Automated System Security Process for a Linux Operating System

Group 5 Project

Fulfillment of Group Project Requirement

TABLE OF CONTENT

TABLE OF FIGURES

Abstract

Security vulnerabilities are one of the main concerns in the Telecommunications and Information Systems industry, at present. The use of networked computing devices and services via the Internet or an Enterprise intranet carries an inherent security risk potential (Drepper, 2005). An Operating System (OS) is at the core of each networked computing device, running the kernel, system applications, and other processes critical to the system. A networked computing device is as secure and/or robust as its OS. Historically and, due to the open nature of their initial development, UNIX-like OSs are particularly vulnerable. Linux OS distributions, generally, tend to be "insecure by default" (Ratliff, 2007). For the most part, Linux distributions are built to provide the user with many active applications pre-configured. This characteristic poses a serious problem because it provides hackers and internal intruders with readily available tools to access the system's files, as well as other critical system resources (Ratliff, 2007).

The "best" typical solution to these types of problems has been to audit the source code and fixing the bugs—with enough resources this is achievable. However, we need to also factor-in the human ingredient. Programmers will overlook or miss one problem or another, and there is also new code being created to upgrade or patch systems—hence the likelihood of introducing new or undiscovered bugs is real. The cost of investing on an all-encompassing auditing system is prohibitive (Drepper, 2005). An alternate solution to programming code modification is to harden the operating system. That means making modification to core and system applications, and other system-critical files that contain vulnerabilities—removing, disabling, or modifying those elements of the system files and variables that make it vulnerable (Ratliff, 2007).

This paper presents the process by which our team decided to implement the aforementioned solution with successful results, even when compared to a commercial solution like grSecurity, or to the long preached bastionization process for various Linux distributions usually recommended by Linux experts. We used, as a point of departure, guidelines published by the National Security Agency, to the general public, and added pseudo-automated process to make it easier and more intuitive to the user.

INTRODUCTION

Security vulnerabilities are one of the main concerns in the Telecommunications and Information Systems industry, at present. The use of networked computing devices and services via the Internet or an Enterprise intranet carries an inherent security risk potential (Drepper, 2005). An Operating System (OS) is at the core of each networked computing device, running the kernel, system applications, and other processes critical to the system. A networked computing device is as secure and/or robust as its OS. Historically and, due to the open nature of their initial development, UNIX-like OSs are particularly vulnerable. Linux OS distributions, generally, tend to be "insecure by default" (Ratliff, 2007). For the most part, Linux distributions are built to provide the user with many active applications pre-configured. This characteristic poses a serious problem because it provides hackers and internal intruders with readily available tools to access the system's files, as well as other critical system resources (Ratliff, 2007).

One of the main causes of security vulnerabilities is that bugs in programs are practically unavoidable, especially with programming languages, like C, C++, JAVA, and C-Objective. Those programming languages are used to develop systems kernel implementations, system applications, add-on "apps", as well as other system utilities critical to the computing device (Drepper, 2005). The combination of the Linux OS's openness, with the use of those types of programming languages makes for a vulnerable recipe.The DoD recently released a statement that open source software is just as viable and sometimes even more secure than commercial software. This site has some interesting information, as well as a link to the memo, "Clarifying Guidance Regarding Open Source Software (OSS)" -Jonathan Buck 11/30/09 9:44 PM

THE PROBLEM

According to Drepper, we should look at security problems under two main categories: as either, remotely exploitable or locally exploitable (Drepper, 2005). Locally exploitable security problems imply internal intruders or someone already working in the same environment that the target computer system resides. Their closeness to the system gives the intruder immediate access and the ability to execute code on the target machine. Those security breaches are a lot harder to counter or detect because all the OS's functionalities are at the intruder's disposal allowing him or her to even use self-compiled source code (Drepper, 2005).

