Initial Configuration

Table of contents

The following pages provide instructions regarding general configuration of the BlueField DPU.

- Modes of Operation
- System Configuration and Services
- Host-side Interface Configuration
- Secure Boot

Modes of Operation

The NVIDIA® BlueField® DPU has several modes of operation:

- DPU mode, or embedded function (ECPF) ownership, where the embedded Arm system controls the NIC resources and data path (**default**)
- Zero-trust mode which is an extension of the ECPF ownership with additional restrictions on the host side
- NIC mode where the DPU behaves exactly like an adapter card from the perspective of the external host

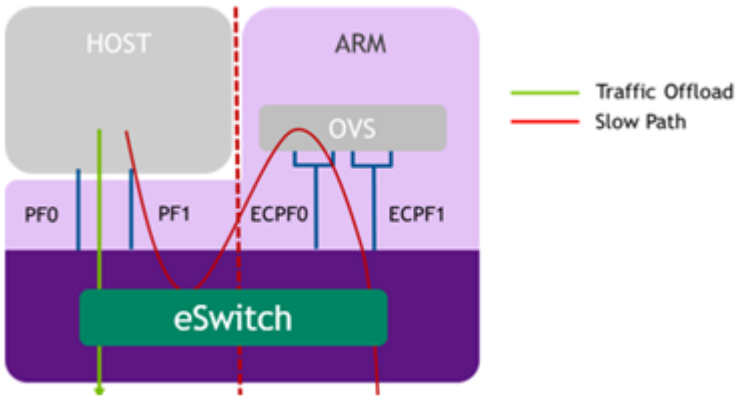
DPU Mode

This mode, known also as embedded CPU function ownership (ECPF) mode, is the default mode for BlueField DPU.

In DPU mode, the NIC resources and functionality are owned and controlled by the embedded Arm subsystem. All network communication to the host flows through a virtual switch control plane hosted on the Arm cores, and only then proceeds to the host. While working in this mode, the DPU is the trusted function managed by the data center and host administrator—to load network drivers, reset an interface, bring an interface up and down, update the firmware, and change the mode of operation on the DPU device.

A network function is still exposed to the host, but it has limited privileges. In particular:

1. The driver on the host side can only be loaded after the driver on the DPU has loaded and completed NIC configuration.
2. All ICM (Interface Configuration Memory) is allocated by the ECPF and resides in the DPU's memory.
3. The ECPF controls and configures the NIC embedded switch which means that traffic to and from the host (DPU) interface always lands on the Arm side.



When the server and DPU are initiated, the networking to the host is blocked until the virtual switch on the DPU is loaded. Once it is loaded, traffic to the host is allowed by default.

There are two ways to pass traffic to the host interface: Either using representors to forward traffic to the host (every packet to/from the host would be handled also by the network interface on the embedded Arm side) or push rules to the embedded switch which allows and offloads this traffic.

In DPU mode, OpenSM must be run from the DPU side (not the host side). Also, management tools (e.g., sminfo, ibdev2netdev, ibnetdiscover) can only be run from the DPU side (not from the host side).

Zero-trust Mode

Zero-trust mode is a specialization of DPU mode which implements an additional layer of security where the host system administrator is prevented from accessing the DPU from the host. Once zero-trust mode is enabled, the data center administrator should control the DPU entirely through the Arm cores and/or BMC connection instead of through the host.

For security and isolation purposes, it is possible to restrict the host from performing operations that can compromise the DPU. The following operations can be restricted individually when changing the DPU host to zero-trust mode:

- Port ownership – the host cannot assign itself as port owner
- Hardware counters – t he host does not have access to hardware counters
- Tracer functionality is blocked
- RShim interface is blocked
- Firmware flash is restricted

Enabling Zero-trust Mode

To enable host restriction:

1. Start the MST service.
2. Set zero-trust mode. From the Arm side, run:

```
$ sudo mlxprivhost -d /dev/mst/<device> r --disable_rshim --
disable_tracer --disable_counter_rd --disable_port_owner
```

> **Note**
>
> Graceful shutdown and p ower cycle are required if any `--disable_*` flags are used.

Disabling Zero-trust Mode

To disable host restriction, set the mode to privileged. Run:

```
$ sudo mlxprivhost -d /dev/mst/<device> p
```

The configuration takes effect immediately.

> **Note**

Graceful shutdown and p ower cycle are required when reverting to privileged mode if host restriction has been applied using any `--disable_*` flags.

NIC Mode

In this mode, the DPU behaves exactly like an adapter card from the perspective of the external host.

Note

The following instructions presume the DPU to operate in DPU mode. If the DPU is operating in zero-trust mode, please return to DPU mode before continuing.

NIC Mode for BlueField-3

Note

When BlueField-3 is configured to operate in NIC mode, Arm OS will not boot.

NIC mode for BlueField-3 saves power, improves device performance, and improves the host memory footprint.

Configuring NIC Mode on BlueField-3 from Linux

Enabling NIC Mode from Linux

Before moving to NIC mode, make sure you are operating in DPU mode by running:

```
host/dpu> sudo mlxconfig -d /dev/mst/mt41692_pciconf0 -e q
```

The output should have `INTERNAL_CPU_MODEL= EMBBEDDED_CPU(1)` and `EXP_ROM_UEFI_ARM_ENABLE = True (1)` (default).

To enable NIC mode from DPU mode:

1. Run the following on the host or Arm:

   ```
   host/dpu> sudo mlxconfig -d /dev/mst/mt41692_pciconf0 s
   INTERNAL_CPU_MODEL=1 INTERNAL_CPU_OFFLOAD_ENGINE=1
   ```

2. Perform a graceful shutdown and power cycle the host.

Disabling NIC Mode from Linux

To return to DPU mode from NIC mode:

1. Run the following on the host:

   ```
   host> sudo mlxconfig -d /dev/mst/mt41692_pciconf0 s
   INTERNAL_CPU_MODEL=1 INTERNAL_CPU_OFFLOAD_ENGINE=0
   ```

2. Perform a graceful shutdown and power cycle the host.

Configuring NIC Mode on BlueField-3 from UEFI

1. Access the Arm UEFI menu by pressing the Esc button twice

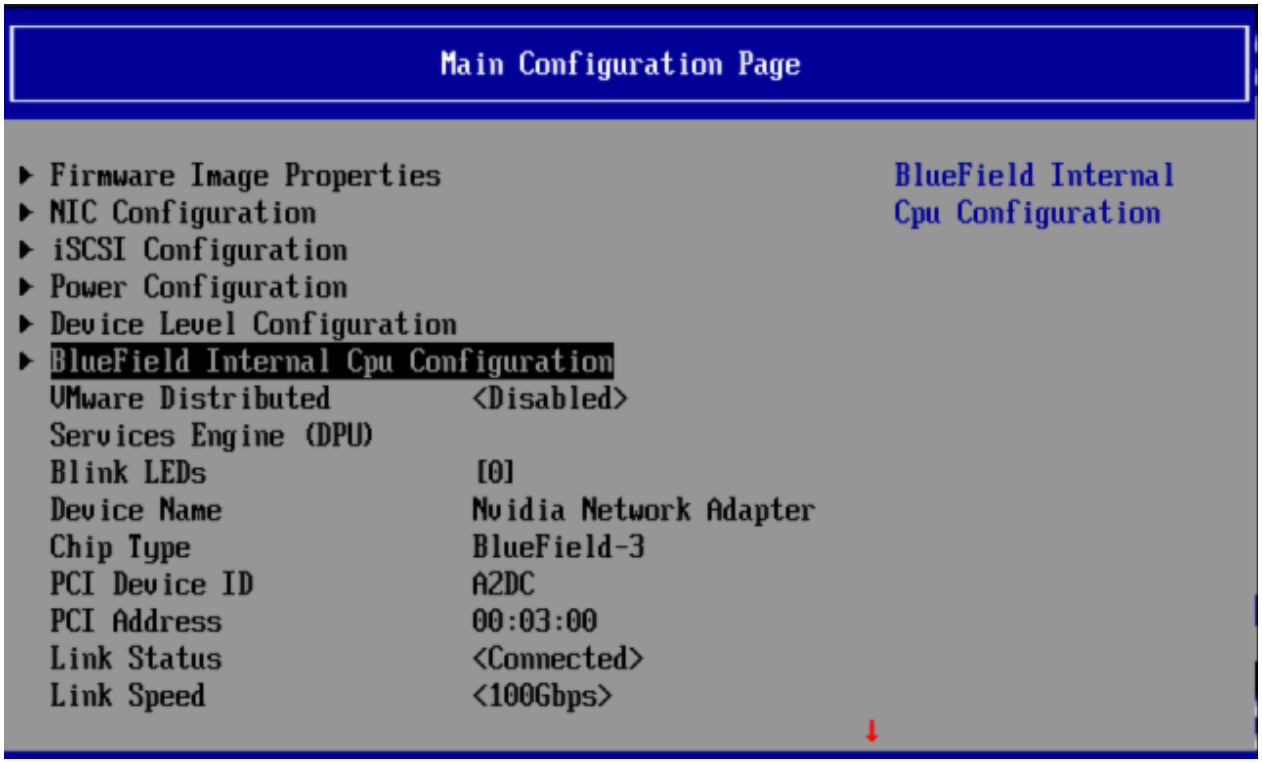
2. Select "Device Manager".

3. Select "Network Device List".

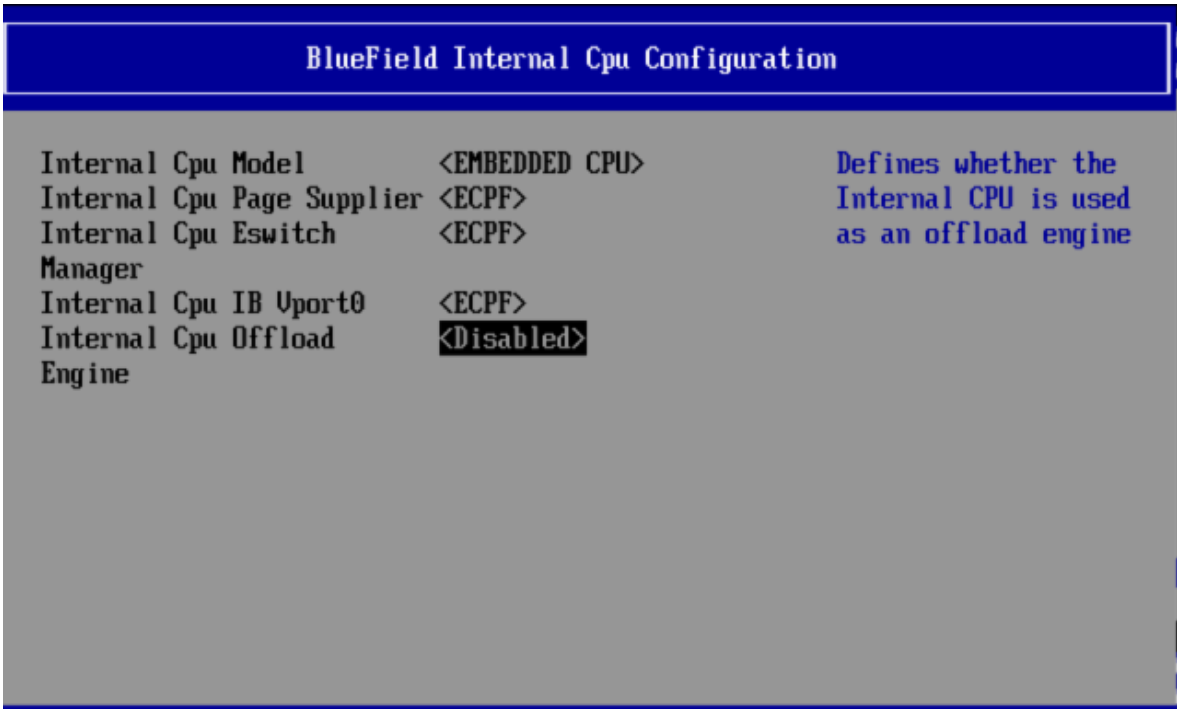
4. Select the network device that presents the uplink (i.e., select the device with the uplink MAC address).

