Global Information Assurance Certification Paper

Security through Configuration Control at Scale – An Introduction to Ansible

GIAC (GSEC) Gold Certification

Author: Patrick Neise, patrick.neise@gmail.com
Advisor: Rob Vandenbrink
Accepted: 01 Feb 2016

Abstract

The ability for companies and individuals to deploy infrastructure to cloud service providers has led to the rapid growth and visibility of numerous new products and services. While the ease and speed of deployment allows companies to quickly respond to changes in the marketplace, it also presents challenges to ensure secure deployment, configuration, and management of the supporting infrastructure. While proper use of any configuration management tool can improve the reliability and security of a deployment, the relative steep learning curve, agent based host management, and potentially vulnerable communication methods of the major offerings present additional challenges to their secure implementation. The agentless, Secure Shell (SSH) communication, Python and YAML Ain't Markup Language (YAML) based product Ansible, a relative newcomer to the field of configuration management, provides a capability that is easy to learn while providing secure and scalable implementation at scale. This paper will identify the major differences between Ansible and other configuration management tools in order to identify possible implications to information security practitioners. Additionally, previously unidentified information security specific uses for Ansible will be identified and discussed as a driver for further research.

1. Introduction

As new technologies and concepts are developed there is usually a noticeable change in the use and employment of existing technologies. For example, there is a current growth trend of concepts such as cloud computing, the merging of development and operations (DevOps), microservice based architectures, agile development, and continuous integration. These trends have provided the ability to rapidly develop, deploy, and grow product and service offerings. A byproduct of the market growth is the increase in scale, scope, and frequency of attacks against the services and their data. Many of these attacks are portrayed as sophisticated events by well-resourced attackers. The more likely scenario is that attackers are taking advantage of common misconfiguration in the deployment and operation of the services.

With spending on cloud services expected to exceed $180 billion this year, companies cite visibility, collaboration, rapid development, consistency, security, lower risk, and cost savings as primary drivers. (Hendricks, 2015) With the forecasted growth of Software as a Service (SaaS) expected to take over the majority of cloud workloads by 2018 (Columbus, 2015), the associated deployment and management of applications to support those services will also continue to grow year over year.

Although cited by many companies as a reason to shift to SasS providers (Hendricks, 2015), security in the cloud is still subject to proper configuration management. With the number of attacks continuing to increase, there has not necessarily been a similar increase in the required skill level of the attackers. Actually, according the Verizon Data Breach Investigation Report, attacks are focusing more on configuration issues and poor coding practices instead of actual software vulnerabilities (Kirk, 2010).

The combination of market growth for SaaS offerings and the continued focus of attacker on configuration errors highlights the need for secure deployment and configuration of applications and services at the scale. A simple to use, yet extensible, automation tool could prove extremely useful in not only speeding up the process of deployment and configuration, but in ensuring that deployment in done in the most secure method, every time. In the continuous integration or continuous deployment environment of DevOps, gone are the days of maintaining static configurations over long periods of

time. New configurations, new services, and even new architectures are being deployed all of the time in order to meet customer demand and to utilize new products and services.

A relative newcomer to the field of IT automation, Ansible provides a simple but powerful solution for deployment, configuration management, and orchestration. Built from the ground up to support multi-tier architectures, integrations with the major cloud service provides, and low learning curve place Ansible at the top of the list for securing deployment and configuration management at scale.

2. Ansible

At its core Ansible is an IT automation engine that is capable of meeting needs in several areas of IT management. Provisioning of cloud resources, deploying applications, configuration management, continuous delivery, security and compliance, and orchestration are several of the use cases Ansible seeks to satisfy. Agentless design, Secure Shell (SSH) based communication, and simple language place Ansible in the position of being able to support many needs of today's complex and ever changing IT environment. Additionally, the integration with current cloud providers, virtualization and containerization technologies, and major operating systems allows users to fit the capabilities of Ansible into their existing environments or ease the transition into newer technologies.

