General Virtual Hosting with SoftwarePot

Surányi, Péter

(Doctoral Program in Computer Science)

Advised by Kazuhiko Kato

Submitted to the Graduate School of
Systems and Information Engineering
in Partial Fulfillment of the Requirements
for the Degree of Master of Engineering
at the
University of Tsukuba

January 2004

Abstract

Present-day computers provide sufficient resources to share them between hosting services for multiple Internet sites. This kind of resource sharing is called virtual hosting. However, continuous increase in the amount of network attacks force us to focus on security. In order to realize safe virtual hosting, resources for hosted services need to be isolated. Virtual machine managers (VMMs) are commonly utilized for this purpose, however they require a significant amount of extra work for setting up and resources for running them. OS specific solutions such as FreeBSD jail provide more efficient execution, but still require the preparation of a full basic operating system for each virtual host. This paper proposes a general method for virtual hosting of arbitrary services based on the SoftwarePot Secure Execution System and describes how the virtual hosting facility was implemented. Our approach allows reducing the work required for setting up the virtual hosting system, while maintaining a relatively low runtime overhead.

Contents

List of Figures

Chapter 1

Introduction

At the time when the Internet was beginning to gain wider acceptance, virtually all sites were hosted on dedicated servers, with the single task of servicing requests for that site. However, with the computing power of servers increasing rapidly and the Internet expanding explosively, there was a strong demand to share resources between several sites.

Dedicated servers generally only had a single network interface with a single Internet address assigned to. Interface aliasing, i.e. assigning multiple Internet addresses to the same network interface, a technique now commonly utilized in virtual hosting, is also a relatively new concept. This means that many of the server programs still in use were not designed for being employed on a machine having multiple Internet addresses. These programs accept connections targeted to any of the addresses assigned to the machine, which can easily cause conflicts in an environment running several services for multiple hosts.

Virtual hosting was primarily realized by extending the server software to add support for running services for multiple Internet addresses. This was implemented in many World Wide Web (HTTP) and file (FTP) servers. While embedding virtual hosting support in the server software is arguably the most effective method for sharing resources, it also has its drawbacks. Adding support for multiple addresses increases server code complexity significantly as care must be taken to use the correct address both with incoming and outgoing connections. Since there is no way for the program to decide which of the multiple addresses available to use, it must be specified by the administrator. As different server programs usually require different ways of configuration, this can mean considerable extra work. This approach also brings up security concerns: a malicious attacker may cause failure of services, damage or leakage of information for any of the hosted sites by finding and exploiting a security vulnerability in merely a single one of the hosted sites (e.g. a faulty

CGI script). To address these issues, recent trends favor running services for different virtual hosts in confined environments utilizing virtual machine monitors (VMMs) or isolation services provided by the operating systems.

Virtualization was not considered in the design for the predominant IA32 (also known as Intel x86 or i386) architecture. Because of this, solutions providing complete virtualization (e. g. VMware[19], VirtualPC[15]) require a considerable amount of extra computing resources to achieve this goal, especially in the case of I/O intensive tasks. Besides the runtime overhead, another drawback of the virtual machine approach is that a separate copy of the operating system (guest OS) with its basic services needs to be installed and run within the virtual execution space. This requires additional work for setting up and increases memory and storage requirements as well.

Other VMMs such as Xen[3] and User Mode Linux(UML)[6] provide a limited level of virtualization that is sufficient for their goals. In the case of Xen, this avoids much of the runtime overhead, but it requires significant alteration of the guest OS code. UML's original goal is to run Linux within the user level of another Linux operating system. While originally created for aiding development of kernel code, UML is capable of virtual hosting, but having all I/O data pass through user-mode virtual device drivers increases runtime overhead considerably. These VMMs also don't solve the setup, memory and storage cost problems mentioned before.

UNIX and UNIX-like operating systems provide a system call named `chroot` that is capable to execute programs in a sandboxed file system environment, prohibiting access to all but a single subtree of the file system. Running network services in a `chroot` environment does avoid some security threats but it does not provide support for virtual hosting. To overcome this, version 4.0 of the FreeBSD operating system introduced a new system call `jail`[9] that provides a file system sandbox similar to that of `chroot`, with the additional feature of limiting the addresses used in network connections to a single one, thus making virtual hosting of practically any service possible.

The aim of this research is to provide a method for virtual hosting of arbitrary services that does not depend on OS specific services, and maintains an adequate level of security with a tolerable runtime overhead. Ease of deployment is also considered as a significant aspect. The system presented achieves these goals by employing the SoftwarePot Secure Execution System[10]. In SoftwarePot, programs and the belonging data files are encapsulated in an archive file, and are shown in a virtual file system view upon execution. To meet the purposes of this research, the SoftwarePot system was extended to support virtual

networking.

The author believes that this system, while designed to be highly portable, is a solution both lightweight and flexible enough to successfully address problems of runtime overhead in virtual machine monitors and lack of flexibility in OS-provided features such as `jail`.

The rest of this document is organized as follows. Chapter 2 describes the SoftwarePot Secure Execution System that has been utilized and extended in this research. Chapter 3 explains in detail about the design and implementation of the proposed system. Chapter 4 shows results of evaluation comparisons of the presented system and its competitors. Chapter 5 reviews related work. Finally, Chapter 6 concludes and discusses future work.

