# Script to Accompany Introductory GENI Slides

This file and the accompany slides live at:

http://groups.geni.net/geni/wiki/GENIEducation/IntroTalks

# Notes

| A vertical line in the text indicates when to advance the animation (for example, with the down arrow on the keyboard)

# Table of Contents

GENI Terminology 2

How SSH with a Private Key Works 4

GENI Expiration and Renewal 7

Behind the Scenes of GENI Experimentation: An Introduction to GENI Tools 8

GENI Support for Automating Resource Reservation 10

# GENI Terminology

There are five terms which are helpful for understanding GENI. They are slice, project, experimenter, resource, and aggregate. Let’s talk about them one at a time and see how they all fit together.

<NEW SLIDE>

An experimenter is a researcher who uses GENI resources. Different types of experimenters have different roles and permissions. For example, an advisor and their graduate student have different roles and permissions. Likewise, a teacher, a TA, and their student have different roles and permissions.

<NEW SLIDE>

An experimenter creates an account when they log into the GENI Portal at portal.geni.net.

<NEW SLIDE>

The GENI Portal leverages the InCommon federation to support single sign on authentication by experimenters.

Students, faculty and staff will have an account provided by their school (for example, the University of Utah) which they use to access campus resources.

They can then use that account to login to GENI as long as their school is a member of InCommon (with some caveats). \*\*This is exactly like using your Facebook or Google account to access third-party websites.\*\*

For these experimenters, there are no new passwords. | All other experimenters can request an account on the GENI Project Office’s identity provider.

Anyone with an account can log in, but they will have no privileges, as you must be a member of a project to do anything interesting.

<NEW SLIDE>

Projects organize research in GENI. Projects contain both people and their experiments (referred to here as slices). And a project is led by a single responsible individual, known as the project lead. The project lead takes responsibility for whatever happens within their project and agrees to respond appropriately if a problem is discovered with resources in their project.

<NEW SLIDE>

A slice is a container or label for resources used in an experiment.

A slice can contain resources from one or more aggregates.

It belongs in a single project.

And has an expiration.

Finally slice names are public, reusable, and unique within a project.

This means:

* You should assume that your slice name could be seen by others so it should not contain private information
* A slice name for an expired slice may be reused, but there can only be one slice at a time with a given name in a given project.

<NEW SLIDE>

A resource is a piece of reservable infrastructure. A resource can be a bare metal piece of equipment or can be virtualized.

For example, bare metal computers and virtual machines are both resources

Wireline networking resources like VLANs as well as wireless resources WiMax are valid resources.

Resources are described using resource specification, which are universally refered to as RSpecs.

<NEW SLIDE>

An aggregate is the people and software which manage a set of reservable resources. GENI racks are the most common type of aggregate. In addition, OpenFlow resources are their own aggregates as are WiMax resources.

<NEW SLIDE>

Let’s put this all together.

An experimenter is a member | of a project which is | led by the project lead.

Within this project there are | one or more slices.

| The slice contains virtual machines connected by a VLANs | which are from one or more aggregates.

# How SSH with a Private Key Works

Secure Shell or SSH is universally used to remotely login to systems running UNIX or UNIX-like operating systems, such as Linux.

In GENI, we use SSH with a private key. This is new to many people, so to understand how this works, let’s first review a more familiar concept:

how ssh with a *password* works.

<NEW SLIDE>

| This is Jane. | She wants to log into to a remote system using ssh with a password.

| From her terminal on her local machine, | she types ssh followed by her user name, in this case jdoe, then an @ sign and the name of the remote machine.

| Then she is prompted for the password for her user account on this remote machine.

| She enters the password and the remote machine compares the hash of that password to the hash of her password stored on the remote machine and lets the client in.

| When she is done, she types exit.

| If she wants to log into another machine, she repeats the same procedure and enters the password for the account on this other remote machine.

Note that if she has many machines to log into, she will enter her password many times. This is comparable to having to manually enter the combination on a padlock on a gym locker. Every time you want in, you have to spin the lock.

