LVFS

Release 1.5.2-31-gaaac860

Richard Hughes

May 16, 2024

CONTENTS:


LVFS
Linux Vendor
Firmware Service

CHAPTER

ONE

INTRODUCTION

Updating firmware on devices is traditionally difficult for users on Linux systems. Not knowing exact hardware details, where to look for updates or how to run the Windows-specific flashing tools makes it almost impossible to update firmware on devices.

As a result, *"broken"* hardware is being returned to the vendor and customer systems are left in an insecure state even when updates have been released that fix the specific issues. Linux as the OS is now mainstream and vendors need to support these customers.

The LVFS is a secure web service that can be used by OEM's to upload firmware archives and can also be used by users to securely download metadata about available updates and optionally, the updates themselves.

Millions of customer devices are being updated every month thanks to the LVFS!

1.1 The Problem

Linux users have traditionally had problems with keeping hardware up to date with firmware updates. There are three components to this problem:

- They do not know what exact hardware they have installed, the current firmware version, or even if the devices support being upgraded at all.
- They do not know where to look for updates; often searching the various vendor websites is an exercise in frustration and as a result most users do not bother.
- Windows-specific flashing tools do not work on Linux; a significant number of Linux users keep a Windows virtual machine for essential business-critical software that is not available on Linux. This will not work for firmware update utilities that require low level hardware access.

The fwupd project can query supported hardware for the current firmware versions and also deploy new firmware versions to devices, but requires metadata from the LVFS to know the details about available updates. It also requires vendors to prepare the firmware with the required metadata and to use a standardized deployment framework e.g. DFU or UEFI UpdateCapsule.

Using the information from from higher level software centers can show the user the update description in their own language and offer the update to be installed using just three clicks of the mouse. Security updates are handled in the same way as other OS updates meaning it is just one mechanism for the user to understand.

The LVFS supplies the data in a secure format, allowing the fwupd project to install the update safely. Existing approaches have been OEM specific which meant that a large amount of engineering effort was required, making this approach only financially viable for enterprise use-cases.

There are a significant number of legal problems with the redistribution of firmware, and we have been working with vendors finding acceptable methods of redistribution whilst ensuring confidentially throughout the process. Being

backed by a large Linux vendor with heterogeneous support for many vendors and platforms puts the LVFS in exactly the right place to build this kind of shared infrastructure.

1.2 System Architecture

The architecture is built into three layers: a presentation layer, a mechanism and a data-provider and each can be replaced as required as they all use standard protocols.

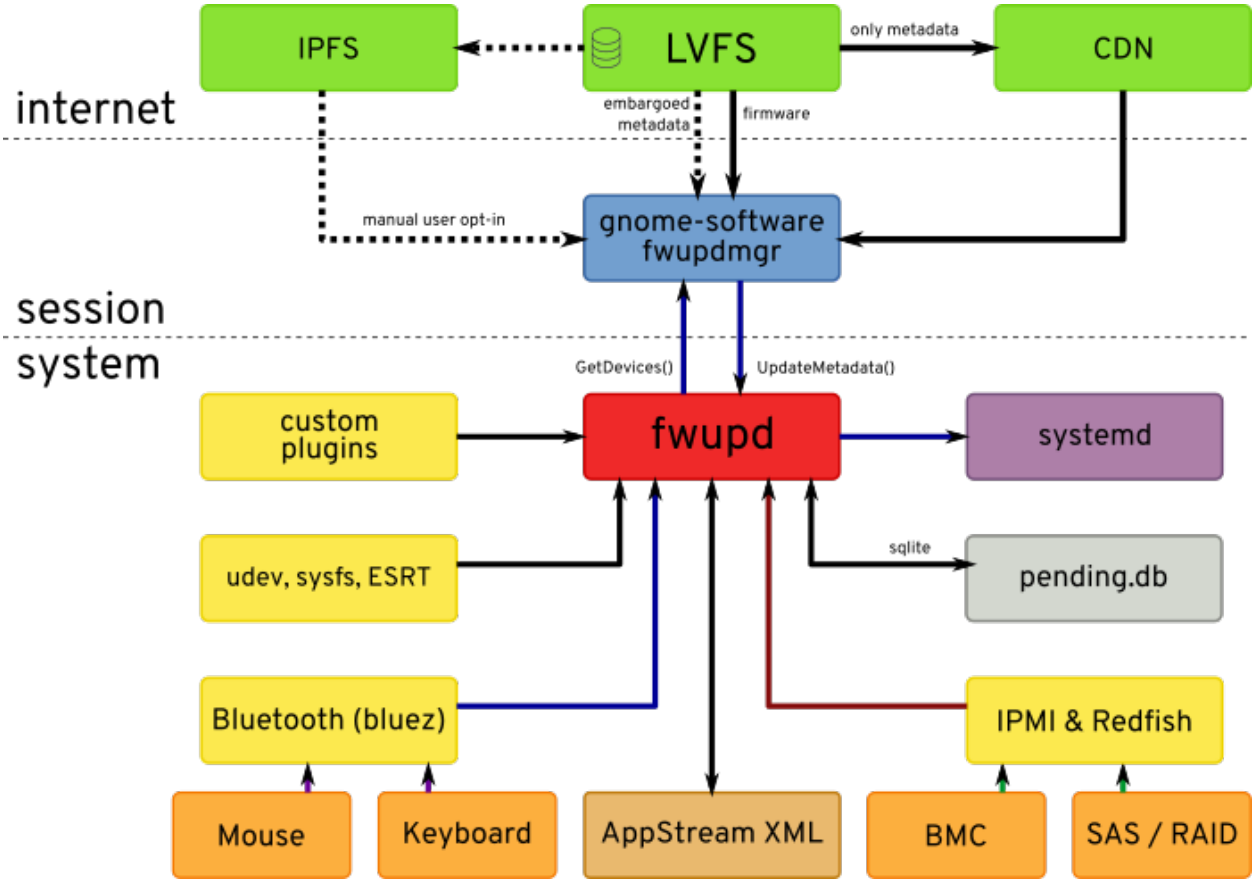


Fig. 1: Architecture plan, showing each subsystem

1.3 GNOME Software

GNOME Software is an application store designed to make installing, removing and updating both easy and beautiful. It is available for Linux and used by millions of people on the following distributions:

- RHEL and CentOS 7.4 or newer
- Fedora 22 or newer
- Ubuntu 16.04 (Xenial) or newer
- Debian 9 (Stretch) or newer
- openSUSE 15.0 or newer

- Arch from 2017-06-13

For most desktop systems, at session start-up the metadata XML and detached signatures are checked for a specified age, and if required newer files are automatically downloaded from the LVFS and pushed into fwupd over D-Bus. When the update list is required we query the fwupd daemon over D-Bus for any pending updates. If there are updates that need applying then they are downloaded and the user is notified and the update details are shown in the specified language. The user has to explicitly agree to the firmware update action before the update is performed.

Fig. 2: GNOME Software

1.4 fwupd

This project provides a system-activated daemon `fwupd` with a D-Bus interface that can be used by unprivileged clients. Clients can perform system wide upgrades and downgrades according to a security policy, which uses PolicyKit to negotiate for authorization if required. The command line tool `fwupdmgr` can be used to administer headless clients on the command line over SSH or using a management framework like Red Hat Satellite or Dell CCM.

The daemon parses metadata in AppStream format from the LVFS along with a detached GPG or PKCS#7 signature. The `.cab` archives which must contain at least a `.metainfo.xml` file and a detached GPG/PKCS#7 signature of the firmware payload. Other files are permitted in the archive which allows the same deliverable to be used for the Windows Update system.

Internally fwupd creates a device with a unique ID, and then a number of GUIDs are assigned to the device by the plugin. It is these GUIDs specified in the update metadata file that are used to match a firmware file to a device. Although it is usually the responsibility of the system vendor to generate a new GUID if the hardware requires a different firmware file, we can match an update that only applies to specific versions of hardware using CHID GUIDs.

Adding more plugins to fwupd is of course possible, but where possible vendors should use the existing code and for instance add an ESRT data table when building the system firmware.

1.4.1 Offline Updates

When the user agrees to a UEFI firmware update the firmware is unpacked into the EFI System Partition, several UEFI keys are set and the system reboots. On reboot the `fwupd.efi` binary is run before the bootloader is started and the firmware `UpdateCapsule` UEFI runtime source is called.

For most devices (e.g. USB, Thunderbolt, Synaptics, etc.) the update is performed without requiring a reboot.

1.5 LVFS

The LVFS provides an OEM-facing website that requires a username and password to access the secure console. There is no charge to vendors for the hosting or distribution of content, although there are some terms of service to vendors distributing content.

This service should only be used to distribute firmware that is flashed onto non-volatile memory. It is not designed for firmware that has to be uploaded to devices every time the device is used.

When `.cab` firmware files are submitted the following actions are performed:

1. The update metadata in the archive is checked.
2. The firmware capsule is signed with our GPG key or PKCS#7 certificate. Clients **do not** verify the signatures in the catalog file as this is for Windows Update only
3. The new cab file is repacked. Only required files are included in the cabinet file, typically making the download size much smaller
4. The metadata is added to our database.

Many ODMs are distinct and decoupled from the OEM, and in most cases the ODM is allowed to upload new firmware but not make it available for users. For this use case, users on the LVFS can have different attributes, for example:

- Unprivileged users that can upload files to the testing target
- Read only access to all analytics data for a specific vendor
- Quality assurance users that can modify all firmware uploaded to a specific vendor
- Trusted users that can move files to the testing or stable target, and can move files from testing to stable
- Manager users that can add new users to an existing vendor

1.6 Conclusions

The LVFS has grown to be an essential part of the Linux ecosystem used by over one hundred vendors, 15 of which are multi-billion dollar companies. The LVFS is a mature service providing important functionality for Linux users.

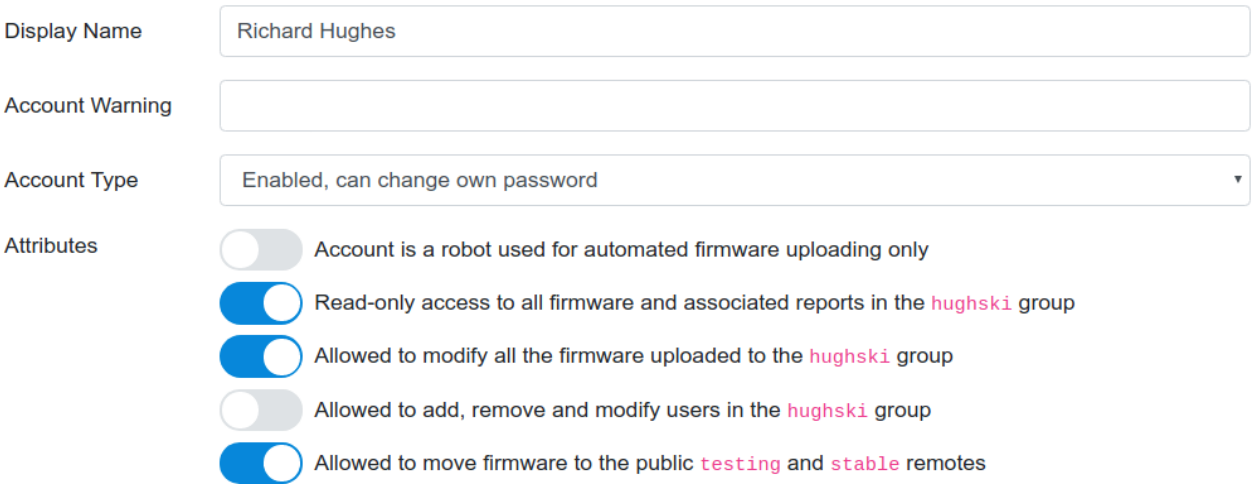


Fig. 3: Admin controlling the user permissions.

1.6.1 Future Work

Various vendors are working on custom plugins for fwupd as they either cannot retrofit older hardware with the ESRT data table, or because they want more control over the low level flashing protocol. We certainly would encourage any new vendors wanting to use the LVFS and fwupd to use a well-known standard like DFU or UEFI UpdateCapsule with ESRT as it means there is no application code to write.

From a system administrators point of view, it will also soon be possible to get notified of updates and perform upgrades using the Cockpit framework as well as the usual client tools.

1.6.2 Related Projects

The Dell Repository Manager allows you to update the firmware on various models of Dell enterprise hardware. There are several software (e.g. the SSU and SBUU) and hardware elements specific to Dell (e.g., the LCC or USC) and most of the stack is proprietary.

Microsoft provides a service called Windows Update which takes driver updates from vendors, optionally performs some quality control on the update, signs the firmware and then hosts the firmware on a CDN. The entire stack is proprietary and for Microsoft Windows only.

CHAPTER

TWO

GETTING AN ACCOUNT

There is no charge to vendors for opening an account or for distribution of content. You can start the process by opening a ticket with as much information you have, or just with questions or for more details.

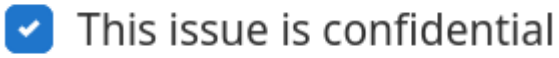
2.1 Information to Supply

- The vendor full legal name
- The public homepage for this vendor
- A link to a high resolution logo for the vendor
- The domain used for email address assigned to this vendor, e.g. `@realtek.com,@realtek.com.tw`
- The update protocol are you using, and if it is already supported in fwupd
- Legal permission that you have the required permission to upload to the LVFS. There is an example document which can be modified, signed, and uploaded as an attachment to the GitLab issue. We can create a vendor account *without this*, but the account will not be able to push firmware to the public remotes until this document is provided.
- The `Vendor ID` for all hardware uploaded by this vendor (from `fwupdmgr get-devices` e.g. `USB:0x046D`)
- The reverse DNS AppStream ID namespace prefix you plan to use for all uploaded firmware, e.g. `com.hp`
- The URL to use for any possible security incident response (PSIRT), e.g. `https://www.vendor.com/security`
- An assigned "vendor manager" that can create new accounts on the LVFS in the future, and be the primary point of contact
- If you going to be acting as an ODM or IHV to another vendor, e.g. uploading firmware on their behalf

If you are acting as an ODM or IHV to another vendor:

- Which OEM(s) will you be uploading for?
- Do you have a contact person for the OEM? If so, who?
- Will you be QAing the update and pushing to stable yourselves, or letting the OEM do this?

