Madhu Akula, Akash Mahajan

Security Automation with Ansible 2

Leverage Ansible 2 to automate complex security tasks like application security, network security, and malware analysis

Packt

5
Automating Web Application Security Testing Using OWASP ZAP

The OWASP **Zed Attack Proxy** (commonly known as **ZAP**) is one of the most popular web application security testing tools. It has many features that allow it to be used for manual security testing; it also fits nicely into **continuous integration/continuous delivery** (**CI/CD**) environments after some tweaking and configuration.

More details about the project can be found at `https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project`.

Open Web Application Security Project (**OWASP**) is a worldwide not-for-profit charitable organization focused on improving the security of software. Read more about OWASP projects and resources at `https://www.owasp.org`.

OWASP ZAP includes many different tools and features in one package. For a pentester tasked with doing the security testing of web applications, the following features are invaluable:

Feature	**Use case**
Intercepting proxy	This allows us to intercept requests and responses in the browser
Active scanner	Automatically run web security scans against targets
Passive scanner	Glean information about security issues from pages that get downloaded using spider tools and so on
Spiders	Before ZAP can attack an application, it creates a site map of the application by crawling all the possible web pages on it
REST API	Allows ZAP to be run in headless mode and to be controlled for running automated scanner, spider, and get the results

As you may have guessed, in this chapter, for security automation we will invoke ZAP in headless mode and use the API interfaces provided by it to do the scanning and security testing.

ZAP is a Java-based software. The typical way of using it will involve the following:

- **Java Runtime Environment** (**JRE**) 7 or more recent installed in the operating system of your choice (macOS, Windows, Linux)
- Install ZAP using package managers, installers from the official downloads page

You can find the latest updated stable links here: `https://github.com/zaproxy/zaproxy/wiki/Downloads`.

While we can build a playbook to do exactly that, the developer world is moving toward concepts of CI/CD and continuous security. An approach in which we can bootstrap a stable version of ZAP as and when required would be ideal.

The best way to achieve that is to use OWASP ZAP as a container. In fact, this is the kind of setup Mozilla uses ZAP in a CI/CD pipeline to verify the baseline security controls at every release.

If you are wondering about the connection between Mozilla and OWASP ZAP, Simon Bennetts leads the OWASP ZAP project and works at Mozilla. Read his blog post about ZAP baseline scans at `https://blog.mozilla.org/security/2017/01/25/setting-a-baseline-for-web-security-controls/`.

Installing OWASP ZAP

We are going to use OWASP ZAP as a container in this chapter, which requires container runtime in the host operating system. The team behind OWASP ZAP releases ZAP Docker images on a weekly basis via Docker Hub. The approach of pulling Docker images based on tags is popular in modern DevOps environments and it makes sense that we talk about automation with respect to that.

Official ZAP is now available with stable and weekly releases via the Docker container at Docker Hub: `https://github.com/zaproxy/zaproxy/wiki/Docker`.

Installing Docker runtime

Docker is an open platform for developers and system administrators to build, ship, and run distributed applications whether on laptops, data center VMs, or the cloud. To learn more about Docker, refer to `https://www.docker.com/what-docker`.

The following playbook will install Docker Community Edition software in Ubuntu 16.04:

```
- name: installing docker on ubuntu
  hosts: zap
  remote_user: "{{ remote_user_name }}"
  gather_facts: no
  become: yes
  vars:
    remote_user_name: ubuntu
    apt_repo_data: "deb [arch=amd64]
https://download.docker.com/linux/ubuntu xenial stable"
    apt_gpg_key: https://download.docker.com/linux/ubuntu/gpg

  tasks:
    - name: adding docker gpg key
      apt_key:
```

```
    url: "{{ apt_gpg_key }}"
    state: present
- name: add docker repository
  apt_repository:
    repo: "{{ apt_repo_data }}"
    state: present
- name: installing docker-ce
  apt:
    name: docker-ce
    state: present
    update_cache: yes
- name: install python-pip
  apt:
    name: python-pip
    state: present
- name: install docker-py
  pip:
    name: "{{ item }}"
    state: present

  with_items:
    - docker-py
```

Docker requires a 64-bit version OS and a Linux kernel version equal to or greater than 3.10. Docker runtime is available for Windows and macOS as well. For the purposes of this chapter, the containers we will use are Linux-based. So the runtime can be in Windows, but the container running in that will be a Linux-based one. These are the standard OWASP ZAP containers available for use.

OWASP ZAP Docker container setup

The two new modules to deal with Docker containers that we will be using here are `docker_image` and `docker_container`.

These modules require you to be using a 2.1 and higher version of Ansible. Right now would be a good time to check your version of Ansible using the —`version` flag.

If you need to get the latest stable version using `pip`, run the following command:

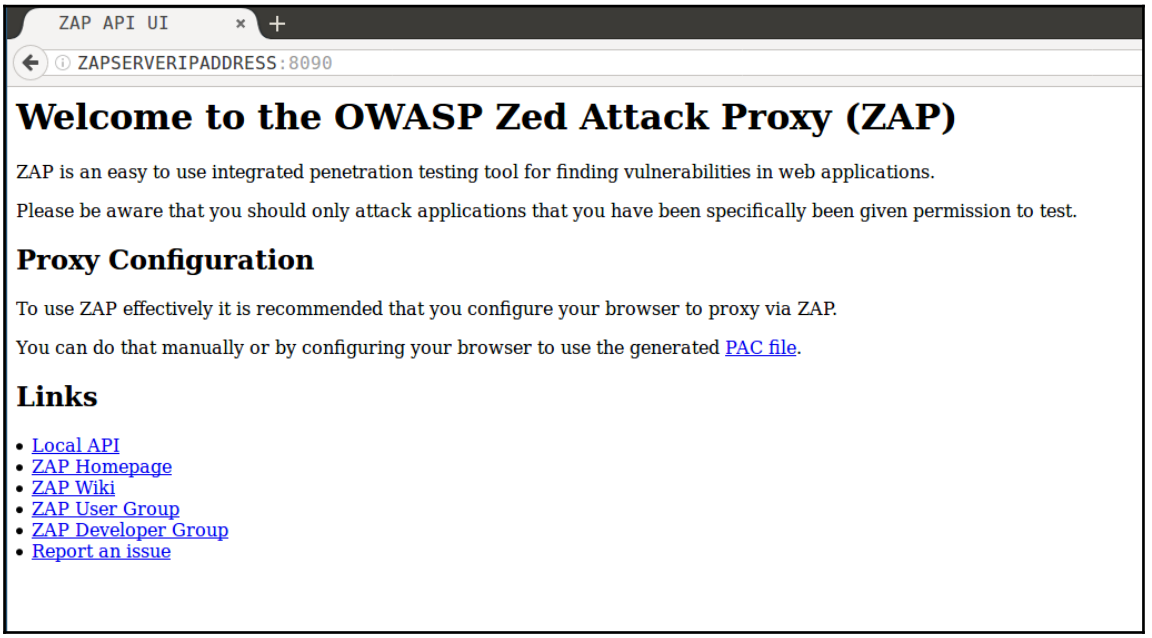
```
pip install ansible --upgrade
```

The following playbook will take some time to complete as it has to download about 1 GB of data from the internet:

```
- name: setting up owasp zap container
  hosts: zap
  remote_user: "{{ remote_user_name }}"
  gather_facts: no
  become: yes
  vars:
    remote_user_name: ubuntu
    owasp_zap_image_name: owasp/zap2docker-weekly

  tasks:
    - name: pulling {{ owasp_zap_image_name }} container
      docker_image:
        name: "{{ owasp_zap_image_name }}"

    - name: running owasp zap container
      docker_container:
        name: owasp-zap
        image: "{{ owasp_zap_image_name }}"
        interactive: yes
        state: started
        user: zap
        command: zap.sh -daemon -host 0.0.0.0 -port 8090 -config
api.disablekey=true -config api.addrs.addr.name=.* -config
api.addrs.addr.regex=true
        ports:
          - "8090:8090"
</span>
```

In the following configuration, we are saying `api.disablekey=true`, which means we are not using any API key. This can be overwritten by giving the specific API key. `api.addrs.addr.name=.*` and `api.addrs.addr.regex=true` will allow all IP addresses to connect to the ZAP API. More information about ZAP API key settings can be found at `https://github.com/zaproxy/zaproxy/wiki/FAQapikey`.

You can access the ZAP API interface by navigating to `http://ZAPSERVERIPADDRESS:8090`:



OWASP ZAP API Web UI

A specialized tool for working with Containers - Ansible Container

Currently, we are using Docker modules to perform container operations. A new tool, `ansible-container`, provides an Ansible-centric workflow for building, running, testing, and deploying containers.

This allows us to build, push, and run containers using existing playbooks. Dockerfiles are like writing shell scripts, therefore, `ansible-container` will allow us to codify those Dockerfiles and build them using existing playbooks rather writing complex scripts.

The `ansible-container` supports various orchestration tools, such as Kubernetes and OpenShift. It can also be used to push the build images to private registries such as Google Container Registry and Docker Hub.

Read more about `ansible-container` at `https://docs.ansible.com/ansible-container`.

Configuring ZAP Baseline scan

The ZAP Baseline scan is a script that is available in the ZAP Docker images.

More details about OWASP ZAP Baseline scan can be found at `https://github.com/zaproxy/zaproxy/wiki/ZAP-Baseline-Scan`.

This is what the script does:

- Runs ZAP spider against the specified target for one minute and then does a passive scan
- By default, reports all alerts as warnings
- This script is intended to be ideal to run in a CI/CD environment, even against production sites

Before setting up and running the ZAP Baseline scan, we want to run a simple vulnerable application so that all scans and testing using ZAP are running against that application, rather than running the scans against real-world applications, which is illegal without permission.