2.1. Getting Started

Nearly all major Linux distributions package Ansible in their repositories today, allowing for installation via yum or apt on your distribution of choice. Alternatively, as a Python based package, Ansible can be installed from the Python package manager, pip. Finally, the latest development version of Ansible can be cloned from the development branch on GitHub. Ansible can be run from nearly any Linux distribution with Python 2.6 or 2.7 installed.

The discussion and examples presented below are centered around the default use of SSH as the transport mechanism for Ansible. For flexibility and backwards

compatibility, Ansible provides the means to use several other transport mechanisms, as well as a pull only method that periodically grabs instructions from a git repository.

As there is no client-server model and there are no agents to install on the managed systems, installing Ansible on the control machine is really the bare minimum needed to begin employing Ansible in an environment. The only requirement for the managed node is that it must have Python installed and be accessible via SSH. On the simplest level, the controlling host makes a SSH connection to the managed node and Ansible uploads a module of commands to be executed on the managed host via Python. There is additional support for PowerShell on Microsoft Windows servers that will be discussed as well.

The fundamental components of an inventory, modules, and playbooks provide the user with all of the necessary tools to begin utilizing Ansible across various environments in support of numerous use cases. The built-in modules provide for simple interaction with various hosts, cloud service providers, software packages, etc. Additionally, the community features of Ansible provide access to user generated content that can further simplify the integration of Ansible into a user's workflow.

2.2. Components

2.2.1. Inventory

The inventory file is a collections of hosts that Ansible is aware of and can manage. The simplest version of an inventory file is a text file with a hostname per line. Although this simple text file would be sufficient to manage a large number of hosts, the ability to group hosts, define variables, and even dynamically generate an inventory provide an extremely powerful and efficient means to manage large number of diverse hosts.

The inventory files are formatted in the same manner as INI files with sections and name, value pairs. In Ansible, the section name signifies a group of hosts, such as web servers or database servers. The use of grouping allows for the same set of command to be run against all hosts within that group at the same time. Additionally, a single host

can be assigned to multiple groups at the same time allowing for fine grained control over the managed hosts.

Overriding of default configuration items, such as the username used by Ansible to connect to the remote SSH server or the name of the database to connect to on a MongoDB instance, can be accomplished within the inventory file. This becomes particularly useful when defining specific ports for services that differ from the default port setting or when using different usernames to connect to hosts and run modules. Variables can be used within the inventory file or broken out into a separate file if the number of hosts and variables makes the management within a single file to complex. The examples discussed later will maintain the hosts and variables within a single inventory file. Additionally, within the inventory file variables can be included in line with the hostname or as another section within the inventory file. The latter is accomplished by creating a new section and appending ":vars" to the end of the group name to which the variables apply.

A relatively simple, and contrived for this discussion, example of an inventory file used to deploy a web application is below. This simple example demonstrates the use of the INI format, groups, in line and group variables that could appear in an inventory file used to manage a web application deployment.

```
[database:vars]
db_name = test
db_user = mongo
[webservers]
webserver1.domain.com
webserver2.domain.com
[proxy]
proxy.domain.com ansible_ssh_port=2222
[database]
mongo.domain.com
```

Although not covered in detail here, Ansible also supports the dynamic generation of inventory files. One of the key use cases for Ansible is the management of hosts within

hosted environments such as Amazon EC2, Microsoft Azure, etc. Many of those systems or other host provisioning tools already keep track of which hosts you have deployed to the environment. By creating a dynamic inventory script Ansible can generate an up to date and accurate listing of all hosts in the environment and place them into the appropriate predefined groups as discussed above. The ability to dynamically create an inventory of hosts combined with other features to be discussed later, allows the user to specify the overall expected structure of the environment without having to worry about details such as individual IP address for each host in the system. Detailed example dynamic inventory scripts can be located in the Ansible github repository (https://github.com/ansible/ansible) under the contrib/inventory directory.

2.2.2. Playbook

With an inventory file in place Ansible now knows about all of the hosts of concern, and depending on the level of detail in the inventory file those hosts are also grouped in a manner that allows finer grain control of what tasks to execute against each host. The list of tasks to execute against a particular host are contained within what Ansible refers to as a playbook. A playbook can be simply thought of as a configuration script that runs a predetermined set of tasks against a set of hosts. The power in playbooks comes from the simplicity, modularity, and extensibility provided by the overall Ansible approach to configuration management.

