

# **Securing Linux OS by hardening GRUB boot loader**

# Index

<b>Serial No.</b>	<b>TITLE</b>	<b>Page No.</b>
	<b>Securing Your Linux Box</b>	3
1	Introduction	3
2	Types of Bootloaders for Linux	3
3	Booting process for GRUB	4
4	Booting process after GRUB	4
5	How to Hack	6
6	Method 1	6
7	How to secure	7
8	Method 2	7
9	How to secure	8
10	Problems with GRUB	9
11	Method 3	10
	<b>Understanding GRUB</b>	12
1	Aim	13
2	Design	13
3	Architecture of GRUB	13
4	Flow of execution	13
5	Our required area	14
6	Important files associated with Stage2	14
7	Files to be scrutinized	14
8	Procedure	14
9	Implementation	15
10	Conclusion	17

# SECURING YOUR LINUX BOX

## INTRODUCTION

A running server is prone to various types of security threats. We categorize them as

- Local Security threats
- Remote/Network Security threats

While there are many articles that have explained in great detail the Networking security threats and mechanisms to deal with it, there are a few that have also dealt with the local security aspect. These types of attacks primarily entail gaining illegal root access to an unattended system by running a brute-force password cracking program or by simply rebooting the system and passing some standard arguments to the boot loader program.

One of the frequently used methods to gain the superuser or root access is through the default boot loader that is used to initialize the main OS. In case of Linux, this can be either LILO (Linux Loader) or GRUB (Grand Unified Boot Loader).

The objective of this article is to give an insight into GRUB boot loader particularly from the security point of view. The initial part of the article talks the same while the latter part deals with the analysis and implementation of a patch to GRUB, with which one of the security breaches could be overcome.

The GRUB version 0.93 has been considered in this study and we have been able to successfully create and test the patch on it.

## ***TYPES OF BOOTLOADERS FOR LINUX***

Basically, Linux can be loaded by two types of bootloaders. These are :

1. LILO - Linux Loader
2. GRUB - Grand Unified Bootloader

In this paper we will be mainly focusing on GRUB.

## ***BOOTING PROCESS FOR GRUB***

The GRUB needs the following 4 files for its execution :

1. Stage 1 file
2. Stage 1.5 (optional)
3. Stage 2 file
4. grub.conf file

Once BIOS finishes its POST, the control is passed over to stage 1. The source code for stage 1 will be in assembly language as it is the first program to be executed on the system.

After stage 1 initializes the GRUB and completes its tasks, the control goes to stage 2, with an intermediate stage of stage 1.5.

Stage 2 is completely responsible for loading the image of our Linux kernel. It retrieves the commands to be executed from the file `/boot/GRUB/grub.conf`. This file contains sufficient information for stage 2 to load the kernel. It is this file which provides the path for the kernel image to be loaded, which should be in `/boot`.

It then loads an initial ram disk image with which it can load the kernel. The path for this program, namely `initrd`, could be found in the `grub.conf`. Obviously, all the files needed by stage 2 will be located in `/boot` in the file system hierarchy.

After loading the kernel, the GRUB reaches its end .

## ***BOOTING PROCESS AFTER GRUB***

Now we will see how processes are executed on their way to the shell. Once GRUB is executed, we are in the kernel. The kernel mounts the `init-rd` image (it would be quite puzzling to be calling this process 'mounting' since it is not commonly known around, inspite of being a fact) and gives the control to `init-rd` image, whose function is described below.

The second shell used after the GRUB shell by the system is the NASH shell. It is the smallest shell with a minimal set of commands needed to execute the initial ram-disk image, namely `init-rd`. The essential routines done by the `init-rd` are :

1. To insmod the `ext3.o` module
2. Then loading the VFS of linux

3. Ultimately mounting the root file system in the read-only mode
4. Mounting the Block-devices for transferring other programs (in order to proceed with the boot-up)

Once all init-rd tasks are finished, the control returns back to the kernel and the NASH shell life ends. Then begins the execution of Init. Init is a part of the kernel which is responsible for loading and initializing all the other parts of the system. Init employs certain files for its own operation. The important file is `/etc/inittab`. It directs the init, depending upon the chosen run level. Init first executes the scripts in the file `/etc/rc.sysinit` as indicated by the line

**`si::sysinit:/etc/rc.d/rc.sysinit`**

in the `/etc/inittab` file.

As a part of execution of this script file, init remounts the root file system in read-write mode and then proceeds to execute the scripts of the `rc.sysinit` file.

