LolFuzz — Unrestricted Fuzzing at Scale

# Abstract

This proposal is a response to the mutual need that we have with regards to setting up, managing, and sharing an individual's fuzzing infrastructure. At the present each of our needs are satisfied independently via custom-scripting, preparing a range of templates in an ad-hoc manner, or usage of open-source tools that are available. The issues that these all have in common are that sharing resources or results against a particular project or campaign with another member is typically performed by sharing the unprocessed results from a run or having to hand-write/hand-hold the other participants through the creator's infrastructure. In most instances this will suffice as the research field we're in is a very independent one, however this does not scale in that re-visiting a particular project involves either locating (and maybe repairing) the previous deployed infrastructure or starting over from scratch.

For the past half-decade Administrators have decided to learn to code and have opened up APIs to virtualization so that developers or fellow staff are able to script management of a cluster. This combined with the evolution of application development towards scalability has built a community described as the DevOps community. Due to the various cluster solutions that have been implemented, different APIs each with differing intentions and requirements have come to exist. These APIs intend to simplify resource management by exposing the ability to compartmentalize resources in a cluster. This has results in various API some of them with differing options and different intentions. In order to work around the differently motivated solutions that have been created, the DevOps community has developed solutions that aim to generalize interacting with these different solutions. This project aims to utilize some of the technologies that have been developed by this community in order to remove any restrictions against any particular virtualization platform that could exist.

The DevOps community embraces a concept called Infrastructure as Code (IaC). This idea is specifically to allow a developer to describe the technologies their application depends on and the relationships between an application's different components so that it may be shared and even modified by other developers. Inter-developer communication can then be facilitated using a version control system and then used to track how an application's needs changes over time. This allows multiple developers to describe each component as a generic resource and can then modify each resource without mistakenly affecting another requirement of the application. Another significance of this model is that the same description can then be used by an administrator when deploying/maintaining the application in a production environment. This allows an administrator to scale the application up or down without needing to understand anything specific about the application as long as they're able to fulfill the requirements of the resource/component.

Some of the main ideas that the author would prefer to solve using these technologies are the idea of describing the requirements of a fuzzing environment and abstracting its ability to scale. Another aspect that he'd prefer is to simplify the development/customization of a mutation-engine or input-case generation. The final point is to allow one to customize how result aggregation can be performed and develop further research without affecting the current infrastructure in use. Utilizing these technologies should then allow an individual to share/modify the current state of a fuzzing infrastructure amongst multiple users or develop tools/research at any of these stages without interfering with an entire instance.

## Setup Terminology

* Target — the target software that is being campaigned against.
* Repository — a base collection of code before it is specialized for a target.
* Project — the repository that is responsible for containing a fuzzing environment after it has been specialized for the given target.
* Resource — a general machine (or container on a machine) inside a virtualization environment providing some need.
* Role — a policy/list-of-rules that is applied to a resource in order to satisfy a particular need. More than one role may be applied to a machine.
* Machine or VM — a single running virtual machine based on an image of some kind.
* Template — a description or image of some kind that is used to build a Machine/VM
* Container — a resource or packaged-tool providing some particular functionality that can co-exist with other containers on a single machine.
* Host — the user's machine that is used to connect to the virtualization environment.
* Server — the virtualization environment that is responsible for virtualization a number of machines.
* Master — a controlling machine (or number of machines) of some sort that has the ability to communicate with other virtual machines in a project's cluster and manages shared resources such as distributed storage or applications.
* Test-case — a single generated test-case that is tested on the target software
* Pod — any number of machines/resources that are used to process a single test-case.
* Instance — a single Pod that is brought up with a specific goal in mind.
* Cluster — a number of Pods that are used to satisfy the user's goal. The number of Pods can then be scaled according to the user.

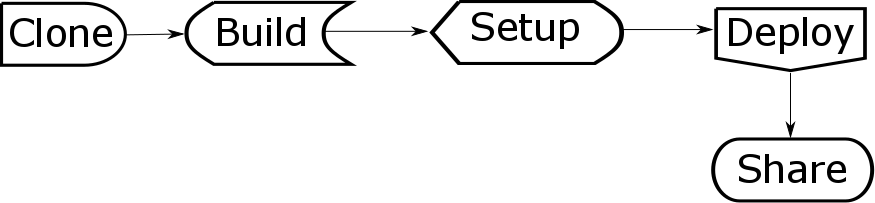
## Setup Workflow

When fuzzing a target, there are multiple states that the project will be in before fuzzing would begin. The initial workflow is organized as follows: Building, Specialization, and Deployment. Once the project reaches the "Deployment" state, then management of the resources in the project can begin. This can then allow different instances to be brought up with particular goals such as: Development, Single-Processing, and Continuous-Processing. Each of these states will be described later. During the first three states, each of these are used to produce an item that is included in a configuration file based on YAML, JSON, or some other format. This configuration will then be saved within the repository that is shared with all users maintaining the project.

The Building state is simply a state that builds the main templates that are required for the entire project. These will only need to be done once for the master machine, and any operating-systems/machines that are required by the target. It is significant to note that these machines are intended to be general and can simply be built by reusing another template and then configuring it specific to the project. The only requirements of these machines are that they're remotely provisionable by pushing with either SSH, WinRM, or another similar protocol. After these templates exist, their names will need to be noted by the user to be used during Specialization. These names will then be referred by the user when entering the "Specialization" phase.

Within the Specialization phase, the user will modify the target-specific templates that were produced during the "Building" phase. This phase requires the most manual work in that the user must either provision the installation of the target software or manually install the software and any of its requirements into a machine. This will produce a set of templates that can be used to bring up an instance of the software and its requirements. As an example of this phase, if the target Adobe Acrobat Reader is chosen, the user must connect to the VM on the server, transfer Adobe Acrobat's installer on the target, and then go through the installer. If it is a software based on a Posix environment, then they must compile the target software or any variations of the target software (such as ASAN versions) on the VM. It is prudent to note, that one can use a containerization solution such as Docker for the differing variations of the target. If some complex software is to be deployed that is dependent on communication between multiple machines such as IBM's LDAP Server with Kerberos, the user must produce one template for the client, one template for the server, and one other for the Kerberos TGT Server. Once these templates are produced, the user must note these template names within a configuration file within the repository. These templates are then used by the project to describe a Pod which will be used to drive the target. One specific is the template for the Master. This template will simply need to be configured with the project-name and is in most-cases automatically pre-configured. At this point, the Master has been specialized and may be brought up at any time.

The final phase of setting up a project involves Deployment. This phase involves defining the provisioning required for each particular VM and the application of any Roles to each machine that will compose a Pod. Based on the templates that were defined during the Specialization phase, the user may use any of these templates to describe any of the various machine roles that will be used within a single Pod. A user is allowed to re-use any of the templates defined during Specialization as a unique Role. Later, these roles will typically be applied by a standard provisioning tool when the user chooses to deploy the actual Pod. It is intended that this is driven entirely by a declarative configuration and so due to this there is an opportunity for a user to define numerous styles of Pod deployment if they so choose. The details of Roles and how they fit within a Pod's design will be described in the next section. Also during this phase the user may deploy the master template and use it to provision a test deployment of a Pod. This can be used to assist a user with troubleshooting a particular Pod deployment. This may also be used to provision a development environment so the user may develop a tool against the Target software.



The majority of these configuration phases are intended to be used to produce configuration files that will be used by the entire project in order to manage the relationships between each resource in a Pod. At this stage, the configuration files produced can also be used to provision any number of machines in either the virtualization environment or even locally on a user's machine for testing. The master machine will be used by a Pod in order to handle numerous tasks such as non-push provisioning, orchestrate the overall resources for the cluster, and application of roles or commands that are made by a user. The master must include support for containerization and may be used to host various platform-independent tools that can work outside a Pod as well as providing shared resources for any of the requirements of the entire Project. These tools may be used to expose a web-interface, or perform post-processing on the raw output of each Pod, or really anything the user chooses to deploy. It is intended that this template is scalable linearly in order to accommodate an increased or decreased need for resources during the project's lifetime. More details about what the Master provides will be described later.

