1. INSPEC DEVELOPER COURSE

A hands-on InSpec developer course.

After attending this course the student will be able to:

- Describe the InSpec framework and its capabilities
- Describe the architecture of an InSpec profile
- Build an InSpec profile to transform security policy into automated security testing
- Run an InSpec profile against a component of an application stack
- View and analyze InSpec results
- Report results

We will be spending most of the course hands-on working with the tools and in the Unix command line, so as to grow an understanding of how InSpec actually works.

Don't fixate on the tools used, nor the specific use cases we develop in the course, instead focus on learning the how and why. How and why is far more important. This course is not about tools although we'll be using them. You'll spend far more time writing code. (Or at least cutting-and-pasting code.)

2. AUTHOR

- Aaron Lippold alippold@mitre.org
- Mohamed El-Sharkawi melsharkawi@mitre.org

3. THANK YOU TO

4. TABLE OF CONTENTS

↥ back to top

5. ABOUT INSPEC

- InSpec is an open-source, community-developed compliance validation framework
- Provides a mechanism for defining machine-readable compliance and security requirements
- Easy to create, validate, and read content
- Cross-platform (Windows, Linux, Mac)
- Agnostic to other DevOps tools and techniques
- Integrates into multiple CM tools

5.1. ORCHESTRATION, CONFIGURATION MANAGEMENT, VALIDATION TO DEPLOYMENT

InSpec operates with most orchestration and CM tools found in the DevOps pipeline implementations

5.2. AUTOMATING SECURITY VALIDATION USING INSPEC

5.3. PROCESSING INSPEC RESULTS

↥ back to top

6. COURSE OVERVIEW

6.1. INSPEC PROFILE STRUCTURE

6.2. INSPEC CONTROLS STRUCTURE

6.3. INSPEC RESULTS

6.3.1. FAILURE

6.3.2. PASS

6.3.3. MULTIPLE CONTROLS

6.4. TOOLING AND REPORTING

↥ back to top

7. COURSE ENVIRONMENT SETUP

7.1. DOWNLOAD AND INSTALL VIRTUALBOX

https://www.virtualbox.org/wiki/Downloads

7.2. DOWNLOAD AND INSTALL VAGRANT

https://www.vagrantup.com/downloads.html

7.3. CLONE OR DOWNLOAD-UNZIP THIS COURSE

↥ back to top

7.4. SETUP ENVIRONMENTS

Start by creating a working directory. We recommend ~/learn-inspec.

`mkdir ~/learn-inspec` [or from Windows cmd prompt: `mkdir Desktop/learn-inspec`]

Next, move to your working directory.

`cd ~/learn-inspec` [or from Windows cmd

prompt: `cd Desktop/learn-inspec`]

7.4.1. RUN VAGRANT TO INSTALL THE VIRTUAL ENVIRONMENT

Navigate to the `InSpec 102 Dev` folder and run the following command:

```
$ vagrant up
```

Wait for vagrant to finish standing up the virtual environments.

7.4.2. SETUP NETWORK IN VIRTUALBOX

Open VirtualBox and shut down the 3 vm's that were created `workstation`, `target`, `target-centos6`.

Open Preference settings for VirtualBox (**not** the settings for the VM's)

- Go to the network tab
- From there click the + symbol to add a new NatNetwork
- Once you do that your preferences should look like this below: Alt text

Click ok to save the settings.

The following step you will repeat for the 3 vm's `workstation`, `target`, `target-centos6`.

- Select the virtual machine
- Click on settings for the virtual machine
- Navigate to the network tab
- For `Attached to:` Select `NatNetwork`
- For `Name` make sure the same NatNetwork is selected for all the virtual machines
- Click on Advanced dropdown and For `Promiscuous Mode:` make sure to select `Allow VMs`
- Once you do these steps your preferences should look like this below:

look like this below:

- Next you need to Select the Shared Folders
- Click the + symbol to add a new Shared Folder

- For Folder Path select the dropdown and select `Other`, navigate to your `~/learn-inspec` folder and select that 

- Select the checkbox for Auto-mount 

- Click ok to confirm the shared folder.