5. Select "NVIDIA Network adapter - $<uplink-mac>".

6. Select "BlueField Internal Cpu Configuration".



   - To enable NIC mode, set "Internal Cpu Offload Engine" to "Disabled".
   - To switch back to DPU mode, set "Internal Cpu Offload Engine" to "Enabled".



Updating ATF and UEFI in BlueField-3 NIC Mode

Once in NIC mode, updating ATF and UFEI can be done using `preboot-install.bfb` by running:

```
# bfb-install --bfb <BlueField-BSP>.bfb --rshim rshim0
```

NIC Mode for BlueField-2

In this mode, the ECPFs on the Arm side are not functional but the user is still able to access the Arm system and update `mlxconfig` options.

> **Note**

When NIC mode is enabled, the drivers and services on the Arm are no longer functional.

Enabling NIC Mode on BlueField-2

To enable NIC mode from DPU mode:

1. Run the following from the x86 host side:

```
$ mst start
$ mlxconfig -d /dev/mst/<device> s INTERNAL_CPU_MODEL=1 \
INTERNAL_CPU_PAGE_SUPPLIER=1 \
INTERNAL_CPU_ESWITCH_MANAGER=1 \
INTERNAL_CPU_IB_VPORT0=1 \
INTERNAL_CPU_OFFLOAD_ENGINE=1
```

> **Note**
>
> To restrict RShim PF (optional), make sure to configure `INTERNAL_CPU_RSHIM=1` as part of the `mlxconfig` command.

2. Perform a graceful shutdown and power cycle the host.

> **Note**
>
> Multi-host is not supported when the DPU is operating in NIC mode.

Note

To obtain firmware BINs for BlueField-2 devices, please refer to the BlueField-2 firmware download page.

Disabling NIC Mode on BlueField-2

To change from NIC mode back to DPU mode:

1. Install and start the RShim driver on the host.
2. Disable NIC mode. Run:

```
$ mst start
$ mlxconfig -d /dev/mst/<device> s INTERNAL_CPU_MODEL=1 \
INTERNAL_CPU_PAGE_SUPPLIER=0 \
INTERNAL_CPU_ESWITCH_MANAGER=0 \
INTERNAL_CPU_IB_VPORT0=0 \
INTERNAL_CPU_OFFLOAD_ENGINE=0
```

Note

If `INTERNAL_CPU_RSHIM=1`, then make sure to configure `INTERNAL_CPU_RSHIM=0` as part of the `mlxconfig` command.

3. Perform a graceful shutdown and power cycle the host.

System Configuration and Services

This page provides information on system services and scripts based on the default DPU OS (i.e., Ubuntu).

First Boot After BFB Installation

During the first boot, the cloud-init service configures the system based on the data provided in the following files:

- `/var/lib/cloud/seed/nocloud-net/network-config` – network interface configuration
- `/var/lib/cloud/seed/nocloud-net/user-data` – default users and commands to run on the first boot

RDMA and ConnectX Driver Initialization

RDMA and NVIDIA® ConnectX® drivers are loaded upon boot by the `openibd.service`.

> **Note**
>
> The `mlx5_core` kernel module is loaded automatically by the kernel as a registered device driver.

One of the kernel modules loaded by the `openibd.service`, `ib_umad`, triggers modprobe rule from `/etc/modprobe.d/mlnx-bf.conf` file that runs the `/sbin/mlnx_bf_configure` script. See Default Ports and OVS Configuration for more information.

Firewall Configuration

The Ubuntu BFB image includes the following firewall configuration by default (enabled):

```
$ cat /etc/iptables/rules.v4

*mangle
:PREROUTING ACCEPT [45:3582]
:INPUT ACCEPT [45:3582]
:FORWARD ACCEPT [0:0]
:OUTPUT ACCEPT [36:4600]
:POSTROUTING ACCEPT [36:4600]
:KUBE-IPTABLES-HINT - [0:0]
:KUBE-KUBELET-CANARY - [0:0]
COMMIT
*filter
:INPUT ACCEPT [41:3374]
:FORWARD ACCEPT [0:0]
:OUTPUT ACCEPT [32:3672]
:DOCKER-USER - [0:0]
:KUBE-FIREWALL - [0:0]
:KUBE-KUBELET-CANARY - [0:0]
:LOGGING - [0:0]
:POSTROUTING - [0:0]
:PREROUTING - [0:0]
-A INPUT -j KUBE-FIREWALL
-A INPUT -p tcp -m tcp --dport 111 -j REJECT --reject-with icmp-
port-unreachable
-A INPUT -p udp -m udp --dport 111 -j REJECT --reject-with icmp-
port-unreachable
-A INPUT -i lo -m comment --comment MD_IPTABLES -j ACCEPT
-A INPUT -d 127.0.0.0/8 -m mark --mark 0xb -m comment --comment
MD_IPTABLES -j DROP
-A INPUT -m mark --mark 0xb -m state --state RELATED,ESTABLISHED
-m comment --comment MD_IPTABLES -j ACCEPT
```

```
-A INPUT -p tcp -m tcp ! --dport 22 ! --tcp-flags FIN,SYN,RST,ACK
SYN -m mark --mark 0xb -m state --state NEW -m comment --comment
MD_IPTABLES -j DROP
-A INPUT -f -m mark --mark 0xb -m comment --comment MD_IPTABLES -
j DROP
-A INPUT -p tcp -m tcp --tcp-flags FIN,SYN,RST,PSH,ACK,URG
FIN,SYN,RST,PSH,ACK,URG -m mark --mark 0xb -m comment --comment
MD_IPTABLES -j DROP
-A INPUT -p tcp -m tcp --tcp-flags FIN,SYN,RST,PSH,ACK,URG NONE -
m mark --mark 0xb -m comment --comment MD_IPTABLES -j DROP
-A INPUT -m mark --mark 0xb -m state --state INVALID -m comment -
-comment MD_IPTABLES -j DROP
-A INPUT -p tcp -m tcp --tcp-flags RST RST -m mark --mark 0xb -m
hashlimit --hashlimit-above 2/sec --hashlimit-burst 2 --
hashlimit-mode srcip --hashlimit-name hashlimit_0 --hashlimit-
htable-expire 30000 -m comment --comment MD_IPTABLES -j DROP
-A INPUT -p tcp -m mark --mark 0xb -m state --state NEW -m
hashlimit --hashlimit-above 50/sec --hashlimit-burst 50 --
hashlimit-mode srcip --hashlimit-name hashlimit_1 --hashlimit-
htable-expire 30000
-m comment --comment MD_IPTABLES -j DROP
-A INPUT -p tcp -m mark --mark 0xb -m conntrack --ctstate NEW -m
hashlimit --hashlimit-above 60/sec --hashlimit-burst 20 --
hashlimit-mode srcip --hashlimit-name hashlimit_2 --hashlimit-
htable-expire 30000 -m comment --comment MD_IPTABLES -j DROP
-A INPUT -m mark --mark 0xb -m recent --rcheck --seconds 86400 --
name portscan --mask 255.255.255.255 --rsource -m comment --
comment MD_IPTABLES -j DROP
-A INPUT -m mark --mark 0xb -m recent --remove --name portscan --
mask 255.255.255.255 --rsource -m comment --comment MD_IPTABLES
-A INPUT -p tcp -m tcp --dport 22 -m mark --mark 0xb -m conntrack
--ctstate NEW -m recent --set --name DEFAULT --mask
255.255.255.255 --rsource -m comment --comment MD_IPTABLES
-A INPUT -p tcp -m tcp --dport 22 -m mark --mark 0xb -m conntrack
--ctstate NEW -m recent --update --seconds 60 --hitcount 50 --
```

```
name DEFAULT --mask 255.255.255.255 --rsource -m comment --
comment MD_IPTABLES -j DROP

-A INPUT -p tcp -m tcp --dport 443 -m mark --mark 0xb -m
conntrack --ctstate NEW -m recent --set --name DEFAULT --mask
255.255.255.255 --rsource -m comment --comment MD_IPTABLES
-A INPUT -p tcp -m tcp --dport 443 -m mark --mark 0xb -m
conntrack --ctstate NEW -m recent --update --seconds 60 --
hitcount 10 --name DEFAULT --mask 255.255.255.255 --rsource -m
comment --comment MD_IPTABLES -j DROP
-A INPUT -p udp -m udp --dport 161 -m mark --mark 0xb -m
conntrack --ctstate NEW -m recent --set --name DEFAULT --mask
255.255.255.255 --rsource -m comment --comment MD_IPTABLES
-A INPUT -p udp -m udp --dport 161 -m mark --mark 0xb -m
conntrack --ctstate NEW -m recent --update --seconds 60 --
hitcount 100 --name DEFAULT --mask 255.255.255.255 --rsource -m
comment --comment MD_IPTABLES -j DROP
-A INPUT -p tcp -m tcp --dport 22 -m mark --mark 0xb -m conntrack
--ctstate NEW,ESTABLISHED -m comment --comment MD_IPTABLES -j
ACCEPT
-A INPUT -p tcp -m tcp --dport 443 -m mark --mark 0xb -m
conntrack --ctstate NEW,ESTABLISHED -m comment --comment
MD_IPTABLES -j ACCEPT
-A INPUT -p tcp -m tcp --dport 179 -m mark --mark 0xb -m
conntrack --ctstate NEW,ESTABLISHED -m comment --comment
MD_IPTABLES -j ACCEPT
-A INPUT -p udp -m udp --dport 68 -m mark --mark 0xb -m conntrack
--ctstate NEW,ESTABLISHED -m comment --comment MD_IPTABLES -j
ACCEPT
-A INPUT -p udp -m udp --dport 122 -m mark --mark 0xb -m
conntrack --ctstate NEW,ESTABLISHED -m comment --comment
MD_IPTABLES -j ACCEPT
-A INPUT -p udp -m udp --dport 161 -m mark --mark 0xb -m
conntrack --ctstate NEW,ESTABLISHED -m comment --comment
MD_IPTABLES -j ACCEPT
```

```
-A INPUT -p udp -m udp --dport 6306 -m mark --mark 0xb -m
conntrack --ctstate NEW,ESTABLISHED -m comment --comment
MD_IPTABLES -j ACCEPT
-A INPUT -p udp -m udp --dport 69 -m mark --mark 0xb -m conntrack
--ctstate NEW,ESTABLISHED -m comment --comment MD_IPTABLES -j
ACCEPT
-A INPUT -p udp -m udp --dport 389 -m mark --mark 0xb -m
conntrack --ctstate NEW,ESTABLISHED -m comment --comment
MD_IPTABLES -j ACCEPT
-A INPUT -p tcp -m tcp --dport 389 -m mark --mark 0xb -m
conntrack --ctstate NEW,ESTABLISHED -m comment --comment
MD_IPTABLES -j ACCEPT
-A INPUT -p udp -m udp --dport 1812:1813 -m mark --mark 0xb -m
conntrack --ctstate NEW,ESTABLISHED -m comment --comment
MD_IPTABLES -j ACCEPT
-A INPUT -p udp -m udp --dport 49 -m mark --mark 0xb -m conntrack
--ctstate NEW,ESTABLISHED -m comment --comment MD_IPTABLES -j
ACCEPT
-A INPUT -p tcp -m tcp --dport 49 -m mark --mark 0xb -m conntrack
--ctstate NEW,ESTABLISHED -m comment --comment MD_IPTABLES -j
ACCEPT
-A INPUT -p udp -m udp --sport 53 -m mark --mark 0xb -m conntrack
--ctstate NEW,ESTABLISHED -m comment --comment MD_IPTABLES -j
ACCEPT
-A INPUT -p tcp -m tcp --sport 53 -m mark --mark 0xb -m conntrack
--ctstate NEW,ESTABLISHED -m comment --comment MD_IPTABLES -j
ACCEPT
-A INPUT -p udp -m udp --dport 500 -m mark --mark 0xb -m
conntrack --ctstate NEW,ESTABLISHED -m comment --comment
MD_IPTABLES -j ACCEPT
-A INPUT -p udp -m udp --dport 4500 -m mark --mark 0xb -m
conntrack --ctstate NEW,ESTABLISHED -m comment --comment
MD_IPTABLES -j ACCEPT
-A INPUT -p udp -m udp --dport 1293 -m mark --mark 0xb -m
conntrack --ctstate NEW,ESTABLISHED -m comment --comment
MD_IPTABLES -j ACCEPT
```

```
-A INPUT -p tcp -m tcp --dport 1293 -m mark --mark 0xb -m
conntrack --ctstate NEW,ESTABLISHED -m comment --comment
MD_IPTABLES -j ACCEPT
-A INPUT -p udp -m udp --dport 1707 -m mark --mark 0xb -m
conntrack --ctstate NEW,ESTABLISHED -m comment --comment
MD_IPTABLES -j ACCEPT
-A INPUT -p tcp -m tcp --dport 1707 -m mark --mark 0xb -m
conntrack --ctstate NEW,ESTABLISHED -m comment --comment
MD_IPTABLES -j ACCEPT
-A INPUT -i lo -p udp -m udp --dport 3786 -m conntrack --ctstate
NEW,ESTABLISHED -m comment --comment MD_IPTABLES -j ACCEPT
-A INPUT -i lo -p udp -m udp --dport 33000 -m conntrack --ctstate
NEW,ESTABLISHED -m comment --comment MD_IPTABLES -j ACCEPT
-A INPUT -p icmp -m mark --mark 0xb -m comment --comment
MD_IPTABLES -j ACCEPT
-A INPUT -p udp -m udp --sport 5353 --dport 5353 -m mark --mark
0xb -m conntrack --ctstate NEW,ESTABLISHED -m comment --comment
MD_IPTABLES -j ACCEPT
-A INPUT -p udp -m udp --dport 33434:33523 -m mark --mark 0xb -m
comment --comment MD_IPTABLES -j REJECT --reject-with icmp-port-
unreachable
-A INPUT -p udp -m udp --dport 123 -m mark --mark 0xb -m
conntrack --ctstate NEW,ESTABLISHED -m comment --comment
MD_IPTABLES -j ACCEPT
-A INPUT -p udp -m udp --dport 514 -m mark --mark 0xb -m
conntrack --ctstate NEW,ESTABLISHED -m comment --comment
MD_IPTABLES -j ACCEPT
-A INPUT -p udp -m udp --dport 67 -m mark --mark 0xb -m conntrack
--ctstate NEW,ESTABLISHED -m comment --comment MD_IPTABLES -j
ACCEPT
-A INPUT -p tcp -m tcp --dport 60102 -m mark --mark 0xb -m
conntrack --ctstate NEW,ESTABLISHED -m comment --comment
"MD_IPTABLES: Feature HA port" -j ACCEPT
-A INPUT -m mark --mark 0xb -m comment --comment MD_IPTABLES -j
LOGGING
-A FORWARD -j DOCKER-USER
```

```
-A OUTPUT -o oob_net0 -m comment --comment MD_IPTABLES -j ACCEPT
-A DOCKER-USER -j RETURN

-A LOGGING -m mark --mark 0xb -m comment --comment MD_IPTABLES -j
NFLOG --nflog-prefix  "IPTables-Dropped: " --nflog-group 3
-A LOGGING -m mark --mark 0xb -m comment --comment MD_IPTABLES -j
DROP
-A PREROUTING -i oob_net0 -m comment --comment MD_IPTABLES -j
MARK --set-xmark 0xb/0xffffffff
-A PREROUTING -p tcp -m tcpmss ! --mss 536:65535 -m tcp ! --dport
22 -m mark --mark 0xb -m conntrack --ctstate NEW -m comment --
comment MD_IPTABLES -j DROP
COMMIT
*nat
:PREROUTING ACCEPT [1:320]
:INPUT ACCEPT [1:320]
:OUTPUT ACCEPT [8:556]
:POSTROUTING ACCEPT [8:556]
:KUBE-KUBELET-CANARY - [0:0]
:KUBE-MARK-DROP - [0:0]
:KUBE-MARK-MASQ - [0:0]
:KUBE-POSTROUTING - [0:0]
-A POSTROUTING -m comment --comment "kubernetes postrouting
rules" -j KUBE-POSTROUTING
-A KUBE-MARK-DROP -j MARK --set-xmark 0x8000/0x8000
-A KUBE-MARK-MASQ -j MARK --set-xmark 0x4000/0x4000
-A KUBE-POSTROUTING -m mark ! --mark 0x4000/0x4000 -j RETURN
-A KUBE-POSTROUTING -j MARK --set-xmark 0x4000/0x0
-A KUBE-POSTROUTING -m comment --comment "kubernetes service
traffic requiring SNAT" -j MASQUERADE --random-fully
COMMIT
```

