Process Containment using Virtual Machines

Daniel Stamate
Computer Science Department, Drexel University
ds352@drexel.edu

ABSTRACT

This paper introduces various techniques and levels of virtualization and their effort in achieving process containment. Virtual machines have evolved since the 1960's when they were initially researched, and after two decades of hibernation virtual machines are making a strong comeback. Introduced in this paper are three types of virtualization techniques: hardware-level, operating system-level and high language-level virtualization. Each level has a distinct approach in solving the isolation problem.

1. INTRODUCTION

Isolation is a precondition for building secure and fault tolerant systems. Although inter-process communication is often required, most processes should be isolated for the simple fact that bugs can and do exist. When one process strays from normal execution, it has the ability to take other services down. It is wishful thinking to believe otherwise. Processes have certain requirements in order to run, and by safeguarding these requirements we offer protection to the process and the environment it operates in. Memory and processor time are the main necessities, although input/output operations may also be necessary. The solution seems simple: give every process its own memory location, its own time slice and allow them to sequentially access the I/O devices. This can be theoretically achieved, although the penalty is usually a significant degradation in performance. This degradation can be so extreme, that the system becomes inoperable (see RVM).

This paper is broken into several sections. Section 2 attempts to consolidate the notion of process containment by defining what it means to achieve isolation. Sections 3 introduces various techniques of process containment that are used by modern operating systems. These methods can be used in conjunction in order to enhance an existing security model. Section 4 discusses the virtual machine approach and the differences between three types of virtualization.

2. PROCESS CONTAINMENT DEFINED

Memory:

As mentioned before, there are three types of containment. The first type and the hardest to secure is memory. In today's operating systems, every process is given its own memory space based on its requirements; however their address space can and usually will grow. Initially, the address space of a process is broken into three main sections: stack, data and text [1]. The text section is a static portion and it contains the executable statements of a process. The data section holds the variables and constants of the program. The stack section is the run-time stack of the application which holds function

names and their parameters as they are being used. To achieve isolation, processes must be restricted to their own address space, over which they are given full control. At times, either purposely or un-purposely, a process may attempt to access a memory location outside its address space. Unless the caused fault is critical, the process may be shutdown. In today's operating systems, many situations arise when efficiency wins over security, and more then one process shares a part of its address space with another. These situations are considered contradictions to the memory isolation problem, even when they may result in higher performance.

Input/Output:

Input/output refers to the accessing of peripheral devices such as CD-ROMs, printers, network cards, etc. Most processes need to invoke I/O system calls, which for the most part are resolved successfully. Process containment must assure us that processes won't monopolize I/O devices, which creates a denial-of-service attack against the rest of the processes. Clearly process isolation is more then preventing processes from directly communicating, but also involves the more arduous task of preventing resource starvation.

Processor:

As with I/O operations, the CPU must be protected against greedy processes. Fair CPU scheduling is a goal which every modern operating system is trying to achieve, although some are more successful then others. As it will later be shown, virtual machines are capable of achieving fair distribution of time slices to all processes under their control.

3. CONTAINMENT METHODS

Since many operating system techniques for process isolation are incorporated in the virtual machine architecture, it is important to have a prior understanding of these methods. Although not a complete list of tools and techniques, the following should give the reader a broad review.

Access Control List / Chroot:

Access control lists represent the common tools for verifying whether a process is allowed to access a file/directory in the UNIX model. This is accomplished by checking whether the user itself is granted permission to access the file, or whether he is part of a group which holds that permission. The UNIX operating system provides other tools to further restrict a potentially dangerous process. The "chroot" command can be interpreted as a less restrictive sandbox, by only restricting file access. This command obscures the remaining file system by changing the root directory of the process. Well informed attackers can circumvent this security feature by exploiting security holes in other processes which run with root privilege.