- Once you do these steps your preferences should look like this below: 

- Once more click ok to confirm and save the settings

Once the above operations are completed you can startup up the `workstation` and `target` vm's since we will start with those.

Workstation Credentials:

u: vagrant

p: vagrant

Target Credentials:

u: root

The workstation can connect to the target by the target's ip, perform an `ifconfig` on the target to get it's ip. Run curl TARGET_IP and you see that NGINX is running.

```
$ curl TARGET_IP

<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
```

↥ back to top

8. STUDYING AN INSPEC PROFILE

Let's start by creating a profile that will contain NGINX tests.

Start by moving to the /root directory.

```
$ cd ~
```

Next, create an InSpec profile named my_nginx.

```
inspec init profile my_nginx
```

The terminal output should look like the following:

```
$ inspec init profile my_nginx
Create new profile at /root/my_nginx
 * Create directory controls
 * Create file controls/example.rb
 * Create file inspec.yml
 * Create directory libraries
 * Create file README.md
```

↥ back to top

8.1. UNDERSTANDING THE PROFILE STRUCTURE

Let's take a look at how the profile is structured. We'll start with how a profile's files are structured and then move to what makes up an InSpec control.

First, run `tree` to see what's in the `my_nginx` profile.

```
$ tree my_nginx
      my_nginx
       ├── README.md
       ├── controls
       │   └── example.rb
       ├── inspec.yml
       └── libraries

      2 directories, 3 files
```

Here's the role of each component.

- `README.md` provides documentation about the profile, including what it covers and how to run it.
- The `controls` directory contains files which implement the InSpec tests.
- `inspec.yml` provides metadata, or information, about the profile. Metadata includes the profile's description, author, copyright, and version.
- The `libraries` directory contains resource extensions. A resource extension enables you to define your own resource types. You won't work with resource extensions in this module.

↥ back to top

8.2. UNDERSTAND A CONTROL'S STRUCTURE

Let's take a look at the default control file, `controls/example.rb`.

```
# encoding: utf-8
# copyright: 2018, The Authors

title 'sample section'

# you can also use plain tests
describe file('/tmp') do
  it { should be_directory }
end

# you add controls here
control 'tmp-1.0' do                        # A unique ID for
  impact 0.7                                # The criticality,
  title 'Create /tmp directory'             # A human-readable
  desc 'An optional description...'
```

This example shows two tests. Both tests check for the existence of the `/tmp` directory. The second test provides additional information about the test. Let's break down each component.

- `control` (line 12) is followed by the control's name. Each control in a profile has a unique name.
- `impact` (line 13) measures the relative importance of the test and must be a value between 0.0 and 1.0.
- `title` (line 14) defines the control's purpose.
- `desc` (line 15) provides a more complete description of what the control checks for.
- `describe` (lines 16 — 18) defines the test. Here, the test checks for the existence of the `/tmp` directory.

In Ruby, the `do` and `end` keywords define a block. An InSpec control always contains at least one `describe` block. However, a control can contain many `describe` blocks.

↥ back to top

8.3. UNDERSTAND A DESCRIBE BLOCK'S STRUCTURE

As with many test frameworks, InSpec code resembles natural language. Here's the format of a describe block.

```
describe <entity> do
  it { <expectation> }
end
```

An InSpec test has two main components: the subject to examine and the subject's expected state. Here, `<entity>` is the subject you want to examine, for example, a package name, service, file, or network port. The `<expectation>` part specifies the desired result or expected state, for example, that a port should be open (or perhaps should not be open.)

Let's take a closer look at the `describe` block in the example.

```
describe file('/tmp') do
  it { should be_directory }
end
```

Because InSpec resembles human-readable language, you might read this test as "/tmp should be a directory." Let's break down each component.

8.3.1. FILE

file is an InSpec resource. If you're familiar with Chef, you know that a resource configures one part of the system. InSpec resources are similar. For example, the InSpec file resource tests for file attributes, including a file's owner, mode, and permissions. The example examines the /tmp directory.