This configuration is provided by the `bf-release` package and is installed during the first boot of the Ubuntu OS after the BFB installation using the `cloud-init` service and the `/var/lib/cloud/seed/nocloud-net/user-data` configuration file.

To disable this default firewall configuration after OS is UP, run:

```
$ rm -f /etc/iptables/rules.v4
$ iptables -F
```

To disable this default firewall configuration during the BFB installation, use `bf.cfg` with the following command in the `bfb_modify_os` function:

```
bfb_modify_os()
{
    perl -ni -e "if(/^write_files:/../^users/) {next unless
m{^users}; print} else {print}" /mnt/var/lib/cloud/seed/nocloud-
net/user-data
}
```

Host-side Interface Configuration

The NVIDIA® BlueField® DPU registers on the host OS a "DMA controller" for DPU management over PCIe. This can be verified by running the following:

```
#  lspci -d 15b3: | grep 'SoC Management Interface'
27:00.2 DMA controller: Mellanox Technologies MT42822 BlueField-2
SoC Management Interface (rev 01)
```

A special driver called RShim must be installed and run to expose the various BlueField management interfaces on the host OS. Refer to section "Install RShim on Host" for information on how to obtain and install the host-side RShim driver.

When the RShim driver runs properly on the host side, a sysfs device, `/dev/rshim0/*`, and a virtual Ethernet interface, `tmfifo_net0`, become available. The following is an example for querying the status of the RShim driver on the host side:

```
# systemctl status rshim
  rshim.service - rshim driver for BlueField SoC
     Loaded: loaded (/lib/systemd/system/rshim.service; disabled;
vendor preset: enabled)
     Active: active (running) since Tue 2022-05-31 14:57:07 IDT;
1 day 1h ago
       Docs: man:rshim(8)
    Process: 90322 ExecStart=/usr/sbin/rshim $OPTIONS
(code=exited, status=0/SUCCESS)
   Main PID: 90323 (rshim)
      Tasks: 11 (limit: 76853)
     Memory: 3.3M
```

```
     CGroup: /system.slice/rshim.service
             90323 /usr/sbin/rshim
May 31 14:57:07 …  systemd[1]: Starting rshim driver for
BlueField SoC...
May 31 14:57:07  … systemd[1]: Started rshim driver for BlueField
SoC.
May 31 14:57:07  … rshim[90323]: Probing pcie-0000:a3:00.2(vfio)
May 31 14:57:07  … rshim[90323]: Create rshim pcie-0000:a3:00.2
May 31 14:57:07  … rshim[90323]: rshim pcie-0000:a3:00.2 enable
May 31 14:57:08  … rshim[90323]: rshim0 attached
```

If the RShim device does not appear, refer to section "RShim Troubleshooting and How-Tos".

Virtual Ethernet Interface

On the host, the RShim driver exposes a virtual Ethernet device called `tmfifo_net0`. This virtual Ethernet can be thought of as a peer-to-peer tunnel connection between the host and the DPU OS. The DPU OS also configures a similar device. The DPU OS's BFB images are customized to configure the DPU side of this connection with a preset IP of 192.168.100.2/30. It is up to the user to configure the host side of this connection. Configuration procedures vary for different OSs.

The following example configures the host side of `tmfifo_net0` with a static IP and enables IPv4-based communication to the DPU OS:

```
#  ip addr add dev tmfifo_net0 192.168.100.1/30
```

> **Note**
>
> For instructions on persistent IP configuration of the tmfifo_net0 interface, refer to step "Assign a static IP to tmfifo_net0" under "Updating Repo Package on Host Side".

Logging in from the host to the DPU OS is now possible over the virtual Ethernet. For example:

```
ssh ubuntu@192.168.100.2
```

RShim Support for Multiple DPUs

Multiple DPUs may connect to the same host machine. When the RShim driver is loaded and operating correctly, each board is expected to have its own device directory on sysfs, `/dev/rshim<N>`, and a virtual Ethernet device, `tmfifo_net<N>`.

The following are some guidelines on how to set up the RShim virtual Ethernet interfaces properly if multiple DPUs are installed in the host system.

There are two methods to manage multiple `tmfifo_net` interfaces on a Linux platform:

- Using a bridge, with all `tmfifo_net<N>` interfaces on the bridge – the bridge device bares a single IP address on the host while each DPU has unique IP in the same subnet as the bridge
- Directly over the individual `tmfifo_net<N>` – each interface has a unique subnet IP and each DPU has a corresponding IP per subnet

Whichever method is selected, the host-side `tmfifo_net` interfaces should have different MAC addresses, which can be:

- Configured using `ifconfig`. For example:

  ```
  $ ifconfig tmfifo_net0 192.168.100.1/24 hw ether
  02:02:02:02:02:02
  ```

- Or saved in configuration via the `/udev/rules` as can be seen later in this section.

In addition, each Arm-side `tmfifo_net` interface must have a unique MAC and IP address configuration, as BlueField OS comes uniformly pre-configured with a generic

MAC, and 192.168.100.2. The latter must be configured in each DPU manually or by DPU customization scripts during BlueField OS installation.

Multi-board Management Example

This example deals with two BlueField DPUs installed on the same server (the process is similar for more DPUs).

This example assumes that the RShim package has been installed on the host server.

Configuring Management Interface on Host

> **Note**
>
> This example is relevant for CentOS/RHEL operating systems only.

