unir
LA UNIVERSIDAD DE INTERNET

FORMACIÓN PROFESIONAL

UNIDAD FORMATIVA 5

Servidores

Configuración de sistemas

Índice

Configuración de sistemas

En la administración de sistemas, muy rara vez tenemos un único sistema que mantener en el tiempo, siendo lo habitual que tengamos que **gestionar la configuración** de una flota de servidores, de modo que puedan seguir dando servicio en un entorno de requisitos cambiantes y que evolucionan con el tiempo.

Desde el advenimiento de las tecnologías cloud y las tecnologías de virtualización, a este proceso habría que añadir **la gestión automatizada** del software de esos servidores, de modo que se puedan tanto crear máquinas preconfiguradas desde cero como actualizar aquel SW que ya esté instalado pero necesite una versión distinta por el motivo operacional que sea.

Y ya por último, debemos considerar los escenarios que se nos presentan cuando usamos tecnologías como Kubernetes, o pretendemos aunar los diferentes niveles de servicio que tenemos en los servicios de cloud para conseguir un único propósito: en este caso se dice que tenemos que **orquestar** servicios, haciendo uso de herramientas y tecnologías muy diferentes para poder hacer que servicios con interdependencias entre sí puedan converger a un estado final específico.

Como sysadmins, tenemos a nuestra disposición tecnologías y software que nos ayudarán a conseguir estos objetivos. Además, como DevOps, sabemos que debemos aplicar donde sea posible conceptos de mantenibilidad, reutilización, legibilidad y fiabilidad.

No hay excepciones en la incorporación de controles de seguridad a un entorno DevOps. Sí hay diferencias esenciales en cuanto a su aplicación, ya que la velocidad de iteración y las herramientas disponibles son diferentes. Por tanto, cada vez hay más organizaciones que integran la seguridad en la metodología DevOps desde las primeras etapas. La segunda mitad de la asignatura profundizará en esta integración.

Objetivos

- Aprender cómo usar un script para automatizar tareas sencillas.
- Estudiar qué herramientas nos proveen de gestión automatizada de la configuración.
- Entender el concepto de SecDevOps o securización de sistemas y procesos en DevOps.

Scripting en servidores

Una de las herramientas que históricamente ha acompañado a la administración de sistemas es el script de shell: se llaman así porque usualmente se escriben usando algún lenguaje de shell como ***sh***, ***bash***, o similares. Pero no se limita a estos y nos podríamos encontrar scripts en ***Powershell*** en sistemas Windows o incluso scripts en lenguajes más potentes como Python o Perl o, incluso, lenguajes compilados como *Go*.

Existen infinidad de scripts para muy diferentes propósitos, muchos de los cuales ya vienen instalados en cualquier sistema operativo y se emplean, por ejemplo, para gestionar servicios durante el arranque. Usos habituales serían:

- Gestionar tareas repetitivas para reducir error humano, como podría ser un procesamiento de logs.
- Parar / lanzar / reiniciar un servicio, aunque veremos que lo más cómodo es aunar todo en un único fichero.
- Secuenciar los pasos necesarios para construir o probar una aplicación, para su invocación desde sistemas de integración continua sin intervención humana…

En esta sección veremos ejemplos de estos tres escenarios, aunque debemos entender que existen muchos, muchos más escenarios posibles, lo que genera muchos **escenarios de oportunidad** a un sysadmin para aportar valor automatizando tareas que ahorren tiempo y eviten errores en operativas habituales.

Tipos de scripts

A muy grandes rasgos, podemos crear scripts de dos tipos:

- Muy específicos, enfocados en hacer una única tarea.
- De propósito general, que admiten una serie de **parámetros** que amplían su uso a muy diferentes tareas.

Evidentemente, entre ambos tipos hay una serie de casos intermedios que dependen del problema que estemos intentando resolver. La **filosofía UNIX** (por el sistema operativo) es aquella que pone en valor scripts especializados, que se enfocan en resolver una sola tarea (y, por tanto, es más probable garantizar que se hace bien). Para conseguir objetivos más complejos se usará **composición** de estas herramientas, de manera que la salida de unos se convierta en la entrada de otros.

Parseo de logs

Vamos a crear un script para resolver el siguiente problema:

> *'Queremos saber las 5 IPs que más códigos de error HTTP reciben'.*

Los servidores web, como Tomcat, *nginx* o Apache, pueden dejar logs de su actividad en un fichero concreto, **rotado** frecuentemente o cuando pasa de un tamaño determinado, que nos dan información sobre todas las peticiones entrantes (*access.log)* o los errores encontrados durante la ejecución *(error.log)*.

Lo primero que tenemos que hacer es encontrar una muestra de uno de esos ficheros. Este es un ejemplo del contenido de un log de acceso de un servidor Apache:

```
$ cat access.log
1.202.218.8 - - [20/Jun/2012:19:05:12 +0200] "GET /robots.txt HTTP/1.0" 404 492 "-" "\"Mozilla/5.0"
208.115.113.91 - - [20/Jun/2012:19:20:16 +0200] "GET /logs/?C=M;O=D HTTP/1.1" 200 1278 "-" "Mozilla/5.0
(compatible; Ezooms/1.0; ezooms.bot@gmail.com)"
```

Como podemos ver, después de la cadena ***HTTP/1.0"*** está el código HTTP que necesitamos. Si queremos saber cuáles de estos son errores, debemos buscar aquellos que sean **distintos de un HTTP 200** y contarlos (en HTTP se considera error cualquier código de error mayor o igual a 300, aunque los códigos HTTP 100 apenas se usan).