## Pod Terminology

* Master — the machine that contains any shared resources used to coordinate the entire Pod or provide services for the Pod.
* Generator — a role that is responsible for generating a test-case in some form. This can involve generating it from thin-air or consuming some input and mutating it in some form.
* Trigger — a role that is responsible for taking any number of test-cases from a generator and executing it against the target.
* Monitor — a role that is responsible for monitoring the state of the target and reporting it.
* Agent — a role that is responsible for capturing any other kind of state to be delivered to a Monitor
* Driver — an abstraction that ties the processing of a test-case within the Pod together and allows a user to not have to worry about details related to the communication between the different roles.
* Package — an object of some kind responsible for storing the output of the trigger, the monitor, or any number of agents. This may be communicated over any kind of local or remote protocol as long as it is understood by the driver.
* Report — A combination of packages that is submitted back to the master

## Pods and Roles

A Pod essentially describes the pipeline that is used by a Target to both generate and process a single-test-case. In order for a Pod to work, there are some scripts/tools that must be developed. Primarily it is intended that the majority of generic roles would be included within the base repository and chosen by the user during the Deployment phase, but it would also be possible for the user to develop the application required to satisfy a particular role. Any specific software requirements for any of the roles can be defined by a base policy as specified in whichever provisioning tool that is utilized during deployment. At that point, a Pod will automatically be configured with whichever roles are being applied to it.

Roles are important to distinguish in that they may exist as multiple independent components/tasks, or be satisfied by a single software component/service that communicates within a single process. It is up to the user to specify which role goes on which target when defining the Pod during the Deployment phase. Thus, a single machine may contain multiple roles deployed on them and include any number/type of agents that they so choose.

To simplify the explanation of a Pod, it may be described as a pipeline that begins with a Generator and ends with a Monitor. The role that is responsible for producing a test-case is known as the Generator. The generator is one of the only roles that are allowed to communicate outside the Pod. This software may query a shared resource for test-cases that need to be mutated, or to query output that was stored from a previous hit-trace in order to determine how it should mutate a test-case. A simple generator could be one that generates random data. A generator acts similarly to a server and when it is queried with some kind of input should produce some kind of output test-case. In most cases this role will simply query the Master for some mutated input and then serve it, but if the cost of generation is more expensive or needs to be performed as a single instance per Pod can exist as so. One reason why it's suggested that a Generator is included as part of the Pod pipeline is in order to allow a user to directly control which inputs will be used by a particular Pod instance. Any number of generators may also be chained together such as in instance where multiple inputs are queued to a ramdisk and then handed off to the trigger for performance.

The trigger is the component that is responsible for driving the entire Pod and is responsible for resetting the target and executing a test-case against it. Although this component can be easily generalized, it is expected that a user implement this as per whichever target they are campaigning against. After the driver receives input from a Generator, the Trigger will take this test-case and load the target application with it. It's important to distinguish that a Trigger is not just limited to loading a test-case into a target application but may also be responsible for initializing a handshake, sending traffic to the target application, and then closing the socket. This trigger may also be responsible for executing a particular system-call to perform local-kernel testing. The driver for the trigger will signal to each of its agents any input that it deems is necessary, and then execute the test-case. It may also produce a package that is submitted to the monitor's driver to include in its report.

An interesting role that may be used within a Pod is the role of the agent. An agent may be used in order to capture any other kind of data that may be out-of-band of the trigger or monitor. This can be used to capture extra data such as a packet capture, data sent through a proxy, an authentication token of some kind, or even something as simple/dumb as a timestamp in order to include as a package for the monitor to submit back to the master. Another example may be if a trigger is being used to test system-calls within a VM, the trigger may choose to submit the code it is about to execute to an agent whose responsibility is to submit it as a package to the monitor's driver in order to capture the input before the platform crashes. The monitor's driver may then collect the code that was being executed by the trigger from the agent and bundle it alongside any of the other packages that it will submit to the master alongside its report. The trigger will submit any input chosen by the user to each agent with a unique-id or input of any kind (such as a IP4 and TCP port to listen on). Then when the monitor chooses to submit a report back to the master, its driver can request any out-of-band packages from each agent and then bundle it along with the report.

The other role that is allowed to communicate outside the pod is the Monitor role. By default it may simply be a debugger that will monitor the health of the target but may be anything such as a serial debugger or remote debugger of some kind, or simply the output from some custom program. If the target state is deemed unhealthy (i.e. sig11), a report will be produced as a package. When the monitor's driver receives the packages produced by the monitor, it will query all agents and then bundle their packages together into a report which will get submitted back to the master in order to expose to the user in some way. There is no distinguishing between any number of packages produced by the monitor and/or agents. This way if the monitor wants to include something a crash dump, output from stdout/stderr, and the results of a hit-tracer it may do so.

The driver is simply a general abstraction around the different roles that run as services and is responsible for synchronization between each role in a system. The driver is intended to be chosen during the deployment phase and should allow the user to write their roles in any language they choose and in any form they so choose. This way if a tool that they want to use runs w/ fork/exec semantics or as a service, they may choose whichever one satisfies their needs and each respective component will receive whatever is needed in order to synchronize the roles.

|  |  |
| --- | --- |
|  | In this chart, the trigger (in green) fetches a sample from the generator, and then signals a message (with a package multi-casted to all agents). When the target is run and the monitor determines that its health should be reported on, the monitor (in red) will ask all agents (and trigger) to submit their collected data. After this is done, all packages received from the components will be consolidated into a report which is stored on the master. |

## Cluster Management

Once the Pod has been defined and is in a deployable state then the infrastructure may be managed. There should not be many things that are needed to be managed by the user, but it is intended that the user be able to perform all management from their individual Host. This should consist of bringing up some number of Pods using one of the deployments that they or another user had defined during the Deployment phase. Another option for deployment could consist of bringing up a development environment using a single Pod in order to develop a new agent, trigger, or monitor of some kind. This may be adjusted on a per-user basis via a localized dotfile in their home directory to be merged with anything that's required by the project itself when provisioning a pod. Upon any kind of modification of the infrastructure, the infrastructure state will then be updated so that multiple users may have access to any of the resources that are part of the project.

Although it is intended that the user be able to perform all infrastructure management from their individual host, if it is chosen to do so it may also be feasible to include all of the required tools on the master VMs so that a user may shell into one of them to avoid installing tools on their host. At this point management for the infrastructure should be independent of which user initially created the project. In order to orchestrate deployments, each Pod's identifier must be persisted in some way. It's suggested to persist this in a key-store on the master VMs. If necessary extra metadata for each pod can also be stored per identifier which can be used by any member of a pod for auto-discovery and configuration. If a user wants to take the last master VM down, it should be possible to serialize the infrastructure's deployment state and included as part of the repository in case another user wishes to pick the project up again. An example of what these states might contain will be described later.

|  |  |
| --- | --- |
|  | In the following diagram, a user that wishes to use the project may simply pull from the project’s repository. Once this is done, they will then have access to all of the Pod deployments that exist. At this point if they wish they made then add a custom deployment and then bring up a number of instances. |

## Master Terminology

* Service — a component that provides a globally accessible service to Pods in the cluster
* Application — a component that provides a unique service to the user
* Container — an atomic accessible unit that is used to package a service or an application
* Store — a service that stores various things needed by the cluster
* Provisioning — preparing a machine with any required software in order to satisfy a given role

## Master

The Master VM or VMs (if defined as a cluster) should contain a key-value store which would be reachable by every member of the pod. Excluding the base-services on the VM (such as those needed for management of lower-level distributed services or maintaining peers required for clustering), multiple applications and services may exist. Some basic services that could exist would be a database or service (such as Beanstalk) in order to facilitate report output from a Pod's driver. Another service which may exist could be a shared filesystem, or a centralized server for provisioning of Pods if required by some provisioning tool.

By allowing the master VM to be defined as a cluster, some applications may also be deployed to allow the user to automatically post-process report output as produced by each Pod. This can be something such as a web-application or a database for querying report output, or even an application that exposes infrastructure management provided by the project as a different protocol. By utilizing containers, this can also be scaled according to the resources that are available within the virtualization environment.