For ease of readability and modularity, playbooks are written in YAML (YAML Ain't Markup Language) syntax. While YAML is very similar in structure to Javascript Object Notation (JSON), the noticeable difference is in the human readability of the format. At a high level, playbooks are a list of plays, and plays are a list of tasks to run against specified hosts. The list of lists structure is suited extremely well for description in a YAML formatted syntax and provides a repeatable structure for building playbooks.

Each play must contain a set of hosts and a list of tasks to execute against those hosts. While identifying hosts and tasks is the minimum requirement, optional items such as a name and variables, or vars, provide increased functionality and readability/feedback for the user.

As discussed in the previous section, the hosts field can reference individual hosts from the inventory file or more typically the group name of a set of hosts within the inventory. By specifying a group name in the playbook, Ansible will run the specified tasks against all of the hosts in the group. Although not required, the name field is extremely useful in that it contains a description of what the play accomplishes and Ansible will print out the name field to the console when running the playbook. Another very useful option is vars, or variables, which is a list of variables and their values. Variables will be discussed in more detail in the examples later to demonstrate the ability to pass information to the playbooks for modularity as well as for security reasons.

The major remaining piece of a play is the list of tasks to be executed against the target hosts. Tasks must consist of a module name, to be discussed in the next section, and the values of the argument to be passed to the module. Tasks, just like plays, can have a name field as well. It is also highly recommended to name tasks for readability and understanding of what the task accomplishes. Additionally, Ansible provides the ability to start a play at a particular task and skip the preceding tasks. This functionality requires the task to be named.

A brief playbook is shown below as discussed in the Ansible documentation. This playbook contains only one play with three tasks related to configuration of a web server. The modules introduced in the playbook will be discussed in the next section.

```
---
- name: Configure apache webserver
  hosts: webservers
  vars:
    http_port: 80
    max_clients: 200
  remote_user: root
  tasks:
  - name: ensure apache is at the latest version
    yum: name=httpd state=latest
  - name: write the apache config file
    template: src=/srv/httpd.j2 dest=/etc/httpd.conf
    notify:
    - restart apache
  - name: ensure apache is running (and enable it at boot)
    service: name=httpd state=started enabled=yes
  handlers:
  - name: restart apache
    service: name=httpd
    state=restarted
```

The above playbook demonstrates all of the concepts and syntax covered so far while introducing a couple of new concepts such as modules, templates, and handlers. The only play in the playbook configures apache webservers and targets hosts in the webservers group. For consistency, this playbook references the previous inventory file example that also contains a webservers group, meaning that in this example all of the tasks would be run against webserver1.domain.com and webserver2.domain.com each time the playbook is run.

Before describing the tasks above, it is important to point out that Ansible tasks are idempotent. This means that tasks define the desired end state, not how to arrive at that end state. For example, a task may specify that a particular package must be installed on the target host. If that package is already installed Ansible takes no action, otherwise

the package is installed to place the host in the desired state. This means that playbooks can be run multiple times against the host and the only items executed on the host are those that differ from the desired state specified in the tasks.

The three tasks in the example playbook above demonstrate the relatively simple process of ensuring that the version of apache installed on the webserver is the latest version, copying the configuration file from the controlling host to the target, and starting the httpd service. This simple example is a great representation of idempotency in action. If the version of apache is up to date, nothing occurs, otherwise the most recent version of apache is installed via the yum package manager. Next, if the configuration file on the target host is already the same as that on the controlling server, nothing happens. If the file does not exist, or is different, the file is copied to the host and new concept of notifying a handler is introduced.

The combination of notify and handlers allows the user to group particular actions that need to occur at the end of the playbook, and only if there is a need to perform the function. In this example, the apache server will be restarted by the handler only if the configuration file on the host has been updated. With the potential for several files to be copied or other changes made to the server that would require a service restart, the concept of handlers allows the playbook to restart the service only once, and only if required by the actions taken in previous tasks. The final task ensures that the apache server is running at that the service will be started whenever the host is rebooted.

