Applying a Multi-level Security Mechanism to a Network Address Translation Scheduler

Arthur McDonald[1] and Haklin Kimm[1]

[1]Computer Science Department
East Stroudsburg University of Pennsylvania
East Stroudsburg PA 18301
E-mail: haklkimm@esu.edu

Haesun Lee[2] and Ilhyun Lee[2]

[2]Department of Science and Mathematics
University of Texas of the Permian Basin
4901 E. University Blvd. Odessa, TX 79762
E-mail: lee_h@utpb.edu

Abstract - In this paper, we consider a scheduling algorithm being applied with multi-level security that allows two or more hierarchical classification levels of information to be processed simultaneously. There are various load scheduling algorithms pre-built into the Linux Virtual Server system that have been tested and proven effective for distributing the load among the real servers. While these algorithms may work effectively, there is no current scheduling algorithm that considers a multi-level security protocol to determine which clients have access rights among the servers.

1. Introduction

In the late 1990s, several open source developers created a software package for Linux to take advantage of scalability and cluster computing. The Linux Virtual Server (LVS) project allows many machines to be networked together into a highly available, high performance virtual server. While it seemingly appears to be a single server to machines outside of the cluster (referred to as clients), the machine may consist of hundreds of separate servers, handling requests for HTTP, FTP, DNS, TELNET or others. When a client connects to the cluster of servers, the connection is processed by the director, or master node of the virtual server. The director sends these connections, referred to as the load, to one of the many servers in the cluster based on some scheduling algorithm.

As of now, there are several load scheduling algorithms already developed and implemented in LVS, including round-robin, least-connection, destination hashing and source hashing. While these scheduling algorithms work well in various situations, there is no way for the administrator to specify, based on some protocol, what security level clients can have to access certain servers in the cluster.

We present the implementation of a multi-level security load scheduling algorithm for Linux Virtual Servers. A multilevel secure (MLS) system is defined as a system with a mode of operation that allows two or more hierarchical classification levels of information to be processed simultaneously [1,3,10,11,15] .

Security administrators can specify which users are authorized to access data and programs depending on their security levels. Using the algorithm that we present in this paper, information can be kept at several levels of security on separate server machines, which will only grant access to the data if the security level of the client machine is identifiable and has been given the proper security clearance. The security levels are statically assigned by the security administrator of the LVS and as of this writing there is no way to bypass the scheduler.

2. Linux Virtual Servers

One of the main concepts in a Linux Virtual Server is the ability for the director to forward data packets sent by the client to the appropriate real server and vice-versa. There are three ways for the director to perform this task: Network Address Translation (NAT), IP-Tunneling and Direct Routing.

2.1 Network Address Translation (NAT)

Network Address Translation (NAT) is a way of manipulating the source and/or the destination address of a data packet. As described in [2, 12], data packets contain both address information, such as the IP address of the source and destination, as well as data. NAT works on the theory that the addressing information is independent from the data. Therefore, a machine stationed somewhere along the route between the source machine and the intended destination could change the addressing information (or any other information in the packet header) without affecting the connection [4, 12].

Traditionally, a machine acting as a Network Address Translator is used to connect an isolated range of private IP addresses, such as a Virtual Private Network (VPN), to an external realm of globally unique IP addresses, such as the Internet. When a private network's internal IP addresses cannot be used outside of the network because they are invalid or for security reasons the addresses must be kept within the private network, Network Address Translation is used [7].

2.2 IP Tunneling

IP Tunneling, sometimes referred to as IP encapsulation, is a technique to encapsulate an IP

packet within IP packets. When a client requests a service from the LVS, a packet destined for the virtual IP address is examined by the director, which chooses the real server using the scheduling algorithm. The director then encapsulates the packet within the data section of another IP packet and forwards it to the destined real server [17]. When the encapsulated packet reaches the real server, it decapsulates the packet and processes the request. However, data sent back to the client from the real server is not routed back through the director; the real server has the IP address of the client, which was in the header of the encapsulated packet. The real server sends all results directly to the client address.