Chapter 2

The SoftwarePot Secure Execution System

Installing and running programs on a system implies numerous security risks. Most recent applications consist of several files that need to be installed in the correct location in order for the program to function properly. This is usually performed by installer or package manager programs. In most cases installers are (and generally require to be) executed with super-user or administrator privileges, having full access to the machine's resources. This means that the installer is technically able to arbitrarily read, modify, overwrite or delete files, or initiate network transfers at will. Even for installers with no malicious intent it is often the case that some file to be installed is in conflict with a pre-installed file on the system, in which case either installing the new file or retaining the original one may cause one of the applications fail to operate correctly. In general, the user has to trust both the creator of the installer and the location it obtained it from, having no other way of ensuring that no unwanted effect is caused by the installation process.

SoftwarePot[17] is designed for safely executing programs that are not perfectly trusted by the user. This may include programs of untrusted source or programs that may be suspicable of security vulnerabilities.

2.1 Design

In SoftwarePot, similarly to most installers, the program and the related data files are stored together in a compressed data file, called *pot file*. However, contrary to the traditional installation methods, these files are not installed in the real target location, instead they are extracted in a temporary directory. During execution, the program is run in a special

environment called *pot space*, that provides a virtual file system view, with the files in the archive mapped to their correct locations. Access to all other resources are governed by the system based on the security policy specification provided at execution time. This makes accessing external files possible by mapping them into the virtual file system view. Multiple pot files may share the same pot space, allowing non-essential data files or different versions of libraries to be distributed as separate pot files.

Operating the SoftwarePot system consists of using two commands, namely `makepot` and `execpot`. The command `makepot` creates a pot file based on a specification file describing files to include in the archive and directories to map at runtime etc., passed as a parameter for `makepot`. The command `execpot` extracts and runs pot files. It takes as optional argument specification files that can override the mappings and other settings used for creating the pot file.

The following is the sample specification file that can be used for generating a pot file with a single executable (`myserver`). In this example, the library `libc.so.6` is mapped into the pot space dynamically on the execution site.

```
<Skeleton>
  <Entry path="/mybin/myserver">
    <Arg>arg1</Arg>
  </Entry>
  <StaticFiles>
    <StaticFile vpath="/mybin/myserver">
      <Load protocol="local">
        <Path>/home/user/src/myserver-1.0/myserver</Path>
      </Load>
    </StaticFile>
  </StaticFiles>
  <DynamicFiles>
    <DynamicFile vpath="/lib/libc.so.6">
      <Load protocol="local">
        <Path>/lib/libc.so.6</Path>
      </Load>
    </DynamicFile>
  </DynamicFiles>
</Skeleton>
```

2.2 Implementation

In many operating systems, all access to system resources is performed via system calls. This means that all system resource references can be monitored by intercepting the corre-

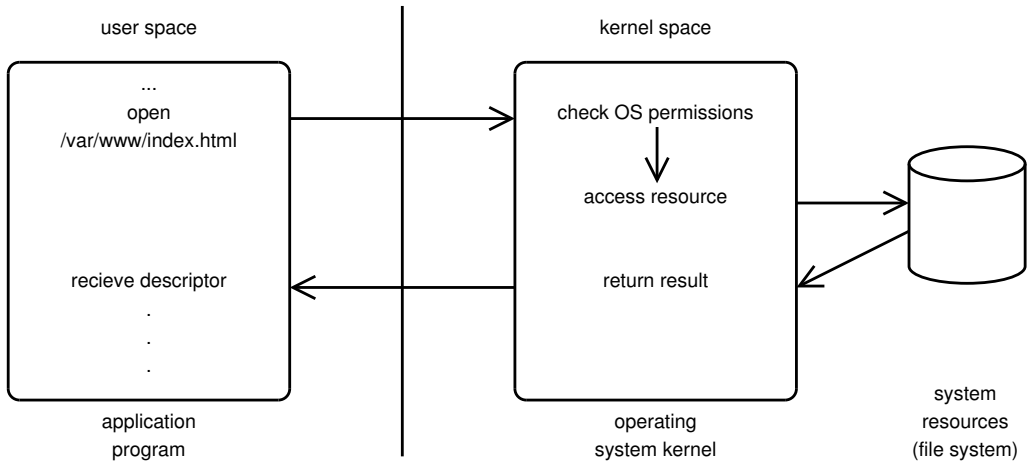

Figure 2.1: System call execution - normal flow

sponding system call(s). This is the approach taken in SoftwarePot to ensure safe execution of programs.

Figure 2.1 and figure 2.2 show the execution of a system call in the native system and in a SoftwarePot environment, respectively. When running in SoftwarePot, every system call that is necessary for enforcing the specified policy is intercepted and its arguments are checked. To achieve the virtual file system view, file path arguments are rewritten to point to the actual location of the file.

