Towards automated Cyber Range sensing and updating

A tentative Developer Guide

Security Governance - Sapienza

Pietro Borrello - 1647357

March 11, 2019

Contents

1 Introduction

A Cyber Range is a controlled virtual environment, which usually replicates an existing infrastructure. It is used to train targeted personnel on realistic cybersecurity scenarios to gain additional knowledge, and to test how the infrastructure would react to possible attacks.

Replicating an existing infrastructure, a cyber range can be used to test different (destructive or not) scenarios to gather information on how the hosts and the network would respond to changes in the environment, without having to worry ruining the real infrastructures, in testing patches and fixes for the various systems.

Existing works show how to automatically build a cyber range, given some textual specifics that would identify which hosts and services are needed for each machine in the cyber range [1], automatically instantiating templates with the required software installed. Therefore the initial state of a cyber range is perfectly known and controllable.

The problem arises when, due to the mutable nature of the cyber range, when testing attacks or patches in the infrastructures, the state of any machine changes in unpredictable ways, making difficult for an analyst to understand which is the current state of the environment to reason or make analysis on. Even worse, in case of any change needed on any machine, (for example starting a service in all the machines), current state of the art gives the analyst few options: either he/she manually changes the state of all machines, keeping in mind all the differences in the systems that can be active, or he/she has to re-instantiate the whole cyber range, editing template files and risking loosing the current state.

In this paper we propose how to tackle the problem of automated cyber range sensing and updating, leveraging state of the art IT automation tools, able to transparently orchestrate and monitor heterogeneous system. Being part of IT managers and sysadmin community, applying IT automation tools to a cyber range permits the analyst to leverage a centralised utility to manage easily the whole environment.

In the rest of the paper we will briefly describe the design choices that led us to the final implementation, and we will give an overview on how the system is implemented and how to use it, to let the community be able to understand and modify our implementation to better fit their goals.

2 Design Choices

The goal of the paper is designing and implementing a system that would be able to gather and update transparently the state of any system inside the cyber range. We define the state of a system as the set of:

- Installed Packages
- Active Firewall Rules
- Open Ports
- Connected Networks
- Routing Tables
- Running Services
- System Logs

So the goal will be in fact to design a system that is able to gather or modify these information. We set as an additional goal, to automatically identify the CVE that are related to the existing software in a system, to ease the process of generating the attack paths in combination with the firewall rules and routing tables.

Among the various IT automation tools, we choose Ansible [2], due to its simplicity in use, and low impact on system requirements, as needing only SSH access to systems, and relying on the Python runtime to work.

We designed an Ansible server to be placed inside the cyber range, since it could be inaccessible from the outside, that will be able to inspect and manage all the systems in the environment. The server will be able to reach, connect to and manage any system transparently, regardless of the underlying operative system, wether being Windows or any of the different Linux distribution. We define explicit support to the following systems:

- Windows 7, 8, 10
- Windows Server 2012, 2016, 2019
- Ubuntu 12.04, 16.04, 18.04
- Debian 9
- CentOS 7

- OpenSUSE 42
- Linux SuperAlpine 3.8

Compatibility with similar systems is likely, but not guaranteed. However, is highly probable that only small changes will be needed to extend the implementation to other OSes, as it will be more clear in the next sections.

3 Ansible Overview

We implemented the system leveraging the Ansible platform, designing the server that will have to collect state information on all the hosts. The Ansible platform is a orchestration engine specifically designed to manage different hosts in parallel, to ease the job of performing repetitive tasks on a great number of hosts. It is able to run the specified list of commands against the desired set of hosts transparently.

Each command issued by ansible is effectively executed by an ansible module, that is no more than a collection of functionalities to control system resources (i.e. the `package` module to install a package). A call to a module with a set of parameters is defined as a `task` (i.e. install the `vim` package through the `package` module, and collect the result).

Issuing a task to all the hosts in a set, it's just a matter of executing the following command with ansible:

```
$ ansible all -i hosts -m [MODULE] -a [ARGS]
```

A task in ansible describes the desired state to be reached after command execution so, for example, to ensure that the `vim` package is installed on all the hosts, and to install if missing, we would run:

```
$ ansible all -i hosts -m package -a "name=vim state=present"
```

Vice-versa, to uninstall a package, ensuring that all the host will reach the state of the package not being present in the system, we would run:

```
$ ansible all -i hosts -m package -a "name=nano state=absent"
```

Since it would be inefficient to restrict ourselves on running a single task at a time, ansible aggregates tasks to be executed in a playbook. A playbook describes the list of tasks to be executed against the hosts, calling the underlying modules to perform the commands. Additionally the playbook contains a name for each task, so that ansible will report a short readable summary of tasks execution. A example of a playbook could be:

```
# playbook.yaml
- hosts: all
  tasks:
    - name: Ensuring good editors are installed
      package: name=vim state=present

    - name: Ensuring evil editors are not installed
      package: name=nano state=absent
```

That describes the overall state that the system should reach at the end of the execution and Ansible will issue the right commands on each host to ensure the state is reached. A playbook can be issued simply calling:

```
$ ansible-playbook playbook.yml
```

And can additionally be extended with variables, to alter playbook execution, depending on different factors (i.e. install different packages depending on the OS)

Ansible supports roles, that are no more than a modularisation of a playbook. It's a self contained list of tasks, with their own metadata, that can be included in any playbook, to enhance the playbook functionalities, running the tasks defined in the role. A playbook can include and use multiple roles, to be able to build and combine functionalities on top of others.