Running a vulnerable application container

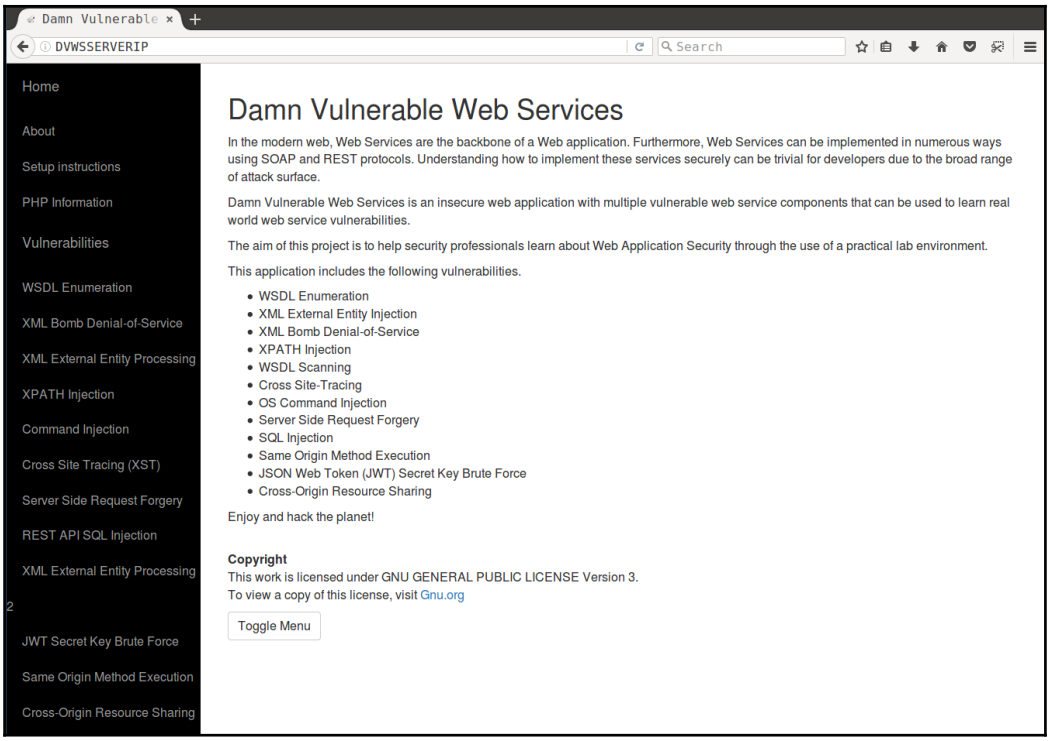
We will be using the **Damn Vulnerable Web Services** (**DVWS**) application (for more information, you can visit `https://github.com/snoopysecurity/dvws`). It is an insecure web application with multiple vulnerable web service components that can be used to learn real-world web service vulnerabilities.

The following playbook will set up the Docker container for running the DVWS application:

```
- name: setting up DVWS container
  hosts: dvws
  remote_user: "{{ remote_user_name }}"
  gather_facts: no
  become: yes
```

```
vars:
  remote_user_name: ubuntu
  dvws_image_name: cyrivs89/web-dvws

tasks:
  - name: pulling {{ dvws_image_name }} container
    docker_image:
      name: "{{ dvws_image_name }}"

  - name: running dvws container
    docker_container:
      name: dvws
      image: "{{ dvws_image_name }}"
      interactive: yes
      state: started
      ports:
        - "80:80"
```

Once the playbook is successfully executed, we can navigate to `http://DVWSSERVERIP`:



DVWS application home page

Now, we are ready to perform our OWASP ZAP Baseline scan against the DVWS application, by running the Baseline scan playbook.

Running an OWASP ZAP Baseline scan

The following playbook runs the Docker Baseline scan against a given website URL. It also stores the output of the Baseline's scan in the host system in HTML, Markdown, and XML formats:

```
- name: Running OWASP ZAP Baseline Scan
  hosts: zap
  remote_user: "{{ remote_user_name }}"
  gather_facts: no
  become: yes
  vars:
    remote_user_name: ubuntu
    owasp_zap_image_name: owasp/zap2docker-weekly
    website_url: {{ website_url }}
    reports_location: /zapdata/
    scan_name: owasp-zap-base-line-scan-dvws

  tasks:
    - name: adding write permissions to reports directory
      file:
        path: "{{ reports_location }}"
        state: directory
        owner: root
        group: root
        recurse: yes
        mode: 0770

    - name: running owasp zap baseline scan container against "{{
website_url }}"
      docker_container:
        name: "{{ scan_name }}"
        image: "{{ owasp_zap_image_name }}"
        interactive: yes
        auto_remove: yes
        state: started
        volumes: "{{ reports_location }}:/zap/wrk:rw"
        command: "zap-baseline.py -t {{ website_url }} -r {{ scan_name
}}_report.html"

    - name: getting raw output of the scan
      command: "docker logs -f {{ scan_name }}"
      register: scan_output
```

```
- debug:
    msg: "{{ scan_output }}"
```

Let's explore the parameters of the preceding playbook:

- `website_url` is the domain (or) URL that you want to perform the Baseline scan, we can pass this via `--extra-vars "website_url: http://192.168.33.111"` from the `ansible-playbook` command
- `reports_location` is the path to ZAP host machine where reports get stored

The following screenshot is the scanning report output from OWASP ZAP:

ZAP Scanning Report

Summary of Alerts

Risk Level	Number of Alerts
High	0
Medium	3
Low	5
Informational	2

Alert Detail

Medium (Medium)	X-Frame-Options Header Not Set
Description	X-Frame-Options header is not included in the HTTP response to protect against 'ClickJacking' attacks.
URL	http://192.168.33.111/dvws/vulnerabilities/cors/?C=S;O=A
Method	GET
Parameter	X-Frame-Options
URL	http://192.168.33.111/dvws/about.php
Method	GET
Parameter	X-Frame-Options
URL	http://192.168.33.111/dvws/vulnerabilities/xst/?C=M;O=D
Method	GET
Parameter	X-Frame-Options
URL	http://192.168.33.111/dvws/vulnerabilities/xxe2/
Method	GET
Parameter	X-Frame-Options
URL	http://192.168.33.111/dvws/vulnerabilities/?C=N;O=A
Method	GET
Parameter	X-Frame-Options
URL	http://192.168.33.111/dvws/vulnerabilities/hiddendir/?C=M;O=A
Method	GET

OWASP ZAP Baseline scan HTML report

To generate reports in the Markdown and XML formats, add `-w report.md` and `-x report.xml`, respectively, to command.

Security testing against web applications and websites

Until now, we have seen how to run a Baseline scan using the OWASP ZAP container. Now we will see how we can perform active scans against web applications. An active scan may cause the vulnerability to be exploited in the application. Also, this type of scan requires extra configuration, which includes authentication and sensitive functionalities.

Running ZAP full scan against DVWS

The following playbook will run the full scan against the DVWS application. Now we can see that the playbook looks almost similar, except the flags sent to command:

```
- name: Running OWASP ZAP Full Scan
  hosts: zap
  remote_user: "{{ remote_user_name }}"
  gather_facts: no
  become: yes
  vars:
    remote_user_name: ubuntu
    owasp_zap_image_name: owasp/zap2docker-weekly
    website_url: {{ website_url }}
    reports_location: /zapdata/
    scan_name: owasp-zap-full-scan-dvws

  tasks:
    - name: adding write permissions to reports directory
      file:
```

```
        path: "{{ reports_location }}"
        state: directory
        owner: root
        group: root
        recurse: yes
        mode: 0777

    - name: running owasp zap full scan container against "{{ website_url
}}"
      docker_container:
        name: "{{ scan_name }}"
        image: "{{ owasp_zap_image_name }}"
        interactive: yes
        auto_remove: yes
        state: started
        volumes: "{{ reports_location }}:/zap/wrk:rw"
        command: "zap-full-scan.py -t {{ website_url }} -r {{ scan_name
}}_report.html"

    - name: getting raw output of the scan
      raw: "docker logs -f {{ scan_name }}"
      register: scan_output

    - debug:
        msg: "{{ scan_output }}"
```

The OWASP ZAP full scan checks for a lot of vulnerabilities, which includes OWASP TOP 10 (for more information visit `https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project`) and many others. This can be intrusive to the application and it sends active requests to the application. It may cause damage to the functionality based on the vulnerability that exists in the application:

ZAP Scanning Report

Summary of Alerts

Risk Level	Number of Alerts
High	5
Medium	5
Low	5
Informational	2

Alert Detail

High (Medium)	Anti CSRF Tokens Scanner
Description	A cross-site request forgery is an attack that involves forcing a victim to send an HTTP request to a target destination without their knowledge or intent in order to perform an action as the victim. The underlying cause is application functionality using predictable URL/form actions in a repeatable way. The nature of the attack is that CSRF exploits the trust that a web site has for a user. By contrast, cross-site scripting (XSS) exploits the trust that a user has for a web site. Like XSS, CSRF attacks are not necessarily cross-site, but they can be. Cross-site request forgery is also known as CSRF, XSRF, one-click attack, session riding, confused deputy, and sea surf. CSRF attacks are effective in a number of situations, including: * The victim has an active session on the target site. * The victim is authenticated via HTTP auth on the target site. * The victim is on the same local network as the target site. CSRF has primarily been used to perform an action against a target site using the victim's privileges, but recent techniques have been discovered to disclose information by gaining access to the response. The risk of information disclosure is dramatically increased when the target site is vulnerable to XSS, because XSS can be used as a platform for CSRF, allowing the attack to operate within the bounds of the same-origin policy.
URL	http://192.168.33.111/dvws/vulnerabilities/jwt/api.php
Method	POST
Evidence	<form method="post" action="">
URL	http://192.168.33.111/dvws/vulnerabilities/jwt/login.php
Method	GET

OWASP ZAP full scan for DVWS application report

The preceding screenshot is the report from the OWASP ZAP full scan for the DVWS application. We can clearly see the difference between the Baseline scan and the full scan, based on the number of vulnerabilities, different types of vulnerabilities, and risk rating.