2.3 Direct Routing

In a Linux Virtual Server that uses Direct Routing as the forwarding method, all machines are configured to accept data to the Virtual IP address of the LVS [8, 16]. When a client requests a service, the director examines the packet and determines if it matches a service offered by the LVS. If there is a match, then a real server is chosen using the scheduling algorithm and the data packet is directly forwarded, without any modification, to the physical MAC address of the real server [9]. When the real server receives the forwarded packet, the request is processed by the real server and returned directly to the client, bypassing the director [9].

3. Load Scheduling

One of the most important concepts in a Linux Virtual Server is the concept of load scheduling. As of this writing, the following six scheduling algorithms are implemented in the core LVS distribution [18].

3.1 Round Robin

The round robin algorithm sends incoming requests to the next server in the server list. This algorithm treats all the real servers equally, regardless of the number of incoming connections or response time of the server.

3.2 Weighted Round Robin

This algorithm is similar to the normal round-robin method, except each real server is also assigned a weight. More jobs are given to real servers with a greater weight. The weight factor is adjusted as the load for the server changes. This scheduling algorithm is beneficial when there is a significant difference in the load capacity of the real servers.

3.3 Least Connection

This algorithm distributes requests to the real server with the fewest number of established connections. This algorithm is considered a dynamic scheduling algorithm because it needs to count the live connections for each server dynamically.

3.4 Weighted Least Connection

The weighted least connection scheduling algorithm is a superset of the least connection algorithm, in which weights can also be assigned to real servers. Real servers with a higher weight will receive a larger percentage of the connections.

3.5 Destination/source Hash Scheduling

The destination hash scheduling algorithm uses a static hash table to look up the destination IP address of data packets and assign a real server accordingly. Source hash scheduling is similar to the destination hash algorithm, except that the real server is chosen based on the source IP address of the data packet. Unfortunately, there is not much documentation on the destination or the source hash scheduling algorithms as these are still considered experimental.

4. Scheduling with Multilevel Security

While various load scheduling algorithms may work effectively, there is no current scheduling algorithm that considers a multi-level security protocol to determine which clients have access rights among the servers.

4.1 The Kernel

The kernel actually does very little itself, but provides the tools for which services can be built. In fact, any program running on the computer is forced to interact with the kernel in order to access hardware devices, protecting them from damage [5].

Interaction between applications and the kernel is provided through system calls. User applications and system programs are said to exist and execute in user space. All kernel operations are executed in kernel space.

4.2 Functions

Our algorithm uses a variety of functions to schedule incoming requests across the Linux Virtual Server. Some of these functions are required for compatibility with the already existing LVS code and have only been slightly modified for this project. This section details those functions that have been used for the project.

4.2.1 ip_vs_mls_init_svc

This function is used by the LVS code when setting up the virtual server to use the multi-level security scheduling. When an administrator sets up the Virtual Server, they must specify what type of service (telnet, http, etc…) to set up, along with what scheduling algorithm to use. This function takes a

parameter as a pointer to an ip_vs_service data type, and initializes its scheduling data to the list of destination real servers.

4.2.2 ip_vs_mls_done_svc

This function is called by the LVS code when the service is no longer offered by the Virtual Server. This function simply returns 0 to let the system know that the service is no longer being used.

4.2.3 ip_vs_mls_update_svc

This function is called by the LVS code when the Virtual Server is updated. For example, if a new real server is added to the cluster, then the service of scheduling data needs to be updated to the new list of destinations. This function is identical to ip_vs_mls_init_svc. It takes the ip_vs_service pointer as a parameter and updates the scheduling data, returning 0 to the system.