Each Ansible role is defined by different parts:

- tasks - the main list of tasks to be executed by the role.
- handlers - which may be used by this role, to implement callback mechanism.
- defaults - default variables for the role.
- vars - other variables for the role.
- files - which can be statically deployed via this role.
- templates - templates which can be dynamically deployed via this role.
- meta - role metadata.

To deal with all the possible differences that the hosts ansible has to connect to, may have, Ansible automatically collects the so called `facts` at the first connection to each host. Ansible `facts` gather all the information that ansible is able to get from the host, including the operating system family and distribution, the underlying architecture, the path of the commands that will have to execute, the interfaces to connect to, and so on.

All the ansible `facts` for each hosts are available as variables during playbook execution. For example when ansible connects to a Debian host, any instance of the variable `ansible_os_family` in the task to execute will be evaluated as "debian".

The execution model of ansible can be simplified imaging that the ansible server connects in parallel through SSH to each host in the `host` file and performs the following actions:

1. collect facts for the host
2. resolve task dependencies on `facts variables`
3. execute each task in each role sequentially for the host
4. collect results

Our ansible system has to deal both with different operating system distributions (as Ubuntu, CentOS, SuperAlpine and so on) but also with different operating system families (as Linux or Windows), making the task of transparently collect the state independently from the underlying system, a challenging task. Leveraging `facts variables`, that are evaluated at runtime with the value of the corresponding fact for each host, we are able to differentiate which command to execute for each host type.

4 Implementation

We defined a `System-Collector` role, able to perform all the different tasks needed to gather machine states, and a single playbook `collect.yml` that would apply in parallel the `System-Collector` role to all the hosts in the cyber range. The goal of the `System-Collector` is to transparently collect the different features that characterise the defined state for each host. Using Ansible `facts` we defined modular tasks to be executed based on the operating system family and distribution of the target host. Due to the

fact that Ansible uses completely different modules to manage different os families, Linux vs Windows host division was based on task division. The task division was performed by the `System-Collector` including the task tailored for the os family:

```
# in tasks/main.yml

- include_tasks: "{{ ansible_system | lower}}.yml"
```

The `include_tasks` directive includes the right module either for Linux systems (`linux.yml`), or for Windows ones (`win32nt.yml`).

4.1 Linux Support

Thanks to the fact that Ansible is also able to distinguish between different linux distributions, the `linux.yml` tasks too are implemented in a modular fashion through ansible `facts`. While the sequence of actions to perform is defined by the `linux.yml` itself, the exact command to execute on the host through SSH is defined based on the `ansible_distribution fact` that will select the right command for the right operating system distribution. For example to gather the installed packages for each host in `linux.yml` we have:

```
# in tasks/linux.yml

- name: Gathering installed packages
  shell: "{{ installed_packages_cmd }}"
  register: installed_packages
```

Where {{ `installed_packages_cmd` }} refers to a variable that is included thanks to:

```
# in tasks/main.yml

- name: set os-specific variables
  include_vars: '{{ item }}'
  with_first_found:
    - "{{ ansible_distribution | lower}}.yml"
    - "{{ ansible_os_family | lower}}.yml"
    - default.yml
```

So in case of a debian system the exact command will be executed as:

```
# in vars/debian.yml

installed_packages_cmd: dpkg -l
    | awk 'BEGIN{print "Name,Version"}; FNR>5 {print $2","$3}'
    | sed 's/,[[:digit:]]\+:/,/; s/\ubuntu\|[-~+]\[^,]*$//'
```

While for example for an OpenSUSE system the value of the {{ `installed_packages_cmd` }} variable would be:

```
# in vars/suse.yml

installed_packages_cmd: rpm -qa --qf "%{NAME},%{VERSION}\n"
    | awk 'BEGIN{print "Name,Version"}; {print}'
```

Therefore we are able to transparently support any different linux distribution that ansible could connect to, through the use of ansible variables that are evaluated at runtime depending on the connected host. To add support for a new linux distribution, it is sufficient to add the right variable file in `vars/` containing the commands to execute on the host.

4.2 Windows Support

Since Ansible doesn't seem to properly identify the exact Windows version, we identified a common set of commands compatible with all the windows os that we wanted to support. We leveraged the Powershell support from Windows 7 to Windows 10, to define the proper set of commands to execute on all the hosts.

