Chapter 10 - Server Security and Ansible

The first configuration to be performed on any new server—especially any server with any exposure (direct or indirect) to the public Internet)—is security configuration.

There are nine basic measures to ensure servers are secure from unauthorized access or intercepted communications:

1. Use secure and encrypted communication.
2. Disable root login and use `sudo`.
3. Remove unused software, open only required ports.
4. Use the principle of least privilege.
5. Update the OS and installed software.
6. Use a properly-configured firewall.
7. Make sure log files are populated and rotated.
8. Monitor logins and block suspect IP addresses.
9. Use SELinux (Security-Enhanced Linux).

Your infrastructure is as weak as the weakest server; in many high-profile security breaches, one poorly-secured server acts as a gateway into the rest of the network. Don't let your servers be *those* servers! Good security also helps you achieve the holy grail of system administration—100% uptime.

In this chapter, you'll learn about Linux security and how Ansible helps secure your servers, following the basic topics above.

A brief history of SSH and remote access

In the beginning, computers were the size of large conference rooms. A punch card reader would merrily accept pieces of paper with instructions the computer would run, and then a printer would etch the results into another piece of paper. Thousands of mechanical parts worked harmoniously (when they *did* work) to compute relatively simple commands.

As time progressed, computers became somewhat smaller, and interactive terminals became more user-friendly, but they were still wired directly into the computer being used. Mainframes came to the fore in the 1960s, originally used via typewriter and teletype interfaces, then via keyboards and small text displays. As networked computing became more mainstream in the 1970s and 1980s, remote terminal access was used to interact with the large central computers.

The first remote terminal interfaces assumed a high level of trust between the central computer and all those on the network, because the small, centralized networks used were physically isolated from one another.

Telnet

In the late 1960s, the Telnet protocol was defined and started being used over TCP networks (normally on port 23) for remote control over larger private networks, and eventually the public Internet.

Telnet's underlying technology (a text-based protocol to transfer data between different systems) was the basis for many foundational communications protocols in use today, including HTTP, FTP, and POP3. However, plain text streams are not secure, and even with the addition of TLS and SASL, Telnet was never very secure by default. With the advent of SSH (which we'll get to in a bit), the protocol has declined in popularity for most remote administration purposes.

Telnet still has uses like configuring devices over local serial connections, or checking if a particular service is operating correctly on a remote server (like an HTTP server on port 80, mysql on port 3306, or munin on port 4949), but it is not installed by default on modern Linux distributions.

Plain text communications over a network are only as secure as the network's weakest link. In the early days of computer networking, networks were usually isolated to a specific company or educational institution, so transmitting things like passwords or secrets in plain text using the TCP protocol wasn't such a bad idea. Every part of the network (cabling, switches, and routers) was contained inside a secured physical perimeter. When connections started moving to the public Internet, this changed.

TCP packets can be intercepted over the Internet, at any point between the client and server, and these packets can easily be read if not encrypted. Therefore, plain text protocols are highly insecure, and should never be used to transmit sensitive information or system control data. Even on highly secure networks with properly-configured firewalls, it's a bad idea to use insecure communication methods like plain text rlogin and telnet connections for authentication and remote control.

Try running `traceroute google.com` in your terminal. Look at each of the hops between you and Google's CDN. Do you know who controls each of the devices between your computer and Google? Do you trust these operators with all of your personal or corporate secrets? Probably not. Each of these connection points—and each network device and cable connecting them—is a weak point exposing you to a man-in-the-middle attack. Strong encryption is needed between your computer and the destination if you want to ensure data security.

rlogin, rsh and rcp

`rlogin` was introduced in BSD 4.2 in 1983, and has been distributed with many UNIX-like systems alongside Telnet until recently. rlogin was used widely during the 80s and much of the 90s.

Just like Telnet, a user could log into the remote system with a password, but rlogin additionally allowed automatic (passwordless) logins for users on trusted remote computers. rlogin also worked better than telnet for remote administration, as it worked correctly with certain characters and commands where telnet required extra translation.

However, like Telnet, rlogin still used plain text communications over TCP port 513 by default. rlogin also didn't have many safeguards against clients spoofing their true identities. Some of rlogin's intrinsic flaws were highlighted in a 1998 report by Carnegie Mellon, rlogin: The Untold Story[119].