Testing web APIs

Similar to the ZAP Baseline scan, the fine folks behind ZAP provide a script as part of their live and weekly Docker images. We can use it to run scans against API endpoints defined either by OpenAPI specification or **Simple Object Access Protocol** (**SOAP**).

The script can understand the API specifications and import all the definitions. Based on this, it runs an active scan against all the URLs found:

```
- name: Running OWASP ZAP API Scan
  hosts: zap
  remote_user: "{{ remote_user_name }}"
  gather_facts: no
  become: yes
  vars:
    remote_user_name: ubuntu
    owasp_zap_image_name: owasp/zap2docker-weekly
    website_url: {{ website_url }}
    reports_location: /zapdata/
    scan_name: owasp-zap-api-scan-dvws
    api_type: openapi
>
  tasks:
    - name: adding write permissions to reports directory
      file:
        path: "{{ reports_location }}"
        state: directory
        owner: root
        group: root
        recurse: yes
        mode: 0777

    - name: running owasp zap api scan container against "{{ website_url
}}"
      docker_container:
        name: "{{ scan_name }}"
        image: "{{ owasp_zap_image_name }}"
        interactive: yes
        auto_remove: yes
        state: started
        volumes: "{{ reports_location }}:/zap/wrk:rw"
        command: "zap-api-scan.py -t {{ website_url }} -f {{ api_type }} -r
{{ scan_name }}_report.html"

    - name: getting raw output of the scan
      raw: "docker logs -f {{ scan_name }}"
      register: scan_output
```

```
  - debug:
      msg: "{{ scan_output }}"
```

Continuous scanning workflow using ZAP and Jenkins

Jenkins is an open source automation server. It is used extensively in CI/CD pipelines. These pipelines usually refer to a series of automated steps that occur based on triggers, such as code commits to version control software or a new release being created.

We already saw the example of ZAP Baseline's scans being part of the Mozilla release cycle. We can integrate ZAP with Jenkins. While there are many ways we can do this, a useful set of steps will be the following:

1. Based on a trigger, a new ZAP instance is ready for scanning
2. The ZAP instance runs against an automatically deployed application
3. The results of the scan are captured and stored in some format
4. If we choose, the results can also create tickets in bug tracking systems such as Atlassian Jira

For this, we will set up our pipeline infrastructure first:

1. Set up Jenkins using a playbook
2. Add the official OWASP ZAP Jenkins plugin
3. Trigger the workflow using another playbook

The official OWASP ZAP Jenkins plugin can be found at `https://wiki.jenkins.io/display/JENKINS/zap+plugin`.

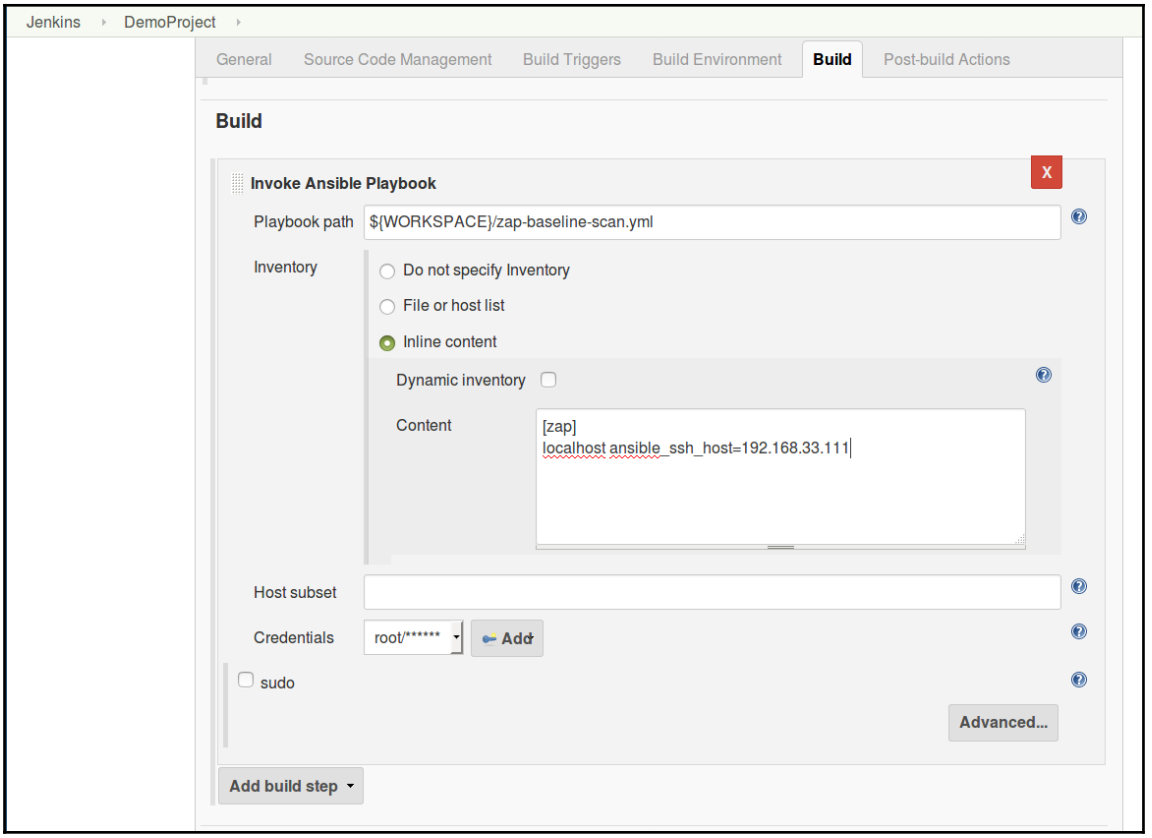
Setting up Jenkins

Set up Jenkins on the server to be used as a CI/CD platform for OWASP ZAP. This will return the Jenkins administrator password and once it has been done, we can install the Ansible plugin:

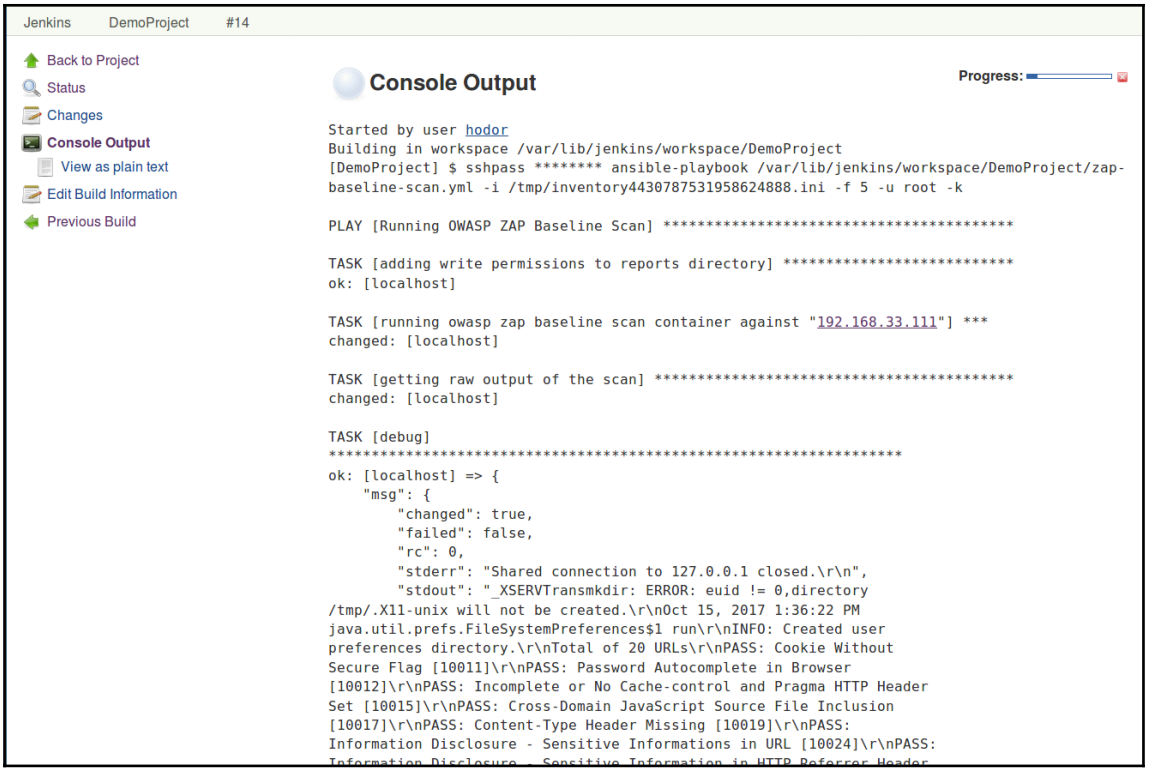
```
- name: installing jenkins in ubuntu 16.04
  hosts: jenkins
  remote_user: {{ remote_user_name }}
```

```
gather_facts: False
become: yes
vars:
  remote_user_name: ubuntu

tasks:
  - name: adding jenkins gpg key
    apt_key:
      url: 'https://pkg.jenkins.io/debian/jenkins-ci.org.key'
      state: present

  - name: jeknins repository to system
    apt_repository:
      repo: 'deb http://pkg.jenkins.io/debian-stable binary/'
      state: present
  - name: installing jenkins
    apt:
      name: jenkins
      state: present
      update_cache: yes

  - name: adding jenkins to startup
    service:
      name: jenkins
      state: started
      enabled: yes

  - name: printing jenkins default administration password
    command: cat "/var/lib/jenkins/secrets/initialAdminPassword"
    register: jenkins_default_admin_password

  - debug:
      msg: "{{ jenkins_default_admin_password.stdout }}"
```