Para resolver este problema, vamos a usar un fichero de ejemplo ubicado en github y lo haremos ejecutando comandos desde la línea de comandos (más rápido para probar).

```
$ curl -o access.log
'https://raw.githubusercontent.com/misterarias/examples/master/Common%20Data%20Formats/apache_logs/apache_logs'
$ grep  -v 'HTTP\/1\.." 20.' access.log  \  # filtrar mensajes OK
  | awk '{print $1}' \   # nos quedamos solo con las IPs
  | uniq -c \          # contamos las ocurrencias de cada IP
  | sort \             # ordenamos por el número de ocurrencias
  | tail -5            # Devolvemos las N=5 últimas
  27 144.76.194.187
  27 199.168.96.66
  30 75.97.9.59
  38 130.237.218.86
 147 75.97.9.59
```

Hemos empleado los siguientes conceptos/comandos:

- grep **-v** <cadena> <ruta> busca las líneas que **no contienen** <cadena> en el fichero o directorio.
- 'HTTP\/1\.." 20.' es una **expresión regular**, que *matchea* cualquier versión 1. "algo" de HTTP y cualquier código 200 válido (200, 201, 202).
- awk '{print $1}' - se queda con el primer campo, usando el espacio como separador. Separar por campos en bash es un fastidio, por eso se recurre a herramientas externas como *awk* en estos casos, aunque se podría usar *cut* o *sed.*
- El operador pipe (|), que toma la salida de un comando y la usa como entrada al siguiente.

Si quisiéramos convertir esto en un script, para cualquier fichero de logs de entrada, haríamos las siguientes modificaciones:

```
$ cat logger.sh
#!/bin/bash
ENTRADA=${1:?Missing input file}

grep -v 'HTTP\/1\.." 20.' "${ENTRADA}" | awk '{print $1}' | uniq -c | sort | tail -5
```

En la línea 2 creamos una **variable**, cuyo valor es el **primer ($1)** argumento que se le pase al script. Con la construcción *:?* de Bash (sabemos que usamos bash por el *shebang* de la primera línea) hacemos que el script falle si no hay valor para el argumento de entrada.

Para usar una variable, simplemente la precedemos de un dólar (***$***), aunque además la estamos rodeando de llaves (***{}***). Debemos acostumbrarnos a esta sintaxis, pues facilita la composición de variables.

Gestión de servicios

Como vimos en el tema anterior, usando herramientas como systemd, Upstart o sysctl podemos gestionar los diferentes servicios que se ejecutan en una máquina: podremos pararlos, reiniciarlos o hacer que se reinicien para, por ejemplo, leer una nueva configuración. Vale la pena pararse a ver cómo lo hacen, pues el formato es algo que podemos reusar en scripts de propósito general.

Si tenemos una máquina Ubuntu disponible (con Vagrant, por ejemplo), veamos el contenido de un sencillo fichero de arranque de un servicio:

```
$ cat /etc/init.d/cryptdisks
#! /bin/sh
### BEGIN INIT INFO
# Provides:          cryptdisks
# Required-Start:    checkroot cryptdisks-early
# Required-Stop:     umountroot cryptdisks-early
# Should-Start:      udev mdadm-raid lvm2
# Should-Stop:       udev mdadm-raid lvm2
# X-Start-Before:    checkfs
# X-Stop-After:      umountfs
# X-Interactive:     true
# Default-Start:     S
# Default-Stop:      0 6
# Short-Description: Setup remaining encrypted block devices.
# Description:
### END INIT INFO
```

```
set -e

if [ -r /lib/cryptsetup/cryptdisks-functions ]; then
        . /lib/cryptsetup/cryptdisks-functions
else
        exit 0
fi

INITSTATE="remaining"
DEFAULT_LOUD="yes"

case "$CRYPTDISKS_ENABLE" in
[Nn]*)
        exit 0
        ;;
esac

case "$1" in
start)
        do_start
        ;;
stop)
        do_stop
        ;;
restart|reload|force-reload)
        do_stop
        do_start
        ;;
force-start)
        FORCE_START="yes"
        do_start
        ;;
*)
        echo "Usage: cryptdisks {start|stop|restart|reload|force-reload|force-start}"
        exit 1
        ;;
esac
```

En este caso es un script de arranque del servicio de encriptación del disco duro presente en Ubuntu, que se ejecutará automáticamente al inicio del SO, ya que está ubicado en la carpeta *init.d* y, además, cuenta con ciertos metadatos de que el sistema operativo (a través de SystemV) utilizará para saber cuándo debe arrancarlo.

Lo que nos introduce este script es lo siguiente:

1. **Bloque *case/esac***, que es el equivalente al bloque *switch/case* de Java, pero con un potente *matcher* basado en expresiones regulares. En este caso, procesamos una serie de comandos (*start, stop…)* y en su ausencia mostramos un útil mensaje con información al usuario de cómo ejecutar el script.
2. Instrucción ***dot (.)***, que lo que hace es añadir el código fuente del fichero que se le pasa como parámetro al script actual. Actúa como un **import** de otros lenguajes y, en este caso, se usa para poder invocar funciones presentes (*do_start, do_stop…)* en otro fichero.
3. Un ejemplo de **condicional bash**.

```
if [ -r /lib/cryptsetup/cryptdisks-functions ]; then ... fi  .
```

Nos vale para dar un error en caso de que no exista el fichero y no sea legible. Esta información no tenemos por qué tenerla siempre en mente y la podemos consultar al momento usando *man* (el comando *test* es un sinónimo de la sintaxis *[…] ;*)

```
$ man test
```

Herramientas de automatización

Ahora que conocemos estos scripts podríamos pensar en el siguiente paso: cómo poder ejecutar estos scripts de manera automatizada o cómo hacerlo en una serie de servidores de forma automática o semiautomática.