`rsh` ("remote shell") is a command line program used alongside rlogin to execute individual shell commands remotely, and `rcp` ("remote copy") is used for remote file copies. `rsh` and `rcp` inherited the same security problems as rlogin, since they use the same connection method (over different ports).

SSH

Secure Shell was created in 1995 by Finland native Tatu Ylönen, in response to a password-sniffing attack[120] at his university. Seeing the flaws in plain text communication for secure information, Tatu created Secure Shell/SSH with a strong emphasis on encryption and security.

His version of SSH was developed for a few years as freeware with liberal licensing, but as his SSH Communications Security Corporation[121] began limiting the license and commercializing SSH, alternative forks began to gain in popularity. The most popular fork, OSSH, by Swedish programmer Bjoern Groenvall, was chosen as a starting point by some developers from the OpenBSD project.

OpenBSD was (and still is!) a highly secure, free version of BSD UNIX, and the project's developers needed a secure remote communication protocol, so a few project members worked to clean up and improve OSSH[122] so it could be included in OpenBSD's 2.6 release in December 1999. From there, it was quickly ported and adopted for all major versions of Linux, and is now ubiquitous in the world of POSIX-compliant operating systems.

How does SSH work, and what makes it better than telnet or rlogin? It starts with the basic connection. SSH connection encryption works similarly to SSL for secure HTTP connections, but its authentication layer adds more security:

1. When you enter `ssh user@example.host` to connect to the `example.host` server as `user`, your client and the host exchange keys.

[119]http://resources.sei.cmu.edu/asset_files/TechnicalReport/1998_005_001_16670.pdf
[120]http://en.wikipedia.org/wiki/Secure_Shell#Version_1.x
[121]http://www.ssh.com/
[122]http://www.openbsd.org/openssh/history.html

2. If you're connecting to a host the first time, or if the host's key has changed since last time you connected (this happens often when connecting via DNS rather than directly by IP), SSH will prompt you for your approval of the host key.
3. If you have a private key in your `~/.ssh` folder matching one of the keys in `~/.ssh/authorized_keys` on the remote system, the connection continues to step 4. Otherwise, if password authentication is allowed, SSH prompts you for your password. There are other authentication methods as well, such as Kerberos, but they are less common and not covered in this book.
4. The transferred key is used to create a session key used for the remainder of the connection, encrypting all communication with a cipher such as AES, 3DES, Blowfish or RC4 ('arcfour').
5. The connection remains encrypted and persists until you exit out of the remote connection (in the case of an interactive session), or until the operation being performed (an `scp` or `sftp` file transfer, for example) is complete.

SSH uses encrypted keys to identify the client and host (which adds a layer of security over `telnet` and `rlogin`'s defaults), and then sets up a per-session encrypted channel for further communication. This same connection method is used for interactive `ssh` sessions, as well as for services like:

- `scp` (secure copy), SSH's counterpart to rlogin's `rcp`.
- `sftp` (secure FTP), SSH's client/server file transfer protocol.
- SSH port forwarding (so you can run services securely over remote servers).
- SSH X11 forwarding (so you can use X windows securely).

(A full list of features is available on OpenBSD's site: OpenSSH Features[123]).

The full suite of SSH packages also includes helpful utilities like `ssh-keygen`, which generates public/private key pairs suitable for use when connecting via SSH. You can also install the utility `ssh-copy-id`, which speeds up the process of manually adding your identity file to a remote server.

SSH is fairly secure by default—certainly more so than telnet or rlogin's default configuration—but for even greater security, there are a few extra settings you should

[123]http://www.openbsd.org/openssh/features.html

use (all of these settings are configured in `/etc/ssh/sshd_config`, and require a restart of the `sshd` service to take effect):

1. **Disable password-based SSH authentication**. Even though passwords are not sent in the clear, disabling password-based authentication makes it impossible for brute-force password attacks to even be *attempted*, even if you have the additional (and recommended) layer of something like Fail2Ban running. Set `PasswordAuthentication no` in the configuration.
2. **Disable root account remote login**. You shouldn't log in as the root user regardless (use `sudo` instead), but to reinforce this good habit, disable remote root user account login by setting `PermitRootLogin no` in the configuration. If you need to perform actions as root, either use `sudo` (preferred), or if it's absolutely necessary to work interactively as root, login with a normal account, then `su` to the root account.
3. **Explicitly allow/deny SSH for users**. Enable or disable SSH access for particular users on your system with `AllowUsers` and `DenyUsers`. To allow only 'John' to log in, the rule would be `AllowUsers John`. To allow any user *except* John to log in, the rule would be `DenyUsers John`.
4. **Use a non-standard port**. Change the default SSH port from 22 to something more obscure, like 2849, and prevent thousands of 'script kiddie' attacks that look for servers responding on port 22. While security through obscurity is no substitute for actually securing SSH overall, it provides a slight extra layer of protection. To change the port, set `Port [new-port-number]` in the configuration.