After completion of the `rc.sysinit` file, init executes the `/etc/rc.d/rc` which is another script file. If any run level parameter is supplied to the kernel from the LILO or GRUB prompt, the `rc` file will be executed so that the required run level is reached in the end. Init achieves this by passing the run level parameter to `rc` while calling it.

The function of the `rc` file is to load the subsystem by stimulating the `start()` procedure associated with each of the applications contained by the `/etc/init.d` directory. One can see all the applications that begin automatically during the booting by visiting the `/etc/init.d` directory.

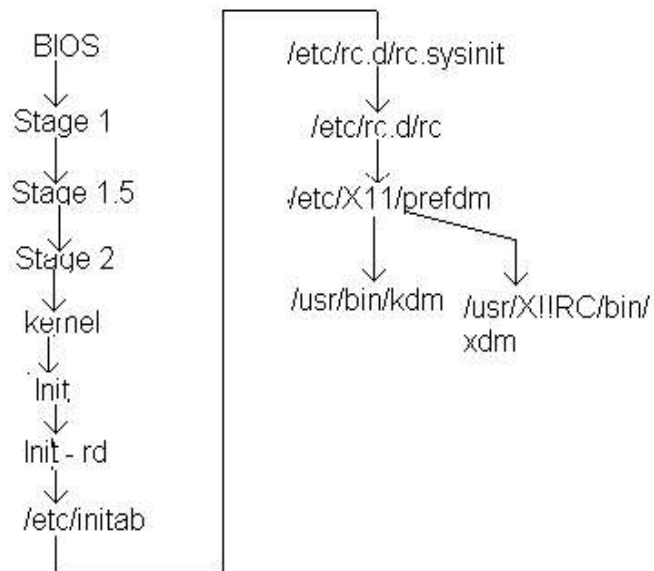
Once the script in the `rc` file is executed, the control returns back to init which finally executes the script file `/etc/X11/prefdm`.

The execution of the `prefdm` file decides which GUI interface (out of GNOME, KDE or XDM) is to be loaded and brings the required X display manager up. The GUI interface is decided based upon the content of the `/etc/sysconfig/desktop` file, in which the variable `DISPLAYMANAGER` could be set to GNOME or KDE or XDM.

- If the variable is set to KDE, `/usr/bin/kdm` will be executed.
- If the variable is set to XDM, `/usr/X11RC/bin/xdm` will be executed.

Now we have trickled our way down to GUI for us to be able to look at the login screen!

All these processes can be summarized in a sequential manner as,



One can further probe into the gaps between prefdm and the login window of GUI, which may be an exercise for the reader.

## HOW TO HACK?

Now, we have gained essential information for gaining root access to the system. So let us now understand hacking.

### Method1

Of course, this is the most well known and easiest method. The procedure is to boot up the system to init level 1 i.e. in the Single user mode. We can do so with :

### **LILO: linux init 1** (for LILO)

for GRUB, simply press 'a' when the boot up screen is displayed.

And at the end of the line displayed, type init 1 and press the enter key.

With this, you will be given the root shell. Now you can change the root password by running the 'passwd' command. After rebooting the system, hacking in would be possible.

It can be quite puzzling for anyone as to why Linux has such an easy and deliberate key to gain the root access. The answer is - even if the root user forgets his own root password, he should be able to recover the system. And the above discussed method ends up being the easiest one for achieving the same.

## **How to secure?**

The only way to secure the system from this menace is by prompting the user for root password even when he boots the system in the single user mode. We can easily achieve this by adding a single line to the /etc/inittab file.

**~~:S:wait:/sbin/sulogin**

This line will instruct the init to prompt for the root password by executing the 'sulogin' program.

## **Method2**

Even if you have protected your system from any unauthorized attacks by the first method, there are always other ways of your system being hacked into. Also we have not yet protected our kernel from receiving arguments through the command line which calls for some more system strengthening to be done.

When the system boots up, in case of LILO, you could pass the argument as

**init=/bin/bash or init=/bin/sh**

or by choosing the 'a' option if you are running GRUB

The init boots up the system and ends up in a Bash shell. You will be now given the root access, though the root file system is mounted in read-only mode, which means

that while you can read everything from the system, you would not be able to do anything other than that.

The root password cannot be changed by anyone! Now the quest is to remount the root file system in read-write mode as follows :

```
#mount -o rw,remount / (for a LILO Booted Kernel)
```

The same command does not work with a GRUB loaded kernel.

```
#mount -n -o remount, rw / (for a GRUB Booted Kernel)
```

Since we are remounting a read-only root file system, nothing could be written to the /etc/fstab and /etc/mstab file. And the -n parameter will further ensure the same.

