Hardening Your Servers Using Ansible

One of the advantages of using an orchestration and configuration tool such as Ansible is that it can be utilized to generate and deploy a complex set of configurations in a repeatable task across many hosts. In this chapter, we will look at a tool that scans your hosts using Ansible, dynamically generates a remediation playbook, and then runs it for you.

We will also look at running two different security tools that scan the WordPress installation we have used throughout the previous chapters.

This chapter covers the following topics:

* The scanning tools
* The playbook

Technical requirements

Following our excursion into the cloud, we will return to our local machine and launch an Ubuntu 22.04 virtual machine using Multipass; as we will be running a workload that requires a bit more disk space than we have been used to so far, we will be altering the specs of the virtual machine when we launch it to increase the disk space and RAM.

As we will be installing a lot of different software on the virtual machine, your Multipass virtual machine will need to be able to download packages from the internet; there will be around 3 GB of various packages and configuration files to download.

You can find a complete copy of the playbook accompanying this chapter in the repository at https://github.com/PacktPublishing/Learn-Ansible-Second-Edition/tree/main/Chapter14.

The scanning tools

Before we dive into the Playbook, let’s quickly look at the three tools we will be running, starting with the one that does the most, **OpenSCAP**.

OpenSCAP

First, we will be looking at one of Red Hat’s tools, called OpenSCAP. Before we continue, the next section will contain many abbreviations.

So, what is SCAP? The **Security Content Automation Protocol** (**SCAP**) is an open standard that encompasses several components, all of which are open standards themselves, to build a framework that allows you to automatically assess and remediate your hosts against the **National Institute of Standards and Technology** (**NIST**) *Special Publication 800-53*.

This publication is a catalog of controls applied to all U.S. federal IT systems, apart from those maintained by the **National Security Agency** (**NSA**). These controls have been effected to help implement the **Federal Information Security Management Act** (**FISMA**) of 2002 across U.S. federal departments.

SCAP is made up of the following components:

* **Asset Identification** (**AID**) is a data model used for asset identification.
* **Asset Reporting Format** (**ARF**) is a vendor-neutral and technology-agnostic data model for transporting information on assets between different reporting applications and services.
* **Common Configuration Enumeration** (**CCE**) is a standard database of recommended configurations for common software. Each recommendation has a unique identifier. At the time of writing, the database hadn’t been updated for over a decade.
* **Common Configuration Scoring System** (**CCSS**) is the continuation of CCE. It is used for generating a score for various software and hardware configurations across all types of deployments.
* **Common Platform Enumeration** (**CPE**) identifies hardware assets, operating systems, and software in an organization’s infrastructure. Once identified, this data can then be used to search other databases to threat-assess the asset.
* **Common Weakness Enumeration** (**CWE**) is a common language for dealing with and discussing the causes of weaknesses in system architecture, design, and code that may lead to vulnerabilities.
* **Common Vulnerabilities and Exposures** (**CVE**) is a database of publicly acknowledged vulnerabilities. Most system administrators and IT professionals will have encountered the CVE database at some point. Each vulnerability receives a unique ID; for example, most people will know CVE-2014-0160, also known as **Heartbleed**. The Heartbleed vulnerability was a severe security flaw in OpenSSL (a cryptographic software library) that allowed attackers to steal sensitive information, such as passwords and private keys, from the memory of affected systems by exploiting a bug in the OpenSSL’s implementation of the **transport layer security** (**TLS**)/**datagram transport layer security** (**DTLS**) heartbeat extension.
* **Common Vulnerability Scoring System** (**CVSS**) is a method that helps capture the characteristics of a vulnerability to produce a normalized numerical score, which can then be used to describe the impact of a vulnerability, for example, low, medium, high, and critical.
* **Extensible Configuration Checklist Description Format** (**XCCDF**) is an XML format for describing security checklists. It can also be used for configuration and benchmarks and provides a common language for all the parts of SCAP.
* **Open Checklist Interactive Language** (**OCIL**) is a framework for expressing questions to an end user and the procedures to process the responses in a standardized way.
* **Open Vulnerability and Assessment Language** (**OVAL**) is defined in XML and aims to standardize the transfer of security content across all of the tools and services offered by NIST, the MITRE Corporation, the **United States Computer Emergency Readiness Team** (**US-CERT**), and the United States **Department of Homeland Security** (**DHS**).
* **Trust Model for Security Automation Data** (**TMSAD**) is an XML document that aims to define a common trust model that can be applied to the data being exchanged by all components that make up SCAP.

As you can imagine, thousands of man-years have gone into producing SCAP and its components to make its foundation. Some of the projects have been around in one form or another since the mid-90s, so they are well-established and considered the de facto standard when it comes to security best practices; however, I am sure you think that it all sounds very complicated – after all, these are standards that have been defined and are being maintained by scholars, security professionals, and government departments.

This is where OpenSCAP comes in. The OpenSCAP project, maintained by Red Hat and certified by NIST for supporting the SCAP standard, allows you to apply all the best practices we have discussed using a command-line client.

Note

The automatic remediation scripts in OpenSCAP are a work in progress, and there are known issues that we will address toward the end of the chapter. Because of this, your output may differ from that covered in this chapter.

