

# Barracuda WAF Automation Lab

## Infrastructure as Code

---

January 2020

Brett Wolmarans [bwolmarans@barracuda.com](mailto:bwolmarans@barracuda.com)

(818) 292-7981

# What do we hope to achieve?

Crawl

Walk

Run

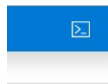
# Components

Azure Portal



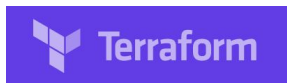
Interactive Public Cloud Portal

Azure Shell



Convenient “Shell in a Can”

Terraform



Infrastructure Automation Tool

Postman



API Automation Prototyping Tool

Ansible



Configuration Automation Tool

Barracuda WAF



A popular WAF

Ubuntu



A popular distro

Docker



A popular container ecosystem

# Contents

- Desired end state
  - What we hope to achieve in a declarative fashion
- Highly compressed Introduction to
  - Azure Portal + Shell
  - Terraform
  - Postman
  - Ansible
  - Barracuda REST API
- Details of the Terraform Files
- Details of the Postman playbook
- Details of the Ansible playbook
- Self-Paced, hands-on Labs for most of the above...

# Desired End State

- New, separate Azure resource group with vNet, vNics, Public IP's, and security groups
- BWAF Vm latest version, running in Azure resource group
- Ubuntu 16 VM running DVWA in a docker container
- WAF front-end Service with a security policy protecting the DVWA application running as a back-end server object

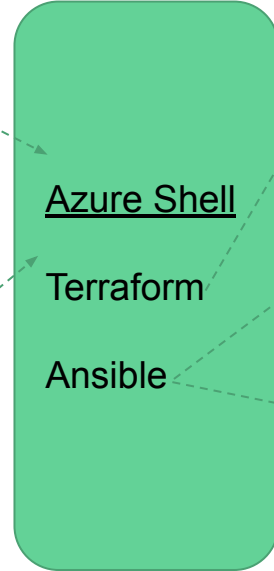
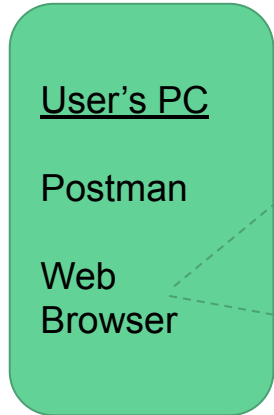
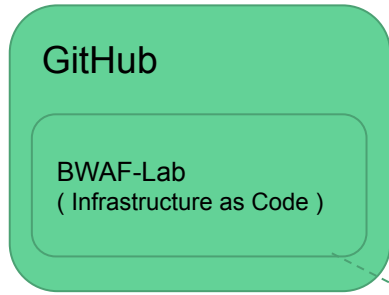
( See Diagram on next Slide )

How we get there:

- Terraform to build infrastructure ( infrastructure as code )
  - Our Terraform file is `bwaf-playground.tf`
- Postman to prototype and develop our configuration RESTful pipeline ( concept to code )
- Ansible to deploy our configuration as code ( code to configuration )
  - Our Ansible playbook is `bwaf-playbook.tf`

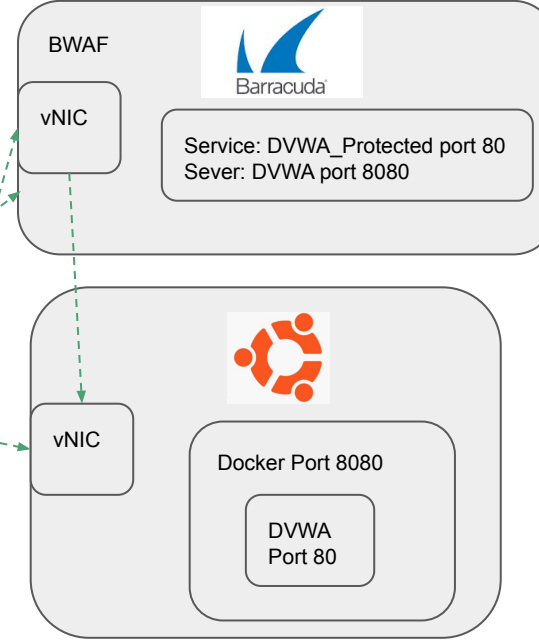
## Desired End State

**Automated, Repeatable,  
Idempotent, Declarative**



## Azure

### Resource Group





# Concepts Section

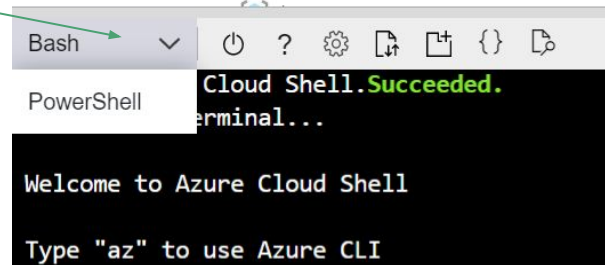


# Azure Cloud Shell

- Become familiar with Azure Cloud Shell
- It is both Powershell and BASH so hopefully you are at least a little bit familiar with a BASH shell. ( You may need to switch to BASH, it may default to Powershell for you )
- It times out after 20 minutes, but uses persistent storage\* so don't worry, you won't lose your work
- Ctrl-C to copy, Shift-Insert to paste or can use right-click
- Please do this command now, and paste output in Zoom Chat

```
az account list --output table
```

- A quick FYI on a concept we call Unique ID in this lab:
  - You will need a unique ID for your resource group
  - Your ID will be your first name followed by your phone number ( no spaces/hyphens/parentheses) for example my first name is Brett and my phone number is +1(818) 292-7981, so my ID will be brett8182927981



