Launch by Lunch

Databases, DevOps, and Development

Encrypting Docker containers on a Virtual Server

Jan 13 2014

Background

In my last post I went over setting up Dokku on a DigitalOcean droplet. In this post we'll improve that setup a bit by add some encrypted swap space. We'll also be encrypting all Dokku files, Docker containers, and the application log files for our deployed Dokku apps.

Why Encrypt?

Encryption is used to prevent unauthorized people from reading your data. When done at the file system level, encryption has the additional benefit that it makes it *much* easier to dispose of the data on a disk.

An import part of disposing of any server (*or more generally any hard drive*) is wiping the disks. With a cloud server, you are not in custody of the physical disks and securely wiping them becomes even more important. If you're not careful about it then bad things can happen.

To wipe an unencrypted disk you would need to over write the entire disk with zeroes (*or random gibberish from `/dev/urandom`*). Otherwise any data that was previously there could be retrieved. The larger the disk, the longer this will take. For example a 1 TB disk with a sequential write speed of 75 MB/s would take almost 4 hours to zero out.

On the other hand, with an encrypted disk you can simply destroy the encryption keys and *nobody* will be able to read the original data again. Since the encryption keys are relatively small (*ex: the header and key slots for LUKS total only a couple of MB*), zeroing them out is very quick.

Limitations

The steps in this guide will setup an encrypted block device. The encryption key for the block device will be stored in plaintext along side it. This is being done for convenience and will allow the virtual server to reboot without manual intervention (*i.e. we don't have to enter the pass phrase at boot time*).

This is not secure!

Just to clarify further ... This is not really secure! Having the key stored in plain text alongside the encrypted block device is kind of like having a lock on your door and the key on the door step. If someone has access to your virtual server then they will be able to get the decryption keys. This includes any backup snapshots that archive the virtual server's entire disk.

Still though, this is better than nothing and will allow you to more easily destroy your virtual server's disk when you're done using it. At the end of this post I'll go over some additional ideas to improve this configuration further.

Setup

A standard DigitalOcean droplet comes with a single disk allocated with the full space of the droplet. In our case it's approximately 30 GB:

```
root@launchbylunch:~# df -h
Filesystem      Size  Used Avail Use% Mounted on
/dev/vda         30G  6.1G   22G  22% /
none            4.0K     0  4.0K   0% /sys/fs/cgroup
udev            494M  4.0K  494M   1% /dev
tmpfs           100M  248K  100M   1% /run
none            5.0M     0  5.0M   0% /run/lock
none            498M  7.0M  491M   2% /run/shm
none            100M     0  100M   0% /run/user
```

Since we don't have a separate hard drive (*it's all one disk*), we'll be creating a file on the existing non-encrypted filesystem and using that as a block device (*via a loop device*).

Install Packages

There's only one package to install, `cryptsetup`. Install it via `apt-get`:

```
$ apt-get -y install cryptsetup
```

Create Block Device Files

Throughout this post I'll be using `$CRYPTFS_ROOT` as the path on the unecrypted filesystem. If you'd like to have the encrypted block device and key stored elsewhere, simply modify the line that sets it's value.

Create a directory at the root of the filesystem to hold our encrypted block device.

```
# Location on the root filesystem where we'll store everything:
$ CRYPTFS_ROOT=/cryptfs

# Make the directory to store our encrypted block device:
$ mkdir -p $CRYPTFS_ROOT
```

Next, we'll create the encryption key for our block device. The key will be randomly generated from `/dev/urandom`. For most purposes this should be fine but for a more secure setup this key should generated on a non-virtual server (*ex: your desktop*) and uploaded to the virtual server.

```
# Create a random encryption key:
$ dd if=/dev/urandom of=$CRYPTFS_ROOT/key bs=4K count=1
```

Next, we'll create a 2 GB swap file. We'll be initializing it with zeroes so that it gets fully allocated. This will take up to 10-20 seconds as it will need to write everything to disk.

```
# Create and allocate a 2 GB swap file:
$ dd if=/dev/zero of=$CRYPTFS_ROOT/swap bs=1M count=2048
```

Next, we'll create our encrypted block device. Since we don't really care about the existing data on the disk we'll create a sparse file using the `truncate` command. Alternatively, if we wanted to overwrite any existing data on disk we could allocate it similar to how the swap file was allocated (*though this would take substantially longer*).

```
# Create a sparse 20 GB file:
$ truncate -s 20G $CRYPTFS_ROOT/disk
```

Finally we'll change the permissions on all these files so only `root` can access them:

```
$ chmod -R 700 "$CRYPTFS_ROOT"
```

Setup Swap

Now we'll add an entry to your `/etc/crypttab` for a swap file. We'll also add an entry to `/etc/fstab` so that the swap is automatically enabled at each boot.