We'll cover how to configure some of these particular options in SSH in the next section.

The evolution of SSH and the future of remote access

It has been over a decade since OpenSSH became the *de facto* standard of remote access protocols, and since then Internet connectivity has changed dramatically. For reliable, low-latency LAN and Internet connections, SSH is still the king due to its simplicity, speed, and security. But in high-latency environments (think 3G or 4G mobile network connections, or satellite uplinks), using SSH is often a painful experience.

In some circumstances, just *establishing a connection* takes time. Additionally, once connected, the delay inherent in SSH's TCP interface (where every packet must reach its destination and be acknowledged before further input will be accepted) means entering commands or viewing progress over a high-latency connection is an exercise in frustration.

Mosh[124], "the mobile shell", a new alternative to SSH, uses SSH to establish an initial connection, then synchronizes the following local session with a remote session on the server via UDP.

Using UDP instead of TCP requires Mosh to do a little extra behind-the-scenes work to synchronize the local and remote sessions (instead of sending all local keystrokes over the wire serially via TCP, then waiting for stdout and stderr to be returned, like SSH).

Mosh also promises better UTF-8 support than SSH, and is well supported by all the major POSIX-like operating systems (it even runs inside Google Chrome!).

It will be interesting to see where the future leads with regard to remote terminal access, but one thing is for sure: Ansible will continue to support the most secure, fast, and reliable connection methods to help you build and manage your infrastructure!

Use secure and encrypted communication

We spent a lot of time discussing SSH's heritage and the way it works because it is, in many ways, the foundation of a secure infrastructure—in almost every circumstance, you will allow SSH remote access for your servers, so it's important you know how it works, and how to configure it to ensure you always administer the server securely, over an encrypted connection.

Let's look at the security settings configured in `/etc/ssh/sshd_config` (mentioned earlier), and how to control them with Ansible.

For our secure server, we want to disable password-based SSH authentication (make sure you can already log in via your SSH key before you do this!), disable remote root login, and change the port over which SSH operates. Let's do it!

[124]https://www.usenix.org/system/files/conference/atc12/atc12-final32.pdf

```
- hosts: example
  tasks:
    - name: Update SSH configuration to be more secure.
      lineinfile:
        dest: /etc/ssh/sshd_config
        regexp: "{{ item.regexp }}"
        line: "{{ item.line }}"
        state: present
      with_items:
        - regexp: "^PasswordAuthentication"
          line: "PasswordAuthentication no"
        - regexp: "^PermitRootLogin"
          line: "PermitRootLogin no"
        - regexp: "^Port"
          line: "Port 2849"
      notify: restart ssh

  handlers:
    # Note: Use 'sshd' for Red Hat and its derivatives.
    - name: restart ssh
      service: name=ssh state=restarted
```

In this extremely simple playbook, we set three options in SSH configuration (`PasswordAuthentication no`, `PermitRootLogin no`, and `Port 2849`) using Ansible's `lineinfile` module, then use a handler we define in the `handlers` section to restart the ssh service.

If you change certain SSH settings, like the port for SSH, you need to make sure Ansible's inventory is updated. You can explicitly define the SSH port for a host with the option `ansible_ssh_port`, and the local path to a private key file (identity file) with `ansible_ssh_private_key_file`, though Ansible uses keys defined by your `ssh-agent` setup, so typically a manual definition of the key file is not required.

Disable root login and use `sudo`

We've already disabled root login with Ansible's `lineinfile` module in the previous section, but we'll cover a general Linux best practice here: don't use the root account if you don't absolutely need to use it.

Linux's `sudo` allows you (or other users) to run certain commands with root privileges (by default—you can also run commands as another user), ensuring you can perform actions needing elevated privileges without requiring you to be logged in as root (or another user).