*\*only for \$home*

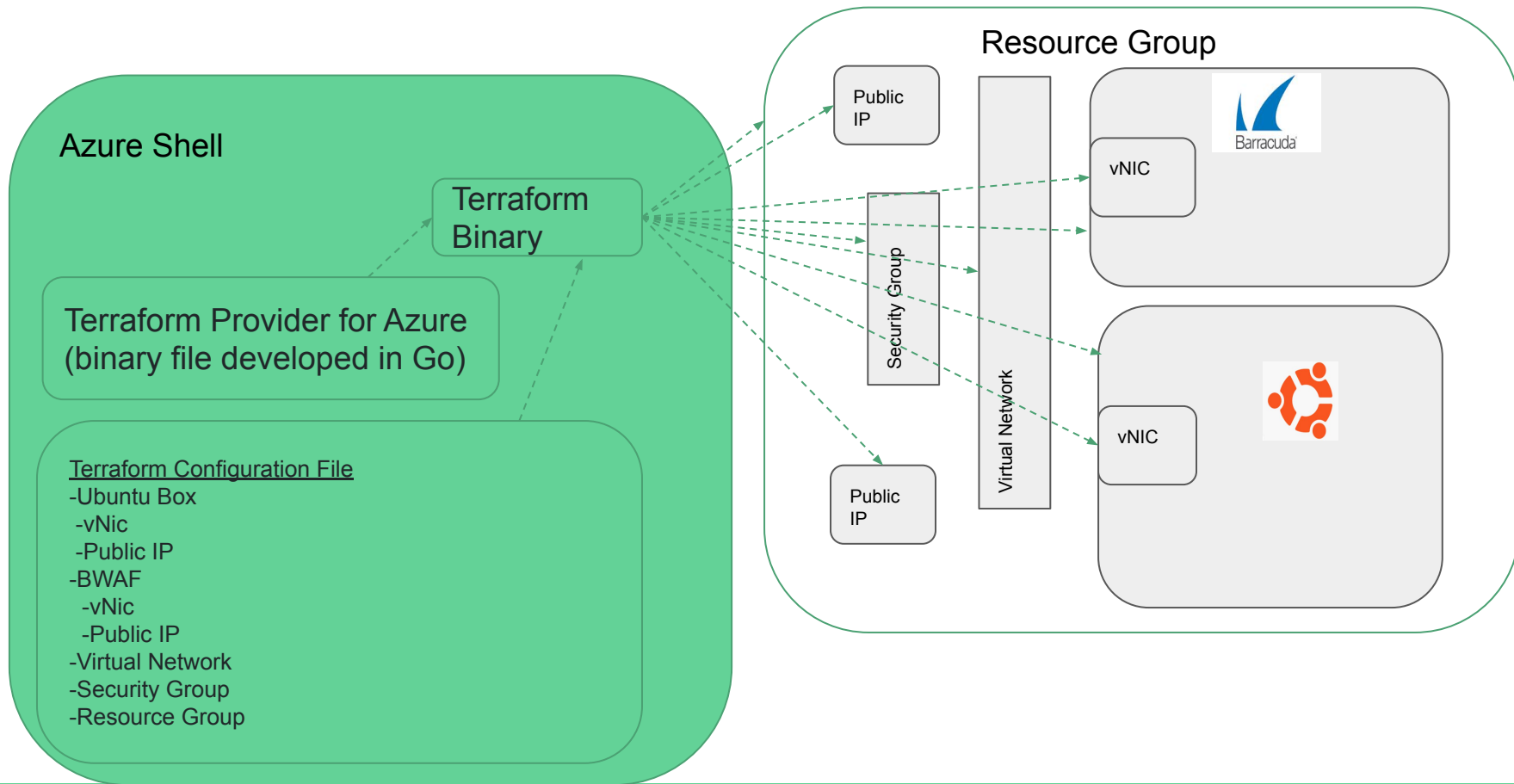
# Terraform in a Rather Small Nutshell



Terraform is an **Infrastructure** management tool.

- Infrastructure as Code
  - Infrastructure is described using a high-level configuration syntax.
  - This allows a blueprint of your datacenter to be versioned and treated as you would any other code.
  - Infrastructure can be shared and re-used.
- Execution Plans
  - Terraform has a "planning" step where it generates an execution plan.
  - The execution plan shows what Terraform will do when you call apply.
  - This lets you avoid any surprises when Terraform manipulates infrastructure.
- Resource Graph
  - Terraform builds a graph of all your resources, and parallelizes the creation and modification of any non-dependent resources.
  - This way, Terraform builds infrastructure as efficiently as possible, and operators get insight into dependencies in their infrastructure.
- Change Automation
  - Complex changesets can be applied to your infrastructure with minimal human interaction.
  - With execution plan and resource graph, you know exactly what Terraform will change and in what order, avoiding many possible human errors.

# Terraform visual block diagram



# Terraform Language File Syntax and Structure

Instead of YAML, Terraform uses **HCL** ( Hashicorp Configuration Language ) ( or JSON )

## Arguments, Blocks, and Expressions

---

The syntax of the Terraform language consists of only a few basic elements:

```
resource "aws_vpc" "main" {  
  cidr_block = var.base_cidr_block  
}  
  
<BLOCK TYPE> "<BLOCK LABEL>" "<BLOCK LABEL>" {  
  # Block body  
  <IDENTIFIER> = <EXPRESSION> # Argument  
}
```

- *Blocks* are containers for other content and usually represent the configuration of some kind of object, like a resource. Blocks have a *block type*, can have zero or more *labels*, and have a *body* that contains any number of arguments and nested blocks. Most of Terraform's features are controlled by top-level blocks in a configuration file.
- *Arguments* assign a value to a name. They appear within blocks.
- *Expressions* represent a value, either literally or by referencing and combining other values. They appear as values for arguments, or within other expressions.

# Why HCL?

JSON doesn't allow comments and HCL is more readable? YAML is harder?

```
variable "ami" {  
    description = "the AMI to use"  
}
```

This would be equivalent to the following json:

```
{  
  "variable": {  
    "ami": {  
      "description": "the AMI to use"  
    }  
  }  
}
```

# Terraform Configuration File Structure



As mentioned already, but worth repeating, Terraform files use HCL ( Hashicorp Configuration Language ) ( or JSON, but HCL is easier to read ). One can think of it as YAYAML ( sorry )

## Arguments and Blocks

The Terraform language syntax is built around two key syntax constructs: arguments and blocks.

### Arguments

An *argument* assigns a value to a particular name:

```
image_id = "abc123"
```

### Blocks

A *block* is a container for other content:

```
resource "aws_instance" "example" {  
  ami = "abc123"  
  
  network_interface {  
    # ...  
  }  
}
```

Argument names, block type names, and the names of most Terraform-specific constructs like resources, input variables, etc. are all *identifiers*.

Identifiers can contain letters, digits, underscores ( `_` ), and hyphens ( `-` ). The first character of an identifier must not be a digit, to avoid ambiguity with literal numbers.

`#` begins a single-line comment, ending at the end of the line.

# Terraform Configuration File Snippet



```
# Create virtual network
```

```
resource "azurerm_virtual_network" "myterraformnetwork" {  
    name                        = "bwaf_tf_vnet"  
    address_space              = ["10.0.0.0/16"]  
    location                   = "eastus"  
    resource_group_name = azurerm_resource_group.mytf.name  
}
```



# RESTful APIs in a very small nutshell

APIs for servers or network devices are not new. In the old days, we used to use Expect ( an extension to TCL ) to configure network devices over serial terminal servers, and modems, then eventually over telnet, and then, if you were highly sophisticated and in a secure environment, SSH, but that would have been for government work that had high standards of secrecy. There are of course other API type interfaces like SNMP, and of course, API's for lower-level languages.

Fortunately ( or unfortunately, depending on your perspective ) things have changed, and the standard for automation is a RESTful interface as a best practice API pattern. If you're new to API's, at least you don't have to get good at screen scraping with Expect!

REST stands for Representational state transfer

RESTful Web services allow the requesting systems to access and manipulate textual representations of Web Resources

Note that you will see RESTful API's often also referred to as REST API's as well, both terms are in common use in the industry.



