Images and dm-crypt and LVM2... Oh my!

Hal Pomeranz, Deer Run Associates

Disk layouts using the Linux Logical Volume Manager (LVM2) are increasingly becoming the norm for new Linux installs. And very often the physical volume used by LVM2 has been encrypted via dm-crypt. A recent email from a Sec508 student asking for a procedure for mounting these images prompted me to codify this information into a blog posting.

Investigating the Image

When initially presented with the image, you may not know whether LVM2 or dm-crypt has been employed. So let's start from scratch:

```
# md5sum sda.dd
f4c7a8d54b9b0b0b73ec03ef4cf52f42 sda.dd
# mmls -t dos sda.dd
DOS Partition Table
Offset Sector: 0
Units are in 512-byte sectors

     Slot    Start        End          Length       Description
00:  Meta    0000000000   0000000000   0000000001   Primary Table (#0)
01:  -----   0000000000   0000002047   0000002048   Unallocated
02:  00:00   0000002048   0000499711   0000497664   Linux (0x83)
03:  00:01   0000499712   0041940991   0041441280   Linux (0x83)
04:  -----   0041940992   0041943039   0000002048   Unallocated
```

You'll obviously want to verify the md5sum of the image before you begin your investigation. I'm also showing it here so that we can verify at the end of our procedure that we haven't modified the disk image in any way.

The mmls output shows two Linux file systems-- one rather small and the other consuming most of the disk-- and some unallocated regions. I'd already be suspicious that encryption was being used here, since this layout suggests the typical "unencrypted /boot and large encrypted volume" layout common to modern Linux installers. We also don't see a swap partition in this disk layout, suggesting that the swap area is hidden away inside the larger volume.

But let's check out the two Linux file systems to see if we can get more information. First, the smaller partition at the front of the disk:

```
# fsstat -o 2048 sda.dd
FILE SYSTEM INFORMATION
--------------------------------------------
File System Type: Ext3
Volume Name:
Volume ID: 2c71dce497ca7d83694ea172de905590

Last Written at: Tue Oct 5 11:06:10 2010
Last Checked at: Tue Oct 5 10:04:19 2010
```

```
Last Mounted at: Tue Oct 5 11:05:31 2010
Unmounted properly
Last mounted on: /boot
[...]
```

This does, indeed, seem to be the /boot partition.

Now let's see what we get when we run fsstat against the other partition:

```
# fsstat -o 499712 sda.dd
Cannot determine file system type
```

So the partition does not contain a file system type that fsstat recognizes. It seems unlikely that there would be no file system in this rather large partition, so the most likely explanation is LVM2 and/or a dm-crypt volume. But how can we test this theory?

The next step is to set up a loop device so that we can access the second partition more easily:

```
# expr 499712 \\* 512
255852544
# losetup -f
/dev/loop0
# losetup -r -o 255852544 /dev/loop0 sda.dd
```

The losetup command we'll be using to create the loop device accepts file offsets, but uses bytes as the unit for the offset rather than sectors. So we multiply the starting sector from our earlier mmls output by the sector size to figure out the correct byte offset. Then we run "losetup -f" to figure out the name of the first available loop device. Finally we combine these two pieces of information into the losetup command to actually create the device. Notice the use of the "-r" flag here, which creates a read-only device.

Now that we have the loop device we can investigate things a little further. First let's see if we're dealing with an encrypted volume:

```
# cryptsetup luksUUID /dev/loop0
cf20c757-20ed-4969-956a-b835e91aaec6
# cryptsetup luksDump /dev/loop0
LUKS header information for /dev/loop0

Version:  1
Cipher name:  aes
Cipher mode:  cbc-essiv:sha256
Hash spec:  sha1
Payload offset: 2056
MK bits:  256
MK digest:  31 7f cf d7 e6 c5 56 3f e7 32 fa 5a 77 93 c0 36 db 89 88 62
MK salt:  9f 92 f3 92 16 63 8e dd 92 82 28 e3 06 3c fb 4a
 a5 e5 87 79 ca 36 f2 04 cc 3c e4 5c 55 7a 3b ff
MK iterations:  10
UUID:  cf20c757-20ed-4969-956a-b835e91aaec6



Key Slot 0: ENABLED
 Iterations:  115085
 Salt:  b4 09 36 e1 9c 9c 6b 42 c2 c5 28 54 75 97 92 ed
 aa 43 4c a6 7c 3c 35 9d 49 0e 10 5b 1c dc 53 2d
 Key material offset: 8
 AF stripes:  4000
Key Slot 1: DISABLED
```

```
Key Slot 2: DISABLED
Key Slot 3: DISABLED
Key Slot 4: DISABLED
Key Slot 5: DISABLED
Key Slot 6: DISABLED
Key Slot 7: DISABLED
```