Note: If you wish for the ticket to remain private (only viewable by the LVFS administrators) you **must** mark it as confidential as otherwise the ticket is viewable by public users:



Note: Vendors who can upload firmware updates are in a privileged position where files can be installed on end-user systems without authentication. This means we have to do careful checks on new requests, which may take a few days to complete.

2.2 Vendor Groups

On the LVFS there are several classes of user that can be created. By default users are created as *upload only* which means they can only view firmware uploaded by themselves.

Users can be *promoted* to QA users by the vendor manager so that they can see (and optionally modify) other firmware in their vendor group. QA users are typically the people that push firmware to the testing and stable remotes.

There can be multiple vendor groups for large OEMs, for instance an OEM might want a *storage* vendor group that is isolated from the *BIOS* team. Alternatively, vendors can use Azure to manage users on the LVFS. Contact the LVFS administrator for more details if you would like to use this.

2.2.1 Adding Users

The vendor manager can add users to an existing vendor group. If the vendor manager has additional privileges (e.g. the permission to push to stable) then these can also be set for the new user.

New users have to match the username domain glob, so if the value for the vendor is `@realtek.com,@realtek.com.tw` then `dave@realtek.com.tw` could be added by the vendor manager – but `dave@gmail.com` would be forbidden.

2.2.2 Trusted Users

Vendor groups are created initially as `untrusted` which means no users can promote firmware to testing and stable.

Once a valid firmware has been uploaded correctly and been approved by someone in the the LVFS admin team we will *unlock* the user account to the `trusted` state which allows users to promote firmware to the public remotes.

Note: In most cases we also need some kind of legal document that shows us that the firmware is legally allowed to be redistributed by the LVFS.

For instance, something like this is usually required:

<vendor> is either the sole copyright owner of all uploaded firmware, or has permission from the relevant copyright owner(s) to upload files to Linux Vendor Firmware Service Project a Series of LF Projects, LLC (known as the "LVFS") for distribution to end users. <vendor> gives the LVFS explicit permission to redistribute the unmodified firmware binary however required without additional restrictions, and permits the LVFS service to analyze the firmware package for any purpose. <signature>, <date>, <title>

2.3 Export Control

Some firmware may contain binary code that has been deemed subject to some kind of export control. The exact meaning of *export control* has been defined in various places, including Export Administration Regulations (EAR), and International Traffic in Arms Regulations (ITAR).

Code capable of strong encryption like AES, RSA or 3DES may be subject to export control and it may be forbidden to distribute to users located in specific embargoed countries like Cuba, Iran, North Korea, Sudan or Syria.

Note: Although there is a specific and notable export exception for "software updates" in EAR, it should of course be the decision of the legal team of the OEM to make the decision themselves.

The list of countries is usually specified *per-vendor* which means it is applied for all firmware in the vendor account. It can also be specified *per-firmware*, which might be useful where just one specific firmware is explicitly covered under export control, for instance for a model only designed to be sold to the US military. This can be specified in the metadata block for the firmware `component`:

```
<custom>
  <value key="LVFS::BannedCountryCodes">IR,SY</value>
</custom>
```

Only LVFS admin team and vendor manager can edit the vendor export control list. It is specified according to ISO3166, which would typically be `CU,IR,KP,SD,SY` for most large vendors.

Note: Like all other services hosting files, the LVFS uses GeoIP data to identify which country the user is downloading files from. This is not a perfect science, and although the assigned list of IP blocks is updated daily some false positives and false negatives can occur.

2.4 End User License Agreements

Legal teams of vendors sometimes request that we make the end user agree to a license agreement or legal declaration before deploying the update. There are several reasons why have chosen to not support EULAs:

- The majority of updates applied in the enterprise are done "*unattended*" and also done at scale with thousands of devices. Forcing the end-user to do any interactive action makes these automated or "*headless*" updates impossible.
- Allowing other users to "*pre-accept*" the end-user license agreement isn't what this legal mechanism was designed for – for example is it legally binding if the junior sysadmin accepts the agreement on the end-users behalf? Or does it have to be accepted by someone from the destination legal team with the authority to do so – which needs to be recorded for audit purposes.
- Vendors often want to use a "*generic*" boilerplate legal agreement that controls how the user is allowed to use the hardware using overly broad language that is either not applicable to the device, totally confusing to the end user, or by adding restrictions on an already purchased product.
- Vendors often want to show a EULA so that if broken firmware gets deployed then it becomes the users fault for attempting the upgrade action and the vendor cannot be considered responsible in any way. **This isn't fair to customers** – risky or untested updates should never be pushed to millions of end users.

- LVFS is **used all over the world**, and users might not even understand the language the EULA is written in. Legal jurisdictions also differ between the nations of this world, and the EULA might not be legally binding or permissible.

We've been asked to add support for EULAs a few times and the answer has always been no. The almost-universal consensus from the community was that allowing EULAs is a terrible idea that would be a slippery slope, encouraging vendors to take the *easy option* and show pages of overly restrictive boilerplate legalese for each update.

If your legal department disagrees, please let them know that every vendor shipping firmware on the LVFS has agreed that a EULA was not actually required.

Note: The UI can show the release notes and an optional update message, but it is purely advisory and the user is free to ignore or suppress it – by disabling the condition in the source code or even patching the binary executable. The front-end client (e.g. GNOME Software or Google Chrome) also has no requirement to implement showing either. This UI was not designed for EULA text and should not be used in this way.

2.5 Alternate Branches

We typically only allow the silicon vendor, the ODM or the OEM to upload firmware for hardware, and only if that entity has legal permission to upload the file to the LVFS. The security model for fwupd relies on standardized registries like USB and PCI, along with immutable DMI information to ensure that only the correct vendors can ship firmware for their own hardware, and nothing else.

This strict rule breaks down where the OEM responsible for the hardware considers the device *end-of-life* and so will no longer receive updates (even for critical security issues). There may also be a situation where there exists an alternate (not provided by the vendor) free software re-implementation of the proprietary firmware, which may be desired for licensing reasons.

In these situations we allow another legal entity to also upload firmware for the hardware, but with a few restrictions:

- The user must manually and explicitly opt-in to the new firmware stream, perhaps using `fwupdmgr switch-branch`, with a suitable warning that there is no vendor support available and that the hardware warranty is now invalid. This means that the alternate firmware must set the device branch appropriately without any additional configuration.
- The alternate firmware must not ship with any code, binaries or generated assets from the original hardware vendor (perhaps including trademarks) unless written permission is provided in writing by the appropriate vendor.

Some real world examples might be providing a Open Source BCM57xx GPL firmware for Broadcom network hardware, or providing a coreboot system firmware for a long-EOLed Lenovo X220 ThinkPad. In this instance, the LVFS may be the legal entity distributing the firmware, which is actually provided by a trusted contributor who has permissions to upload and hardware to test the update. In other cases another legal entity (like coreboot itself) or an individual trusted contributor may be considered the distributor.

In **all** cases the specifics should be discussed with the LVFS maintainers, as should any concerns by licensors or existing distributors.

Note: It is insanity to throw a perfectly working machine into landfill just because it's considered EOL by the original hardware vendor and no longer receiving security updates.

If we can help provide alternate safe firmware, these machines then provide inexpensive access for education and employment for those otherwise unable to afford devices.

CHAPTER

THREE

METADATA

The LVFS needs additional information about the firmware which is included in the uploaded cabinet archive.

3.1 MetaInfo Files

The `.metainfo.xml` file describes the device and firmware and is extra metadata added to the firmware archive by the OEM or ODM. The file is XML format, and uses a subset of the AppStream component specification.

An example `metainfo.xml` file looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Copyright 2018 Richard Hughes <richard@hughsie.com> -->
<component type="firmware">
  <id>com.hughski.ColorHugALS.firmware</id>
  <name>ColorHugALS</name>
  <name_variant_suffix>Black Friday Special Edition</name_variant_suffix>
  <summary>Firmware for the Hughski ColorHug Ambient Light Sensor</summary>
  <description>
    <p>
      Updating the firmware on your ColorHugALS device improves performance and
      adds new features.
    </p>
  </description>
  <provides>
    <firmware type="flashed">84f40464-9272-4ef7-9399-cd95f12da696</firmware>
  </provides>
  <url type="homepage">http://www.hughski.com/</url>
  <metadata_license>CC0-1.0</metadata_license>
  <project_license>proprietary</project_license>
  <releases>
    <release urgency="high" version="3.0.2" date="2017-02-09" install_duration="120">
      <checksum filename="my-custom-name.bin" target="content"/>
      <description>
        <p>This stable release fixes the following bugs:</p>
        <ul>
          <li>Fix the return code from GetHardwareVersion</li>
          <li>Scale the output of TakeReadingRaw by the datasheet values</li>
        </ul>
      </description>
      <issues>
```

(continues on next page)

(continued from previous page)

```
        <issue type="cve">CVE-2016-12345</issue>
        <issue type="cve">CVE-2017-54321</issue>
        <issue type="dell">DSA-2020-321</issue>
        <issue type="intel">INTEL-SA-54321</issue>
        <issue type="intel">INTEL-TA-12345</issue>
        <issue type="lenovo">LEN-28775</issue>
        <issue type="vince">257161</issue>
      </issues>
    </release>
  </releases>
  <!-- we can optionally restrict this update to specific fwupd versions,
  or even previous firmware or bootloader versions -->
  <requires>
    <id compare="ge" version="0.8.0">org.freedesktop.fwupd</id>
    <firmware compare="ge" version="0.1.2"/>
    <firmware compare="ge" version="0.3.4">bootloader</firmware>
  </requires>
  <custom>
    <value key="LVFS::VersionFormat">example</value>
    <value key="LVFS::UpdateProtocol">org.acme.example</value>
  </custom>
  <!-- these keywords are optional and are used for searching -->
  <keywords>
    <keyword>bios</keyword>
    <keyword>dfu</keyword>
  </keywords>
</component>
```

3.2 Using GUIDs

GUID means 'Globally Unique Identifier' and is a 128-bit integer number used to identify a device. GUIDs are often formatted as strings such as `84f40464-9272-4ef7-9399-cd95f12da696`. Another name for GUID is UUID ('Universally Unique Identifier') and the two terms can be used interchangeably. When using GUIDs on the LVFS they **should always be lowercase**.

In fwupd the GUID is generated from the `DeviceInstanceId` strings, so for a single USB device the GUIDs would be generated like this:

```
$ python
>>> import uuid
>>> print uuid.uuid5(uuid.NAMESPACE_DNS, 'USB\VID_0A5C&PID_6412&REV_0001')
52fd36dc-5904-5936-b114-d98e9d410b25
>>> print uuid.uuid5(uuid.NAMESPACE_DNS, 'USB\VID_0A5C&PID_6412')
7a1ba7b9-6bcd-54a4-8a36-d60cc5ee935c
>>> print uuid.uuid5(uuid.NAMESPACE_DNS, 'USB\VID_0A5C')
ddfc8e56-df0d-582e-af12-c7fa171233dc
```

You also can use the online generator to manually convert Instance IDs to GUIDs.

Having multiple GUIDs for each device allows the vendor to choose the GUID for what should match; to match on the `vendor+product+revision` you'd choose the first one, and the `vendor+device` you would use the second. We only

really use the third GUID for fixing a vendor name, or other very broad quirks that apply to all USB devices from a specific vendor.

In the case for PCI devices and other technologies like NVMe, you can dump the GUIDs generated by fwupd using this tool:

```
sudo /usr/libexec/fwupd/fwupdtool --plugin-whitelist nvme get-devices --verbose
...
using e22c4520-43dc-5bb3-8245-5787fead9b63 for NVME\VEN_1179&DEV_010F&REV_01
using 83991323-9951-5adf-b743-d93e882a41e1 for NVME\VEN_1179&DEV_010F
using ad9fe8f7-cdc4-52c9-9fea-31b6f4988ffa for NVME\VEN_1179
...
```

More details about the GUID generation scheme used in each plugin can be found in the `README.md` file in each plugin directory.

Note: Metainfo files can contain as many lines of `<firmware type="flashed">` as required and any device with any of the GUIDs will match the firmware file.

3.3 AppStream ID

The AppStream `<id>` has to be unique for each device firmware stream as it used to *combine* separate `<release>` tags in the `.metainfo.xml` files into the metadata catalog that is downloaded by end users.

Choosing the correct AppStream ID is thus very important for correct operation of the front end tools.

Firstly, the AppStream ID should have a lowercase prefix that matches the reverse-DNS name of your vendor, similar to Java. For instance, appropriate prefixes would be `com.lenovo...` or `org.hughski...`.

The ID should also contain the model type, and perhaps also the module that is being updated if there are (or will be) multiple updates for the same hardware. For instance, we would build the ID further into `org.hughski.ColorHug2.BIOS...`.

The ID should always have a suffix of `.firmware`, which means the finished AppStream ID for this hardware would be `org.hughski.ColorHug2.BIOS.firmware`

Note: The ID has to be totally specific to the GUIDs used to match the device. For hardware that uses a different firmware stream it is important that the AppStream ID does not match existing firmware with the same ID. The LVFS will warn you if you try to upload firmware with the same ID and different sets of GUIDs.

Including the mode name is just convention; you can use the partial GUID appended if this helps, e.g. `com.hughski.ColorHug84f40464.firmware`

Warning: Never include forward or backwards slashes in the ID.

3.4 Update Category

By telling the LVFS the firmware category to use for the component the front end can correctly translate the update type in the UI. Also for this reason, `.metainfo.xml` files **should not** include the words `ME`, `EC`, `BIOS`, `Firmware`, `Device` or `Update` in the component name and they will be removed if included.

The component category can be set as part of the `metainfo.xml` file or set from the LVFS web console. Most users will want to include the extra metadata to make the upload process quicker for QA engineers. To do this, add this to the metainfo file:

```
<categories>
  <category>some-value-here</category>
</categories>
```

3.4.1 Allowed Category Values

Value	Displayed Name
X-System	System Update
X-Device	Device Update
X-EmbeddedController	Embedded Controller Update
X-ManagementEngine	Management Engine Update
X-Controller	Controller Update
X-CorporateManagementEngine	Corporate ME Update
X-ConsumerManagementEngine	Consumer ME Update
X-ThunderboltController	Thunderbolt Controller
X-PlatformSecurityProcessor	Platform Security Processor
X-CpuMicrocode	CPU Microcode Update
X-Configuration	Configuration Update
X-Battery	Battery Update
X-Camera	Camera Update
X-TPM	TPM Update
X-Touchpad	Touchpad Update
X-Mouse	Mouse Update
X-Keyboard	Keyboard Update
X-StorageController	Storage Controller Update
X-NetworkInterface	Network Interface Update
X-VideoDisplay	Video Display Update
X-BaseboardManagementController	BMC Update
X-UsbReceiver	USB Receiver Update
X-Drive	Drive Update
X-FlashDrive	Flash Drive Update
X-SolidStateDrive	SSD Update
X-Gpu	GPU Update
X-Dock	Dock Update
X-UsbDock	USB Dock Update
X-FingerprintReader	Fingerprint Reader Update
X-GraphicsTablet	Graphics Tablet Update

3.5 Update Protocol

The LVFS needs to know what protocol is being used to flash the device. The protocol value is used to provide information about the security of the firmware update to end users.

The update protocol can be set as part of the `metainfo.xml` file or set from the LVFS web console. Most users will want to include the extra metadata to make the upload process quicker for engineers. To do this, add this to the metainfo file:

```
<custom>
  <value key="LVFS::UpdateProtocol">some-value-here</value>
</custom>
```

The latest allowed values for `LVFS::UpdateProtocol` can be found using the LVFS.

3.6 Device Integrity

Some update protocols just transport the image to the target device and make no guarantee of the signing requirements. Such "generic" protocols include NVMe, ATA, DFU and many others.

The device integrity mechanism can be set as part of the `metainfo.xml` file or set from the LVFS web console.

To do this, add this to the metainfo file:

```
<custom>
  <value key="LVFS::DeviceIntegrity">some-value-here</value>
</custom>
```

The allowed values for `LVFS::DeviceIntegrity` are:

- `signed` : The firmware payload is verified on-device the payload using strong cryptography such as RSA, AES or ECC. It is usually not possible to modify or flash custom firmware not provided by the vendor.
- `unsigned`: The firmware payload is unsigned and it is possible to modify and flash custom firmware.

3.7 Version Format

Some hardware returns the version number as a string such as `1.23.4567`, and this is easily handled as a semantic version. In other cases we are not so lucky, and the hardware returns a `uint16_t` or `uint32_t` with no extra metadata about how it should be formatted. This lack of specification precision means that different vendors have chosen to convert the large integer number to various different forms.

The latest allowed values for `LVFS::VersionFormat` can be found on the LVFS.

To override the default of `unknown` vendors should ship extra metadata in the `metainfo.xml` file:

```
<requires>
  <id compare="ge" version="1.2.0">org.freedesktop.fwupd</id>
</requires>
<custom>
```

(continues on next page)

(continued from previous page)

```
  <value key="LVFS::VersionFormat">intel-me</value>
</custom>
```

If the version format is unspecified, and cannot be derived from the `LVFS::UpdateProtocol` then a warning will be shown during upload and the firmware cannot be moved to stable until this is resolved.

Various security teams also want us to always show the device firmware version with the correct format, even if an update is not available. This may be for audit reasons, or just so customers know the version of the firmware compared to release notes written for another operating system. For instance, if the vendor release notes says the firmware should be any version above `39.0.45.x` (formatted as a quad) and the user is running `39.0.11522` (formatted as a triplet) it is not clear to the user what to do.

To change from the default `triplet` version format we can set a fwupd *quirk* on the hardware device. For instance, changing the UEFI Lenovo ME device to use the `intel-me` format. Quirk files can be added upstream for future fwupd versions, or simply copied to `/usr/share/fwupd/quirks.d`. The fwupd daemon will detect the new file and refresh devices as required.

3.8 Device Flags

Some device flags can be populated from the firmware metadata, rather than the more traditional way of setting in a per-plugin `.quirk` file or in the plugin code itself. This allows device flags to be pushed from the server to the client.

For instance, on some hardware, the UEFI UpdateCapsule process would fail to deploy because there was no HDMI/DP display attached. Firmware can *opt-in* to this new requirement by setting a device flag that gets copied to the local fwupd device. If the client is new enough, then any firmware that opts-in then the user will be warned *before* the update is scheduled that a display must be connected to continue – which is a much better user experience than it failing after the user has rebooted to deploy the update.

To use this, the internal device flag can be populated from the firmware metadata:

```
<custom>
  <value key="LVFS::DeviceFlags">display-required</value>
</custom>
```

The exact flags that are allowed for each protocol are restricted, and the allowed values for `LVFS::DeviceFlags` (along with fwupd requirements) can be found via the LVFS.

Note: Only fwupd versions greater or equal to 1.9.1 are able to copy device flags from the metadata, and only devices with the `FU_DEVICE_INTERNAL_FLAG_MD_SET_FLAGS` flag set. If you *require* the device flag to be set for a successful update then you should also have the correct fwupd version requirement to ensure the flag get copied. e.g.

```
<requires>
  <id compare="ge" version="1.9.6">org.freedesktop.fwupd</id>
</requires>
```

Please contact the LVFS administrator if other flags from FuDevice or FwupdDevice are required to be set by the metadata.

3.9 Adding Restrictions

When the user requests updates for a specific device, all the GUIDs provided by the device will be match against any of the GUIDs in the metadata. To limit these matches using a variety or requirements the <requires> tag can be used. For instance, the update can be conditional on the firmware version of another device, or on the kernel version of the installed system.

Requirements can use different methods to compare version numbers.

Type	Example	Description
eq	1.2.3	Equal
ne	1.2.3	Not equal
lt	1.2.3	Less than
le	1.2.3	Less than or equal
gt	1.2.3	Greater than
ge	1.2.3	Greater than or equal
glob	??FWA*	Filename glob
regex	FW[1-7]	Perl compatible regular expression

3.9.1 Using CHID

Newer versions of fwupd can restrict updates to a specific Computer Hardware ID, much like Microsoft update:

```
<requires>
  <id compare="ge" version="1.0.8">org.freedesktop.fwupd</id>
  <hardware>6de5d951-d755-576b-bd09-c5cf66b27234</hardware>
</requires>
```

If multiple <hardware> entries are specified (using an OR | separator) then any may be present.

```
<requires>
  <id compare="ge" version="1.0.8">org.freedesktop.fwupd</id>
  <hardware>6de5d951-d755-576b-bd09-c5cf66b27234|27234951-d755-576b-bd09-c5cf66b27234</
↪hardware>
</requires>
```

Using fwupd >= 1.9.10 the uploader can also deny updates to a specific Computer Hardware IDs:

```
<!-- only newer versions of fwupd understand 'not_hardware' requirements -->
<requires>
  <id compare="ge" version="1.9.10">org.freedesktop.fwupd</id>
  <not_hardware>6de5d951-d755-576b-bd09-c5cf66b27234|27234951-d755-576b-bd09-c5cf66b27234
↪</not_hardware>
</requires>
```

CHIDs can also be added or removed in the LVFS web UI, but only before the firmware is published to stable channel.

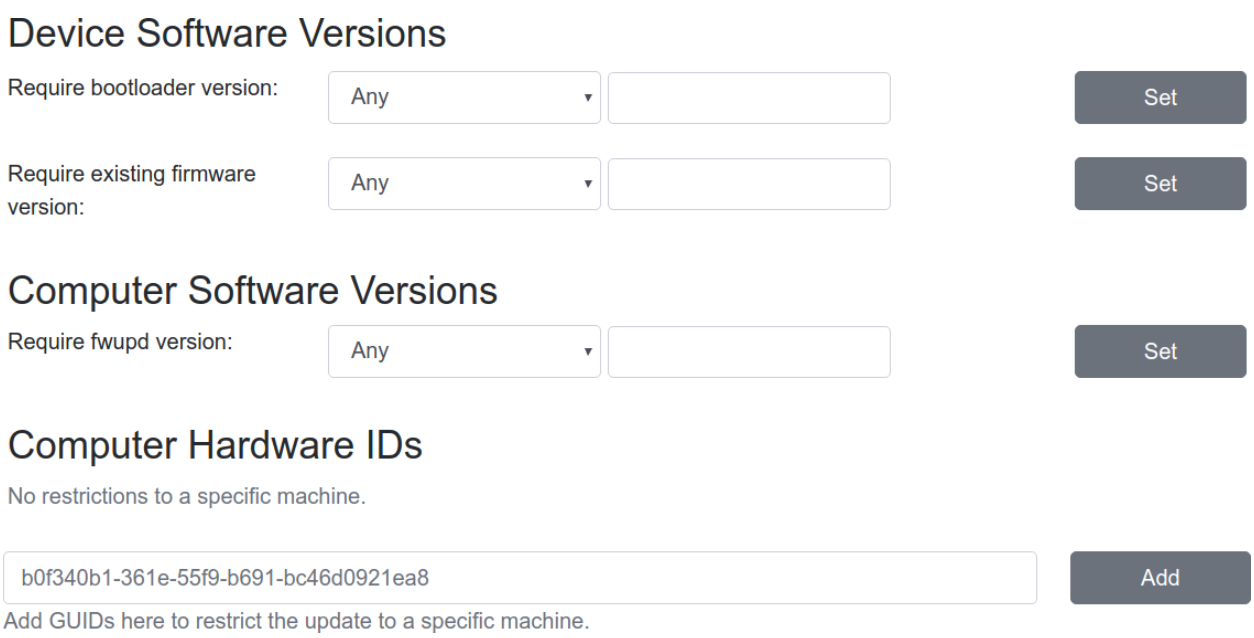


Fig. 1: Modifying requirements of an uploaded firmware.

3.9.2 Other Firmware Version

Newer versions of fwupd can restrict updates on one device depending on the version of firmware on another device. This is most useful when requiring a minimum EC controller version before updating a system firmware, or when a modem firmware needs a specific fix for the baseband firmware:

```
<requires>
  <id compare="ge" version="1.1.3">org.freedesktop.fwupd</id>
  <firmware compare="ge" version="0.1.2">6de5d951-d755-576b-bd09-c5cf66b27234</firmware>
</requires>
```

Newer versions of fwupd can restrict updates on one device depending if another firmware GUID exists on the system of any version. This is similar to the CHID method above but uses the GUID of the firmware, not a hardware ID.

This can be used to ensure that a specific embedded controller is detected for a specific system firmware update, for example.

```
<requires>
  <id compare="ge" version="1.2.11">org.freedesktop.fwupd</id>
  <firmware>6de5d951-d755-576b-bd09-c5cf66b27234</firmware>
</requires>
```

3.9.3 Parent Version

For composite devices such as docks you might want to restrict the child device with respect to the parent, for instance requiring the parent to have greater than a specific bootloader version number.

The other useful thing to use this for is checking if the parent has a specific GUID (of any version) which allows us to match against the common VID&PID instance IDs. This would allow us to restrict a generic child device update to a specific OEM vendor parent.

Depth is specified as `1` to match the parent device and `2` to match the grandparent device:

```
<requires>
  <id compare="ge" version="1.3.4">org.freedesktop.fwupd</id>
  <firmware depth="1" compare="ge" version="0.1.2">bootloader</firmware>
  <firmware depth="1">12345678-1234-1234-1234-123456789012</firmware>
</requires>
```

Newer versions of fwupd can understand an OR requirement using a `|` separator between the listed GUIDs.

```
<!-- only newer versions of fwupd understand parent OR requirements -->
<requires>
  <id compare="ge" version="1.8.9">org.freedesktop.fwupd</id>
  <firmware depth="1">12345678-1234-1234-1234-123456789012|6de5d951-d755-576b-bd09-
↪c5cf66b27234</firmware>
</requires>
```

3.9.4 Sibling Version

Composite devices can also specify that a device sibling has to exist, optionally with a specific version. To do this, specify the depth as `0`:

```
<!-- only newer versions of fwupd understand the 'depth' property -->
<requires>
  <id compare="ge" version="1.6.1">org.freedesktop.fwupd</id>
  <firmware depth="0">12345678-1234-1234-1234-123456789012</firmware>
</requires>
```

3.9.5 Child Version

Composite devices can also restrict the parent device with respect to the child. This is useful when a generic parent device has vendor-specific child devices attached. To do this, specify the depth as `-1` to match any child device.

```
<!-- only newer versions of fwupd understand the negative 'depth' property -->
<requires>
  <id compare="ge" version="1.9.7">org.freedesktop.fwupd</id>
  <firmware depth="-1">12345678-1234-1234-1234-123456789012</firmware>
</requires>
```

3.9.6 Client Features

Versions of fwupd >=1.4.5 can restrict updates depending on the features the client can provide. For instance, if the tools are being run in non-interactive mode then it may not be possible to ask the user to perform a manual action.

Some devices may need to show the user some text or an image of how to manually detach the firmware from runtime mode to bootloader mode.

```
...
<screenshots>
  <screenshot type="default">
    <caption>Unplug the controller, hold down L+R+START for 3 seconds until both LEDs␣
↪are flashing then reconnect the controller.</caption>
    <image>https://raw.githubusercontent.com/hughsie/8bitdo-firmware/master/screenshots/
↪FC30.png</image>
  </screenshot>
</screenshots>
...
<requires>
  <id compare="ge" version="1.4.5">org.freedesktop.fwupd</id>
  <client>detach-action</client>
</requires>
...
```

Other firmware may require showing the user a message or image on how to reset the hardware when the firmware update has completed. This specific post-update message functionality is only available for specific protocols and implemented in some versions of fwupd and GNOME Software.

This action can be performed with one or two metadata keys set in the `.metainfo.xml` file, or chosen using the LVFS component editor.