# Barracuda WAF RESTful API



The automation interface for the Barracuda WAF is a proper RESTful API. It a JSON-RPC request pattern corresponding to field values in the configuration database of the BWAF.

The documentation root can be found here:

<https://campus.barracuda.com/product/webapplicationfirewall/doc/45024857/rest-api/>

We are now on version 3.1, and the examples use an auto documentation method called Swagger which is (among other things) a best practice to automatically document an API, and let you click on buttons to automatically generate examples of correct syntax, this makes life a LOT easier.

But there are also good examples in the version 1 of the API, and it is recommended to try those examples as well. The examples provided use the “curl” command for making requests to the API.

Version 1 can be found here: <https://campus.barracuda.com/product/webapplicationfirewall/doc/73698476/rest-api-version-1-v1>

# Use Cases for the BWAF RESTful API



The BWAF REST API provides remote administration and configuration of the Barracuda Web Application Firewall.

The API provides an easier way to perform frequent tasks that might be time consuming to do individually using the web interface.

For example, using the API, you can:

- create a service
- add a server to the service
- create a security policy

So the use cases are automation, integration into an existing framework, to work with a Devops style pipeline, or simply to perform repetitive tasks, or even more simply, to remove human error.

These are all valid use cases for the API.

# Postman



Postman is a popular, GUI based, **prototyping** tool for automation, particularly RESTful services.

Postman is great for prototyping and developing a suite of RESTful calls for a devops pipeline.

Postman is not at this time suitable for production level configuration management.

## Design & Mock

Communicate the expected behavior of an API by simulating endpoints and their responses without having to set up a backend server.

[Learn More](#)

## API Client

Quickly and easily send REST, SOAP, and GraphQL requests directly within Postman.

[Learn More](#)

## Automated Testing

Automate manual tests and integrate them into your CI/CD pipeline to ensure that any code changes won't break the API in production.

[Learn More](#)

## A Postman Collection

Is a series of API calls.

A simple collection is shown here:

Login

Get services

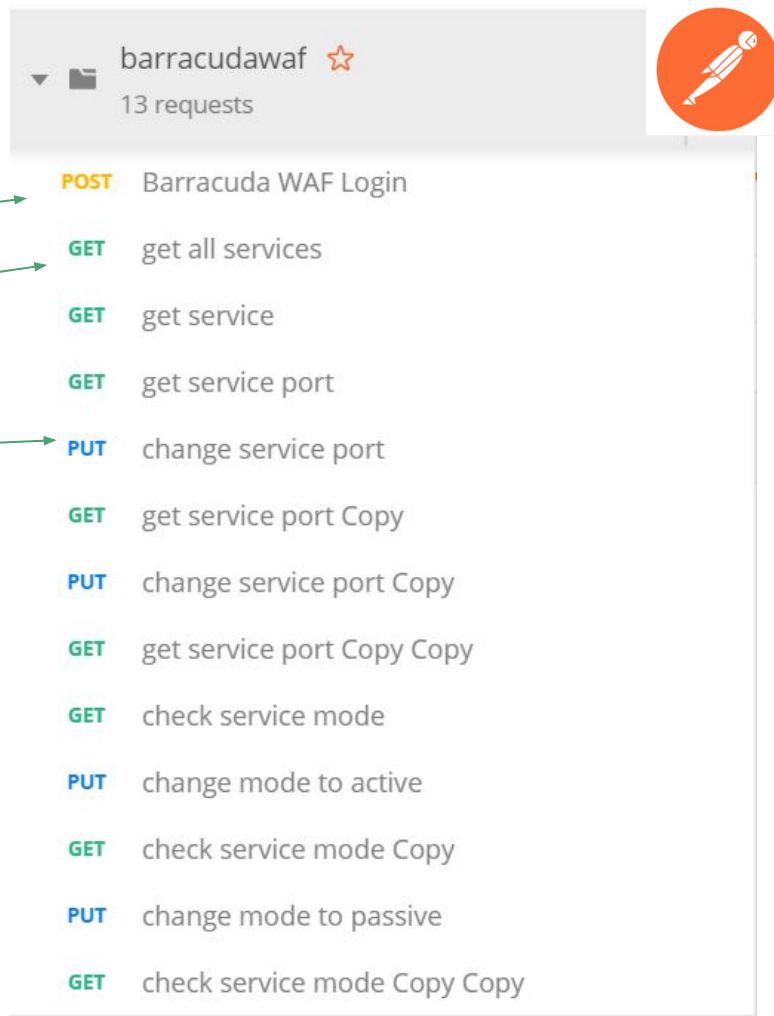
Change port

Etc

Each API call has an HTTP verb, and can be called interactively, see the response, to debug your RESTful actions. You can even add test cases.

You can set environment variables.

It's a very useful API prototyping tool.



**A Zoom in on one API call**, in this case, the BWA login, showing the returned authorization token.

This token can be induced into the environment and used automatically for authorizing the other API calls in the collection.



▶ Barracuda WAF Login

POST

http://{{url}}/restapi/v3.1/login

Params

Authorization

Headers (10)

Body ●

Pre-request Script

Tests ●

Query Params

	KEY	VALUE	DI
	Key	Value	D

Body

Cookies

Headers (4)

Test Results

Status: 200 OK

Time: :

Pretty

Raw

Preview

Visualize BETA

JSON ▼

≡

1

{

2

"token": "eyJ1c2VyIjoiYWRTaW4iLCJwYXNzd29yZCI6IjBmNjBjZDQzY2ZmMTlkNmVMOk

3

}

# Newman - postman collection as a script



Newman is a scriptable way to take a postman collection and run it in an automated fashion. Still dev/test though, not really devops production, although Postman is positioning their solutions as moving in the direction of a full configuration management alternative.

```
ec2-user@kali:~$ newman run barracudawaf.postman_collection.json -e barracudawaf.postman_environment.json --verbose -r cli,json
newman
```

barracudawaf

## → Barracuda WAF Login

POST http://40.85.190.9:8000/restapi/v3.1/login [200 OK, 279B, 124ms]

prepare	wait	dns-lookup	tcp-handshake	transfer-start	download	process
112ms	8ms	(cache)	1ms	98ms	15ms	778µs

## → get all services

GET http://40.85.190.9:8000/restapi/v3.1/services [200 OK, 7.06KB, 251ms]

prepare	wait	dns-lookup	tcp-handshake	transfer-start	download	process
2ms	895µs	(cache)	(cache)	246ms	3ms	169µs

## → get service

GET http://40.85.190.9:8000/restapi/v3.1/services/webgoat99 [404 Not Found, 344B,

prepare	wait	dns-lookup	tcp-handshake	transfer-start	download	process
1ms	543µs	(cache)	(cache)	135ms	2ms	54µs

## → get service port

GET http://40.85.190.9:8000/restapi/v3.1/services/webgoat99/servicePort [404 Not Found, 344B, 100ms]

	executed	failed
iterations	1	0
requests	13	0
test-scripts	19	0
prerequisite-scripts	13	0
assertions	5	5
total run duration: 2.9s		
total data received: 8.98KB (approx)		
average response time: 152ms [min: 100ms, max: 251ms, s.d.: 44ms]		
average first byte time: 146ms [min: 96ms, max: 246ms, s.d.: 46ms]		

# failure

1. AssertionError

detail

Port is 8080  
expected '{"msg":"Object \\'webgoat99\'  
system."', "token":"eyJwYXNkd29yZCI6ImI  
at assertion:0 in test-script  
inside "get service port"

# Ansible



Ansible is a **configuration** management tool.