Then, we can add the playbook to the project. When the new trigger happens in the Jenkins build, the playbook will start to scan the website to perform the Baseline scan:



Once the playbook triggers, it will execute the playbook against the URL and return the ZAP Baseline scan output:

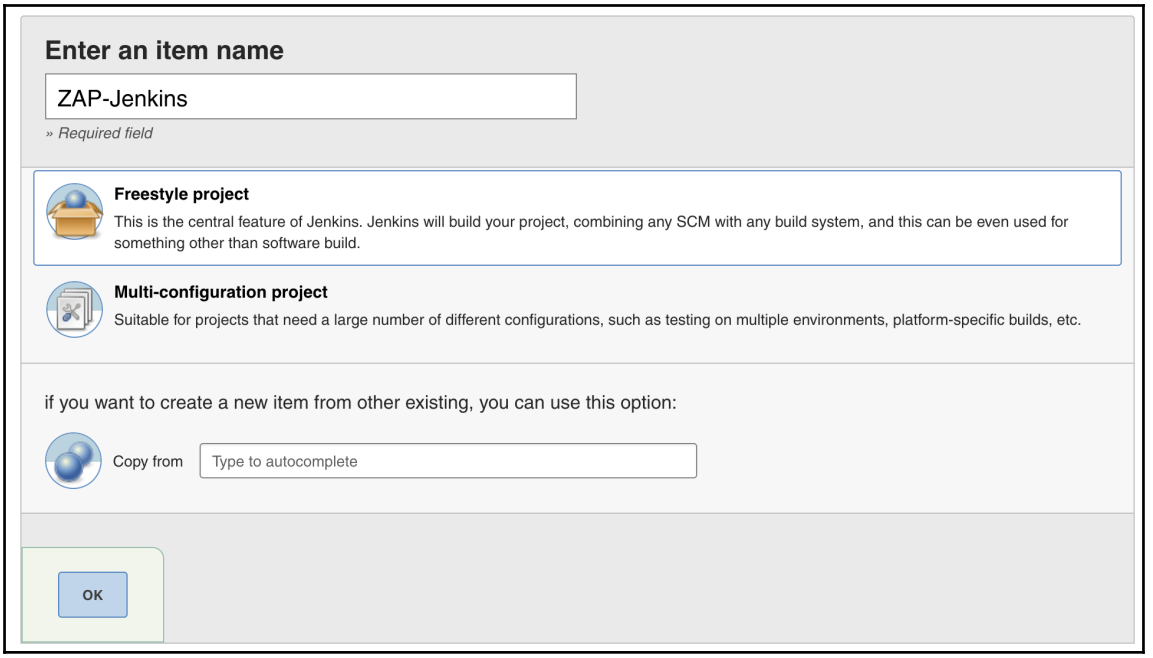


Setting up the OWASP ZAP Jenkins plugin

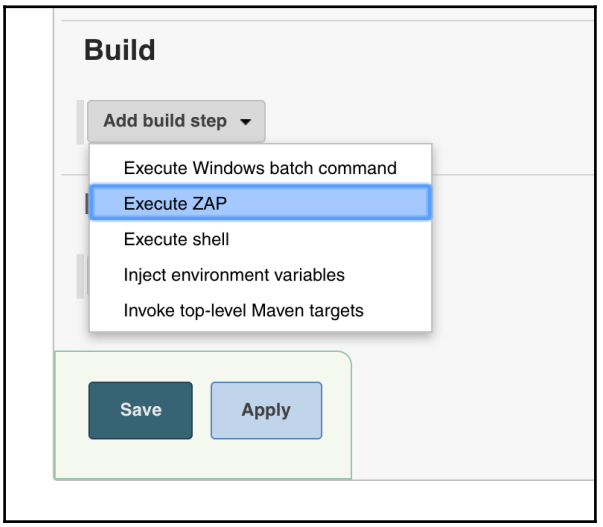
OWASP ZAP working in tandem with Jenkins is a fairly well-known setup. We already know how to set up Jenkins. We can install the official ZAP Jenkins plugin using our playbook.

Once the playbook is ready, a bit of manual configuration is required. We start after our playbook has installed Jenkins and restarted the server so that the plugin is available for our build jobs.

Let's create a new build job and call it `ZAP-Jenkins`, as shown in the following screenshot:



This will be a freestyle project for us. Now we will add the magic of ZAP to this:

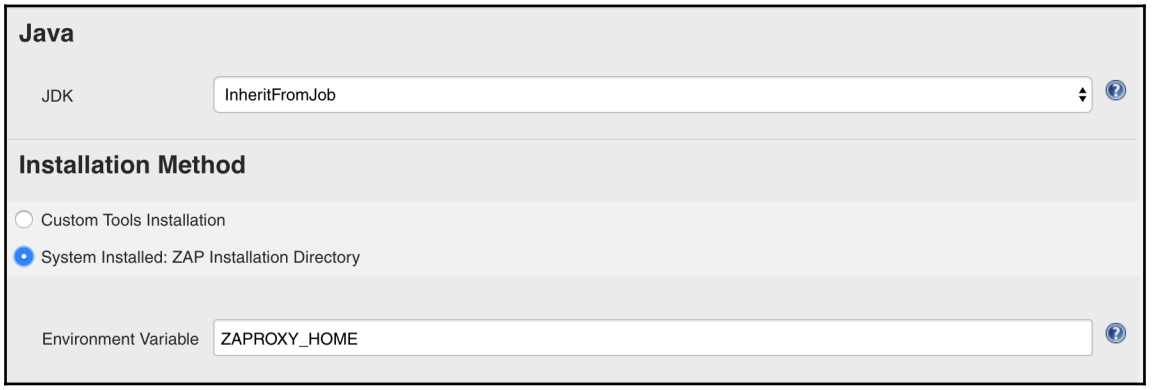


We are following the instructions as given on the Jenkins page for the official plugin: `https://wiki.jenkins.io/display/JENKINS/zap+plugin`.

Some assembly required

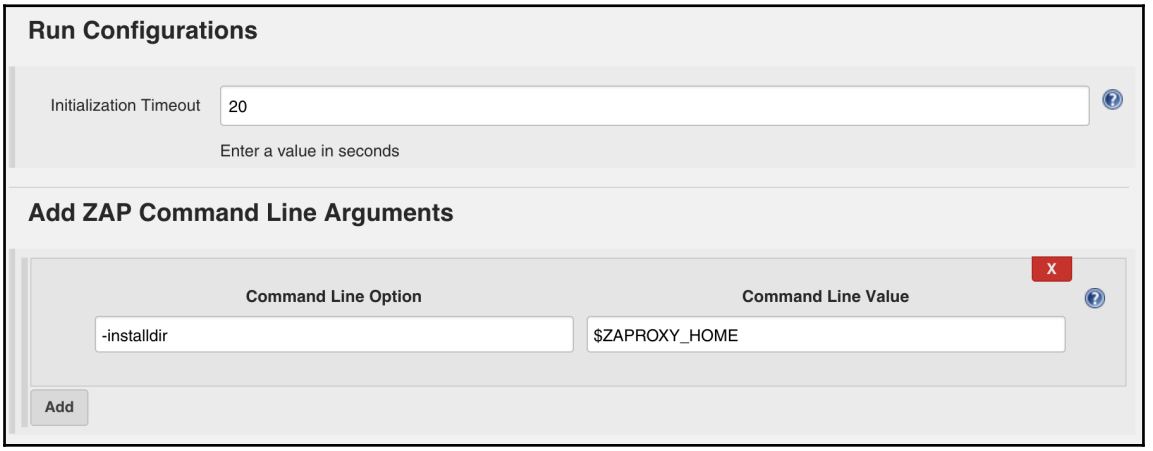
Specify the interface's IP address and the port number on which ZAP should be listening. Usually, this port is `8080`, but since Jenkins is listening on that, we choose `8090`:

For JDK, we choose the only available option, **InheritFromJob**:



For the installation method, we select the ZAP that is already installed on `/usr/share/owasp-zap`. We add this value to a `ZAPROXY_HOME` environment variable in `/etc/environment`.

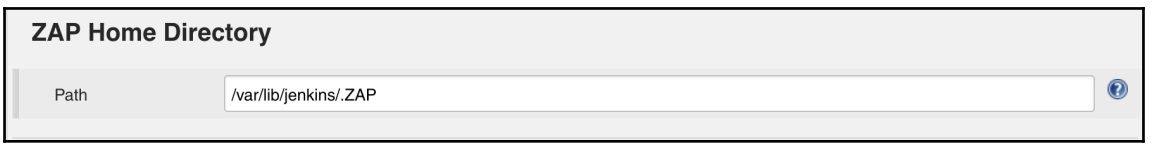
By doing this, we have ensured that the environment variable values will survive a system reboot as well:



We specify a fairly small value for a timeout to ensure that in case something goes wrong, we don't have to wait long to see that the build failed or ZAP isn't responding.