The cryptsetup command is the primary administrative interface to dm-crypt volumes. The "luksUUID" sub-command is enough to confirm that you're dealing with an encrypted volume, and "luksDump" provides more detailed information. Note that if you're scripting this process, there's also an "isLuks" command that simply returns true or false depending on whether the volume is encrypted.

Note that if you run this command on a volume that's not encrypted, you see output like this:

```
# cryptsetup luksUUID /dev/sda1
/dev/sda1 is not a LUKS partition
```

But in this case, our disk image definitely contains an encrypted volume. Accessing this volume is the subject of the next section. If you get a negative result from your cryptsetup commands, then you're probably dealing with just a plain LVM2 volume. Instructions for accessing LVM2 volumes will follow in the section after we deal with our encryption problem.

But before we get into all of that, let me pre-empt a question that I know several people are going to raise. Yes, I am aware that the cryptsetup command has a "-o" option for specifying a sector offset. So why did I go to the trouble of creating the loopback device when I could just do something like this:

```
# cryptsetup -o 499712 luksDump sda.dd
Command failed: sda.dd is not a LUKS partition
```

Why didn't this work? Trust me, I spent a great deal of time trying to answer that question myself. Ultimately, I ended up running cryptsetup via strace to see what was going on. Here are the relevant lines of output:

```
[...]
open("sda.dd", O_RDONLY|O_SYNC|O_DIRECT) = 3
ioctl(3, BLKSSZGET, 0x7fffeba890cc) = -1 ENOTTY (Inappropriate ioctl for device)
close(3) = 0
[...]
```

The ioctl() call is trying to retrieve the physical sector size (BLKSSZGET) but getting the "Inappropriate ioctl for device" error, which causes the program to fail. Apparently the Linux kernel doesn't like this ioctl() being applied directly to our image file. What's interesting is that creating a loop device binding to our file and then calling cryptsetup on the loop device works fine, as demonstrated in the example above. There's really little very difference between the two cases from a practical perspective, but apparently the loop device is necessary to fake out the kernel enough to allow the ioctl().

Getting Into The Encrypted Volume

In any event, we're dealing with an encrypted volume and we want to access the data that's inside of it. That means that we're going to have to discover the pass phrase that was used to encrypt the volume. There's no magic short-cut to getting through this pass phrase: you either need the owner of the system to tell it to you or leave it written down on a handy Post-It Note(tm). A brute-force

attack is possible, but you'd probably better assume that anybody who went to the trouble of encrypting their hard drive also went to the trouble of picking a long pass phrase. Good luck with that.

But let's be optimistic and assume that you've obtained the pass phrase to decrypt the drive. With that hurdle out of the way, actually getting into the encrypted volume is a simple cryptsetup command:

```
# cryptsetup luksOpen /dev/loop0 unencrypted
Enter LUKS passphrase for /dev/loop0:
key slot 0 unlocked.
Command successful.
# ls -l /dev/mapper/unencrypted
brw-rw---- 1 root disk 253, 8 Oct 5 17:42 /dev/mapper/unencrypted
```

"cryptsetup luksOpen ..." prompts you to enter the pass phrase. Assuming you enter the pass phrase correctly, it unlocks the appropriate encryption key and sets up a device mapping to the unencrypted device. The name of the mapped device-- "unencrypted" in this case-- is supplied as the last argument to "luksOpen". It doesn't really matter what you call this device. Just pick a unique name.