# DevOps Tools

There are various DevOps tools that can provide solutions to some of the problems needed by this project. This section a quick summary of each, a link to their website, some opinions on their usage, and predictions on their future. The next section is on page 14 for those whom are already familiar with the DevOps toolset.jj

## Provisioning

Provisioning tools are responsible for applying roles to a VM or container. They can either work in push or pull mode and are intended to replace shelling into a target and executing some commands. The difference between them and SSH is they intend to be platform independent and allow one to specify the state a machine/container should be in. This way if the state of the machine/container is changed, they can immediately be re-provisioned without suffering from configuration "drift"

### Puppet — http://www.puppet.com

Puppet was essentially the first major open source configuration tool. It contains a declarative language and is based on Ruby. It's based on a client-server architecture which means in order to provision a machine, one would need to run a Puppet Server. It includes support for Microsoft Windows but was an afterthought in its design. This is a pull-oriented system, where an agent on each machine will query the Puppet Server to look for changes. The protocol is HTTP-based.

### Chef — http://www.chef.io

This is a fork of Puppet. The idea is that one would write a recipe as part of a cookbook that describes what state a machine would be in. This can be run in a client-server (pull) mode with an agent, but also has support to run in a push mode. Originally it was written in Ruby, but there's a rewrite of it going on in the works based on Erlang. The protocol is also HTTP-based.

### Ansible — http://www.ansible.com

Ansible is different from the other provision tools in that it's agentless. This means that each server requires SSH or WinRM to be bound to a port in order to be provisioned. It's written in Python, and although it claims to have support for Windows their "scheduler" is fork-based. Their modules also require python to be installed on each system. Due to them not using a real RPC of any kind they're literally executing modules by including code inside a .py file template using a format-string. They use the concept of playbooks which includes a list of commands to execute on each target machine. Due to the tool not using a declarative language that describe the roles a machine should have and orient it towards what list of commands should be run on a machine/container, one would need to write multiple "plays". One to undo a previously run command, and one to apply a new state. This was the first provisioning tool that claimed to have windows support and thus the one that the author has personally had more experience with. However after using it, it's just a little bit more generic than “mssh”.

### SaltStack — https://saltstack.com

Saltstack is probably the most mature out of all the provisioning tools. Some of the terminology is weird to get the hang of, but uses a declarative language and was designed with support for role inheritance out-of-the-box. It includes support for an agentless mode which requires SSH/WinRM and python on a target machine. If run with an agent, however, it uses ZeroMQ for communication. It also comes with installers for literally all platforms and is very well supported. It uses Jinja2 for defining roles using templates and includes a modular plugin system for retrieving attributes from any provisioned machine if necessary.

### Otter — http://inedo.com

Otter is essentially a Windows-specific provisioning \_and\_ infrastructure tool. It's supported by a company and thus will be maintained as long as that company exists. It's based around Powershell and includes support for managing Linux via agents. It's only being described here because some of us like Powershell.

## Containerization

Containerization allows one to package different parts of an application in layers that can be merged together. This allows one to run an application or number of applications compartmentalized on top of a filesystem. This whole world is pretty much ruled by Docker and thus they've set the standard.

### Docker — http://Docker.com

Docker is pretty much the standard. Some of the features that were included in the Linux kernel (control-groups and namespaces) helped birth Docker. Docker has drivers to support composing filesystem layers, support for virtualizing applications, and isolation of system calls, etc. The negative side is that it depends on Linux. However, recent conversation with Rich Johnson points out that native Docker support for UI applications exists on Windows 10 and Windows Server 2016. Previous versions of Windows rely on Msys and thus performance is pretty unnatural.

### Rkt — https://coreos.com/blog/rocket.html

Rocket was developed as an alternative to Docker due to it evolving into this crazy platform that does hundreds of different things. Docker originally was designed to be layers that can be stacked together in order to compartmentalize the different parts of an application. Unfortunately it turned into just a tool for people to just script installations and the idea of composability was lost entirely. Docker has now turned into this single service process model that's running as root and used to manage numerous Docker instances. This essentially discards security since everything runs as root and so Rkt was conceived to work around that issue. Rkt is designed with composability in mind and is able to run as various users. However, it only works on Linux and does not have the community that Docker has.

### Solaris/Illumos Zones — http://docs.oracle.com/cd/E26502\_01/html/E29024/toc.html

Yes. Solaris has support for branded zones which can be used for stacking virtualization, and a union-capable filesystem. The difference between zones and FreeBSD jails are that they can be stacked on top of one another which satisfies the containerization primitive.

## Infrastructure as Code (Iac)

IaC is described in the abstract of this document. However, the author's experience with the tools that aim to solve this problem are either suffering from immaturity or we have issues with using it against our ESX cluster due to permission issues.

### CloudFormation — https://aws.amazon.com/cloudformation/

This is amazon's solution to the IaC problem. It's pretty complete, well-documented, and even includes a web-interface to manage infrastructure and monitor cluster load. Its only weakness is that it's AWS only.

### Kubernetes — http://kubernetes.io

Kubernetes is essentially the open-source standard for scalable management of infrastructure. It's fairly complex and utilizes load-balancers in order to distribute work between the different parts of an application. It has a powerful command line and even includes a web-interface. It has support for various back-ends as well. The major issue it has is that an atomic execution unit for a cluster is a Docker container which means that their concept of infrastructure involves each component existing inside a Docker instance. There's no concept of a VM, and due to the requirements of Docker no real support for Windows. Maybe due to Windows 10 and Windows Server 2k16 this might change, but that could also mean that there's no support for remote kernel debugging or any other specific features. Some research on this might need to be done, but this author wouldn't hold his breath.

### Docker-Swarm — https://www.Docker.com/products/Docker-swarm

This was pointed out to me by Rich Johnson. It essentially turns multiple Docker containers into a single automatically scalable Docker instance. It exposes an API or is useable by command line. Due to the way Docker works, one can package all the parts of their application into a single Docker image and then use Docker-Swarm to manage the entire cluster. As described previously, native support for Docker exists on Windows 10 and Windows Server 2016. Defining a Pod might require one to type in the commands to install their target application.

### Apache Mesos — https://mesos.apache.org

This was discovered while writing this proposal. It essentially is a distributed scheduling system for managing Docker containers. It supports distributing “cron” jobs throughout a cluster and has the ability to deploy new images as resource are needed. It’s used by Apple’s Siri, eBay, Yelp, and twitter. Probably just a little bit more mature than some of the other solutions.

### tsuru/Docker-Cluster — https://github.com/tsuru/Docker-cluster

Docker by some Brazilian company. Very simple scheduler for managing Docker instances.

### Decking — http://decking.io

Also using Docker as an atomic unit of execution. Pretty simple and stupid, but includes a scheduler for describing how to scale up.

### Terraform — http://terraform.io

This tool is written by Hashicorp which are the same folks that wrote Vagrant. This aims to be the main IaC solution for arbitrary platforms with no requirements on Docker or anything. It includes a declarative language for describing how an infrastructure might look which is stored internally as JSON. When modifying an infrastructure, it will build a dependency graph, bring up nodes as required by the target application, and apply configurations or trigger provisioning based on each resource type defined by the user. It has the option to specify variables which can be used to easily control how many nodes (or PODs based on how you describe the relationship) to be brought up. It does not come with a scheduler as that is intended to be solved by another tool or can be written by a developer.

This was one of the tools that the author had personally tested as the documentation pretty much abstracted the problem of describing machine relationships entirely. The way it works is each resource is described in a .tf file and can include support for variables. When applying the .tf file, terraform will then communicate to the virtualization environment what needs to be cloned and such whilst outputting any variables that you want to export from a resource. Once applying this state to the infrastructure, Terraform will store the state of the infrastructure to a local file or to a shared resource that is accessible by every Terraform user. One issue the author had was related to enumeration of network interfaces due to a permissions issue on our ESX instance. It also seems like support for ESX might be immature as they haven't updated the usage of the govmomi module (VMWare's VIX API for Golang). This is due to the current instance of govmomi being v0.12.0 and Terraform still using v0.6.2.

### Nomad — http://nomad.io

Another famous Hashicorp tool. This one is intended to work together with Terraform in order to schedule jobs on a cluster. You provide a job-description to execute as either a service or a single-execution command and it will create-and-provision VMs/containers as needed, and then destroy them when done. It intends to automatically handle scheduling based on what resources are available and will handle balancing the load amongst all resources that are currently available. They include a custom job-description language which will tie into whatever virtualization provider you prefer. In most cases this is to augment the aspect of infrastructure automation that Terraform doesn't provide.