4.2.4 __init ip_vs_mls_init

This function first initializes the scheduler by calling INIT_LIST_HEAD. It then registers the scheduler with register_ip_vs_scheduler. This function does several things. First, it checks to make sure the scheduler is not already registered in the LVS system. If it is not, then the scheduler is added to a linked list of schedulers available to the Virtual Server and the module use count is incremented by 1. This list holds all the information that the LVS system needs to be able to use the scheduler for load distribution.

4.2.5 __exit ip_vs_mls_cleanup

This function is called by the module_exit function when the module is no longer used by the kernel. Its only job is to unregister the scheduler from the scheduler list by calling unregister_ip_vs_scheduler that deletes the scheduler from the linked list and decreases the module use count.

4.2.6 ip_vs_mls_schedule

This function is the scheduler itself. When a new connection is established between a client and the director, the LVS system calls this function to determine what real server to send the request to. Unlike the previous functions described, which were basic modifications to the initialization and exit function of previous scheduling modules, this function has been totally written from scratch in order to create a multi-level security scheduling algorithm.

This function does several things. First, it establishes the security level of the client by calling ip_vs_mls_get_security_level. Then, it loops through the list of destination real servers and assigns the destination according to the security level. Finally, it returns a pointer to the destination real server's IP address to the LVS code, which handles all communication from there.

4.2.7 ip_vs_mls_get_security_level

This is the function that determines the security level of the client trying to access the Linux Virtual Server. The ip_vs_mls_get_security_level has one basic job, but that job is the most important in this algorithm. While the primary purpose of this function is to clearly determine the security level, it must perform several tasks to accomplish this seemingly trivial job.

First, the ip_vs_mls_get_security_level function is passed to it as a parameter the source address of the client machine, stored as a 4 byte unsigned integer (__u32) and named source_address. It then opens a pair of data files stored in the root directory of the director, **levela.dat** and **levelb.dat**. These files contain the IP addresses for the two security levels that have access to this Virtual Server. The function must then read each of these IP addresses from the data file, convert the 15 byte string data to __u32 unsigned integer and then compare it with the client's source_address.

If there is a match, then the function returns the security level based on what file the IP address was found in, return a 1 if the value was in **levela.dat**, and return a 2 if it was in **levelb.dat**. Finally, if both files are parsed and the IP address was not matched, then return a 0 to let ip_vs_mls_schedule() know that the user should not have any access to the Virtual Server.

5. Implementation

This section describes the lab setup that was used for our Linux Virtual Server, including hardware and software configuration, the steps taken to set up our system, and how the system, including our newly written algorithm was tested and debugged.

5.1 Installing Linux Virtual Server

This section details the steps taken to set up our network lab to act as a Linux Virtual Server. Up to now, the 4 machines were running various operating systems networked together through an Ethernet hub. When the steps in the following sections were completed, this network would be a fully functional Linux Virtual Server and Client, running on a private network within the East Stroudsburg University computer laboratory.

5.1.1 Patching the Director Kernel

The very first step in installing the LVS, was to decide on which kernel to use. After extensive

research on the Linux Virtual Server Mailing List, it was decided to use the 2.4.x stable build of the kernel. More specifically, version 2.4.18 was the kernel version used in the ESU-LVS. We created a directory on our director machine name /apps/download and downloaded the fresh kernel there. Next, the compressed kernel source code was unpacked by running the following command[1]:

[root@director /apps]# gunzip –c /apps/download/linux-2.4.18.tar.gz | xvf –

This command decompressed the kernel code into the directory /apps/linux-2.4.18. This was now the location of the Linux kernel source code.

The next step was to patch the kernel with the Virtual Server code. The kernel was then patched with this new code by executing the following command:

[root@director linux]# patch –p1 </apps/download/linux-2.4.18-ipvs-1.0.4.patch.gz

This command installed the LVS code into the /linux/net/ipv4/ipvs directory of the new kernel.