8.3.2. IT

The `it` statement validates one of your resource's features. A `describe` block contains one or more `it` statements. `it` enables you to test the resource itself. You'll also see `its`, which describes some feature of the resource, such as its mode or owner. You'll see examples of both `it` and `its` shortly.

8.3.3. SHOULD

`should` describes the expectation. `should` asserts that the condition that follows *should* be true. Alternatively, `should_not` asserts that the condition that follows *should not* be true. You'll see examples of both shortly.

8.3.4. BE_DIRECTORY

`be_directory` is an example of a matcher. A matcher compares a resource's actual value to its expected value. InSpec provides several predefined matchers. The `file` resource provides the `be_directory` matcher.

↥ back to top

9. EXPLORING THE INSPEC SHELL

Before we test our NGINX configuration, let's plan which resources and matchers we'll need.

When writing InSpec code, many resources are available to you.

- You can explore the InSpec documentation to see which resources and matchers are available.
- You can examine the source code to see what's available. For example, you can see how file and other InSpec resources are implemented.
- You can also use examples, such as profiles provided on Chef Supermarket, as a guide.

There's also InSpec shell, which enables you to explore InSpec interactively. In this part, you'll use the InSpec shell to discover which resources you can use to test your NGINX configuration.

You're not required to use InSpec shell to develop your profiles. Some users find the InSpec shell to be a useful way to get immediate feedback and explore what's available. You can also use InSpec shell to debug your profiles.

↥ back to top

9.1. ENTER THE SHELL

Run `inspec shell` to enter the interactive session.

```
$ inspec shell
Welcome to the interactive InSpec Shell
To find out how to use it, type: help

You are currently running on:

     Name:       ubuntu
     Families:   debian, linux, unix
     Release:    16.04
     Arch:       x86_64
```

Run `help` to see what commands are available.

```
inspec> help
    You are currently running on:

        Name:      ubuntu
        Families:  debian, linux, unix
        Release:   16.04
        Arch:      x86_64

    Available commands:

        `[resource]` - run resource on target machine
        `help resources` - show all available resources that c
        `help [resource]` - information about a specific resou
        `help matchers` - show information about common matche
        `exit` - exit the InSpec shell
```

Run `help resources` to see which resources are available.

```
inspec> help resources
          - aide_conf
          - apache
          - apache_conf
          - apt
          - audit_policy
          - auditd
          - auditd_conf
          ...
          - file
          ...
          - xml
          - yaml
          - yum
          - yumrepo
```

You see `file` and other resources listed.

↥ back to top

9.2. EXPLORE THE FILE RESOURCE

Earlier, you saw this `describe` block.

```
describe file('/tmp') do                    # The actual test
  it { should be_directory }
end
```

Let's run a few commands from the InSpec shell to see how the `file` resource works.

InSpec is built on the Ruby programming language. InSpec matchers are implemented as Ruby methods. Run this command to list which methods are available to the `file` resource.

```
inspec> file('/tmp').class.superclass.instance_methods(false).
        => [:allowed?,
         :basename,
         :block_device?,
         :character_device?,
         :contain,
         :content,
         :directory?,
         ...
         :sticky,
         :sticky?,
         :suid,
         :symlink?,
         :to_s,
         :type,
```

You can use the arrow or Page Up and Page Down keys to scroll through the list. When you're done, press `Q`.

`InSpec shell is based on a tool called pry. If you're not familiar with pry or other REPL tools, later you can check out pry to learn more.`

As an example, call the `file.directory?` method.

```
inspec> file('/tmp').directory?
        => true
```

You see that the `/tmp` directory exists on your workstation container.

InSpec transforms resource methods to matchers. For example, the `file.directory?` method becomes the `be_directory` matcher. The `file.readable?` method becomes the `be_readable` matcher.