This means you can use it to build configuration on existing infrastructure, for example, on a host, copying a folder of web pages from a repository, and specifying that folder to be the default web site, and starting the web server.

Ansible could in theory, be used to build infrastructure, and while some people do indeed use it for that, usually Terraform would be regarded as the first choice when it comes to building infrastructure from code, and Ansible would be regarded normally as a better choice for configuration management.

Ansible does not use HCL ... Ansible uses **YAML**

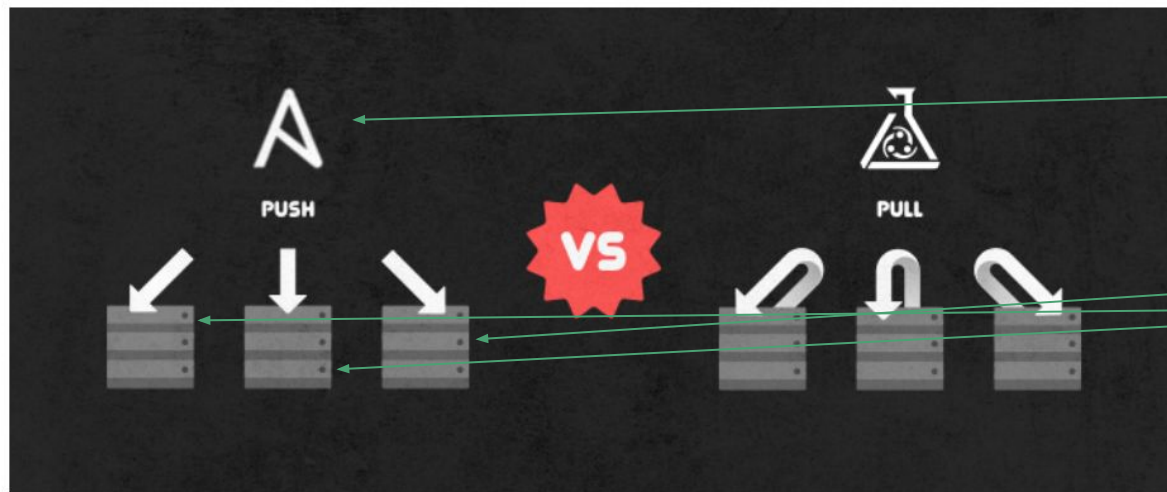
# Concepts



You'll often hear that **Ansible is agent-less and uses a push approach** (as opposed to pull).

In a nutshell, Chef or Puppet work by installing an agent on the hosts they manage. This agent is pulling changes from a master host, using their own channel (usually not SSH).

Ansible typically likes to use SSH. As we will see in the labs, we can use RESTful calls as an alternative to SSH.



There is a concept of a

control node

and

managed nodes.

Only the control node  
needs Ansible installed  
on it.



# MANAGE YOUR INVENTORY IN SIMPLE TEXT FILES



By default, Ansible represents what machines it manages using a very simple INI file that puts all of your managed machines in groups of your own choosing.

Here's what a plain text inventory file looks like:

```
[webservers]
www1.example.com
www2.example.com

[dbservers]
db0.example.com
db1.example.com
```

This familiar format is INI ( ini ) file format, just like in Windows.

Ansible playbooks use YAML, not INI formats.

And while YAML is supported for the inventory file, you will find that most examples and organizations use the INI format for the inventory file, and YAML for the playbook file, so keep that in mind.

# PLAYBOOKS: A SIMPLE+POWERFUL AUTOMATION LANGUAGE



Playbooks can finely orchestrate multiple slices of your infrastructure topology, with very detailed control over how many machines to tackle at a time. This is where Ansible starts to get most interesting.

Does everyone know what **YAML** stands for?

The following example is a simple playbook that perform two tasks: updates the `apt` cache and installs `vim` afterwards:

```
---
- hosts: all
  become: true
  tasks:
    - name: Update apt-cache
      apt: update_cache=yes

    - name: Install Vim
      apt: name=vim state=latest
```

# Self-Paced Labs Section

# Formatting and Readability

A note on formatting to help with readability:

Text in the **consolas font and highlighted green** means something you enter in the azure shell at the bash prompt, or do by clicking or typing in your browser.

Copy and paste is your friend, use highlight the text, use **ctrl-c to copy, and shift-insert or r-click** into the azure shell. *The command you need to copy may not entirely fit on one line, it may span two lines in this deck, so please be sure to highlight and copy the entire command, or it will not work.*

Text in **yellow highlight** is something you may wish to pay attention to

Text in **blue highlight** may be something to pay attention to as well, and also might be output from a command.

PS: Please don't print out these labs, they are not formatted nor intended for printing.

# Login to Azure, open the shell



Login to Azure and open your Azure shell

Create an SSH Key ( we will use this for our Ubuntu box )

Copy and Paste the following:

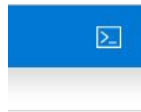
```
ssh-keygen -m PEM -t rsa -b 2048
```

Then just press [ENTER] accepting all the defaults ( **Note: if you have existing SSH keys please back them up FIRST** )

Click here to open the shell

Name	Type	Location
bwaf_tf_nic1	Network interface	East US
bwaf_tf_nic2	Network interface	East US

# Accept the terms for BWAF



Issue the following command ( if you all share the same subscription, then this may only need to be done once, but doesn't hurt if you all do it )

( copy and paste ( shift-insert to paste into azure shell ))

```
az vm image accept-terms --urn barracudanetworks:waf:hourly:latest
```

# Git the infrastructure as code



We're doing infrastructure as code, so we need the code in order to create the infrastructure. The code that will create our desired state is housed in the following repository, so you need to bring that code into your azure shell:

Then change into that directory, and look at a directory listing using ls -l

```
git clone https://github.com/bwolmarans/bwaf-lab.git
```

```
cd bwaf-lab
```

( copy and use shift-insert to paste in, or type in the command)

```
brett@Azure:~$ git clone https://github.com/bwolmarans/bwaf-lab.git
Cloning into 'bwaf-lab'...
remote: Enumerating objects: 45, done.
remote: Counting objects: 100% (45/45), done.
remote: Compressing objects: 100% (43/43), done.
remote: Total 45 (delta 18), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (45/45), done.
Checking connectivity... done.
```

# What the Github Repo code consists of



```
barracudawaf.postman_environment.json  
bwaf-dvwa.yaml  
bwaf-playbook.yaml  
bwaf-playground.tf  
myazure_rm.yml  
README.md  
resources_list.png
```

- Postman environment file
- The Web server DVWA ansible playbook
- The BWAF service Ansible playbook
- The Terraform Infrastructure as Code
- The Dynamic Inventory Azure plug-in
- A Readme
- A graphical image showing a snapshot of resources ( viewable @ <https://github.com/bwolmarans/bwaf-lab> )
-



# Terraform Section

# The Terraform Configuration File

**Examine the Terraform Configuration file**, spend 5 minutes on this part.

Compare it to what you know about Terraform configuration files.