```
...
<custom>
  <value key="LVFS::UpdateMessage">Please turn the device off and back on again for the␣
↪update to complete</value>
  <value key="LVFS::UpdateImage">https://people.freedesktop.org/~hughsient/temp/unifying-
↪power.png</value>
</custom>
...
<!-- only newer versions of fwupd understand 'client' requirements -->
<requires>
  <id compare="ge" version="1.4.5">org.freedesktop.fwupd</id>
  <client>update-action</client>
</requires>
...
```

Note: You can include either the `UpdateImage` or `<image>` PNG file in the cabinet archive rather than uploading it. In this case use a URL with a `file://` prefix, e.g. `file://unifying-power.png`.

The image will still be mirrored onto the LVFS CDN at upload time, as there is no ability for GUI clients to read binary files from inside the cabinet archive. This means that internet access will still be required when deploying firmware if the image is specified.

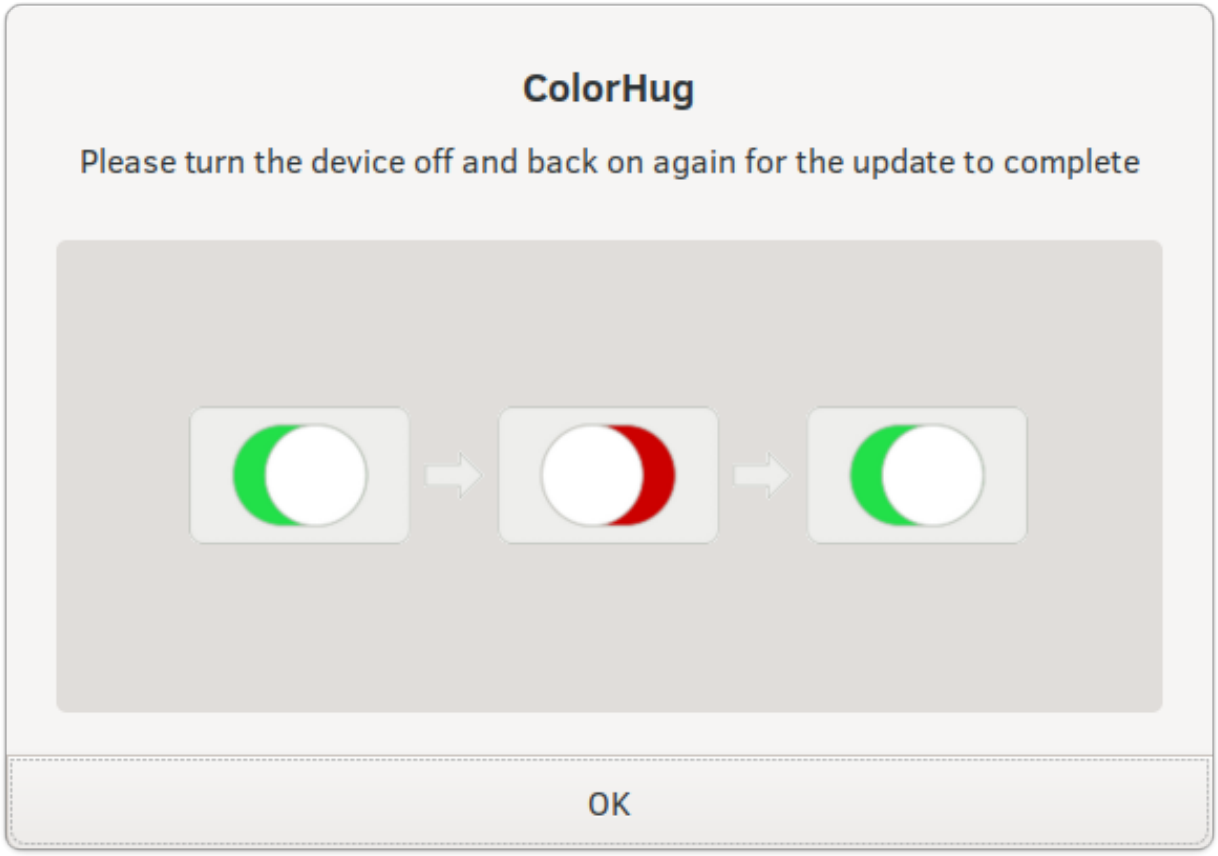


Fig. 2: Showing the user some instructions to reboot the hardware.

3.9.7 Recommended Requirements

All fwupd versions understand "hard" requirements; those that cannot be ignored.

Some vendors may want to add some *suggested* requirements, which can be ignored by the end user if required. Typically this would be done using the `--force` command line option.

A good example here would be from a server OEM who release a set of updates once per quarter. It is expected that the end user only updates from the previous n-3 quarter releases (and not older releases), and it is also vendor recomendation that updates are installed in a specific order. In the field, *however*, admins may want to override this, either due to security policy requirements or to work around specific hardware issues.

```
...
<!-- only newer versions of fwupd understand 'recommends' requirements -->
<requires>
  <id compare="ge" version="1.6.2">org.freedesktop.fwupd</id>
</requires>
<recommends>
  <firmware compare="ge" version="0.1.2">6de5d951-d755-576b-bd09-c5cf66b27234</firmware>
</recommends>
...
```

3.9.8 Restricting Direct Downloads

If you'd rather not have users downloading the .cab archive directly you can opt to hide the direct download links in the LVFS search results. To do this, add this to the metainfo file:

```
<!-- most OEMs do not need to do this... -->
<custom>
  <value key="LVFS::InhibitDownload"/>
</custom>
```

3.9.9 Embargoed and Sanctioned Countries

The LVFS administrator can configure the policy for all firmware owned by the vendor to be blocked from download in embargoed or otherwise sanctioned countries.

The blocked ISO 3166 country codes can also be specified in the firmware itself, using the `LVFS::BannedCountryCodes` metadata key.

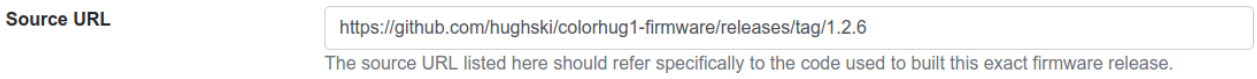
```
<custom>
  <value key="LVFS::BannedCountryCodes">SYR</value>
</custom>
```

3.10 Source Requirements

If a vendor is distributing firmware which contains GPL licensed parts (for example the Linux kernel) then they must include a source URL for the GPL licensed parts in the `releases` section in the metainfo file. This should point to the release-specific source code that can be used to rebuild the binary from the code, for instance:

```
<release>
  <project_license>GPL-2.0-or-later</project_license>
  <release urgency="low" version="1.2.6" >
  <url type="source">https://github.com/hughski/colorhug1-firmware/releases/tag/1.2.6</
↪url>
</release>
```

GPL firmware without source information can not be moved to testing or stable. You can also edit or add the source URL in the existing *Update Details* section in the component view:



3.11 Component Tags

Tags can be used to identify a vendor-specific keyword to the component release, for example identifying components included in a specific service pack or combined update.

Users with fwupd version >= 1.7.3 can install multiple firmware files using the tag value. For example setting `HostBkc=vendor-factory-2021q1` in `/etc/fwupd/daemon.conf` and then doing ``fwupdmgr sync-bkc`` will install all firmwares with that matching tag.

```
<custom>
  <value key="LVFS::UpdateProtocol">some-value-here</value>
</custom>
<tags>
  <tag namespace="lvfs">vendor-factory-2021q1</tag>
</tags>
```

Tag element values should be lowercase, have no whitespace and be namespaced with the vendor name to avoid conflicts.

Vendors wanting to use component tags should ask the LVFS administrator to enable access.

3.12 Device Icons

The icon show in GUI fwupd clients is normally set by the plugin automatically. In some cases the plugin may not know the appropriate icon until firmware has been uploaded to the LVFS.

For this cosmetic purpose the firmware uploader can specify the stock icon in the `metainfo.xml` file which gets put in the the AppStream metadata and used by the graphical clients. In most cases specifying the icon is not required.

To manually override the icon to one of the stock values, use this:

```
<component>
  ...
  <icon type="stock">battery</icon>
  ...
<component>
```

Valid stock icons include:

ac-adapter	
audio-card	
audio-headphones	
audio-headset	
audio-input-microphone	
audio-speakers	
auth-fingerprint	
auth-otp	
battery	
camera-photo	
camera-video	
camera-web	
colorimeter-colorhug	
computer	

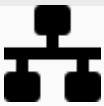
continues on next page

Table 2 – continued from previous page

dock	
dock-usb	
drive-harddisk-ieee1394	
drive-harddisk	
drive-harddisk-solidstate	
drive-harddisk-system	
drive-harddisk-usb	
drive-multidisk	
drive-optical	
drive-removable-media	
gpu	
input-dialpad	
input-gaming	
input-keyboard	
input-mouse	
input-tablet	
input-touchpad	

continues on next page

Table 2 – continued from previous page

media-flash	
media-floppy	
media-optical	
media-removable	
media-tape	
modem	
multimedia-player	
network-vpn	
network-wired	
network-wireless	
pda	
phone	
printer	
printer-network	
scanner	
thunderbolt	

continues on next page

Table 2 – continued from previous page

uninterruptible-power-supply	
usb-hub	
usb-receiver	
video-display	

3.13 Composite Hardware

A vendor can build a single `.cab` archive with multiple firmware files with different `.metainfo.xml` files describing them. This allows a single file to be used to update either multiple devices, or a single *composite* device. An example of a composite device would be a Dell dock, where electrically there are various components connected using USB, but physically it looks like one piece of hardware. Wacom does the same for the various Intuit tablets.

Some tools such as gnome-software may be unable to show more than one update description for the single .cab file. The LVFS also needs to know how to sort the components inside the firmware when showing the logged in user.

To solve this, assign the firmware components a priority, where higher numbers are better. For example `main.metainfo.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<component priority="1" type="firmware">
  <id>com.hughski.dock.firmware</id>
  <name>Hughski Dock Update</name>
  ...
</component>
```

and also `rts1234.metainfo.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<component type="firmware">
  <id>com.hughski.dock.rts1234.firmware</id>
  <name>RTS1234 Update for Hughski Dock</name>
  ...
</component>
```

and `atmel567.metainfo.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<component type="firmware">
  <id>com.hughski.dock.atmel567.firmware</id>
  <name>ATMEL567 Update for Hughski Dock</name>
  ...
</component>
```

The priority can also influence the composite install order. Client-side, fwupd will check the explicit device order (e.g. `Flags=install-parent-first`) and then fallback to the component priority where higher numbers are installed first. This can be used to ensure that releases are always installed in a specific order.

```
[hughsie@hughsie-work$ fwupdmgr get-topology
○
├─ 20EQS64N0C System Firmware                2c1302f31806a0e0d57c377d99e18dae56351413
└─ Dell dock 130W DP                         ef3b3397619993975d045fa2cd00f379f823d0b8
   ├─ RTS5487 in Dell dock                   c23d967939a8badce8b91fc618849f17482b8efd
   ├─ RTS5413 in Dell dock                   fbda92b3b414094763dd6f499db94bdae3f810ee
   ├─ Package level of Dell dock             7bf361a43bd87062755675ce6285472728a4ed50
   └─ VMM5331 in Dell dock                   0b0fcf7385e675a1772fda850694c16ad8d81a7a
```

Fig. 3: Showing the topology of a dock device.

If the priority is specified on the component it is used for all releases, but if per-release control of the composite device is needed, it can instead be set on the `release` node, e.g.

```
<releases>
  <release priority="5">
    ...
  </release>
</releases>
<requires>
  <!-- only newer versions of fwupd understand per-release priorities -->
  <id compare="ge" version="1.9.10">org.freedesktop.fwupd</id>
</requires>
```

In the case where the update order is different between releases the component or device should probably also have `Flags=install-all-releases` to ensure that every version is installed with a predictable release order.

3.14 Further Details

There are currently quite strict limits on the format of the release description included in the `description` part of the metainfo, or edited on the LVFS. For instance, OEMs are not allowed to include links within the text and have to adhere to a limiting style guide. As a workaround, all firmware can now specify an additional url:

```
<release>
  <url type="details">https://www.hughski.com/releases/colorhug1/1_2_6.pdf</url>
</release>
```

This should point to a website page or PDF description of the **specific** release. This would allow vendors to provide more information about specific CVEs or provide more technical information mentioned in the update details. Whist the update details should still be considered the "primary" method to convey information about the firmware release, the URL may be useful for larger OEMs with existing contractual requirements.

3.14.1 Release Urgency Values

It is important to set the urgency of the release to the correct value as this may influence how the client notifies the user. For instance, critical updates may cause a daily session notification to the user, but low priority updates might only be visible when the user manually visits the software center.

Value	Meaning
`low`	Low importance
`medium`	Medium importance, e.g. optional update
`high`	High importance, e.g. recommended update
`critical`	Critical importance, e.g. urgent or security issue

3.15 Screenshots

In some circumstances we may need to ask the user to perform an action to manually put the device into a special firmware-update mode. We can achieve this using a translatable update caption and an optional line-art image:

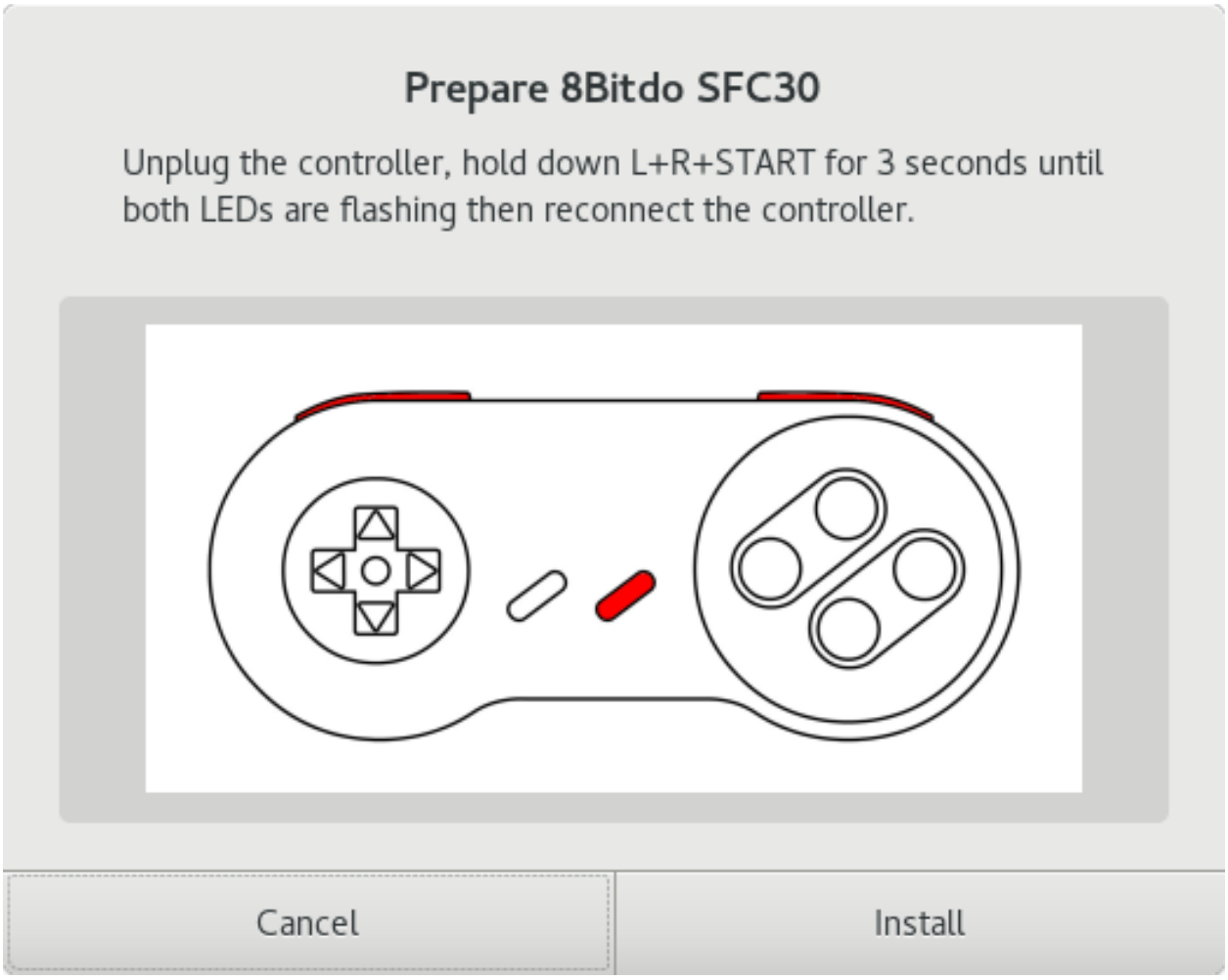


Fig. 4: Showing the user some instructions before updating firmware.