Si bien podríamos utilizar el protocolo SSH para enviar comandos a todas las máquinas bajo nuestro dominio, más tarde o más temprano empezaríamos a encontrarnos con problemas de toda índole:

- La gestión adecuada de usuarios y de sus claves es un problema que no escala.
- Debemos ejecutar periódicamente tareas de actualización del software.
- Las configuraciones de las máquinas pueden empezar a diverger si no se controlan.
- Diferentes máquinas tienen diferentes necesidades de versiones de software, lo que hace que el script que va en un grupo pueda no funcionar en otro.
- Problemas físicos pueden hacer que algunos procesos dejen de funcionar.

Sin duda, el problema de la gestión de la configuración es un quebradero de cabeza y la razón de la 'mala fama' de la tarea de los *sysadmin*, al ser procesos muy **tediosos**, donde los errores tienen **gran impacto** y pueden ser muy difíciles de **diagnosticar** y **solucionar.**

Para ayudar en estas tareas existen paquetes de software que nos ayudan a gestionar estas **configuraciones como código**, automatizar las tareas de instalación y mantenimiento y, en algunos casos, a orquestar las tareas y la creación de servicios intermedios necesarios para llegar al estado necesario.

Idempotencia

Una de las características más deseables de cualquier herramienta de gestión automatizada de la configuración de nuestros sistemas es la llamada **idempotencia de las operaciones.**

Este concepto se refiere a que se espera que la repetida aplicación del mismo cambio o de la misma operación al sistema no haga cambios si este ya se encuentra en el estado deseado. Es decir, que se puede aplicar cuantas veces sea necesario, que la herramienta nos garantiza que no se harán cambios.

Esto es una característica muy deseable, pues no solo nos permite ganar tiempo, sino que nos da la fiabilidad de que no se van a modificar datos de forma inesperada o innecesaria por los scripts y herramientas que se empleen.

Ejemplos de idempotencia

- Al instalar una base de datos SQL y crear un esquema por defecto, no querremos que la herramienta borre lo que hay en cada aplicación para recrearlo de nuevo, pues podríamos perder datos.
- Al instalar paquetes en un sistema Linux, no querríamos volver a descargarlos cuando ya se encuentren fijados a la versión adecuada (*version pinning*, en inglés) para evitar un trasiego inútil de datos.
- Cuando tenemos que levantar una serie de sistemas en secuencia, **puede que** no debamos tirarlos y empezar de cero en cada ejecución.
    - Esto no siempre es cierto: existe una técnica de despliegue muy común en las clouds, llamada la ***infraestructura inmutable***, en la que siempre que se saca una nueva versión de un software, se levanta **desde cero** toda la infraestructura necesaria y, en cuanto se tienen garantías de que esta funcionando OK, se **decomisa** toda la anterior.

Automatización y Orquestación

Estos dos términos se suelen emplear de modo casi intercambiable, cuando en realidad no son equivalentes.

La **automatización** de tareas se refiere al uso de técnicas y mecanismos que nos ayudan a eliminar las consecuencias del factor humano en las operaciones: errores, tareas repetitivas, entornos dispares… Mediante la aplicación de disciplinas como **infraestructura como código,** automatizaciones, revisiones por los *peers* del código que gobierna las operaciones habituales, y el empleo de herramientas para poder escalar las tareas, se consiguen resultados impresionantes que eran quizá impensables hace unos pocos años no sin una gran disciplina y mucho trabajo de muchos profesionales.

La **orquestación** va quizá un paso más allá y está ligada a la complejidad que entraña el manejo de las clouds y de los entornos de manejo y despliegue de contenedores, como Kubernetes. En estos entornos, la cantidad de interdependencias entre elementos crece a medida que añadimos complejidad a nuestros entornos. Si tomamos el símil de una orquesta, el administrador de estos sistemas actúa sin dudas como el director de la misma, eligiendo qué elementos deben desplegarse en cada momento, en qué cantidad y con qué parámetros, para dar paso a los siguientes que dependen de ellos, y así hasta converger al estado deseado.

Actualmente, las tecnologías que se emplean para la gestión automatizada de la configuración **no tienen por qué** ofrecer también orquestación de los despliegues, pero comienza a ser así: tanto Chef como Ansible así lo pregonan en sus webs (en Chef, en la versión de pago), pero hay muchas más herramientas que las que veremos aquí que si que prometen conseguir la orquestación total.

Herramientas

La primera herramienta que se considera pionera de la gestión de la configuración fue CFEngine. Sin embargo, los que mayor popularidad alcanzaron fueron Puppet y Chef, y por ello todavía hoy en día se comparan habitualmente con Ansible.

Según Heap, Puppet y Chef son herramientas más similares entre sí de lo que lo son con Ansible, aunque todas ellas realizan funciones similares. Tanto Puppet como Chef se basan en el **uso de un servidor centralizado** para gestionar todo lo relativo al estado de la configuración requerido de los hosts y sus metadatos relacionados. Ansible, por el contrario, no necesita utilizar un servidor centralizado al que se conecten los agentes desplegados en las máquinas a gestionar, ya que no necesita el uso de agentes; es, por tanto, agentless. Esto es una característica muy importante, ya que, al utilizar herramientas como Puppet y Chef, cada agente que ejecuta en una máquina gestionada se conectará periódicamente con el servidor centralizado para comprobar si existen cambios de la configuración y los aplicará automáticamente. Ansible, en cambio, delega completamente en el usuario final la labor de propagar los cambios de configuración cuando se requiera.

Ansible es más parecido a SaltStack (Salt), otra herramienta que también está escrita en Python y utiliza ficheros YAML para la configuración. Tanto Ansible como Salt están diseñadas fundamentalmente como motores de ejecución, donde la definición de la configuración del sistema no es más que una lista de comandos que ejecutar, que se abstraen mediante módulos reutilizables, los cuales se encargan de proporcionar una interfaz idempotente a los servidores.