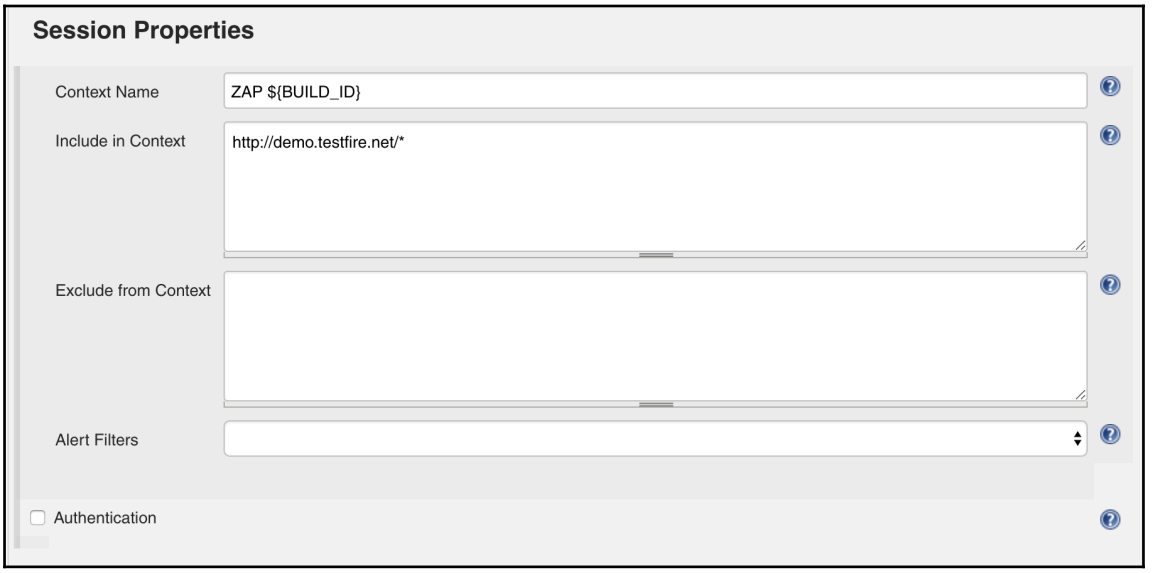
We also specify a command-line option to tell Jenkins what the install directory for ZAP is.

You may need to click on the **Advanced** button to see these options.

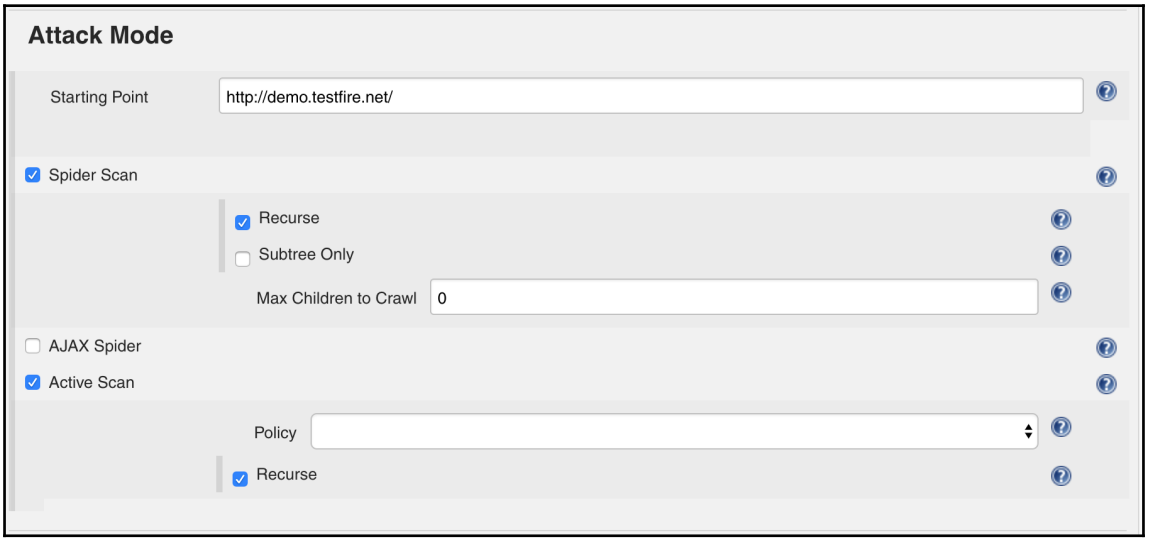


We specify the path to the ZAP home directory:

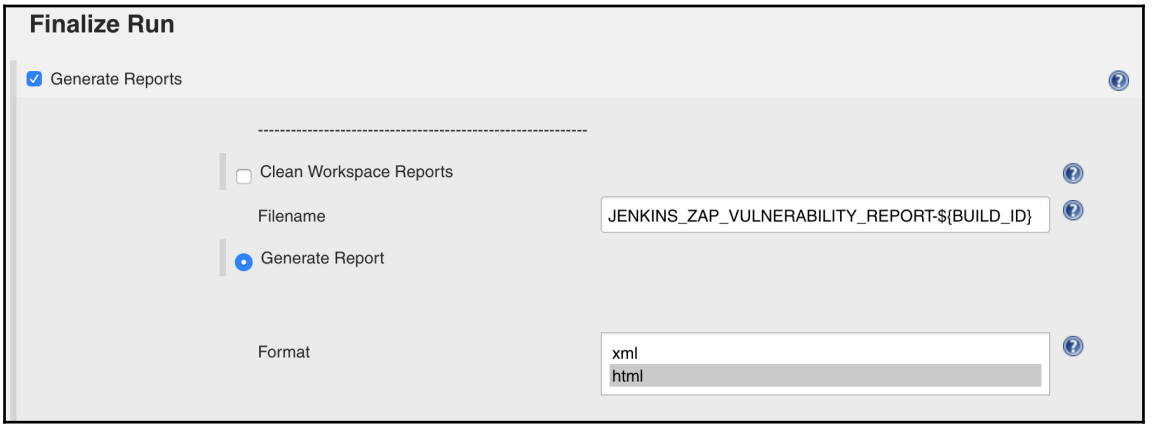
Then we configure where we plan to load the ZAP session from:



The context name, scope, and exclusions are shown here:



This is the starting point of the URL to test. The kind of test we are planning to do is **Spider Scan**, default **Active Scan**:



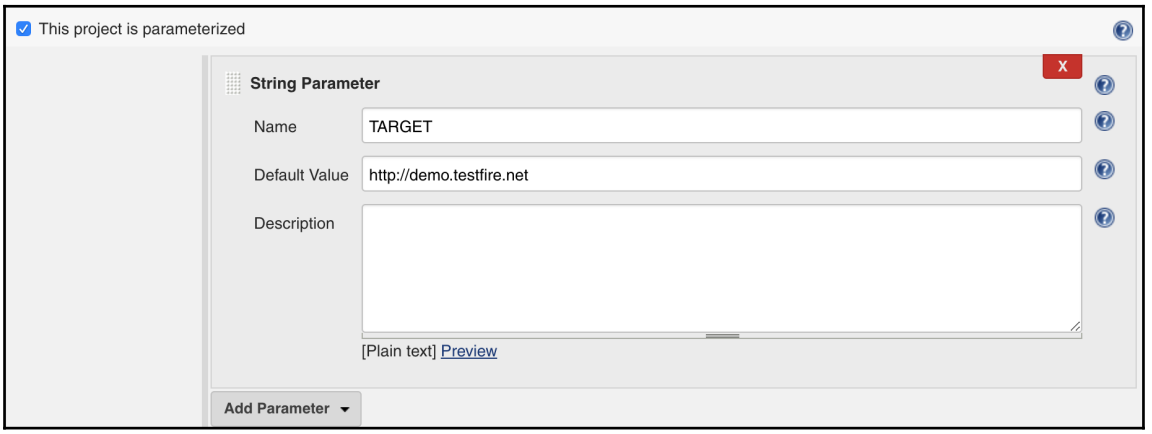
Finally, we specify the filename for the report that will be generated. We are adding the `BUILD_ID` variable to ensure that we don't have to worry about overwriting the reports.

Triggering the build (ZAP scan)

Once the job is configured, we are ready to trigger the build. Of course, you can manually click **Build now** and get going.

But we will configure the build job to be triggered remotely, and at the same time pass the necessary target information.

Under **General** check **This project is parameterized**:



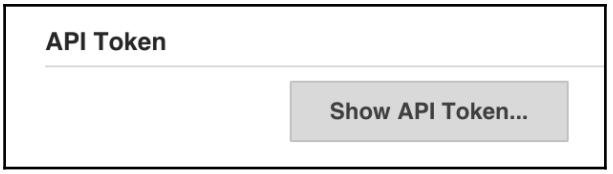
Inside that, we add a `TARGET` parameter with a default value.

Under **Build Triggers**, we specify an authentication token to be passed as a parameter while remotely triggering a build:

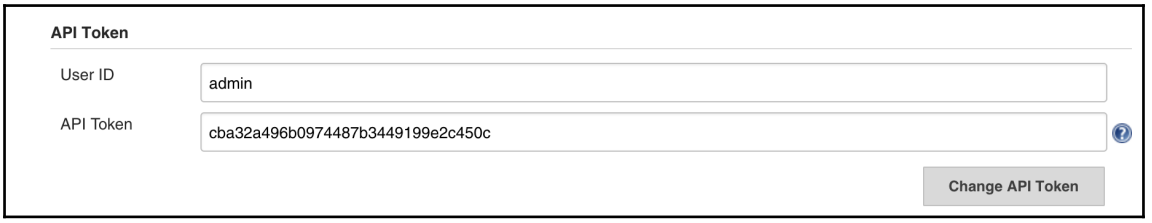
Try to ensure that this token is sufficiently lengthy and random, and not the simple word we have used as an example.