The final steps to patch the kernel involved installing our new scheduling algorithm code. This required several steps. With these changes made, the newly patched 2.4.18 kernel was ready to be configured with xconfig.

5.1.2 Kernel Configuration on Director

Configuring the kernel to include the Linux Virtual Server code when compiled was done by running **make xconfig** within the **/apps/linux-2.4.18/linux/** directory. Xconfig is an X Windows Graphical User Interface (GUI) front-end for configuration of the kernel options.

For each option in the Xconfig configuration, there are three choices: Y, N or M. Selecting one of these choices either compiles the option into the kernel code (Y), does not compile the option (N), or compiles the option as a module that can be loaded into the kernel later (M). Besides the default options in the configuration file, there were several modifications that we needed to make in order for the Linux Virtual Server code to properly run.

Secondly, the IP: Netfilter Configuration was configured with the following options:

- The final section that needed to be configured was the Linux Virtual Server options:
- The last option was also set. Under Code maturity level options on the main menu, the Prompt for development and/or incomplete code/drivers option was set to Yes.

With all of the configuration settings set, the new configuration was saved and the X**config** application was exited. Next, the command **make dep** was executed. The "make" program is designed to rebuild only those components that need to be rebuilt, based on some notion of what things have changed (since the previous build) and a set of rules expressing "dependencies".

5.1.3 Building Modules and Compiling the Kernel

Once the dependencies were made, the kernel modules were compiled by running **make modules**. This command compiled all of the components selected in the configuration as modules. Next, **make modules_install** was run to copy the modules to the appropriate directory where they can be used by the kernel when needed.

With the modules compiled and installed, the final step was to run the command **make bzImage**. This command compiled all of the kernel code into a compressed kernel image saved in the **/linux/arch/i386/boot** directory of the new kernel. The final step was to edit the **/etc/lilo.conf** file.

5.1.4 Testing the New Kernel

With the new kernel compiled and the boot entry loaded using lilo, the system was rebooted to the new kernel. Testing the kernel simply required monitoring the boot process of the system and insuring that no errors occurred. The system booted normally, and all functionality of the operating system and its applications was intact. We concluded that the new kernel was currently stable and the compilation of the source code was a success.

5.2 Installing IPVSADM

With the newly patched and compiled kernel running the Linux Virtual Server code, the next step was to install the IPVSADM program. IPVSADM is the user application interface to the LVS. IPVSADM is used to run all the administering of the Virtual Server. This includes setting up real servers that the director controls, setting up services that run on the real servers, giving weights to real servers and setting the scheduling algorithm for the LVS.

To install IPVSADM to administer our Virtual Server, we downloaded the appropriate version of the program that matched our kernel version from http://www.linuxvirtualserver.org/software/. The file version we used is named **ipvs-1.21.tar.gz**. We created a temporary directory on the director machine called **/apps/lvs** and saved the compressed IPVSADM program to this directory. Next, we unzipped the file by running the following command:

[root@director/lvs]# gunzip –c /apps/lvs/ipvsadm-1.21.tar.gz | tar xvf –

This command unzipped the files to the **/apps/lvs/ipvsadm-1.21** directory. After navigating

[1] Note: [root@director /apps]# is the command prompt on our Linux machine, the command begins with gunzip.

to this directory, we compiled and installed IPVSADM by running **make install**. The makefile that comes with the IPVSADM code takes care of all the necessary compiling of the code, and installs it for use on the director. To test that the program was properly installed, the **ipvsadm** command was run.

5.3 Setting up LVS for Services (Using IPVSADM)

There are several ways to set up a Linux Virtual Server for Network Address Translation. Looking on the Internet, various setup scripts were found that have been developed by other researchers in the LVS community. While some of these scripts may have worked for this project, it was decided that this project was more specialized and that the LVS should be set up from scratch, rather than relying on a pre-existing setup script. Reading through the Linux Virtual Server mini-HOWTO [14] some basic instructions on setting up the LVS by hand were found.