1. Create a `bf_tmfifo` interface under `/etc/sysconfig/network-scripts`. Run:

```
vim /etc/sysconfig/network-scripts/ifcfg-br_tmfifo
```

2. Inside `ifcfg-br_tmfifo`, insert the following content:

```
DEVICE="br_tmfifo"
BOOTPROTO="static"
IPADDR="192.168.100.1"
NETMASK="255.255.255.0"
ONBOOT="yes"
TYPE="Bridge"
```

3. Create a configuration file for the first BlueField DPU, `tmfifo_net0`. Run:

```
vim /etc/sysconfig/network-scripts/ifcfg-tmfifo_net0
```

4. Inside `ifcfg-tmfifo_net0`, insert the following content:

```
DEVICE=tmfifo_net0
BOOTPROTO=none
ONBOOT=yes
NM_CONTROLLED=no
BRIDGE=br_tmfifo
```

5. Create a configuration file for the second BlueField DPU, `tmfifo_net1`. Run:

```
DEVICE=tmfifo_net1
BOOTPROTO=none
ONBOOT=yes
NM_CONTROLLED=no
BRIDGE=br_tmfifo
```

6. Create the rules for the `tmfifo_net` interfaces. Run:

```
vim /etc/udev/rules.d/91-tmfifo_net.rules
```

7. Restart the network for the changes to take effect. Run:

```
# /etc/init.d/network restart
Restarting network (via systemctl):                        [  OK  ]
```

Configuring BlueField DPU Side

BlueField DPUs arrive with the following factory default configurations for tmfifo_net0.

Address	Value
MAC	`00:1a:ca:ff:ff:01`
IP	`192.168.100.2`

Therefore, if you are working with more than one DPU, you must change the default MAC and IP addresses.

Updating RShim Network MAC Address

> **Note**
>
> This procedure is relevant for Ubuntu/Debian (`sudo` needed), and CentOS BFBs. The procedure only affects the `tmfifo_net0` on the Arm side.

1. Use a Linux console application (e.g. screen or minicom) to log into each BlueField. For example:

```
# sudo screen /dev/rshim<0|1>/console 115200
```

2. Create a configuration file for `tmfifo_net0` MAC address. Run:

```
# sudo vi /etc/bf.cfg
```

3. Inside `bf.cfg`, insert the new MAC:

```
NET_RSHIM_MAC=00:1a:ca:ff:ff:03
```

4. Apply the new MAC address. Run:

```
sudo bfcfg
```

5. Repeat this procedure for the second BlueField DPU (using a different MAC address).

> **Info**
>
> Arm must be rebooted for this configuration to take effect. It is recommended to update the IP address before you do that to avoid unnecessary reboots.

> **Note**
>
> For comprehensive list of the supported parameters to customize `bf.cfg` during BFB installation, refer to section "bf.cfg Parameters".

Updating IP Address

For Ubuntu:

1. Access the file `50-cloud-init.yaml` and modify the tmfifo_net0 IP address:

```
sudo vim /etc/netplan/50-cloud-init.yaml
```

```
                tmfifo_net0:
                     addresses:
                     - 192.168.100.2/30    ===>>>
192.168.100.3/30
```

2. Reboot the Arm. Run:

```
sudo reboot
```

3. Repeat this procedure for the second BlueField DPU (using a different IP address).

> **Info**
>
> Arm must be rebooted for this configuration to take effect. It is recommended to update the MAC address before you do that to avoid unnecessary reboots.

For CentOS:

1. Access the file `ifcfg-tmfifo_net0`. Run:

```
# vim /etc/sysconfig/network-scripts/ifcfg-tmfifo_net0
```

2. Modify the value for `IPADDR`:

```
IPADDR=192.168.100.3
```

3. Reboot the Arm. Run:

```
reboot
```

Or perform `netplan apply`.

4. Repeat this procedure for the second BlueField DPU (using a different IP address).

> **Info**
>
> Arm must be rebooted for this configuration to take effect. It is recommended to update the MAC address before you do that to avoid unnecessary reboots.

Permanently Changing Arm-side MAC Address

> **Note**
>
> It is assumed that the commands in this section are executed with root (or `sudo`) permission.

The default MAC address is `00:1a:ca:ff:ff:01`. It can be changed using `ifconfig` or by updating the UEFI variable as follows:

1. Log into Linux from the Arm console.
2. Run:

```
$ "ls /sys/firmware/efi/efivars".
```

3. If not mounted, run:

```
$ mount -t efivarfs none /sys/firmware/efi/efivars
$ chattr -i /sys/firmware/efi/efivars/RshimMacAddr-8be4df61-
93ca-11d2-aa0d-00e098032b8c
$ printf "\x07\x00\x00\x00\x00\x1a\xca\xff\xff\x03" > \
  /sys/firmware/efi/efivars/RshimMacAddr-8be4df61-93ca-11d2-
aa0d-00e098032b8c
```

The `printf` command sets the MAC address to `00:1a:ca:ff:ff:03` (the last six bytes of the `printf` value). Either reboot the device or reload the tmfifo driver for the change to take effect.

The MAC address can also be updated from the server host side while the Arm-side Linux is running:

1. Enable the configuration. Run:

```
# echo "DISPLAY_LEVEL 1" > /dev/rshim0/misc
```

2. Display the current setting. Run:

```
# cat /dev/rshim0/misc
DISPLAY_LEVEL   1 (0:basic, 1:advanced, 2:log)
BOOT_MODE       1 (0:rshim, 1:emmc, 2:emmc-boot-swap)
BOOT_TIMEOUT    300 (seconds)
DROP_MODE       0 (0:normal, 1:drop)
SW_RESET        0 (1: reset)
DEV_NAME        pcie-0000:04:00.2
```

```
DEV_INFO        BlueField-2(Rev 1)
PEER_MAC        00:1a:ca:ff:ff:01 (rw)
PXE_ID          0x00000000 (rw)
VLAN_ID         0 0 (rw)
```

3. Modify the MAC address. Run:

```
$ echo "PEER_MAC  xx:xx:xx:xx:xx:xx" > /dev/rshim0/misc
```

For more information and an example of the script that covers multiple DPU installation and configuration, refer to section "Installing Full DOCA Image on Multiple DPUs" of the *NVIDIA DOCA Installation Guide*.

OOB Ethernet Interface

The OOB interface is a gigabit Ethernet interface which provides TCP/IP network connectivity to the Arm cores. This interface is named `oob_net0` and is intended to be used for management traffic (e.g. file transfer protocols, SSH, etc). The Linux driver that controls this interface is named `mlxbf_gige.ko`, and is automatically loaded upon boot. This interface can be configured and monitored by use of standard tools (e.g. ifconfig, ethtool, etc). The OOB interface is subject to the following design limitations:

- Only supports 1Gb/s full-duplex setting
- Only supports GMII access to external PHY device
- Supports maximum packet size of 2KB (i.e. no support for jumbo frames)

The OOB interface can also be used for PXE boot. This OOB port is not a path for the boot stream. Any attempt to push a BFB to this port will not work. Please refer to How to use the UEFI boot menu for more information about UEFI operations related to the OOB interface.

OOB Interface MAC Address

The MAC address to be used for the OOB port is burned into Arm-accessible UPVS EEPROM during the manufacturing process. This EEPROM device is different from the SPI Flash storage device used for the NIC firmware and associated NIC MACs/GUIDs. The

value of the OOB MAC address is specific to each platform and is visible on the board-level sticker.

> **⚠ Warning**
>
> It is not recommended to reconfigure the MAC address from the MAC configured during manufacturing.

If there is a need to re-configure this MAC for any reason, follow these steps to configure a UEFI variable to hold new value for OOB MAC.:

> **ⓘ Note**
>
> The creation of an OOB MAC address UEFI variable will override the OOB MAC address defined in EEPROM, but the change can be reverted.

1. Log into Linux from the Arm console.
2. Issue the command `ls /sys/firmware/efi/efivars` to show whether efivarfs is mounted. If it is not mounted, run:

   ```
   mount -t efivarfs none /sys/firmware/efi/efivars
   ```

3. Run:

   ```
   chattr -i /sys/firmware/efi/efivars/OobMacAddr-8be4df61-93ca-11d2-aa0d-00e098032b8c
   ```

4. Set the MAC address to 00:1a:ca:ff:ff:03 (the last six bytes of the printf value).

```
printf "\x07\x00\x00\x00\x00\x1a\xca\xff\xff\x03" >
/sys/firmware/efi/efivars/OobMacAddr-8be4df61-93ca-11d2-aa0d-
00e098032b8c
```

5. Reboot the device for the change to take effect.

To revert this change and go back to using the MAC as programmed during manufacturing, follow these steps:

1. Log into UEFI from the Arm console, go to "Boot Manager" then "EFI Internal Shell".
2. Delete the OOB MAC UEFI variable. Run:

```
dmpstore -d OobMacAddr
```

3. Reboot the device by running "reset" from UEFI.
4. Log into Linux from the Arm console.
5. Issue the command `ls /sys/firmware/efi/efivars` to show whether efivarfs is mounted. If it is not mounted, run:

```
mount -t efivarfs none /sys/firmware/efi/efivars
```

6. Run:

```
chattr -i /sys/firmware/efi/efivars/OobMacAddr-8be4df61-93ca-
11d2-aa0d-00e098032b8c
```

7. Reconfigure the original MAC address burned by the manufacturer in the format `aa\bb\cc\dd\ee\ff`. Run:

```
printf "\x07\x00\x00\x00\x00\<original-MAC-address>" >
/sys/firmware/efi/efivars/OobMacAddr-8be4df61-93ca-11d2-aa0d-
00e098032b8c
```

8. Reboot the device for the change to take effect.

Supported ethtool Options for OOB Interface

The Linux driver for the OOB port supports the handling of some basic ethtool requests: get driver info, get/set ring parameters, get registers, and get statistics.

To use the ethtool options available, use the following format:

```
$ ethtool [<option>] <interface>
```

Where `<option>` may be:

- `<no-argument>` – display interface link information
- `-i` – display driver general information
- `-S` – display driver statistics
- `-d` – dump driver register set
- `-g` – display driver ring information
- `-G` – configure driver ring(s)
- `-k` – display driver offload information

- `-a` – query the specified Ethernet device for pause parameter information
- `-r` – restart auto-negotiation on the specified Ethernet device if auto-negotiation is enabled

For example:

```
$ ethtool oob_net0
Settings for oob_net0:
        Supported ports: [ TP ]
        Supported link modes:   1000baseT/Full
        Supported pause frame use: Symmetric
        Supports auto-negotiation: Yes
        Supported FEC modes: Not reported
        Advertised link modes:  1000baseT/Full
        Advertised pause frame use: Symmetric
        Advertised auto-negotiation: Yes
        Advertised FEC modes: Not reported
        Link partner advertised link modes:  1000baseT/Full
        Link partner advertised pause frame use: Symmetric
        Link partner advertised auto-negotiation: Yes
        Link partner advertised FEC modes: Not reported
        Speed: 1000Mb/s
        Duplex: Full
        Port: Twisted Pair
        PHYAD: 3
        Transceiver: internal
        Auto-negotiation: on
        MDI-X: Unknown
        Link detected: yes
```

```
$ ethtool -i oob_net0
driver: mlxbf_gige
version:
```

```
firmware-version:
expansion-rom-version:
bus-info: MLNXBF17:00
supports-statistics: yes
supports-test: no
supports-eeprom-access: no
supports-register-dump: yes
supports-priv-flags: no
```

```
# Display statistics specific to BlueField-2 design (i.e.
statistics that are not shown in the output of "ifconfig
oob0_net")
$ ethtool -S oob_net0
NIC statistics:
    hw_access_errors: 0
    tx_invalid_checksums: 0
    tx_small_frames: 1
    tx_index_errors: 0
    sw_config_errors: 0
    sw_access_errors: 0
    rx_truncate_errors: 0
    rx_mac_errors: 0
    rx_din_dropped_pkts: 0
    tx_fifo_full: 0
    rx_filter_passed_pkts: 5549
    rx_filter_discard_pkts: 4
```

IP Address Configuration for OOB Interface

The files that control IP interface configuration are specific to the Linux distribution. The udev rules file (`/etc/udev/rules.d/92-oob_net.rules`) that renames the OOB interface to `oob_net0` and is the same for Yocto, CentOS, and Ubuntu:

```
SUBSYSTEM=="net", ACTION=="add",
DEVPATH=="/devices/platform/MLNXBF17:00/net/eth[0-9]",
NAME="oob_net0"
```

The files that control IP interface configuration are slightly different for CentOS and Ubuntu:

- CentOS configuration of IP interface:
    - Configuration file for `oob_net0`: `/etc/sysconfig/network-scripts/ifcfg-oob_net0`
    - For example, use the following to enable DHCP:

```
NAME="oob_net0"
DEVICE="oob_net0"
NM_CONTROLLED="yes"
PEERDNS="yes"
ONBOOT="yes"
BOOTPROTO="dhcp"
TYPE=Ethernet
```

    - For example, to configure static IP use the following:

```
NAME="oob_net0"
DEVICE="oob_net0"
IPV6INIT="no"
NM_CONTROLLED="no"
PEERDNS="yes"
ONBOOT="yes"
BOOTPROTO="static"
IPADDR="192.168.200.2"
PREFIX=30
```

```
GATEWAY="192.168.200.1"
DNS1="192.168.200.1"
TYPE=Ethernet
```

- For Ubuntu configuration of IP interface, refer to section "Default Network Interface Configuration".