A great way to generate sufficiently random strings in Linux/macOS is to use the OpenSSL command. For the `hex` output (`20` is the length of the output), use **`openssl rand -hex 20`**. For the `base64` output (`24` is the length of the output), use **`openssl rand -base64 24`**.

At this point, all we have to do is note the **API Token** of the logged in user (from `http://JENKINS-URL/user/admin/configure`):



Clicking **Show API Token** will show the token:



We can now use a command-line tool, such as `curl`, to see if this works.

The format of the link is `curl "http://username:API-TOKEN@JENKINS-URL/job/ZAP-Jenkins/buildWithParameters?TARGET=http://demo.testfire.net&token=ansible2security"`.

This will trigger the build and the application will get scanned for security issues.

Playbook to do this with automation

To perform the preceding trigger, we can use the following Ansible playbook. This can be used in our Ansible Tower to schedule the scan as well.

The following playbook can store the the API Token key using Ansible Vault, feature to store secret data in an encrypted format in playbooks. We will learn more about Ansible Vault usage in `Chapter 11`, *Ansible Security Best Practices, References and Further Reading*.

To create an Ansible Vault encrypted variable, run the following command. When it prompts for a password, give a password to encrypt this variable and it requires while executing the playbook

```
echo 'YOURTOKENGOESHERE' | ansible-vault encrypt_string --stdin-name
'jenkins_api_token'
```

After executing, it returns the encrypted variable which we can use in the playbook it self directly as a variable:

```
- name: jenkins build job trigger
  hosts: localhost
  connection: local
  vars:
    jenkins_username: username
    jenkins_api_token: !vault |
          $ANSIBLE_VAULT;1.1;AES256
3663656331393231336631303062323262333833363836346534333963636235353436353636
66161
3062666536613764396439326534663237653438616335640a6135646433666234626663616
33763
3132616130366665336634393136626533323838393765643566306136366564343133663838
53436
3532646434376533390a6463326466396531613431653638326162333332323231306230343
13032
6664353733663463326334636331343766626232306438653961633336461323533336
    jenkins_host: 192.168.11.111
    jenkins_target_url: 'http://demo.testfire.net'
    jenkins_token: ansible2security
>
  tasks:
    - name: trigger jenkins build
      uri:
        url: "http://{{ jenkins_username }}:{{ jenkins_api_token }}@{{
jenkins_host }}/job/ZAP-Jenkins/buildWithParameters?TARGET={{
jenkins_target_url }}&token={{ jenkins_token }}"
        method: GET
      register: results
    - debug:
        msg: "{{ results.stdout }}"
```

To perform the `ansible-vault` decryption while executing the playbook, the playbook execution command looks like this:

```
$ ansible-playbook --ask-vault-pass main.yml
```

ZAP Docker and Jenkins

There is a great blog series by the folks at Mozilla about configuring the ZAP Docker with Jenkins. Rather than repeating what they have to say, we thought it made sense to point you to the first post in that series.

For further reading, you can check out the interesting blog *Dockerized, OWASP-ZAP security scanning, in Jenkins, part one at* `https://blog.mozilla.org/webqa/2016/05/11/docker-owasp-zap-part-one/`.

Summary

OWASP ZAP is a great addition to any security team's arsenal of tools. It provides complete flexibility in terms of what we can do with it and how it can fit into our setup. By combining ZAP with Jenkins, we can quickly set up a decent production-worthy continuous scanning workflow and align our process around it. Ansible allows us to install and configure all of these great tools using playbooks. This is great as it is mostly a one-time effort and then we can start seeing the results and the reports for ZAP.

Now that we are on our way to automating security tools, next we shall see the most popular vulnerability assessment tool, Nessus, and how we can build a similar workflow for vulnerability assessment for software and networks.

10
Writing an Ansible Module for Security Testing

Ansible primarily works by pushing small bits of code to the nodes it connects to. These codes/programs are what we know as Ansible modules. Typically in the case of a Linux host these are copied over SSH, executed, and then removed from the node.

As stated in the Ansible Developer Guide (the best resource for all things Ansible-related):

> *"Ansible modules can be written in any language that can return JSON."*

Modules can be used by the Ansible command-line, in a playbook, or by the Ansible API. There are already hundreds of modules that ship with Ansible version 2.4.x.

Have a look at the module index on the Ansible documentation site: `http://docs.ansible.com/ansible/latest/modules_by_category.html`.

Currently, there are over 20 categories of modules with categories such as cloud, storage, Remote Management, and Windows.

Sometimes in spite of all the modules out there, you may need to write your own. This chapter will take you through writing a module that you can use with your Ansible playbooks.

Ansible has an extremely detailed development guide (`http://docs.ansible.com/ansible/latest/dev_guide/index.html`) that is the best place to start if you are planning to contribute your modules to be shipped with Ansible.

This chapter is not at all meant to replace that. Consider that if you plan to write modules for your internal use and you are not fussed about distributing them, this chapter offers you a simple-to-follow path where we will end up with a working module for enabling security automation, which has been our goal throughout.

We will look at the following:

- How to set up the development environment
- Writing an Ansible hello world module to understand the basics
- Where to seek further help
- Defining a security problem statement
- Addressing that problem by writing a module of our own

Along with that, we will try to understand and attempt to answer the following questions:

- What are the good use cases for modules?
- When does it make sense to use roles?
- How do modules differ from plugins?

Let's get started with a simple hello world module.

Getting started with a hello world Ansible module

We will pass one argument to our custom module and show if we have success or failure for the module executing based on that.

Since all of this is new to us, we will look at the following things:

- The source code of the hello world module
- The output of that module for both success and failure
- The command that we will use to invoke it

Before we get started, all of this is based on the Ansible Developer Guide! The following code is in Python.

Code

We use Python for many scripting tasks, but we are not experts in it. But we believe this code is simple enough to understand:

```
from ansible.module_utils.basic import AnsibleModule

module = AnsibleModule(
    argument_spec=dict(
        answer=dict(choices=['yes', 'no'], default='yes'),
    )
)

answer = module.params['answer']
if answer == 'no':
    module.fail_json(changed=True, msg='Failure! We failed because we
answered no.')

module.exit_json(changed=True, msg='Success! We passed because we answered
yes.')
```

1. We are importing some modules.
2. The second part is just how we need to declare the arguments we will accept for the module.
3. In our code, we can refer to the arguments the way we have taken the value of the `answer` variable.
4. Based on the answer, if it is `no`, we indicate failure.
5. If the answer is `yes`, we indicate success.

Let's see what the output of this looks like if we provide answer as `yes`:

```
$ ANSIBLE_LIBRARY=. ansible -m ansible_module_hello_world.py -a answer=yes
localhost

 [WARNING]: provided hosts list is empty, only localhost is available

localhost | SUCCESS => {
    "changed": true,
    "msg": "Success! We passed because we answered yes."
}
```

And if the answer is `no`:

```
$ ANSIBLE_LIBRARY=. ansible -m ansible_module_hello_world -a answer=no
localhost

 [WARNING]: provided hosts list is empty, only localhost is available

localhost | FAILED! => {
    "changed": true,
    "failed": true,
    "msg": "Failure! We failed because we answered no."
}
```

The main difference in the output is the indication of either the `SUCCESS` or `FAILED` status and the message that we provided.

Since we haven't set up the development environment so far, we set an environment variable for this command:

- `ANSIBLE_LIBRARY=.` indicates that search the module to be executed in the current directory
- With `-m`, we call our module
- With `-a`, we pass the module argument, which in this case is answered with possible values of `yes` or `no`
- We end with the host that we want to run the module on, which is local for this example

While Ansible is written in Python, please note that the modules can be written in any language capable of returning messages in JSON. A great starting point for Rubyists is the Ansible for Rubyists (`https://github.com/ansible/ansible-for-rubyists`) repository on Github. Chapter 5 of *Learning Ansible* by Packt has covered this as well.

Setting up the development environment

The primary requirement for Ansible 2.4 is Python 2.6 or higher and Python 3.5 or higher. If you have either of them installed, we can follow the simple steps to get the development environment going.

From the Ansible Developer Guide:

1. Clone the Ansible repository: `$ git clone https://github.com/ansible/ansible.git`
2. Change the directory into the repository root directory: `$ cd ansible`
3. Create a virtual environment: `$ python3 -m venv venv (or for Python 2 $ virtualenv venv`
4. Note, this requires you to install the `virtualenv` package: `$ pip install virtualenv`
5. Activate the virtual environment: `$ . venv/bin/activate`
6. Install the development requirements: `$ pip install -r requirements.txt`
7. Run the environment setup script for each new dev shell process: `$ . hacking/env-setup`

You should end up with a `venv` prompt at this point. Here is a simple playbook to set up the development environment.