Although the LVS code had been compiled and was running properly on the director, there were still several adjustments that would need to be made on the director and the real servers in order for communications between the machines would operate correctly.

5.3.1 Setup for the Director

For the director, we developed a script that did the following:

- Turn on IP forwarding
- Turn off ICMP redirects
- Create a Virtual IP address
- Set the default gateway
- Clear the IPVSADM tables

IP Forwarding is turned off because all forwarding of packets from the client to the real servers is done in the LVS code.

An ICMP redirect is an error message sent by a router to the sender of a data packet. If the router believes it received the packet in error, i.e., there exists a better route that the packet could be sent from the sender to the destination, it will notify the sender that it should send subsequent packets through a different gateway [6].

Just by this definition of the ICMP redirect it is seen why it would be necessary to turn these redirects off in the Virtual Server. The client machine does not know that the packets it sends to the director's virtual IP are being forward to one of the various real servers. It would therefore not make any sense if the director sent an ICMP redirect error message to the client, because the client machine believes that the director is the destination of the data packet, where in reality, one of the real servers is the destination. We turn off ICMP redirects to ensure that the director simply forwards the incoming packets, rather than try to inform the client of a better route to take.

The Virtual IP Address is one of the fundamental aspects of a Linux Virtual Server. This is the IP address that the outside client(s) connects to. For our Virtual Server, we set the Virtual IP address to 192.168.0.100.

The default gateway, in a network using subnets like our LVS, is the router that forwards network traffic to a destination outside the LVS. For the director, we set the default gateway to 192.168.0.255. This is just a generic IP address that is on the network that the client and the Virtual IP address are on (X.X.0.X).

Finally, we cleared the IPVSADM tables to prepare them to be filled with our settings for the LVS. This step is technically not really necessary, since there has not been anything set in the IPVSADM tables yet, but it is a good general practice to clear the tables when setting up a Virtual Server from scratch.

5.3.2 Setup for the Real servers

For each of the real servers, a script was run that performed the following tasks:

- Set the default gateway
- Check if the default gateway was reachable
- Check for Virtual IP of director
- Set IP forwarding to Off

The default gateway was set to the real IP address of the director. This forces all outgoing traffic from the real servers to be sent through the director, which is what is intended in the Network Address Translation setup of the LVS.

The check to see if the default gateway is reachable and the check for the Virtual IP are simply issuing PING commands to the respective IP address to make sure a response is received.

Finally, IP forwarding was set to Off in the real servers also, because packets will not be forwarded from real servers. These machines process the data and return it to the director, which forwards the data back to the client.

5.3.3 Using IPVSADM

With the director and real servers set up for Network Address Translation, we then used IPVSADM to set up the Virtual Server. The first step was to set up the type of service that our LVS would offer, along with the scheduling algorithm that should be used. For this thesis, we used the hypertext transfer protocol (http) service with our newly created multi-level security scheduling algorithm (mls). To set up this service we executed the following IPVSADM command:

[root@director /]# ipvsadm –A –t 192.168.0.100:http –s mls

This command sets up the http service using multi-level security scheduling on the Virtual IP address of the director (192.168.0.100). The –A option tells IPVSADM to add the following service, in our case http. The option –t tells the program that it is a TCP service and –s is the option for the scheduler, mls.

Next, the real servers were set up using IPVSADM by executing the following commands:

[root@director/]# ipvsadm –a –t 192.168.0.100:http –r 192.168.1.1:http –m –w 1

[root@director/]# ipvsadm –a –t 192.168.0.100:http –r 192.168.1.2:http –m –w 1

The –a option tells the program that a new real server is being added to the server list, that will use TCP service (-t option). The –r option is followed by the IP addresses of the real server being added. The option –m is needed for Network address translation. This option tells IPVSADM to use masquerading. The final option, -w, sets the weight for the real server. For this LVS, both real servers were assigned a weight of 1.