Una característica común de todas estas herramientas, tanto Ansible y Salt como Chef y Puppet, es que son declarativas, en lugar de imperativas. La gran mayoría de los elementos de configuración que proporcionan permiten que el usuario defina el estado de configuración deseado de sus hosts y sea la propia herramienta la encargada de alcanzarlo mediante la aplicación de las acciones que considere necesarias. Para lograr esto es importante el concepto de *idempotencia*, que permite aplicar tantas veces como queramos una definición de configuración sobre un host y su estado resultante será el mismo en todas las ejecuciones. La idempotencia es una característica importante en este tipo de herramientas. La explicaremos con detalle más adelante.

Como ya hemos mencionado más arriba, Ansible no requiere de ningún agente en las máquinas que gestiona. Esto es lo que se denomina modelo *agentless*. En este modelo es necesario enviar los cambios de la configuración a las máquinas, cuando así se requiera, a demanda (cuando haya cambios en la configuración, o en las propias máquinas, generalmente). Este modelo es diferente del de Puppet y Chef (con agente), donde se usa un servidor centralizado que almacena la copia maestra de la configuración y al que las máquinas consultan periódicamente para asegurarse de que tienen el estado de su configuración actualizada.

Este modelo tiene ventajas e inconvenientes; la ventaja principal es que, una vez que haces cambios, puedes enviarlos inmediatamente a las máquinas, sin esperar a que un proceso demonio (el agente) compruebe si hay cambios. La desventaja es que tú eres el responsable de distribuir esos cambios a tus máquinas, mientras que con Puppet y Chef basta simplemente con guardar (commit) tus cambios en el servidor centralizado, teniendo la certeza de que serán distribuidos pronto. Cabe destacar que Ansible puede configurarse para utilizar este modelo de funcionamiento *pull*, pero no es lo más habitual.

Chef

Chef es uno de los frameworks más extendidos para la gestión de la infraestructura como código, proporcionando una extensa biblioteca de primitivas y utilidades para la gestión de todos los recursos que se usan en los procesos de construcción y mantenimiento de la infraestructura de sistemas. Chef utiliza un lenguaje específico de dominio (DSL), que está basado en el lenguaje de programación Ruby y proporciona una capa de abstracción sobre los recursos que permite, tanto a un administrador de sistemas como a un desarrollador, definir y aprovisionar fácilmente entornos escalables.

El framework Chef está compuesto por **tres componentes básicos** que interactúan entre sí: el servidor Chef, las máquinas gestionadas denominadas *nodos* y la estación de trabajo Chef (Chef workstation).

- *Servidor Chef*

El servidor Chef constituye el centro de los datos de configuración, conteniendo los cookbooks, que son los módulos de configuración de recursos, las políticas que se aplicarán a los diferentes nodos, y los metadatos, que describen cada uno de los nodos administrados por Chef.

- *Nodos*

Los nodos usan la herramienta Chef Client para conectarse con el Chef Server y solicitar los detalles de configuración que deben aplicarse sobre la máquina donde se ejecutan. A este proceso de aplicar los cambios sobre los nodos se le denomina *Chef run*.

Cabe destacar que, cuando un nodo se crea y se registra en el Chef server, el Chef Client se instala en el proceso de arranque, para iniciarse cuando la máquina se levanta.

En Chef, se llama **rol** a un conjunto de atributos y una lista de recetas para el nodo. Cada nodo puede tener uno o más roles y estos pueden ser reutilizados por varios nodos, por lo que se puede definir un conjunto de nodos que tengan el mismo rol dentro de nuestro sistema. La **lista de ejecución** se refiere a la lista de recetas asociadas con un nodo a través del rol o las propias recetas de las que depende; el orden de ejecución será el mismo orden en que está definida.

- *Chef workstation*

La estación de trabajo Chef, también denominada *repositorio de Chef* (Chef-repo), contiene la estructura del proyecto gestionado por Chef y lo maneja el desarrollador o administrador desde su workstation. Todos los componentes de Chef que constituyen nuestro sistema están definidos aquí: **cookbooks, entornos, roles** y **pruebas**. Una buena práctica que es frecuente es la de mantener el chef-repo en un sistema de control de versiones para así gestionarlo como si fuera código fuente de una aplicación.

La estructura de directorios de un Chef-repo puede variar, ya que algunos usuarios prefieren un único Chef-repo para mantener todos los cookbooks, mientras que otros prefieren almacenar cada cookbook en un repositorio separado, pero, en cualquier caso, el repositorio Chef (Chef-repo) debe poder gestionar toda la infraestructura. Para ello, debe contener la información de todas sus partes.

Puppet

Puppet es una herramienta de gestión de la configuración escrita en lenguaje Ruby con licencia de código abierto GPLv2, aunque también dispone de una versión comercial denominada *Enterprise*. Se ejecuta habitualmente con un modelo cliente-servidor, aunque también se puede ejecutar en modo *stand-alone* (independiente).

Fue desarrollado por el ingeniero Luke Kanies en la compañía *Puppet Labs* (anteriormente llamada *Reductive Labs*). Kanies ha estado trabajando con sistemas Unix y con la administración de sistemas desde el año 1997; y ha desarrollado Puppet por la insatisfacción que le provocaban las herramientas de gestión de la configuración existentes en aquel entonces. En el año 2005 Kanies fundó la empresa *Puppet Labs* para el desarrollo de herramientas de automatización de código abierto. Al cabo de poco tiempo, esta empresa lanzó Puppet, su producto principal.

Puppet es capaz de gestionar la configuración de servidores basados en las plataformas UNIX (incluyendo OSX) y Linux, así como servidores basados en Microsoft Windows.

Asimismo, es utilizado habitualmente para la gestión a lo largo de todo el ciclo de vida de las máquinas: comenzando desde la creación y primera instalación, continuando por las actualizaciones y el mantenimiento, y finalizando, al acabar su vida útil, migrando los servicios que proporcionaba a otros sistemas. El diseño de Puppet está pensado para estar en continua interacción con las máquinas que gestiona, al contrario que otras herramientas de aprovisionamiento que únicamente se encargan de la etapa de construcción de las máquinas.