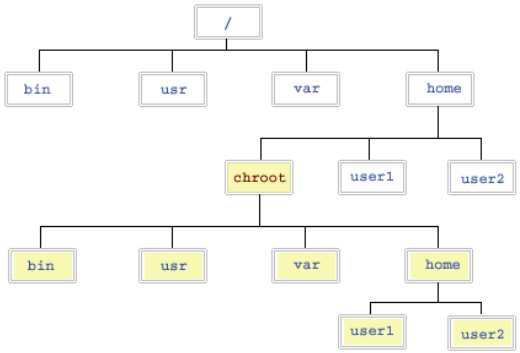

Figure 1. Chroot example

Besides changing the root directory of a process, modern operating systems have several other tools in their arsenal. Systrace is one of those tools, and it provides process containment based on predefined policy files. Systrace is available on most BSD systems and it is capable of either launching programs or attaching to pre-existing processes. In either case, it monitors and intercepts every system call that the process may generate. Based on a predefined user or global policy, systrace either grants or denies the system call. The policy files can be manually or automatically generated. The ability to automatically generate these files is what makes this tool extremely flexible. The process is automated by simply running the application and exercising its complete functionally. Each application has its own policy file, and if a system call is not explicitly represented in the policy, then it is blocked by default. This feature allows users to interactively generate the policy file. This becomes important when the application may want to perform an action which is not essential to its execution, and the user may want to prevent this scenario from occurring.

Sandboxes:

Sandboxes are facilities which provide an environment for a process to run within. Sandboxes should not be confused with virtual machines which either replicates or creates a distinct environment. Sandboxes are usually set-up using existing operating system tools in conjunction with one or more non-standard tools such as the FMAC tool [10]. Sandboxes are usually less complicated then full-blown virtual machines. This flexibility allows sandboxes not only to control a process, but to be able to control individual threads associated with that process. This fine grained isolation is useful for today's internet based applications such as web servers [6]. Sandboxes prevent processes from interacting with the outside world, which controls and surrounds them. As long as the process is accessing internal memory or is not attempting to perform I/O operations, the sandbox does not interfere. When one process attempts to communicate with another process or tries to perform an I/O operation outside its sandbox, the system intervenes and resolves the issue based on predefined criteria. This is usually accomplished using some type of an access list.

A sandbox achieves isolation by buffering sensitive information. It provides copies of the real resource instead of the resource itself. The FMAC tool is such an example, and it achieves isolation by coping files from the file system into the sandbox environment. These files are needed for the execution of the contained application, and

must be determined ahead of execution [10]. As with any sandbox type environment, there is a set-up time required for the FMAC tool to become effective, however, this is much shorter then setting up a virtual machine environment [6].

There are many benefits and uses for sandboxing systems, due to their flexibility. Testing applications which may bring down the entire system unless isolated, is a perfect example of the need for containment. During the testing phase of application development, it is imperative that the new version is controlled. Furthermore, the ability to control individual threads makes sandboxing even a better candidate for testing environments since multiple threads can be ran in individual sandboxes and observations of the different outputs can be compared [6]. This method of containment can also be used as a monitoring tool or a debugging tool. These types of applications are directly tied with the ever-supervising capability of a sandbox. The figure below illustrates how a typical sandbox works.

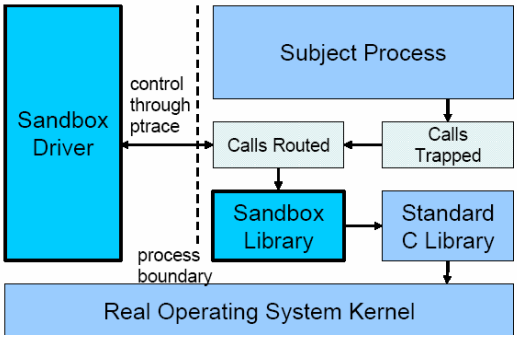

Figure 2. General sandbox [6]

As the figure indicates, the approach of this system is to intercept privileged commands. The call is trapped and routed to a sandbox library and potentially a policy library which either grants or refuses access to that particular system call. In most cases, such as I/O, a copy of the real file is made available to the confined application. Sandbox systems such as Alcatraz and MAPbox use this approach [6].