The author had concerns about the maturity of this tool and although it would be neat to describe each task in a fuzz-run as a job, at most there'd only be maybe 2 or 3 different job types. The author believes that there aren't enough job types during fuzzing to warrant the need for a job scheduling tool like this and one would be fine with just using straight-up provisioning to handle infrastructure. Still it's there if one wants to go down that path or figures out a need for it.

### Vagrant — http://vagrantup.com

This is the most famous tool written by Hashicorp and is written for synchronizing development environments between multiple developers. It allows a developer to define how a development environment would look like which can be committed to a repository. Other developers can then pull the repository and then use it to re-create an application's development environment on their own system. It wasn't really intended for management of infrastructure but it can still be done with a little bit of hackery.

This was another tool that the author had personally tested as it was what we had originally believed to be the best solution. The author had written some code on top of this to drive VM deployment and provisioning according to a configuration file which was clumsy. Cory had also tested sharing some VMs with me but due to machine state being decentralized and stored per-developer, are unable to interact with each other's VMs without having their IDs. This can probably be worked around by installing a single vagrant environment and sharing it amongst users so that each person can see any modifications to the infrastructure.

### Otto — https://ottoproject.io

This was intended to be the next iteration of Vagrant. Vagrant was originally focused on sharing environments between developers and allowing developers to easily bring up an environment for whatever project type they were working on. It had big goals as it intended to be the go-to application for developers, but of course it became vaporware. Hashicorp now focuses on infrastructure instead of development and other similar solutions.

### OpenStack — http://www.openstack.org

Probably one of the most complex "cloud" infrastructure solutions out there. Infrastructure management is exposed to the user as a web-based API. Based on Dozens of technologies. Supports all kinds of craziness... Pretty much overkill.

### Salt-Cloud — https://docs.saltstack.com/en/latest/topics/cloud/

SaltCloud allows one to bring up hosts in various virtualization (or cloud) solutions and automatically provision them with whatever role one would want. This is based on SaltStack and due to the power of SaltStack's templating language and its API can be used to automatically manage infrastructure state as the user changes it. SaltStack wasn't originally intended to do this, but..it's capable of it. Some code might need to be written in order for it to be elegant. The majority of virtualization providers are supported.

Once the author encountered permission issues utilizing Terraform to create a client/server Pod from templates (due to enumeration of network interfaces), he ended up looking at using just straight-up provisioning to automate the management infrastructure. Since SaltStack has inherent support for roles, and code can be written to organize the way nodes in a Pod can be brought up. Salt-Cloud might be a solution that could suffice.

## Building

There's aren't many tools out there that automate building. Most people build a VM per virtualization instance manually and just ensure that there's a private/public key for SSH to give to the user. Due to Vagrant being one of the earlier tools, there's hundreds of Vagrant boxes that are provided by default which people will import into their virtualization environment. Some use PXE booting and either Kickstart or Bootstrap or Autounattend (on Windows), but most of this is manual.

### Packer — http://packer.io

Another tool by Hashicorp. This is pretty much the best tool for installing an operating system and building a template in an unattended manner. The author had done some work on it earlier in the year in order to support building a template in ESX with a Soundcard, Serial port, or a USB bus. It's based on Golang, and can output to pretty much any kind of virtualization solution or write to any kind of virtualization format. Also can output a Vagrant box which can then be loaded into any cloud provider that Vagrant supports which is pretty much all of them. If you're not using a homegrown solution, this is pretty much the standard way to automatically install an operating system for every platform.

### Cobbler — http://cobbler.github.io

This is pretty much a Linux distribution with tools that simplifies good old-fashioned PXE booting a number of machines. Everything is then served over TFTP and then you can automatically provision everything once the OS has been installed. This is unique in that it's useful if you're needing to bring up a bunch of bare-metal machines within your environment. Fortunately it works with virtualized environments as well.

## Environments

These would pretty much be used to create the base platform that one would provision to deploy their application(s).

### Docker Hub — http://hub.Docker.com

This provides a handful of Docker environments that can be used to package an application. There are numerous ones that include base environments by default, but again since this revolves around Docker and if an environment is required that's not Docker then this would not be useful to solve the problem.

### Atlas — https://atlas.hashicorp.com

Another Hashicorp solution that provides a searchable way of retrieving pre-built images that have been made by other people. This is probably one of the more mature places to download pre-built images. Each machine is pre-loaded with SSH and a vagrant public key so that you can easily clone and provision. Most of these are VirtualBox due to Vagrant being designed with VirtualBox in mind.

### CoreOS — http://coreos.com

A very minimalistic Linux (like Alpine) that is designed from the ground-up for clustering. A lot of the orchestration technologies that have come out for Linux have been initially designed by CoreOS. The distribution is pretty simple, driven by a configuration file or values in a key-store, and provides containerization out of the box. Updates are handled by simply grabbing an Etcd lock, installing on the spare partition, and then booting into it. After some health checks, the partitions are synchronized. It comes with two services to facilitate node-discovery and clustering, namely Etcd and fleetd. Installation is super-quick and easily scriptable although it's using a custom version of cloud-init for pre-provisioning the image on bootup. The author's prototype build is based on CoreOS and uses Etcd's key-store for self-discovery. This is the platform that is recommended to use with Kubernetes.

### ProjectAtomic — http://www.projectatomic.io

This is RedHat's response to CoreOS. They include a wrapper around Docker so that you can automatically label and tag Docker instances when deploying. (The author had to write his own for CoreOS). With regards to updates, it's different in that it uses RedHat’s standard way of updating software via running a tool and keeping a local cache of filesystem state. One thing which is also different is that since it's using real package management, their version of cloud-init is using the very latest version instead of a custom one which includes a couple more features when prepping an image for deployment..

## Stores for Service Discovery

Stores are typically used for service discovery and persisting configuration in a cluster. Although DNS w/ updates satisfies this and has solved this forever with the “\_tcp” and “\_udp” subdomains (rfc6763), the DevOps folks re-solved this. Typically service-discovery and configuration are stored with key-value pairs and as such there's many databases/caches that can do this. They mainly used to keep track of address/ports for the different services within a cluster.

### Zookeeper

Developed for Hadoop. Uses Java. Only mentioned because it was the first.

### Etcd

Comes with CoreOS by default. Is simply a distributed key-value store. It's using the raft algorithm to share knowledge amongst all of its peers. It can be communicated with via REST. It's very fast and minimalistic with also support for ACLs. It doesn't support service-discovery/registration out of the box, but it's not too hard to write code that does that either. Most of the infrastructure management tools such as Terraform and Kubernetes can register services using Etcd. Service registration would look like a PUT request to http://Etcd-server/.../service-name/host:port or if you need the next host for a unique-id you can request to http://Etcd-server/.../pod-id/service and get a list of addresses.

### Registrator — http://gliderlabs.com/registrator/latest/

This is just a simple service-registration/discovery-system that can be backed by Etcd or Consul. It's based around Docker but can be applied towards anything since it's literally just making a web-request to store status based on node health.

### confd — https://github.com/kelseyhightower/confd

This is a simple tool for converting configuration files into templates and then populating those templates using data stored in either Etcd or Consul. In most cases configuration changes would be done by provisioning, but if for some reason one would prefer a dynamic configuration they can specify this tool to run as a service. When the key in Etcd or consul that is being monitored changes, then confd will go through and re-create the configuration file restarting the service that the configuration depends on if necessary.

### Consul

Hashicorp's solution. Written in Go. Uses a gossip protocol to share data with other members of the cluster. As opposed to just storing configuration, it also comes with a service-discovery and registration system that can be based on REST or even DNS. This would mean that each node would need its DNS configured towards an instance of Consul and then name-resolution would then be used to locate the different components to communicate with. Another neat feature is that it has the ability to work across datacenters as well and also support ACLs. Requests are similar to Etcd except that you can also use DNS to query for services.

### Kube-DNS/Kube-Proxy

Kubernetes uses Kube-Proxy to monitor service state which will automatically get updated internally. At this point service-discovery can be handled using DNS and Etcd due to the Kube-DNS service.

### Docker DNS Server

Yes, Docker has it. It's extraordinarily immature, and of course revolves entirely around Docker.