SoftwarePot is currently implemented in the following systems:

- Linux 2.4 / Intel IA32
- Linux 2.4 / ARM
- Solaris / SPARC

In order to provide portability and extendibility, SoftwarePot is implemented in a modular manner. Methods for intercepting system calls and controlling the execution of a program can be substantially different from system to system. Therefore, they were implemented in a separate library called the Reference Monitor library. This library provides all interaction between SoftwarePot and the executed program. The Solaris implementation uses the /proc interface for In the case of Linux, the reference monitor library is implemented using a loadable kernel module. Tampering with the kernel is not essential however, this implementation approach was chosen for performance reasons. Indeed, the

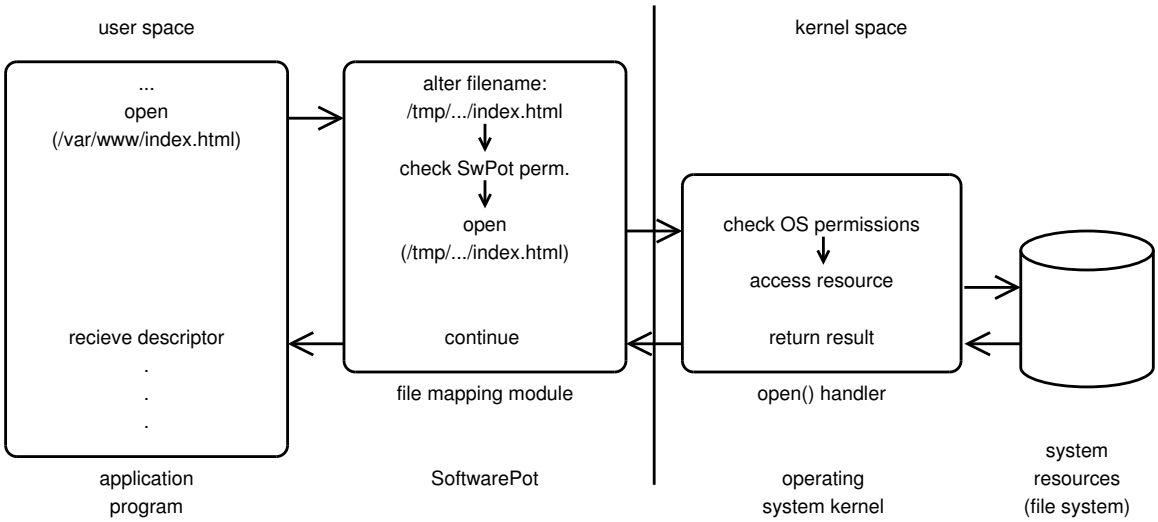

Figure 2.2: System call execution flow with SoftwarePot

reference monitor library is believed to be implementable using the `ptrace` facility found in most UNIX systems.

2.3 Extensibility

SoftwarePot employs modular design in order to provide a high level of extensibility. Two kinds of modules can be used for extending SoftwarePot:

- Protocol modules
  SoftwarePot supports several methods for accessing files in the virtual view. It is possible for example for a file appearing in the virtual view of the Pot process to be retrieved through HTTP or other protocols dynamically when the process accesses the file. Protocol modules can be used to define these kinds of file access methods.

- Runtime modules
  Runtime modules can incorporate in SoftwarePot, intercept any system call and control the execution of the Pot process. This feature is employed in our implementation for virtual hosting.

Chapter 3

Virtual Hosting with SoftwarePot

The amount and severity of network intrusions on the Internet is increasing day by day. Hosting services for several sites on the same machine not only increases the probability of that server being chosen as a target for an attack, but it also widens the area of possible damage in the case that a security breach occurred.

Security vulnerabilities are frequently discovered in server software, operating systems, and other pieces of software commonly employed on server machines. These weaknesses need to be dealt with as soon as possible in order to avoid a possible break-in that may cause failure (downage) of the service or leakage, alteration or modification of the data. Dynamic content created by the site author (e.g. CGI or other WWW scripts) that are not part of the server software itself may also contain security holes and make an otherwise secure server prone to attacks.

In a virtual hosting scenario, with a single server hosting services for multiple sites, this may mean that having merely one service with a security vulnerability may gain access for the attacker to data belonging to all of the hosted sites. Even if there are no security holes in the server software at all, a single faulty user script may cause exposure of data of all of the hosted sites.

Considering these factors, it can be easily seen that resource isolation is a vital problem in virtual hosting. The author believes that the virtual file system view of SoftwarePot is powerful enough to both support a high level of resource isolation and to absorb the differences between hosting sites. This research focuses on extending the capabilities of the SoftwarePot system to make it capable and easier to use for virtual hosting. This chapter describes in detail the problems with the original SoftwarePot system with regard to virtual hosting and how they were addressed.

3.1 Issues

While SoftwarePot is an efficient and convenient system for running applications in confined environments, it was not designed for virtual hosting. Therefore, when applying SoftwarePot for virtual hosting, several problems arise.

In order to provide virtual hosting, we need to have several Internet Protocol (IP) addresses assigned to the machine. Several approaches are available for assigning addresses to virtual machines. VMware, User Mode Linux and Xen utilize a separate virtual network, bridging it to the actual network interface. This method provides a high level of separation at device level, however it requires forwarding packets between the virtual and the actual network interface. FreeBSD jail utilizes IP address aliases, reserving multiple IP addresses on the same interface. This method allows direct delivery of packets to the process by the kernel. This approach is taken in the proposed method as well, as it is believed to be the most efficient one.