Remotely exploitable intrusions are classified as more serious in nature because the intruder can be located anywhere in the machine, or network, or even at the nearest hop router from the Internet cloud, or even the network's DMZ (Drepper, 2005). However, their very same location limits what the intruders can do, since only applications and their associated files accessible via the network services provided by the machine can be exploited—limiting the range and type of exploits at the intruder's disposal (Drepper, 2005). In order for an intruder's attack to be successful, the intruder needs to provide some type of specially designed input. A good example of that would be input that creates buffer overflows—a situation in which meaningless data is stored beyond a predetermined memory space, hence, overwriting whatever

data was located in that memory space and beyond (Drepper, 2005). Buffer overflows can be avoided, for example, with the use of controlled runtimes by conducting memory access checks for validity. However, that is not part of the standard C or C++ runtime, hence, many applications and system processes become vulnerable.

Those inherent bugs can be used by the remote intruders in several manners: 1) A well crafted buffer on the stack can be overwritten when the overwriting action by the invading code causes a naturally occurring jump to a memory space selected by the intruder—hence, overwriting the original return address. 2) A pointer variable may also be overwritten. If the target variable is located in a memory space in front of the overflowed buffer, it could be referenced and overwritten, as well—allowing the intruder to write a value also controlled by the intruder into a particular memory space. 3) A normal variable may be overwritten, hence, modifying the state of a program—resulting on permission escalation or wrong results (Drepper, 2005). Typically, those incidents, when prompted by a naïve user usually end in a system crash. However, remote intruders purposely designed an attack in such a way that instead of crashing, the system's control is taken away from the OS to perform intrusion-derived tasks, or the OS is "coerced" into running unpredictable code (Drepper, 2005).

Another source of danger is the possibility that the intruder take over a process or a thread, in which case the intruder needs to inject malicious code into a running process and cause the code to be executed (Drepper, 2005). Yet, another type of intrusion would be for the intruder to change the behavior of a target application without enabling a security escalation, but instead causing havoc by changing the outcome of a computation—altering a banking or commodities transaction, for example (Drepper, 2005).

A Solution

The "best" typical solution to these types of problems has been to audit the source code and fixing the bugs—with enough resources this is achievable. However, we need to also factor-in the human ingredient. Programmers will overlook or miss one problem or another, and there is also new code being created to upgrade or patch systems—hence the likelihood of introducing new or undiscovered bugs is real. The cost of investing on an all-encompassing auditing system is prohibitive (Drepper, 2005).

An alternate solution to programming code modification is to harden the operating system. That means making modification to core and system applications, and other system-critical files that contain vulnerabilities—removing, disabling, or modifying those elements of the system files and variables that make it vulnerable (Ratliff, 2007). This process is also known as "Removing the Condition" (Drepper, 2005). In general terms, an operating system can be harden following the "Least Privilege Principle"—coined by the National Institute of Standards and Technology (NIST). The Least Privilege Principle describes the behavior of the Role-Based Access Controls (RBAC) developed by NIST for its mainframe systems: "a user should be given no more privilege than necessary to perform a job" (Barkley, J., 1995). A user may play a specific role within an organization according to the type of job that he or she performs.

For example a clerical data-entry person in a Human Resources (HR) department may need read and write privileges for a relational database management system (RDBMS), to input or modify data into specific employee tables. That person's role is restricted to the particular RDBMS tables within the purview of their duties. A CEO may also have access granted to the same RDBMS, however, to conduct more critical business operations. The CFO may also have access granted to the same RDBMS to conduct specific financial transactions. Each user would be granted access and permissions according to the role that they play within the same organization. Each one of them needs only a certain minimum level of privilege to the RDBMS—depending on their specific functions within the organization—to perform their duties. Clearly the CEO would need a higher privilege level of access to the data, than the CFO. Likewise, the CFO would need a higher privilege level, than the HR clerk.

The Principle of Least Privilege requires that a user be given only the privileges necessary to perform a job. In order to establish a least privilege principle approach, it is required that the user's job be identified, as well as understanding the minimum set of privileges required to do that job, while restricting the user to a predetermined domain with those privileges and nothing else (Barkley, J., 1995; Frisch, 2009). The point is to deny users access to transactions that are not required for the performance of their jobs. If granted or circumvented, denied privileges will provide the intruders with an opportunity to bypass organizational security policies, hence, giving access to sensitive areas of the computing environment. Our Team, through the use of RBAC, will attempt to implement an operating system hardening scheme that will enforce the "Least Privilege Principle" (Barkley, J., 1995; Frisch, 2009), in an automated fashion.