A more robust and restrictive tool is the jail tool which is supported by the FreeBSD operating system [6]. The jail tool sets up a partitioned environment by including the file system, network resources, processes etc. into a managed environment [11]. This approach allows for limited root activity in the jailed environment while maintaining the existing UNIX security model. Some of the limitations include [11]:

- Loading kernel modules
- Modifying kernel runtime parameters
- Modifying network configuration, interfaces, addresses and routing tables
- Accessing network resources not associated with the jail
- Mounting and unmounting file systems
- Creating device nodes

Although it may seem restrictive, most applications have no quandaries running in this environment. Normal running processes do not need the above listed operations only normal functionality such as reading/writing to a disk, accessing I/O devices, making network connections, creating files and directories etc. Process visibility and interaction

however has been reduced, by making changes to some of the mechanisms in FreeBSD such as the “procfs” and “sysctl” calls. “procfs” or process file system, provides a view of the system process table inside the file system and “sysct” allows users to modify kernel parameters at runtime [13,14]. These types of functions must be disabled along with other sensitive methods. Network operations have also been reduced to one IP address. Any other IP address to which the jailed process may attempt to bind will be redirected to its default IP address, which also includes the loop back IP (127.0.0.1) [11].

The installation of the jail system is a miniature installation of the entire operating system and must be completed per individual jail. This is done to maximize the independence of each jail unit. After the host operating system is installed, the host environment sets up the jailed environments. Any process created inside the jail will forever remain there, or so it’s expected. Although there is an administrator for the jail environment, he highly restricted over the entire system. As an example, the root of the jailed environment can shutdown but he cannot restart, since that would require the host’s permission. Outside processes can interfere with the jailed environment, however the reversed is not possible.

Although this environment was presented before virtual machines were introduced, there are similarities between the two systems. In fact, the jail environment is looked at as a hybrid environment due to its striking similarity to virtual machines.

3. VIRTUAL MACHINES

The definition of a virtual machine given in the 1960s is: “software abstraction with the looks of a computer system’s hardware” [9]. Since the 60’s the virtual machine has evolved and separated over three distinct layers. The virtualization layers are: hardware-level virtualization, operating system-level virtualization, and high level language virtualization. Each layer has benefits and drawbacks and provides various methods of process containment. Besides process containment, virtual machines offer different properties such as compatibility and encapsulation. Following is a brief description of the three types of virtual machines.

Hardware-level virtualizations:

In this virtualization technique, the virtual machine sits right on top of the physical hardware. Due to this placement, any software system which is compatible with the original hardware will be compatible with the virtual machine. At a first glance hardware-level virtualization achieves software compatibility, but more importantly it achieves isolation as it will later be shown.

IBM’s VM/370 is a hardware-level virtual machine and it is one of the first implemented virtual machines, dating back to the 1960’s. The VM/370 was designed to service the many activities conducted in the “Cambridge center including operating system research, application development, and report preparation by programmers, scientists, secretaries and managers” [2]. The VM/370 was composed of a Control Program (CP), Conversational Monitor System (CMS) and a Remote Spooling and Communication Subsystem (RSCS). The CP is the “operating system” that simulates multiple copies of the underlying hardware. The CMS is an “operating system” which is controlled by a single user, and the RSCS is an “operating system” responsible for

information transfers among different virtual machines [2]. The combination of these "operating systems" constitutes the virtual memory monitor. Newer hardware level virtual machines include the VMWare ESX/GSX Server, which shares some of the techniques implemented by the VM/370.

Virtual machines of this type are comprised of thin layer of software called the virtual machine monitor (VMM). VMMs run directly on the hardware exporting the abstraction of the virtual machine to look and act like the real hardware. The interface of the VMM supports any application which was designed for the original physical hardware. There are several challenges which must be taken into account. The virtual machine monitor must pass information between the real hardware and the virtual machine efficiently, safely and transparently. The VMM controls the memory management unit, the central processing unit, and various input/output devices. The code required to control these units is extremely small relative to the size of modern operating system. Two consequences result from the small size of this monitoring layer, which on average is more then ten times more compact. First, our confidence increases because the code is smaller and easier to debug and second, it is more efficient.