The file name is bwaf-playground.tf

# Terraform Init

Issue the following commands to initialize terraform, which in this case means, among other things, one major thing this does is it downloads the azure provider and the “random” provider, which again, are binary code modules.

```
terraform init
```

```
brett@Azure:~/bwaf-lab$ terraform init
```

```
Initializing the backend...
```

```
Initializing provider plugins...
```

- Checking for available provider plugins...
- Downloading plugin for provider "random" (hashicorp/random) 2.2.1...
- Downloading plugin for provider "azurerm" (hashicorp/azurerm) 1.41.0...

```
The following providers do not have any version constraints in configuration,  
so the latest version was installed.
```

```
To prevent automatic upgrades to new major versions that may contain breaking  
changes, it is recommended to add version = "..." constraints to the  
corresponding provider blocks in configuration, with the constraint strings  
suggested below.
```

```
* provider.azurerm: version = "~> 1.41"  
* provider.random: version = "~> 2.2"
```

```
Terraform has been successfully initialized!
```

```
You may now begin working with Terraform. Try running "terraform plan" to see  
any changes that are required for your infrastructure. All Terraform commands  
should now work.
```

```
If you ever set or change modules or backend configuration for Terraform,  
rerun this command to reinitialize your working directory. If you forget, other  
commands will detect it and remind you to do so if necessary.
```

```
brett@Azure:~/bwaf-lab$
```

# Terraform Plan

Issue the following commands to see changes required for your infrastructure:

```
terraform plan
```

Examine the output of that command:

```
brett@Azure:~/bwaf-lab$ terraform apply
var.rg_name
  Your resource group requires a unique name if you are sharing a subscrip
182927981

Enter a value: brett8182927981
```

- How many resources are going to be created?
- Are all of those resources accurately shown on the (oversimplified) diagram we saw earlier, or did I forget to add some?

|

# Terraform Apply

Issue the following commands to create your infrastructure:

```
terraform apply
```

Answer “yes” when asked

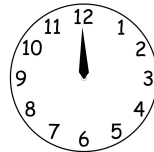
```
Do you want to perform these actions?  
Terraform will perform the actions described above.  
Only 'yes' will be accepted to approve.
```

```
Enter a value: yes
```

```
brett@Azure:~/bwaf-lab$ terraform apply  
var.rg_name  
Your resource group requires a unique name if you are sharing a subscrip  
182927981  
  
Enter a value: brett8182927981
```

```
azure_rm_virtual_machine.myterraformbwaf: Still creating... [13m11s elapsed]  
azure_rm_virtual_machine.myterraformbwaf: Still creating... [13m21s elapsed]  
azure_rm_virtual_machine.myterraformbwaf: Still creating... [13m31s elapsed]  
azure_rm_virtual_machine.myterraformbwaf: Creation complete after 13m38s [id=/subscriptions/182927981/resourceGroups/brett8182927981/providers/Microsoft.Compute/virtualMachines/azure_rm_virtual_machine.myterraformbwaf]  
  
Apply complete! Resources: 12 added, 0 changed, 6 destroyed.  
  
Outputs:  
  
public_ip_address =  
brett@Azure:~/bwaf-lab$
```

About 13.5 minutes



# Examine the Infrastructure

1. In the Azure Portal ( above the shell) find the resource group you have created, using your unique ID
2. Within the resource group, find the virtual machines, the vnet, the vnics, the security groups, and so on.
3. Open up the terraform configuration file ( use your bash skills for this, but don't accidentally change it with an editor ) , or scroll up in your Azure shell to see the output of Terraform Plan, and compare it to what has been created.
4. Login to the BWAf ( how did you know what password to use? Interesting side topic. ) and do not change anything, but notice it has not been configured. There are no services. That is because we are going to use Ansible for configuration management, not Terraform.

# CURL Section



# Hands-on with the Barracuda WAF RESTful API



Refer to the BWAFF API documentation referenced earlier in this presentation.

Let's **try some VERY simple examples**, starting off with CURL, then moving on to Postman.

The first thing to know is like most REST API's, you login once, and receive a Token, which you can then pass with each subsequent request. This way, you don't have to authenticate with a username and password for every request. It's a bit like a cookie, if you're familiar with that pattern.



# Logging in to the BWAF using Curl



You have to login to get your token first. If you try to do commands without a token, you will get this error message:

```
{"error":{"msg":"Please log in to get valid token","status":"401","type":null}}
```

You can do this right in your Azure shell.

Remember copy ( ctrl-C ) and paste ( shift-insert ) are your friends.

PS: sorry for the small font! The line below wraps around onto two lines.

So please select the entire line, all the way from the C in curl, to the last single quote '

Paste it into a scratch pad like notepad first, and replace <your BWAF public IP> with your actual BWAF public IP

```
curl http://<your BWAF public IP>:8000/restapi/v3.1/login -X POST -H Content-Type:application/json -d '{"username": "admin", "password": "Hello123456!"}'
```

```
{"token":"eyJwYXNzd29yZCI6ImI4ZGU4MzU0N2M2OTNjOTgxZTRmMjE3Yzk1NGYzYWMzliwidXNlciI6ImFk\nnbWluliwiZXQiOiIxNTc5Mzg5OTQ4In0=\n"}
```



If you see a token as a response to your curl, congratulations, you have made what may be your first API call to the BWAF 3.1 API. Copy this Token ( not the whole thing, just the scrambled text, everything between the last two quotation marks, even the last slash n ), and paste it into a scratchpad like notepad. Remember, **not the token on this slide**, the token on **your Azure Shell** screen. You will use this token in the next few API calls.

# Create service on port 80 using Curl and REST API v3.1



You can do this right in your Azure shell.

Remember copy ( ctrl-C ) and paste ( shift-insert ) are your friends.

PS: sorry for the small font! The line below wraps around onto two lines.