APPROACH

Hardening Linux

There is always a trade-off when securing a computer system. On the one hand there is convenience of use and features, and on the other there is overall protection and the removal of risks (Frisch, 2009; Ratliff, 2007; Bauer, 2005). This is a dichotomy of sorts: the easier it is to use, the less secure it is (Frisch, 2009). Regardless of its distribution, a typical Linux OS when installed "out-of-the-box" is rich in convenient features, which one may or may not need to use. Those features, in addition to adding convenience also add vulnerabilities to the system. Hardening a Linux system requires that modifications be made to its system files, some of which are directly associated to the kernel. Hence, care must be taken to address those issues that are absolutely necessary to make the OS more efficient—security wise.

Hardening Steps

Hardening a Linux system is a major undertaking, which involves many activities (Bauer, 2005). Hardening activities can be group according to what needs to be done to the system (Frisch, 2009). For example the following groupings are recommended: preliminary planning and preparation—what to do before getting started; securing and restricting physical access to

the system and its components; installation—this includes the OS itself, as well as any security patches required; securing services and daemons--configuring daemons to reduce security risks; securing local file systems by assigning appropriate permissions according to the "least privilege principle", and removing insecure items; restricting root access—limiting root access to the system console and a small group of system administrators; configuring user authentication and remote access logins by requiring users to authenticate when they log in; setting up remote access specifying user authentication from remote systems and network-based access control; setting up ongoing system monitoring to detect any unauthorized changes to the system—to name a few (Frisch, 2009). Two important points need to be emphasized also: 1) Hardening should be done before the system is placed on the network—systems attached to the network prior to hardening may already be compromised. The hardening process should start from a good system state. The OS should also be reinstalled before any hardening is commenced. 2) Hardening should be based on the "Least Privilege Principle" security model. The OS should be open only as much as it needs to be to function appropriately (Frisch, 2009; NSA, 2007).

Preliminary Planning

For that purpose, our Team selected the National Security Agency's (NSA) "Guide to the Secure Configuration of Red Hat Enterprise Linux 5 (NSA, 2007). The team chose to work on a version of Red Hat Linux, to secure its configuration and perform a "hardening" process of the system. Since the Community enterprise Operating System (CentOS) is a free opensource bit for bit copy of the Red Hat Enterprise Linux Operating System, CentOS 5.3 will be our testing platform. Further, we decided to write scripts and to run the suite testing—script by script—using a VM Ware Virtual Machine (VM).

Securing Physical System

"Securing Physical System" refers to who has access to the machine, and what physical security is in place to prevent physical attacks(removing HDDs, resetting BIOS, inserting optical media or removable drives). This aspect of the process was addressed with the use of a VM in which CentOS 5.3 was installed. Since members of our team were not geographically located near each other, we thought it would be best for each member to run the same process on their respective laptop computers. Most of the members have Microsoft Windows Vista or Windows XP systems. Hence, in order to avoid any permanent changes to our respective systems, it was best to run one or more VM's on each laptop to test the scripts that our programmers wrote. also. The VMs allow us to "roll back" to a clean system. By making a copy of the original VM, we were able to start over (if needed) with a clean installation each time we run a test.

Installing the Operating System

Usually, this would be a lengthy process, where a user decides to install the system "as is", or to install it at the expert level. If one is familiar with the particular Linux distribution, it would be wise to actually take the time to select which packages or modules to install based on what we need to do with it. The recommended practice is to install a minimal OS configuration. That means the minimum necessary to boot the system (Frisch, 2009). Once the system boots

successfully, install additional packages as needed. The end result is an installation that only contains those modules that are actually needed. The main purpose is to prevent intruders from attacking security holes and exploits that otherwise would be available to them (Frisch, 2009).