OpenSCAP, like many Red Hat projects, has support for Ansible, and the current release introduces support for automatically generating Ansible playbooks to remediate non-conformance discovered during an OpenSCAP scan.

The next two tools we will be looking at will be scanning our WordPress site, starting with WPScan.

WPScan

The second tool we will be running is called **WPScan** and we will use it to scan our WordPress site. WPScan is a command-line tool that can perform various security assessments and vulnerability tests on WordPress installations. It can detect common configuration errors, outdated themes, weak passwords, and other potential risks. WPScan is easy to install – especially as we will be using the container version and running it using Docker, which we will also be going for the third and final tool, OWASP ZAP.

OWASP ZAP

Web vulnerabilities such as SQL injection, cross-site scripting, broken authentication, and insecure deserialization can threaten our WordPress site’s security and quality. To help identify and prioritize such vulnerabilities, we can use **OWASP ZAP**. This tool, the third and final one we will cover in the chapter, generates reports, alerts, and graphs that assist us in visualizing and addressing the findings. Moreover, OWASP ZAP is user-friendly and easy to install, making it a valuable resource for enhancing our site’s security and overall quality.

The playbook

We will split the playbook into a few different roles to run the various scanning tools that will be running in the chapter – as you can see from the site.yml file, we are adding some conditions to the roles containing our tasks. The start of the file looks like all of the other playbook files we have been running:

- name: "Scan our WordPress Ansible Playbook and stack"

  hosts: ansible\_hosts

  gather\_facts: true

  become: true

  become\_method: "ansible.builtin.sudo"

  vars\_files:

    - 'group\_vars/common.yml'

CopyExplain

As mentioned, roles are where this playbook starts to differ from the previous playbooks we have been running up to this point in the book.

As you can see from the following source, we are defining tags alongside the roles themselves:

  roles:

    - { role: 'common', tags: ['openscap','scan'] }

    - { role: 'docker', tags: ['docker','scan'] }

CopyExplain

As you can see, we are using the openscap, scan, and docker tags followed by wordpress, which used the roles directly from [*Chapter 5*](https://subscription.packtpub.com/book/cloud-and-networking/9781835088913/5), *Deploying WordPress*:

    - { role: 'stack\_install', tags: ['wordpress'] }

    - { role: 'stack\_config', tags: ['wordpress'] }

    - { role: 'wordpress', tags: ['wordpress'] }

CopyExplain

Finally, we have roles that run scans and openscap:

    - { role: 'scan', tags: ['scan'] }

    - { role: 'openscap', tags: ['openscap'] }

CopyExplain

So, what does this mean? Well, later in the chapter, when it comes to running the playbook, we will only be running specific roles; for example, to run OpenSCAP, we will use the following commands:

$ ansible-playbook -i hosts site.yml --tags "openscap" --extra-vars "scap\_options\_remediation=true"

$ ansible-playbook -i hosts site.yml --tags "openscap"

CopyExplain

When running the first command, it will run just the common and openscap roles and run the remediation Ansible Playbook and bash script, both of which will be automatically generated during the initial scan – it will also download a copy of the results, an implementation guide, a copy of the playbook, and a copy of the bash scripts.

The second of the two commands will rerun the scan host and download a copy of the results again.

Once we have finished running OpenSCAP, we will then redeploy our host and run the following:

$ ansible-playbook -i hosts site.yml --tags "wordpress"

CopyExplain

This, as I am sure you will have guessed, will run the three wordpress roles. Then, with WordPress installed, we can run the following:

$ ansible-playbook -i hosts site.yml --tags "scan"

CopyExplain

This will execute the common, docker, and scan roles.

We can also run these commands to run just one of the two scanning tools that the scan role runs:

$ ansible-playbook -i hosts site.yml --tags "scan" --extra-vars "scan\_types=zap"

$ ansible-playbook -i hosts site.yml --tags "scan" --extra-vars "scan\_types=wpscan"

CopyExplain

But we are getting ahead of ourselves; let’s work our way through the preceding roles before we think about running the playbook.

The common role

This role contains a single task in roles/common/tasks/main.yml, and its only job is to set a fact containing the current date and time:

- name: "Set a fact for the date"

  ansible.builtin.set\_fact:

    the\_date: "{{ lookup('pipe', 'date +%Y-%m-%d-%H%M') }}"

CopyExplain

You might think, *“That seems a little basic.”* However, as we will be using the the\_date variable several times throughout the roles in this playbook, we only want it to be generated once as it will be used to create file and folder names that are then called later in tasks.

If we use {{ lookup('pipe', 'date +%Y-%m-%d-%H%M') }} to insert the date dynamically as part of other variables and tasks, we need to be cautious. This is because some parts of the playbook can take several minutes to finish running.

For instance, we may create a file called myfile-2024-02-16-1300.yml at one point in the playbook. However, if we dynamically set the date and time, and several tasks later, it takes five minutes for the playbook to get to that task, we could reference a file called myfile-2024-02-16-1305.yml. This would result in an error as the file does not exist. Therefore, we should only use the date and time lookup once during the playbook run.