Using sudo also forces you to be more explicit when performing certain actions with security implications, which is always a good thing. You don't want to accidentally delete a necessary file, or turn off a required service, which is easy to do if you're root.

In Ansible, it's preferred you log into the remote server with a normal or admin-level system account, and use the `sudo` parameter with a value of `yes` with any play or playbook include requiring elevated privileges. For example, if restarting Apache requires elevated privileges, you would write the play like so:

```
- name: Restart Apache.
  service: name=httpd state=restarted
  become: yes
```

Add `become_user: [username]` to a task to specify a specific user account to use with `sudo` (this will only apply if `become` is already set on the task or in the playbook).

You can also use Ansible to control sudo's configuration, defining who should have access to what commands and whether the user should be required to enter a password, among other things.

As an example, set up the user `johndoe` with permission to use any command as root via `sudo` by adding a line in the `/etc/sudoers` file with Ansible's `lineinfile` module:

```
- name: Add sudo rights for deployment user.
  lineinfile:
    dest: /etc/sudoers
    regexp: '^johndoe'
    line: 'johndoe ALL=(ALL) NOPASSWD: ALL'
    state: present
```

If you're ever editing the sudoers file by hand, you should use `visudo`, which validates your changes and makes sure you don't break sudo when you save the changes. When using Ansible with `lineinfile`, you have to use caution when making changes, and make sure your syntax is correct.

Another way of changing the sudoers file, and ensuring the integrity of the file, is to create a sudoers file locally, and copy it using Ansible's `copy` module, with a validation command, like so:

```
- name: Copy validated sudoers file into place.
  copy:
    src: sudoers
    dest: /etc/sudoers
    validate: 'visudo -cf %s'
```

The `%s` is a placeholder for the file's path, and will be filled in by Ansible before the sudoers file is copied into its final destination. The same parameter can be passed into Ansible's `template` module, if you need to copy a filled-in template to the server instead of a static file.

The sudoers file syntax is very powerful and flexible, but also a bit obtuse. Read the entire Sudoers Manual[125] for all the details, or check out the sample sudoers file[126] for some practical examples.

[125] http://www.sudo.ws/sudoers.man.html
[126] http://www.sudo.ws/sudo/sample.sudoers

Remove unused software, open only required ports

Before the widespread use of configuration management tools for servers, when snowflake servers were the norm, servers would become bloated with extra software no longer in active use, open ports for old and unnecessary services, and old configuration settings opening up potential attack vectors.

If you’re not actively using a piece of software, or there’s an obsolete cron task, get rid of it. If you’re using Ansible for your entire infrastructure, this shouldn’t be an issue, since you could just bring up new servers to replace old ones when you have major configuration and/or package changes. But if not, consider adding in a ‘cleanup’ role or at least a task to remove packages that shouldn’t be installed, like:

```
- name: Remove unused packages.
  apt: name={{ item }} state=absent purge=yes
  with_items:
    - apache2
    - nano
    - mailutils
```

With modules like `yum`, `apt`, `file`, and `mysql_db`, a `state=absent` parameter means Ansible will remove whatever packages, files or databases you want, and will check to make sure this is still the case during future runs of your playbook.

Opening only required ports helps reduce the surface area for attack, requiring only a few firewall rules. This will be covered fully in the “Use a properly-configured firewall” section, but as an example, don’t leave port 25 open on your server unless your server will be used as an SMTP relay server. Further, make sure the services you have listening on your open ports are configured to only allow access from trusted clients.

Use the principle of least privilege

Users, applications, and processes should only be able to access information (files) and resources (memory, network ports, etc) necessary for their operation.

Many of the other basic security measures in this chapter are tangentially related to the principle of least privilege, but user account configuration and file permissions are two main areas directly related to the principle.

User account configuration

New user accounts, by default, have fairly limited permissions on a Linux server. They usually have a home folder, over which they have complete control, but any other folder or file on the system is only available for reading, writing, or execution if the folder has group permissions set.

Usually, users gain access to other files and services through two methods:

1. Adding the user to another group with wider access privileges.
2. Allowing the user to use the `sudo` command to execute commands and access files as `root` or another user.

For the former method, please read the next section on file permissions to learn how to limit access. For the latter, please make sure you understand the use of sudoers as explained earlier in this chapter.