The next step on this activity is to apply all the security patches that are available. Further, upgrade the installed packages to their newer versions—addressing security holes. Once the packages configuration, updating, and building is completed, the source code needs to be backed up and remove from the system. Although, this is a good practice, using a VM-based CentOS 5.3 installation saved the extra work. Another step on this process is to build a custom kernel for the system. The custom kernel should disable features that are not needed. If it is not there, intruders cannot exploit it (Frisch, 2009). Once the kernel is configured, proceed to build and install it with its associated modules. Boot the kernel to test it functionality. Back up, both the default installed kernel with its modules, as well as the kernel source code directory, and then remove it. For the purpose of our project, our team decided to continue using the standard CentOS 5.3 install. The last step on this activity is to configure the system boot process. Configure the boot loader for automatic booting, disallowing any boot time user intervention. The boot loader must require a password, as well. It is also recommended that a root password be required upon entering single user mode. The ctrlaltdel inittab entry most also be commented out to prevent the key combination from triggering a reboot (Frisch, 2009).

Securing Services

Linux system services include printing, electronic mail, file and web serving, and remote access (telnet, ftp, rlogin, rsh, imap, etc.)—a great deal of the hardening process involves securing them *(Frisch, 2009; Bauer, 2005).* In this process again, the same corollary applies—install only those packages that the system actually needs. However, if removing the service is not possible, then every effort should be made to disable the unneeded service *(Frisch, 2009; Bauer, 2005)*. The following check list gives a general idea of what needs to be done.

- Disable or remove all unneeded services
- Specify logging and access control for all services, allowing only the minimum access necessary
- Use chroot to run a service in a confined directory, when appropriate
- Create a special user to run server processes
- Use secure versions of daemons if they are available
- Restrict access to facilities/services like cron to system administrators
- When allowed by a server, set a limit on the number of daemon processes. This can help prevent denial of service attacks.
- Be sure to secure all services regardless if they are security related or not.

A very important point needs to be made at this time. One needs to find all of the system files associated to those services, before any of those changes are made—in itself not an easy task. Our group carefully studied how the files and services manipulation and/or configuration process is executed, as explained in the NSA guide, within the Linux operating system. This provided us with a deeper understanding of how to make Linux more secure for the user. As of now, the guide explains which files to edit and the manual steps to complete each guideline. These areas include:

- System-wide Configuration
- File Permissions and Masks
- Account and Access Control
- SE Linux
- Network Configuration and Firewalls
- Logging and Auditing
- Services
- Disable All Unneeded Services at Boot Time
- Obsolete Services
- Base Services
- Cron and AT Daemons
- Server Configuration
- Network Configuration

We wrote a suite of multiple bash shells scripts to implement the suggested guidelines established by NSA, making the process of "hardening" the Linux operating system easier for the system administrator. Our script suite ran from the command line, taking user input to define what changes were to be made to the OS. The suite was designed in a modular format to allow for customization and further addition of hardening techniques. The group investigated the usefulness and effectiveness of a kernel security patch such as GrSecurity. This required us to patched and configured security of the kernel, as well as to recompile it.