So please select the entire line, all the way from the C in curl, to the last single quote '

Paste it into a scratch pad like notepad first, and replace <your BWAf public IP> with your actual BWAf public IP, and replace the token shown here, with your actual token.

```
curl -X POST "http://<your BWAf public IP>:8000/restapi/v3.1/services " -H "accept: application/json" -u  
"eyJldCI6IjE1NzkzOTE3NTAiLCJ1c2VyIjoIYWRtaW4iLCJwYXNzd29yZCI6IjNkYWwMDMzNTI2\nZGFjMDUxZThmZmM0YzY4ZWZmU0In0=\n:" -H "Content-Type: application/json" -d '{ "address-version": "IPv4", "app-id":  
"curl_app_id", "ip-address": "<bwaF private ip>", "name": "curl_service", "port": 80, "status": "On",  
"type": "HTTP"}'
```

Get the **private** ip address of the BWAf from the Azure Portal

So in summary: replace your public ip, your token, and your private ip.



```
"curl -X POST "http://104.45.141.170:8000/restapi/v3.1/services " -H "accept: application/json" -u "eyJldCI6IjE1NzkzOTE3NTA1LCJlc2VyIjoiaWRtaW4iLCJwYXZd29yZCIEImkZTCczYzNIbmE4LnZDVmMTU1iwidXNLciIMFkbWluIn0=\n","app-id": "curl_app_id", "ip-address": "10.0.1.5", "name": "curl_service", "port": 88, "status": "On", "type": "HTTP"}'>
```

```
rvice","token":"eyJldCI6IjE1NzkzOTE3NmILCjwYXZd29yZCIEImkZTCczYzNIbmE4LnZDVmMTU1iwidXNLciIMFkbWluIn0=\n","msg":"Successfully created.")
```

Name	Status	Hostname	IP Address	Port
default				
default				
curl_service	✓		10.0.1.5	88

We didn't create a backend server yet. That's next.

# Create the backend server.



We already created a backend Ubuntu server in our Azure resource group running DVWA in a docker container using Terraform. Now let's add that do the BWAf service we just created.

You can do this right in your Azure shell.

Remember copy ( ctrl-C ) and paste ( shift-insert ) are your friends.

PS: sorry for the small font! The line below wraps around onto two lines.

So please select the entire line, all the way from the C in curl, to the last single quote '

Paste it into a scratch pad like notepad first, and replace <your BWAf public IP> with your actual BWAf public IP, and replace the token shown here, with your actual token.

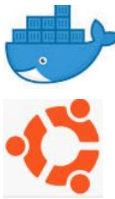
```
curl -X POST "http://<your BWAf public IP>:8000/restapi/v3.1/services/curl_service/servers " -H
"accept: application/json" -u
"eyJldCI6IjE1NzgzOTE3NzMiLCJwYXNzd29yZCI6ImZkZTczYzNlNTMzM2JlOWRkZmQwNTQ1NmE4\nZDViMWI5IiwidXNlciI6
ImFkbWluIn0=\n:" -H "Content-Type: application/json" -d '{ "ip-address": "<ubuntu private ip>",
"status": "In Service", "comments": "string", "port": 8080, "address-version": "IPv4",
"identifier": "IP Address", "name": "curl_server" }
```

Get the private ip of the ubuntu  
vm from the Azure Portal

Showing services 1 - 1 of 1

	Name	Status	Hostname	IP Address	Port
[-] [cloud icon]	default				
[-] [folder icon]	default				
[-] [server icon]	curl_service	✓		10.0.1.5	88
[+] [server icon]	curl_server	✓		10.0.1.4	8080

# Start the web app, using docker, on the back-end server



You can do all this in the Azure shell, you don't have to be in any particular directory. Copy and paste, or type in, the following commands to ssh to the Ubuntu box, become superuser, update it, install docker, and run the usual docker DVWA container. Paste it into a scratch pad like notepad first, and replace <your ubuntu public IP> with your actual ubuntu public IP

```
ssh azureuser@<your ubuntu public IP>  
( answer yes if asked )
```

**Note: You are now in the Ubuntu VM, not the Azure Shell.**

```
sudo su  
apt-get --yes --force-yes update  
apt install docker.io --yes --force-yes  
docker run -d --rm -it -p 8080:80 vulnerables/web-dvwa
```

Wait a minute until the container is fully running, then type **exit** twice to exit out of the SSH session, back to the Azure Shell.

# Test with your web browser.



Browse to `<BWAf public ip address>` and you should see the DVWA login screen. If you see it, no need to login, you are successful! You've provisioned a full service using the BWAf Rest API, congratulations. But you used Curl, which is really quite primitive, so let's get a little more professional than that in the next step, and use something many devops folks use to prototype API automation: Postman.

Before we proceed, please `DELETE` your backend server, and service, using the BWAf GUI via your browser. `Thank you for not forgetting this step.`



# Good work.

Before we proceed, please **DELETE** your backend server, and service, using the BWAf GUI via your browser. **Thank you for not forgetting this step.**

# Postman Section



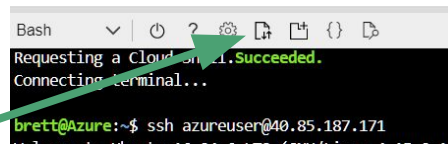


# Postman



Now that we have our Infrastructure, we can prototype some configuration management.

**Download and install Postman** ( use your search-fu )

A screenshot of a terminal window. The title bar says 'Bash'. The terminal text shows 'Requesting a Cloud Shell...Succeeded.' and 'Connecting terminal...'. Below that, a green prompt 'brett@Azure:~\$' is followed by the command 'ssh azureuser@40.85.187.171'.

```
Bash
Requesting a Cloud Shell...Succeeded.
Connecting terminal...
brett@Azure:~$ ssh azureuser@40.85.187.171
```

1. Copy these files from the Azure Shell using the file transfer widget on the Azure Shell toolbar to your PC/Mac

[Barracudawaf.postman\\_collection.json](#) and [barracudawaf.postman\\_environment.json](#)

 barracudawaf.postman\_collection.json barracudawaf.postman\_environment.json

# One more shout-out to Newman



No actions for anyone here, just another plug for Newman. Newman is a great way to run Postman as a script. I originally planned to first show Ansible being used to launch a Postman collection as a newman script, because that is how I and some others might do it.

But I thought that might be a bit confusing. So we're going to skip this part, this is just here as FYI.

```
ec2-user@kali:~$ !n
newman run barracudawaf.postman_collection.json -e barracudawaf.postman_environment.json
newman

barracudawaf

→ Barracuda WAF Login
  POST http://40.78.68.192:8000/restapi/v3.1/login [200 OK, 279B, 324ms]

→ get all services
  GET http://40.78.68.192:8000/restapi/v3.1/services [200 OK, 13.63KB, 307ms]

→ get service
  GET http://40.78.68.192:8000/restapi/v3.1/services/ [200 OK, 13.63KB, 309ms]
```

# The Collection



1. Import up the Collection and Environment in Postman

2. Select the Barracuda Environment

The image shows a screenshot of the Postman application interface. At the top, a dropdown menu displays "No Environment" with a downward arrow, an eye icon, and a gear icon. Below this, the main interface shows a collection named "barracudawaf" selected in the left sidebar. A green arrow points from the "barracudawaf" text in the sidebar to the "barracudawaf" dropdown menu. Another green arrow points from the "Import" button in the top right of the main interface to the "Import" button in the top right of the main interface. The main interface also features a "Send" button, a "Save" button, and tabs for "Request Script", "Tests", "Settings", "Cookies", and "Code". A smaller inset window shows the "Postman" application menu with options like "File", "Edit", "View", and "Help", and buttons for "New", "Import", and "Runner". The "Import" button is highlighted with a green arrow pointing to it from the main interface.