The Docker role

This role contains all of the tasks and variables needed to install and configure Docker on our target host, much like the roles discussed in [*Chapter 4*](https://subscription.packtpub.com/book/cloud-and-networking/9781835088913/4), *Deploying a LAMP Stack*, and [*Chapter 5*](https://subscription.packtpub.com/book/cloud-and-networking/9781835088913/5), *Deploying WordPress*; this role uses the ansible.builtin.apt, ansible.builtin.apt\_key, and ansible.builtin.apt\_repository modules to do the following:

1. Download and install the prerequisites required for Docker to run.
2. Add the **GNU Privacy Guard** (**GPG**) key for the official Docker **advanced packaging tool** (**APT**) repository.
3. Configure the official Docker APT repository.
4. Install Docker itself along with the Docker command-line tool.
5. Ensure that Docker is running and set to start on boot.

To review the full list of tasks and variables for this role, see the following:

* https://github.com/PacktPublishing/Learn-Ansible-Second-Edition/blob/main/Chapter14/roles/docker/defaults/main.yml
* https://github.com/PacktPublishing/Learn-Ansible-Second-Edition/blob/main/Chapter14/roles/docker/tasks/main.yml

Next, we have the roles that install WordPress.

The WordPress roles

As you have already seen from the site.yml file at the start of the Playbook section of this chapter, here, we are just reusing the roles that we discussed at length in [*Chapter 5*](https://subscription.packtpub.com/book/cloud-and-networking/9781835088913/5), *Deploying WordPress*. If you want to review these, you can see them at https://github.com/PacktPublishing/Learn-Ansible-Second-Edition/tree/main/Chapter05/roles.

The scan role

As already mentioned, we will be using Docker to run WPScan and OWASP ZAP; this allows us to reuse the same tasks. Let’s look at roles/scan/tasks/main.yml; first, we need to pull the Docker image or images:

- name: "Pull the Docker image for the scanning tool"

  community.docker.docker\_image:

    name: "{{ item.image }}"

    source: "{{ item.source }}"

  loop: "{{ scan }}"

  when: "item.name in scan\_types"

  loop\_control:

    label: "{{ item.name }}"

CopyExplain

We are switching it up slightly in that we are using loop rather than with\_items; this gives more control over what happens when looping through. In this task, we are using label to show which of the scanning tools is currently processing.

You may also notice that we have a when condition; this allows us to run both of the scans or just one of the two by passing in the name of the scan in the scan\_types variable. When we look at the variables in a moment, you will see that by default, we are passing in the names of both scanning tools.

This pattern of loop, loop\_control, and when will be repeated throughout all the tasks in this role. We have a task that will create a folder on the virtual machine; we will be mounting this folder into the container at runtime so that we can keep a copy of the scan output:

- name: "Create the folder which we will mount inside the container"

  ansible.builtin.file:

    path: "{{ item.log.remote\_folder }}"

    state: "directory"

    mode: "0777"

  loop: "{{ scan }}"

  when: "item.name in scan\_types"

  loop\_control:

    label: "{{ item.name }}"

CopyExplain

Now, with the container image and folder created, we can run the scan:

- name: "Run the scan"

  community.docker.docker\_container:

    detach: "{{ item.detach }}"

    auto\_remove: "{{ item.auto\_remove }}"

    name: "{{ item.name }}"

    volumes: "{{ item.log.remote\_folder }}:{{ item.container\_folder }}"

    image: "{{ item.image }}"

    command: "{{ item.command }}"

  register: docker\_scan

  ignore\_errors: true

  no\_log: true

  loop: "{{ scan }}"

  when: "item.name in scan\_types"

  loop\_control:

    label: "{{ item.name }}"

CopyExplain

As you can see, everything is being passed to the container as variables; this is how we can run two very different tools with a single common task, and more so later when we look at the variables.

You will have also noted that we are adding a few options to the end of this task; these are as follows:

* register: Here, we are just registering the output of the task – nothing special here
* ignore\_errors: This tells Ansible to continue running should it detect an error; in our case, the containers we are running will purposely trigger an error code as they have been designed to halt and not proceed with any further tasks until the scan does not fail
* no\_log: This suppresses the output – as we save the output when running the scan, we do not need the output printed to the terminal when we run the task

As we are registering an output, the next task is a debug line. This follows the same pattern as debug tasks in other chapters, so we will be moving to the task that downloads a copy of the reports:

- name: "Download the report"

  ansible.builtin.fetch:

    src: "{{ item.log.remote\_folder }}{{ item.log.file }}"

    dest: "{{ item.log.local\_folder }}"

    flat: true

    mode: "0644"

  loop: "{{ scan }}"

  when: "item.name in scan\_types"

  loop\_control:

    label: "{{ item.name }}"

CopyExplain

This uses the ansible.builtin.fetch module setting the flat option to true. This option copies the file rather than the full directory path. The final task removes the container, meaning that when we next run a scan, it will start from scratch and spawn a new container rather than reusing the one we have just finished using:

- name: "Remove the scan container"

  community.docker.docker\_container:

    name: "{{ item.name }}"

    state: "absent"

  loop: "{{ scan }}"

  when: "item.name in scan\_types"

  loop\_control:

    label: "{{ item.name }}"

CopyExplain

Now that we know what the tasks look like, let us look at the variables, which can be found in roles/scan/defaults/main.yml. The first variable sets the scan we want to run, and as already mentioned, this gives the name of the two scans:

scan\_types:

  - "{{ common\_scan\_settings.dict.wpscan }}"