The InSpec shell understands the structure of blocks. This enables you to run mutiline code. As an example, run the entire `describe` block like this.

```
inspec> describe file('/tmp') do
inspec>   it { should be_directory }
inspec> end
        Profile: inspec-shell
        Version: (not specified)

          File /tmp
             ✔  should be directory

        Test Summary: 1 successful, 0 failures, 0 skipped
```

In practice, you don't typically run controls interactively, but it's a great way to test out your ideas.

```
A Ruby method that ends in ?, such as directory? is known as a

A predicate method typically returns a value that can be evalu
```

↥ back to top

9.3. EXPLORE THE NGINX RESOURCE

Now's a good time to define the requirements for our NGINX configuration. Let's say that you require:

1. NGINX version 1.10.3 or later.
2. the following NGINX modules to be installed:
   - `http_ssl`
   - `stream_ssl`
   - `mail_ssl`
3. the NGINX configuration file, `/etc/nginx/nginx.conf`, to:
   - be owned by the `root` user and group.
   - not be readable, writeable, or executable by others.

Let's see what resources are available to help define these requirements as InSpec controls.

Run `help resources` a second time. Notice InSpec provides two built-in resources to support NGINX – `nginx` and `nginx_conf`.

```
inspec> help resources
          - aide_conf
          - apache
          - apache_conf
          - apt
          ...
          - nginx
          - nginx_conf
          ...
          - xml
          - yaml
          - yum
          - yumrepo
          - zfs_dataset
          - zfs_pool
```

Run `nginx.methods`. You see the `version` and `modules` methods. You'll use these methods to define the first two requirements.

```
inspec> nginx.class.superclass.instance_methods(false).sort
        => [:bin_dir,
         :compiler_info,
         :error_log_path,
         :http_client_body_temp_path,
         :http_fastcgi_temp_path,
         :http_log_path,
         :http_proxy_temp_path,
         :http_scgi_temp_path,
         :http_uwsgi_temp_path,
         :lock_path,
         :modules,
         :modules_path,
         :openssl_version,
         ...
```

Run `nginx.version` to see what result you get.

```
inspec> nginx.version
        NoMethodError: undefined method `[]' for nil:NilClass
        from /opt/inspec/embedded/lib/ruby/gems/2.4.0/gems/ins
```

Notice the error. This tells us that NGINX is not installed. Recall that you're working on your workstation container environment, which does not have NGINX installed. Run the following package resource to verify.

```
inspec> package('nginx').installed?
        => false
```

Although you've discovered the methods you need – `version` and `modules` – let's run InSpec shell commands against the target that does have NGINX installed to see what results we find. To do so, first start by exiting your InSpec shell session.

```
inspec> exit
```

Run `inspec shell` a second time. This time, provide the `-t` argument to connect the shell session to the target container. This is similar to how you ran `inspec exec` in the Try InSpec module to scan the target from the workstation.

```
$ inspec shell -t ssh://TARGET_USERNAME:TARGET_PASSWORD@TARGET
  Welcome to the interactive InSpec Shell
  To find out how to use it, type: help

  You are currently running on:

      Name:       ubuntu
      Families:   debian, linux, unix
      Release:    16.04
      Arch:       x86_64
```

Remember that the target does not have the InSpec CLI installed on it. Your shell session exists on the workstation container; InSpec routes commands to the target instance over SSH.

Run the `package` resource a second time, this time on the target container.

```
inspec> package('nginx').installed?
        => true
```

You see that NGINX is installed. Now run `nginx.version`.

```
inspec> nginx.version
        => "1.10.3"
```

You see that version 1.10.3 is installed. To complete the example, run `nginx.modules` to list the installed NGINX modules.

```
inspec> nginx.modules
        => ["http_ssl",
         "http_stub_status",
         "http_realip",
         "http_auth_request",
         "http_addition",
         "http_dav",
         "http_geoip",
         "http_gunzip",
         "http_gzip_static",
         "http_image_filter",
         "http_v2",
         "http_sub",
         "http_xslt",
         "stream_ssl",
```

You see that the required modules, `http_ssl`, `stream_ssl`, and `mail_ssl`, are installed.

The nginx_conf resource examines the contents of the NGINX configuration file, `/etc/nginx/nginx.conf`.

Recall that the third requirement is to check whether the NGINX configuration file is owned by `root` and is not readable, writeable, or executable by others. Because we want to test attributes of the file itself, and not its contents, you'll use the `file` resource.