The following playbook will set up the developer environment by installing and setting up the virtual environment:

```
- name: Setting Developer Environment
  hosts: dev
  remote_user: madhu
  become: yes
  vars:
    ansible_code_path: "/home/madhu/ansible-code"

  tasks:
    - name: installing prerequirements if not installed
      apt:
        name: "{{ item }}"
        state: present
        update_cache: yes
      with_items:
        - git
        - virtualenv
        - python-pip
    - name: downloading ansible repo locally
      git:
        repo: https://github.com/ansible/ansible.git
        dest: "{{ ansible_code_path }}/venv"
    - name: creating virtual environment
      pip:
        virtualenv: "{{ ansible_code_path }}"
```

```
        virtualenv_command: virtualenv
        requirements: "{{ ansible_code_path }}/venv/requirements.txt"
```

The following screenshot shows the playbook execution of the developer environment setup for writing your own Ansible modules using the Python virtual environment:

```
PLAY [Setting Developer Environment] *******************************************

TASK [Gathering Facts] *********************************************************
ok: [172.16.1.119]

TASK [installing prerequirements if not installed] *****************************
ok: [172.16.1.119] => (item=[u'git', u'virtualenv', u'python-pip'])

TASK [downloading ansible repo locally] ****************************************
ok: [172.16.1.119]

TASK [creating virtual environment] ********************************************
changed: [172.16.1.119]

PLAY RECAP *********************************************************************
172.16.1.119               : ok=4    changed=1    unreachable=0    failed=0
```

Planning and what to keep in mind

The Ansible Developer Guide has a section on how should you develop a module (`http://docs.ansible.com/ansible/latest/dev_guide/developing_modules.html#should-you-develop-a-module`).

In the section, they have multiple points on what to keep in mind before going ahead and developing a module.

Does a similar module already exist? It's always a good idea to check the current modules to see whether what you plan to build has been done before. The good news is, so far nobody has built an **Open Web Application Security Project** (**OWASP**) **Zed Attack Proxy** (**ZAP**) module.

Has someone already worked on a similar *Pull Request?* Again, maybe the module hasn't been published but that doesn't mean that folks are not working on it already. The document provides three convenient links to check if a similar PR is already in place.

Additionally, it asks if rather than a module, we should look at an action plugin or role. The main reason we think it makes sense for us to develop the module is the fact that it will run on the nodes. ZAP provides an API endpoint if it is already running and we intend for our module to make it easy for us to run ZAP scans on hosted ZAP instances.

So, this is the plan for now:

1. Create a module that will connect to a hosted ZAP instance.
2. Provide the module with two main pieces of information:
    - IP address of the hosted ZAP
    - Target URL for scanning
3. By calling the module, we will have a task for scanning the target application.

OWASP ZAP module

OWASP ZAP has an API that we can use. Additionally, there is a Python module for consuming the API. We will try and use that to learn how to write our own Ansible modules.

Create ZAP using Docker

For our development, let's use a Docker container to get ZAP going. Since we plan to use the API, we will run the container in headless mode:

```
$ docker run -u zap -p 8080:8080 -i owasp/zap2docker-stable zap.sh -daemon
-host 0.0.0.0 -port 8080 -config api.disablekey=true -config
api.addrs.addr.name=.* -config api.addrs.addr.regex=true
```

Explanation of the command

- While we are doing dev, we can disable the API key: `-config api.disablekey=true`
- Allow access to the API from any IP: `-config api.addrs.addr.name=.* -config api.addrs.addr.regex=true`
- Listen to port `8080`

If everything worked fine, you will see the following output:

```
2594 [ZAP-daemon] INFO org.parosproxy.paros.extension.ExtensionLoader  - Initializing Easy way to
 replace strings in requests and responses
2689 [ZAP-daemon] INFO org.zaproxy.zap.extension.callback.ExtensionCallback  - Started callback s
erver on 0.0.0.0:40083
2689 [ZAP-daemon] INFO org.zaproxy.zap.extension.dynssl.ExtensionDynSSL  - Creating new root CA c
ertificate
3089 [ZAP-daemon] INFO org.zaproxy.zap.extension.dynssl.ExtensionDynSSL  - New root CA certificat
e created
3091 [ZAP-daemon] INFO org.zaproxy.zap.DaemonBootstrap  - ZAP is now listening on 0.0.0.0:8080
```

Creating a vulnerable application

For a vulnerable application, we can host one of our own but let's use the same online vulnerable application we used for the OWASP ZAP + Jenkins integration in `Chapter 5`, *Automating Web Application Security Testing Using OWASP ZAP* - `http://testphp.vulnweb.com/`

Ansible module template

We will take the sample code given in the module development guide to get started: `http://docs.ansible.com/ansible/latest/dev_guide/developing_modules_general.html#new-module-development`.

This template has a well-commented code and it is written in a manner that makes it easy for us to get started. The code is divided into the following parts:

- Metadata
- Documenting the module
- Functions we will be using

Metadata

This section contains the information about the modules:

```
ANSIBLE_METADATA = {
    'metadata_version': '1.1',
    'status': ['preview'],
    'supported_by': 'community'
}
```

This module isn't supported officially, hence the use of `community`.

Documenting the module

The module documentation is generated from the module code itself. The `DOCUMENTATION` docstring is compulsory for the modules to be created now.

The easiest way to get started is to look at this example: `https://github.com/ansible/ansible/blob/devel/examples/DOCUMENTATION.yml`.

The list of fields required here are:

- `module`: Module name
- `short_description`: Short description
- `description`: Description
- `version_added`: Indicated by `X.Y`
- `author`: Your name and twitter/GitHub username
- `options`: Each of the options supported by the module
- `notes`: Anything else that a module user should be aware of
- `requirements`: We list additional package requirements

For more details about the fields, visit `http://docs.ansible.com/ansible/latest/dev_guide/developing_modules_documenting.html#fields`.

Source code template

Here are some snippets of the source code that we will work with to write our module. We have already discussed the metadata and documentation parts. We also need to write docstrings for examples and what the module will be returning.

Our imports—we can import all the modules we need for our module here:

```
from ansible.module_utils.basic import AnsibleModule
```

The main code block—inside the function `run_module` we work and do the following:

1. Define all the arguments we need for the module to work.
2. Initialize the results dictionary.
3. Create the `AnsibleModule` object and pass it common attributes that we may need:

```
def run_module():
    # define the available arguments/parameters that a user can pass to
    # the module
    module_args = dict(
        name=dict(type='str', required=True),
        new=dict(type='bool', required=False, default=False)
    )

    # seed the result dict in the object
    # we primarily care about changed and state
    # change is if this module effectively modified the target
    # state will include any data that you want your module to pass back
    # for consumption, for example, in a subsequent task
    result = dict(
        changed=False,
        original_message='',
        message=''
    )

    # the AnsibleModule object will be our abstraction working with Ansible
    # this includes instantiation, a couple of common attr would be the
    # args/params passed to the execution, as well as if the module
    # supports check mode
    module = AnsibleModule(
        argument_spec=module_args,
        supports_check_mode=True
    )
```

4. Working with exceptions and results:

```
# during the execution of the module, if there is an exception or a
   # conditional state that effectively causes a failure, run
   # AnsibleModule.fail_json() to pass in the message and the result
   if module.params['name'] == 'fail me':
       module.fail_json(msg='You requested this to fail', **result)

   # in the event of a successful module execution, you will want to
   # simple AnsibleModule.exit_json(), passing the key/value results
   module.exit_json(**result)
```

Just remember the following:

- If we hit any kind of errors or exceptions, we invoke the `fail_json` function of the `AnsibleModule` object
- If everything worked out well, we invoke the `exit_json` function of the same object

Invoking our function completes the code:

```
def main():
    run_module()

if __name__ == '__main__':
    main()
```

At this point, we have the following things in place and we are ready for the next steps:

Template of the module code	Ready
Vulnerable application that we need to scan (target)	Ready
OWASP ZAP Proxy with API enabled and running in headless mode (host and port)	Ready
OWASP ZAP Python API code that we can refer to	Pending

We want to focus on writing the Ansible module rather than spending time learning the complete OWASP ZAP API. While we recommend that you do, it's fine to wait until you have gotten the module working.

OWASP ZAP Python API sample script

OWASP ZAP Python API package comes with a very handy script that is complete in terms of code for spidering and doing an active scan of a web application.

Download the code to study it from `https://github.com/zaproxy/zaproxy/wiki/ApiPython#an-example-python-script`.

Here are some snippets from sample code that we are interested in at this point. Import the Python API client for OWASP ZAP. This is installed using `pip install python-owasp-zap-v2.4`:

```
from zapv2 import ZAPv2
```

Now, we connect to the ZAP instance API endpoint. We can provide the host and port for the OWASP ZAP instance as an argument to our module:

```
zap = ZAPv2(apikey=apikey, proxies={'http': 'http://127.0.0.1:8090',
'https': 'http://127.0.0.1:8090'})
```

Provide the host/IP address of the website that we want to scan:

```
zap.urlopen(target)
# Give the sites tree a chance to get updated
time.sleep(2)

print 'Spidering target %s' % target
scanid = zap.spider.scan(target)

# Give the Spider a chance to start
time.sleep(2)
while (int(zap.spider.status(scanid)) < 100):
    print 'Spider progress %: ' + zap.spider.status(scanid)
    time.sleep(2)

print 'Spider completed'
# Give the passive scanner a chance to finish
time.sleep(5)

print 'Scanning target %s' % target
scanid = zap.ascan.scan(target)
while (int(zap.ascan.status(scanid)) < 100):
    print 'Scan progress %: ' + zap.ascan.status(scanid)
    time.sleep(5)
```

```
print 'Scan completed'

# Report the results

print 'Hosts: ' + ', '.join(zap.core.hosts)
print 'Alerts: '
pprint (zap.core.alerts())
```

This code is a great starter template for us to use in our module.

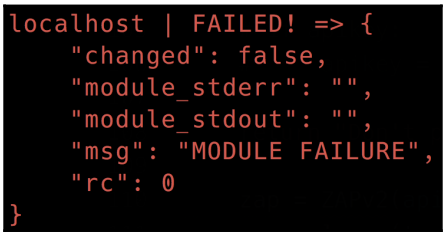
Here, we are ready with OWASP ZAP Python API code that we can refer to.