Since we are remounting the file system which in turn is going to update some flags, the -o parameter is used.

The other parameters are obvious to understand.

Once we have succeeded with the remounting of the file system, just run the 'passwd' command and change the root password.

Now reboot the system and take control of the root.

## How to secure?

1. The only intuitive method to protect the system from this and any other kind of attack is to protect the kernel from getting user-supplied arguments from the LILO or GRUB prompt. This can be done by protecting the LILO or grub.conf file.

Add "**password=urpassword**" to the kernel definition of the lilo.conf or grub.conf.

Change the mode of the file to 600 as

```
#chmod 600 grub.conf or lilo.conf
```

This is mandatory because the file should not be allowed to be modified by non-root users.

You can make the lilo.conf/grub.conf file immutable in another way also

```
#chattr +i/etc/lilo.conf
```

```
#chattr +i/boot/grub/grub.conf (if you are using GRUB)
```

If you want to modify the file, you can set it to the write mode

```
#chattr -i/etc/lilo.conf
```

```
#chattr -i/boot/grub/grub.conf
```

## Problems with GRUB

When you are loading the kernel, the contents of the lilo.conf file will not be displayed nor can you modify them from the LILO prompt.

But in the case of GRUB, you can modify the grub.conf file from the GRUB prompt. When you press 'e' and your GRUB screen appears, the following information from the grub.conf will be displayed and one can modify it as per requirement.

1. The kernel image to be loaded
2. The initrd-image to be loaded
3. If you have protected the grub.conf by password, then you have to remember that the password line will also be displayed (so this is where you have to take care!)

But there still exists one more method to protect the GRUB from displaying the menu of options. When the GRUB screen is displayed, press 'c' to get the GRUB command line.

Now, run the following command :

```
#md5crypt <password>
```

You will be given the encrypted password. Now add the following line to the beginning of grub.conf file :

```
password -md5 <encrypted password>
```

Save the grub.conf file and make it immutable by the methods described earlier and reboot your system.

Now if you want to edit the grub.conf file by pressing 'e' in the GRUB screen, you will be prompted to press 'p' and will be asked for the password.

Ultimately, you have three levels of passwords now.

1. The first password is to be entered to make the grub.conf editable from the GRUB screen.
2. The second password required is for passing arguments to the kernel.
3. The third one is the system's *root* password.

Obviously, the first level protects the rest. If one is able to break the first level, they would be able to break through the other two passwords, according to the design of GRUB.

## Method 3

A powerful method with which any type of highly secured Linux system can be intruded into physically is the **Rescue Mode Method**.

Let us assume that the root user has secured the system with a md-5 encrypted GRUB password. Obviously, the root password would be unknown to the attacker. Also, the BIOS is not password protected.

Now the attacker can gain control over the system by inserting the first installation CD of any Linux distribution, which bundles the Rescue Mode. The BIOS features can be modified so that the system first attempts booting from a CDRom. The procedure for doing so is described below :

- a) Switch on the system and grab the BIOS screen by holding the Del key down
- b) Select the boot option and hold on to the **Boot-device priority menu**
- c) Change the priority to CDRom, as per the vendor of the system
- d) Save the changes and quit from the BIOS

The insertion of installation CD would now take the attacker to the installation interface instead of the usual GRUB. This interface provides the boot prompt as

**boot :**

The Rescue Mode can be attained by issuing the following command :

**boot : linux rescue**

After gliding through a series of stock questions, the system will be mounted temporarily in **/mnt/sysimage** and a Bash shell prompt will be thrown on the screen.

All that is to be done is to change the root to system's actual root. This could be achieved by commanding the shell with

**\$chroot /mnt/sysimage**

The attacker will be given the root privilege of the system now. There are two options for the attacker to proceed with from here.

1. Changing the system's root password by executing the 'passwd' command
2. Editing the **/boot/GRUB/grub.conf** file to eliminate the md-5 encrypted boot password

For any wise attacker, the first method will be the most advantageous. Ultimately, the attacker will reboot the system (this time without any CD) and will log in as the root user.

For safer handling of the system data, one can execute the sync command and change the root to a temporary one and then reboot the system.