Not separating virtual machines at a network device level means that all processes (including SoftwarePot processes) are able to use any of the IP addresses available. In order to use SoftwarePot processes for virtual hosting, this needs to be limited to the address of the virtual site.

Numerous server programs read the hostname of the machine in order to determine the IP address, some of them also communicate this address to their peers. In order to make these programs aware of their virtual IP addresses, it is necessary to provide them with a virtual hostname.

3.2 Design

Server programs work by binding to a specific port number on the machine and waiting for clients to connect to that port. Binding to a port number is achieved by the `bind()` system call. While it is possible to specify an IP address to bind to, numerous server programs default to binding to the so-called "any" address (`INADDR_ANY`), which causes the program to receive connections on all IP addresses assigned to the host. Several programs give an option for specifying the bind address explicitly. However, the means of configuration is different from server to server. In a virtual hosting scenario, it is often more desirable to handle the address configuration centrally in a unified way. [1]

[1] xinetd[16] makes it possible to run several servers with a centralized configuration. This method, however, does not provide any isolation mechanisms between the services.

Besides incoming connections, many server programs, most importantly proxy servers, also initiate connections to external hosts. When creating connections to other machines, the `connect()` system call is utilized. Unless the network socket has been bound to a specific address, which is generally not the case, the outgoing connection is assigned to a local address and port by the kernel. In virtual hosting, having the connection bound to an address that is different from the one assigned to the machine, may cause one of the following troubles.

- The connected host is misinformed about the origin of the connection, causing accesses to be associated with a different virtual host.
- If the machine is connected to several networks, depending on the routing configuration, this may allow clients from one network having the server make a connection to the other network that is supposed to be separated from the client.

3.3 Implementation

To summarize the previous sections, the following main features need to be implemented in order to utilize SoftwarePot for virtual hosting.

- virtual IP address - forcing process to use assigned address
  - incoming connections (`bind()`)
  - outgoing connections (`connect()`)
- virtual hostname - report virtual hostname to the process

This section describes in detail how these features were implemented. Current implementation is based on the Linux/IA32 version of SoftwarePot, however it is designed to be easily portable to other platforms.

3.3.1 Incoming connections

The only thing required for IP address virtualization for incoming connections is to ensure that the argument passed to the `bind()` contains the IP address assigned to the virtual machine. This can be achieved by a simple SoftwarePot module that intercepts calls to `bind()` and rewrites the address in the argument to that of the virtual host. Additionally, for safety reasons, it is implemented so that it issues a warning when the address (before

rewriting) neither matches the "Any" address nor the virtual host address. (The running program may be aware of its virtual address by performing a name server lookup on the virtual hostname.)

3.3.2 Outgoing connections

Implementing virtualization with regard to outgoing connections is a more sophisticated problem. Several approaches can be thought of in order to make sure that outgoing connections are bound to the correct address.

The address a connection is bound to is decided by the operating system kernel based on its routing tables and other internal data. The most straightforward method would be to check and ensure that these tables are correctly setup. However, this procedure would depend highly not only operating system, but, in the case of Linux, also its kernel version and configuration. Also, these kinds of low-level interactions with kernel structures are contrary to our goals of high portability and affinity with the original SoftwarePot architecture.

In order to achieve this goal in the system call level provided by the SoftwarePot infrastructure, it is necessary to make sure that the socket is bound to the local address before `connect()` is called. This could be achieved by forcing the pot-process to call `bind()` for each socket before a `connect()` call is made.

While the Reference Monitor Library makes it possible to intercept calls to `connect()`, by this point of time, the control of the process has been passed over to the kernel system call handler. This means that it is not possible to make the process invoke a `bind()` call before control has returned from the `connect()` call. This makes implementation as a single SoftwarePot module infeasible.

A different method is required for ensuring the binding of outgoing connections. As the interception point provided by SoftwarePot is in a position too late in the control flow, process needs to be overtaken in an earlier.

In UNIX systems, it is rarely the case that a system call is called directly from the code of an application program. Practically all programs access system calls via the wrapper functions of the same name provided in the C library. Also, it is important to note that almost all software for UNIX systems is made available in dynamically linked format[4] [14]. By utilizing the library preload function of recent loaders[12], it is possible to preload a small wrapper library that defines a function called `connect()`. This causes the wrapper library function to be called every time the application makes a `connect()` call. The preloaded