  - "{{ common\_scan\_settings.dict.zap }}"

CopyExplain

Next up in roles/scan/defaults/main.yml, we have a block of variables that could be commonly used across both scanning tools:

common\_scan\_settings:

  detach: false

  auto\_remove: false

  source: "pull"

  local\_folder: "output/"

  report\_name: "{{ the\_date }}-results-"

  dict:

    wpscan: "wpscan"

    zap: "zap"

CopyExplain

Finally, we have the primary scan variable, which is the one we have been looping over; it starts with WPScan:

scan:

  - name: "{{ common\_scan\_settings.dict.wpscan }}"

    image: "wpscanteam/wpscan:latest"

    source: "{{ common\_scan\_settings.source }}"

    detach: "{{ common\_scan\_settings.detach }}"

    auto\_remove: "{{ common\_scan\_settings.auto\_remove }}"

    container\_folder: "/tmp/{{ common\_scan\_settings.dict.wpscan }}/"

    command: "--url http://{{ ansible\_host }} --enumerate u --plugins-detection mixed --format cli-no-color --output /tmp/{{ common\_scan\_settings.dict.wpscan }}/{{ common\_scan\_settings.report\_name }}{{ common\_scan\_settings.dict.wpscan }}.txt"

    log:

      remote\_folder: "/tmp/{{ common\_scan\_settings.dict.wpscan }}/"

      local\_folder: "{{ common\_scan\_settings.local\_folder }}"

      file: "{{ common\_scan\_settings.report\_name }}{{ common\_scan\_settings.dict.wpscan }}.txt"

CopyExplain

The block that follows is the one for OSWAP ZAP:

  - name: "{{ common\_scan\_settings.dict.zap}}"

    image: "ghcr.io/zaproxy/zaproxy:stable"

    source: "{{ common\_scan\_settings.source }}"

    detach: "{{ common\_scan\_settings.detach }}"

    auto\_remove: "{{ common\_scan\_settings.auto\_remove }}"

    container\_folder: "/zap/wrk/"

    command: "zap-baseline.py -t http://{{ ansible\_host }} -g gen.conf -r {{ common\_scan\_settings.report\_name }}{{ common\_scan\_settings.dict.zap }}.html"

    log:

      remote\_folder: "/tmp/{{ common\_scan\_settings.dict.zap }}/"

      local\_folder: "{{ common\_scan\_settings.local\_folder }}"

      file: "{{ common\_scan\_settings.report\_name }}{{ common\_scan\_settings.dict.zap }}.html"

CopyExplain

As you can see, we pass in the different container images and commands to run the scan while using the same variables. Because of this, we could keep the tasks used in the role completely neutral, meaning that we didn’t have to consider anything custom to the tool we were running.

That concludes the scan role, leaving us with, as I am sure you will have already guessed from how long the tool explanation was at the start of the chapter, the most complex role in the playbook: OpenSCAP.

The OpenSCAP role

When writing a playbook, it is essential to know how the tool you are automating works; given that OpenSCAP is a little complex, let’s review the steps needed to manually run a scan and remediate the problems it finds using an automatically generated Ansible playbook and a shell script.

Note

While the commands to run OpenSCAP follow, you do not need to follow along; these are provided to illustrate the process we need to follow in our Playbook role.

First, we need to download and install OpenSCAP itself, along with a few tools we will also need:

$ sudo apt-get install unzip curl libopenscap8

CopyExplain

Next up, we need to download the actual content – these definitions cover several different operating systems and various levels of compliance. The GitHub repository for this content can be found at https://github.com/ComplianceAsCode/content, and at the time of writing, the current release is 0.1.71.

Get the release URL for the zip file, which contains the files we need from the releases page, then download and unzip on the host:

$ wget https://github.com/ComplianceAsCode/content/releases/download/v0.1.71/scap-security-guide-0.1.71.zip

unzip scap-security-guide-0.1.71.zip

CopyExplain

Now that we have OpenSCAP and the definition files installed, we can get some information on what is available for our Ubuntu 22.04 operating system:

$ sudo oscap info --fetch-remote-resources scap-security-guide-0.1.71/ssg-ubuntu2204-ds.xml

CopyExplain

This will give us the name of the profile we want to use; in our case, it is xccdf\_org.ssgproject.content\_profile\_cis\_level1\_server. Once we have this, we can run the scan itself:

$ oscap xccdf eval --profile xccdf\_org.ssgproject.content\_profile\_cis\_level1\_server  --results-arf result.xml --report report.html scap-security-guide-0.1.71/ssg-ubuntu2204-ds.xml

CopyExplain

This will generate two output files: an HTML copy of a report containing everything that needs fixing in a nicely digestible format we can read, and a second XML file containing the same information in a format OpenSCAP can read.