Process

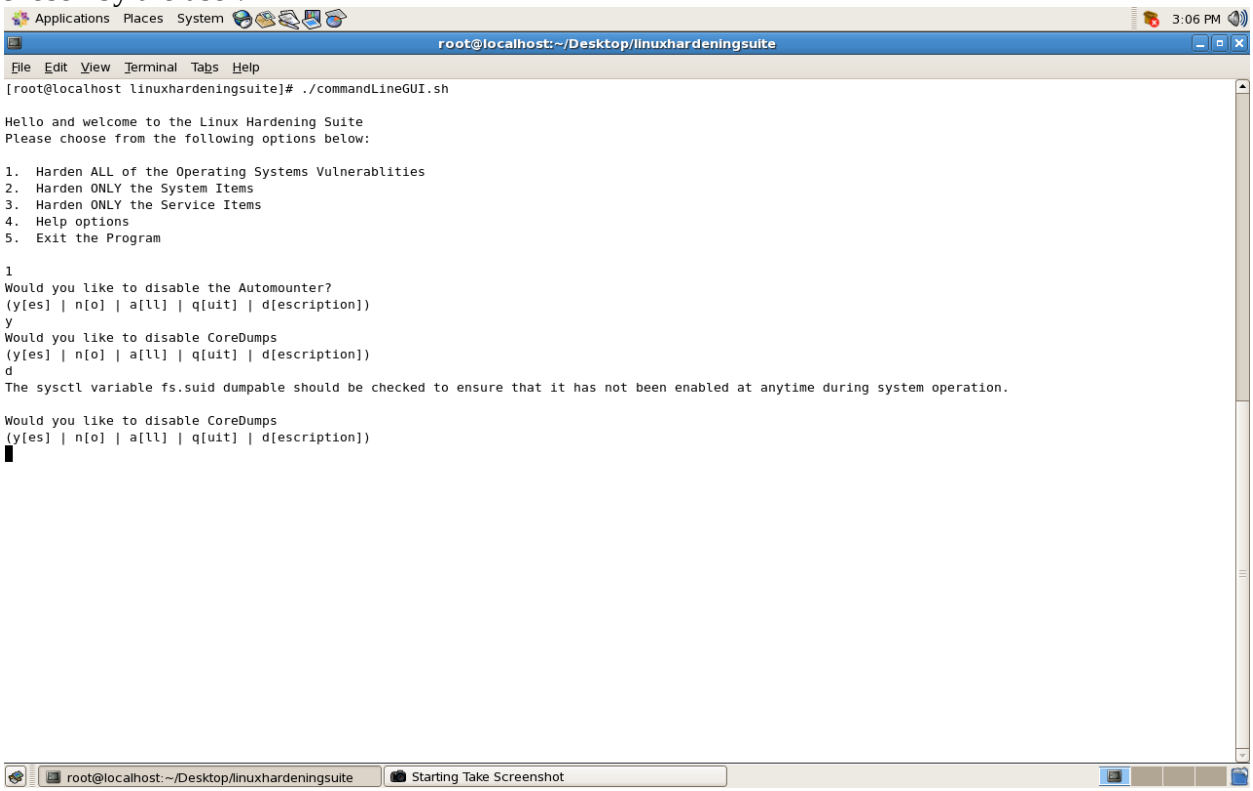
Implemented at: http://code.google.com/p/linuxhardeningsuite/source/browse/#svn/trunk

The suite is designed in a modular fashion to allow for very simplistic extensibility. We also variablized and functionalized as much as possible to allow for simplicity of logic as well as modifications and customizations. To achieve the modular method, a main script looks into predefined folders and runs every script within those folders. The folders allow for organization as well as the ability to run a specific subset of scripts through command line options. The individual scripts that contain the problems and solutions do not actually run any functional code. They only export variables that contain the necessary code, which the main script then evaluates.

The run.sh script is the main logic component that drives the execution of the solution modules. This main script can take command-line options that define if all scripts or specific folders will be. There is a usage function that tells the user when there was a problem trying to interpret the options. The usage can also be displayed by passing the '-h' option. The main script also sets up and writes to a log file that is time stamped in its name to ensure each run of the script generates a unique log. A log function is also used for greater control over what information is logged. Once the script is running, it will enter a set of nested for loops with go through the selected directories and search for scripts to execute. Each script sets global variables which are then utilized within a "runScript" function. This function queries the user for a response to the script defined question. Depending on the user's response, the script then executes a solution or moves on.

We also created the lib.sh script that is sourced during the run.sh script's execution. The lib.sh contains a function used for some of the pre-checks called checkService and a function called changeOrAdd used to execute some solutions. The checkService function takes in a service name and the state (on or off) that is being checked. The function utilizes those arguments and checks to see if it is enabled or disabled on the machine. The changeOrAdd function is used to reduce the complexity of the sed statements needed for some of the solutions. The function takes three arguments, what to search for, what to replace with, and the filename to execute the change on. The find and replace is then executed through a sed substitute statement.