File permissions

Every Ansible module that deals with files has file ownership and permission parameters available, including `owner`, `group`, and `mode`. Almost every time you handle files (using `copy`, `template`, `file`, etc.), you should explicitly define the correct permissions and ownership. For example, for a configuration file (in our example, the GitLab configuration file) that should *only* be readable or writeable by the root user, set the following:

```
- name: Configure the GitLab global configuration file.
  file:
    path: /etc/gitlab/gitlab.rb
    owner: root
    group: root
    mode: 0600
```

File permissions may seem a bit obtuse, and sometimes, they may cause headaches. But in reality, using octal numbers to represent file permissions is a helpful way to encapsulate a lot of configuration in three numbers. The main thing to remember is the following: for each of the file's *user*, *group*, and for *everyone* (each of the three digits), use the following digits to represent permission levels:

```
7: rwx (read/write/execute)
6: rw- (read/write)
5: r-x (read/execute)
4: r-- (read)
3: -wx (write/execute)
2: -w- (write)
1: --x (execute)
0: --- (no permissions)
```

Basically, 4 = read, 2 = write and 1 = execute. Therefore read (4) and write (2) is 6 in the octal representation, and read (4) and execute (1) is 5.

Less experienced admins are overly permissive, setting files and directories to `777` to fix issues they have with their applications. To allow one user (for example, your web server user, `httpd` or `nginx`) access to a directory or some files, you should consider setting the directory's or files' *group* to the user's group instead of giving permissions to *every user on the system*!

For example, if you have a directory of web application files, the *user* (or in Ansible's terminology, "owner") might be your personal user account, or a deployment or service account on the server. Set the *group* for the files to a group the web server user is in, and the web server should now be able to access the files (assuming you have the same permissions set for the user and group, like `664`).

Update the OS and installed software

Every year, hundreds of security updates are released for the packages running on your servers, some of them fixing critical bugs. If you don't keep your server software up to date, you will be extremely vulnerable, especially when large exposures like Heartbleed[127] are uncovered.

At a minimum, you should schedule regular patch maintenance and package upgrade windows, and make sure you test the upgrades and patches on non-critical servers to make sure your applications work *before* applying the same on your production infrastructure.

With Ansible, since you already have your entire infrastructure described via Ansible inventories, you should be able to use a command like the following to upgrade all installed packages on a RHEL system:

```
$ ansible webservers -m yum -a "name=* state=latest"
```

On a Debian-based system, the syntax is similar:

```
$ ansible webservers -m apt -a "upgrade=dist update_cache=yes"
```

The above commands will upgrade *everything* installed on your server. Sometimes, you only want to install security-related updates, or exclude certain packages. In those cases, you need to configure `yum` or `apt` to tell them what to do (edit `/etc/yum.conf` for `yum` on RHEL-based systems, or use `apt-mark hold [package-name]` to keep a certain package at its current version on Debian-based systems).

Automating updates

Fully automated daily or weekly package and system upgrades provide even greater security. Not every environment or corporation can accommodate frequent automated upgrades (especially if your application has been known to break due to past

[127] http://heartbleed.com/

package updates, or relies on custom builds or specific package versions), but if you do it for your servers, it will increase the depth of your infrastructure's security.

As mentioned in an earlier sidebar, GPG package signature checking is enabled by default for all package-related functionality. It's best to leave GPG checks in place, and import keys from trusted sources when necessary, *especially* when using automatic updates, if you want to prevent potentially insecure packages from being installed on your servers!

Automating updates for RHEL systems

RHEL and CentOS 6 and 7 (and older versions of Fedora) uses a cron-based package, `yum-cron`, for automatic updates. For basic, set-and-forget usage, install `yum-cron` and make sure it's started and set to run on system boot:

```
- name: Install yum-cron.
  yum: name=yum-cron state=present

- name: Ensure yum-cron is running and enabled on boot.
  service: name=yum-cron state=started enabled=yes
```

Further configuration (such as packages to exclude from automatic updates) is done in the `yum.conf` file, at `/etc/yum.conf`.

For RHEL and CentOS 8 and later (and the latest versions of Fedora), you would need to install `dnf-automatic` instead of `yum-cron`, and the service name that you need to make sure to start is `dnf-automatic-install.timer` (instead of `yum-cron`).