<NEW SLIDE>

| Contrast this to the case where Jane logs in using SSH with a private key.

In this case, she has generated a cryptographically linked public-private key pair in advance. The public key is stored on the remote machine and the | private key is stored only on the local machine.

When she wants to connect, she need to tell SSH about the private key so SSH can use it to authenticate the session.

There are many ways Jane can tell ssh about her private key, but one easy way is to use a piece of software called an SSH agent.

| On a terminal on her local machine, Jane types ssh-add and the full path to her private key. Ssh-add is the command to tell the ssh agent about her private key.

| When she presses enter, she will be prompted to enter the *passphrase* for her SSH key.

Passphrase and password are very similar sounding words, but they are actually very different concepts. The passphrase protects the private key on her local machine and is only known by her and never leaves her local machine. This prevents someone who had improperly gained access to her local machine from using her private key to log in to other remote machines.

| Once Jane enters the passphrase on her private key, | she is ready to log into a remote machine using the usual ssh command.

Unlike before where Jane authenticated her session with a password, now Jane authenticates her session using the private key.

When Jane opens the ssh connection to the remote machine, | the server creates a challenge which it encrypts using her public key | and sends to the client.

Her client decrypts the challenge | using her private key which only she has access to. | And then replies to the server with a message that includes the original challenge. | Now the server is able to determine that the client was able to decrypt the original challenge and lets the newly authenticated client finish connecting.

| At this point, it may appear as if nothing has been gained. Before she entered one password to log into one machine. Now she has entered one passphrase to log into one machine.

| However, when Jane goes to connect to a second machine, she is logged right in because the ssh agent automatically authenticates the ssh connection with her private key.

| This is true for subsequent logins as well. While this is convenient for a few nodes, this becomes critical if you have many machines to use. Now Jane can log into 10, 20, or 100 machines without having to enter her password 10, 20, or 100 times.

| Finally, when Jane is done for the day, she removes her private key from the ssh agent using ssh-add –D so that if someone were to improperly gain access to her local machine then they still could not access her accounts on the remote machines.

<NEW SLIDE>

Using SSH with a private key is not only easier to use but is also more secure than using SSH with a password. This is because the private key is much longer than the 8-16 character password that you would use instead.

Because of the added security, all compute nodes in GENI use SSH with a private key.

<NEW SLIDE>

| You should *never* be prompted to enter a password to login to a GENI compute node. If you are, then something has *always* gone wrong. Either you are not specifying the private key OR the GENI software has failed to load your public key on your node and you should ask for help.

# GENI Expiration and Renewal

GENI resources are provided for free on a first-come-first-serve basis. GENI uses expiration times to ensure resources are returned to the pool of available resources for use by other experimenters. To prevent reserved resources from returning to the pool, an experimenter must renew their resources before they expire.

<NEW SLIDE>

| Resources have an expiration that is | bound by an individual aggregate’s limit. | In addition, resources are reserved within a slice and the resource reservation’s expiration time | can not be any later than the slice’s expiration time.

Moreover, different resources within the same slice may have distinct expiration times.

| In turn, the slice’s expiration time | can not be any later than the project’s expiration time (if it has one).

| So in general, to renew your reservation, you must renew (or extend the expiration time) of both the resource reservation and the slice containing those resources.

# Behind the Scenes of GENI Experimentation: An Introduction to GENI Tools

<NEW SLIDE>

Today let’s examine what happens behind the scenes when you reserve resources in GENI?

All GENI resource reservation tools use the same data formats and APIs. These are resource specifications or RSpecs and the GENI aggregate manager API or AM API.

<NEW SLIDE>

|The researcher interacts with the GENI federation using | a tool.

GENI has | a clearinghouse which is responsible for managing | users, projects, and slices.

| Aggregates are the software the makes | resources available to users.

The researcher uses | clearinghouse APIs to speak to the clearinghouse.