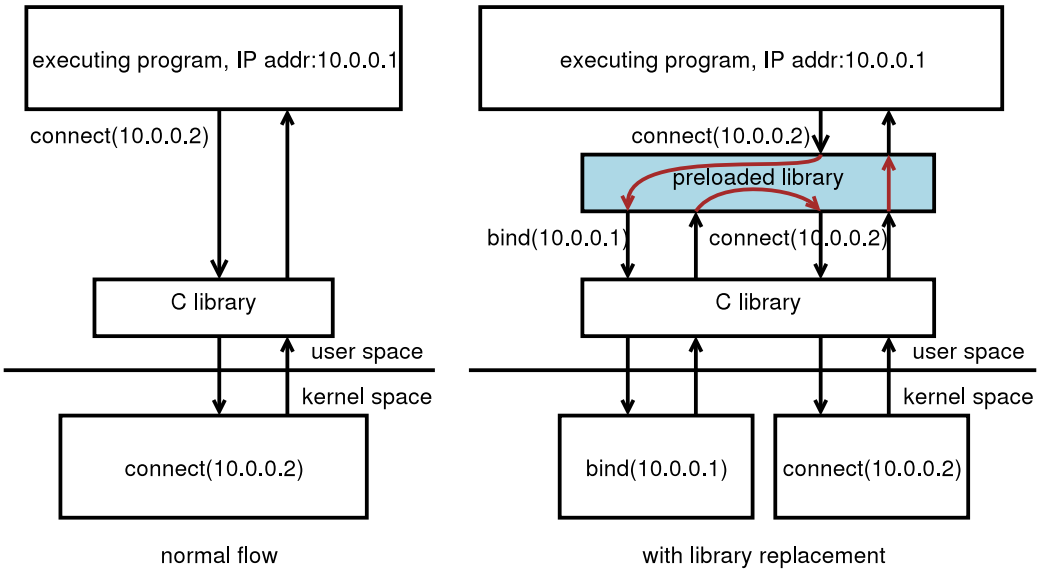

Figure 3.1: Flow of execution with and without library interposition

library can call `bind()` to bind the socket to the local address [2], and then have the linker execute the original `connect()` code in the C library (which finally executes the system call with the same name). Figure 3.1 shows an example of how library interposition changes execution flow.[3] In order for the wrapper library to get preloaded, it is necessary to specify its path in either the `LD_PRELOAD` environmental variable or the `/etc/ld.so.preload` file. As environmental variables can be changed by user processes and are difficult to enforce, the proposed system is implemented by mapping a file containing its virtual path to `/etc/ld.so.preload`.

This approach succeeds at intercepting execution before entering the `connect()` system call for almost all programs. However in a security-conscious scenario as virtual hosting is, it is necessary to consider all cases.

The proposed system is not capable of detecting `connect()` calls invoked from statically-linked binaries, or any other code that bypasses the standard C library function call. To avoid security problems caused by these techniques, a violation check system has been implemented as a separate SoftwarePot module. It takes track of all `connect()`, `bind()`

[2]The library need not be aware of the local address. It is sufficient to give the "any" address as a parameter for bind(), as our SoftwarePot module will rewrite it to the virtual host address before it is executed.

[3]This technique is called library interposition and has been applied to different areas such as software reuse[18], profiling[5] and stack protection[2]

and other socket manipulation system calls and keeps a list of sockets bound to the local IP address. Every time `connect()` is called, it checks if the corresponding socket is on the list of bound ones and in the case it is not, denies execution of the system call.

It is important to note that many server programs don't connect to external hosts or only create connections for looking up hostnames. For these programs using this module and library is not necessary, network safety can be ensured by disabling `connect` completely or restricting to only allow creating connections targeted to the Domain Name Server.

3.3.3 Virtual Hostname

In order to obtain the hostname of the machine they're running on, most programs utilize `gethostname()`. In Linux, this is a C library function that is implemented so that it calls the `uname()` system call to fetch the hostname from the kernel. The virtual hostname feature is implemented by a module intercepting the `uname` system call.

3.4 Features

Two kinds of basic uses can be though for the deployment of this system for virtual hosting.

- Local hosting
  A person or company may want to run one or more of its own services (possibly for different sites) on the same server machine in confined environments.

- Remote hosting
  A service provider may provide virtual hosting services based on the presented system.

In the former case, it is often sufficient to utilize "empty" pot files that contain no files in the archive, instead map all the necessary files into the pot-space at run time. In this case SoftwarePot works mostly as a sandbox tool, making sure that only resources that are necessary (and allowed) are accessed. This reduces the initialization time when executing the pot file by avoiding the extraction phase.

In the latter case, SoftwarePot can provide a means of transferring a whole virtual hosting environment in a single file, that is capable of being run as it is. While this can be achieved by transferring hard disk images for VMMs, they tend to be several hundred megabytes or even more than one gigabyte in size. A SoftwarePot hosting environment only needs to contain the server software (that is usually not more than a few megabytes)

and the belonging data.[4] For users that cannot or do not wish to create pot files for their services, a confined login shell can be set up. This allows the user to log in and install the services at his wish.

[4] In the case that the creator and the hosting sites run different operating system distribution or release, including common libraries in the pot file may be required as well.

Chapter 4

Evaluation

This chapter presents performance evaluation of the proposed method. First, micro-benchmark was conducted to measure the level of runtime overhead for each virtual hosting technique. Then, an application benchmark was performed to evaluate each method in a more realistic environment.

The performance of virtual hosting with SoftwarePot is compared to that of FreeBSD jail, User Mode Linux and Xen. Benchmarks using VMware[19] GSX Server were also performed, however the results are omitted due to the fact that the end user licence does not allow their publication. VMware Workstation 3.2 is the most recent known version that does not prohibit publishing benchmark results, however, that version does not support our hardware utilized in the experiment. However, based on the paper by Barham et al [3], it is known that VMware 3.2 running Linux has approximately 30-80% runtime overhead on network transactions, when compared to native Linux.