Puppet tiene un modelo de funcionamiento sencillo, fácil de entender y de aplicar, que se compone de tres componentes básicos:

- Despliegue.
- Lenguaje de configuración y capa de abstracción de recursos.
- Capa transaccional.

Despliegue

El despliegue de Puppet sigue habitualmente un modelo cliente-servidor. El servidor recibe el nombre de **Puppet Master**, mientras que el cliente de Puppet que se ejecuta en las máquinas (hosts) que se van a gestionar se llama **agente** y el propio host se define como un nodo.

El proceso de Puppet Master se ejecuta como un daemon en el servidor, donde se almacena la configuración necesaria de todo el entorno gestionado. Los agentes se identifican con el servidor Puppet Master al conectarse y utilizan el estándar SSL para establecer un canal de comunicación cifrado, a través del cual se obtiene la configuración que se va a aplicar.

Cada uno de los agentes Puppet puede estar ejecutándose como un proceso *daemon* (demonio) mediante mecanismos tales como cron (o incluso el agente puede ejecutarse manualmente cuando así se requiera). Lo más habitual es configurar el agente para que se ejecute como un daemon, conectándose periódicamente con el servidor para determinar si su configuración está actualizada o, en caso contrario, para descargar y aplicar cualquier cambio requerido en la configuración.

También se puede ejecutar Puppet en modo **independiente**, sin contar con una conexión a un Master. La configuración debe encontrarse instalada localmente en la máquina y se ejecuta el binario de Puppet que comprobará y aplicará los cambios necesarios para alcanzar esa configuración.

Lenguaje de configuración y capa de abstracción de recursos

Puppet cuenta con un lenguaje declarativo que sirve para definir los distintos elementos de la configuración, que se denominan **recursos**. Este aspecto declarativo es una característica importante de Puppet con respecto a otras herramientas de configuración. Mediante el lenguaje se declara el estado **deseado** de la configuración, tal como indicar que un determinado paquete debería estar instalado o un determinado servicio no debería estar ejecutándose. En cambio, otras herramientas utilizadas para la gestión de la configuración, tales como Shell o Perl, son de naturaleza **imperativa** o procedimental: indican las operaciones concretas a realizar en lugar de declarar el estado final deseado.

La mayoría de los scripts personalizados que se implementan para automatizar la configuración de un sistema son un ejemplo del modelo imperativo.

Esto significa que los usuarios de Puppet simplemente declaran el estado deseado de sus hosts: qué paquetes deben ser instalados, qué servicios deberían correr, etc. En otras palabras, Puppet se encargará de lograr el estado deseado y los administradores de sistemas abstraerán la configuración de los hosts a base de definir los recursos.

Capa transaccional

La capa transaccional es el motor de Puppet. Se conoce como **transacción** en Puppet al proceso de configuración de cada máquina, esto es:

- Interpretación y compilación de la configuración.
- Transmisión de la configuración compilada a cada agente.
- Aplicación de la configuración en la máquina del agente.
- Informe del resultado de la ejecución al Puppet Master.

Lo primero que hace Puppet es el análisis de la configuración para calcular cómo debería aplicarse en el agente, creando para ello un grafo que incluye los distintos recursos y las relaciones entre ellos, así como con cada agente. Esto permite a Puppet establecer en qué orden aplicar cada recurso al host, basándose en las relaciones que se crean.

Acto seguido, con los recursos obtenidos previamente, Puppet compila un catálogo de los mismos para cada agente. Este catálogo es lo que se envía a cada host para que el agente de Puppet lo aplique sobre su máquina. Con el resultado de esta ejecución se genera un informe, que es lo que se envía posteriormente de vuelta al Puppet Master.

La capa transaccional permite crear configuraciones y aplicarlas repetidamente en el host, garantizando la idempotencia de las mismas.

Sin embargo, Puppet no es totalmente **transaccional**: las transacciones no se registran (más allá de los registros informativos) y, por lo tanto, no se pueden deshacer como se hace con algunas bases de datos. Lo que sí podemos hacer es configurar estas transacciones en modo NOOP (del inglés *no operation mode*), lo que nos posibilita el probar la ejecución de la configuración sin realmente realizar ningún cambio.

Ansible

La primera versión de Ansible apareció en el 2012 como un pequeño proyecto de apoyo de la mano de Michael DeHaan y ha tenido una ascensión meteórica en popularidad, con más de 40.000 estrellas y 5.000 contribuidores únicos en GitHub.

Ansible es uno de los proyectos de código abierto con más popularidad que puedes encontrar en GitHub. No solo ha alcanzado un gran éxito, sino que gigantes como la NASA, Spotify o Apple lo han adoptado como herramienta de gestión de la configuración.

Ansible utiliza ficheros en formato YAML: los ficheros YAML son muy fáciles de leer y usar, al contrario que lo que ocurre con otros formatos o lenguajes como JSON o XML, que no son tan amigables. Sin embargo, incluye algunas particularidades, siendo las principales las de ser sensible a los caracteres de tabulación (no le gustan), a los espacios en blanco y a la sangría de cada línea. Si alguna vez has escrito código fuente en lenguaje de programación Python, todo esto no será ningún problema.

Python es el lenguaje de programación en el que está escrito Ansible. El ejecutable principal y todos los módulos son compatibles con Python 2.7 o Python 3.5, lo que significa que funcionan con cualquier versión de Python2 por encima de la 2.7 o Python 3 por encima de la 3.5.

DeHaan eligió Python para Ansible porque no requiere dependencias adicionales en las máquinas que se van a gestionar. En aquella época, la mayoría de las herramientas de gestión de la configuración existentes necesitaban de la instalación de Ruby como prerrequisito.