Realistically, there is some overhead associated with the addition of this layer; however, after considering the security and compatibility issues which are resolved, it is easy to overlook the performance drop. VMMs provide a 1:1 mapping between the actual resources and the virtual machine and therefore most applications will run at the same speed as they would have without the VMM. The only situation where there is overhead, is when the VMM needs to maintain virtual machine isolation or transparency. Transparency is achieved through the effort on the side of the virtual machine to "fool" the application into believing that it's running on real hardware. The software running on top of the virtual machine is clueless to this fact.

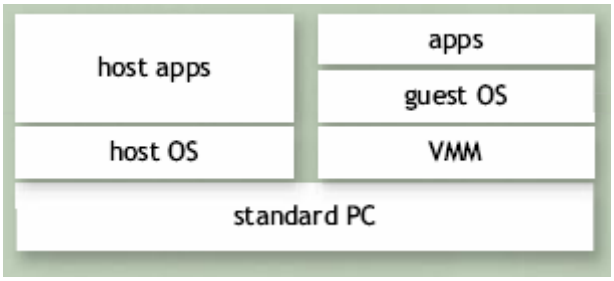

Figure 3. Hardware-level virtualization

Since the VMM stands between the operating system and the hardware it can control every instruction which passes through it. It should be noted that this type of virtualization constitutes the broadest level of process containment. The process here will mostly be the operating system although in the case of embedded devices, there might be an application. In the typical use, the operating system will be the contained process. Hardware-level virtualization is the last means of protection against stray processes assuming that the process has taken over the operating system. As the other virtualization methods are introduced, we can add finer grain to our isolation and therefore our protection against processes.

The security potential of this type of virtualization can be contextualized by providing you with an example of hardware-level virtualization. The National Security Administration uses hardware-level virtualization to keep users from accessing top-secret information. The system is called NetTop. Its users, which freely roam the internet, share the same hardware with those that have access to top-secret information. The top secret information may even reside on the same hardware [9]. Since the 1960's until today, the isolation provided by VMM has been compared to having separate physical machines [9].

Logical partitioning which is the process of running more then one virtual machine through the VMM combined with a technique called server consolidation can create a server hive which can be easily controlled by one management console [12]. The servers are isolated although the share the same hardware. These techniques reduce the number of machines needed, power supply, floor space and makes administration easier. The isolation property of this system assures us that the failure of one server does not compromise the others. That is not to say that the rest of the servers may not be taken down in the same fashion as the original server, a problem which is beyond the scope of this paper.

With logical partitioning and isolation, the VMM can assure that denial-of-service attacks are impossible. If considering the server example described earlier, a misbehaving server cannot starve the others of resources. The VMM is in charge of the CPU, MMU and the virtual machine running the server. Therefore the VMM manages resource allocation, and in theory and practice, it does a better job then most modern operating systems. Because of this isolation capability, each server is guaranteed a level of performance based on user defined policies.

The problem which arises in hardware-level virtualization is the need to define who is in control of hardware resources. At a quick glance, we would say that the virtual machine monitor should be in charge. However the operating system should decide when a memory page is no longer needed and it needs to be paged out. The virtual machine shout not manage the application memory management. That is the task of the operating system if we want to evade suboptimal decision making [9]. To circumvent this problem, the application/operating system would have to posses the knowledge that it runs on a virtual machine. In this case, optimal decisions can be made at the expense of transparency [2].

Operating system-level virtualization:

In this virtualization technique, the virtual machine is situated between the operating system and the application. Since the virtual machine emulates the operating system, any application which is compatible with the underlying operating system is also compatible with the virtual machine. The above example of the FreeBSDs jail system is an example of an operating system-level virtual machine. The jail system is a hybrid system and not a complete virtual machine since it uses existing isolation methods such as chroot and UNIX permission methods which are part of the operating system.