Now it's possible that the encrypted volume contains a Linux file system rather than a series of LVM2 volumes. Let's see what fsstat has to say:

```
# fsstat /dev/mapper/unencrypted
Cannot determine file system type
```

No joy. It could be an XFS, JFS, or ResierFS volume or some other file system type that fsstat doesn't recognize. But let's forge ahead under the assumption that LVM2 is going to come into play.

Dealing with LVM2 Volumes

Regardless of whether the image was encrypted or not, the process is the same for decoding any LVM2 configuration that might be present. In the case of an encrypted volume, the LVM2 commands will operate on the /dev/mapper/unencrypted device created by cryptsetup. If there is no encryption layer, then the commands will operate on the /dev/loop0 device we created with losetup. As you'll see in the examples below, you won't actually be specifying the device name, so the difference is transparent to the user.

The first step is to run vgscan ("volume group scan") to look for LVM2 volume groups attached to system devices:

```
# vgscan
 Reading all physical volumes. This may take a while...
 Found volume group "RAID6" using metadata type lvm2
 Found volume group "VG000" using metadata type lvm2
 Found volume group "Root" using metadata type lvm2
```

The vgscan command actually found several volume groups on my analysis machine. However, two of them were pre-existing volumes that I had configured when installing the system. You can use the pvdisplay ("physical volume display") command to show the volume group associated with a particular device:

```
# pvdisplay /dev/mapper/unencrypted
 --- Physical volume ---
 PV Name /dev/mapper/unencrypted
```

```
 VG Name VG000
 PV Size 19.76 GB / not usable 2.00 MB
 Allocatable yes (but full)
 PE Size (KByte) 4096
 Total PE 5058
 Free PE 0
 Allocated PE 5058
 PV UUID 0vsBWa-89Bl-N2eF-XzLU-S3aM-xvHk-YOQF3u
```

So "VGooo" is the name of the volume group on our encrypted device. The next step is to activate this volume group and ascertain what logical volumes it contains:

```
# vgchange -a y VG000
 5 logical volume(s) in volume group "VG000" now active
# lvscan
 ACTIVE '/dev/RAID6/Backup' [2.05 TB] inherit
 ACTIVE '/dev/RAID6/Scratch' [2.00 TB] inherit
 ACTIVE '/dev/VG000/root' [976.00 MB] inherit
 ACTIVE '/dev/VG000/usr' [7.45 GB] inherit
 ACTIVE '/dev/VG000/var' [3.72 GB] inherit
 ACTIVE '/dev/VG000/swap' [1.86 GB] inherit
 ACTIVE '/dev/VG000/home' [5.77 GB] inherit
 ACTIVE '/dev/Root/Root' [10.00 GB] inherit
 ACTIVE '/dev/Root/Home' [617.94 GB] inherit
 ACTIVE '/dev/Root/Usr' [20.00 GB] inherit
 ACTIVE '/dev/Root/Usr_Local' [20.00 GB] inherit
 ACTIVE '/dev/Root/Var' [20.00 GB] inherit
 ACTIVE '/dev/Root/Swap' [10.00 GB] inherit
# ls /dev/VG000/
home root swap usr var
# ls /dev/mapper/VG000-*
/dev/mapper/VG000-home /dev/mapper/VG000-swap /dev/mapper/VG000-var
/dev/mapper/VG000-root /dev/mapper/VG000-usr
```

The vgchange command makes VGooo active ("-a y"). Then lvscan ("logical volume scan") scans all active volume groups looking for logical volumes. Unfortunately, there is no way to tell lvscan to restrict the scan to a particular volume group, so it will report all logical volumes on all volume groups.

But as a result of these commands we now have a host of new device nodes associated with VGooo. You can pull strings out of these devices using strings or srch_strings. You can apply your favorite forensic tools to these devices. You could even dd the devices if you wanted to make a raw image file of each partition.

And of course we can actually reassemble the file system now using the mount command. In this case, the names of the logical volumes are a clue to where each volume should be mounted. In the absence of a meaningful naming scheme, I suggest using fsstat on each partition to find out where it was last mounted. Alternatively you can try mounting each volume in turn until you locate the root file system and the /etc/fstab file, which will tell you where the other volumes should be mounted.