The encryption key for this will be randomly chosen at each boot from `/dev/urandom` and never saved. This means that if the virtual machine is powered off no application data that was written to swap will be recoverable (*that's a good thing!*).

```
# Add a line to /etc/crypttab for our swap file:
$ echo "swap $CRYPTFS_ROOT/swap /dev/urandom swap" >> /etc/crypttab

# Add swap entry to /etc/fstab:
$ echo "/dev/mapper/swap none swap defaults 0 0" >> /etc/fstab
```

Encrypted Block Device Setup

Next up we'll create a LUKS volume in our 20 GB file. To do so we'll first need to mount it using a loop device.

```
# Pick an unused loop back device and save it to a variable:
$ LOOP_DEVICE=$(losetup -f)

# Mount our block device file to it:
$ losetup $LOOP_DEVICE $CRYPTFS_ROOT/disk
```

Now we'll use `cryptsetup` to format it as a LUKS volume. We'll be using the default settings for LUKS.

```
# Create a LUKS volume with the default options using our key file
$ cryptsetup --batch-mode --key-file=$CRYPTFS_ROOT/key luksFormat $LOOP_DEVICE

# Open it up so it appeares in /dev/mapper
$ cryptsetup luksOpen --key-file=$CRYPTFS_ROOT/key $LOOP_DEVICE cryptfs
```

The encrypted block device should now show up as `/dev/mapper/cryptfs`. We can check the cipher details for it by running `cryptsetup status`. It should look like this:

```
$ cryptsetup status cryptfs
/dev/mapper/cryptfs is active and is in use.
  type:    LUKS1
  cipher:  aes-cbc-essiv:sha256
  keysize: 256 bits
  device:  /dev/loop0
  loop:    /cryptfs/disk
  offset:  4096 sectors
  size:    41938944 sectors
  mode:    read/write
```

Now we'll create a filesystem on the encrypted block device. We're creating an ext4 filesystem:

```
$ mkfs.ext4 /dev/mapper/cryptfs
```

Finally lets create a mount point for it and add it to `/etc/crypttab` and `/etc/fstab` so that it gets automatically mounted at boot time.

```
# Create a mount point for the encrypted filesystem:
$ mkdir -p /mnt/cryptfs

# Add to /etc/crypttab using the fixed key at $CRYPTFS_ROOT/key
$ echo "cryptfs  $CRYPTFS_ROOT/disk  $CRYPTFS_ROOT/key  luks" >> /etc/crypttab

# Add it to /etc/fstab:
$ echo "/dev/mapper/cryptfs  /mnt/cryptfs  ext4  defaults  0  1" >> /etc/fstab
```

Now that the setup is complete let's reboot the server once and test it out. If everything was properly set up, then after the reboot the encrypted filesystem should be mounted to `/mnt/cryptfs` and we should have 2 GB of swap:

```
# Verify that the encrypted block device was mounted:
$ mount | grep cryptfs
/dev/mapper/cryptfs on /mnt/cryptfs type ext4 (rw)

$ free -m
             total       used       free     shared    buffers     cached
Mem:           995        514        480          0         86        182
-/+ buffers/cache:        245        749
Swap:         2047          0       2047
```

Docker and Dokku Setup

At this point we have a mount point at `/mnt/cryptfs` that will transparently encrypt and decrypt any data stored there. You could symlink other parts of your filesystem to subdirectories there to have them encrypted as well.

```
# Make a directory in the encrypted filesystem
$ mkdir -p /mnt/cryptfs/foo

# Create a symlink pointing to it for /var/lib/foo:
$ ln -s /mnt/cryptfs/foo /var/lib/foo
```

Unfortunately, this does not currently work with volume mounts for Docker containers(*see this issue*). A work around though is to use bind mounts. Bind mounts allow you to mount one part of a file system to another location. Conceptually they're very similar to symlinks.

Bind Mounts

We'll now add bind mounts to our `/etc/fstab` for all the directories that we'd like to be stored on our encrypted filesystem. This will include our all our Docker and Dokku related files, all `/home` directories, and the eventual location of nginx and our Dokku application log files.