Secure Boot

These pages provide guidelines on how to operate secured NVIDIA® BlueField® DPUs. They provide UEFI secure boot references for the UEFI portion of the secure boot process.

Note

This section provides directions for illustration purposes, it does not intend to enforce or mandate any procedure about managing keys and/or production guidelines. Platform users are solely responsible of implementing secure strategies and safe approaches to manage their boot images and their associated keys and certificates.

Note

Security aspects such as key generation, key management, key protection, and certificate generation are out of the scope of this section.

Secure boot is a process which verifies each element in the boot process prior to execution, and halts or enters a special state if a verification step fails at any point during the boot. It is based on an unmodifiable ROM code which acts as the root-of-trust (RoT) and uses an off-chip public key, to authenticate the initial code which is loaded from an external non-volatile storage. The off-chip public key integrity is verified by the ROM code against an on-chip public key hash value stored in E-FUSEs. Then the authenticated code and each element in the boot process cryptographically verify the next element prior to passing execution to it. This extends the chain-of-trust (CoT) by verifying elements that have their RoT in hardware. In addition, no external intervention in the authentication process is permitted to prevent unauthorized software and firmware from being loaded. There should be no way to interrupt or bypass the RoT with runtime changes.

Supported BlueField DPUs

Secured BlueField devices have pre-installed software and firmware signed with NVIDIA signing keys. The on-chip public key hash is programmed into E-FUSEs.

To verify whether the DPU in your possession supports secure boot, run the following command:

```
# sudo mst start
# sudo flint -d /dev/mst/mt41686_pciconf0 q full | grep "Life
cycle"
Life cycle:            GA SECURED
```

“GA SECURED” indicates that the BlueField device has secure boot enabled.

To verify whether the BlueField Arm has secure boot enabled, run the following command from the BlueField console:

```
ubuntu@localhost:~$ sudo mlxbf-bootctl | grep lifecycle
lifecycle state: GA Secured
```

UEFI Secure Boot

> **Note**
>
> This feature is available in the NVIDIA® BlueField®-2 and above.

UEFI Secure Boot is a feature of the Unified Extensible Firmware Interface (UEFI) specification. The feature defines a new interface between the operating system and firmware/BIOS.

When enabled and fully configured on the DPU, UEFI Secure Boot helps the Arm=based software running on top of UEFI resist attacks and infection from malware. UEFI Secure Boot detects tampering with boot loaders, key operating system files, and unauthorized option ROMs by validating their digital signatures. Malicious actions are blocked from running before they can attack or infect the system.

UEFI Secure Boot works as a security gate. Code signed with valid keys (whose public key/certificates exist in the DPU) gets through the gate and executes while blocking and rejecting code that has either a bad or no signature.

The DPU enables UEFI secure boot with the Ubuntu OS included in the platform's software.

Verifying UEFI Secure Boot on DPU

To verify whether UEFI secure boot is enabled, run the following command from the BlueField console:

```
ubuntu@localhost:~$ sudo mokutil --sb-state
SecureBoot enabled
```

As UEFI secure boot is not specific to BlueField platforms, please refer to the Canonical documentation online for further information on UEFI secure boot:

- https://wiki.ubuntu.com/UEFI/SecureBoot
- https://wiki.ubuntu.com/UEFI/SecureBoot/Signing

Main Use Cases for UEFI Secure Boot

UEFI secure boot can be used in 2 main cases for the DPU:

Method	Pros	Cons
Using the default enabled UEFI secure boot (with Ubuntu OS or any Microsoft-signed boot loader)	Relatively easy	Limited flexibility; only allows executing NVIDIA binary files
		Dependency on Microsoft or NVIDIA as signing entities

Method	Pros	Cons
See "Using Default Enabled UEFI Secure Boot" for more.	Autonomy, as you control your own keys (no dependency on Microsoft or NVIDIA as signing entities)	You must create your own capsule files to enroll and customize UEFI secure boot
Enabling UEFI Secure Boot with a custom OS (other than the default Ubuntu) See "Enabling UEFI Secure Boot with Custom OS" for more.		

Signing binaries is complex as you must create X.509 certificates and enroll them in UEFI or shim which requires a fair amount of prior knowledge of how secure boot works. For that reason, BlueField secured platforms are shipped with all the needed certificates and signed binaries (which allows working seamlessly with the first use case in the table above).

NVIDIA strongly recommends utilizing UEFI secure boot in any case due the increased security it enables.

Verifying UEFI Secure Boot on DPU

To verify whether UEFI secure boot is enabled, run the following command from the BlueField console:

```
ubuntu@localhost:~$ sudo mokutil --sb-state
SecureBoot enabled
```

As UEFI secure boot is not specific to BlueField platforms, refer to the Canonical documentation online for further information on UEFI secure boot to familiarize yourself with the UEFI secure boot concept:

- https://wiki.ubuntu.com/UEFI/SecureBoot
- https://wiki.ubuntu.com/UEFI/SecureBoot/Signing

Using Default Enabled UEFI Secure Boot

As part of the default settings of the DPU, UEFI secure boot is enabled and requires no special configuration from the user to use it with the bundled Ubuntu OS.

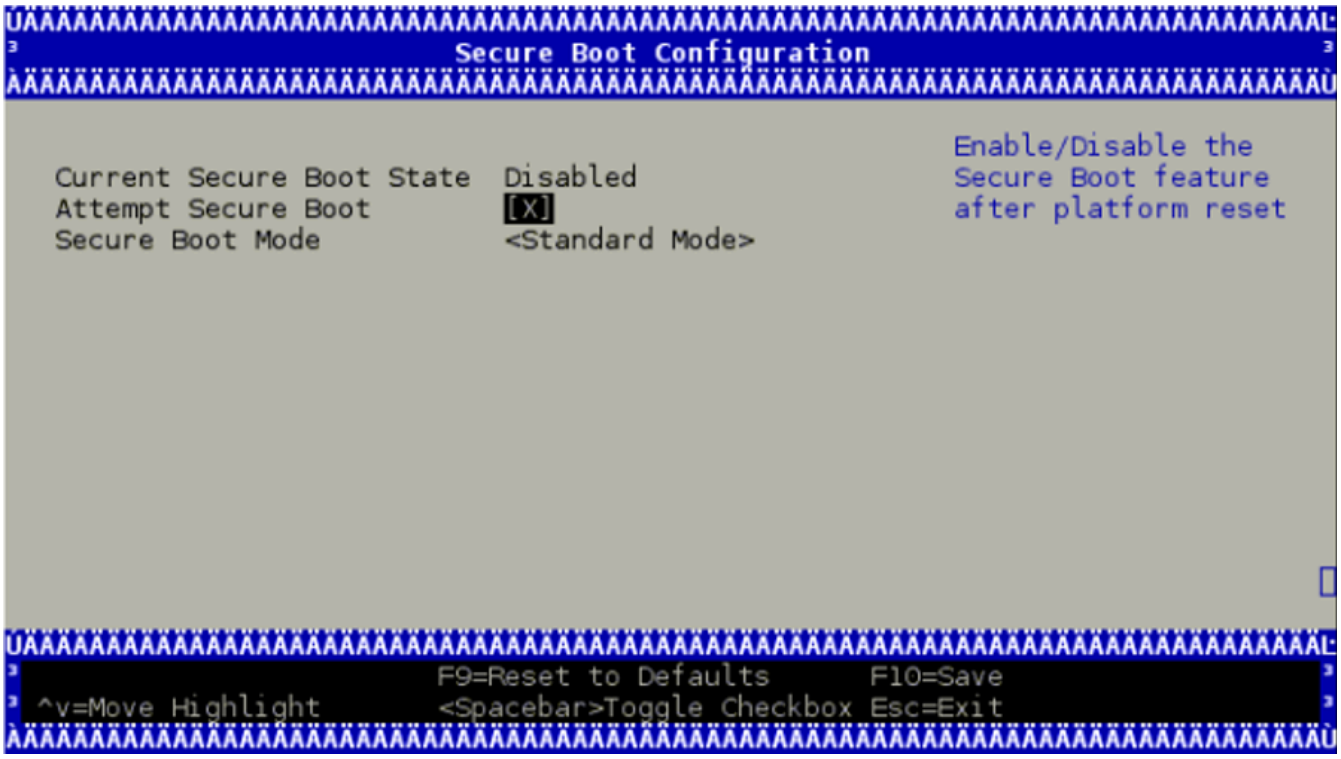
Disabling UEFI Secure Boot

UEFI secure boot can be disabled per device from the UEFI menu as part of the DPU boot process which requires access to the BlueField console.

To disable UEFI secure boot, reboot the platform and stop at the UEFI menu.

> **Note**
>
> On BlueField devices with UEFI secure boot enabled, the UEFI menu is password-protected to prevent unwanted changes to the UEFI settings. The default password is `bluefield` .

From the UEFI menu screen, select "Device Manager" then "Secure Boot Configuration". If "Attempt Secure Boot" is checked, then uncheck it and reboot.



> **Warning**

Disabling secure boot permanently is not recommended in production environments.

Info

It is also possible to disable UEFI secure boot using Redfish API for DPUs with an on-board BMC. For more details, please refer to your NVIDIA sales representative to receive the *NVIDIA BlueField DPU Initial Deployment Guide*.

Existing DPU Certificates

As part of having UEFI secure boot enabled, the UEFI databases are populated with NVIDIA self-signed X.509 certificates. The Microsoft certificate is also installed into the UEFI database to ensure that the Ubuntu distribution can boot while UEFI secure boot is enabled (and generally any suitable OS loader signed by Microsoft).

The pre-installed certificate files are:

- NVIDIA PK key certificate
- NVIDIA KEK key certificate
- NVIDIA db certificate
- Microsoft db certificate

Enabling UEFI Secure Boot with Custom OS

This section lists the required steps to enable using UEFI secure boot with a custom OS (other than the default Ubuntu).

Note

All processes described in the following subsections require some level of testing and knowledge in how operating system boot flows and bootloaders work.