# Implementation

Basing the entire project around Git allows each user to share and improve upon the base-project. The directory structure of the repository could look something like the following which breaks down each step into a directory of work with tools in order to isolate the work required for each stage. By doing it in this format, the user is enabled to do any of the stages manually if necessary. The output for each stage can then write any required configuration into the root of the repository. The prototype uses GNU Make for each stage. For management and building any tools, the root directory can contain a Makefile that takes variables for the different types of deployments.

|  |  |  |  |
| --- | --- | --- | --- |
| What a directory structure of components might look like during the lifetime of a project. | | | |
| Build/  Build/Templates  Specialize/  Deploy/  Deploy/Deployments/\*/  Shared/Agents/\*/  Shared/Triggers/\*/  Shared/Generators/\*/  Shared/Monitors/\*/  Tools/\*/  bin/…  src/… | Trigger/\* | Monitors/\* | Deployments/\* |
| Agents/\* | Generators/\* |

The tools that are being considered are as follows:

* Interface — GNU Make
* Builder — Packer
* Store — Etcd
* Provisioning — SaltStack and Saltstack Server
* Infrastructure — Terraform or Salt-Cloud
* Containers(?) — Docker or Rkt
* Driver — Native Shell (cmd or bash) w/ a directory of helper tools, or Golang could be used

## Setup

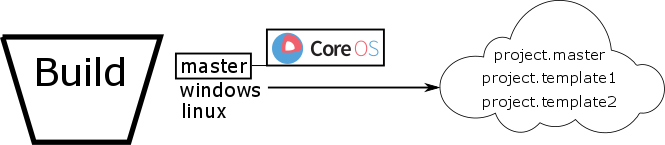
Setting up requires the manual labor to bring up a particular project. First the user will clone the base repository and then proceed to go through the stages of deployment. Most of this should be driven by minimalistic configuration files. Experiments have been done using YAML, but could really be anything. Due to provisioning tools being potentially complex, they are not exposed to the user directly. Instead, it's intended that provisioning configurations are generated by templates using inputs specified in the project's configurations. This should allow different parts of the setup process to be generically described. Also since a YAML configuration is being used to generate templates, if a user wishes to modify the provisioning scripts directly, they may do so as well. Once the project is configured and the templates are generated, the user may then commit the clone into a new repository specific to the target they are fuzzing. This way other users have access to the work they have done.

|  |  |
| --- | --- |
|  | In order to build the main project, the base repository must be cloned and these steps must be executed. The build step will create the templates/images that represent the unique operating systems that will compose a Pod.  The Specialize step will then customize these operating system images with the target software and anything default the user may want on the virtual machine. This will result in images that can later be cloned and provisioned for a role.  Once the final deploy step is reached, the user will define via configuration what a fuzzing instance will look like. This includes describing which images are used, and how many ways they may be provisioned. During this step, various deployment types may be designed/modified.  At this point, the user may commit their deployment types and configuration into a new repository so that other users may re-use the images defined therein. |

### Build

The build directory is responsible for automatically building the base-templates for an operating system. The only output for this directory is either a template or a template-name for the base operating system. This allows a user to skip this step entirely if the operating system has already been built or been manually built by the user. Automatic builds are performed by Hashicorp's Packer. Under this directory are a list of predefined templates in packer's template format based on “boxcutter-windows” for the Windows platform, and “chef-bento” for the Linux platform. There are some tools to help GNU Make generate/customize configurations for the user. These tools will be used to customize either of the templates for whatever specific configuration the user decides. Also under this directory is the template used for building the Master image in order to specialize it for the cluster. CoreOS was chosen as it includes numerous default software and is well supported by the community.

To build these templates a Makefile in the root of the project can be called that reads variables from a file that is specified before build. This can recursively call into Make in order to specify template names for the next stage and emit the public/private key in order to communicate with each machine.



Once complete, then they're ready to be specialized for the project.

### Specialize

The Specialize directory is responsible for customizing the platform VMs specific to the project. During this phase, the master cluster is seeded with the project name. Any software that is necessary for provisioning the target (such as a package manager or language runtime) is then installed on each platform image and finally the user must install the target software within the platform image. This directory will include some scripts to assist the user with provisioning some target software using the platform's package manager if the user does not wish to do it manually. The provisioning tool (SaltStack) can then be used in "push" mode to install the target software. If provisioning is chosen by the user to install the target software then the requirement to persist the platform image in the virtualization environment if the project is un-deployed is removed. Some of these produced images may also satisfy more than one role in a Pod. In these instances, the images may be modified while provisioning during the deployment phase. The output of this directory produces the actual image names or templates that are used to build the entire cluster.

|  |
| --- |
|  |

### Deploy

Now that the images are built, ready to be provisioned, and a master image exists that can be used to orchestrate the cluster, the Pod is ready to be configured for deployment. A user will specify via a configuration file which images as per the Specialize step are to be used per each deployment type they want to provide to other users. These configuration files will then be used to build the configuration specific to the provisioning tool in order to support the user's requests. It's significant to note that a Pod does not need to be complex and may exist as just a single VM such as if a user is fuzzing a file loader.

With regards to the configuration and SaltStack's support for inheritance, one may specify the base configuration for each VM inside a Pod and then specify multiple deployments for how they wish to organize different Pod styles. Also inside this configuration the user will specify which images include any of the Agents, Generators, Triggers, or Monitors. At the very minimum a Trigger and a Monitor must be specified. Some default deployments will be included such as one for bringing up a single development environment given a Pod. Some samples should also be included in a samples subdirectory. Also during this phase the user may test out a deployment to test out which tools will fit for their particular deployment. A developer deployment can also read from a user's local dotfile so that a user may customize what software they use on their target. This can be used to automatically install tools such as IDA, gdb, radare, etc. on a particular user's development environment.

|  |  |  |  |
| --- | --- | --- | --- |
| Before deployment, the Pod must be defined. This points images created in the previous step to their name within the Pod. During this, images may be made unique via roles as defined by provisioning. | | | |
|  | | | |
| Now, any number of deployments may be created and then deployed. | | | |
|  |  |  |  |
|  |  |

Other than the configuring of Pod deployments, the Master images are also ready to be configured and deployed. The project's master is responsible for orchestrating how each Pod deployment works utilizing a distributed store. This allows any user to specify any number of services/applications (under Tools/\*/) to include on the cluster. This will be used to seed the store and can be modified at any time. This will enable any user to add more tools as they come up with them so that later when more developer resources become available they may be shared with other users of the project and more importantly update the "base" repository with them. Once the user specifies the Tools to be on the master (such as shared disk space for example, or a database for querying), a seed for the key-store will be serialized so that it can be committed to the project's repository.

When the first master is brought up, this seed will be loaded into the distributed key-store. If the master already exists, then the configuration on the key-store will be modified. This is so that if any tool/application requires more virtualization resources than currently available, they can scale as many master images as they so choose. Due to usage of fleetd and flanneld, each service can exist on any member of the project master cluster.

Deployment of any number of Pods is part of the idea of IaC and could be solved using tools such as Terraform or SaltCloud combined with SaltStack. This would be done by processing the user's input configuration and using it to build the specific states as required by either of these DevOps tools. If neither of these tools solve the problem effectively then something could be rigged until a proper solution is produced by the open-source world.

## Master Services

On the master there are only a couple services that must exist which are required for orchestration of deployments. If a user wishes to include other services (Tools/\*), these can be dynamically modified. Some examples of these other services could be some post-processing for reports that are generated by each Pod's Driver or Monitor. Another service which was suggested earlier could be a shared data-store of some kind for input to a Generator. One may also write a service for the master that mutates data in a data-store so that a Trigger can just fetch a test-case and execute it using the rest of the Pod. This type of configuration allows for arbitrary flexibility. Later if one so chooses they may then write a web-application to manage the cluster.

### Containers

Each tool is expected to interact with data generated by a Pod or independent of a Pod. This should allow a user to develop a tool in a platform independent fashion. If this is possible, then if the user chooses to run the tool inside a container then it can be automatically scaled by the master cluster. This allows the Tools directory to essentially be a list of containers. Each tool/container should be composed of two layers. One of them should be the read-only file system including any binaries or code/requirements for the tool, and the other should be the writeable filesystem part of said tool. Fortunately this should be relatively easy to automate with tools like Docker or Rocket.