No solo no hay ninguna dependencia o requisito adicional de instalación de otros lenguajes para las máquinas a gestionar, sino que no hay ningún otro requisito adicional. **Ansible aprovecha el protocolo SSH** para ejecutar sus comandos de forma remota en las máquinas que gestiona, por lo que no requiere la instalación de ningún otro protocolo o sistema de comunicación. Esto supone una gran ventaja debido a:

- Las máquinas gestionadas van a ejecutar exclusivamente la aplicación o aplicaciones que le correspondan, sin ningún otro proceso ejecutándose en segundo plano que compita por la CPU y memoria de la máquina.
- Al hacer uso de SSH, toda su funcionalidad está disponible para aprovecharla. Puedes, por ejemplo, utilizar un host como máquina de salto para alcanzar otro host. Además, no existe la necesidad de incluir un mecanismo de autenticación propio; puedes utilizar el que te proporciona SSH.

Seguridad en Devops

DevOps es un término usado para describir la mejor comunicación y colaboración entre profesionales de desarrollo de software y operaciones de infraestructura. Es posible incluir el aspecto de la seguridad en dicha definición para obtener ***SecDevOps*** [4]:

> *SecDevOps es un paradigma en el que se integran los procesos de desarrollo de software y operaciones considerando requisitos de seguridad y conformidad*

Estas definiciones se pueden considerar un punto de partida, pero no engloban aspectos como la automatización o infraestructura como código, tan presentes en cualquier entorno DevOps. SecDevOps incorpora ambos al ámbito de la seguridad.

Grupos de trabajo como www.devsecops.org recopilan herramientas y ejemplos para automatizar análisis de seguridad y que la seguridad se convierta en una parte **integral** de DevOps: de este modo, los controles se pueden integrar directamente en el producto en vez de insertarlo a la fuerza una vez construido.

En un clima tan competitivo como el actual, aquellas organizaciones que quieran tener alguna opción de éxito deben ser ágiles y asumir los riesgos que empresas menos atrevidas no están dispuestas a asumir. La protección de los activos está a cargo de los equipos de seguridad, que deben colaborar con los equipos de sistemas, desarrollo, producto y dirección. Esto no es algo nuevo y aplica tanto a empresas tradicionales como a aquellas que han adoptado DevOps. Sin embargo, los equipos de seguridad parecen tener dificultades para integrarse con la disciplina.

Si en DevOps se intentan acercar los enfoques de desarrollo y operaciones para conseguir los objetivos de la organización, lo mismo debería ocurrir cuando se incorpora un nuevo equipo a la metodología. Para asegurarse que un entorno DevOps es seguro se debe comenzar con una estrecha integración entre los desarrolladores, los operadores y los ingenieros de seguridad. **La seguridad debe servir al cliente como una función del servicio**, y los objetivos internos de los equipos de seguridad y los equipos de DevOps deben estar alineados.

En aquellos entornos en los que se ha integrado la seguridad en DevOps, los ingenieros de seguridad son una pieza más del pipeline: pueden añadir controles directamente sobre el producto en vez de emitir una recomendación en forma de una documentación que debe llegar a un desarrollador que puede implementarla o no, de mejor o peor manera. La idea fundamental es que, tras la adopción de técnicas ágiles por parte de desarrolladores y operadores, los ingenieros de seguridad hagan lo mismo y pasen de defender sus métricas a defender a toda la organización.

Seguridad basada en pruebas

El mito de los hackers que penetran en firewalls gubernamentales o decodifican la encriptación con sus teléfonos inteligentes es propio de excelentes películas, pero pobres ejemplos del mundo real. Los casos reales son más mundanos: los atacantes buscan objetivos fáciles, como vulnerabilidades de seguridad conocidas, equipos sin parchear, aplicaciones expuestas con las contraseñas predefinidas de instalación o con credenciales expuestas en código fuente.

El primer objetivo al implementar una estrategia de seguridad continua debe ser cuidar el nivel básico: aplicar conjuntos de controles elementales en la aplicación y la infraestructura de la organización y probarlos continuamente. Estos controles tienen un alcance reducido y pueden ser muy concretos; [5] cita varios ejemplos de controles:

- El inicio de sesión con el usuario ***root*** debe estar deshabilitado en todos los sistemas.
- Los sistemas y las aplicaciones deben ser parcheados a la última versión disponible dentro de los 30 días de su lanzamiento.
- El uso de HTTP debe limitarse a una redirección automática a la dirección HTTPS equivalente.
- Los secretos y las credenciales no deben almacenarse con el código de la aplicación.
    - Como alternativa, se pueden gestionar en una caja de seguridad virtual accesible solo para operadores.
- Los puntos de acceso administrativos o privilegiados de las aplicaciones deben estar protegidos, preferentemente detrás de una VPN.

Esta lista de controles no tiene por qué establecerse unilateralmente por el equipo de seguridad. Debería existir un consenso en el que tanto seguridad como desarrollo y operaciones entienden el valor de cada control y están de acuerdo en implementarlo.

Pruebas de aplicación

Las pruebas de la aplicación pueden contener únicamente pruebas unitarias o de estilo en las que el código se analiza de manera estática, aunque es habitual que se ejecuten también pruebas end-to-end o de integración. En estos casos, especialmente si la aplicación es un servidor, es necesario arrancar este con el código de la nueva funcionalidad, preferiblemente en un entorno nuevo que se pueda desmantelar al terminar las pruebas.

Vulnerabilidades web

En este tipo de pipelines las pruebas no tienen por qué restringirse a las funcionalidades de la aplicación. Hay utilidades como OWASP Zed Attack Proxy, o ZAP[1], que permiten automatizar comprobaciones contra vulnerabilidades típicas de aplicaciones web. Por ejemplo:

- Ataques de ***cross-site scripting***, o de scripting cruzado, en los que un atacante consigue introducir código JavaScript en el contenido de una página. Eso se puede conseguir incluso sin modificar el código fuente, simplemente introduciendo el código en, por ejemplo, el campo de descripción del perfil de un usuario en una red social. Si ese campo no se valida, el navegador de un usuario que visualice la información del perfil del atacante ejecutará el código contenido en ese campo.