As one would realize, this method proves to be omni-potent. Even if the root user has had his system BIOS password-protected, the password can be easily cracked by dismantling the BIOS battery and then remounting it onto the motherboard. This could then be followed by our Rescue Disk method to gain root access to the system.

# UNDERSTANDING GRUB

## [A SECURITY OVERVIEW]

This chapter will help us in figuring out the actual problem and setting up of our targets. The notion embroiled in all sorts of root attacks is to pass arguments to kernel while booting the box. However, GRUB provides a solution to circumvent this. One can do so by adding the line, `password = "ourpasswd"` to the kernel definition in the `/boot/grub/grub.conf` (or) `boot/grub/menu.lst`. A secured kernel definition in `grub.conf` would look like the one shown below :

```
Title ELX Power Desk 4.0
Root (hd0,0)
Kernel /boot/vmlinuz-2.4.20-8elx ro root=LABEL=/
Initrd /boot/initrd - 2.4.20-8elx.img
Password = "cake"
```

The 'password' command could be added anywhere after the 'title' command. An execution of the above script by GRUB would prompt the user for password. As a result, even if the intruder had passed a parameter to kernel with the 'a' or 'e' option of GRUB, he would not be able to proceed without providing the password.

If the intruder uses the 'e' option of GRUB, it would display the entire kernel definition from `grub.conf` in editable form. The resulting interface would display the details as shown below :

```
Title ELX Power Desk 4.0
Root (hd0,0)
Kernel /boot/vmlinuz-2.4.20-8elx ro root=LABEL=/
Initrd /boot/initrd - 2.4.20-8elx.img
Password = "cake"
```

The problem is obvious from the above analysis. The editable feature of GRUB displays the entire kernel definition including the 'password' command and its parameter. Now the attacker could use this displayed password to gain root access to the machine by passing parameters like `"init 1"` (or) `"init=/bin/bash"`.

## AIM

From the above analysis, it is clear that our goals are :

- a) To prevent the password from being displayed by GRUB, when the 'e' option is selected
- b) This should be achieved regardless of the position of the line in which the command 'password' appears in grub.conf.

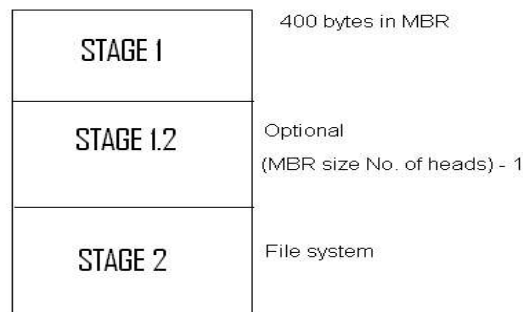
## DESIGN

### ***Architecture of GRUB***

In accordance with the rudimentary concept, the Master Boot Record (MBR) is contained in the boot sector which is the first sector in the disk. MBR comprises of the partition table and the Boot Partition Block (BPB). GRUB is logically divided into three sections :

- Stage 1
- Stage 1.5 (optional)
- Stage 2

Stage 1 will be stored permanently in the MBR. As the MBR is constrained by a size of 512 bytes, stage 1 size is fixed at an approximate 400 bytes.

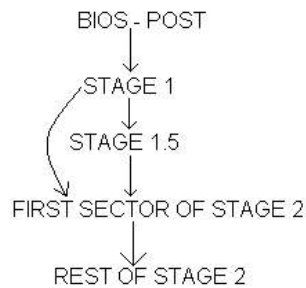


Stage 2 will be kept in the file system. The optional stage 1.5 is placed in the location (MBR size - no. of sectors per head) -1.

### ***Flow of Execution***

Once the BIOS completes its POST test, the control gets transferred to the stage 1 of GRUB. Stage 1 is a self loadable file. The main function of stage 1 is to load the first sector of Stage 2 from the file system. This is achieved by using other information

present in the MBR. The initial part of stage 2 then loads the rest of the stage 2 and performs its stock operations. If the optional stage 1.5 is specified, then stage 1 will load stage 1.5 which in turn will load stage 2. This is the entire flow of execution within GRUB, which can be graphically represented as shown below :



### ***Our Required Area***

Now we have to identify the portion or area of GRUB that requires alteration. With this alteration, GRUB will not display the password in the config file. The function of stage 2 is to retrieve the configuration file from /boot and execute the commands in GRUB shell file. So it is now clear that our focus is to be on stage 2.