A command line GUI was generated to create easy execution of the scripts. The GUI prompts the user for input on what section of the script they would like to run and the run script is executed accordingly. The suite of scripts will then continue to run, until the exit option is chosen by the user.



Evaluation or Results

The suite of scripts was initialized in root mode, since we anticipated that most of the actions done by the script would require the user to be privileged at that level. The main run.sh script managed the workload, as we expected, conducting the appropriate checks per script, and logged any actions taken by the user, who operated in root mode. The results of our project were on target, as we expected. The entire suite worked as we anticipated. All of the scripts, having being tested in a stand-alone environment, worked appropriately.

We succeeded on automating most of the Linux hardening process, per the NSA guide lines, as it was stated in our proposal. This process makes the daunting task to harden a Linux

system, much easier, in an intuitive manner. Any user with some familiarization with the Linux Operating System would be able to do the same using this suite.

Issues

The biggest issue we dealt with was a direct effect of our code variablization methodology. Nesting variables within variables and the effects of variable expansion forced us to learn which characters to escape as well as different ways of defining or encapsulating variables.We also had to understand how each command is bash can treat regular expressions differently. There were subtle but significant differences when using grep and sed numerous times throughout our implementation. Another common issue was the readability of complex code within variables. One way we solved this was by defining common functions within the lib.sh script. Another way we solved this was placing the code on multiple lines the way code is normally written, however this introduced new problems. With a mixture of backslashes and semicolons, this manner of encapsulating the code greatly increased readability.

Testing grSecurity Approach

After developing our own set of hardening scripts for this project, we wanted to explore other readily available options to compare their use to the use of our own suite. We decided to examine the grsecurity kernel patch set. Our suite of hardening scripts is run by root in user mode while the system is in its full init state. The kernel patches would be applied to the kernel source while the system is not in multi-user mode.

Overview

One of the most attractive features of the Linux operating system is the ability to customize and recompile one's operating system kernel. It allows a security conscious user to have a very tight grasp on the vulnerability of the host. By removing modules that introduce security risks, a Linux system administrator or security professional can reduce the system's overall exposure to malicious attacks.

A Linux system can also be hardened by using a freeware kernel patch such as grsecurity. This patch set can be applied at the time of kernel compilation. The goal of the patch set is to reduce user access to kernel functions and prevent users from having visibility into the operations of the kernel. Using a configuration menu, the user compiling the kernel can toggle a multitude of security features that will enhance the Operating System's resilience to attack.

Features

This patch set offers several useful features such as disabling the loading of kernel modules at run time. This change prevents an attacker from inserting a malicious module into the kernel while the system is operational. Any change to kernel modules would have to be done at boot time. Another grsecurity option is to hide kernel symbols. This setting prevents a user from gaining information such as the list of loaded kernel modules and displaying kernel symbols unless that user has a specific role assigned in grsecurity's Role Based Access Control system.

grSecurity also protects the system from brute force attacks against forking daemons such as sshd or apache by causing the parent process of a process killed or crashed due to illegal instructions to be delayed for a period of 30 seconds. This time delay allows an administrator to determine the reason for the crashed process before allowing the parent to use fork() and exec() to create more children. This will prevent denial of service types of attacks where an attacker could cause a process executing on the system to continue to spawn child processes until the system is out of resources to fork any additional processes and the system must be rebooted.

Another common technique used to exploit an Operating System is to introduce malicious code into the stack or heap of an exploited program. If a page of memory is readable, it is also executable. If the attacker was able to insert the malicious code into the stack or heap, the code could then be executed, as the Operating System does not distinguish malicious instructions from good instructions. An instruction in memory that is being executed by a program with higher privilege than the attacker could write to files that the attacker would otherwise not have

access to and cause the rest of the system to become compromised. grsecurity provides a setting that causes memory to be non-executable and therefore bypassing this vulnerability.

Many exploitation risks come from the fact that certain addresses are known by attackers and can be used to the detriment of the Operating System. A setting is offered in grsecurity to randomize the location of certain dangerous addresses such as the top of a job's kernel stack, the top of the userspace stack, and the base address of the executable.

