# Full Disk Encryption on OVH UEFI Servers

Sep 22, 2025 • Tiago Ilieve

For a while, it has been possible to boot OVH servers in rescue mode and, using QEMU, install an OS in any way you want with its regular installation ISO. This includes unsupported modes such as a non-standard filesystem on `/` or using full disk encryption. The only requirement was to map the disks to regular QEMU devices, such as `-hda /dev/sda` arguments. I don't remember exactly when I first did that, but there are references to this process in their community forums dating back to at least 2020. I might have seen someone mention it in a blog post a few years earlier.

Since I usually opt for their cheaper line of servers, from Kimsufi or SoYouStart, I'm used to old hardware. The server I'm currently replacing, for instance, is an Intel Xeon E3-1245 V2 from 2012 (!) with regular SATA SSDs, where the BIOS boots from the MBR. What happens is that the same custom OS installation procedure doesn't work for newer servers, which use UEFI and are powered by NVMe disks. I tried multiple combinations of:

- Regular installation as if it were a BIOS-based server with SSD drives.
- Passing them as NVMe drives using `-drive file=/dev/nvme0n1 and -device nvme`.
- UEFI installation with `-bios /usr/share/ovmf/OVMF.fd`, using the OVMF firmware.
- Creating an EFI System Partition (ESP) both on and off a RAID-1 array.
- Using the default "entire disk" automatic partitioning from the Debian installer.
- Performing a regular installation with no RAID at all.
- Cloning this regular installation onto both disks to ensure either of them would boot.

And probably a few other combinations I don't recall. What's important is that none of them worked. Every time I checked the boot logs via KVM/IPMI, I got an error like: `rEFInd - Chain on hazd drive failed. Next.` I kept wondering what could be missing, but more importantly, replicating the installation exactly as I wanted was achieving the final goal. I didn't want to install an unsupported OS. I only wanted a Debian installation with full disk encryption, which isn't supported by the OVH web-based OS installer. That's when I started looking into how to encrypt an existing Linux installation.

There are guides like Encrypt an existing Debian 12 system with LUKS, which aren't exactly wrong but are overly complicated and contain unnecessary steps. Every time I see a complicated guide, I wonder how to simplify the process. Starting from that and with the always-on-point instructions from the Arch Linux Wiki, I was able to encrypt an existing Debian installation.

## Encrypting an Existing Debian System

The process goes like this:

Do a regular Debian 13 (Trixie) installation using the OVH web installer. The most important part is to keep `/boot` separate from other partitions and on RAID-1 if you're using RAID. It will also create the ESP partition separately. The goal is to never need to touch these partitions via SSH.

After the installation is finished, log in to the new system and install the required packages to boot it. The `dropbear-initramfs` package is what allows us to unlock the encrypted root partition via SSH. For it to work, you need to set its own `authorized_keys` file, since it won't have access to anything on disk before unlocking.

```
$ sudo apt install cryptsetup-initramfs dropbear-initramfs
(...)
$ sudo vim /etc/dropbear/initramfs/authorized_keys
```

Configure the server to boot in rescue mode and restart it. After logging in again, check the current partitions to identify the data partition to be encrypted.

```
# lsblk | grep -v nbd
NAME        MAJ:MIN RM   SIZE RO TYPE  MOUNTPOINTS
nvme1n1     259:0    0 419.2G  0 disk
├─nvme1n1p1 259:1    0   511M  0 part
├─nvme1n1p2 259:2    0     1G  0 part
│ └─md2       9:2    0  1022M  0 raid1
└─nvme1n1p3 259:3    0 417.7G  0 part
  └─md3       9:3    0 835.1G  0 raid0
nvme0n1     259:4    0 419.2G  0 disk
├─nvme0n1p1 259:5    0   511M  0 part
├─nvme0n1p2 259:6    0     1G  0 part
│ └─md2       9:2    0  1022M  0 raid1
├─nvme0n1p3 259:7    0 417.7G  0 part
│ └─md3       9:3    0 835.1G  0 raid0
└─nvme0n1p4 259:8    0     2M  0 part
```

In this case, we are interested in `/dev/md3`, the root partition on top of RAID-0. Check the filesystem for errors so that other tools don't complain about it later.

```
# e2fsck -f /dev/md3
e2fsck 1.47.0 (5-Feb-2023)
Pass 1: Checking inodes, blocks, and sizes
Pass 2: Checking directory structure
Pass 3: Checking directory connectivity
Pass 4: Checking reference counts
Pass 5: Checking group summary information
root: 31979/54730752 files (1.1% non-contiguous), 3972359/218920960 blocks
```

Now comes a very important part: shrink the filesystem but not the whole partition. The goal is to leave space for the LUKS header to be created at the end of the partition.

```
# resize2fs -p -M /dev/md3
resize2fs 1.47.0 (5-Feb-2023)
Resizing the filesystem on /dev/md3 to 957980 (4k) blocks.
Begin pass 2 (max = 512733)
Relocating blocks             XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Begin pass 3 (max = 6681)
Scanning inode table          XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Begin pass 4 (max = 3388)
Updating inode references     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
The filesystem on /dev/md3 is now 957980 (4k) blocks long.
```