The only exception was due to the lack of support for the Powershell `NetTCPIP` module in Windows 7, due to kernel limitations. The module was tailored to gather information on the network connections, routing tables and firewall rules for the host, that was a fundamental part for the host state. We dealt with the lack, implementing a fallback mechanism. The ansible server, since it is not able to distinguish from the various windows versions, always tries to execute the Powershell module to gather state information, and when failing, due to missing modules, it will rescue the execution from the error, executing long legacy `cmd.exe` commands, with the following template

```
# in tasks/win32nt.yml

- name: Command Name
```

```
block:
- name: Command Name
  win_shell: "Powershell command"
rescue:
- name: Command Name Win7 compatibility mode
  win_shell: "cmd.exe command"
```

We avoided relying just on the legacy commands, that would have been anyway compatible with the new OSes, since they rely on parsing the output strings of commands that cannot be considered stable in the future, conversely than dealing with Powershell objects. Therefore we kept the approach restricted to stable legacy systems.

4.3 Input Format

Once we defined all the tasks for any type of host, the ansible server we designed just expects the list of hosts against which to run the information gathering. Ansible reads the inventory of hosts to connect to from an hosts file. The file has to contain, for each host, the information that ansible needs to properly connect to the host. These information include the hostname, the address of the host to connect to, the service through which ansible will connect and the username and password/private_key_file ansible will use. If the service is not specified SSH is assumed by default. The format is an host per line with the following template:

```
# in hosts

<HOSTNAME>  ansible_host=<IP>  ansible_user=<USERNAME>  \
    ansible_ssh_pass=<SSH_PASSWORD>
```

With the only caveat that windows host don't support ssh by default, so the WinRM protocol must be used, with the following syntax in case of Win hosts:

```
# in hosts

<HOSTNAME>  ansible_host=<IP>  ansible_user=<USERNAME>  \
    ansible_password=<PASSWORD>  ansible_connection=winrm \
    ansible_winrm_server_cert_validation=ignore
```

4.4 Output Format

We decided to provide cvs files to the analyst as output. So all the commands that we execute have to transform the objects or string that they collect in csv compatible files. We decided to provide csv files instead on relying on ansible internal objects, that could have been manipulated with a python module, due to the volatile nature of the internal API, that seems to change frequently at any release. A csv file seems to be the most robust and stable way to provide output.

5 Running Ansible

To run the program against a particular hosts set, described in the `hosts` file, and produce the csv files representing host state, under the `output/` directory type:

```
$ ansible-playbook -i hosts collect.yml
```

While in case of needing to execute different actions on all hosts, like installing a package, or running a shell command, run:

```
$ ansible all -i hosts -m [MODULE] -a [ARGS]
```

Where `[MODULE]` refers to the ansible module to run (i.e. `shell` or `package`) For example to run `echo foo` on all hosts:

```
$ ansible all -i hosts -m shell -a "echo foo"
```

6 Additional Task: Searching for CVEs

While developing state gathering tasks, we noticed that to compute an attack graph for the hosts involved, the list of vulnerabilities for each host was the only information missing. Since the knowledge of all the installed packages in a host is the only requirement to compute cve for the host, we developed an additional python module to obtain these CVEs.

The python module simply parses the installed packaged for each host, with their version, and tries to match them with a product and vendor to obtain a CPE [3]. A CPE its just a standard to describe a piece of software with its version. Once inferred the CPE, the module selects all the CVEs

referring that particular CPE from an online searching source [4], and sorts them based on the CVSS Score.

The module therefore is able to identify vulnerabilities from the local state of the machine, including also software not exposed in the network. This allows us to take into consideration CVE that would be impossible to obtain through a network scanning tool, as CVE referring Local Privilege Escalation.

6.1 Known Problems

The module relies on finding CVE, through CPE, that have to be matched from the package name. The problem is that such match is not deterministic, and in general is difficult to obtain a product and vendor for the CPE, just knowing the package name. The proposed solution involves building a local database of CPEs, following [5], and trying to have a best match for each particular package name with a product to obtain the vendor, so the whole operation is best effort. Any proposal to improve the matching could dramatically improve the CVE discovery capability, and a standardisation to match CPE could be beneficial for the whole process. We consider it an interesting open problem to tackle.

7 Conclusion

In this paper we presented how we designed and implemented a system to automatically collect state information from a network of heterogeneous hosts through Ansible. We showed how, leveraging ansible features, we are able to transparently manage the differences between the various operating systems and distribution to build a general description of the various hosts.

We additionally described how to improve the state description including CVE from the list of installed packages, in order to build an attack graph from the collected information.

To conclude, we believe that our approach is general enough to make it easy to extend when needing to support the analysis of additional operating system distributions, as only having to define the commands to run to collect the state, but still powerful enough to gather any required state information. So the paper may represent a good starting point for the community towards automatic cyber range management.

References

[1] CyRIS: A Cyber Range Instantiation System for Facilitating Security Training, Cuong Pham, Dat Tang, Ken-ichi Chinen, https://www.researchgate.net/publication/311621279_CyRIS_a_cyber_range_instantiation_system_for_facilitating_security_training

[2] Ansible, Red Hat, https://www.ansible.com/

[3] NIST, Official Common Platform Enumeration, https://nvd.nist.gov/products/cpe

[4] CVE-Search, GitHub, https://github.com/cve-search/cve-search

[5] CPE-Dictionary, GitHub, https://github.com/kotakanbe/go-cpe-dictionary