Automating updates for Debian-based systems

Debian and its derivatives typically use the `unattended-upgrades` package to configure automatic updates. Like `yum-cron`, it is easy to install, and its configuration is placed in a variety of files within `/etc/apt/apt.conf.d/`:

```
- name: Install unattended upgrades package.
  apt: name=unattended-upgrades state=present

- name: Copy unattended-upgrades configuration files in place.
  template:
    src: "../templates/{{ item }}.j2"
    dest: "/etc/apt/apt.conf.d/{{ item }}"
    owner: root
    group: root
    mode: 0644
  with_items:
    - 10periodic
    - 50unattended-upgrades
```

The template files copied in the second task should look something like the following:

```
# File: /etc/apt/apt.conf.d/20auto-upgrades
APT::Periodic::Update-Package-Lists "1";
APT::Periodic::Unattended-Upgrade "1";
```

This file provides configuration for the `apt` script that runs as part of the unattended upgrades package, and tells apt whether to enable unattended upgrades.

```
# File: /etc/apt/apt.conf.d/50unattended-upgrades
Unattended-Upgrade::Automatic-Reboot "false";

Unattended-Upgrade::DevRelease "false";

Unattended-Upgrade::Allowed-Origins {
        "${distro_id}:${distro_codename}";
        "${distro_id}:${distro_codename}-security";
        "${distro_id}ESM:${distro_codename}";
//      "${distro_id}:${distro_codename}-updates";
//      "${distro_id}:${distro_codename}-proposed";
//      "${distro_id}:${distro_codename}-backports";
};
```

This file provides further configuration for unattended upgrades, like whether to automatically restart the server for package and kernel upgrades requiring a reboot, or what `apt` sources should be checked for updated packages.

Make sure you get notifications or check in on your servers periodically so you know when they'll need a manual reboot if you have `Automatic-Reboot` set to `false`!

Use a properly-configured firewall

If you were building a secure bank vault, you wouldn't want to have many doors and windows leading into the vault. You'd instead build reinforced concrete walls, and have one or two strong metal doors.

Similarly, you should close any port not explicitly required to remain open on all your servers—whether in a DMZ in your network or open to the entire Internet. There are dozens of different ways to manage firewalls nowadays, from `iptables` and helpful tools like `ufw` and `firewalld` that help make iptables configuration easier, to AWS security groups and other external firewall services.

Ansible includes built-in support for configuring server firewalls with `ufw` (common on newer Debian and Ubuntu distributions) and `firewalld` (common on newer Fedora, RHEL, and CentOS distributions).

Configuring a firewall with `ufw` on Debian or Ubuntu

Below is an entire firewall configuration to lock down most everything on a Debian or Ubuntu server, allowing traffic only through ports 22 (SSH), 80 (HTTP), and 123 (NTP):

```
- name: Configure open ports with ufw.
  ufw:
    rule: "{{ item.rule }}"
    port: "{{ item.port }}"
    proto: "{{ item.proto }}"
  with_items:
    - { rule: 'allow', port: 22, proto: 'tcp' }
    - { rule: 'allow', port: 80, proto: 'tcp' }
    - { rule: 'allow', port: 123, proto: 'udp' }

- name: Configure default incoming/outgoing rules with ufw.
  ufw:
    direction: "{{ item.direction }}"
    policy: "{{ item.policy }}"
    state: enabled
  with_items:
    - { direction: outgoing, policy: allow }
    - { direction: incoming, policy: deny }
```

If you run a playbook with the above rules, the log into the machine (or use the `ansible` command) and run `sudo ufw status verbose`, you should see the configuration has been updated to the following:

```
$ sudo ufw status verbose
Status: active
Logging: on (low)
Default: deny (incoming), allow (outgoing), disabled (routed)
New profiles: skip

To                         Action      From
--                         ------      ----
22/tcp                     ALLOW IN    Anywhere
80/tcp                     ALLOW IN    Anywhere
123/udp                    ALLOW IN    Anywhere
22/tcp (v6)                ALLOW IN    Anywhere (v6)
```

```
80/tcp (v6)                ALLOW IN    Anywhere (v6)
123/udp (v6)               ALLOW IN    Anywhere (v6)
```

Configuring a firewall with `firewalld` on RHEL, Fedora, or CentOS

The same firewall configuration can be done via `firewalld` for RHEL-based systems with similar ease:

```
- name: Configure open ports with firewalld.
  firewalld:
    state: "{{ item.state }}"
    port: "{{ item.port }}"
    zone: external
    immediate: yes
    permanent: yes
  with_items:
    - { state: 'enabled', port: '22/tcp' }
    - { state: 'enabled', port: '80/tcp' }
    - { state: 'enabled', port: '123/udp' }
```

The `immediate` parameter was added in Ansible 1.9, and is required to make the rules effective immediately when the `permanent` parameter is set to `yes`. If you are running an older version of Ansible, you will need to restart to see your changes, or set `permanent` to `no`.