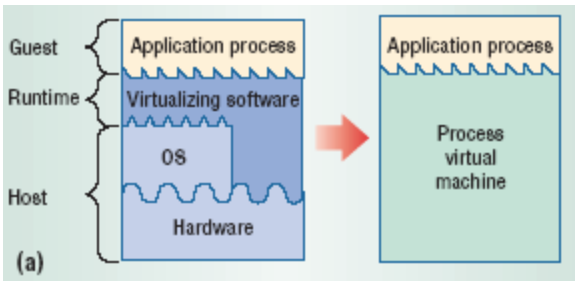

Figure 4. Operating system-level virtualization

Another example of virtualization at the operating system is the VMWare Workstation. It is configured to run on Windows or Linux, and it is capable of abstracting the hardware so that other operating system can be installed. As the figure above demonstrates, the application can be anything which is meant to run in the host operating system. Another name for this type of virtualization is "process virtual machine" [8]. A process virtual machine is obviously not limited to hosting another operating system; they are usually used to setup separate servers such as the popular Apache Server. As in hardware-level virtualization, several benefits are provided by this architecture such as compatibility, emulation, encapsulation, optimization and obviously isolation.

Unfortunately, operating system-level virtualization produces a considerate amount of overhead. The monitoring software is much larger then the virtual memory manager (VMM), which decreases our confidence that the software is bug-free. To give an example of its complexity, consider the steps required for every instruction provided by the monitored process. The monitoring application will fetch a program instruction, decode it, determine its validity and emulate its execution. For every instruction there must be at least three overhead instructions besides any overhead introduced by the operating system.

Another type of virtual machine which resides at this virtualization level is called the recursive virtual machine (RVM). Since the operating system sees the virtual machine as a process, there are no restrictions to what type of children that process may spawn. Therefore, the virtual machine may spawn anther virtual machine and hence the name RVM. This level of recursion can theoretically go on indefinitely, however practically it cannot. The figure below illustrates the exponential increase in complexity when more levels of recursion are introduced [5].

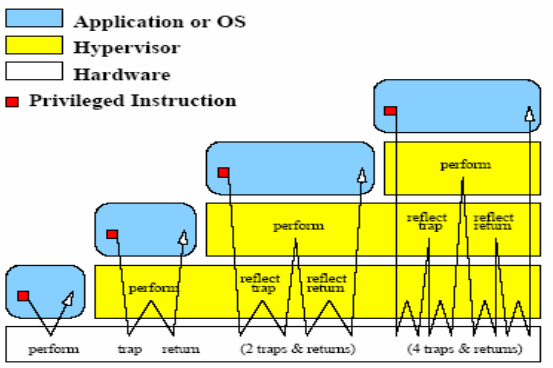

Figure 5. Recursive virtual machine

A different approach to virtual machines is to revert from the horizontal decomposition of operating system functionality and to implement a vertical decomposition [5]. It would contain of stackable virtual machine monitors with individual interfaces. Their model is called nested process architecture, and each virtual monitor has one function. There will be memory nester, a CPU nester, a scheduling nester an I/O nester etc. Each process runs in its own nester. Process isolation is guaranteed on several levels. Since a UNIX process model is used, each process has a virtual-like environment to run in (address space, CPU time slice, etc). The memory nester efficiently provides memory space by using a hierarchical memory remapping. The memory isolation achieved should be mentioned, since the nested process cannot obtain memory from anywhere but the memory nester. Since the nester is using a hierarchical remapping, only memory already available to the memory nester can be distributed. Therefore a process cannot access memory in other locations. This isolation is tightly protected due to the recursive nature of the system. The hierarchical model is also used for process scheduling and various other resources.

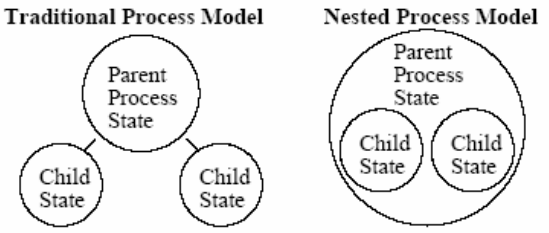

Figure 6. Traditional vs. Nested process model

The encapsulation property of this system has many benefits such as hierarchical resource management, state visibility and reference visibility. Process isolation is strongly dependent upon these properties. Unlike the UNIX model, state visibility allows a parent process to "see" the entire state of a child process sub tree [5]. This heighten visibility allows the parent process to manage the child process state regarding demand paging, debugging, checkpointing, process migration and replication. Another benefit of this system is that processes are treated as memory locations and therefore files. Since the entire sub tree is under the parent's process control, the process can be moved or deleted effortlessly. When a process is caught to misbehave, it is easily contained and removed, including all of its children.