### ***Important files associated with Stage2***

Upon probing, the source code of GRUB could be confusing initially. A deeper and iterative probing into the source code would help us find the specific file that we would like to work upon. The files associated with the stage 2 part are organized into a folder named stage 2 which provides room to 60 files (which include 26 Header files, 6 assembly files, 3 make files and 25 C files).

### ***Files to be scrutinized***

Out of all the files mentioned above, only the file **stage2.c** holds the main() function of stage 2. It is this file that retrieves information from the grub.conf file, while the others define base functions like disk-io, file-handling, shell, etc..

### ***Procedure***

1. When each line from grub.conf is read for display, look for the line with the command 'password'

2. If found, block the line from being displayed
3. Modify the code written for the cursor key operation (up and down arrow), so that editable menu is presented to the intruder, creating an illusion of absence of the command 'password' in the kernel definition.
4. Ensure that the navigability through the interface is smooth, and is not affected by the modification in the code.

Thus with our design, the presence of the command 'password' would be completely hidden from the intruder without leaving any traces. The user would thereby realize the presence of the 'password' command only when it is executed. This is the crux of our design.

### ***Implementation***

Actually stage 2 identifies the configuration file as menu.lst. The configuration file is to be named as menu.lst by the **asm.S**, an assembly coded file of stage 2. Hence it will look for the file menu.lst. This method was adopted while developing the original code of GRUB-0.93. But later, with the advent of various patches, the source code was patched so that it recognizes grub.conf as the configuration file.

Most of the Linux distributions incorporating GRUB have had this patch applied to the source code. This is the reason why the configuration file was referred to as grub.conf in the earlier part of this article.

The commands specified in grub.conf or menu.lst are actually functions that are defined and implemented in the file **builtins.c**. Hence, each line in the grub.conf is a call to a function followed by parameter.

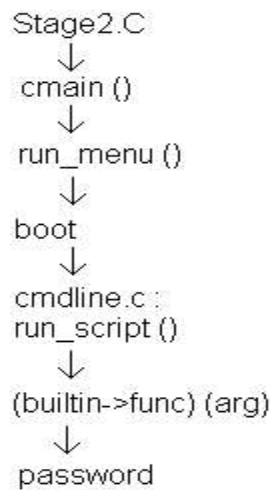
For example, in root (hd0,0), root is the function and hd0 and 0 are the parameters. The functions and its parameters are parsed by the routines **skip\_to()** and **find\_command** and are executed by the function **run\_script()**. These functions are held together in the file **cmdline.c**.

The routines in stage2.c and their declarations are listed below :

- static int open\_preset\_menu (void)
- static int read\_from\_preset\_menu (char \*buf, int maxlen)
- static void close\_prest\_menu (void)
- static char \* get-entry (char \*list, int num, int nested)
- static void print\_entry (int y, int highlight, char \*entry)

- static void print\_entries (int y, int size, int first, int entryno, char \*menu\_entries)
- static void print\_entries\_raw (int size, int first, char \*menu-entries)
- static void print\_border (int y, int size)
- static void run-menu (char \*menu\_entries, char \*config\_entries, int num-entries, char \*heap, int entryno)
- static int get\_line\_from\_config(char \*cmdline, int maxlen, int read\_from\_file)
- void cmain(void)

Out of all these routines, the routines embroiled in the display of information in the editable interface are run\_menu(), print\_entry(), print\_entries() and get\_entry(). These are the functions that need alterations. The overall flow of execution in stage 2 alone could graphically be represented as shown below :



The implementation begins with a simple function which formulates the first step in the algorithm.

```

static int checkpass(char *line)
{
    if ( (*line++ == 'p') && (*line++ == 'a') && (*line++ == 's') && (*line == 's'))
        return 1;
    else
        return 0;
}

```

The other modified functions are as below :

- static void  
print\_entry (int y, int highlight, char \*entry)
- static void  
print\_entries (int y, int size, int first, int entryno, char \*menu\_entries)
- static void  
run\_menu (char \*menu\_entries, char \*config\_entries, int num\_entries, char \*heap,  
int entryno, int ispass, char \*pass, int line)

The following are the changes that were made to the last (the third function listed above) but very crucial function :

1. The first change was made to the code for option 'e'
2. The second change was made using the navigation tools(up and down arrow keys)

## ***CONCLUSION***

Even though we have overcome one of the problems in security, the system could still be attacked, provided the attacker is a Linux-geek. On the other hand, our patch works fine across all versions of GRUB leaving no clue to an ordinary attacker.

Even the rescue mode threat can be mitigated by protecting the initialization process with password. However this is beyond the scope of this article.