Problems with the grsecurity approach

A major downfall of the grsecurity approach is that the kernel must be recompiled to implement the security features that it offers. This task was undertaken for the purpose of comparison in the project. The first step that one must do to recompile the kernel is to locate the source code. In order to prevent damage to the current kernel, a new kernel was downloaded from www.kernel.org and the corresponding grsecurity patch downloaded from www.grsecurity.net. The kernel source must then be extracted and the patch applied to the kernel source. After it is patched, the kernel can be configured using a graphical menu by invoking the command *make menuconfig*. Settings that are modified in the menu can also be modified using a configuration file. The relevant settings and values are explained in the GUI. After the configuration is complete, the menu can be exited and the configuration saved. At this point the kernel source is ready to be recompiled. This is accomplished by executing the following three commands: *make all*, *make modules_install*, and *make install*. The task can take several hours depending on how many modules are being included and the speed of the processor. After these commands have finished, all of the necessary files have been created and updated and the only thing left to do is to set the new kernel as the default boot option in the grub menu list. When this is done, the machine can be rebooted and if the configuration was successful, the new hardened kernel will boot with the options specified in the configuration menu.

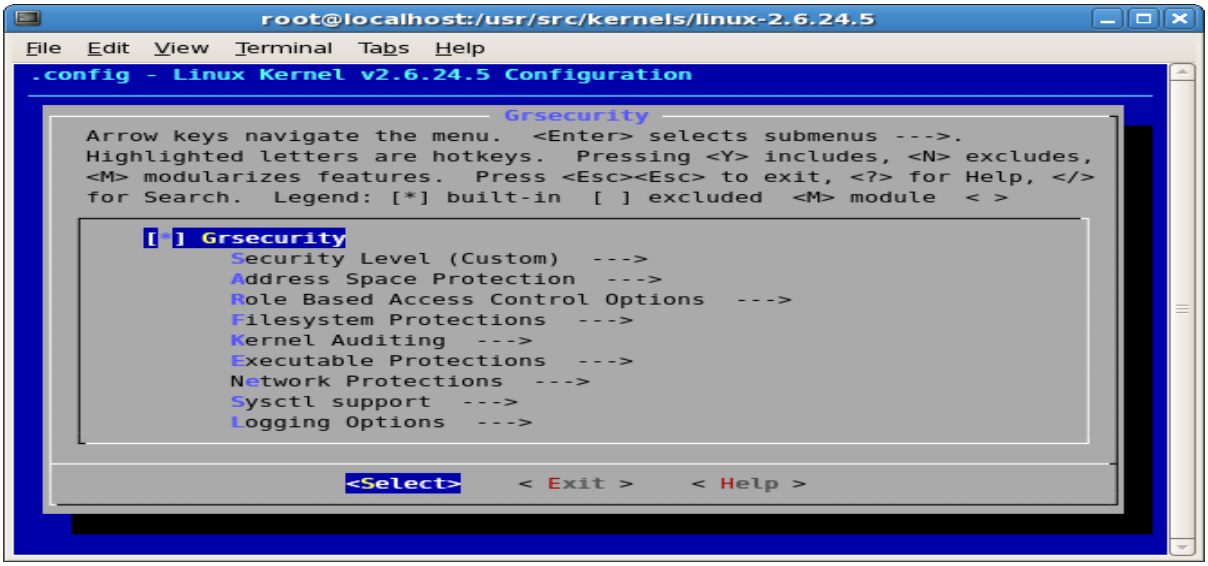


Figure 1 Runnig grSecurity

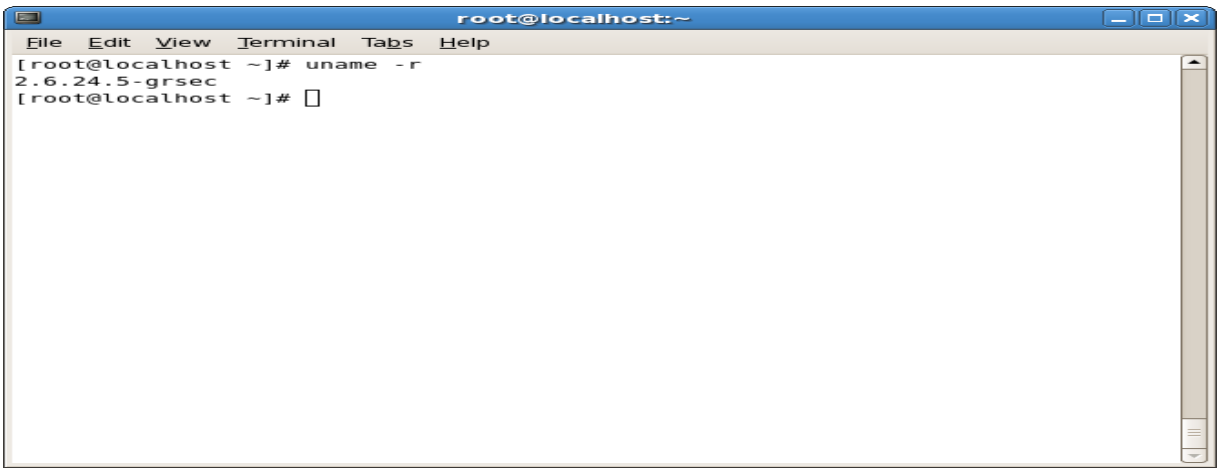


Figure 2 Terminal View

Results of grSecurity Evaluation

Obviously, grsecurity provides many benefits that would not be possible with a runtime security hardening solution. The drastic global changes that this patch set implements is only possible by modifying the kernel source. However, the process is very cumbersome to accomplish this as it requires recompilation of the kernel. For an experienced Linux systems administrator, this can still be a time consuming task, but it is