Options for Enabling UEFI Secure Boot

There are 3 main ways for signing custom binaries and running them on the DPU with UEFI secure boot enabled:

#	Method	Pros	Cons
1	Sign OS loader (e.g., Shim) by Microsoft. See "Signing OS Loader by Microsoft" for more.	Does not require access to the BlueField console	Dependency on Microsoft as signing entity
2	Shim – enroll a machine owner key (MOK) certificate in the shim and use the private part to sign your files. See "Enrolling MOK Key" for more.	Easy	• Limited flexibility: Only allows executing a custom kernel or load a custom module. It does not allow executing UEFI applications, UEFI drivers, or OS loaders. • Dependency on Microsoft or NVIDIA as signing entities • Not scalable: Requires access to BlueField console per device (i.e., UART console required)
3	UEFI – enroll your own key certificate in the UEFI database and use the private part to sign your files. See "Enrolling Your Own Key to UEFI DB" for more.	Autonomy, as you control your keys (not dependent on Microsoft or NVIDIA as signing entities)	• Requires adding your key certificate to database manually • Requires access to BlueField console per device (i.e., UART console required)

#	Method	Pros	Cons
			• Not scalable: Requires access to BlueField console per device (i.e., UART console required)

For generation of custom keys and certificates, see section "Generation of Custom Keys and Certificates".

Signing binaries for UEFI secure boot is complex as you must create X.509 certificates and enroll them in UEFI or shim which requires a fair amount of prior knowledge of how secure boot works. See the processes used to enroll keys and to sign UEFI binaries in the rest of this document.

Secure booting binaries for executing a UEFI application, UEFI driver, OS loader, custom kernel, or loading a custom module depends on the certificates and public keys available in the UEFI database and the shim's MOK list.

Signing OS Loader by Microsoft

Custom Kernel Images

One option to boot custom binaries on a DPU is to sign the OS loader (shim) by Microsoft following the Microsoft guidelines which are updated and maintained by Microsoft. The certificates/keys must be embedded within the shim OS loader so it may boot, in addition the custom Kernel binary image and the custom Kernel modules must be signed accordingly.

NVIDIA Kernel Modules

In this option, the NVIDIA db certificates should remain enrolled. This is due to the out-of-tree kernel modules and drivers (e.g., OFED) provided by NVIDIA which are signed by NVIDIA and authenticated by this NVIDIA certificate in the UEFI.

Note

Signing binaries with Microsoft is a process the involves lead time which must be taken into consideration. This course of action requires

testing to making sure the complied BFB image including the signed Microsoft bootloader works properly.

Enrolling MOK Key

To boot a custom kernel or load a custom module, you must create a MOK key pair. The newly created MOK key must be an RSA 2048-bit. The private part is used for signing operations and must be kept safe. The public X.509 key certificate in DER format must be enrolled within the shim MOK list.

Once the public key certificate is enrolled within the shim, the MOK key is accepted as a valid signing key.

Note that kernel module signing requires a special configuration. For example, the `extendedKeyUsage` field must show an OID of 1.3.6.1.4.1.2312.16.1.2. That OID informs shim that this is meant to be a module signing certificate.

The following is an example of OpenSSL configuration file for illustration purposes:

```
HOME                    = .
RANDFILE                = $ENV::HOME/.rnd
[ req ]
distinguished_name      = req_distinguished_name
x509_extensions         = v3
string_mask             = utf8only
prompt                  = no

[ req_distinguished_name ]
countryName             = US
stateOrProvinceName     = Westborough
localityName            = Massachusetts
0.organizationName      = CampanyX
commonName              = Secure Boot Signing
emailAddress            = example@example.com
```

```
[ v3 ]
subjectKeyIdentifier    = hash
authorityKeyIdentifier  = keyid:always,issuer
basicConstraints        = critical,CA:FALSE
extendedKeyUsage        =
codeSigning,1.3.6.1.4.1.311.10.3.6,1.3.6.1.4.1.2312.16.1.2
nsComment               = "OpenSSL Generated Certificate"
```

To enroll the MOK key certificate, download the associated key certificate to the BlueField file system and run the following command:

```
ubuntu@localhost:~$ sudo mokutil --import mok.der
input password:
input password again:
```

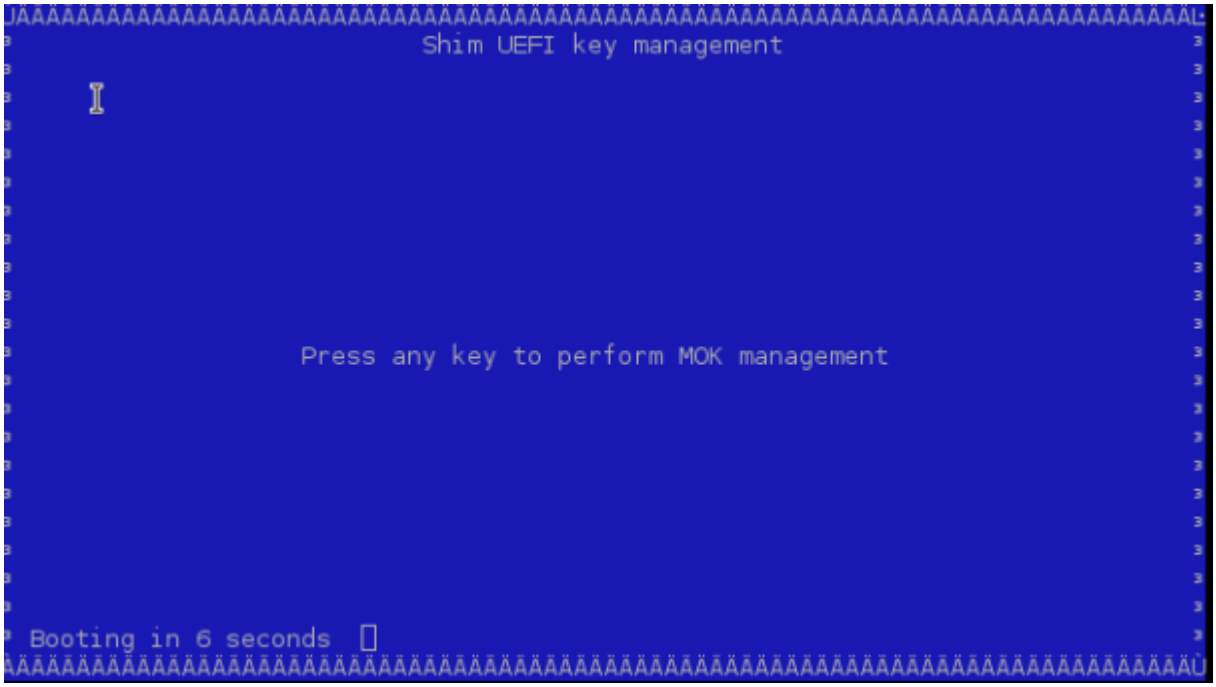
You must follow the prompts to enter a password to be used to make sure you really do want to enroll the key certificate.

Note that the key certificate is not enrolled yet. It will be enrolled by the shim upon the next reboot. To list the imported certificate file to enroll:

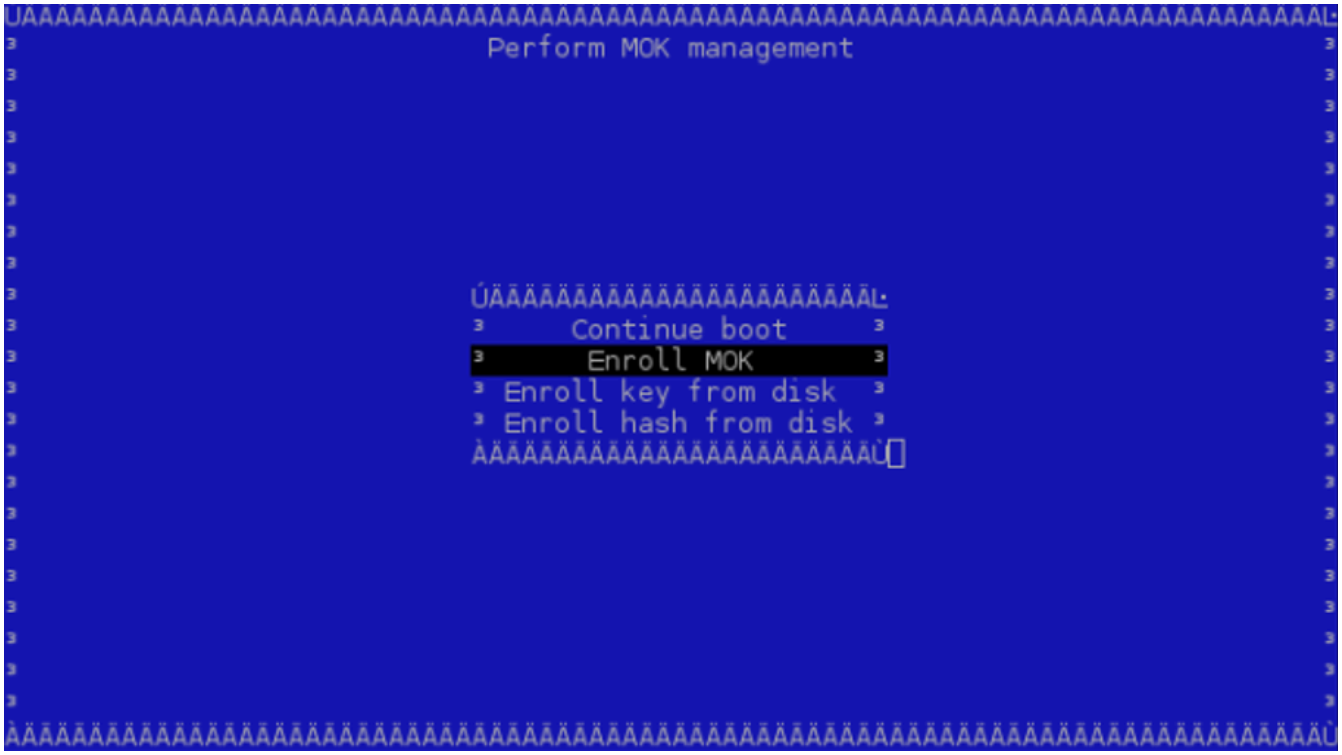
```
ubuntu@localhost:~$ sudo mokutil --list-new
```

A reboot must be performed.

Just before loading GRUB, shim displays a blue screen which is actually another piece of the shim project called "MokManager". You may ignore the blue screen showing the error message. Press "OK" to enter the "Shim UEFI key management" screen.



Select "Enroll MOK" and follow the menus to finish the enrolling process.



You may look at the properties of the key you are adding to make sure it is indeed correct using "View key". MokManager will ask for the same password you typed in earlier when running mokutil before reboot. MokManager will save the key and you will need to reboot again.

To list the enrolled certificate files, run the following command:

```
ubuntu@localhost:~$ sudo mokutil --list-enrolled
```

Generation of Custom Keys and Certificates

To boot binaries not signed with the existing public keys and certificates in the UEFI database (like the Microsoft certificate and key described in "Signing OS Loader by Microsoft"), create an X.509 certificate (which includes the public key part of the public–private key pair) that can be imported either directly though the UEFI or, more easily, via shim.

Creating a certificate and public key for use in the UEFI secure boot is relatively simple. OpenSSL can do it by running the command `req`.

For illustration purposes only, this example shows how to create a 2048-bit RSA MOK key and its associated certificate file in DER format:

```
$ openssl req -new -x509 -newkey rsa:2048 -nodes -days 36500 -
outform DER -keyout "mok.priv" -out "mok.der"
```

An OpenSSL configuration file may be used for key generation. It may be specified using `--config path/to/openssl.cnf`.

> **Note**
>
> Detailed key and certificate generation are beyond the scope of this document. Any organization should choose the proper way to generate keys and certificates based on their security policy.

The following sections refer to the db private key as `key.priv` and its DER certificate as `cert.der`. Similarly, the MOK private key is referred to as `mok.priv` and its DER

certificate as `mok.der`.