# Change Environment Variables



Change the Ubuntu and BWAf internal ip addresses to align with how Terraform has instantiated these virtual machines. Look in your resource group in the Azure portal. Add them to the Postman environment.

barracudawaf		Edit
VARIABLE	INITIAL VALUE	CURRENT VALUE
token	eyJwYXNzd29yZCI6ImM4YmNhNTgwOWI0ODc2OTgwMDI2NmM5OTdiMjc5NTMxliwZxQjQilxNTc5NDE1NjE1liwidXNlci6lMkYwLWlud0=	eyJwYXNzd29yZCI6ImM4YmNhNTgwOWI0ODc2OTgwMDI2NmM5OTdiMjc5NTMxliwZxQjQilxNTc5NDE1NjE1liwidXNlci6lMkYwLWlud0=
url	40.85.190.9:8000	40.85.190.9:8000
service	postman_service	postman_service
bwaf_private_ip		10.0.1.4
ubuntu_private_ip		10.0.1.5
Globals		Edit
No global variables		

# The Runner

Go ahead and run the collection. The postman GUI has a few moving parts and is not the most intuitive GUI



The screenshot displays the Postman interface with the 'Collection Runner' modal open. The left sidebar shows a list of collections, with 'barracadawaf\_original' selected. The central panel shows the details of this collection, including a list of requests: POST Barracuda WAF Login, PUT Create a frontend service, PUT Create a backend server, GET get all services, GET get all details about a service, GET get service port, PUT change service port, GET get service port Copy, PUT change service port Copy, GET get service port Copy Copy, GET check service mode, PUT change mode to active, GET check service mode Copy, PUT change mode to passive, and GET check service mode Copy Copy. The 'Collection Runner' modal is open, showing the 'barracadawaf' environment, 1 iteration, 0 ms delay, and a list of requests to be run. A 'Run barracadawaf...' button is at the bottom. A green arrow points to the 'Run' button in the top bar of the collection details panel.

History Collections APIs BETA

New Collection Trash

barracadawaf\_original  
15 requests

POST Barracuda WAF Login  
PUT Create a frontend service  
PUT Create a backend server  
GET get all services  
GET get all details about a service  
GET get service port  
PUT change service port  
GET get service port Copy  
PUT change service port Copy  
GET get service port Copy Copy  
GET check service mode  
PUT change mode to active  
GET check service mode Copy  
PUT change mode to passive  
GET check service mode Copy Copy

DNS-LTM-POP\_v12.1

API This collection is not linked to any API

Share Run View in web ...

Collection Runner

File Edit View Help

My Workspace Run In Command Line

POST Barracuda WAF Login  
PUT Create a frontend service  
PUT Create a backend server  
GET get all services  
GET get all details about a service

Environment barracadawaf

Iterations 1

Delay 0 ms

Log Responses For all requests

Data Select File

☐ Keep variable values  
☐ Run collection without using stored cookies  
☒ Save cookies after collection run

Run barracadawaf...

RUN ORDER Deselect All Select All

☒ POST Barracuda WAF Login  
☒ PUT Create a frontend service  
☒ PUT Create a backend server  
☒ GET get all services  
☒ GET get all details about a service  
☒ GET get service port  
☒ PUT change service port  
☒ GET get service port Copy  
☒ PUT change service port Copy  
☒ GET get service port Copy Copy  
☒ GET check service mode  
☒ PUT change mode to active  
☒ GET check service mode Copy  
☒ PUT change mode to passive  
☒ GET check service mode Copy Copy

# A successful run



Here is what it looks like for a successful run. Does yours look the same? Are there any mistakes that needs fixing?

Hint: Yes, there is a mistake in one of tests for the API call. **Fixing it is optional.** If you want to fix it, **you must change the test which looks for port 8080, to look for port 80 instead. And find the "Save" button, and click "Save"**

Iteration 1

- POST** Barracuda WAF Login `http://40.85.190.9:8000/r...af_`  
This request does not have any tests.
- POST** Create a frontend service `http://40.85.190.9:8000/r...`  
This request does not have any tests.
- POST** Create a backend server `http://40.85.190.9:8000/r...`  
This request does not have any tests.
- GET** get all services `http://40.85.190.9:8000/r...cudawaf_01`  
This request does not have any tests.
- GET** get all details about a service `http://40.85.190.9:8000/r...`  
This request does not have any tests.

GET `http://{{url}}/restapi/v3.1/services/{{service}}?groups=Service&parameters=port`

Params ● Authorization ● Headers (8) Body Pre-request Script **Tests ●** Settings

```
1 pm.test("Port is 8080", function () {  
2   pm.expect(pm.response.text()).to.include("8080");  
3 });  
4  
5
```



# Good work.

Before we proceed, please **DELETE** your backend server, and service, using the BWAF GUI via your browser. **Thank you for not forgetting this step.**



# One more thing

Before we proceed, please kindly **end** your docker container.

```
ssh azureuser@<your ubuntu public IP>  
sudo su  
docker ps
```

( the left-most column is the docker container id, **copy the container id** )

```
docker kill <paste your container id>
```

**Exit**

**Exit**

```
azureuser@bwaf-tf-vmubuntu:~$ sudo su  
root@bwaf-tf-vmubuntu:/home/azureuser# docker ps  
CONTAINER ID        IMAGE               COMMAND             CREATED  
dd289080dad6        vulnerables/web-dvwa  "/main.sh"         21 minu  
root@bwaf-tf-vmubuntu:/home/azureuser# docker kill dd289080dad6  
dd289080dad6  
root@bwaf-tf-vmubuntu:/home/azureuser#
```

Thank you for not forgetting this step.



# Ansible Section



# Ansible Playbook



Please examine the Ansible Playbook using any method you like, but please spend 5 minutes looking at all of it.

It's not an example of perfection. But if you aren't familiar with Ansible, you can learn from it. Compare it to what you have learned earlier in this course about Ansible concepts.