### Store

The store is used to save any kind of shared configuration for the entire cluster. As specified during deployment, a seed is generated which describes each deployment type that can exist. This is used to keep track of each Pod type that has been deployed. If any provisioning tool or configuration tool as required by the cluster needs to store any information, this may also be stored within the store. This store should also be accessible by any member of a Pod so that Pod component may use something defined within this configuration. This can be done with either Consul or Etcd. Etcd comes with CoreOS by default so it's the natural choice.

|  |  |  |
| --- | --- | --- |
| Various services can be run on the master, if they are containerized then clustering on the master will simply consist of duplicating the image. There are many types of services that may be provided to assist any of the generators or an application that is exposed to the user. | | |
|  |  |  |

### Data-store

This could literally be implemented using anything, in some cases it might not even be needed. Either NFS, SMB, iSCSI, DAV can be done.

## Pod Tools/Components

In order to facilitate the execution of each test-case, a number of tools must exist. As described during deployment, a Pod does not need to be complex and may consist of just a single machine if a user is fuzzing something as simple as a file loader. It's expected that each component may be written in any particular language and thus may have any kind of software requirements. Each tool will include a configuration for any particular software requirements as required by the component which will be handled during provisioning of a deployment. This should allow a user to specify any tool, executable, or script in order to load a test-case into a target and monitor for its health. The output of any of these tools can then be saved and submitted to the monitor when a report is being made.

By designing it in this manner, this will allow development resources to be applied only when the particularly complex is desired by the user. Initially the Generator would simply apply a user's command to some data-store. The Trigger will reset a target program and then pack the sample that was fetched for the monitor. Then the monitor will unpack the sample filename and start the target application using the file as input. Upon a crash, the monitor can either run “!analyze” or “!exploitable” and also produce a crash-dump. The crash-dump, the sample, and the results of “!analyze” or “!exploitable” can then be bundled as a report and submitted to the server. As more complexity is needed, it can be added either per-project or to the base repository (if written generically) when more resources are available.

Some basic terminology must be defined:

* Component — the type of script the user has specified in the Pod's Role to use during deployment. A generator, trigger, monitor, or agent. Should be very simple.
* Driver — an application that is responsible for running the user's script and providing an API based on some local IPC in order to capture output from a user's component.
* Store — a data-store such as Etcd that is accessible from any member of a Pod
* Test-case — the single test-case that is executed through a Pod.
* Pack — an isolated packet of some kind of data that is generated by any component which will be used later to compose a report
* Report — a list of packs generated by each component in a Pod that is saved to a data-store of some kind on the master.
* Generator — a service that is responsible for producing test-cases that are used by the Trigger
* Trigger — a component that will load a test-case into the target software
* Monitor — a component that will monitor the health of the target software. This is the only part that makes a decision/has a conditional.
* Agent — a component that will collect out-of-band data of any kind to include with the bundled report to the master.

### Driver

A driver (src/\*) could be written for each component in order to abstract the sending of different Packs to the Monitor for submitting back to the Master. It was considered to just write this driver in native shell-script (Powershell, Cmd, or Bourne) with some binary helpers for communicating each pack, but it may be written in anything. The intention of the driver is so that the user may describe a tool to be as simple as a shell-script that executes their target program using a test-case as input, or passing arguments to a tool such as tcpdump, or radamsa. At this point the user component may emit some output in some form, and then submit it to the driver using basic I/O so that it will get bundled for the report. Initially it was considered that the driver would simply need to fork and exec the user's component or something similar to just capture any kind of output they'd want to emit, but if necessary more than one driver type can be developed if the user chooses to write their tool as a continuously running service of some sort in order to control execution performance. Some examples can be described later.

The driver would simply specify inputs to the user's component via environment variables or some mechanism, execute the user's chosen tool, and then remain responsible for capturing outputs and packaging it to the monitor. Any tool may produce any number of outputs and so each of these outputs can be bundled together as a report and then stored when a Pod as completed execution of the test-case. Some examples of outputs can be stdout/stderr from a debugging instance, or a crash-dump, or even binary data that is written to an anonymous pipe that is inherited by the target process.

Although the driver can be satisfied with some shell-script and maybe some helpers, if a more complete language is chosen then some form of IPC should be exposed to the user's tool so that it is accessible from an arbitrary languages so that the driver can communicate the resulting pack to each component within a Pod. If a complete language is chosen to wrap the user's tools, then using a multicasting-type protocol such as ZeroMQ is recommended for orchestrating each execution of a test-case. Each component/role within the Pod does not need to exist as separate processes or tasks as long as each component can produce a pack that communicates to any number of agents which is then aggregated by the monitor.

By including a generic driver in the base repository that will simply exec a user's tool that is written per project, this allows each component used by deployment to be written with very little development effort. Any particular options that may come from a configuration, or from a Package during processing of a test-case can be handed off to the user's tool either via the environment, or via some specific helper-tool that the user may run to read input provided by the driver. This helper-tool (or driver fabric) can process I/O via some local IPC, or a multicasting protocol such as ZeroMQ which the author is a huge fan of.

### Generator

The generator is simply a tool that produces some kind of test-case when signaled. This can be specified by the user as a script that reads from a data-store and runs a tool such as Radamsa on it, some python that mutates some input, or even some number of bytes emitted from a pseudo-random number generation device. A default tool that should be bundled with the base repository is one that will generate a list of test-cases as specified by the user. This default tool will allow a user to test a range of cases to assist with triaging. When generating a test-case it is up to the generator's driver to produce a unique-id of some kind by using a hash or some other algorithm. The generator should also be able to communicate with any of the global services that are available on the master. This allows for a generator to take input from another component and use it to influence its test-case generation.

### Trigger

The trigger is simply a tool that will feed an input that is produced by a Generator into the target software. This can be done by either spawning the target application with the test-case as a command line argument, or sent to a server, or the test-case can even be included as part of the input for an Agent on a network-proxy which will send the traffic to a server. The trigger will always produce at least one pack which is always sent to the Monitor component. Any other packs that are produced by the trigger will be sent to any agents within the same segment.

### Monitor

The monitor is simply a tool to monitor the health of a target application. This can be as generic as a debugger being attached to the target process, to something as simple as tailing a log-file while looking for an error message. If the user chooses, then this component could also be responsible for launching the target process so that it may be started with a debugger directly attached to it. This would be performed by defining a Trigger that includes the test-case as part of its package and then as the monitor receives it, it will be used to start the target application.

When an error is detected by the user's monitor, it is up to the user's to emit a final report package, and then return/emit a state in order to request any packs generated by any agents. At this point, the monitor's driver will take the pack generated by the user's tool, and request any packs produced by the Trigger as well as any agents within the Pod's segment. The monitor driver will then submit each pack together as a report to the master cluster so that a user or service can interact with them.

### Agent

An agent is simply an arbitrary tool that is used to produce a Pack using input from a Trigger. As an example of a situation where an Agent may be used would be if the user wants to log something such as a Packet capture. At this point the user's trigger can submit to each agent the target host/port to all agents, which can then be included as input to the user's tool which would be a shell-script that runs tcpdump. If a user wants to use an agent as a proxy that mutates its input, their Trigger could simply forward a test-case to an Agent as an input Pack, and then their mutator would be an agent that mutates the test-case, and then submits it to the target server. This allows a user to deploy agents up to an arbitrary complexity as they see fit/find a need.

### Examples of some Pods

The very first deployment would exist on a single VM and can be driven entirely within a single process as stimulated by the driver. Once receiving a test-case from the generator chosen by the user, the trigger can emit the test-case as a Pack. Once the monitor receives the test-case, it will start the process with a debugger attached. If a crash happens, the monitor will emit the report to the driver which will then ask all agents to submit any out-of-band data. Since there's no agents defined for the Pod, only the report will be submitted back to the Master.

Another type of deployment would be a client-server type of architecture using a proxy. In this situation a test-case would be generated and submitted to the trigger. The trigger would then pass-through the test-case to all of its agents (really only one which is the proxy). The proxy would then submit the test-case to the target. Once the target crashes, the monitor will ask each agent for any packs and then submit it to the master.

A more complex example could be a Pod that describes a system using a service that is authenticated by a system such as Kerberos. This can be done by storing the trigger on the client with an agent on both Kerberos server and the client. This way both agents can store the exchange of the TGT. Then if the service on the target machine crashes. The monitor can request from both the agent on the client and Kerberos master the ticket that was exchange. This can then be submitted to the master VM that stores all the reports.