For example, her tool will | request a new slice on her behalf and the clearinghouse | will return a slice credential.

In turn, her tool uses the | aggregate manager API (or AM API ) to request resources from the aggregate. | For example, it passes the slice credential and carefully formatted request RSpecs to the aggregate and the aggregate replies in kind.

Let’s examine this process in more detail.

<NEW SLIDE>

GENI resource specifications (or RSpecs) are plain XML documents that describe resources. For example, this is a complete RSpec for a single node with one interface. The green text indicates this is an RSpec (in fact it is a request rspec). The blue text describes the node and the orange text indicates the single interface on the node.

<NEW SLIDE>

There are three flavors of RSpec: advertisements, requests, and manifests.

| Aggregates use advertisements to describe the resources they have *available*

| Experimenters use requests to describe the resources they *want*.

| Aggregates use manifests to inform experimenters what resources were *actually allocated* to them.

<NEW SLICE>

Experimenter’s tools and the aggregates pass these RSpecs back and forth to each other using the aggregate manager API or AM API.

For example, the experimenter’s tool | will ask the aggregate what resources are available, using the ListResources command. | The aggregate will return an advertisement RSpec specifying what resources are available.

| The experimenter will then select the desired resources and their tool | will describe those resources using a request RSpec which is passed to the aggregate using a CreateSliver call. The aggregate will either accept or reject that request. | If it accepts the request, the aggregate will return a Manifest RSpec. The manifest is very similar to the request, but more fleshed out. For example, the manifest will contain login information and more details, such as IP addresses, that may not have been in the request.

| A little while later, the experimenter may have forgotten what resources she had, so | she makes another ListResources call (this time specifying a slice) to find out what resources she has. | The aggregate will reply with a new copy of the manifest. | |

<NEW SLIDE>

Since all GENI resource reservation tools speak the AM API and RSpec, pick the tool that works best for *you* and your current situation.

# GENI Support for Automating Resource Reservation

There are three basic techniques to use when automating GENI experiments.

They are:

* Creating custom OS images
* Creating scripts that run at node boot time which are called “install scripts”
* Using open source configuration management tools such as chef or ansible

Let’s take them one at a time…

<NEW SLIDE>

There are a variety of existing operating system images you can use on GENI. If one of these is not appropriate for your experiment, you can create a custom image. Regardless of whether you use a standard image or a custom image, the image is specified in the RSpec with the disk\_image tag.

<NEW SLIDE>

*Use text from slide*

<NEW SLIDE>

GENI also allows the use of “install and execute scripts” which are just scripts which run when the node boots. They are commonly shell scripts but they can be written in any language – for example python.

In fact, install scripts don’t have to be a single script at all. Despite their name they are actually a compressed tarball which can contain an arbitrary number of files including scripts and flat files.

<NEW SLIDE>

|To use an install script, the experimenter puts the .tar.gz file on a publicly available webserver -- here it is myserver.com – | specifying this location in their RSpec.

The experimenter also specifies where this file should be place on her node when it is booted – here it is /local.

| When the node is booted, | it downloads the specified tarball and places it in the specified location. | Then it decompresses the file. ||| Finally, it executes whatever commands or scripts are specified with the “execute” tag in the rspec – in this case, install-script.sh which is a file that was in the original tarball.

<NEW SLIDE>

*Use text from slide*

<NEW SLIDE>

Finally, you can use standard open source configuration management tools such as ansible and chef to configure your nodes.

Configuration management tools ensure an experiment is in a known configuration regardless of it’s original state. And they make it easy to reproduce experiment configurations. This feature makes it easy to do multiple runs with the same setup or with systematic variations such as changing parameters and scaling topologies.

<NEW SLIDE>

*Use text from slide*

<NEW SLIDE>

Automating your experiments is important for experiment repeatability and reproducibility. It also makes life easier for the experimenter.

Use of a combination of OS images, install scripts, and configuration management tools to automate your experiments.