Remember the playbook is named `bwaf-playbook.yaml`

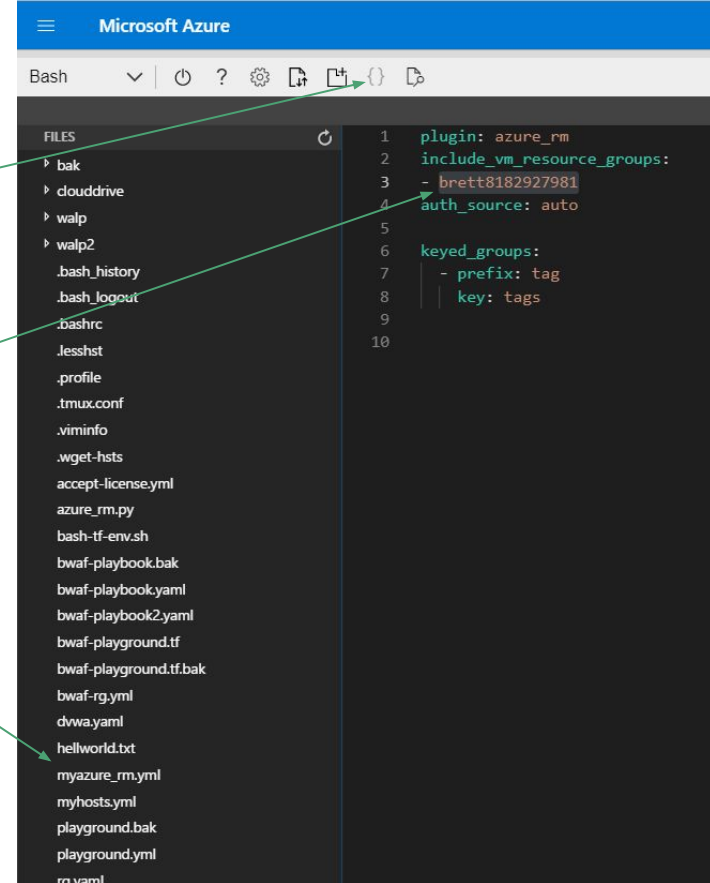
# Ansible & dynamic Inventory in the public cloud

The inventory is not static in this case, so we don't know the public IP ahead of time. Azure provides a dynamic inventory plugin to Azure, that we can use to get it. But we must edit one file to specify our unique ID, so get read to edit! Editing files is easy in Azure shell, simply click the { } edit icon and edit the file named **myazure\_rm.yaml**

**Change** the string **change\_me** to **your unique ID**. In the example shown here, I have changed it to **brett8182927981**, but it may be your first name\_ last name so for me that would be **brett\_wolmarans** ( the one you used earlier )

Right click, Save the file

Right click, and Quit



```
1 plugin: azure_rm
2 include_vm_resource_groups:
3   - brett8182927981
4   auth_source: auto
5
6 keyed_groups:
7   - prefix: tag
8     key: tags
9
10
```

# Run the ansible playbook on the BWAF

We needed the dynamic inventory because our infrastructure was created dynamically, and we don't know the public IP address ahead of time, hence the need for the dynamic inventory file.

We now can run the playbook named `bwaf-playbook.yaml` to configure the BWAF

```
ansible-playbook -i ./myazure_rm.yml ./bwaf-playbook.yaml --limit bwaf_tf_vmbwaf*
```



Ansible command

Dynamic Inventory file

Our playbook

Just our BWAF, not the Ubuntu box

# Run our Ansible playbook for the DVWA application

Now, we will run the playbook `bwaf-dvwa.yaml` to run our Dxxx Vulnerable Web Application. It actually does three things:

1. Updates the Ubuntu 16 box
2. Installs docker
3. Runs the dvwa docker image

Our remote  
username



```
ansible-playbook -i ./myazure_rm.yml ./bwaf-dvwa.yaml --limit bwaf_tf_vmub* --key-file ~/.ssh/id_rsa --u azureuser
```

Ansible command



Dynamic Inventory file



Our playbook



Just our Ubuntu box, not BWAf



our private SSH key



Make sure to answer “yes” when you see the prompt below:

```
TASK [Gathering Facts] *****
The authenticity of host '40.117.169.170 (40.117.169.170)' can't be established.
ECDSA key fingerprint is SHA256:Pi6W+UgCeHcNAkpjUo4ugDwHJQLV+0L56fcKELxtTs.
Are you sure you want to continue connecting (yes/no)? yes
ok: [bwaf_tf_vmubuntu_6445]
```

# Manual verification

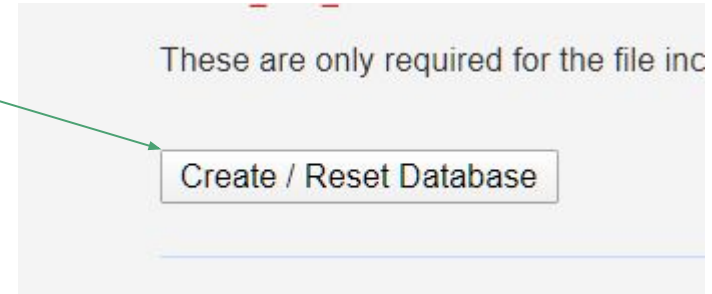


1. Find your BWAF public IP in the Azure portal
2. Use your web browser, browse to the BWAF, login, and verify in the BWAF GUI the WAF configuration look good
3. How did you know what password to use? Is this secure, insecure? Discuss with the person seated next to you...

# Finalize DVWA application setup



1. Verify that you can access the DVWA app by browsing to the Ubuntu box public IP address, port 8080, like this  
`http://<Ubuntu public ip>:8080`
2. Login as user `admin`, password is `password`
3. Click “`Create / Reset Database`”
4. Scroll down and Click `login` at the very bottom of the screen  
( if you wait 30 seconds maybe it will do it for you )





# SQL Injection testing

1. Log back in to DVWA as user **admin**, password is **password**
2. Choose SQL Injection from the left menu
3. Enter **1** for the user, it will return a single user

**Vulnerability: SQL Injection**

User ID:

ID: 1  
First name: admin  
Surname: admin





# SQL Injection testing

1. Perform a simple SQL Injection
  - a. Enter for the user `' or '1'='1`
  - b. Do you get a list of all users? ( hint: yes )
  - c. This is because this web application is Dxxx vulnerable, and is not protected.
2. Now, in a new browser tab, browse to the BWF service `http://<bwaf public ip>:80` , login, and repeat step 1 above.
3. Did the BWF successfully block the SQL injection attack? Did the BWF log the attack? What do you need to change to get a successful block?

## Vulnerability: SQL Injection

`' or '1'='1`

User ID:

ID: ' or '1'='1  
First name: admin  
Surname: admin

ID: ' or '1'='1  
First name: Gordon  
Surname: Brown

ID: ' or '1'='1  
First name: Hack  
Surname: Me

ID: ' or '1'='1  
First name: Pablo  
Surname: Picasso

ID: ' or '1'='1  
First name: Bob  
Surname: Smith

# Final Step: Cleaning up the Infrastructure

```
terraform destroy
```

Enter **your** first name + **your** phone number when asked. For example, my name is Brett and my phone number is (818) 292-7981, so I will enter brett8182927981

Answer “yes” when asked

When it finishes, verify the Resource Group is gone from your Azure Subscription, by browsing around in the Azure Portal.

It should take 3.5 minutes.

# Good work.

Before we proceed, please `terraform destroy` your entire infrastructure. Or if you really want, you do it manually in the Azure Portal, using your browser.

Thank you for not forgetting this step.

The End