Inter-process communication (IPC) channels, are also under the parent's process full control. Different nesters for different physical components means that the file system nester would interfere only in file system related IPCs. This layering provides not only modularity but containment as well, since each process is monitored by several nesters. CPU scheduling works in a similar manner. Since the parent process is given a finite time slice, it cannot distribute more time to its children then it already has. This keeps rogue processes from denying resources to other processes, which in turn provides more isolation. This type of containment was presented in the hardware-level virtualization as a VMM control.

High level language virtualization:

These virtual machines sit on the application layer since they are viewed by the operating system as an ordinary user application. The strength of this type of virtualization is that any program written and compiled for this virtual machine will run independent of the operating system and thus independent of the physical hardware. Java Virtual Machine is an example of such a virtualization and their motto is: "write once, run anywhere". The java byte code which is generated from a compiled java program should run in any java virtual machine.

The Java Virtual Machine (JVM) is fairly robust. It provides process containment through the use of a sandbox. Due to its popularity I will use the JVM as a synonym for high-level language virtual machine. The JVM has three main components which help it achieve process containment and security. These components are the class loader, class verifier and security manager. Java also has several features such as type-safe reference casting, structured memory access, automatic garbage collection, array bounds checking and null reference checking [7]. These built-in features make it harder for a stray or malicious process to cause damage such as reading or writing to the local disk, making a network connection to other hosts, creating new processes, loading new dynamic libraries and directly calling native methods. Harder is the keyword.

One way that JVM helps protect against a malicious process is through its unspecified memory layout. It is a form of protection because the "attacker" cannot predict how the byte code is laid out; therefore he cannot predetermine jump points. Another hurdle in the way of an attacker is the JVM class-file verifier which is responsible for checking each class file for proper structure. This safety mechanism is not indigenous to the JVM, it can be found on other virtual machines. Further security measures are derived from the class loader. The class loader tries to load every class tree in one name-space. This name-space would be an isolated component and it will not allow access to other isolated component directly. It would have to be given permission to do so, which shows one way JVMs contain processes to their name-space. The name-space containment method is similar to that of the jailed environment from the FreeBSD operating system.

The JVM is not perfect; one of its shortcomings is the potential for object reference leaks. These leaks may allow some programs to prevent other applications from using a particular method [7]. Other problems exist such as the sharing of the events and finalization queues. Their handling of threads has the potential of blocking the execution of another application. Furthermore, the monopolization of resources is still a possible attack against JVMs [3]. One way of successfully containing application processes and achieving isolation is to have the processes run on separate JVMs and therefore separate OS processes. This solution is described in below, and it involves making changes to the JVM to strengths some of its weak points. As always, the downfall is performance.

The MVM or Multitasking Virtual Machine is another type of high-level virtual machine. Its development goals were to give the illusion that a JVM and all its core APIs and mechanisms exist, to restrict any interference among executing application, to be efficient and scalable [3]. To be able to achieve these goals, the virtual machine must thoroughly examine each component of the JVM and restrict its sharing if it determines that it may lead to interference among existing processes. By default, user-defined native

code is non-sharable while some system or runtime classes must be dynamically analyzed to determine their share-ability. If the share-ability distinction is not clear, the component must be copied in order to preserve their original form [3]. Another method of containment can also be achieved by keeping the heap tasks disjoint, however no guarantee is given that the physical data is also separated. The isolation between processes is therefore achieved, since they cannot directly communicate, they cannot share objects and they cannot share memory.