Error handling within Ansible is designed to fail fast by default. If an error occurs, it must be dealt with unless decided otherwise within the playbook. However, not properly dealing with failures for a task that occurs early in a playbook may result in undesired results. By default, playbook execution will stop for a host that has a failure, but tasks for other hosts within the playbook will continue. While failing fast will identify issues early, the host may be left in an unstable state. For example, if a task that installs several software packages fails, the resulting application or service may not correctly function. Ansible provides several options and techniques to plan for and deal with possible errors that may occur during task execution.

At this simplest level the tasks can be designed to just ignore any errors by adding "ignore_errors: yes" to the task. While this solution is not ideal, it will allow the remainder of the tasks for that host to complete. This can be useful in testing, but in practice can cause cascading errors, such as not setting a variable that gets used further into the playbook. Similarly, if there were any notification to handlers during playbook execution, the handler would not execute by default if any task for the host fails. This default behavior can be overridden by setting "force_handlers" to "True". This technique can be used to restart a service after a configuration change, even if a subsequent task within the playbook for that host fails. Finally, the playbook can include the "fail" module in order to generate custom messages when the playbook fails based on certain conditions being met.

2.2.3. Modules

While playbooks can be thought of as the structure of what tasks to accomplish, modules are the components that actually do the work on the target host. The module is what actually gets executed in each task of the playbook. In addition to being executed from within playbooks, modules can be executed from the command line with the 'ansible' command:

```
ansible webservers -m ping
ansible dbservers -m command -a "/sbin/reboot -t now"
```

The first command above will execute the ping module against all of the web servers in the inventory and report the results to the screen while the second command would reboot all database servers in the inventory.

Ansible ships with a very robust set of core modules covering topics from cloud services to system level commands. Modules can install a package, restart a service, copy a configuration file, or create and image in Amazon ec2. In addition to the core modules, Ansible also currently ships with extra modules that provide additional functionality, the main difference being that extras are largely maintained by the Ansible community while the core modules are maintained by the Ansible team itself.

Two important concepts to cover regarding modules are that modules are declarative and idempotent. Being declarative means that modules are used to describe the desired end state that the host is required to be in after module execution. In other words, the module just states what the host should look like in the end, but not how to get there. For example, in order to ensure that a particular user account is present on the host, the following task that calls the user module could be added to the playbook:

```
user: name=apache group=web
```

The above task would ensure there is a user named 'apache' in the group 'web' on the target host, without specifying how to add that user to the host. The second, and very nice property of Ansible, is that modules are idempotent. In the above example, being idempotent means that if the 'apache' user did not exist then the module would create the user. However, if the user already exists, then the module will do nothing. The significance of being idempotent means that playbooks can be run as many times a desired against a host without undesired consequences due to Ansible only making changes required to put the host in the desired end state. In other words, even with an extremely large playbook of multiple tasks, nothing on the host will change if it is in the correct end state. This is significantly different than running bash scripts on a host repeatedly, where the starting state of the host can have a direct impact on the successful execution of the script.

Although not covered in detail here, custom modules can also be developed for cases where using the 'command' module is too complicated or there is not currently a module that meets the user's need. Custom modules can be written in any language, however writing modules in Python is simpler due to the increased functionality provided by Ansible, which is also written in Python. For example, Ansible makes writing Python based modules easier by providing a base 'AnsibleModule' Python class to begin building the module.

The basic flow for all modules, specifically Python modules in the case, are listed below.

1. Create a Python script with the arguments passed to the module.

2. Copy the module to the target host
3. Invoke the module (Python script) on the host
4. Parse the standard JSON output

While this basic structure is the same for modules written in other languages, the biggest difference is that a separate arguments file must be created and passed to the host. Finally, there are expected parameters that the module should return, and the output must be in correctly formatted JSON for Ansible to properly parse the results. Further information on exactly how to create custom modules in available in detail in the Ansible Docs on the Ansible website as well as other third party resources.