For the sake of completeness, let's go ahead and put the entire file system back together:

```
# mount -o ro,noexec /dev/VG000/root /mnt
# mount -o ro,noexec /dev/VG000/usr /mnt/usr
# mount -o ro,noexec /dev/VG000/var /mnt/var
# mount -o ro,noexec /dev/VG000/home /mnt/home
# mmls -t dos sda.dd
DOS Partition Table
Offset Sector: 0
```

```
Units are in 512-byte sectors

 Slot Start End Length Description
00: Meta 0000000000 0000000000 0000000001 Primary Table (#0)
01: ----- 0000000000 0000002047 0000002048 Unallocated
02: 00:00 0000002048 0000499711 0000497664 Linux (0x83)
03: 00:01 0000499712 0041940991 0041441280 Linux (0x83)
04: ----- 0041940992 0041943039 0000002048 Unallocated
# expr 2048 \\* 512
1048576
# mount -o ro,noexec,loop,offset=1048576 sda.dd /mnt/boot
# mount | grep /mnt
/dev/mapper/VG000-root on /mnt type ext4 (ro,noexec)
/dev/mapper/VG000-usr on /mnt/usr type ext4 (ro,noexec)
/dev/mapper/VG000-var on /mnt/var type ext4 (ro,noexec)
/dev/mapper/VG000-home on /mnt/home type ext4 (ro,noexec)
/home/hal/stuff/crypt-lvm/sda.dd on /mnt/boot type ext4 (ro,noexec,loop=/dev/loop1,offset=1048576)
```

Mounting the logical volumes is as simple as calling mount on the appropriate devices. You'll notice that I'm specifying the "ro" ("read-only") option here even though the underlying loop device we created with losetup was also created as a read-only device. In my world, it never hurts to be careful. We're also using "noexec" here so that we don't accidentally end up executing any (possibly malicious) binaries from our mounted image.

If we also want to mount the /boot partition, then we need to fish that out of our original disk image. Like losetup, the mount command wants us to specify the image offset in bytes-- in fact the mount command is really calling losetup under the covers and the offset option here is being handed directly to the losetup command. So we calculate the byte offset of the start of the /boot partition from the output of mmls and use that in our list of options to mount.

Great! Now we have a fully connected file system that we can investigate using standard file system forensic tools, or even simple Unix commands like find and grep. And we've gone from a raw disk image of a an encrypted LVM2 volume to a fully operational file system. And that's pretty awesome.

Tearing It All Down

But what about when the investigation is over and we don't need to look at the image anymore? How do we close things down neatly?

The first step is to unmount any file systems you may have mounted out of the image:

```
# umount /mnt/boot
# umount /mnt/usr
# umount /mnt/var
# umount /mnt/home
# umount /mnt
```

With the file systems unmounted, we can call vgchange to deactivate volume group VG000:

```
# vgchange -a n VG000
 0 logical volume(s) in volume group "VG000" now active
```

You'll notice that the vgchange command above is nearly identical to the command we used to activate the volume, except that we use "-a n" to mark the volume as *not* active.

Next we need to tear down the /dev/mapper/unencrypted device we created with cryptsetup. As you might be guessing at this point, there's a cryptsetup sub-command for this operation:

```
# cryptsetup luksClose /dev/mapper/unencrypted
```

Last but not least, we should drop the /dev/loop0 device we created:

```
# losetup -d /dev/loop0
```

And incidentally this teardown process has been a nice review of all of the different layers we had to drive through in order to accomplish our mission!

But let's also confirm that none of our operations has modified the original disk image:

```
# md5sum sda.dd
f4c7a8d54b9b0b0b73ec03ef4cf52f42 sda.dd
```

You can either scroll back up to the top of this article, or just take my word for it. We're good.

Wrapping Up

Encryption and LVM2 can add substantial complexity to the task of accessing data in your Linux disk images. But the commands presented here should allow you to navigate these murky waters with no difficulty. And taking care with your use of the "read-only" option at various strategic points can prevent corruption of your image file.

Hal Pomeranz is an Independent IT/Security Consultant, a SANS Institute Faculty Fellow, and a GCFA. He can often be found in a darkened room, gazing into a monitor, cackling softly to himself, and muttering, "My god! It's full of stars!"