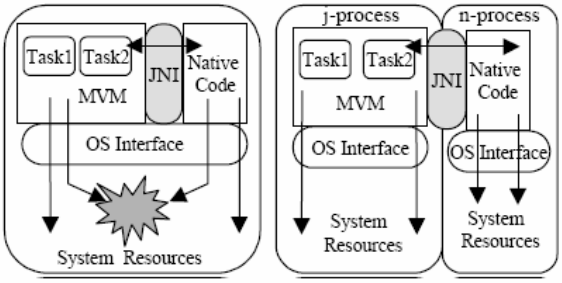

Figure 7. MVM architecture

The only way MVM allows inter-process communication is through standard controllable mechanisms such as sockets. This mechanism is considered safe. Another method that enhances the security capabilities is the non-replication of error buffers, which avoids accidental mixing of error messages. Although this may not be a critical problem, it is nevertheless a problem. A more serious issue is the memory management of the MVM, more specifically recuperating heap memory. When a process becomes uncooperative, MVM has difficulties recover its allocated space. The isolation properties of this virtual memory does guarantee against excessive memory collection, therefore denial-of-service attacks are possible against MVMs. To circumvent some of the problems associated with un-trusted user code and JVM, multitasking virtual machines by default execute user-code in separate processes.

4. CONCLUSIONS

In this paper I've outlined several methods for achieving processes containment. I've defined the term "process containment" by noting the resources which need isolation such as memory, input/output resources and CPU time-slices. Several isolation methods currently implemented in modern operating systems have been introduced. The broad term Virtual Machine was reduced to three different virtualization methods: hardware-level, operating system-level and high-language level virtualization.

Each method of achieving process isolation systematically attempts to solve the memory and resource problems associated with today's multi-user systems. There is no silver bullet and every method has benefits and downfalls. Although the virtual machine implementation has the potential of solving the isolation problem, it introduce overhead. As in most situations, there exists a balance between security/process containment and cost/overhead. Although all the above techniques could be combined to construct a secure environment, the overhead would be tremendous.

5. REFERENCES

1. A. Silberschatz, P.B. Galvin, G. Gagne. *Operating System Concepts (6th edition).* John Wiley & Sons, Inc. Danvers, MA 2003.
2. R. J. Creasy. *The Origin of the VM/370 Time-Sharing System.* Journal of Research and Development, Volume 25, Number 5, Page 483. IBM 1981.
3. G. Czajkowski, L. Daynes. *Multitasking without Compromise: a Virtual Machine Evolution.* ACM SIGPLAN Notices, Volume 36, Issue 11, Pages 125-138. ACM Press, New York, NY 2001.
4. S. Chan-Tin, S. Reiss. *Sandboxing programs.* Computer Science Department, Brown University. http://www.cs.brown.edu/people/sct/ .
5. B. Ford, M Hibler, J Lepreau, P Tullmann, G. Back, S. Clawson. *Microkernels Meet Recursive Virtual Machines.* USENIX 1996, retrieved July 17 from http://www.usenix.org/publications/library/proceedings/osdi96/hibler.html.
6. PH. Kamp, R. Watson. *Building Systems to be Shared Securely.* ACM Queue Volume 2, Number 5, July/August 2004.
7. G. Czajkowski. *Application Isolation in the Java Virtual Machine.* ACM Press, Pages 354-366, New York, NY 2000.
8. J.E. Smith, R. Nair. *The Architecture of Virtual Machines.* IEEE Volume 38, Number 5, Pages 32-38, May 2005.
9. M. Rosenblum. *The Reincarnation of Virtual Machines.* ACM Queue Volume 2, Number 5, July/August 2004.
10. V. Prevelakis, D. Spinellis. *Sandboxing Applications.* Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference, June 25-30, 2001.
11. P. Kamp, R.N.M Watson. *Jails: Confining the omnipotent root.* Retrieved July 18, 2005 from http://docs.freebsd.org/44doc/papers/jail/jail.html
12. M. Rosenblum, J.Chapin, D. Teodosiu, S. Devine, T. Lahiri, A. Gupta. *Implementing Efficient Fault Containment for Multiprocessors.* Communications of the ACM Volume 39, Issue 9, Pages 52-61, September 1996.
13. Sysctl Man Pages from FreeBSD Man Pages, retrieved July 18 from http://www.gsp.com/cgi-bin/man.cgi?section=8&topic=sysctl
14. Procfs Man Pages from FreeBSD Man Pages, retrieved July 18 from http://www.gsp.com/cgi-bin/man.cgi?section=5&topic=procfs