To achieve this the firmware needs to declare the public location of the image in the metainfo file:

```
<?xml version='1.0' encoding='UTF-8'?>
  <component type="firmware">
    ...
```

(continues on next page)

(continued from previous page)

```
    <screenshots>
      <screenshot type="default">
        <caption>Unplug the controller, hold down L+R+START for 3 seconds until both␣
↪LEDs are flashing then reconnect the controller.</caption>
        <image>https://raw.githubusercontent.com/hughsie/8bitdo-firmware/master/
↪screenshots/FC30.png</image>
      </screenshot>
    </screenshots>
    ...
  </component>
```

In the public metadata the URL is rewritten to use the LVFS CDN to preserve the privacy of the remote client.

The screenshot will only be shown by the front end client when the device has the `_NEEDS_BOOTLOADER` flag.

Please also add a `<client>` requirement if the update cannot be performed without showing the image or caption.

3.16 Generic Components

Vendors can include an extra `.metainfo.xml` file with the `<component type="generic">` to supply information used by the LVFS to identify the top-level device. This is only useful when there is no obvious existing high-priority component that can be used for display.

This would be useful for a dock to have the title *WonderDock2* rather than showing a seemingly random sub-component of it on the public pages.

An example `generic.metainfo.xml` file would look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Copyright 2020 Richard Hughes <richard@hughsie.com> -->
<component priority="9" type="generic">
  <id>com.hughsie.WonderDock.firmware</id>
  <name>WonderDock</name>
  <summary>Firmware for the ACME WonderDock</summary>
  <url type="homepage">http://www.hughsie.com/</url>
  <metadata_license>CC0-1.0</metadata_license>
</component>
```

A generic component can also be created for composite firmware manually on the LVFS for firmware that has already been uploaded.

3.17 Style Guide

When all vendors use the same style everything looks more consistent for the end user. Here are some of our suggestions:

3.17.1 <name>

- Use a short device name, e.g. *"ThinkPad P52s"* or *"ColorHug 2"*.
- Use a UTF-8 character (e.g. ™ or ®) rather than `(R)` if required
- Don't include the vendor name

Note: If a component matches two components with the same GUID, please use a forward slash to delimit each model and specifier, for instance, you SHOULD do this:

- `<name>ThinkPad T580/ThinkPad P52s</name>`

. . . and NOT do any of these:

- `<name>ThinkPad T580/P52s</name>`
- `<name>FC30,NES30</name>`
- `<name>ColorHug2 &amp; ColorHug2.1</name>`.

To avoid specifying multiple components in the `<name>` you could also have *one* firmware payload in the cabinet archive referenced by *two* different `.metainfo.xml` files – each with a different DMI CHID, parent or child requirement.

3.17.2 <name_variant_suffix>

- Only use this optional tag if the `<name>` would be duplicated, e.g. if there are two variants of the same hardware
- Use a short string, as it will be appended to the visible name with brackets if required
- Don't duplicate any part of the name

3.17.3 <branch>

- Only use this optional tag if there are multiple vendors providing different firmware streams for the same hardware.
- Use a familiar lower case single word string, as it will be shown in the UI

3.17.4 <summary>

- Refer to the type of hardware, e.g. *"Firmware for the Hughski ColorHug Colorimeter"*
- Include the vendor name before the full device description
- Use a UTF-8 character (e.g. ™ or ®) rather than `(R)` if required

3.17.5 <description>

- Try to avoid explaining the implementation details of the fix, e.g. *"Ensure accurate color profile creation with high screen brightness."* rather than *"Fix overflow in counter when over 500 Lux detected."*
- Do not use overly technical descriptions when simpler text would suffice, e.g. use *"Fix LED color during system start up."* rather than *"Fix LED color during POST."*
- Try to describe fixed bugs and new features from the point of view of the user and how it affects them
- For security or important updates also include the effect of not applying the update, e.g. *"Disk corruption resulting in possible data loss may occur until this update is installed."*

3.17.6 <release tag="N1NET43W" ...>

- The release tag may be optional or required based on component category and vendor policy. If provided it can used to show a vendor-specific text identifier that is different from the version number.
- The tag may be unique only to the model, or be unique for the entire vendor.
- This attribute should not be used if the tag is not used to identify the specific firmware on the vendor homepage.
- Depending on vendor policy, the release tag may be displayed with the header *External release Software ID* or *Machine Type Model*.

CHAPTER

FOUR

UPLOADING FIRMWARE

4.1 Creating a Cabinet Archive

The `.cab` archive format was chosen to match the format expected by Windows Update. This allows vendors to produce one deliverable that can be submitted to the LVFS for signing and then to Microsoft Update, or the other way around. Signatures from one process will not overwrite signatures from another.

It is recommended you name the archive with the vendor, device and version number, e.g. `hughski-colorhug-als-1.2.3.cab` and is suggested that the files inside the `cab` file have the same basename, for example:

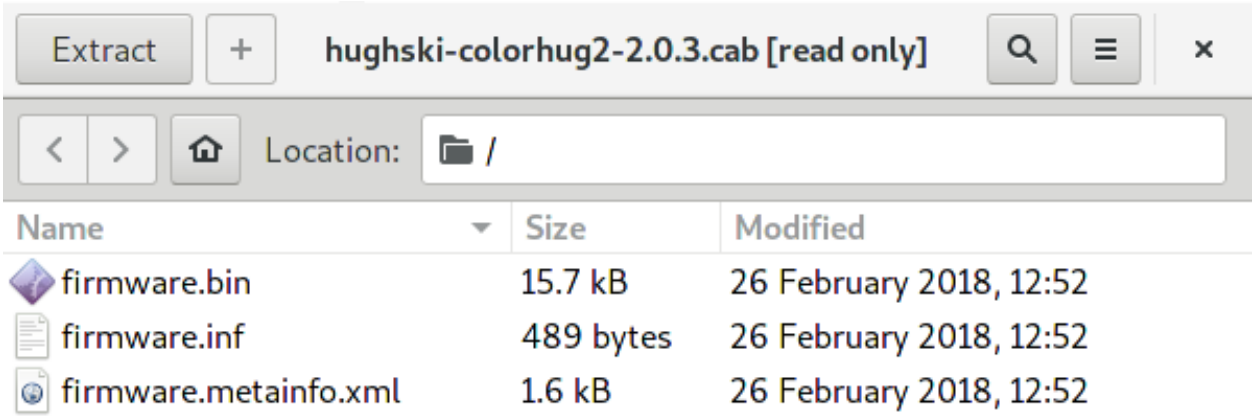


Fig. 1: Files inside a typical archive

4.1.1 Using Linux

Cabinet archives can be created easily on Linux with the the `gcab` command line program. For example:

```
$ gcab -c -v acme-product-name-v1_2_3.cab firmware.metainfo.xml firmware.bin
firmware.metainfo.xml
firmware.bin
```

4.1.2 Using Windows

When building archives on Windows you will need to use the `makecab.exe` program. This works slightly different to `gcab` in that it needs a *manifest* to be created of all the files that are included. To create the manifest create a file called `config.txt` with the following contents:

```
.OPTION EXPLICIT
.set Cabinet=on
.set Compress=on
.set MaxDiskSize=0
.set DiskDirectoryTemplate=.
.set DestinationDir=DriverPackage
firmware.metainfo.xml
firmware.bin
```

Then run `makecab` to create the `1.cab` archive:

```
C:\> makecab /F config.txt
Cabinet Maker - Lossless Data Compression Tool

2,098,010 bytes in 2 files
Total files:              2
Bytes before:     2,098,010
Bytes after:      1,595,399
After/Before:             76.04% compression
Time:                      2.12 seconds ( 0 hr  0 min  2.12 sec)
Throughput:              968.26 Kb/second
```

Warning: If you forget the `.OPTION EXPLICIT` in the manifest then the size of the archive is limited to 1.38Mb. If you try including a firmware with a size greater than this you will see `Invalid folder index` when trying to use fwupdmgr as the archive is not valid.

4.2 Signing The Archive

The upload process repacks the uploaded archive into a new cabinet file and signs the firmware image using a detached GPG or PKCS#7 signature so client tools can be sure the firmware actually originated from the LVFS. Any existing Windows Update signatures are also copied into the new archive although are not used on Linux. The signed archive is prefixed with the hash of the uploaded file to avoid clashes with other uploaded files and to make the download location non-predictable.

4.3 Remotes

Normally firmware is uploaded to a `private` remote. This firmware is available to only the user that uploaded it, and any QA users in the vendor group. It is not visible to end-users, other vendors or to fwupd running locally.

Firmware can be moved to a so-called `embargo` remote that means that is included in the private metadata catalog that is available for any users in the same vendor group. It is not available to any other vendors (even vendors acting as ODM or OEM) and is also not available to the public.

Once the firmware is moved to `testing` it is available to the general public, typically a few thousand users who have opted-in to testing pre-release firmware.

Then the firmware can be moved to `stable` which makes it available to tens of millions of public users.

4.4 Affiliated Vendors

The affiliates feature on the LVFS may be interesting to larger OEMs, as it allows users working for other ODMs to upload firmware on the OEMs behalf.

First, some nomenclature:

- **OEM**: Original Equipment Manufacturer, the user-known company name on the outside of the device, e.g. Sony, Panasonic, etc.
- **ODM**: Original Device Manufacturer, typically making parts for one or more OEMs, e.g. Foxconn, Compal, etc.

There are some OEMs where the ODM is the entity responsible for uploading the firmware to the LVFS. The per-device QA is typically done by the OEM, rather than the ODM, although it can be both. Allowing the ODM to log in as the OEM is not good design from a security, privacy or audit point of view.

The LVFS administrator can mark other vendors as *affiliates* of other vendors. This gives the ODM permission to upload firmware that is *owned* by the OEM to the LVFS, and that appears in the OEM embargo metadata. The OEM QA team is also able to edit the update description, move the firmware to testing and stable (or delete it entirely) as required. The ODM vendor account also doesn't have to appear in the search results or the vendor list, making it hidden to all users except the OEM.

This also means if an ODM like Foxconn builds firmware for two different OEMs, they also have to specify which vendor should own the firmware at upload time. This is achieved with a simple selection widget on the upload page, but is only shown if affiliations have been set up.

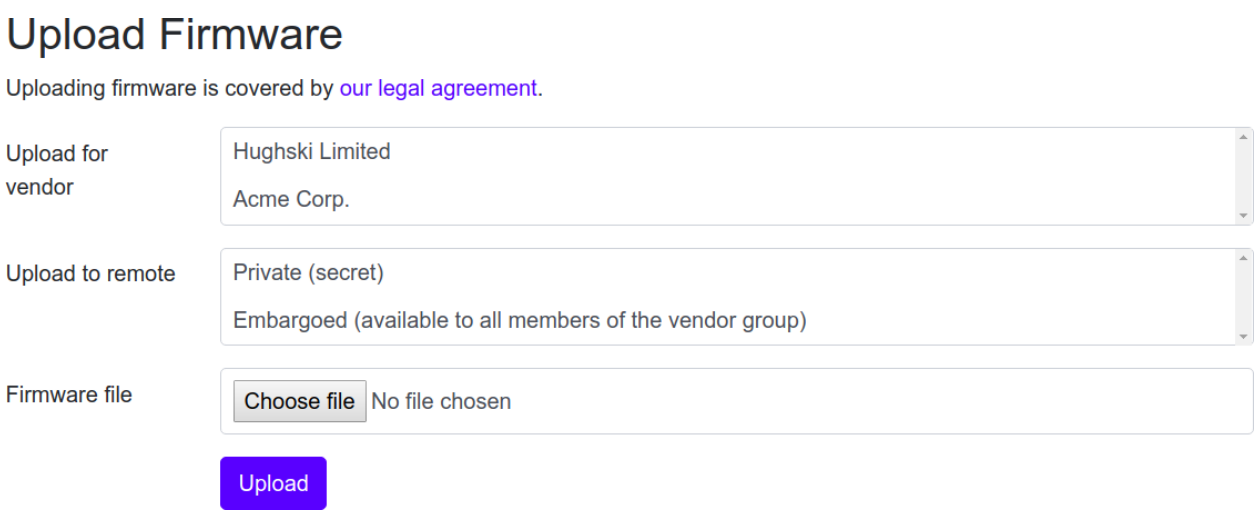


Fig. 2: Upload page for ODM.

The ODM is able to manage their user accounts directly, either using local accounts with passwords, or ODM-specific OAuth which is the preferred choice as it means there is only one place to manage credentials.

4.4.1 Moving Firmware From ODM to OEM

In some instances it is better to upload firmware by the ODM vendor to the ODM group, rather than the affiliated OEM. This would let anyone in the ODM QA group modify the update, for instance changing the update description or performing an end-to-end test.

Once the firmware has been tested, it can be *moved* to the OEM account, although it can only be moved back by the OEM as the ownership has been transferred.

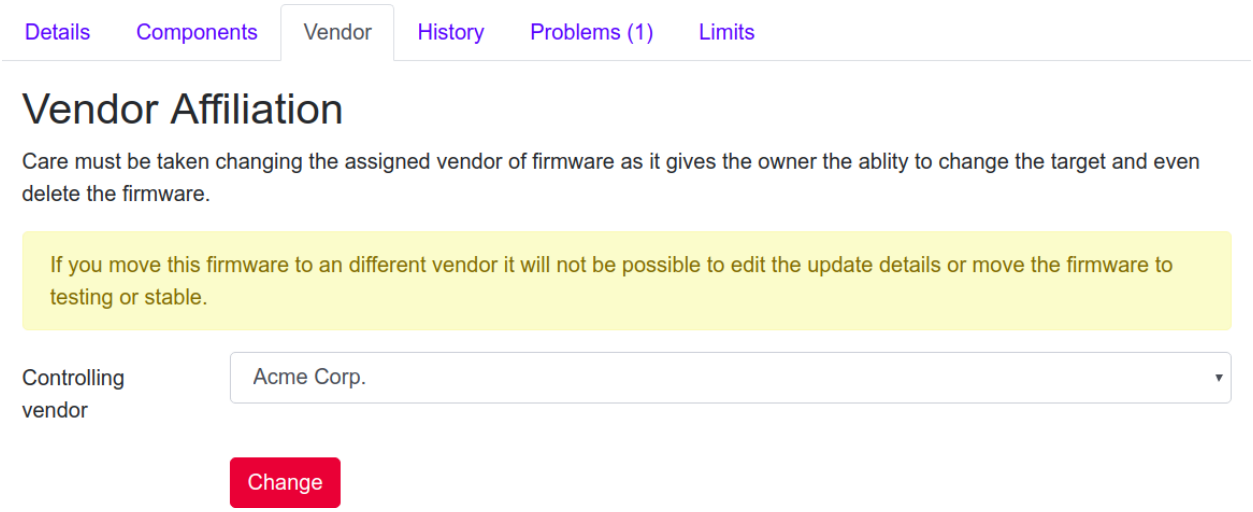


Fig. 3: Moving a firmware to a different vendor.

4.5 Automatic Uploads

You can automate the upload of firmware from a build pipeline by creating a *user token*. This can then be used to upload firmware for that user using a script, e.g.