Connect to the ZAP instance. At this point, we copied the important bits of code that:

1. Connect to the target.
2. Initiate spidering and the active security scan.

But we quickly ran into an error. We were returning a string during an exception, which obviously wasn't in the JSON format as required by Ansible.

This resulted in an error which didn't have enough information for us to take action



Ansible modules should only return JSON, otherwise you may see cryptic errors such as above

A quick reading of conventions, best practices, and pitfalls at `http://docs.ansible.com/ansible/latest/dev_guide/developing_modules_best_practices.html#conventions-best-practices-and-pitfalls` explained the issue to us.

We strongly recommend that you go through this guide if you face any issues during your module writing: `http://docs.ansible.com/ansible/latest/dev_guide/developing_modules_best_practices.html#conventions-best-practices-and-pitfalls`.

Use the OWASP ZAP API documentation to learn more: `https://github.com/zaproxy/zaproxy/wiki/ApiGen_Index`.

Complete code listing

This code is also available on GitHub (`https://github.com/appsecco/ansible-module-owasp-zap`). All comments, metadata, and documentation doctstrings have been removed from this listing:

```
try:
    from zapv2 import ZAPv2
    HAS_ZAPv2 = True
except ImportError:
    HAS_ZAPv2 = False

from ansible.module_utils.basic import AnsibleModule
import time
def run_module():
    module_args = dict(
        host=dict(type='str', required=True),
        target=dict(type='str', required=True)
    )

    result = dict(
        changed=False,
        original_message='',
        message=''
    )

    module = AnsibleModule(
        argument_spec=module_args,
        supports_check_mode=True
    )

    if not HAS_ZAPv2:
        module.fail_json(msg = 'OWASP python-owasp-zap-v2.4 required. pip
install python-owasp-zap-v2.4')

if module.check_mode:
    return result
host = module.params['host']
target = module.params['target']
apikey = None
```

```
zap = ZAPv2(apikey=apikey, proxies={'http':host,'https':host})
zap.urlopen(target)
try:
    scanid = zap.spider.scan(target)
    time.sleep(2)
    while (int(zap.spider.status(scanid)) < 100):
        time.sleep(2)
except:
    module.fail_json(msg='Spidering failed')
time.sleep(5)

try:
    scanid = zap.ascan.scan(target)
    while (int(zap.ascan.status(scanid)) < 100):
        time.sleep(5)
except:
    module.fail_json(msg='Scanning failed')

result['output'] = zap.core.alerts()
result['target'] = module.params['target']
result['host'] = module.params['host']
module.exit_json(**result)

def main():
    run_module()
if __name__ == '__main__':
    main()
```

Depending on the website being spidered and scanned, this can take some time to finish. At the end of its execution, you will have the scanning results in `results['output']`.

Running the module

The choices we have for running the module are as follows:

1. We copy it to the standard path of Ansible library.
2. We provide a path to Ansible library whenever we have our module file.
3. Run this file through a playbook.

The following command will invoke our module for us to test and see the results:

```
ansible -m owasp_zap_test_module localhost -a
"host=http://172.16.1.102:8080 target=http://testphp.vulnweb.com" -vvv
```

Explanation of the command

- `ansible` command line
- `-m` to give the module name, which is `owasp_zap_test_module`
- It will run on `localhost`
- `-a` allows us to pass the `host` and `target` module arguments
- `-vvv` is for the verbosity of output

Playbook for the module

Here is a simple playbook to test whether everything is working:

```
- name: Testing OWASP ZAP Test Module
  connection: local
  hosts: localhost
  tasks:
  - name: Scan a website
    owasp_zap_test_module:
      host: "http://172.16.1.102:8080"
      target: "http://testphp.vulnweb.com"
```

Execute the playbook with this command:

```
ansible-playbook owasp-zap-site-scan-module-playbook.yml
```

```
PLAY [Testing OWASP ZAP Test Module] ********************************************************

TASK [Gathering Facts] **********************************************************************
ok: [localhost]

TASK [Scan a website] ***********************************************************************
ok: [localhost]

PLAY RECAP **********************************************************************************
localhost                  : ok=2    changed=0    unreachable=0    failed=0
```

An important thing to remember is that just because we have a working module doesn't mean that the good folks at Ansible will automatically accept our module to ship with their next version of the software. A lot of additional work is required for our module to be ready to be used by everyone.

As usual, the best guide for this is the developer guide mentioned earlier in this chapter.

One of the easy things to add to our module would be the ability to send the API key as an argument. Most ZAP instances that are being used for regular scanning will already have this configured. Additionally, this key can be protected by the Ansible vault when stored in the playbook.

Adding an API key as an argument

Just by making the following changes, we will be able to add `apikey` as an argument:

- First, we add this to the `module_args` dictionary on lines 76-78: `apikey=dict(type='str',required=False,default=None)`
- Then we check whether `module.params['apikey']` is set to a value of `None`
- If it is not, set it to `apikey = module.params['apikey']`
- Now, if the module is used with the Ansible command-line tool, pass it along with the `target` and `host`, and if it is used in the playbook, pass it there

Adding scan type as an argument

If you have followed so far, you may realize that the scan that we ran is an active scan. The scanner sends attack traffic against the target in an active scan.

Due to that fact, sometimes if the website is large, it may take a long time to finish.

More information about active scans can be found at `https://github.com/zaproxy/zap-core-help/wiki/HelpStartConceptsAscan`.

We would like to add an argument for being able to provide the type of scan to run. So far we have two types:

- **Active**: Sends attack traffic
- **Passive**: Parses all the site files downloaded during the spidering phase

We start by adding this as part of the `module_args`:

```
module_args = dict(
    host=dict(type='str', required=True),
    target=dict(type='str', required=True),
    apikey=dict(type='str',required=False,default=None),
```

```
        scantype=dict(default='passive', choices=['passive','active'])
    )
```

The newly added line is in bold to highlight the change. Notice that we have defined the default value now and this argument is only allowed two choices currently. So if nothing is set, we do the faster, less invasive, passive scan.

We need to get the value of module param into a variable called `scantype`:

```
scantype = module.params['scantype']
```

The logic changes to accommodate two possible values now:

```
if scantype == 'active':
    try:
        scanid = zap.ascan.scan(target)
        while (int(zap.ascan.status(scanid)) < 100):
            time.sleep(5)
    except:
        module.fail_json(msg='Active Scan Failed')
else:
    try:
        while (int(zap.pscan.records_to_scan) > 0):
            time.sleep(2)
    except:
        module.fail_json(msg='Passive Scan Failed')
```

If `scantype` is set and the value is `active`, only then does it go ahead and do an active scan. This improvement makes our module more flexible:

```
Using the new and improved module in our playbook
- name: Testing OWASP ZAP Test Module
  connection: local
  hosts: localhost
  tasks:
  - name: Scan a website
    owasp_zap_test_module:
      host: "http://172.16.1.102:8080"
      target: "http://testphp.vulnweb.com"
      scantype: passive
    register: output
  - name: Print version
    debug:
      msg: "Scan Report: {{ output }}"
```

Using Ansible as a Python module

Using Ansible directly in your Python code is a powerful way of interacting with it. Please note that with Ansible 2.0 and newer, this is not the simplest of way of doing that.

Before we proceed we should let you know what the core Ansible team thinks about using the Python API directly

From http://docs.ansible.com/ansible/latest/dev_guide/developing_api.html

Please note that while we make this API available it is not intended for direct consumption, it is here for the support of the Ansible command line tools. We try not to make breaking changes but we reserve the right to do so at any time if it makes sense for the Ansible toolset.

The following documentation is provided for those that still want to use the API directly, but be mindful this is not something the Ansible team supports.

The following code is from the Ansible Developer Guide documentation: `http://docs.ansible.com/ansible/latest/dev_guide/developing_api.html`:

```
import json
from collections import namedtuple
from ansible.parsing.dataloader import DataLoader
from ansible.vars.manager import VariableManager
from ansible.inventory.manager import InventoryManager
from ansible.playbook.play import Play
from ansible.executor.task_queue_manager import TaskQueueManager
from ansible.plugins.callback import CallbackBase
```

Once all the initial work is done, this is how a task will be executed:

```
try</span>:
    tqm = TaskQueueManager(
              inventory=inventory,
              variable_manager=variable_manager,
              loader=loader,
              options=options,
              passwords=passwords,
              stdout_callback=results_callback,  # Use our custom callback
instead of the ``default`` callback plugin
          )
    result = tqm.run(play)
```

Before Ansible 2.0, the whole process was a lot easier. But this code doesn't work anymore:

```
import ansible.runner

runner = ansible.runner.Runner(
   module_name='ping',
   module_args='',
   pattern='web*',
   forks=10
)
datastructure = runner.run()
```

Summary

In this chapter, we created a working Ansible module for security automation. We started by creating a sort of hello world module that didn't do much, but helped us understand the layout of what a module file could look like. We followed the instructions as per the Ansible developer guide on how to set up an environment for being able to do module development. We articulated our requirement from the module and picked OWASP ZAP as a possible candidate for creating the module.

Using the training wheels, such as the template from the developer docks, we created the module and we saw how to use it using Ansible CLI or a playbook. We added a couple more options to the original code so that we could make the module more useful and flexible. Now we have an OWASP ZAP Ansible module that can connect to any hosted OWASP ZAP that allows access with the API key and executes a passive or active scan on the target.

This is the penultimate chapter of the book. In the next chapter, we will look at additional references, security of our secrets using Ansible Vault, and some world-class references of security automation already enabled using Ansible.


Madhu Akula, Akash Mahajan
Security Automation with Ansible 2
Leverage Ansible 2 to automate complex security tasks like application security, network security, and malware analysis
Packt