`firewalld` doesn't have an explicit command to allow setting default inbound/outbound policies, but you can still use `iptables` commands or manage the firewall via XML files inside `/etc/firewalld`.

If you run `sudo firewall-cmd --zone=external --list-all`, you should see the open ports:

```
$ sudo firewall-cmd --zone=external --list-all
external
  interfaces:
  sources:
  services: ssh
  ports: 123/udp 80/tcp 22/tcp
  masquerade: yes
  forward-ports:
  icmp-blocks:
  rich rules:
```

Some still prefer configuring firewalls with `iptables` (which is sometimes obtuse, but is almost infinitely malleable). This approach is used in the `geerlingguy.firewall` role on Ansible Galaxy, which translates variables like `firewall_allowed_tcp_ports` and `firewall_forwarded_tcp_ports` into `iptables` rules, and provides a `firewall` service for loading firewall rules.

It doesn't really matter what method you use to control access to your server, but the *principle of least privilege* applies here, as in most security-related discussions: only allow access on ports absolutely necessary for the functionality of your server, and restrict the use of those ports to only the hosts or subnets needing access to the services listening on the ports.

When you're building up a firewall, make sure you don't accidentally lock down ports or IP addresses that will lock you out of the server entirely, otherwise you'll have to connect to the server through a local terminal connection and start over!

Make sure log files are populated and rotated

Checking server logs is one of the most effective ways to not only see what attacks have taken place on a server, but also to see trends over time and predict high-traffic periods, potential attack vectors, and potential catastrophe.

But logs are completely worthless if they aren't being populated with effective data, aren't being monitored in any way, or are *the cause* of an outage! Many root cause analyses conclude, “the server’s disk was full because log file *x* took up all the free space”.

I have my eyes on you, 218.78.214.9…

```
sshd[19731]: input_userauth_request: invalid user db2admin
sshd[19731]: Received disconnect from 218.78.214.9: 11: Bye Bye
sshd[19732]: Invalid user jenkins from 218.78.214.9
sshd[19733]: input_userauth_request: invalid user jenkins
sshd[19733]: Received disconnect from 218.78.214.9: 11: Bye Bye
sshd[19734]: Invalid user jenkins from 218.78.214.9
sshd[19735]: input_userauth_request: invalid user jenkins
sshd[19735]: Received disconnect from 218.78.214.9: 11: Bye Bye
sshd[19736]: Invalid user minecraft from 218.78.214.9
sshd[19737]: input_userauth_request: invalid user minecraft
sshd[19737]: Received disconnect from 218.78.214.9: 11: Bye Bye
```

Only you will know what logs are the most important to monitor on your servers, but some of the most common ones are database slow query logs, web server access and error logs, authorization logs, and cron logs. Use tools like the ELK stack (demonstrated in a cookbook in Chapter 8), Munin, Nagios, or even a hosted service to make sure logs are populated and monitored.

Additionally, you should always make sure log files are rotated and archived (according to your infrastructure’s needs) using a tool like `logrotate`, and you should have monitoring enabled on log file sizes so you have an early warning when a particular log file or directory grows a bit too large. There are a number of `logrotate` roles on Ansible Galaxy (e.g. Nick Hammond’s `logrotate` role[128]) that make rotation configuration easy.

Monitor logins and block suspect IP addresses

If you’ve ever set up a new server on the public internet and enabled SSH on port 22 with password-based login enabled, you know how quickly the deluge of script-based logins begins. Many honeypot servers detect hundreds or thousands of such attempts per hour.

[128]https://galaxy.ansible.com/nickhammond/logrotate/

If you allow password-based login (for SSH, for your web app, or for anything else), you need to implement some form of monitoring and rate limiting. At a most basic level, you should install a tool like Fail2Ban[129], which monitors log files and bans IP addresses when it detects too many unsuccessful login attempts in a given period of time.