WARNING: *This guide assumes that you've started with an empty server. Any existing data in `/home` will be inaccessible after adding these bind mounts. A new empty home directory will take it's place. Also, this must be done **before** you install Dokku or Docker as both of their installation directories will also be emptied.*

```
# Loop through the paths that we'd like to be encrypted:
for DIR_NAME in home var/lib/docker var/lib/dokku var/log/dokku var/log/nginx
do
  echo "Adding bind mount for DIR_NAME=$DIR_NAME"
  # Create the path on the encrypted filesystem (if it doesn't exist)
  mkdir -p "/mnt/cryptfs/${DIR_NAME}"

  # Create the path on the actual filesystem (if it doesn't exist)
  mkdir -p "/$DIR_NAME"

  # Add a bind mount entry to /etc/fstab for it:
  echo "/mnt/cryptfs/${DIR_NAME} /$DIR_NAME none bind 0 0" >> /etc/fstab
done
```

Updated @ 2014/Mar/07: *A previous version of this post incorrectly had the directories for the bind mount backwards. Thanks to John (https://github.com/ubergarm) for pointing this out!*

Now let's reboot one more time. After rebooting, verify that everything was mounted properly.

```
$ mount
/dev/vda on / type ext4 (rw,errors=remount-ro)
[ ... truncated ...]
/dev/mapper/cryptfs on /mnt/cryptfs type ext4 (rw)
/mnt/cryptfs/home on /home type none (rw,bind)
/mnt/cryptfs/var/lib/docker on /var/lib/docker type none (rw,bind)
/mnt/cryptfs/var/lib/dokku on /var/lib/dokku type none (rw,bind)
/mnt/cryptfs/var/log/dokku on /var/log/dokku type none (rw,bind)
/mnt/cryptfs/var/log/nginx on /var/log/nginx type none (rw,bind)
```

Installation

Now we can install Dokku and Docker. Follow the instructions from my previous post.

Once the installation is complete, our Docker containers, deployed applications, and all their config environment variables will be persisted to a transparently encrypted filesystem. Additionally, all our applications log files in `/var/log/dokku` will also be encrypted (*I'll be going over how to get Dokku to send application logs there in a later post*).

Destroying Everything

When we want to deprovision the server we can prevent the encryted data from ever being read again by destroying the encryption key and the LUKS header on the block device.

WARNING: *Unless you've created a backup of your encryption key and the LUKS header there is no way to undo this operation!*

```
# Shred the encryption key:
$ shred $CRYPTFS_ROOT/key

# Overwrite the beginning of the encrypted block device where the LUKS header is:
$ dd if=/dev/zero of=$CRYPTFS_ROOT/disk bs=1M count=10
```

Final Thoughts

The biggest hole in all of this is the fact that the encryption key is stored in plaintext alongside the block device file. Unfortunately that's the price to pay for unattended restarts. For a local physical server (*ex: your desktop*) that is rarely restarted, or one that you have direct console access to, entering the passphrase at each boot is not much of an issue. For a virtual server though, it's an extra step that either needs to be done from the virtual console or via a multi-step boot process (*boot, SSH, unlock the encrypted filesystem, resume boot ...*).

One solution to this that I've been considering is a hybrid approach. The server would be setup similar to this post, however rather than having the key file stored locally it would be requested at boot time from an external service.

The external service could notify you of the restart (*ex: on the service's website or via a push notification to your smart phone*). If you approve the request then it would respond with the encryption key, the server would mount the encrypted filesystem, and then resume the boot process. Any request not explicitly approved in X minutes would be automatically rejected. Additionaly once the encrypted filesystem is mounted the server would discard the encryption key (*so it would only be in memory while the server remains on*).

I think that would strike a good balance between security and usability.

Have an idea of how to improve this setup or have a working version of what I'm describing? Tell me about it.

Posted by Sehrope Sarkuni Jan 13 2014 docker dokku digitalocean crypto security

About

I'm Sehrope Sarkuni, founder of JackDB. If you'd like to chat about any of these topics feel free to contact me.

Subscribe

Want to be notified via email for new posts? Enter your email and subscribe below:

email address

Subscribe

Recent Posts

- How I Write SQL, Part 1: Naming Conventions
- AWS Tips, Tricks, and Techniques
- My blog's tech stack: Pelican powered, Dokku deployed
- Encrypting Docker containers on a Virtual Server
- How to set up a private PaaS with Dokku on DigitalOcean

Tags

pelican, git, postgresql, security, paas, s3, dokku, sql, aws, crypto, dkim, blog, 2fa, meta, ec2, digitalocean, databases, docker, email, ssh

Follow @sehrope

About

I'm Sehrope Sarkuni, founder of JackDB. If you'd like to chat about any of these topics feel free to contact me.

Subscribe

Want to be notified via email for new posts? Enter your email and subscribe below:

email address

Subscribe

Recent Posts

- How I Write SQL, Part 1: Naming Conventions
- AWS Tips, Tricks, and Techniques

- My blog's tech stack: Pelican powered, Dokku deployed
- Encrypting Docker containers on a Virtual Server
- How to set up a private PaaS with Dokku on DigitalOcean

Tags

pelican, git, postgresql, security, paas, s3, dokku, sql, aws, crypto, dkim, blog, 2fa, meta, ec2, digitalocean, databases, docker, email, ssh

Follow @sehrope