[1] https://www.zaproxy.org/

- Ataques de ***cross-site request forgery,*** o de peticiones maliciosas cruzadas, en los que una página maliciosa incluye enlaces que redirigen al usuario a algún punto de la aplicación en la que realizará una acción que el usuario no intentaba hacer, como el borrado de un recurso. Esto se consigue sin necesidad de modificar el código de la página original ni introducir datos manipulados en ningún formulario, simplemente confiando en las cookies el dominio de la aplicación, si el usuario ya ha iniciado sesión.

- ***Clickjacking***, o captura de clics, en la que un atacante inserta parte de la aplicación en un marco, o iframe, de una web maliciosa engañando al usuario para que haga clic en un botón de la aplicación que realmente no es visible.

ZAP es capaz de automatizar comprobaciones contra este tipo de vulnerabilidades antes de que el código llegue a producción. Esto no significa que la aplicación no deba escanearse en producción, pero con este tipo de pruebas se puede reducir el número de parches que hay que añadir a raíz de un análisis o una auditoría.

Autenticación

HTTP soporta nativamente autenticación básica con la cabecera *Authentication*, pero las aplicaciones pueden implementar sus propios mecanismos mediante cookies o cabeceras personalizadas. En estos casos, las pruebas deben cubrir elementos como el cifrado no reversible de contraseñas de usuario.

Otra opción es usar protocolos de identidad para delegar la autenticación en una tercera parte, como SAML o OpenID Connect. Estos protocolos, conocidos genéricamente como SSO (***single sign-on***), permiten que una aplicación confíe en esta tercera parte para identificar al usuario sin alojar sus credenciales.

Las pruebas de autenticación no son sencillas, especialmente si se usa un mecanismo propio o si se implementa un mecanismo de SSO. En el primer caso, las pruebas unitarias deberán incluir comprobaciones específicas. En el segundo, las pruebas end-to-end con simulaciones de interacción en un navegador, como Selenium[2] o Taiko[3].

Dependencias

Es habitual que las aplicaciones modernas deleguen funcionalidades en librerías externas (por ejemplo, del repositorio de npmjs.com para NodeJS o pypi.org para Python). Esto acelera el tiempo de desarrollo, ya que permite que un desarrollador sin conocimientos avanzados en un campo concreto pueda hacer uso de código que expone una funcionalidad completa con una interfaz sencilla. Por ejemplo, nadie en la industria intentará escribir de cero un algoritmo de hash MD5, ya que la posibilidad de error es muy alta y las librerías disponibles están de sobra probadas.

[2] https://www.selenium.dev/
[3] https://docs.taiko.dev/

Sin embargo, estas librerías pueden introducir vulnerabilidades propias sin que el desarrollador sea consciente de ello. Los instaladores de dependencias pueden automatizar la actualización de dependencias en cada ejecución del pipeline, pero esto puede provocar otros errores si la librería incorpora cambios no compatibles hacia atrás. Pueden aparecer problemas incluso cuando las librerías siguen las directrices de Semantic versioning[4], así que los desarrolladores tienden a fijar las versiones para actualizarlas solo proactivamente.

Esto lleva a que las aplicaciones no reciban los parches de seguridad que las librerías incorporan en nuevas versiones. Una de las pruebas que se puede incorporar en el pipeline es la de verificar si las versiones en uso sufren de vulnerabilidades conocidas. Por ejemplo, NodeJS incluye la Node Security Platform en su ecosistema. Mediante npm audit es posible obtener la lista de vulnerabilidades de las versiones en uso. También hay analizadores similares para Python, como pyup.io o requires.io.

HTTPS

Las aplicaciones que sirven sitios web o interfaces API REST deben proteger la comunicación HTTP. Como se detalló en el tema 4, el protocolo SSL (y su sucesor, TLS) añaden confidencialidad, autenticación e integridad a HTTP. Habilitar HTTPS no asegura que el tráfico sea seguro por sí solo. Las pruebas deberán evaluar aspectos como:

- La aplicación solo es accesible por HTTPS, no por HTTP. Como mucho, cualquier petición por HTTP deberá redirigir a la misma ruta con prefijo https://.
- Los certificados se han generado con la longitud de clave suficiente, no han caducado y no han sido revocados.
- El servidor solo soportará las suites de cifrado más modernas o, al menos, no soportará suites con vulnerabilidades conocidas. La adopción de algoritmos modernos no tiene por qué ser rápida en todos los clientes, así que las organizaciones deben encontrar un equilibrio entre los clientes que soportan y los algoritmos que quieren dejar de soportar.

Por ejemplo, testssl.sh[5] es una herramienta de línea de comandos que genera informes sobre los certificados y las suites de cifrado ofrecidas por el servidor.

Protección de la infraestructura

Ya se trate de una aplicación con una interfaz HTTPS o un servidor interno que analice un *data lake* periódicamente, las instancias en las que se ejecutan los procesos deben estar protegidas a nivel de red. En un entorno de nube, además, hay que tener en cuenta no solo las instancias de cómputo, sino también otros objetos nativos del proveedor, como balanceadores de carga y redes virtuales.

[4] https://semver.org/
[5] https://testssl.sh/

Pruebas de red

Se puede desplegar un entorno que simule lo máximo posible el de producción, aunque a menor escala, y ejecutar pruebas de acceso sobre él. En este caso no se trata de comprobar vulnerabilidades a nivel de aplicación, sino de verificar que las reglas de firewall (o el objeto equivalente, como un grupo de seguridad) están configuradas correctamente. Como estos entornos son privados se pueden llevar a cabo escaneos de puertos sin afectar el funcionamiento del entorno de producción.