The environment used for the evaluation benchmarks is as follows:

- 2 Dell OptiPlex GX260 PC's
  - CPU: Intel Pentium4 2.80 GHz
  - Memory: 1 Gb
  - Network Interface: Intel PRO/1000
- CentreCOM 9606T Gigabit Ethernet Switch

Operating systems, virtualization environments utilized were as follows:

- FreeBSD 4.9-RELEASE (including `jail`)

- Linux 2.4.24
  - User Mode Linux 2.4.19 (tt-mode)
  - SoftwarePot 1.4 with Virtual Networking
- Linux 2.4.24-skas
  - User Mode Linux 2.4.19 (skas-mode)
- Xen 1.0 with XenoLinux 2.4.22

Separate Kernel Address Spaces (skas) mode is a kernel extension to Linux for running User Mode Linux, making it more stable and improve performance to some degree, compared to the Tracing Thread (tt) mode that runs on top of standard Linux kernel. This is discussed in detail in the next chapter.

4.1 Micro Performance Benchmark

The micro-benchmarks were performed using the network performance benchmark utility `netperf`[8]. The following tests were performed:

- TCP_STREAM
  Measures the throughput via TCP connection
- UDP_STREAM
  Measures the throughput via UDP connection
- TCP_RR
  Measures connection latency by sending requests/responses through a TCP connection
- UDP_RR
  Measures connection latency by sending requests/responses through a UDP connection
- TCP_CRR
  Measures connection latency by repeating the sequence of creating a new connection, sending a request and waiting for a response

TCP and UDP differ from each other most importantly in that TCP maintains a connection between the two peers and guarantees the transmission of each data packet by acknowledging and, if necessary, retransmitting it at protocol level, whereas in UDP, data is sent in datagrams without guarantee of receipt, making it the application's responsibility to manage the state of the connection.

Micro benchmark results are shown in Figure 4.1. To minimize the effect of the network device, these tests were conducted on a single machine using the network loopback interface. However, due to the nature of the tests, they show a high fluctuation so they should be regarded as approximate results only. Native Linux and Native FreeBSD results are shown for comparison. SoftwarePot scores better than User Mode Linux in all fields. However, it cannot overdo any of the kernel-based approaches. One exception is the TCP_STREAM test, where the relative performance of SoftwarePot (compared to native Linux) is better than that of jail (compared to native FreeBSD). UDP_STREAM results for the user space based solutions are extremely low, these are caused by the context switches needed to change to user mode and the aforementioned nature of UDP connections.

4.2 Application Performance Benchmark

Application benchmarks were performed utilizing the Apache HTTP Server[1] and the ApacheBenchmark tool. The aim of this benchmark is to measure the performance of each virtualization system in a more realistic environment. In this test, a server is executing different numbers of hosts (1, 2, ..., 32) and multiple clients connect to each virtual host, downloading files of different sizes. The results are shown in Figure 4.2.

As it can be seen from the results, SoftwarePot scales very well with the increasing number of hosts. Especially in the case of small files, its throughput is decreasing at a much slower pace than that of FreeBSD jail.

It should be noted that in the experiment environment employed, above 24 virtual machines, User Mode Linux (tt-mode) started becoming unreliable and virtual hosts would fail without any apparent reason. Because of this, a UML test with 32 virtual hosts couldn't be performed. Also, for FreeBSD jail, a small amount of HTTP requests failed, causing our evaluation method fail to produce reliable results. While in most cases we could avoid this by repeating the experiments, for the case of 1000KB files with 32 virtual hosts this prevented us from obtaining usable results. Out of the virtualization environments tested, only SoftwarePot managed to perform all of the requests without failure.

	throughput (Mbps)		transactions/s			throughput (Mbps)		transactions/s
	TCP_STREAM 56x4	TCP_STREAM 32x4	TCP_RR	UDP_RR 1x1	UDP_RR 512x4	UDP_STREAM	UDP_STREAM	TCP_CRR
Linux	2654.33	2908.58	96249.59	116745.03	108089.54	send: 4282.37 recv: 2955.13	send: 1813.12 recv: 1075.55	22245.28
SoftwarePot	1923.69	2059.11	13645.21	16255.75	13945.57	send: 3083.49 recv: 465.90	send: 1430.95 recv: 118.46	6756.69
Xen	2609.26	2827.96	54941.21	62363.94	60208.65	send: 3231.68 recv: 1671.05	send: 1193.40 recv: 513.40	6365.50
User Mode Linux	435.01	439.41	2958.88	3109.33	3103.51	send: 334.21 recv: 80.80	send: 95.02 recv: 18.53	1867.89
FreeBSD	1861.11	1875.51	53423.05	57994.57	50947.93	send: 1425.06 recv: 1422.22	send: 537.42 recv: 535.55	n/a
FreeBSD jail	815.45	818.15	53775.42	59105.81	52101.75	send: 1438.09 recv: 1434.06	send: 538.14 recv: 535.16	n/a