You saw earlier how the `file` resource provides the `readable`, `writeable`, and `executable` methods. You would also see that the `file` resource provides the `owned_by` and `grouped_into` methods.

```
inspec> file('/tmp').class.superclass.instance_methods(false).
        => [:allowed?,
         :directory?,
         :executable?,
         :exist?,
         :file,
         :file?,
         :file_version,
         :gid,
         :group,
         :grouped_into?,
         ...
         :owned_by?,
         ...
         :readable?,
```

These 5 `file` methods – `grouped_into`, `executable`, `owned_by`, `readable` and `writeable` – provide everything we need for the third requirement.

Exit the InSpec shell session.

```
inspec> exit
```

↥ back to top

9.4. WRITE THE INSPEC CONTROLS

Now that you understand which methods map to each requirement, you're ready to write InSpec controls.

To review, recall that you require:

1. NGINX version 1.10.3 or later.
2. the following NGINX modules to be installed:
   - `http_ssl`
   - `stream_ssl`
   - `mail_ssl`
3. the NGINX configuration file, `/etc/nginx/nginx.conf`, to:
   - be owned by the `root` user and group.
   - not be readable, writeable, or executable by others.

The first requirement is for the NGINX version to be 1.10.3 or later. To check this, you use the `cmp` matcher. Replace the contents of `/root/my_nginx/controls/example.rb` with this.

```
control 'nginx-version' do
  impact 1.0
  title 'NGINX version'
  desc 'The required version of NGINX should be installed.'
  describe nginx do
    its('version') { should cmp >= '1.10.3' }
  end
end
```

The test has an impact of 1.0, meaning it is most critical. A failure might indicate to the team that this

issue should be resolved as soon as possible, likely by upgrading NGINX to a newer version. The test compares `nginx.version` against version 1.10.3.

`cmp` is one of InSpec's built-in matchers. `cmp` understands version numbers and can use the operators ==, <, <=, >=, and >. `cmp` compares versions by each segment, not as a string. For example, "7.4" is less than than "7.30".

Next, run `inspec exec` to execute the profile on the remote target.

```
$ inspec exec /root/my_nginx -t ssh://TARGET_USERNAME:TARGET_P

  Profile: InSpec Profile (my_nginx)
  Version: 0.1.0
  Target:  ssh://TARGET_USERNAME@TARGET_IP:22

    ✔  nginx-version: NGINX version
       ✔  Nginx Environment version should cmp >= "1.10.3"


  Profile Summary: 1 successful control, 0 control failures, 0
  Test Summary: 1 successful, 0 failures, 0 skipped
```

You see that the test passes.

The second requirement verifies that these modules are installed.

- http_ssl
- stream_ssl
- mail_ssl

Modify your control file like this.

```
...

control 'nginx-modules' do
  impact 1.0
  title 'NGINX version'
  desc 'The required NGINX modules should be installed.'
  describe nginx do
    its('modules') { should include 'http_ssl' }
    its('modules') { should include 'stream_ssl' }
    its('modules') { should include 'mail_ssl' }
  end
end
```

The second control resembles the first; however, this version uses multiple `its` statements and the `nginx.modules` method. The `nginx.modules`

method returns a list; the built-in `include` matcher verifies whether a value belongs to a given list.

Run `inspec exec` on the target.

```
$ inspec exec /root/my_nginx -t ssh://TARGET_USERNAME:TARGET_P

  Profile: InSpec Profile (my_nginx)
  Version: 0.1.0
  Target:  ssh://TARGET_USERNAME@TARGET_IP:22

    ✔  nginx-version: NGINX version
       ✔  Nginx Environment version should cmp >= "1.10.3"
    ✔  nginx-modules: NGINX version
       ✔  Nginx Environment modules should include "http_ssl"
       ✔  Nginx Environment modules should include "stream_ssl
       ✔  Nginx Environment modules should include "mail_ssl"


  Profile Summary: 2 successful controls, 0 control failures,
```

This time, both controls pass.

The third requirement verifies that the NGINX configuration file, `/etc/nginx/nginx.conf`:

- is owned by the root user and group.
- is not be readable, writeable, or executable by others.