6. Conclusion

The code for the multi-level security LVS has been added into the kernel of the director, and the ability to grow the cluster of real servers is therefore very difficult. In a real-world application of a multi-level security LVS the ability for more than two real servers would most likely be needed.

Unfortunately, constraints on the equipment available for this project left us to develop our LVS with two real servers, which is not necessarily a negative. One purpose of this project was to show that multi-level security within a Linux Virtual Server is a valuable research area. In today's global political climate, there has been a push for more research in the area of multi-level security. We feel that, although limited, this project has been a valuable step in the development of a full-blown multi-level security system.

One future goal of this project would be to develop an administration tool to allow easier manipulation of the data files. Perhaps the X windows based GUI application that would allow the user to quickly add or remove a machine or group of machines to a proper security level.

With the current high demand for the need of secure data, multiple user systems must be able to support multi-level security. To keep data safe, it must be insured that unauthorized users do not access that which they have not be given permission to access. Looking back on the research, development and results of this project, we feel that it has been a good first step into the area of multi-level security in a Linux Virtual Server environment. While there is obviously much more work to be done in this field, the future of the Linux Virtual Server Project, and multi-level security across LVS and cluster computing in general, looks very optimistic, as well as challenging.

References

[1] Campbell, J., Ehrsam, T., Tinto, M., Williams, J. "The Future of Multi-Level Secure (MLS) Information Systems,"http://csrc.nist.gov/nissc/1998/proceedings/panelF3.pdf .

[2] Comer, D.E. Computer Networks and Internets: Second Edition. 1999. Prentice Hall, Inc. Upper Saddle River, NJ.

[3] Department of Defense. "Multilevel Security in the Department of Defense: The Basics." March 1995, http://nsi.org/Library/Compsec/sec0.html .

[4] Egevang, K. & Francis, P. *The IP Network Address Translator*. RFC 1631. May 1994.

[5] Galvin, Peter & Silberschatz, A., Operating system concepts, 5th edition, John Wiley and Sons Inc., 605 Third Avenue, New York, 1999.

[6] Gill, Stephen. "ICMP redirects are ba'ad, mkay?" 2002, http://www.qorbit.net/documents/icmp-redirects-are-bad.pdf .

[7] Halai, Murtaza and Allahwala, F.K. "End to End Address Transparency," Cornell University.

[8] Hong, Jonghyuck & Kim, D. "Hierarchical Cluster for Scalable Web Servers." Korea University, Seoul, Korea.

[9] Horman, S. "Linux Virtual Server Tutorial." July 2003.

[10] Kang, M. H., Froscher, J. N., and Eppinger, B.J. "Towards an Infrastructure for MLS Distributed Computing," Naval Research Laboratory, Information Technology Division. Washington, D.C. http://www.acsac.org/1998/presentations/wed-a-330-kang.pdf .

[11] Kimm, H., Chester, R., Griffiths, J., & Kertis, K. "Information Retrieval Tool in Heterogeneous Multilevel Secure Environment," Technical Repor, Oak Ridge National Laboratory. July 1993.

[12] Kohler, E., Morris, R., Poletto, M. "Modular Components for Network Address Translation."

[13] Korzeniowski, Paul. "Managing Server Traffic Loads," SW Expert, November 2001.

[14] Mack, Joseph. "LVS-HOWTO." 2004.

[15] Rowton, R. "Security Architecture and Models." February 2004.

[16] Rubini, Alessandro. "Kernel System Calls." http://www.linux.it/kerneldocs/ksys/ksys.html

[17] Zhang, W., Jin, S., Wu, Q. "Creating Linux Virtual Servers," National Laboratory for Parallel & Distributed Processing. Changsha, Hunan 410073, China.

[18] Zhang, Wensong. "Job Scheduling Algorithms in Linux Virtual Server." http://www.linuxvirtualserver.org/docs/scheduling.html.