manageable. For a novice or home Linux user or an inexperienced Linux systems administrator, the task can be quite challenging. For this reason, the Linux Security Hardening suite produced by our group is a more feasible option for the general population as it allows an average user to make security enhancements to their system without the requirement of recompiling the kernel source code.

Conclusion

In completing the linux hardening suite project we learned how the different system services such as ssh, firewalls, and SELinux can be toggled on and off and how they greatly effect the security of the operating system. We alsogained knowledge on how substantial changing just one line of a system file can be in hardening a machine. The operating system relies heavily on the system files and minors edit can make or break the configuation. It is our hope that the next team who addresses this issue, can take it to the next level—a full automation of the Linux OS hardening process running straight from the kernel.

References

American Psychological Association. (2001). *Publication manual of the American Psychological Association* (5th ed.). Washington, DC: Author.

Barkley, J. (1995, Jan 9). *hissa.nist.gov/rabac/paper/nnode2.html.* Retrieved November 10, 2009, from hissa.nist.gov: http://hissa.nist.gov/rbac/paper/node2.html

Bauer, M. (2005). Hardening Linux. In M. Bauer, *Linux Server Security* (2nd ed., pp. 1-36). Sebastopol, CA, USA: O'Reilly.

Drepper, U. (2005, Dec. 9). *Security Enhancements in Red Hat Enterprise Linux (beside SELinux).* Retrieved November 15, 2009, from US Department of Defense Intranet: http://www.nsa.gov

Frisch, A. (2009, Feb 19). *wwww.softpanorama.org/Commercial_linuxes/Security/hardening.shtml#Introduction.* Retrieved October 10, 2009, from www.softpanorama.org: http://wwww.softpanorama.org/Commercial_linuxes/Security/hardening.shtml#Introduction

NSA. (2007, Dec. 20). *www.nsa.gov/.* Retrieved October 1, 2009, from http://www.nsa.gov: http://www.nsa.gov

Ratliff, E. (2007, April 30). *Linux Hardening.* Retrieved November 10, 2009, from www.softpanorama.org: http://www.softpanorama.org/Commercial linuxes/Security/ hardening.shtml#Introduction