Enrolling Your Own Key to UEFI DB

Some users may need to generate their own keys. For convenience, the processes used to enroll keys into UEFI db as well as to sign UEFI binaries are provided in this document.

To execute your binaries while UEFI secure boot is enabled, you need your own pair of private and public key certificates. The supported keys are RSA 2048-bit and ECDSA 384-bit.

The private part is used for signing operations and must be kept safe. The public part X.509 key certificate in DER format must be enrolled within the UEFI db.

A prerequisite for the following steps is having UEFI secure boot temporarily disabled on the DPU. After temporarily disabling UEFI secure boot per device as in section "Existing DPU Certificates", it is possible to override all the key certificate files of the UEFI database. This allows you to enroll your PK key certificate, KEK key certificate, and db certificates.

The following subsections detail how enrolling can be done.

Using a Capsule

To enroll your key certificates, create a capsule file by way of tools and scripts provided along with the BlueField software.

To create the capsule files, execute the `mlx-mkcap` script. After BlueField software installation, the script can be found under `/lib/firmware/mellanox/boot/capsule/scripts`. This script generates a capsule file to supply the key certificates to UEFI and enables UEFI secure boot:

```
$ ./mlx-mkcap --pk-key pk.cer --kek-key kek.cer --db-key db.cer
EnrollYourKeysCap
```

Note that you may specify as many db certificates as needed using the `--db-key` flag. In this example, only a single db certificate is specified.

To set the UEFI password, you may specify the `--uefi-passwd` flag. For example, to set the UEFI password to `bluefield`, run:

```
$ ./mlx-mkcap --pk-key pk.cer --kek-key kek.cer --db-key db.cer --uefi-passwd "bluefield" EnrollYourKeysCap
```

The resulting capsule file, `EnrollYourKeysCap`, can be downloaded to the BlueField file system to initiate the key enrollment process. From the the BlueField console execute the following command then reboot:

```
ubuntu@localhost:~$ bfrec --capsule EnrollYourKeysCap
```

On the next reboot, the capsule file is processed and the UEFI database is populated with the keys extracted from the capsule file.

> **Note**
>
> Enrolling the PK key certificate file enables the UEFI secure boot.

Enroll Certificate into UEFI DB

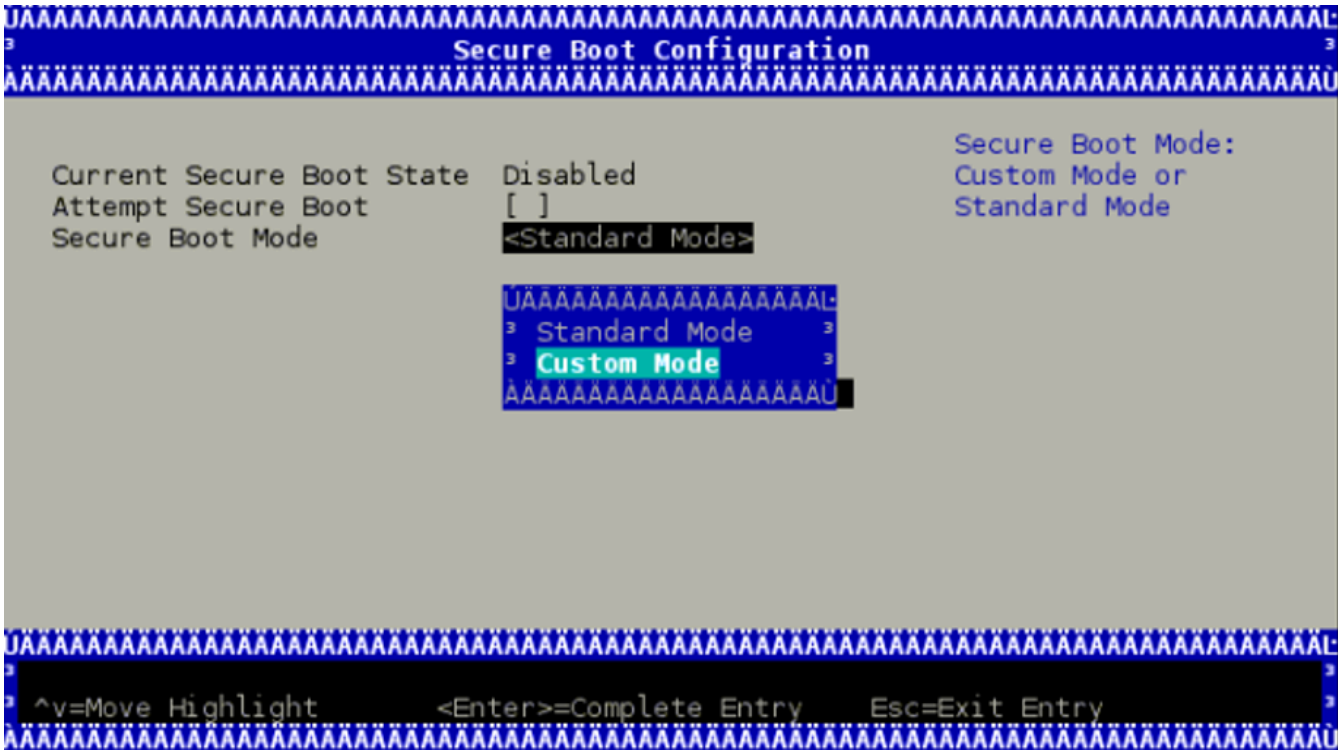
As mentioned, the public part of the X.509 key certificate in DER format must be enrolled within the UEFI db. The X.509 DER certificate file must be installed into the EFI system partition (ESP).

Download the certificate file to BlueField file system and place it into the ESP:

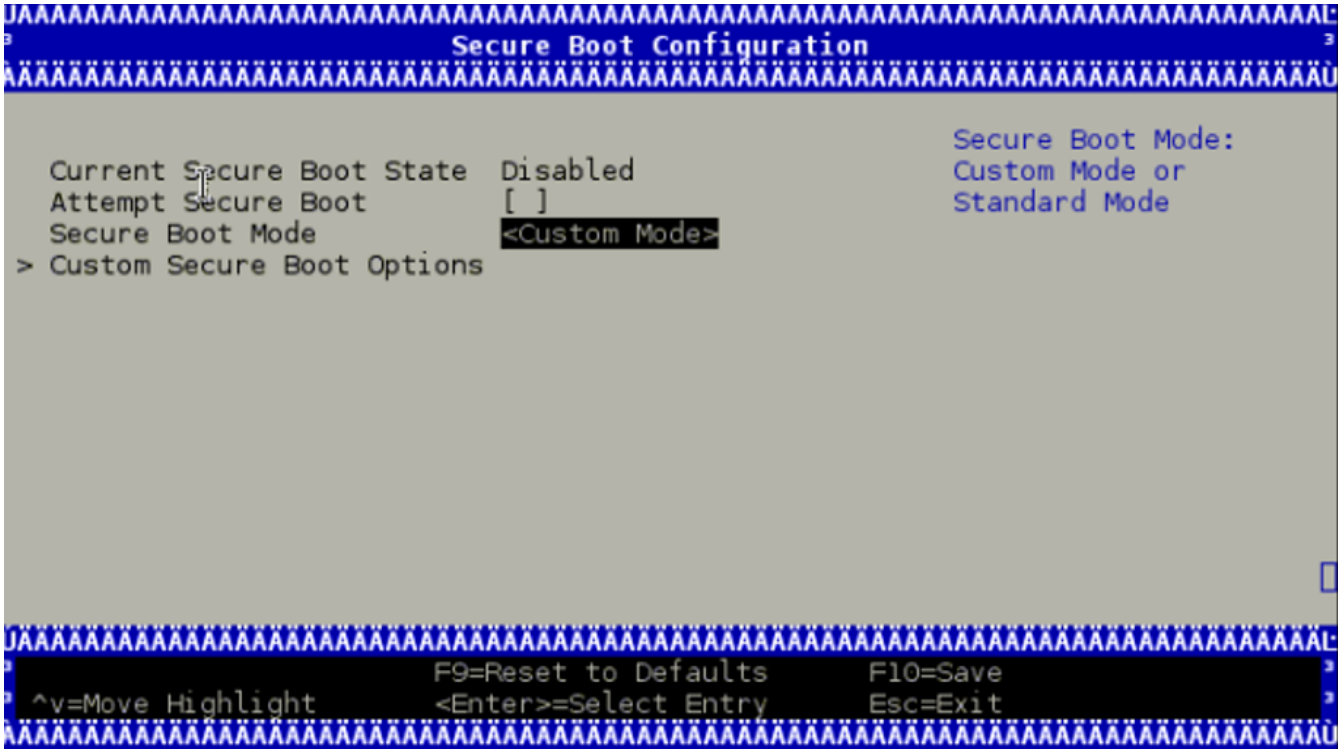
```
ubuntu@localhost:~$ sudo cp path/to/cert.der /boot/efi/
```

To enroll the certificate into the UEFI db, you must to reboot and log in again into the UEFI menu. From the "UEFI menu", select "Device Manager" entry then "Secure Boot Configuration". Navigate to "Secure Boot Mode" and select "Custom Mode" setup.

The secure boot "Custom Mode" setup feature allows a physically present user to modify the UEFI database.

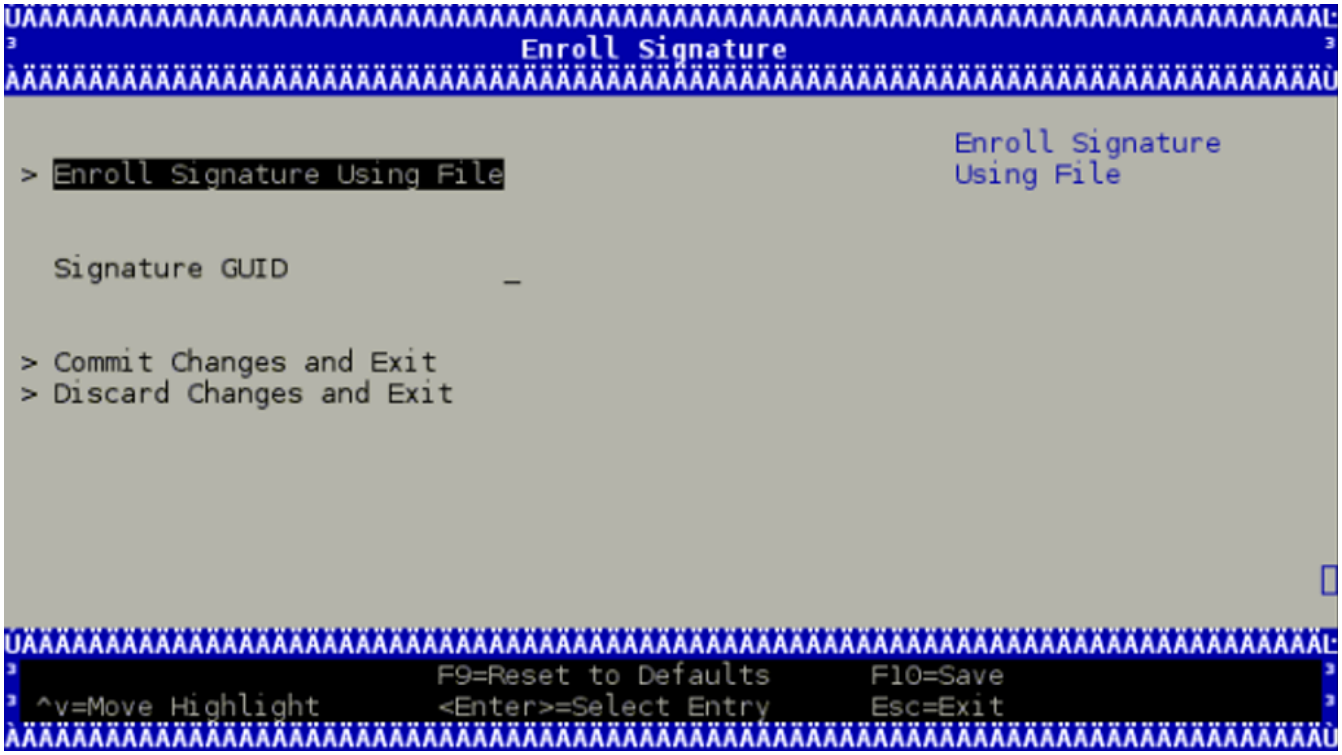


Once the platform is in "Custom Mode", a "Custom Secure Boot Options" menu entry appears which allows you to manipulate the UEFI database keys and certificates.