2.2.4. Roles

As playbooks grow in size and complexity due to increased number of tasks, variables, files, templates, etc. the management of the playbook across numerous hosts and tasks can become extremely difficult. To simplify writing and managing complex playbooks Ansible provides the ability to create roles in order to break apart the playbook into multiple files. Roles can be thought of as a grouping of information needed to configure a particular service or function. For example, roles can be created for webserver and database_server in order to carry out all of the steps necessary to properly configure a web server or database server. The role is a grouping of the tasks, files, templates, variables, and handlers necessary to properly configure the desired functionality.

By grouping all of the information necessary into a single location it is significantly easier to track and ensure that specific dependencies for different requirements are met. To build on this concept, roles can actually be dependencies for other roles. For example, an 'application_server' role can be created that properly configures a RedHat Linux host with all of the necessary configuration and dependencies except for the specific Java Development Kit (JDK) required to run the application. To complete the configuration for a specific application, additional roles for 'jdk7' and 'jdk8' can be created that depend on the 'application_server' role. This concept allows

the user to group common functionality, such as configuring ntp, into a role than can be called by any other role when configuring a target host.

Management of roles consists of creating a folder for each role that in turn contains folders for tasks, files, templates, etc. The role can then be added to a playbook in order to be run against the specified hosts. Additionally, just like modules, arguments can be passed to the role in the playbook to provide an additional level of granularity and control. An example snippet of a playbook with a role is shown below.

```
- name: deploy db server
  hosts: db
  roles:
  - role: database
    database_name: "{{db_name}}"
    database_user: "{{db_user}}"
```

The above playbook simply calls the 'database' role against the db inventory while passing the database name and user as variables to the role. The 'database' role would now execute with all of the associated tasks, variables, files, etc. as previously discussed. Roles could be created for the application server, web server, and proxy server in a similar manner, resulting in a much simpler high level playbook for deploying the entire infrastructure.

2.3. Example Use Case – Securing SSH

With the base functionality of Ansible as described above, a simple example can now be discussed from start to finish. As it will be relevant for discussion in the following section, this example will walk through securely configuring SSH servers in the environment. While this is not a particularly difficult task in the sense of the complexity required from Ansible, it will demonstrate the power and ability of Ansible to simplify a rather complex and tedious process at scale in order to secure SSH servers across an environment. Additionally, while this example walks through securing SSH servers, it is not intended to be a defining standard for secure SSH configuration. There are numerous resources available that focus solely on securing SSH servers.

2.3.1. Inventory

An inventory file for the example.com domain is included below. Although contrived for demonstration purposes, this inventory file demonstrates several of the concepts discussed in the previous section. As there are multiple web servers and database servers, they have been grouped according to their function. The grouping allows all web servers to be simply identified by the 'webserver' group.

```
mail.example.com

[webservers]
web1.example.com
web2.example.com

[database]
db1.example.com
db2.example.com

[ssh_servers:children]
webservers
database
```

The additional concept of groups referencing other groups is demonstrated in this example to group all of the servers with SSH access enabled into a single 'ssh' group. The 'children' qualifier at the end of the 'ssh' group name tells Ansible that the names refer to other groups within the inventory file vice individual hosts. Referencing groups within groups provides yet another level of granularity to the control of numerous hosts across an enterprise level deployment.

2.3.2. Playbook

The following playbook will ensure that the target hosts have the desired secure configuration for the running SSH server. At a high level, the playbook will:

1. Disable password based SSH authentication
2. Disable root account remote login

3. Explicitly allow SSH access for ansible user
4. Add the ansible user identity key to ~/.ssh/authorized_keys

Additionally, the playbook will introduce a few advanced features within playbooks including looping over similar tasks with different values and file lookups.

```
- hosts: ssh_servers
  vars:
    -  ssh_user: "ansible"
    -  ssh_identity_key: "/home/ansible/.ssh/id_rsa.pub"
  tasks:
    - name: Secure remote SSH login.
      lineinfile:
         dest: /etc/ssh/sshd_config
         regexp: "{{ item.regexp }}
         line: "{{ item.line }}"
         state: present
      with_items:
         - regexp: "^PasswordAuthentication"
           line: "PasswordAuthentication no"
         - regexp: "^PermitRootLogin"
           line: "PermitRootLogin no"
      notify: restart ssh
    - name: Allow specific users to SSH.
      lineinfile:
      dest: /etc/ssh/sshd_config
      regexp: "^AllowUsers"
      line: "AllowUsers ansible"
      notify: restart ssh
    - name: Copy user identity to host.
      authorized_key: user={{ ssh_user }}
          key="{{ lookup('file', ssh_identity_key) }}"
      notify: restart ssh
  handlers:
    -  name: restart ssh
       service: name=ssh state=restarted
```

Patrick Neise, patrick.neise@gmail.com

Working through each of the sections of the playbook:

hosts – Specifies the ssh_servers from the inventory file.

vars – The variables define the SSH username and identity file to use within the tasks

tasks – Defines each of the tasks to be accomplished on the hosts. In this section the looping and lookup concepts are introduced. In the first task, the lineinfile module is used to update particular lines within the SSH config file. By defining the 'regexp' and 'line' as templated variables, the section that begins 'with_items' will run the lineinfile module with the values of the variables specified. The following task also uses the lineinfile, but with predefined variables.

In copying the identity file to the target hosts, the lookup plugin is used to find the identity file in order to upload the file to the host using the authorized_key module.

Finally, each of the tasks notifies the 'restart ssh' handler in order to restart the SSH service after any configuration changes have been made. Of note, the SSH service will only be restarted if any of the tasks make changes on the target host.

handlers – Defines the handlers, in this case a handler to restart the SSH service.

Although the tasks in the playbook are not every step necessary to ensure a secure an SSH server, they demonstrate the simplicity and power that Ansible can provide when securing hosts and services at scale.

2.4. Security Concerns

While Ansible can be employed effectively to secure hosts, just like many other tools there are security concerns with the proper and secure use of Ansible. Although Ansible uses SSH for secure communication between the controller and the target hosts, improper configuration of the SSH server on the hosts can lead to security vulnerabilities with the hosts. As shown in the previous example, Ansible can be used to secure the SSH server on the target hosts. In fact, this should be one of the first tasks accomplished on any host that will be managed with Ansible to ensure all future connections will be

conducted via secure means, and the SSH server can be configured to only allow connections from the Ansible controller.

Additionally, when using Ansible to configure servers and deploy applications the user may be required to store sensitive information such as passwords for accounts or SSH deploy keys. Placing sensitive information into variables files would preclude placing Ansible files into a version control system due to the risk of unintentional disclosure of the information. The Ansible vault uses shared secret AES encryption to store sensitive files along-side the unencrypted remaining files. At runtime, the '--ask-vault-pass' command line option passed when running the playbook will prompt the use for the vault password, at which time the files will be decrypted and used in execution of the playbook.

2.5. Security Application

With an understanding of the components and workflows involved in using Ansible for deployment, configuration management, and orchestrations, many potential uses for Ansible within the security community begin to become apparent.

2.5.1. Enforcement of Fundamentals

Simply employing the core functions of Ansible in an environment is a significant step towards controlled, repeatable, and scalable configuration management. Spending the time to develop playbooks that add the required secure configuration to the deployment plans of operations personnel will ensure that every time a new asset is deployed into the environment, it will be configured in a manner consistent with security best practices. The SSH configuration example above represents just a small portion of the amount of configuration that can be accomplished on each host or application in ordered to ensure security from the beginning. The security team can work hand in hand with the operations team in development of playbooks, or even create security specific roles to be included into operations deployment playbooks.

2.5.2. Training Environment

As a variation on using Ansible for orchestration and deployment of a production environment for applications and services, one very interesting use of Ansible is in the

creation of training environments for security personnel. Virtualization is already being heavily leveraged in training environments, but the configuration and management of those environment is tedious and error prone. Additionally, entering changes into the environment can introduce errors and other complications. Using Ansible in combination with virtualization or containerization technology, playbooks and roles can be created to rapidly and reliably create environments for all aspects of security training.