Next, perform the actual encryption. This is also where you set the passphrase to unlock it. This will take time, depending on the disk speed and partition size, as the process rewrites everything, including unused/free space. In this case, it took a little over 25 minutes to encrypt the 835GB partition.

```
# cryptsetup reencrypt --encrypt --reduce-device-size 32M /dev/md3

WARNING!
========
This will overwrite data on LUKS2-temp-03ea47d5-415b-4622-9510-5b7340d6c557.new i

Are you sure? (Type 'yes' in capital letters): YES
Enter passphrase for LUKS2-temp-03ea47d5-415b-4622-9510-5b7340d6c557.new:
Verify passphrase:
Finished, time 25m20s,  835 GiB written, speed 562.6 MiB/s
```

When encryption finishes, open it as a regular device and expand the filesystem to fill the partition again. This will use all available space minus what was reserved for the LUKS header.

```
# cryptsetup open /dev/md3 md3_crypt
Enter passphrase for /dev/md3:
# resize2fs /dev/mapper/md3_crypt
resize2fs 1.47.0 (5-Feb-2023)
Resizing the filesystem on /dev/mapper/md3_crypt to 218916864 (4k) blocks.
The filesystem on /dev/mapper/md3_crypt is now 218916864 (4k) blocks long.
```

Mount the device and update both `/etc/crypttab` and `/etc/fstab`. The former should refer to the filesystem UUID, but the latter can just point to the `/dev/mapper/md3_crypt` device, since it will use the name defined in crypttab.

```
# mount /dev/mapper/md3_crypt /mnt/
```

```
# blkid | grep /dev/md3
/dev/md3: UUID="a9c05676-6aa5-4a7b-acc9-a5eca5de4fed" TYPE="crypto_LUKS"
```

```
# cat /mnt/etc/crypttab
# <target name> <source device>         <key file>      <options>
md3_crypt UUID=a9c05676-6aa5-4a7b-acc9-a5eca5de4fed none luks,discard
```

```
# cat /mnt/etc/fstab
/dev/mapper/md3_crypt   /       ext4    defaults        0       1
UUID=5bc9cb1c-6607-429d-9219-3675ee773b12       /boot   ext4    defaults        0
LABEL=EFI_SYSPART       /boot/efi       vfat    defaults        0       1
```

The last step is to bind-mount the system directories, plus the actual `/boot`, and update the initramfs again in the chroot. This is required for two reasons:

- It needs to grab the SSH `authorized_keys` file defined earlier in this process.
- It needs to be aware of the updated `crypttab` file, which tells it what to unlock during boot.

```
# mount --bind /dev/ /mnt/dev/
# mount --bind /proc/ /mnt/proc/
# mount --bind /sys/ /mnt/sys/
# mount /dev/md2 /mnt/boot/
# chroot /mnt/
# update-initramfs -u -k all
update-initramfs: Generating /boot/initrd.img-6.12.43+deb13-amd64
```

After that, exit the chroot, reboot, and log in via SSH as `root` once your machine is online and responding to pings. Then, after running `cryptroot-unlock` and entering the proper passphrase, the machine will mount the encrypted device and proceed with the boot.

```
To unlock root partition, and maybe others like swap, run `cryptroot-unlock`.


BusyBox v1.37.0 (Debian 1:1.37.0-6+b3) built-in shell (ash)
Enter 'help' for a list of built-in commands.

~ # cryptroot-unlock
Please unlock disk md3_crypt:
cryptsetup: md3_crypt set up successfully
```

Finally, the machine should now be online and running with full disk encryption, except for `/boot` and the ESP partition.

```
$ sudo lsblk
NAME          MAJ:MIN RM   SIZE RO TYPE  MOUNTPOINTS
nvme1n1       259:0    0 419.2G  0 disk
├─nvme1n1p1   259:1    0   511M  0 part  /boot/efi
├─nvme1n1p2   259:2    0     1G  0 part
│ └─md2         9:2    0  1022M  0 raid1 /boot
└─nvme1n1p3   259:3    0 417.7G  0 part
  └─md3         9:3    0 835.1G  0 raid0
    └─md3_crypt 253:0    0 835.1G  0 crypt /
nvme0n1       259:4    0 419.2G  0 disk
├─nvme0n1p1   259:5    0   511M  0 part
├─nvme0n1p2   259:6    0     1G  0 part
│ └─md2         9:2    0  1022M  0 raid1 /boot
├─nvme0n1p3   259:7    0 417.7G  0 part
│ └─md3         9:3    0 835.1G  0 raid0
│   └─md3_crypt 253:0    0 835.1G  0 crypt /
└─nvme0n1p4   259:8    0     2M  0 part
```

The process may be a bit more involved than installing the OS from scratch using built-in encryption options. Still, it's not too complex, provided you don't skip any steps. It's definitely better than running a machine outside of your physical reach that stores everything in plain text.