To enroll your DER certificate file, select "DB Options" and enter the "Enroll Signature" menu. Select "Enroll Signature Using File" and navigate within the EFI System Partition (ESP) to the db DER certificate file.

The ESP path is shown below as "system-boot, [VenHw(*)/HD(*)]".



While enrolling the certificate file, you may enter a GUID along with the key certificate file. The GUID is the platform's way of identifying the key. It serves no purpose other than for you to tell which key is which when you delete it (it is not used at all in signature verification).

This value must be in the following format: 11111111-2222-3333-4444-1234567890ab. If nothing is entered, a GUID of 00000000-0000-0000-0000-000000000000 is created.

Finally, commit the changes and exit. You may be asked to reboot.

Signing Binaries

Signing Custom Kernel and UEFI Binaries

To sign a custom kernel or any other EFI binary (UEFI application, UEFI driver or OS loader) you want to have loaded by shim, you need the private part of the key and the certificate in PEM format.

To convert the certificate into PEM, run:

```
$ openssl x509 -in mok.der -inform DER -outform PEM -out mok.pem
```

Now, to sign your EFI binary, run:

```
$ sbsign --key mok.priv --cert mok.pem binary.efi --output
binary.efi.signed
```

If you are using your db key, use the private part of the key and its associated certificate converted into PEM format for binary signing.

If the X.509 key certificate is enrolled in UEFI db or by way of shim, the binary should be loaded without an issue.

Signing Kernel Modules

The X.509 certificate you added must be visible to the kernel. To verify the keys visible to the kernel, run:

```
ubuntu@localhost:~$ sudo cat /proc/keys
```

For a straightforward result, run:

```
ubuntu@localhost:~$ dmesg | grep -i "X.509"
[    1.869521] Loading compiled-in X.509 certificates
[    1.875441] Loaded X.509 cert 'Build time autogenerated kernel
key: b1a3fbd0178bdb7190387a4187e8e4b0eb476cdc'
[    1.941752] integrity: Loading X.509 certificate: UEFI:db
[    1.947636] integrity: Loaded X.509 cert 'YourSigningDbKey:
a109f01707ba6769c4d546530ba1592c7daedc3b'
[    1.958736] integrity: Loading X.509 certificate: UEFI:db
```

```
[    1.964170] integrity: Loaded X.509 cert 'Microsoft
Corporation UEFI CA 2011:
13adbf4309bd82709c8cd54f316ed522988a1bd4'
[    2.023740] integrity: Loading X.509 certificate:
UEFI:MokListRT
[    2.030090] integrity: Loaded X.509 cert 'YourSingingMokKey:
2012e5122669ffc0cc28827c6134329a6bec0b88'
[    2.040796] integrity: Loading X.509 certificate:
UEFI:MokListRT
[    2.046830] integrity: Loaded X.509 cert 'SomeOrg: shim:
331c1c8963538e327d6e39346f4f53b200987015'
[    2.055796] integrity: Loading X.509 certificate:
UEFI:MokListRT
[    2.062114] integrity: Loaded X.509 cert 'Canonical Ltd.
Master Certificate Authority:
ad91990bc22ab1f517048c23b6655a268e345a63'
```

If the X.509 certificate attributes (`commonName`, etc.) are configured properly, you should see your key certificate information in the result output. In this example, two custom keys are visible to the kernel:

- `YourSigningMokKey` – registered with the shim as a MOK
- `YourSigningDbKey` – registered with UEFI as db

> **Note**
>
> This example is for illustration purposes only. The actual output might differ from the output shown in this example depending on what key was previously enrolled and how it was enrolled.

You may sign kernel modules using either of these approaches:

- `kmodsign` command

- Linux kernel script sign-file

Signing Kernel Modules Using kmodsign

If you are using the `kmodsign` command to sign kernel modules, run:

```
ubuntu@localhost:~$ sudo cat /proc/keys
```

The signature is appended to the kernel module by `kmodsign`.

But if you rather keep the original kernel module unchanged, run:

```
ubuntu@localhost:~$ kmodsign sha512 mok.priv mok.der module.ko
module-signed.ko
```

Refer to `kmosign --help` for more information.

Signing Kernel Modules Using Sign File

To sign the kernel module using the Linux kernel script sign-file, please refer to Linux kernel documentation.

If you are using your db key, use the private part of the key and its associated certificate for binary signing.

To validate that the module is signed, check that it includes the string `~Module signature appended~`:

```
ubuntu@localhost:~$ hexdump -Cv module.ko | tail -n 5
00002c20  10 14 08 cd eb 67 a8 3d  ac 82 e1 1d 46 b5 5c 91
|.....g.=....F.\.|
00002c30  9c cb 47 f7 c9 77 00 00  02 00 00 00 00 00 00 00
|..G..w..........|
```

```
00002c40  02 9e 7e 4d 6f 64 75 6c  65 20 73 69 67 6e 61 74
|..~Module signat|
00002c50  75 72 65 20 61 70 70 65  6e 64 65 64 7e 0a        |ure
appended~.|
00002c5e
```

Ongoing Updates

Update Key Certificates

> **Note**
>
> This requires UEFI secure boot to have been enabled using your own keys, which means that you own the signing keys.

While UEFI secure boot is enabled, it is possible to update your keys using a capsule file.

To create a capsule intended to update the UEFI secure boot keys, generate a new set of keys and then run:

```
$ ./mlx-mkcap --pk-key new_pk.cer --kek-key new_kek.cer --db-key
new_db1.cer --db-key new_db2.cer --db-key new_db3.cer --signer-
key db.key --signer-cert db.pem EnrollYourNewKeysCap
```

Note that `--signer-key` and `--signer-cert` are set so the capsule is signed. When UEFI secure boot is enabled, the capsule is verified using the key certificates previously enrolled in the UEFI database. It is important to use the old signing keys associated with the certificates in the UEFI database to sign the capsule. The new key certificates are intended to replace the existing key certificates after capsule processing. Once the UEFI database is updated, the new keys must be used to sign the newly created capsule files.

To enroll the new set of keys, download the capsule file to the BlueField console and use `bfrec` to initiate the capsule update.

Disable UEFI Secure Boot Using a Capsule

> **Note**
>
> This requires UEFI secure boot to have been enabled using your own keys, which means that you own the signing keys.

It is possible to disable UEFI secure boot through a capsule update. This requires an empty PK key when creating the capsule file.

To create a capsule intended to disable UEFI secure boot:

1. Create a dummy empty PK certificate:

```
$ touch null_pk.cer
```

2. Create the capsule file:

```
$ ./mlx-mkcap --pk-key null_pk.cer --signer-key db.key --
signer-cert db.pem DeletePkCap
```

`--signer-key` and `--signer-cert` must be specified with the appropriate private keys and certificates associated with the actual key certificates in the UEFI database.

To enroll the empty PK certificate, download the capsule file to the BlueField console and use `bfrec` to initiate the capsule update.

> ⚠ **Warning**
>
> Deleting the PK certificate will result in UEFI secure boot to be disabled which is not recommended in a production environment.

Updating Platform Firmware

To update the platform firmware on secured devices, download the latest NVIDIA® BlueField® software images from NVIDIA.com.

Updating eMMC Boot Partitions Image

The capsule file `/lib/firmware/mellanox/boot/capsule/MmcBootCap` is used to update the eMMC boot partition and update the Arm pre-boot code (i.e., Arm trusted firmware and UEFI).

The capsule file is signed with NVIDIA keys. If UEFI secure boot is enabled, make sure the NVIDIA certificate files are enrolled into the UEFI database. Please refer to "UEFI Secure Boot" for more information on how to update the UEFI database key certificates.

To initiate the update of the eMMC boot partitions, run the following command:

```
ubuntu@localhost:~$ sudo bfrec --capsule
/lib/firmware/mellanox/boot/capsule/MmcBootCap
```

After the command completes, reboot the system to process the capsule file. On the next reboot, UEFI will verify the capsule signature. If verified, UEFI will process the capsule file, extract the pre-boot image and burn it into the eMMC boot partitions.

Note that the pre-boot code is signed with the NVIDIA key. The bootloader images are installed into the eMMC with their associated certificate files. The public key is derived from the certificate file and its integrity is verified by the ROM code against an on-chip

public key hash value stored in E-FUSEs. If the verification fails, then the pre-boot code will not be allowed to execute.

Recovering eMMC Boot Partition

If the system cannot boot from the eMMC boot partitions for any reason, it is recommended to download a valid BFB image and boot it over the BlueField platform.

The recovery path relies on the platform to be configured to boot solely from the RShim interface (either RShim USB or RShim PCIe). With this configuration there must not be a way to interrupt or bypass the RoT when secure booting.

You will need to append a capsule file to the BFB prior to booting. Run:

```
$ mlx-mkbfb --capsule MmcBootCap install.bfb recovery_install.bfb
```

Then boot the `recovery_install.bfb` using the RShim interface. Run:

```
$ cat recovery_install.bfb > /dev/rshim0/boot
```

The capsule file will be processed by UEFI upon boot.

Updating SPI Flash FS4 Image

The SPI flash contains the firmware image of the DPU firmware in FS4 format. The firmware image is provided along with the software.

There are two different ways to install the firmware image:

- From the BlueField console, using the following command:

  ```
  ubuntu@localhost:~$ /opt/mellanox/mlnx-fw-
  ```

```
updater/firmware/mlxfwmanager_sriov_dis_aarch64_<bf-dev>
```

- From the PCIe host console, using the following command:

```
# flint -d /dev/mst/mt<bf-dev>_pciconf0 -i firmware.bin b
```

Info

`bf-dev` is 41686 for BlueField-2 or 41692 for BlueField-3.

may be based on or attributable to: (i) the use of the NVIDIA product in any manner that is contrary to this document or (ii) customer product designs.

No license, either expressed or implied, is granted under any NVIDIA patent right, copyright, or other NVIDIA intellectual property right under this document. Information published by NVIDIA regarding third-party products or services does not constitute a license from NVIDIA to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property rights of the third party, or a license from NVIDIA under the patents or other intellectual property rights of NVIDIA.

Reproduction of information in this document is permissible only if approved in advance by NVIDIA in writing, reproduced without alteration and in full compliance with all applicable export laws and regulations, and accompanied by all associated conditions, limitations, and notices.

THIS DOCUMENT AND ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE. TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL NVIDIA BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF NVIDIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Notwithstanding any damages that customer might incur for any reason whatsoever, NVIDIA's aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms of Sale for the product.

Trademarks

NVIDIA and the NVIDIA logo are trademarks and/or registered trademarks of NVIDIA Corporation in the U.S. and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.