Figure 4.1: Micro benchmark results

file size	number of hosts	FreeBSD jail	Linux SoftwarePot	UML skas-mode	UML tt-mode
10KB	1	34249.03	12908.90	5671.76	5586.51
	2	29409.94	12089.30	5760.29	5668.48
	4	22938.74	11082.10	5700.74	5653.48
	8	17448.65	10300.64	5635.02	5503.98
	16	12963.81	8463.53	5375.93	5368.25
	24	7163.91	8093.87	5295.18	5153.04
	32	5570.17	7191.31	4889.95	n/a
100KB	1	72594.62	63773.47	17520.37	17583.48
	2	72627.87	62042.45	17962.07	17589.66
	4	73393.05	58797.29	17784.42	17388.11
	8	72187.86	55353.96	17226.86	17314.51
	16	67604.16	48588.34	16262.19	17090.96
	24	62716.31	46278.31	16336.77	16018.64
	32	53143.01	45298.92	15389.73	n/a
1000KB	1	73037.44	88079.51	23531.56	23115.28
	2	74041.53	87239.96	22942.94	22763.40
	4	73944.19	88851.31	22488.64	23594.20
	8	74976.58	89643.04	22414.98	22176.69
	16	78134.91	90801.36	20956.13	20228.85
	24	79700.09	94788.90	20081.95	20774.74
	32	n/a	94144.51	19639.47	n/a

Figure 4.2: Application benchmark – total throughput (Mbps)

Chapter 5

Related Work

As mentioned in the Introduction and the Evaluation chapters, different methods have been proposed and used for virtual hosting. These approaches can be divided in 3 categories:

- Application level support
- Operating system level support
- Virtual Machine Monitors

All of these methods have their advantages and disadvantages. This chapter describes these approaches in detail and compares them to the proposed method.

5.1 Application Level Support

Many server applications support virtual hosting to some degree. In some cases this may only mean that the user can specify which IP address the server should bind to, allowing to run multiple instances of the program for different addresses independently. Some widely used servers like Apache HTTPd and ProFTPd support IP address based virtual hosting that can be set up in their configuration file[1].

All of these approaches lack any kind of resource isolation. This means that if one of the hosted services contains a security hole, it may be abused to leak, delete or modify data belonging to another service.

[1] Apache HTTPd also provides a hostname-based virtualization feature. This makes it possible to serve different pages for requests addressed to different domain names, even if they share the same IP address. This feature depends on a feature of the HTTP/1.1 protocol [7] that forces the client to transmit the server domainname together with the HTTP request. This kind of virtual hosting is application / protocol specific, making it out of scope for the goal of generic virtual hosting

5.2 Operating System Level Support

The most widely known example of operating system level support for virtual hosting is the `jail`[9] facility of FreeBSD. `jail` provides a confined environment that limits the access to the file system to one subtree of it. It also limits many of the privileges of the super-user, like creation of device files, so that services that need to be run at superuser level can be executed securely as well. Linux V-Server[13] is a project that aims to provide similar functions as kernel extensions to Linux.

While these systems provide a virtual hosting environment with very little runtime overhead, they generally require a complete installation of the base OS inside the limited In contrast, the presented system allows more flexible file system mappings, and by tracing the execution at the system call layer, it also makes it possible to limit access to any system resource, even in the case that the service is being executed with superuser privileges.

5.3 Virtual Machine Monitors

Virtualization techniques have been used since the 1970's, primarily on mainframe systems, to achieve a better utilization of the system resources and to provide compatibility with legacy systems. In recent years, the sudden performance increase of personal computers made it possible to run virtual machines with an agreeable performance on common hardware. Software that allows us to execute and manage one or more virtual machines on a system is called a Virtual Machine Monitor (VMM). VMMs can be divided in two major groups (naming borrowed from King et al[11]):

- Type I VMM: runs without a host OS, accessing the host hardware directly
- Type II VMM: runs on top of the host OS, accessing the host hardware indirectly

5.3.1 Type II VMM's

Discussion is started with Type II VMM's as these are the ones more widely known and used. VMware[19] Workstation, GSX Server and Microsoft VirtualPC[15] provide a complete virtualization of a commodity PC, capable of running most common operating systems such as Windows, Linux, FreeBSD etc, running on top of Windows, Linux (for VMware) or MacOS (for VirtualPC). As the Intel IA32 is not completely virtualizable, this requires a large amount of runtime code translation and other tricks, which reduce execution speed considerably, especially with regard to I/O-intensive task such as networking.

User Mode Linux(UML)[6] is a port of the Linux kernel to Linux. It allow running a fully functional virtual Linux machine on a Linux box in. Disregarding the limitations in the host and guest OS, from a user's point of view, it can be used in a similar fashion like full virtualization systems. However, the user mode implementation causes heavy overhead. Kernel support for User Mode Linux is also available in the form of a host kernel patch, that requires recompiling the kernel. The patch implements improved protection for the UML kernel from its own user-mode processes. As our evaluation results show, this does improve stability but does not come with significant performance increase.

5.3.2 Type I VMM's

A Type I VMM, in full control of the system's resources, can provide a significantly higher performance. However it means that the machine has to be dedicated for its purposes. VMware ESX Server[20] is a version of VMware optimized for virtual hosting. Xen[3] provides a paravirtualized x86 environment, which is capable of running modified versions of commodity Operating Systems such as Linux, WindowsXP or NetBSD. The environment and the OS modifications are designed such that the overhead at I/O operations caused by virtualization becomes minimal. However, this approach requires rewriting several thousand lines of the hosted OS. Also, only a limited set of drivers are usable in the rewritten OS. These limitations cause deployment much more effort-costly than that of inserting a loadable kernel module in the case of our system.