Modify your control file like this.

```
...
control 'nginx-conf' do
  impact 1.0
  title 'NGINX configuration'
  desc 'The NGINX config file should owned by root, be writabl
  describe file('/etc/nginx/nginx.conf') do
    it { should be_owned_by 'root' }
    it { should be_grouped_into 'root' }
    it { should_not be_readable.by('others') }
    it { should_not be_writable.by('others') }
    it { should_not be_executable.by('others') }
  end
end
```

The third control uses the `file` resource. The first 2 tests use `should` to verify the `root` owner and group. The last 3 tests use `should_not` to verify that the file is not readable, writable, or executable by others.

Run `inspec exec` on the target.

```
$ inspec exec /root/my_nginx -t ssh://TARGET_USERNAME:TARGET_P

  Profile: InSpec Profile (my_nginx)
  Version: 0.1.0
  Target:  ssh://TARGET_USERNAME@TARGET_IP:22

    ...
    ×  nginx-conf: NGINX configuration (1 failed)
       ✔  File /etc/nginx/nginx.conf should be owned by "root"
       ✔  File /etc/nginx/nginx.conf should be grouped into "r
       ×  File /etc/nginx/nginx.conf should not be readable by
       expected File /etc/nginx/nginx.conf not to be readable
       ✔  File /etc/nginx/nginx.conf should not be writable by
       ✔  File /etc/nginx/nginx.conf should not be executable
```

This time you see a failure. You discover that `/etc/nginx/nginx.conf` is potentially readable by others. Because this control also has an impact of 1.0, your team may need to investigate further.

Remember, the first step, detect, is where you identify where the problems are so that you can accurately assess risk and prioritize remediation actions. For the second step, correct, you can use Chef or some other continuous automation framework to correct compliance failures for you. You won't correct this issue in this module, but later you can check out the

Integrated Compliance with Chef track to learn more about how to correct compliance issues using Chef.

↥ back to top

9.5. REFACTOR THE CODE TO USE ATTRIBUTES

Your `my_nginx` profile is off to a great start. As your requirements evolve, you can add additional controls. You can also run this profile as often as you need to verify whether your systems remain in compliance.

Let's review the control file, `example.rb`.

```
control 'nginx-version' do
  impact 1.0
  title 'NGINX version'
  desc 'The required version of NGINX should be installed.'
  describe nginx do
    its('version') { should cmp >= '1.10.3' }
  end
end

control 'nginx-modules' do
  impact 1.0
  title 'NGINX version'
  desc 'The required NGINX modules should be installed.'
  describe nginx do
    its('modules') { should include 'http_ssl' }
```

```
control 'nginx-conf' do
  impact 1.0
  title 'NGINX configuration'
  desc 'The NGINX config file should owned by root, be writabl
  describe file('/etc/nginx/nginx.conf') do
    it { should be_owned_by 'root' }
    it { should be_grouped_into 'root' }
    it { should_not be_readable.by('others') }
    it { should_not be_writable.by('others') }
    it { should_not be_executable.by('others') }
  end
end
```

Although these rules do what you expect, imagine your control file contains dozens or hundreds of tests. As the data you check for, such as the version or which modules are installed, evolve, it can become tedious to locate and update your tests. You may also find that

you repeat the same data in across multiple control files.

One way to improve these tests is to use Attributes. Attributes enable you to separate the logic of your tests from the data your tests validate. Attribute files are typically expressed as a YAML file.

Profile Attributes exist in your profile's main directory within the `inspec.yml` for global Attributes to be used across your profile or in your `attributes` folder for custom Attributes. Start by creating this directory.

```
name: my_nginx
title: InSpec Profile
maintainer: The Authors
copyright: The Authors
copyright_email: you@example.com
```

```
license: Apache-2.0
summary: An InSpec Compliance Profile
version: 0.1.0
supports:
  platform: os

attributes:
  - name: user
    type: string
```

Example of adding a array object of servers:

```
attributes:
  - name: servers
    type: array
    default:
      - server1
      - server2
      - server3
```

To access an attribute you will use the attribute keyword. You can use this anywhere in your control code.