Another example could be a local kernel system call fuzzer. In this case, an agent could be stored remotely on the machine that is monitoring the state of the local kernel. The trigger can then submit the code that is to be executed to the remote agent before it executes the code. Then when the monitor detects that the kernel crashes, it may request from the agent that is local to the kernel debugger which may then be bundled with the crash-dump for the user to go through later.

### Storage

Due to the user being able to specify different deployments in a project and allowing other users to manage them as well, configuration information must be accessible at some point when the Pod is brought up. This should be done during the deployment phase. Each deployment type can be treated as a prefix in either Etcd or Consul. Underneath the deployment name would then be a list of each image that was defined by the user, and then under that would be each role. During provisioning, this key will be searched and then used to apply the roles to each image. Under each deployment key, a hash of the deployment configuration and the timestamp can be persisted. This would be done so that one can determine if some particular deployment needs to be updated/re-deployed. Each of these keys are what is serialized as suggested earlier during the deployment phase. This way if the last master is taken down, the deployment rules can still be re-deployed.

This would probably look like:

/conf/deploy/deployment-name/

unique-id : hash-of-deployment-rules(?)

timestamp : seconds since epoch

This could also look something similar to the following pseudo data. Each of the options that are stored underneath each role's key are arbitrary and is left up to the user's script/tool to interpret it however they wish. Later when a pod is brought up, this state is saved and used to start up the respective tool so it should never need to be fetched again when processing a particular test-case.

### This data is a simple file loader that applies radamsa to a corpus, loads it into a suspended process, packs the pid for a gdb instance to attach to the process and resume. “gdb-generic-attach” can be a script that will just call gcore and submit the file if a crash happens.

/conf/deploy/radamsa-attach-fuzz/image-Linux/generator

radamsa-file-loader : extra\_args="-m num=10,ber",path="/path/to/corpus/data-store/"

/conf/deploy/radamsa-attach-fuzz/image-Linux/trigger

gdb-generic-exec : pidvar='PID',send\_pid\_or\_something=true,suspended=true

/conf/deploy/radamsa-attach-fuzz/image-Linux/monitor

gdb-generic-attach: resume=true,pid\_to\_attach\_to=${PACK.PID},corefile=true

### This data applies radamsa to files in a corpus. Packs the test-case that's generated as a filename. When the monitor receives the filename in PACK.SAMPLE, it passes it as an argument to BugId. “my-bugid-command” will capture stdout from the program and submit it as a report.

/conf/deploy/radamsa-load-fuzz/image-bugid/generator

radamsa-file-loader : extra\_args="-m num=10,ber",path="/path/to/corpus/data-store/"

/conf/deploy/radamsa-load-fuzz/image-bugid/trigger

pack-something : package\_sample=true

/conf/deploy/radamsa-load-fuzz/image-bugid/monitor

my-bugid-command : path="c:/target/client.exe",args="${PACK.SAMPLE}", submit\_stdout=true

### This data uses an agent to log a timestamp of start and end as well as a hit-trace. “evo-fuzz-loader” will take an argument of the current state and a path to the current collected hittracer information. “generic-exec” will just start a program and package the pid. “log-hittrace” will log all function hits within the pid that was packed. “my-attach-process” will attach to the process that was given to it in its package. When the process has died, my-attach-process' driver will receive the packs from “log-hittrace” and “log-timestamp” and submit them along the current basicblock, backtrace, and register state.

/conf/deploy/evo-fuzz/ht\_image/generator

evo-fuzz-loader : state="/path/to/state/",profiler\_path="/path/to/profiler/data-store/"

/conf/deploy/evo-fuzz/ht\_image/trigger

generic-exec : send\_pid=true

/conf/deploy/radamsa-evo-fuzz/ht\_image/agent

log-hittrace : pid=${PACK.PID}

log-timestamp : null

/conf/deploy/radamsa-evo-fuzz/ht\_image/monitor

my-attach-process : mypid=${PACK.PID}

### This data uses two images. “generic-loader” will simply fetch a file from a data-store. “send-my-packets” will take the input test-case and send it to whatever is specified in the target option. It may also pack the source TCP port number. Once the “log-a-pcap” agent receives the source port number, it begins logging traffic on “image-server”. “reset-service” will go through the process of resetting the service and waiting for it to die. Then the target process crashes, the monitor's driver will request any packs from “log-a-pcap” (results of packet capture) and submit it along with the crash state to the master.

/conf/deploy/client-server-fuzz/image-client/generator

generic-loader : path="/path/to/corpus/data-store/"

/conf/deploy/client-server-fuzz/image-client/trigger

send-my-packets : target=image-server

/conf/deploy/client-server-fuzz/image-server/agent

log-a-pcap : filter="host image-client and tcp src port ${PACK.TCPPORT}"

/conf/deploy/client-server-fuzz/image-server/monitor

reset-service-and-wait : null

### This data uses a generator that loads 200 samples at a time onto a ramdisk onto the ramdisk-image VM. “multiple-fetch-samples” will then ask generator for some number of samples from the generator. Due to the count option being 10, trigger will then pack 10 filenames which is seen by the “spawn-bugid-with-process” monitor. This monitor will see 10 filenames being passed to it and then start 10 processes at a time using the filenames as arguments. The output for each bugid instance can then be submitted as a report to the master.

/conf/deploy/ramdisk-fuzz/ramdisk-image/generator

ramdisk-loader : data-store-path="/path/to/data-store",count=200

/conf/deploy/ramdisk-fuzz/ramdisk-image/trigger

multiple-fetch-samples : count=10

/conf/deploy/ramdisk-fuzz/ramdisk-image/monitor

spawn-bugid-with-process : options=""

### This deployment type is unique in that it's not actually used to fuzz anything at all, but rather to collect a hit-trace from executing a number of files within the target application. First, the “file-loader” generator will fetch a sample from a path on the data-store. Then the “sample-loader” will request 10 files from it and then pass the sample filename to each agent on the Pod. Afterwards, the “collect-trace” monitor will execute the target process whilst collecting all the basic-blocks. When complete, the driver for the monitor will ask the Pod for any Packages to bundle with the report which will then get submitted back to the master. The user can later choose this deployment to collect a hit-trace for any samples they choose.

/conf/deploy/hittrace-test/machine1/generator

file-loader : data-store-path="/path/to/data-store"

/conf/deploy/hittrace-test/machine1/trigger

sample-loader : count=10

/conf/deploy/hittrace-test/machine1/agent

get-timestamp : null

/conf/deploy/hittrace-test/machine1/monitor

collect-trace : null

If for some reason a user wishes to deploy an instance that is empty in order to facilitate development or manual debugging of some kind. There is the possibility to define a deployment that includes none of the Pod component roles. This would imply that none of the components for the specified Pod are deployed into it and instead may be provisioned with any software/configuration that is specified in a dotfile locally. This would allow a user to bring up the exact environment that the test-case is being processed on and remote into any one of the images. They may then develop the particular component they're interested in and then save it a shared directory. This will then allow them to add the tool to the project's repository, or if it's generic enough to push it into the base repository.

For each Pod that has been brought up, a unique id will be generated to represent it. There's a possibility that the unique-id might not even need to be retained or managed as it could be handled by an IaC tool or could possibly be handled by provisioning. If necessary however each Pod's deployment is keyed by its deployment name, and unique-id. Underneath this unique id is a map of image-role to machine-id or non-changing ip-address. Since machines are considered temporal and once deployed a machine should not need to be changed, this can be used to determine what machines are dedicated to a particular deployment. This can also be used to keep track of the machine count for a particular type of deployment. If necessary, this can also be used to manage deployments on a per-user basis so that one user's deployment can be made to not interfere with another. It's recommended that this would be done entirely via symbolically linking the directory in the project's repository but as such can be implemented however.

/state/client-server-fuzz/{pod-id}

image-client=machine-id,image-server=machine-id # or ipv4, or ipv6

/state/radamsa-evo-fuzz/{pod-id}

image-client=machine-id,image-server=machine-id # or ipv4, or ipv6

## Commands (Post-Deployment)

Once a Pod is defined and deployable and the Master cluster is ready to be deployed, there are only a couple of commands that should need to be exposed to a user. The main command that need to be exposed would allow a user to choose the number of Pods and a particular deployment type in order to deploy it in the cluster. Some other commands that may be added are commands for managing the services/applications for the Master cluster. Each of these commands can be implemented simply with GNU Make and perhaps a couple of helper tools.

