulnerability testing. We measured the time required to clone an FVM virtual machine from the server machine and duplicate all the network applications currently running on the server machine to the new virtual machine. It takes 43 seconds for the server machine on which 9 network applications (the same applications as in fidelity test) are currently running. The time for creating an FVM virtual machine and starting most network applications is very small. Most of this 43 seconds is spent on starting Microsoft SQL Server and MySQL, because the amounts of data they need to copy from the host machine to the virtual machine are 40 MBytes and 30 MBytes, respectively. As a result, these two applications take 19 seconds and 13 seconds, respectively.

As a comparison, we also measured the time required to physically clone our server machine. We use Norton Ghost 2003 to make a copy of the server machine's system disk. The system disk installs all the software used in this paper and is around 3.2 GBytes in size. It takes Norton Ghost 13 minutes to finish this copying. The average disk copy speed is only 4 MBytes per second because Norton Ghost clones a disk by copying files one by one, which incurs significant overhead for small files.

5. Conclusion

Safety of vulnerability assessment is a well-known problem to which there is no adequate solution until now. Although some vulnerability scanning tools such as Nessus and Foundstone attempted to mitigate this problem, as described in Section 2, none of them completely solve it. Even when the "safe scan" option is turned on, some vulnerability testing tools [1] still crash the servers being tested.

This paper describes the design, implementation and evaluation of the first known vulnerability assessment support system called *Vase*, which can automate the vulnerability testing process while guaranteeing its safety. *Vase* is built on the feather-weight virtual machine (FVM) technology, which provides the same isolation property as heavy-weight virtual machine technologies such as VMware but incurs a much smaller start-up overhead and resource requirement. The current FVM prototype runs on the Microsoft Windows and can successfully virtualize system resources such as file, registry, object, network interface, service, and inter-process communication.

Vase can automatically clone a physical machine to an FVM virtual machine, including the set of network services currently running on the physical machine, and redirect vulnerability testing packets toward the cloned virtual machine. Experiments on the *Vase* prototype show that the vulnerability testing using Nessus produces identical results when it is run against the physical machine and against the cloned virtual machine (fidelity), that none of the vulnerability testing runs create any side effects in the host machine (isolation), and that the performance cost of vulnerability testing to the production-mode network applications is minimal (transparency).

References

[1] K. Novak, "VA Scanners Pinpoint Your Weak Spots," Network Computing, 2003. http://www.nwc.com/1412/1412f2.html

[2] Nessus, "Configuring Nessus to perform local security checks on Unix hosts." http://nessus.org/documentation/index.php?doc=ssh

[3] VMware, "VMware Workstation." http://www.vmware.com/products/desktop/ws_features.html

[4] Microsoft, "Microsoft Virtual PC 2004." http://www.microsoft.com/windows/virtualpc/default.mspx

[5] 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 *Proceedings of the nineteenth ACM symposium on Operating systems principles*. ACM Press, 2003, pp. 164–177.

[6] A. Whitaker, M. Shaw, and S. D. Gribble, "Denali: Lightweight Virtual Machines for Distributed and Networked Applications," Technical Report 02-02-01, 2002.

[7] J. Dike, "A user-mode port of the Linux kernel," in *Proceedings of the 4th Annual Linux Showcase and Conference*, 2000.

[8] P. Kamp and R. Watson, "Jails: Confining the Omnipotent Root," in *Proceedings of the 2nd International SANE Conference*, 2000.

[9] Linux VServer Project. http://linux-vserver.org/

[10] Sphera, "Shared Hosting." http://www.sphera.com/business-solutions-shared-hosting.php

[11] SWsoft, "What is Virtual Private Server?" http://www.sw-soft.com/en/virtuozzo/hsp/

[12] Microsoft, "Technical Overview of Terminal Services." http://download.microsoft.com/download/2/8/1/281f4d94-ee89-4b21-9f9e-9a%ccef44a743/TerminalServerOverview.doc

[13] Z. Liang, V. Venkatakrishnan, and R. Sekar, "Isolated Program Execution: An Application Transparent Approach for Executing Untrusted Programs," in *Proceedings of Annual Computer Security Applications Conference*, December 2003.

[14] Foundstone, "FoundScan Engine." http://www.foundstone.com/

[15] Harris, "STAT Scanner." http://www.stat.harris.com/solutions/vuln_assess/scanner_index.asp

[16] eEye Digital Security, "Retina Network Security Scanner." http://www.eeye.com/

[17] K. Austin, "Implementing Vulnerability Assessment with eEyes EVA Suite – Case Study," 2004. http://www.giac.org/practical/GSEC/Kevin_Austin_GSEC.pdf

[18] R. Deraison and R. Gula, "Blended Security Assessments," 2004. http://www.tenablesecurity.com/images/pdfs/blended_security_checks.pdf

[19] I. Ivanov, "API Hooking Revealed," 2002. http://www.codeproject.com/system/hooksys.asp

[20] T. Bourke, "Super smack." http://vegan.net/tony/supersmack/