For example:

```
current_user = attribute('user')

control 'system-users' do
  describe attribute('user') do
    it { should eq 'bob' }
  end

  describe current_user do
    it { should eq attribute('user') }
  end
end
```

For sensitive data it is recommended to use a secrets YAML file located on the local machine to populate the values of attributes. A secrets file will always overwrite a attributes default value. To use the secrets file run

inspec exec and specify the path to that Yaml file using the --attrs attribute.

For example, a inspec.yml:

```
attributes:
  - name: username
    type: string
    required: true
  - name: password
    type: string
    required: true
```

The control:

```
control 'system-users' do
  impact 0.8
  desc '
    This test assures that the user "Bob" has a user installed
    specified password.
  '

  describe attribute('username') do
    it { should eq 'bob' }
  end

  describe attribute('password') do
    it { should eq 'secret' }
  end
end
```

To change your attributes for platform specific cases you can setup multiple --attrs files.

For example, a inspec.yml:

```
attributes:
  - name: users
    type: array
    required: true
```

A YAML file named windows.yml

```
users:
  - Administrator
  - Guest
  - Randy
```

A YAML file named linux.yml

```yaml
users:
  - root
  - shadow
  - rmadison
```

The control file:

```
control 'system-users' do
  impact 0.8
  desc 'Confirm the proper users are created on the system'

  describe users do
    its('usernames') { should eq attribute('users') }
  end
end
```

The following command runs the tests and applies the attributes specified:

```
$ inspec exec examples/profile-attribute --attrs examples/wind
$ inspec exec examples/profile-attribute --attrs examples/linu
```

↥ back to top

9.6. RUNNING BASELINE STRAIGHT FROM GITHUB/CHEF SUPERMARKET

In this module, we use NGINX for learning purposes. If you're interested in NGINX specifically, you may be interested in the MITRE nginx-baseline profile on GitHub. Alternatively, you may also check out the

DevSec Nginx Baseline profile on Chef Supermarket. These profiles implements many of the tests you wrote in this module.

To execute the GitHub profile on your target system, run this `inspec exec` command.

```
$ inspec exec https://github.com/dev-sec/nginx-baseline -t ssh://TARGET_USERNAME:TARGET_PASSWORD@T
```

To execute the Chef Supermarket profile on your target system, run this `inspec supermarket exec` command.

```
$ inspec supermarket exec dev-sec/nginx-baseline -t ssh://TARG
  [2018-05-03T03:07:51+00:00] WARN: URL target https://github.

  Profile: DevSec Nginx Baseline (nginx-baseline)
  Version: 2.1.0
  Target:  ssh://TARGET_USERNAME@TARGET_IP:22

    ...
    ×  nginx-02: Check NGINX config file owner, group and perm
       ...
       ×  File /etc/nginx/nginx.conf should not be readable by
       expected File /etc/nginx/nginx.conf not to be readable
       ...
       ↺  nginx-15: Content-Security-Policy
          ↺  Can't find file "/etc/nginx/conf.d/90.hardening.c
```

You see that many of the tests pass, while others fail and may require investigation.

You may want to extend the `nginx-baseline` with your own custom requirements. To do that, you might use what's called a *wrapper profile*. You can check out Create a custom InSpec profile for a more complete example.

↥ back to top

10. VIEWING AND ANALYZING RESULTS

InSpec allows you to output your test results to one or more reporters. You can configure the reporter(s) using either the --json-config option or the --reporter option. While you can configure multiple reporters to write to

different files, only one reporter can output to the screen(stdout).

```
$ inspec exec /root/my_nginx -t ssh://TARGET_USERNAME:TARGET_P
```

↥ back to top

10.1. SYNTAX

You can specify one or more reporters using the --reporter cli flag. You can also specify a output by appending a path separated by a colon.