We can then take the XML file and generate a more detailed guide on how we could resolve the issues found by running the following:

$ sudo oscap xccdf generate guide  --profile xccdf\_org.ssgproject.content\_profile\_cis\_level1\_server scap-security-guide-0.1.71/ssg-ubuntu2204-ds.xml  > guide.html

CopyExplain

However, as this book is about Ansible, it would be better to have a Playbook to fix as many of the issues as possible, and running the following command will give us just that:

$ sudo oscap xccdf generate fix --fetch-remote-resources --fix-type ansible --result-id "" result.xml > playbook.yml

CopyExplain

Finally, not everything can be resolved using the Playbook method, so having a bash script to fix any issues that can’t be resolved by running the playbook is also a great idea as it will mean less manual work for us to do:

$ sudo oscap xccdf generate fix --fetch-remote-resources --fix-type bash --result-id "" result.xml > bash.sh

CopyExplain

Now we have the Playbook and bash script; we need to run them, copy the playbook to our local machine, and run it using the following:

$ ansible-playbook -i hosts --become -become-method=sudo output/ansiblevm-playbook.yml

CopyExplain

Then we go back to the virtual machine, and run the bash script using the following:

$ sudo bash bash.sh

CopyExplain

You will have seen a lot of output, but if everything goes as planned when you rerun the scan, you should see a lot of issues being reported.

Note

The code in the repo contains the variables and tasks for a feature we will not cover here, as the content we are downloading from GitHub can take up a lot of space on your drive. These tasks are included to remove any unneeded files.

So, now that we have an idea of the steps we need to automate, let’s dive straight in.

First, let’s look at the variables, which can be found in roles/openscap/default/main.yml, and that we will be using within our tasks.

Start with the option that, if set to true, will execute the remediation Playbook and Bash script:

scap\_options\_remediation: false

CopyExplain

Next, we have the packages needed to run OpenSCAP and OpenSCAP itself:

scap\_packages:

  - "unzip"

  - "curl"

  - "libopenscap8"

CopyExplain

Then we have information to download the content from GitHub; note that we are passing the API URL and not the direct download link (more on why later in the chapter):

openscap\_download:

  openscap\_github\_release\_api\_url: "https://api.github.com/repos/ComplianceAsCode/content/releases/latest"

  dest: "/tmp/scap-security-guide"

CopyExplain

Now we have a long list of filenames and details on the profile we need to use:

openscap\_scan:

  ssg\_file\_name: "{{openscap\_download.dest}}/ssg-{{ ansible\_facts.distribution | lower }}{{ ansible\_facts.distribution\_version | replace('.','') }}-ds.xml"

  profile\_search: "cis\_level1\_server"

  output\_dir: "/tmp/"

  output\_file\_xml: "{{ inventory\_hostname }}-result.xml"

  output\_file\_html: "{{ inventory\_hostname }}-report.html"

  output\_file\_guide: "{{ inventory\_hostname }}-guide.html"

  output\_file\_playbook: "{{ inventory\_hostname }}-playbook.yml"

  output\_file\_bash: "{{ inventory\_hostname }}-bash.sh"

  local\_output\_dir: "output/{{ the\_date }}-openscap-results"

CopyExplain

Notice that we are trying not to hardcode any values; for example, when referring to the operating system, we use {{ ansible\_facts.distribution | lower }}{{ ansible\_facts.distribution\_version | replace('.','') }}, which, in our case, gives us ubuntu2204. This means that if OpenSCAP supports it, we can run our Playbook on other Ubuntu distributions without making any changes.

The tasks that use these variables can be found in roles/openscap/tasks/main.yml; we begin with two tasks that install OpenSCAP, the first of which makes sure that the APT cache and our operating system are both up to date:

- name: "Update apt cache and upgrade packages"

  ansible.builtin.apt:

    name: "\*"

    state: "latest"

    update\_cache: "yes"

CopyExplain

The tasks immediately after installing OpenSCAP itself and the other packages we need:

- name: "Install common packages"

  ansible.builtin.apt:

    state: "present"

    pkg: "{{ scap\_packages }}"

CopyExplain

Now, we create the directory where we will be storing the OpenSCAP content we will be downloading from GitHub:

- name: "Create the directory to store the scap security guide content"

  ansible.builtin.file:

    path: "{{ openscap\_download.dest }}"

    state: "directory"

    mode: "0755"

CopyExplain

With our destination folder in place, we can now download the content and unarchive it:

- name: "Download the latest scap security guide content"

  ansible.builtin.unarchive:

    src: "{{ lookup('url', '{{ openscap\_download.openscap\_github\_release\_api\_url }}', split\_lines=false) | from\_json | json\_query('assets[?content\_type==`application/zip`].browser\_download\_url') | last }}"

    dest: "{{ openscap\_download.dest }}"

    creates: "{{ openscap\_download.dest }}/README.md"

    list\_files: true

    remote\_src: true

  register: scap\_download\_result

CopyExplain

On the face of it, while it looks a little complicated, there is quite a bit going on; let’s break down how we are getting the value to populate into the src key.