Chapter 6

Conclusions and Future Work

While recent computers provide sufficient resources to share them between hosting several sites, network intrusions force people to focus on security. In order to realize safe virtual hosting, resources for hosted services need to be isolated. Virtual machine managers are commonly utilized for this purpose, however they require a significant amount of extra work for setting up and resources for running them. FreeBSD's `jail` and some other OS specific solutions provide more efficient execution, but still require the preparation of a full basic operating system for each virtual machine.

This paper proposed a novel approach for virtual hosting based on the SoftwarePot Secure Execution System. By allowing flexible mapping of parts of the host file system, the presented method avoids the requirement of setting up a complete operating system environment for each virtual hosting environment. SoftwarePot was extended with modules enforcing usage of the virtual host address in order to allow virtual hosting of services that were not design to support it and ensure proper separation of network accesses between the hosted services.

6.1 Future Work

The proposed system showed some weaknesses in handling small files and some cases of network connections. The reason for these overheads is that the SoftwarePot watching process needs to be awaken every time a system call is trapped. It has to be investigated if there are ways to reduce this overhead and how they should be implemented. Recent versions of Linux and FreeBSD support some degree of file system mapping at OS level. There is some chance that access of mapped files can be made faster utilizing these methods. Kernel-based support for speeding up handling of network connections are also being considered.

In the future, it would be interesting to develop this into a system in which network virtualization can be used not only for virtual hosting but developing and testing network based applications. This would require utilizing virtual network interfaces and creating an overlay network by routing them through actual network interfaces.

Acknowledgements

I would like to gratefully acknowledge the supervision of Associate Prof. Kazuhiko Kato. I thank Toshio Hirotsu of NTT Network Innovation Laboratories for the technical discussions on virtual hosting and networking in general. I also thank the people at the Operating System and Systems Software Laboratory. I would especially like to thank the people who developed SoftwarePot and gave me help concerning it: Yoshihiro Oyama (presently at the University of Tokyo), Katsunori Kanda and Osamu Nakamura.

Bibliography

[1] Apache Software Foundation. Apache HTTP Server Project. `http://httpd.apache.org/`.

[2] Arash Baratloo, Navjot Singh, and Timothy Tsai. Transparent run-time defense against stack smashing attacks. In *Proceedings of the USENIX Annual Technical Conference*, 2000.

[3] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, and Rolf Neugebauer. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, pages 164–177, Bolton Landing, NY, USA, October 2003.

[4] Tool Interface Standards (TIS) Committee. *Executable and Linking Format (ELF) Specification*, May 1995. Available from `http://x86.ddj.com/ftp/manuals/tools/elf.pdf`.

[5] Timothy W. Curry. Profiling and tracing dynamic library usage via interposition. In *USENIX Summer*, pages 267–278, 1994.

[6] Jeff Dike. A user-mode port of the linux kernel. In *Proceedings of the USENIX Annual Linux Showcase and Conference, Atlanta, GA*, Oct 2000.

[7] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1, 1999. RFC 2616. Available from `http://www.ietf.org/rfc/rfc2616.txt`.

[8] R. Jones. Netperf. `http://www.cup.hp.com/netperf/NetperfPage.html`.

[9] Paul-Henning Kamp and Robert N. M. Watson. Jails: Confining the omnipotent root. In *2nd International System Administration and Networking Conference*, May 2000.

[10] Kato Kazuhiko and Yoshihiro Oyama. SoftwarePot: An encapsulated transferable file system for secure software circulation. *Lecture Notes in Computer Science*, 2609, February 2003.

[11] Samuel T. King, George W. Dunlap, and Peter M. Chen. Operating system support for virtual machines. In *Proceedings of the 2003 USENIX Technical Conference*, June 2003.

[12] John R. Levine. *Linkers and Loaders*. Morgan Kaufmann, 2000.

[13] Linux V-Server Project. `http://www.linux-vserver.org/`.

[14] H. Lu. ELF: From the programmer's perspective, 1995.

[15] Microsoft. Virtual PC. `http://www.microsoft.com/virtualpc/`.

[16] Jose Nazario. Using `xinetd`. *Linux Journal*, 83:136, 138, 140–141, March 2001. Available from `http://www.linuxjournal.com/article.php?sid=4490`.

[17] Yoshihiro Oyama, Katsunori Kanda, and Kazuhiko Kato. Design and implementation of secure software execution system softwarepot. *Computer Systems*, 19(6):2–12, November 2002.

[18] Yannis Smaragdakis. Layered development with (Unix) dynamic libraries. *Lecture Notes in Computer Science*, 2319:33–45, 2002.

[19] VMware, Inc. VMware. `http://www.vmware.com/`.

[20] Carl A. Waldspurger. Memory resource management in VMware ESX server. In *Proceedings of the 5th ACM Symposium on Operating System Design and Implementation (OSDI-02)*, Operating Systems Review, pages 181–194, New York, December 9–11 2002. ACM Press.