Output json to screen.

```
inspec exec /root/my_nginx --reporter json
or
inspec exec /root/my_nginx --reporter json:-
```

Output yaml to screen

```
inspec exec /root/my_nginx --reporter yaml
or
inspec exec /root/my_nginx --reporter yaml:-
```

Output cli to screen and write json to a file.

```
inspec exec /root/my_nginx --reporter
    cli json:/tmp/output.json
```

Output nothing to screen and write junit and html to a file.

```
inspec exec /root/my_nginx --reporter
junit:/tmp/junit.xml
html:www/index.html
```

Output json to screen and write to a file. Write junit to a file.

```
inspec exec /root/my_nginx --reporter
json junit:/tmp/junit.xml | tee
out.json
```

If you wish to pass the profiles directly after specifying the reporters you will need to use the end of options flag --.

```
inspec exec --reporter json
junit:/tmp/junit.xml -- profile1
profile2
```

Output cli to screen and write json to a file.

```
{
    "reporter": {
        "cli" : {
            "stdout" : true
        },
        "json" : {
            "file" : "/tmp/output.json",
            "stdout" : false
        }
    }
}
```

↥ back to top

10.2. SUPPORTED REPORTERS

The following are the current supported reporters:

- cli
- json
- json-min
- yaml
- documentation
- junit
- progress
- json-rspec
- html

You can read more about InSpec Reporters on the documentation page.

↥ back to top

11. AUTOMATION TOOLS

Navigate to the web page for Heimdall Lite

Click on the button `Load Json` 

Click on the button `Browse` 

Navigate to your json output file that you saved from your previous step and select that file then click open.

This will allow you to view the InSpec results in the Heimdall viewer.

↥ back to top

12. CREATE BASIC PROFILE - DAY 2

12.1. DOWNLOAD STIG REQUIREMENTS HERE

Download the latest STIG Viewer located here STIG Viewer 

Download the `Red Hat 6 STIG - Ver 1, Rel 21` located here RHEL6 STIG Download 

↥ back to top

12.2. EXAMPLE CONTROL V-38437

Let's take a look at how we would write a the InSpec control for V-38437:

```
control "V-38437" do
  title "Automated file system mounting tools must not be enab
needed."
  desc  "All filesystems that are required for the successful
system should be explicitly listed in \"/etc/fstab\" by an adm
filesystems should not be arbitrarily introduced via the autom
```

```
    The \"autofs\" daemon mounts and unmounts filesystems, suc
directories shared via NFS, on demand. In addition, autofs can
handle removable media, and the default configuration provides
as \"/misc/cd\". However, this method of providing access to r
not common, so autofs can almost always be disabled if NFS is
if NFS is required, it is almost always possible to configure
statically by editing \"/etc/fstab\" rather than relying on th
  "
  impact 0.3
```

↥ back to top

13. USING WHAT YOU'VE LEARNED

Now you should be able to

- Describe the InSpec framework and its capabilities
- Describe the architecture of an InSpec profile
- Build an InSpec profile to transform security policy into automated security testing
- Run an InSpec profile against a component of an application stack
- View and analyze InSpec results
- Report results

You can contribute to existing profiles that can be found here:

https://github.com/mitre

Otherwise you can create your own profiles if they don't exist using the following security guidelines:

https://iase.disa.mil/stigs/Pages/a-z.aspx

https://www.cisecurity.org/cis-benchmarks/

↥ back to top

14. CLEANUP ENVIRONMENTS

If you're done with your vagrant boxes, run the following command to destroy them: `vagrant destroy -f`

↥ back to top

15. ADDITIONAL RESOURCES

15.1 SECURITY GUIDANCE

https://iase.disa.mil/stigs/Pages/a-z.aspx
https://www.cisecurity.org/cis-benchmarks/

15.2 INSPEC DOCUMENTATION

InSpec Docs
InSpec Profiles
InSpec Resources
InSpec Matchers
InSpec Shell
InSpec Reporters

15.3 ADDITIONAL TUTORIALS

What to Expect When You're InSpec'ing

Getting started with InSpec - The InSpec basics series

Windows infrastructure testing using InSpec – Part I

Windows infrastructure testing using InSpec and Profiles – Part II

15.4 MITRE INSPEC

MITRE InSpec Repositories