We use Ansible’s lookup plugin to fetch and process data from the GitHub API, giving us the latest release information for the OpenSCAP Content GitHub repository:

* {{ lookup('url', '{{ openscap\_download.openscap\_github\_release\_api\_url }}', split\_lines=false) }}: The lookup plugin is being used here with the url lookup type, which fetches data from the given URL that is specified by the openscap\_download.openscap\_github\_release\_api\_url variable, which points to the API endpoint for the latest release of a GitHub repository (https://api.github.com/repos/ComplianceAsCode/content/releases/latest). The split\_lines=false parameter ensures that the fetched content is not split into lines, preserving its JSON structure.
* | from\_json: This part of the code takes the output from the lookup plugin, which is expected to be a JSON string, and converts it into an Ansible data structure (such as a dictionary or a list) that can be further processed.
* | json\_query('assets[?content\_type==`application/zip`].browser\_download\_url'): This uses the json\_query filter with a JMESPath expression to query the converted JSON data. The 'assets[?content\_type==`application/zip`].browser\_download\_url' query looks for items in the assets array where content\_type is application/zip, and then extracts browser\_download\_url. This URL is typically used to directly download the asset from a browser.
* | last: Finally, the last filter is used to get the last URL from the list of URLs returned by the json\_query filter. We are doing this as there might be multiple assets with the application/zip content type, but we are only interested in the most recent or last one listed.

This means that we do not have to hardcode the version number of the latest release into our Playbook, which is helpful as the OpenSCAP content repo is updated at least once every few weeks.

The other options we are passing to the ansible.builtin.unarchive module are as follows:

* dest: The destination directory on the target machine where the archive will be extracted is specified
* creates: This parameter is used as a conditional check to prevent re-downloading and extracting the archive if a particular file exists
* list\_files: When set to true, this option lists all the files in the archive file; we will use this list to copy the files to our destination folder
* remote\_src: Setting this to true indicates that the source archive is located on a remote server, not on the control machine running Ansible; this is needed to download content directly from a URL

The following two tasks move the files to the root of openscap\_download.dest as they would have been unarchived to a folder containing the version number – which we don’t want to use, as it could change between runs:

- name: "Move scap security guide content to the correct location"

  ansible.builtin.shell: "mv {{ openscap\_download.dest }}/{{ scap\_download\_result.files[0] }}/\* {{ openscap\_download.dest }}"

  when: scap\_download\_result.changed

- name: "Remove the downloaded scap security guide content"

  ansible.builtin.file:

    path: "{{ openscap\_download.dest }}/{{ scap\_download\_result.files[0] }}"

    state: "absent"

  when: scap\_download\_result.changed

CopyExplain

Note that we are only running these tasks when the task that downloads the files has changed.

The final bit of information we need before we can run the OpenSCAP scan is which profile to use. To get this, we need to run the command to print information on the profiles available for our operating system:

- name: "Get information of the SCAP profiles available for the target system"

  ansible.builtin.command: "oscap info –profiles –fetch-remote-resources {{ openscap\_scan.ssg\_file\_name }}"

  register: scap\_info

CopyExplain

Now that we have the information on the available profiles registered as scap\_info, we can filter this list based on the contents of openscap\_scan.profile\_search and set a fact:

- name: "Extract profile name based on our selection criteria"

  ansible.builtin.set\_fact:

    profile\_name: "{{ scap\_info.stdout\_lines | select('search', openscap\_scan.profile\_search) | map('regex\_replace', '^(.\*?):.\*$', '\\1') | first }}"

CopyExplain

With the fact set, we can run the scan itself:

- name: "Run OpenSCAP scan"

  ansible.builtin.command: "oscap xccdf eval --profile {{ profile\_name }} --results-arf {{ openscap\_scan.output\_dir }}{{ openscap\_scan.output\_file\_xml }} --report {{ openscap\_scan.output\_dir }}{{ openscap\_scan.output\_file\_html }} {{ openscap\_scan.ssg\_file\_name }}"

  ignore\_errors: true

  no\_log: true

  register: scap\_scan

CopyExplain

As you can see, we are suppressing the output by using no\_log: true; this is because we don’t really need to see the output at this stage and can ignore errors, like in the previous role where we ran WPScan and OSWAP ZAP.

Now that we have the output of the scan, we need to create a folder on our Ansible host to copy the output files to the following:

- name: "Ensure the local output directory exists"

  ansible.builtin.file:

    path: "{{ openscap\_scan.local\_output\_dir }}"

    state: directory

    mode: "0755"

  delegate\_to: "localhost"

  become: false

CopyExplain

As you can see, we are using delegate\_to to ensure that Ansible runs the task on localhost, and we are telling it not to become a privileged user.

Now we can fetch the output.xml and report.html files:

- name: "Copy the SCAP report and results file to local machine"

  ansible.builtin.fetch:

    src: "{{ item }}"

    dest: "{{ openscap\_scan.local\_output\_dir }}/"

    flat: true

    mode: "0644"

  with\_items:

    - "{{ openscap\_scan.output\_dir }}{{ openscap\_scan.output\_file\_xml }}"

    - "{{ openscap\_scan.output\_dir }}{{ openscap\_scan.output\_file\_html }}"

CopyExplain

Next, we need to generate the guide and remediation files:

- name: "generate SCAP guide"

  ansible.builtin.command: "oscap xccdf generate guide --profile {{ profile\_name }} {{ openscap\_scan.ssg\_file\_name }}"

  ignore\_errors: true

  register: scap\_guide

CopyExplain

You may have noticed we are not saving a file here; we are just registering the output. That is because all of the content for the guide is output to the screen when the command is run, so rather than direct the output to a file on the virtual machine and copy it, we can capture the output and then create a file on our local machine that contains this content, essentially a fancy *copy* + *paste* from the remote host to our local one:

- name: "Copy SCAP guide to local machine"

  ansible.builtin.copy:

    content: "{{ scap\_guide.stdout }}"

    dest: "{{ openscap\_scan.local\_output\_dir }}/{{ openscap\_scan.output\_file\_guide }}"

    mode: "0644"

  when: scap\_guide is defined

  delegate\_to: "localhost"

  become: false

CopyExplain

This is then repeated for the remediation Ansible Playbook:

- name: "Generate SCAP fix playbook"

  ansible.builtin.command: "oscap xccdf generate fix --fetch-remote-resources --fix-type ansible --result-id '' {{ openscap\_scan.output\_dir }}{{ openscap\_scan.output\_file\_xml }}"

  ignore\_errors: true

  register: scap\_playbook

- name: "Copy SCAP playbook to local machine"

  ansible.builtin.copy:

    content: "{{ scap\_playbook.stdout }}"

    dest: "{{ openscap\_scan.local\_output\_dir }}/{{ openscap\_scan.output\_file\_playbook }}"

    mode: "0644"

  when: scap\_playbook is defined

  delegate\_to: "localhost"

  become: false

CopyExplain

Then again, for the remediation Bash script:

- name: "Generate SCAP fix bash script"

  ansible.builtin.command: "oscap xccdf generate fix --fetch-remote-resources --fix-type bash --result-id '' {{ openscap\_scan.output\_dir }}{{ openscap\_scan.output\_file\_xml }}"

  ignore\_errors: true

  register: scap\_bash\_script

- name: "Copy SCAP bash script to local machine"

  ansible.builtin.copy:

    content: "{{ scap\_bash\_script.stdout }}"

    dest: "{{ openscap\_scan.local\_output\_dir }}/{{ openscap\_scan.output\_file\_bash }}"

    mode: "0644"

  when: scap\_bash\_script is defined

  delegate\_to: "localhost"

  become: false

CopyExplain

The remaining tasks in the role deal with the remediation work, starting with the playbook:

- name: "Run the remediation playbook"

  ansible.builtin.command: "ansible-playbook -i {{ inventory\_file }} --become --become-method sudo {{ openscap\_scan.local\_output\_dir }}/{{ openscap\_scan.output\_file\_playbook }}"

  when: scap\_options\_remediation

  delegate\_to: "localhost"

  become: false

  register: remediation\_playbook

CopyExplain

Then, as we never kept a copy of the bash script on the target virtual machine, we need to copy it back there:

- name: "Copy the remediation bash script to the target machine"

  ansible.builtin.copy:

    src: "{{ openscap\_scan.local\_output\_dir }}/{{ openscap\_scan.output\_file\_bash }}"

    dest: "{{ openscap\_scan.output\_dir }}"

    mode: "0755"

  when: scap\_options\_remediation

CopyExplain

Once copied, we can run the script:

- name: "Run the remediation bash script"

  ansible.builtin.command: "bash {{ openscap\_scan.output\_dir }}{{ openscap\_scan.output\_file\_bash }}"

  when: scap\_options\_remediation

  register: remediation\_bash\_script

CopyExplain

With that task, the role is complete, and we now have all the pieces in place to run our playbook.

Running the playbook

In [*Chapter 1*](https://subscription.packtpub.com/book/cloud-and-networking/9781835088913/1), *Installing and Running Ansible*, we covered the installation and usage of Multipass; since then, we have been launching our local virtual machines using the same commands. In this chapter, as we need a little more disk space and RAM, we are going to be adding a few extra options when we launch the virtual machine:

$ multipass launch -n ansiblevm --cloud-init cloud-init.yaml --disk 10G --memory 4G

CopyExplain

Once the virtual machine has launched, you can get the IP address of the host by running the following:

$ multipass info ansiblevm

CopyExplain

Once you have the IP address, create a copy of hosts.example, calling its hosts and updating the IP address as we have done in previous chapters. Once your hosts inventory file is in place, we can start to run the playbook, starting with the OpenSCAP scan:

$ ansible-playbook -i hosts site.yml --tags "openscap" --extra-vars "scap\_options\_remediation=true"

CopyExplain

As you can see, we are running using the openscap tag and setting the scap\_options\_remediation variable to true; if you recall, the default for this variable is false, meaning the remediation tasks will be executed during this playbook run.

Once completed, you will find several files in the output folder on your local machine; if you are not following along, then you can find a copy of the output at https://github.com/PacktPublishing/Learn-Ansible-Second-Edition/tree/main/Chapter14/examples/01-scap\_options\_remediation\_true.

As you can see from the following screen, on the initial run, we had 98 failed results:

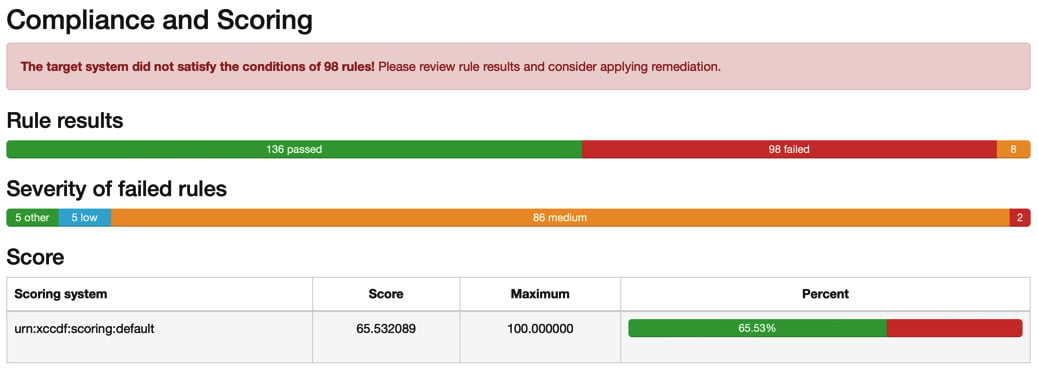


Figure 14.1 – The initial results

As we ran the remediation tasks as part of the playbook run, we know that the score should now be improved, so let’s rerun the playbook – this time skipping the remediation tasks altogether:

$ ansible-playbook -i hosts site.yml --tags "openscap"

CopyExplain

Once completed, you should have another folder of results; again, you can view the results at https://github.com/PacktPublishing/Learn-Ansible-Second-Edition/tree/main/Chapter14/examples/02-scap\_options\_remediation\_false:

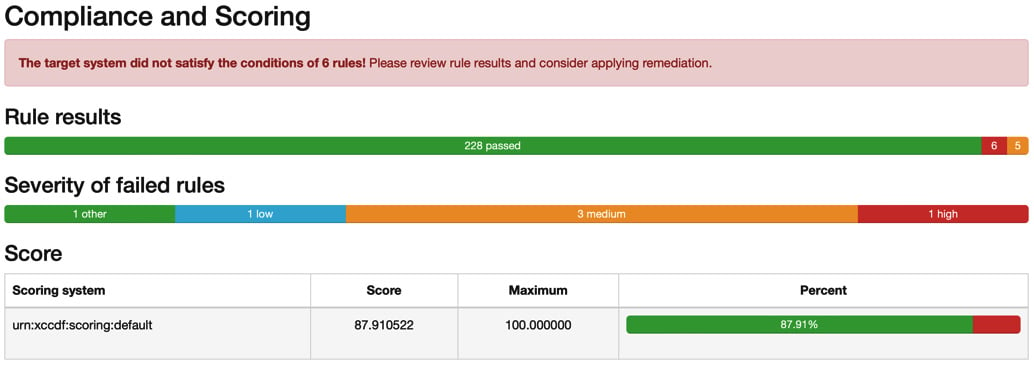


Figure 14.2 – The updated results

As you can see, this has dramatically improved the score, and we only have six failures this time.

Next, we need to install WordPress; let’s start afresh with that. To make a fresh start, run the following commands to terminate the virtual machine and replace it with a new one:

$ multipass stop ansiblevm

$ multipass delete --purge ansiblevm

$ multipass launch -n ansiblevm --cloud-init cloud-init.yaml --disk 10G --memory 4G

$ multipass info ansiblevm

CopyExplain

Update the hosts file with the new IP address and then run the following command to install WordPress:

$ ansible-playbook -i hosts site.yml --tags "wordpress"

CopyExplain

With WordPress installed, you can run the WPScan and OSWAP ZAP scans with the following command:

$ ansible-playbook -i hosts site.yml --tags "scan"

CopyExplain

Once completed, you will have the scan results in the output folder; you can find examples of the results at https://github.com/PacktPublishing/Learn-Ansible-Second-Edition/tree/main/Chapter14/examples. The folder also contains the entire output from each of the playbook runs so far up to this point in the chapter.

Also, as mentioned at the start of the chapter, you can run each of the scans independently of each other using the following commands:

$ ansible-playbook -i hosts site.yml --tags "scan" --extra-vars "scan\_types=zap"

$ ansible-playbook -i hosts site.yml --tags "scan" --extra-vars "scan\_types=wpscan"

CopyExplain

Once you have finished running the playbooks, you can remove the virtual machine by running the following:

$ multipass stop ansiblevm

$ multipass delete --purge ansiblevm

CopyExplain

With the virtual machine cleaned up, that concludes our look at using Ansible to scan and harden our server.

Before we move on to the next chapter, I recommend you look at the remediation playbook, which was generated when we first ran OpenSCAP.

It can be found at https://github.com/PacktPublishing/Learn-Ansible-Second-Edition/blob/main/Chapter14/examples/01-scap\_options\_remediation\_true/ansiblevm-playbook.yml, and as you can see, it contains over 4,600 lines of code!

Summary

In this chapter, we generated a playbook to remediate any CIS level-1 non-compliance errors found during a scan. As well as being cool, it is also convenient if you imagine you are running a few dozen servers that all need to be compliant and that all need an entire audit history.

You now have the foundations of a playbook that you can use to target those hosts daily, audit them, and store the results away from the host itself. Also, if you need to, depending on your configuration, you have a way of automatically resolving any non-conformance found during the scan.

We also ran scans against our WordPress installation and again stored the results away from the host itself – while the WPScan and OSWAP ZAP scans didn’t include any remediation, you could quickly review the results and update your WordPress deployment script to remediate the issues raised at deployment time.

So far, we have been running our Ansible Playbooks from our local machine; in the next chapter, it is time to move from running our Ansible code from our local machines into the cloud and look at how we can use Azure DevOps Pipelines and GitHub Actions to execute our playbooks.