Another DevOps tool that may be used for this kind of situation is Hashicorp's Nomad. Nomad will let one describe a "job" which Nomad will us to automatically bring nodes up and down as needed. If the job is describing a service, the user can specify how many nodes to use. If not, then Nomad can automatically schedule jobs in any Pods that are available. On the backend, Nomad can use software such as Terraform to automatically manage the infrastructure. This is unique in that Nomad may be used entirely for scheduling these jobs, and hence only a few could be bundled with the base repository.

$ make deploy type=develop count=1

$ make deploy type=fuzz-corpus-radamsa count=20

$ make deploy type=fuzz-ramdisk-custom count=100 queue=10

$ make deploy type=load-corpus count=5 options=…

### Pod management

These commands simply specify the number of Pods to use along with the particular deployment. This could be done via GNU Make and maybe a make rule named "manage" along with the deployment name and pod count as variables. The Makefile can then go through and check to see that the current store configuration of all deployments are up to date and verify that the most recent seed has been cached in the project repo. At this point, VMs can either be added or removed according to the count specified as an argument.

This can also be done using an IaC tool as previously described with or without a scheduler. Some different ways are either via an API such as SaltCloud, or even done manually using the data stored in the current master's store. During the deployment phase, a set of rules for either Terraform, or some other tool can be generated. Terraform will then automatically resolve the relationships between each node on a Pod and then update the store on the master cluster with the state of the infrastructure. However, regardless of which way is chosen, each individual pod-id is updated or removed to/from the store as described above.

### Service/Application Management

If necessary, some commands can be added in order to simplify deployment of Services or Applications into the master cluster. Each service and application in the cluster is expected to be containerized for easy management and distribution. If the master cluster is using a scheduler such as fleetd, one may simply submit a container to the cluster or update Etcd to schedule it which is easily done from the command line or GNU Make. For best results, each containerized application should define only its read-only layer and a list of directories it's expected to write to (if the containerization format does not already let you specify it). Once this is done, then the GNU Make rule can automatically build the writeable layer which can be used to easily move the Application/Service around.

These can also be deployed manually if a user chooses, or even custom software can be installed on the image directly. Unfortunately this would mean that if the Master cluster is taken down due to the project being un-deployed, unless the state is manually exported it will be lost.

$ make show-deployment type=develop

…

$ make show-deployment type=fuzz-corpus-radamsa

…

## Services/Applications

There aren't too many services or applications that should be needed by default, and so services/applications should be optional on a per-project basis. Some Services should come with the base repository so that they may be added by a user if they deem it necessary for a project. By designing them in this way, more services can be added depending on the development resources as they become more available or as a user's research expands towards processing input from a system such as this. Services are allowed to communicate with one another, and can be queried by any component within a Pod if the user so chooses to.

|  |  |
| --- | --- |
|  | Various services can be included on the master box for a project. These can provide assistance to each generator used for a Pod instance, or DNS for example. If the provisioning tool requires a centralized location, this can also be deployed inside the master. |

### Services

The default service that will need to exist would be one that stores reports from a monitor. It was suggested to use BeanStalk, but can also be anything that is chosen by the project creator. Initially a Beanstalk container could be provided in the base repository and used for storing reports or samples. Once Beanstalk is setup, it can then either be used by a generator to serve to a Pod, or an Application of some kind to present crash reports to a user by exposing a web-based user-interface. Due to it being an application, this can be done without them needing access to the project repo.

The author has chosen to use SaltStack for provisioning due to its flexibility and maturity, and so SaltStack's Master was made into a Service as well. This allows multiple members of the cluster to be provisioned in SaltStack's most efficient manner. Any provisioning tool can be used, however.

Another service that might be useful would be some kind of distributed storage. This can be SMB for maximum compatibility, or NFS which also has a client for various platforms. This service could also have an application that would allow a user to manage the storage via a web-based user interface. Some solutions will be described in the Application section.

Another optional service could be one that generates mutated files for a user-defined Generator component to pull samples from. This would allow the service to be scalable across multiple machines in the cluster and perhaps organize which types of mutations to test from based on different deployment types.

The author is not sure what would be a useful way to store this type of data, but for feeding input back into an evolutionary fuzzer a user may store the results of different hit-traces that have been fuzzed within a service. A mutator-based generator may then use this data in order to influence the way a test-case is fuzzed. As demonstrated by AFL, this can affect code-coverage in a significant manner.

A health-monitoring/scheduling service for the master cluster or the Pods can also be deployed as a containerized Service. This can be used to measure performance and if so drive a tool such as Terraform to bring up VMs according to a specific requirement as configured by the user. If the user wishes to guarantee that a certain number of test-cases are processed per second for a particular deployment, this service can check to see if performance meets the requirements of the user and if not bring up more VMs for the deployment the user had chosen.

|  |
| --- |
| rrddrbd example 1.png (2481×874) |

A DNS Server could be used for service registration. If Etcd or Consul does not satisfy the user's needs for service-discovery, a Zone may be generated from an Etcd configuration. Depending on how this is implemented this may allow a Pod component to not have to use environment variables to determine where its resources are located at.

|  |
| --- |
| microservices-architectures-with-docker-swarm-etcd-kuryr-and-neutron-9-638.jpg (638×359) |

Network Fabric or DHCP Service. CoreOS provides a service named flanneld that allows for one to encapsulate an IP Network on top of another. This might allow each member of the cluster to use an internal IP addressing scheme that will be isolated from other VMs in the virtual environment. A DHCP server can then be configured to run on this private network layer or assigned using hardware addresses. This could be used to enable a user to manage the addressing within a cluster if develop a need for it.

A KDC service could also be deployed in a container on the master if the user does not want to deploy a unique KDC within each Pod. This would simplify a network in that there'd be only one place that a Pod's client would need to communicate in order to get a TGT. This would also provide a centralized place to administrate all the Pods in a cluster that are using Kerberos for authentication.

### Applications

Applications are services that are exposed to any users w/o needing access to the project repository. These services could be things that expose an interface or API that interfaces with the cluster in some way. In most cases this would be web applications or other similar pieces of software.

An example of an application that could be exposed to the user would be a database that aggregates information from the report database and stores it in an indexable manner into a database. This way a user may then connect with their database client and query the reports however they deem fit. Using a full-text index would be recommended depending on the intermediary report format.

Beanstalkd was suggested as a service to store reports that are emitted by a monitor. If this is done, a user may develop or deploy web-based application can consume jobs off of the Beanstalk service's queue and expose it to the user in a more intuitive form. Beanstalkd may also be used to assist a generator with samples that are generated by a mutation service. At this point one could then use a console to measure how many jobs/samples are being consumed by a deployments pods.

|  |
| --- |
| https://xuri.me/wp-content/uploads/2016/11/aurora-cross-platform-beanstalk-queue-server-console-1.jpg |

Another example of an application that can be developed would be one that monitors reports that are submitted to the Beanstalk Queue and used to measure result performance. A similar service would be one that monitors the output from generators or a generator service in order to measure test-case generation performance. This can be a general service that is included with the base-repository.

In case a user needs to manage services/applications but does not prefer to use the tools provided by the command line, then a web-application that simplifies cluster service management can be exposed. There's a number of them that are available online. Again it is recommended to install these as a container so they can be moved around between different images within a cluster and more importantly between multiple projects.

# Conclusion

The suggestions described in this document describe a project that should allow multiple users to campaign against targets and share fuzzing approaches as well as retain on history of targeting a specific software. Although this is not a complete solution and solves only one minor issue of collaboration with regards to vulnerability discovery. Far too commonly, vulnerability research amongst more than one person is performed with wiki and the exchanging of notes which is non-scalable and far too often will end with “wiki-rot”. It is believed that with proper usage of DevOps tools and a modular design that does not impose a fuzzing methodology upon a researcher, the results of different methodologies towards a target can be shared and built upon. It is also expected that with a modular enough design that output from a particular target or a number of targets can be used to facilitate even more research in terms of scalable automated triaging.