A full capture the flag environment could be created with production quality examples of enterprise services, complete with typical misconfiguration and simulated sensitive data. Additionally, the ability to change flag values and immediately re-deploy the environment allows the creators of the environment to change the goals every time the environment is used.

Another possibility could be the creation of a production like environment that simulates a post breach scenario for incident responders to evaluate. And similar to the construct for capture the flag, the playbooks could be configured to deploy the exact same environment with different indicators of compromise in order to gain full training value out of the environment.

There are many other possible training environment applications that could benefit from the use of Ansible for configuration management, deployment, and orchestration. Additionally, the training team could leverage playbooks and roles created and used by the operations team in order to create near identical environments to those currently being used in production.

Of note, the Software Engineering Institute at Carnegie Mellon University is using Ansible and other DevOps tools in order to generate very large training environments for security personnel (AnsibleFest San Francisco, 2015).

3. Conclusion

With the continuous growth of concepts and technologies including microservices, containerization, continuous integration, etc., the need for and importance

of configuration management and orchestration at scale directly impacts the level of security in production environments.

Ansible, although a relative newcomer to the field, provides a simple to learn, scalable, and secure solution to management environments at the scale and speed necessary for today's demand signal. From the lockdown of deployed services and applications to the creation of training environments for security personnel, and nearly everything in between, Ansible is one of the tools to for security, operations, and development teams to add to their existing tool chains.

Finally, with the support and innovation of the Ansible community new modules, roles, and even completely new use cases are being implemented to configuration management, deployment, and integration at scale.

Patrick Neise, patrick.neise@gmail.com

References

AnsibleFest San Francisco. (2015, 12 24). Retrieved from Ansible: http://www.ansible.com/ansiblefest-videos-sei-sf15

Columbus, L. (2015, 1 24). Roundup of Cloud Computing Forecasts and Market Estimates. Retrieved from Forbes / Tech: http://www.forbes.com/sites/louiscolumbus/2015/01/24/roundup-of-cloud-computing-forecasts-and-market-estimates-2015/

Fidao, C. (2015). Servers for Hackers. N/A: Leanpub.

Fowler, M. (2014, 3 25). Microservices. Retrieved from martinfowler: http://martinfowler.com/articles/microservices.html

Geerling, J. (2015). Ansible for DevOps. N/A: Leanpub.

Hall, D. (2015). Ansible Configuration Management - Second Edition. Packt.

Hendricks, D. (2015, 5 27). Spending Growth in Cloud Computing is Astounding: Here's 7 Reasons Why. Retrieved from Inc.: http://www.inc.com/drew-hendricks/spending-growth-in-cloud-computing-is-astounding-here-s-7-reasons-why.html

Hochstein, L. (2015). Ansible Up & Running. Sebastopol: O'Reilly.

Jackson, B. (2015, 8 31). Why Security Configuration Management Matters. Retrieved from tripwire: http://www.tripwire.com/state-of-security/security-data-protection/security-configuration-management/why-security-configuration-management-matters/

Kirk, J. (2010, 7 29). Data breaches exploit configuration erros, not software vulnerabilities. Retrieved from InfoWorld: http://www.infoworld.com/article/2625548/intrusion-detection/data-breaches-exploit-configuration-errors--not-software-vulnerabilities.html

Prince, B. (2012, 05 21). Massive Data Breach in Utah State Servers Caused by Configuration Errors. Retrieved from eWeek: http://www.eweek.com/c/a/Security/Massive-Data-Breach-in-Utah-State-Servers-Caused-by-Configuration-Errors-851037

Patrick Neise, patrick.neise@gmail.com

Various. (2010). 2010 Data Breach Investigations Report. Retrieved from Verizon Enterprise: http://www.verizonenterprise.com/resources/reports/rp_2010-DBIR-combined-reports_en_xg.pdf

Ylonen, T., Turner, P., Scarfone, K., & Souppaya, M. (2015). NISTIR 7966: Security of Interactive and Automated Access Management Using Secure Shell (SSH). Gaithersburg: NIST.

Patrick Neise, patrick.neise@gmail.com