Bases de datos y almacenamiento

Las aplicaciones harán uso de algún tipo de persistencia, ya sea una base de datos desplegada *ad hoc*, una base de datos como servicio (por ejemplo, RDS en Amazon) o un almacenamiento de bloque o de objeto como Amazon S3. Estos elementos deben ser también protegidos, por lo que las pruebas comprobarán que no se permite acceso sin credenciales o que los puertos de la base de datos solo están abiertos a los servidores de la aplicación.

Otros elementos de infraestructura

Además de la red, hay otros elementos de la infraestructura que no están relacionados directamente con el desarrollo de la aplicación. Por ejemplo, puede haber servidores bastión para el acceso de operadores o puntos de entrada por VPN. Es posible que cada entorno de pruebas no se despliegue con toda esta infraestructura; en estos casos será en el entorno de producción donde se comprueben que las configuraciones son correctas: los equipos bastión solo soportan acceso con clave pública (es decir, no con contraseña), el servidor VPN acepta solo clientes modernos con algoritmos sin vulnerabilidades conocidas, etc.

Protección de las herramientas de CI/CD

Si un atacante consigue acceder a una de las herramientas del pipeline de CI/CD, cualquier otra medida que se haya implementado no servirá de nada. Da igual que todos los elementos de la aplicación estén protegidos y las pruebas finalicen satisfactoriamente: quien tenga acceso completo al pipeline puede modificar a su gusto cualquier elemento de la aplicación o de la infraestructura. Por ejemplo, podría desactivar aquellas que detecten una vulnerabilidad que el atacante quiere activar en la aplicación, o podría cambiar una imagen del repositorio, que ya debería haber sido validada, para ejecutar código arbitrario.

Repositorios de código

Algunas de las buenas prácticas de control de acceso, no solo en aplicaciones de control de versiones sino en cualquier sistema, son las siguientes:

- **Mantener la lista de usuarios con acceso ilimitado al repositorio lo más pequeña posible.** Siempre es necesario que haya uno o varios administradores principales, pero no se debe caer en la tentación de dar acceso indiscriminado a los usuarios solo para facilitar la asignación de permisos.

- **Requerir autenticación multifactor (MFA).** Cada vez más servicios de control de versiones soportan autenticación de doble factor, lo que ofrece una capa de seguridad adicional. Además de las credenciales, el usuario deberá introducir un código que puede obtener, por ejemplo, a través de un SMS, de un email, de una aplicación de móvil o de un *token físico*, que es como se suelen denominar unos dispositivos específicos que muestran un código por pantalla.

- **Auditar regularmente los miembros de los grupos.** Por ejemplo, se pueden comprobar los usuarios que pertenecen a un grupo de desarrolladores a través de la API de GitHub y compararlos con los usuarios de un directorio activo local. Así se evitan discrepancias entre un entorno y otro.

Firma de commits

Los *commits* que llegan a la rama principal de una aplicación pueden contener código fraudulento si no se establecen los controles necesarios. Si el acceso a un repositorio se ve comprometido, un atacante podría inyectar código fuente fraudulento en la aplicación sin que los desarrolladores lo noten. Sistemas como GitHub ofrecen características de seguridad, como la protección de ramas, para evitar operaciones sensibles, como añadir commits directamente a la rama master. Si el acceso basado en roles está bien configurado, este tipo de restricciones proporcionan una buena capa de seguridad.

Pero un atacante sería capaz de deshabilitar estas protecciones si consigue acceder al repositorio. La firma de commits con git proporciona una capa adicional. Git soporta la firma de commits y etiquetas con PGP. Esta funcionalidad consiste en aplicar firmas criptográficas a cada parche, commit y etiqueta utilizando claves que los desarrolladores mantienen en secreto. Los algoritmos que se usan no son diferentes a los que se usan en HTTPS, aunque las herramientas no son las mismas. Si los *commits* se firman con una clave válida dentro de la organización, se pueden considerar confiables.

Infraestructura y contenedores

Las herramientas de integración continua, como Jenkins o CircleCI, así como el acceso a los proveedores de nube, deben estar protegidas con un correcto acceso basado en roles, como en el servicio IAM de AWS.

IAM permite, además, distribuir credenciales de AWS a aplicaciones que se ejecuten en su infraestructura con los roles de IAM. La distribución de secretos es un tema sensible y nada fácil de solucionar, por lo que hay que aprovechar estas funcionalidades allá donde estén. Kubernetes también permite guardar secretos y entregarlos a los pods de manera segura, y Jenkins permite alojar secretos internamente y exponerlos a los trabajos solo donde sea expresamente necesario.

Para finalizar, los paquetes de la aplicación pueden protegerse frente a alteraciones, al igual que los commits. Por ejemplo, si la aplicación se despliega a base de contenedores, se pueden firmar el tráfico enviado a un repositorio de Docker mediante Docker Content Trust[6].

[6] https://docs.docker.com/engine/security/trust/content_trust/

Conclusión

Los equipos de seguridad deben mantener una relación estrecha con sus colegas de desarrollo y operaciones. Solo así podrá adoptar la cultura DevOps y actualizar y adaptar las técnicas tradicionales, como la detección de intrusiones o el escaneo de vulnerabilidades, en el pipeline de CI/CD.

Un programa de seguridad no se define de un día para otro: debe formar parte integral de la estrategia de la compañía y madurar con ella. Solo cuando existe colaboración entre equipos surgirá la confianza necesaria para trabajar juntos con un objetivo común. En resumen, la manera de garantizar la seguridad de los productos en un entorno DevOps es integrar a todos los equipos bajo este paradigma.

LA UNIVERSIDAD EN INTERNET | FORMACIÓN PROFESIONAL

PROEDUCA