Here's a set of tasks you could add to your playbook to install Fail2Ban and make sure it's started on either Debian or RHEL-based distributions:

```
- name: Install fail2ban (RedHat).
  yum: name=fail2ban state=present enablerepo=epel
  when: ansible_os_family == 'RedHat'

- name: Install fail2ban (Debian).
  apt: name=fail2ban state=present
  when: ansible_os_family == 'Debian'

- name: Ensure fail2ban is running and enabled on boot.
  service: name=fail2ban state=started enabled=yes
```

Fail2Ban configuration is managed in a series of `.conf` files inside `/etc/fail2ban`, and most configuration can be done by overriding defaults in a local override file, `/etc/fail2ban/jail.local`. See the Fail2Ban manual[130] for more information.

Use SELinux (Security-Enhanced Linux) or AppArmor

SELinux and AppArmor are two different tools which allow you to construct security sandboxes for memory and filesystem access, so, for example, one application can't easily access another application's resources. It's a little like user and group file permissions, but allowing far finer detail—with far more complexity.

You'd be forgiven if you disabled SELinux or AppArmor in the past; both require extra work to set up and configure for your particular servers, especially if you're

[129] http://www.fail2ban.org/wiki/index.php/Main_Page
[130] http://www.fail2ban.org/wiki/index.php/MANUAL_0_8

using less popular distribution packages (extremely popular packages like Apache and MySQL are extremely well supported out-of-the-box on most distributions).

However, both of these tools are excellent ways to add *defense in depth* to your infrastructure. You should already have decent configurations for firewalls, file permissions, users and groups, OS updates, etc. But if you're running a web-facing application—especially one running on a server with any other applications—it's great to have the extra protection SELinux or AppArmor provides from applications accessing things they shouldn't.

SELinux is usually installed and enabled by default on Fedora, RHEL and CentOS systems, is available and supported on most other Linux platforms, and is widely supported through Ansible modules, so we'll cover SELinux in a bit more depth.

To enable SELinux in `targeted` mode (which is the most secure mode without being almost impossible to work with), make sure the Python SELinux library is installed, then use Ansible's `selinux` module:

```
- name: Install Python SELinux library.
  yum: name=libselinux-python state=present

- Ensure SELinux is enabled in `targeted` mode.
  selinux: policy=targeted state=enforcing
```

Ansible also has a `seboolean` module that allows setting SELinux booleans. A very common setting for web servers involves setting the `httpd_can_network_connect` boolean:

```
- name: Ensure httpd can connect to the network.
  seboolean: name=httpd_can_network_connect state=yes persistent=yes
```

The Ansible `file` module also integrates well with SELinux, allowing the four security context fields for a file or directory to be set, one per parameter:

1. `selevel`
2. `serole`
3. `setype`

4. `seuser`

Building custom SELinux policies for more complex scenarios is out of the scope of this chapter, but you should be able to use tools like `setroubleshoot`, `setroubleshoot-server`, `getsebool`, and `aureport` to see what is being blocked, what booleans are available (and/or enabled currently), and even get helpful notifications when SELinux denies access to an application. Read Getting started with SELinux[131] for an excellent and concise introduction.

Next time you're tempted to disable SELinux instead of fixing the underlying problem, spend a little time investigating the correct boolean settings to configuring your system correctly for SELinux.

Summary and further reading

This chapter contains a broad overview of some Linux security best practices, and how Ansible helps you conform to them. There is a wealth of good information on the Internet to help you secure your servers, including articles and publications like the following:

- Linode Library: Linux Security Basics[132]
- My First Five Minutes on a Server[133]
- 20 Linux Server Hardening Security Tips[134]
- Unix and Linux System Administration Handbook[135]

Much of the security configuration in this chapter is encapsulated in a role on Ansible Galaxy, for use on your own servers: security role by geerlingguy[136].

[131]https://major.io/2012/01/25/getting-started-with-selinux/
[132]https://library.linode.com/security/basics
[133]http://plusbryan.com/my-first-5-minutes-on-a-server-or-essential-security-for-linux-servers
[134]http://www.cyberciti.biz/tips/linux-security.html
[135]http://www.admin.com/
[136]https://galaxy.ansible.com/geerlingguy/security/

```
 _____________________________________
/ Bad planning on your part does not  \
| constitute an emergency on my part. |
\ (Proverb)                           /
 -------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```