```
curl -X POST -F file=@/tmp/foo.cab https://fwupd.org/lvfs/upload/token \
     --user "username@domain.com:USERTOKENHERE"
```

All firmware visible for the user can also be queried in the same way:

```
curl https://fwupd.org/lvfs/firmware/auth --user "username@domain.com:USERTOKENHERE"
```

The mapping from vendor name to LVFS vendor ID can be found using:

```
curl https://fwupd.org/lvfs/vendors/auth --user "username@domain.com:USERTOKENHERE"
```

Warning: **Do not use your login password!** Generate a token when logged in to the LVFS using the User Profile settings.

CHAPTER

FIVE

FIRMWARE TESTING

5.1 Online Tests

When a firmware format is set in the `metainfo.xml` file various tests are performed on the firmware by the LVFS. This includes checking file headers, magic numbers or CRCs for the chosen update protocol.

The update protocol can be changed on the LVFS website, and the correct tests will be run automatically. Firmware that has unresolved test failures cannot be pushed to the `testing` or `stable` remotes. For some tests the failure can be *waived* by a QA user.

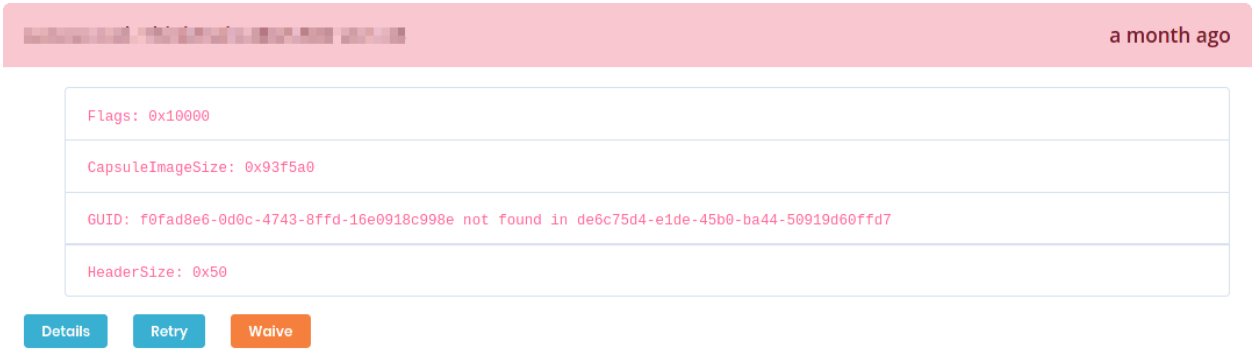
5.1.1 UEFI Capsule

Capsule updates should be uploaded with a valid `CAPSULE_HEADER` that contains a GUID listed in the `metainfo.xml` file.

For reference, the UEFI capsule header is defined like this:

```
typedef struct {
EFI_GUID    CapsuleGuid;
UINT32      HeaderSize;
UINT32      Flags;
UINT32      CapsuleImageSize;
} EFI_CAPSULE_HEADER;
```

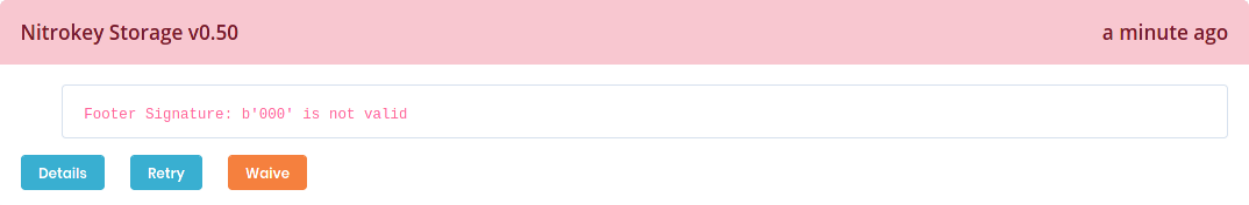
If the header is missing or invalid the test will fail, although the failure can be waived by a QA user.



5.1.2 DFU

DFU updates must be uploaded with a valid `UFD` footer that matches the device revision number with a correct CRC value.

Although these can be waived by a QA user, firmware uploaded without a footer can be installed on any DFU device, which makes this unwise.



`dfu-tool` from the `fwupd` project can convert a *raw* firmware image to include a DFU header, for example:

> $ dfu-tool convert dfu old.raw new.dfu $ dfu-tool set-vendor new.dfu 0xabcd $ dfu-tool set-product new.dfu 0x1234

5.1.3 Blocklist

All update binaries, and shards contained within are scanned for strings which may indicate a problem with the firmware. Example strings are:

- `DO NOT SHIP`
- `To Be Defined By O.E.M`

Although these can be waived by a QA user, firmware should not be uploaded that have this text.

Additionally, the blocklist plugin will search for other information that may add a component claim. For instance the `computrace` claim will be added to any firmware shipping the official Computrace agent, and it will be visible to users when viewing the component information.

5.1.4 Microcode

All UEFI updates are decompressed, and if a processor microcode is found then it is compared with older firmware versions that have been uploaded to the LVFS.

If the microcode has been downgraded then the test will fail, although the failure can be waived by a QA user.

5.1.5 PE Check

Any EFI shards are loaded and will have their PE signatures checked. If any certificate is out of date, or otherwise invalid a test failure will appear. This failure can be waived by a QA user.

5.2 End-to-End testing

5.2.1 Embargo remotes

Once the firmware is in an embargo remote anyone in the vendor group can then download the `vendor-embargo.conf` from the LVFS metadata page and install it locally on their Linux system.

> **Warning:** The `vendor-embargo.conf` file should never be emailed to anyone not in your vendor group.
>
> If you want to allow access to an ODM or OEM this can be done by transferring the ownership of the firmware.

To use the embargo remote:

1. Download a new version of `vendor-embargo.conf` from the LVFS
2. Put your LVFS email in the `Username=` field in `vendor-embargo.conf`
3. Generate a LVFS user token for the `Password=` field in `vendor-embargo.conf`
4. Install it to `/etc/fwupd/remotes.d` if using a distribution build of fwupd, or `/var/snap/fwupd/common/var/lib/fwupd/remotes.d` if using the snap build
5. Use `fwupdmgr refresh` to download the new metadata

> **Warning:**
>
> - Do **not** rename the `vendor-embargo.conf` to `lvfs.conf` – both are required
> - Do **not** manually modify the existing `lvfs.conf` file

- You should use a fwupd versions newer than 1.7.9 – use `sudo apt remove fwupd` then `sudo snap install fwupd` to get a newer fwupd on Ubuntu LTS.

After waiting a few minutes for the LVFS to regenerate the vendor group metadata, the user can do `fwupdmgr refresh` to get the new metadata which includes the new firmware release. Once the new metadata is available on the local system the device can be updated either using `fwupdmgr update` or using GNOME Software.

Note: You can force GNOME Software to update the metadata catalog using the *refresh* button in the left hand side of the header bar in the *Updates* panel.

5.2.2 Testing and stable remotes

You should only move stable firmware to testing and stable after completing an end-to-end test with the embargo remote.

Warning: It can take a few hours to regenerate the `testing` and `stable` remotes and up to **24 hours** for users to download the new metadata catalog. Most vendors see a large spike in downloads the day **after** they move a firmware to stable, and then a steady decay the days after.

5.2.3 Debugging Metadata

If you've moved the firmware to `embargo`, waited for the remote to regenerate, and then done `fwupdmgr refresh` and still do not have any update available you can check for the new release in the downloaded metadata using vim:

```
$ cat /var/lib/fwupd/remotes.d/NAME_OF_VENDOR-embargo/metadata.xml.gz | gunzip | less
```

```
<?xml version='1.0' encoding='UTF-8'?>
<components origin="lvfs" version="0.9">
  <component type="firmware">
    <id>com.8bitdo.fc30.firmware</id>
    <name>FC30 Device Update</name>
    ...
    <requires>
      <id compare="ge" version="0.9.3">org.freedesktop.fwupd</id>
    </requires>
    <screenshots>
      <screenshot type="default">
        <caption>Unplug the controller, hold down L+R+START for 3 seconds until both␣
↪LEDs are flashing then reconnect the controller.</caption>
        <image>https://raw.githubusercontent.com/hughsie/8bitdo-firmware/master/
↪screenshots/FC30.png</image>
      </screenshot>
    </screenshots>
    <releases>
      <release timestamp="1520380800" urgency="medium" version="4.10">
        <location>https://fwupd.org/downloads/2999ee63c0cff96893c1614955f505cb4f0fa406-
↪8Bitdo-SFC30_NES30_SFC30_SNES30-4.10.cab</location>
        <checksum type="sha1" filename="2999ee63c0cff96893c1614955f505cb4f0fa406-8Bitdo-
```

(continues on next page)

(continued from previous page)

```
↪SFC30_NES30_SFC30_SNES30-4.10.cab" target="container">
↪a60593fd1dbb40d7174c99f34b5536f45392bf6c</checksum>
        <checksum type="sha1" filename="N30_F30_firmware_V4.10.dat" target="content">
↪f6e4fe9c56585e200b8754d59eb1e761090bd39f</checksum>
        <description>
          <p>Enhanced the stability of the Bluetooth pairing.</p>
        </description>
        <size type="installed">46108</size>
        <size type="download">53407</size>
      </release>
      <release timestamp="1506038400" urgency="medium" version="4.01">
        <location>https://fwupd.org/downloads/fe066b57c69265f4cce8a999a5f8ab90d1c13b24-
↪8Bitdo-SFC30_NES30_SFC30_SNES30-4.01.cab</location>
        <checksum type="sha1" filename="fe066b57c69265f4cce8a999a5f8ab90d1c13b24-8Bitdo-
↪SFC30_NES30_SFC30_SNES30-4.01.cab" target="container">
↪78ef2663beaa952415c3719447b0d2ff43e837d8</checksum>
        <checksum type="sha1" filename="bluetooth_firmware_v4.01.dat" target="content">
↪f6cacd2cbae6936e9630903d73c3ef5722c4745c</checksum>
        <description>
          <p>Fixed input lag problem when used with other controllers.</p>
        </description>
        <size type="installed">45596</size>
        <size type="download">52085</size>
      </release>
    </releases>
    <provides>
      <firmware type="flashed">7934f46a-77cb-5ade-af34-2bd2842ced3d</firmware>
      <firmware type="flashed">7a81a9eb-0922-5774-8803-fbce3ccbcb9e</firmware>
    </provides>
  </component>
  ...
```

Here you can see a lot of information. Some interesting points:

- The 4.10 and 4.01 `.metainfo.xml` files have been combined into one <component> using the <id> to combine them.
- They always share the same set of screenshots
- They always share the same set of GUIDs
- They always share the same set of requirements

You can also examine the stable metadata the same way:

```
$ cat /var/lib/fwupd/remotes.d/lvfs/metadata.xml.gz | gunzip | less
```

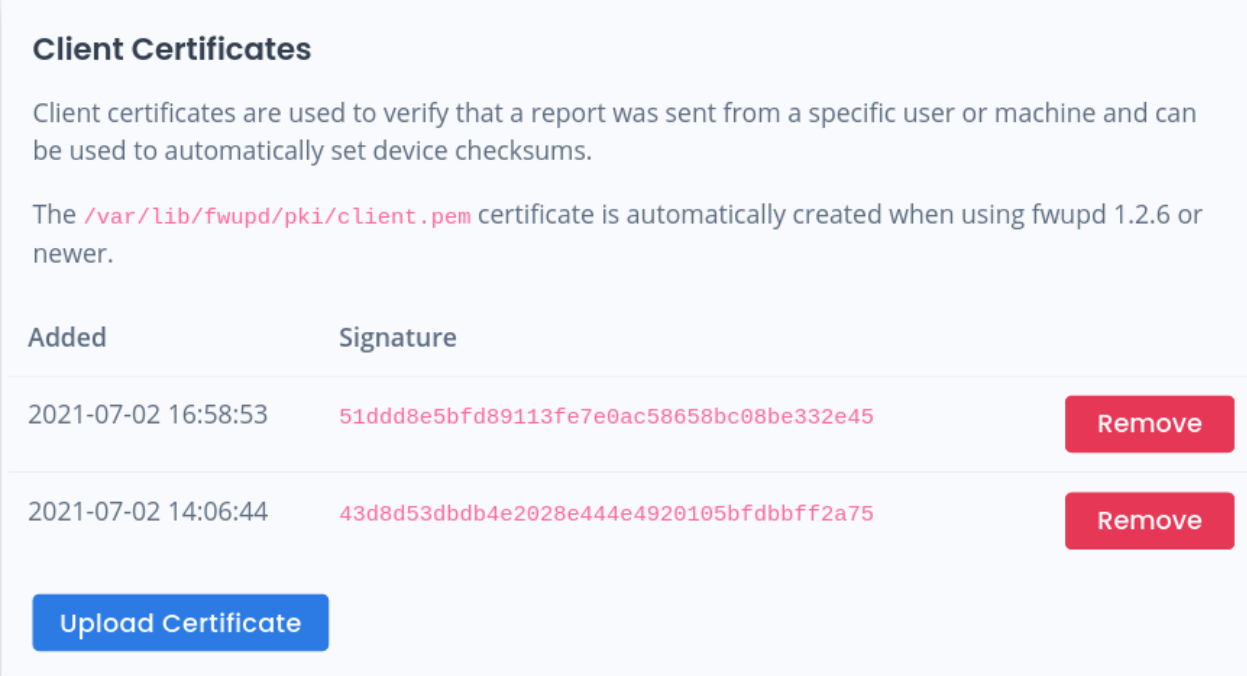
5.2.4 Signed Reports

After each update the fwupdmgr client tools allow the end user to submit a "report" which is used by the firmware owner to validate the firmware deployment is correct. Any failures can be analyzed and patterns found and the metadata can be fixed. For instance, the failures might indicate that the required fwupd version needs to be raised to a higher value, or that the update requires a specific bootloader version.

Part of the anonymous report also includes the device checksum which can be used to verify the firmware is being deployed correctly. All users can submit reports, and there is no way to verify the report has not been modified by the end user before submission. This means reports should not be used for trust, and the information only used when statistically significant.

There is provision in `fwupd` 1.2.6 and newer to actually sign the report contents using a per-machine certificate. This allows the LVFS to verify the report has not been modified after being signed, and also means the LVFS now knows what user submitted the report. If the LVFS knows which user (and thus which vendor) owns which certificate the report can then be used for trusted operations. For instance, setting the "golden" device checksums for the update, or verifying that the firmware was indeed tested on specific hardware.

To do this, the user must add at the certificate from each machine used for testing:



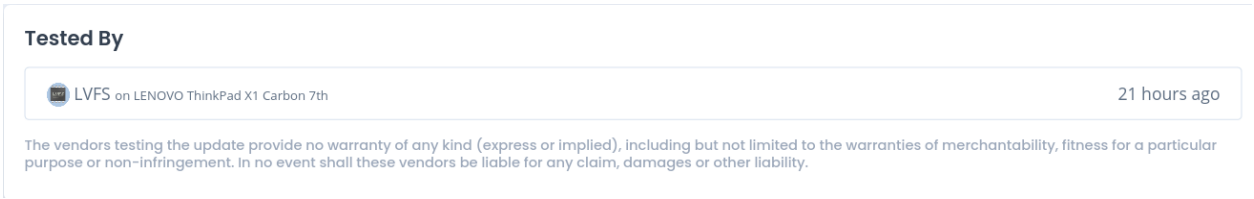
The user can then upload reports to the LVFS in a trusted way by signing the report:

```
$ fwupdmgr refresh
$ fwupdmgr update
...reboot...
$ fwupdmgr report-history --sign
```

This is then reflected in the public device pages for vendors that have public accounts:

Note: The version of fwupd is shown for peripheral updates, and the system DMI information is used for system updates.

Interestingly, this information is also exported into the public metadata, e.g.




```
...
<artifacts>
  <artifact type="binary">
    ...
    <testing>
      <test_result date="2021-07-14">
        <vendor_name id="1">LVFS</vendor_name>
        <device>LENOVO ThinkPad X1 Carbon 7th</device>
        <os version="34" variant="workstation">fedora</os>
        <previous_version>1.2.6</previous_version>
        <custom>
          <value key="RuntimeVersion(org.freedesktop.fwupd)">1.6.2</value>
        </custom>
      </test_result>
    </testing>
  </artifact>
</artifacts>
...
```

Additionally, the signed report can be made available to 3rd parties, typically as part of a hardware certification program.

Note: Sharing the URL allows a user (possibly outside your organization) to read the specific report details but not modify the firmware or report in any way.

5.2.5 Signed Reports With Token

In some automated tested scenarios the image on the DUT is ephemeral and manually uploading the host-generated signing key to the LVFS is not appropriate.

In this case, allow the user to upload a report using Basic authentication so that the LVFS can treat it as "signed" by the report uploader.

This means the user can do (on fwupd >= 1.9.1):

```
$ fwupdmgr modify-remote lvfs Username sign-test@fwupd.org
$ fwupdmgr modify-remote lvfs Password V92KVWWCA5VKYUSX
$ fwupdmgr refresh
$ fwupdmgr update
...reboot...
$ fwupdmgr report-history
```

… and the LVFS treats that as if `--sign` was used if the authentication succeeds.

5.2.6 Offline Reports

A detached test report can be used when the test machine has no public internet connection.

This means the user can do (on fwupd >= 1.9.10):

```
$ fwupdmgr install filename.cab
...reboot...
$ fwupdmgr report-export --sign
$ cp *.fwupdreport /media/YOUR_USBDISK/
```

Then on a different (internet-connected) machine, log into the LVFS, navigate to the specific firmware page and click the *Reports* tab. From there the `/media/YOUR_USBDISK/*.fwupdreport` files can be uploaded:

Note: The LVFS uses the current logged in user to assign the test user and vendor group.

CHAPTER

SIX

CLAIMS

Firmware uploaded to the LVFS is scanned, and attributes about the update are added automatically.

Some claims may be positive, for instance if hardware supports verification. Negative claims are also added, for instance if verification checksums are missing. Informational *neutral* claims are also added, which are not positive or negative, but may be a consideration for the user, e.g. if Computrace is included.

6.1 UEFI Shell

Including the Shell.efi in a firmware update can create additional supply chain security risks. From the UEFI shell it is very easy to downgrade processor microcode or to abuse the existing update process. It also makes attacking SMI handlers much easier, e.g. ThinkPwn.

The EFI shell allows direct RW access to memory using `mm` command, which by itself defeats SecureBoot and everything else that's security is based on memory not being being attacker-controlled.

6.2 Old Microcode

Processor microcode can be thought of runtime firmware for the CPU processor itself. It maps "high level" x86 instructions to hardware micro-opcodes that are specific to the processor. Microcode is supplied as an encrypted blob by CPU vendors like Intel and AMD and cannot be modified in any way by the end user. Only microcode signed by the processor vendor can be loaded onto the CPU.

In some cases, the processor vendor will issue a new microcode to address an issue, which may be security sensitive. This has been done many times in the past, e.g. to fix or mitigate the Spectre, Meltdown and Foreshadow security issues. In some cases microcode updates are even done to increase performance for a specific workload.

If a firmware is tagged as *_containing old microcode* it doesn't always mean that there is an unpatched security issue. Some microcode is vendor-specific, so for instance Lenovo might create an update on the LVFS that updates the version of microcode of CPUID 0x906ec from 0xd2 to 0xd3. Although Dell might be using the same processor, the motherboard hardware is not affected and no update will be prepared.

6.3 Computrace

When a computer equipped with Computrace is reported stolen, the firmware agent attempts to notify the monitoring center, allowing the Absolute Theft Recovery Team to forensically mine the computer using a variety of procedures including key captures, registry scanning, file scanning, geolocation, and other investigative techniques to determine who has the computer and how it is being used. Absolute then works with local law enforcement agencies to recover the computer.

Due to the way the agent works, it's often seen as a "legitimate" firmware implant, which may be a consideration when purchasing hardware.

The Computrace agent is nonfunctional under Linux and only works when using Microsoft Windows XP and newer.

The related LoJax UEFI rootkit hijacks the Computrace agent for malicious puposes.

6.4 EDK Debug Agent

No production firmware should include the EDK Debug Agent as it allows the end user to trivially disable host protections like BootGuard, and potentially also allows unauthenticated access to SMM, which is the most secure layer in the machine.

6.5 HP Sure Start

Every time the PC powers on, HP Sure Start automatically validates the integrity of the BIOS code to help ensure that the PC is safeguarded from malicious attacks.

Once the PC is operational, runtime intrusion detection constantly monitors memory. In the case of an attack, the PC can self-heal using an isolated "golden copy" of the BIOS in less than a minute.

HP Sure Start is a hardware technology available only on some HP hardware.

6.6 Intel BIOS Guard

BIOS guard helps ensure that firmware malware stays out of the BIOS by blocking all software based attempts to modify protected BIOS without the platform manufacturer's authorization.

Typically, this is implemented by blocking SMM writes to the SPI flash chip.

6.7 Intel Boot Guard

Intel Boot Guard is a technology introduced by Intel in the 4th Intel Core generation (Haswell) to verify the boot process. This is accomplished by flashing the public key of the BIOS signature into the write-once field programmable fuses of the CPU itself, typically during the manufacturing process.

In this way it has the public key of the BIOS and it can verify the correct signature of the firmware during every subsequent boot. Once enabled by the manufacturer, Intel Boot Guard cannot be disabled.

6.7.1 Signed Firmware

Firmware can either be signed or unsigned. Signed in this context means the binary code has been either signed or encrypted using private-public asymmetric key cryptography.

It does not include firmware protected with weak symmetric methods such as XTEA as the private key would need to be stored on the device itself, which is insecure. It also does not include firmware "protected" with checksums like CRC32.

Devices supporting signed firmware can **only** be updated by the original OEM and alternate "homebrew" or malicious firmware cannot be written.

6.7.2 Verified Firmware

When devices are flashed with new firmware the device will normally self-check that the data has been written correctly. Some devices just write new data to an SPI flash chip and hope for the best.

6.7.3 Device Checksums

When devices are flashed with new firmware the device will normally verify that the data has been written correctly. Devices supporting verified firmware either allow the host to read back the written firmware at a later time, or will return a internally-calculated checksum.

This allows users to verify that devices have not been tampered with, which may even be a concern before first use due to supply chain attacks.

For UEFI firmware, although the firmware capsule is signed by the OEM or ODM, software can't reliably read the SPI EEPROM from userspace. The UEFI firmware does provide a hash of the firmware, or more specifically, a hash derived from the stored firmware event log.

A final hash of all the TPM firmware events is stored in the TPM chip as `PCR0`.

To list the various `PCRs` on the running system you can use `cat /sys/class/tpm/tpm0/pcrs` for TPMs using protocol 1.2, or `tpm2_listpcrs` for TPMs using protocol 2.0. The PCR0 can be included in the vendor-supplied `firmware.metainfo.xml` in the cabinet archive:

```
<releases>
  <release date="2019-01-08" urgency="high" version="1.2.3">
    <checksum type="sha1" target="device">ce7dd93006be33bcce1a1965cb69634bd0a0fe35</
↪checksum>
    <checksum type="sha256" target="device">
↪c479988947653b403d6a4ebe366cc60eaf7b6e147bd058fb524be418890655c9</checksum>
  </release>
</releases>
```

Multiple *golden* device checksums are possible for each system depending on the specific set up options. For instance, enabling or disabling Intel TXT would change the system `PCR0` checksum.

The device checksums can also be set using the admin console of the LVFS:

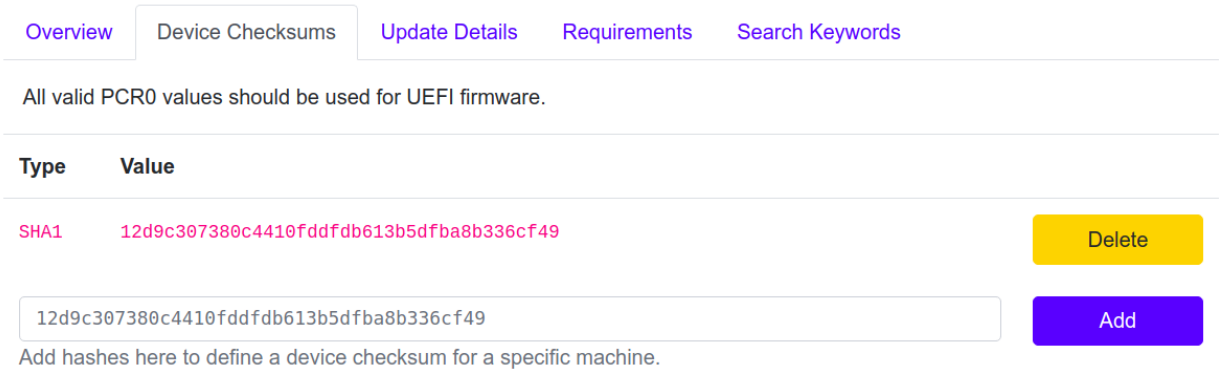


Fig. 1: Adding PCR0 checksums to a component for attestation

6.7.4 Vendor Provenance

The LVFS only allows OEMs, ODMs and silicon vendors to upload firmware. Some OEMs allow the ODM to QA firmware on their behalf and for this reason there are strictly controlled "affiliate relationships" defined on the LVFS.

Furthermore, the AppStream prefix is checked on upload, to prevent the vendor trying to replace or inpersonate another vendors legitimate firmware. This namespacing keeps the OEMs firewalled from each other.

Client side there is another check which verifies the **uploader** of the firmware has the matching set of restrictions for the USB or PCI-assigned vendor ID. For instance, Hughski Limited can only deploy firmware onto devices with `VendorId=USB:0x273F` and so even if the LVFS account for this company was hacked they could not update firmware from Logitech or Wacom.

6.7.5 Source URL

All firmware licensed with a GPL-like license must include links to the exact source release used to build the firmware update. This claim is only shown for firmware that requires a source URL, although can be included even for non-open-source firmware if required.

6.7.6 Virus Safe

All firmware uploaded to the LVFS gets scanned by the ClamAV security scanner. Additionally, when the firmware is no longer embargoed and available to the public it is uploaded to VirusTotal for further anaysis.

6.7.7 FwHunt

Most UEFI firmware images uploaded to LVFS are scanned by the Binarly FwHunt community scanner to check for publicly disclosed security issues. Security issues still under vendor embargo are not detected.

Any potential issues detected are visible to the OEM vendor and uploader, but are not shown to end users. When a firmware image has a detectable issue, the exact details will not be displayed here.

Firmware is scanned with the latest set of public rules at upload time, and may be scanned again at a later date when new rules become available.

Please contact Binarly if you would like more details about FwHunt technology.

6.7.8 End-of-Life

Some devices can be marked as "end-of-life" as they are no longer supported by by the original OEM. These are unlikely to get updates to fix critical security problems.

6.8 Software Bill of Materials

All firmware uploaded to the LVFS gets scanned for both CoSWID data embedded in the SBOM section of the COFF binaries, but also uSWID *external* metadata.

For instance, there may be embedded CoSWID metadata in 75 PE files, where each EFI binary contributes information to the composite package SBoM. This is possible as we can include the CoSWID metadata in the PE files at build time, generating accurate data automatically.

Sometimes it is not possible is in embed the CoSWID metadata directly into a proprietary or vendor-specific section, e.g. AMD microcode or Intel FSP. For these binary blobs it's expected that the IVH or the system integrator will generate some external metadata about the non-free blob and include it in the system image somehow. This might be in an FV section for an EFI image, the DT for an ARM image, or just appended as raw data in a free section in the ROM file.

You also either use uswid directly, or the online generator to build the external SBoM data for the binary deliverable.

If multiple uSWID SBoM metadata sections are detected then they are appended.

CHAPTER

SEVEN

USER TELEMETRY

By allowing fwupd to *phone home* after attempting a firmware update, it allows the hardware vendor that uploaded firmware to know there are problems straight away, rather than waiting for frustrated users to file bugs.

The report contains information that identifies the machine and old/new firmware versions, and in the event of an error, enough debug information to actually be useful. It obviously involves sending the user's IP address to the server too.

We have to be exceptionally careful with users' privacy and trust. We cannot just enable automated collection, and this document outlines what we implemented for fwupd >= 1.0.4. This functionality should be acceptable to even the most paranoid of users.

The `fwupd` daemon stores the result of each attempted update in a local SQLite database. In the event there is a firmware update that has been attempted, we now ask the user if they would like to upload this information to the LVFS. Using GNOME this would just be a slider in the control center privacy panel, although this feature is currently unimplemented.

If the user is using the `fwupdmgr` tool this is what it shows:

```
$ fwupdmgr report-history
Target:                  https://the-lvfs-server/lvfs/firmware/report
Payload:                 {
                         "ReportVersion" : 1,
                         "MachineId" :
↪"9c43dd393922b7edc16cb4d9a36ac01e66abc532db4a4c081f911f43faa89337",
                         "DistroId" : "fedora",
                         "DistroVersion" : "27",
                         "DistroVariant" : "workstation",
                         "Reports" : [
                           {
                             "DeviceId" : "da145204b296610b0239a4a365f7f96a9423d513",
                             "Checksum" : "d0d33e760ab6eeed6f11b9f9bd7e83820b29e970",
                             "UpdateState" : 2,
                             "Guid" : "77d843f7-682c-57e8-8e29-584f5b4f52a1",
                             "FwupdVersion" : "1.0.4",
                             "Plugin" : "unifying",
                             "Version" : "RQR12.05_B0028",
                             "VersionNew" : "RQR12.07_B0029",
                             "Flags" : 674,
                             "Created" : 1515507267,
                             "Modified" : 1515507956
                           }
                         ]
                       }
Proceed with upload? [Y|n]:
```

Using this new information that the user volunteers, we display a few new sections in the LVFS web-console:

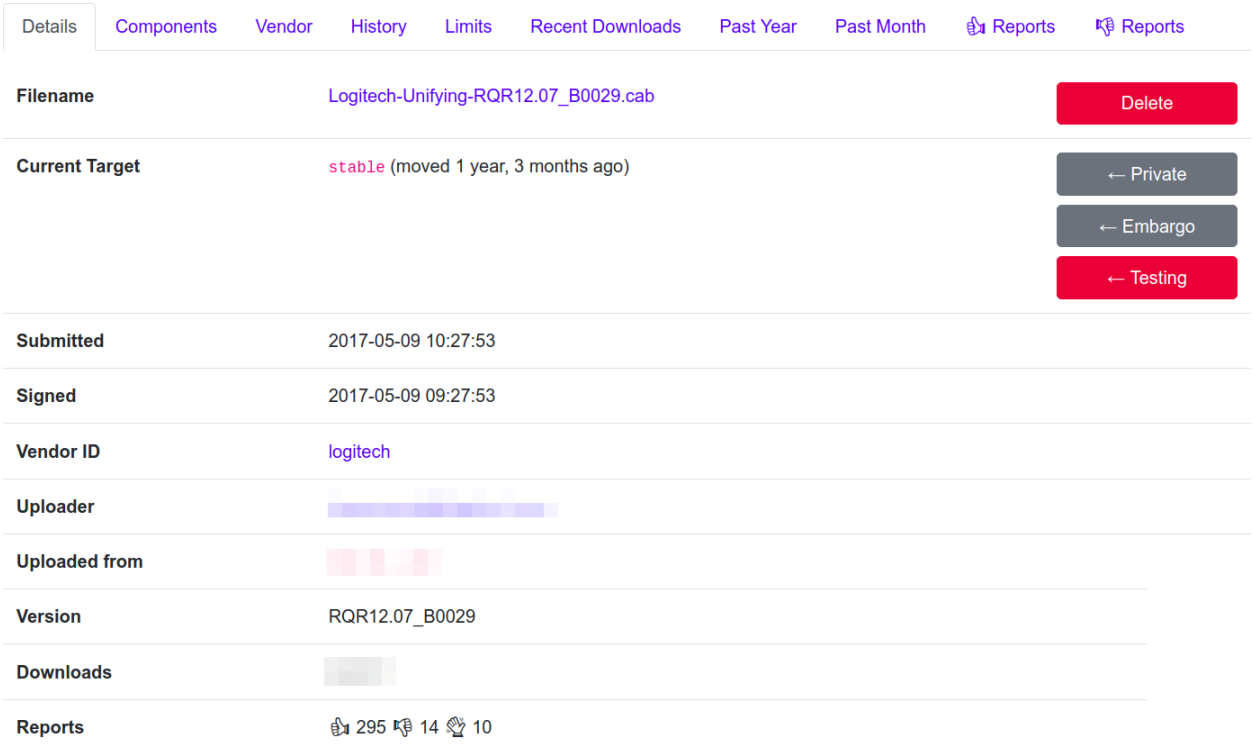


Fig. 1: Firmware view showing the report

Which expands out to the report below:

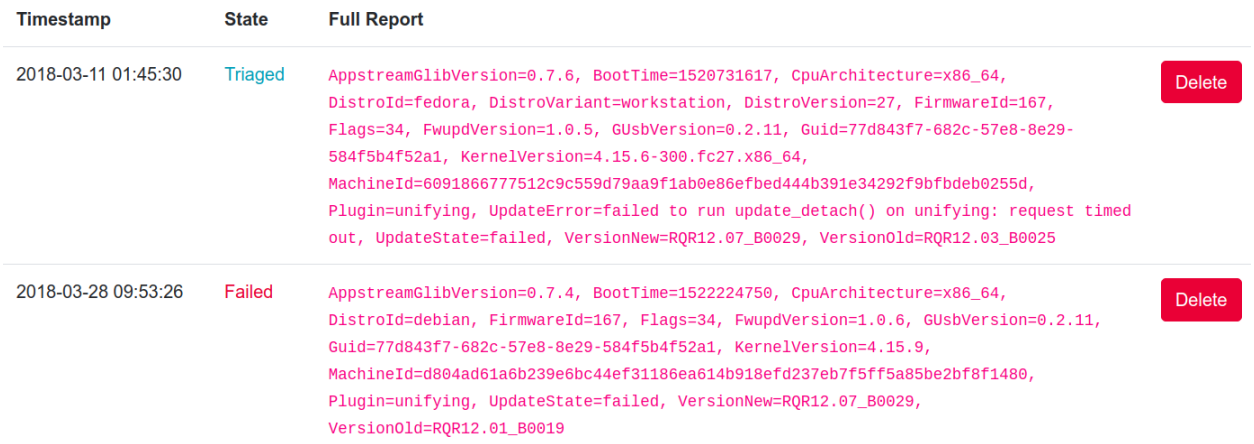


Fig. 2: Report details

This means vendors using the LVFS know the approximate number of successes and failures, and can add different tests to existing QA tests accordingly. This allows the LVFS to automatically pause the specific firmware deployment if > 1% of the reports come back with failures.

Some key points:

- We do not share the IP address with the vendor, and it is not even saved in the database
- The `MachineId` is a salted hash of the machine `/etc/machine-id`

- The LVFS does not store reports for firmware that it did not sign itself, i.e. locally built firmware archives will be ignored and not logged

The user can disable the reporting functionality in all applications by editing `/etc/fwupd/remotes.d/*.conf`

7.1 Vendor Summary

Using firmware telemetry overview a vendor can see all the success and failure reports for all the firmware uploaded to their vendor:

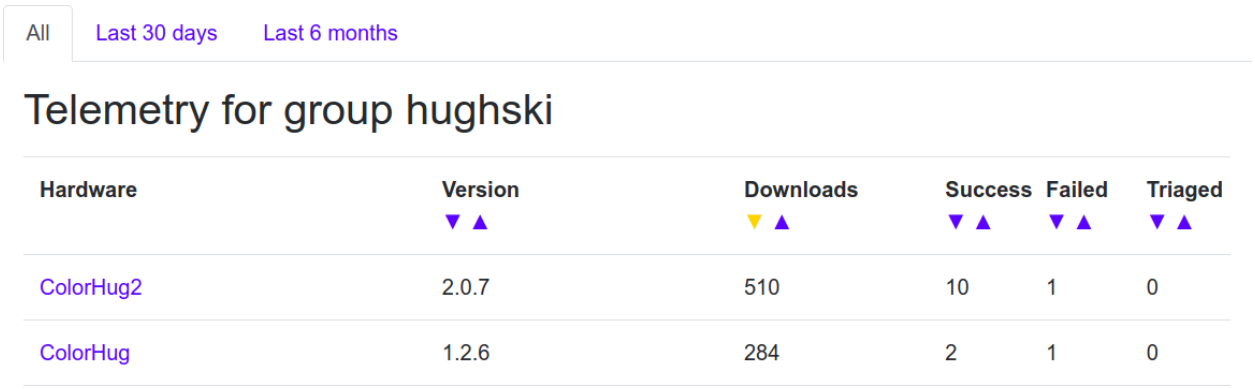


Fig. 3: Telemetry of all firmware

Until more people are running the latest fwupd and volunteering to share their update history it is less useful, but still interesting until then.

7.2 Known Issues

Known issues are problems we know about, and that can be triaged automatically on the LVFS. Of course, firmware updates should not ever fail, but in the real world they do, Of all the failures logged on the LVFS, 95% fall into about 3 or 4 different failure causes, and if we know hundreds of people are hitting an issue we already understand we can provide them with some help.

A good example here is the user not being on AC power when rebooting, which causes a failure, albeit transient and non-fatal. Another example is if the user tries to do the update with an incorrect system configuration, for instance a missing `/boot/efi` partition.

```
Proceed with upload? [Y|n]: y
Update failure is a known issue, visit this URL for more information: https://github.com/hughsie/fwupd/wiki/Common-Problems
[hughsie@localhost build (master %)]$
```

Fig. 4: Notifying the user about known issues

The URL for the user to click on is the result of a rule engine being included in the LVFS. Users on the LVFS with the appropriate permissions can also create and view rules for firmware owned by just their vendor group:

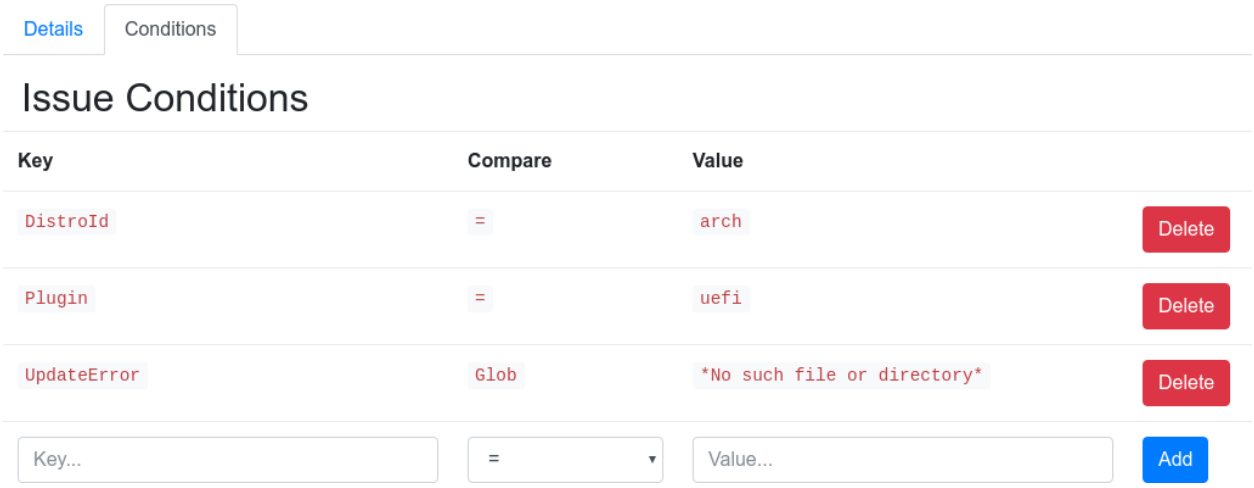


Fig. 5: Issue conditions

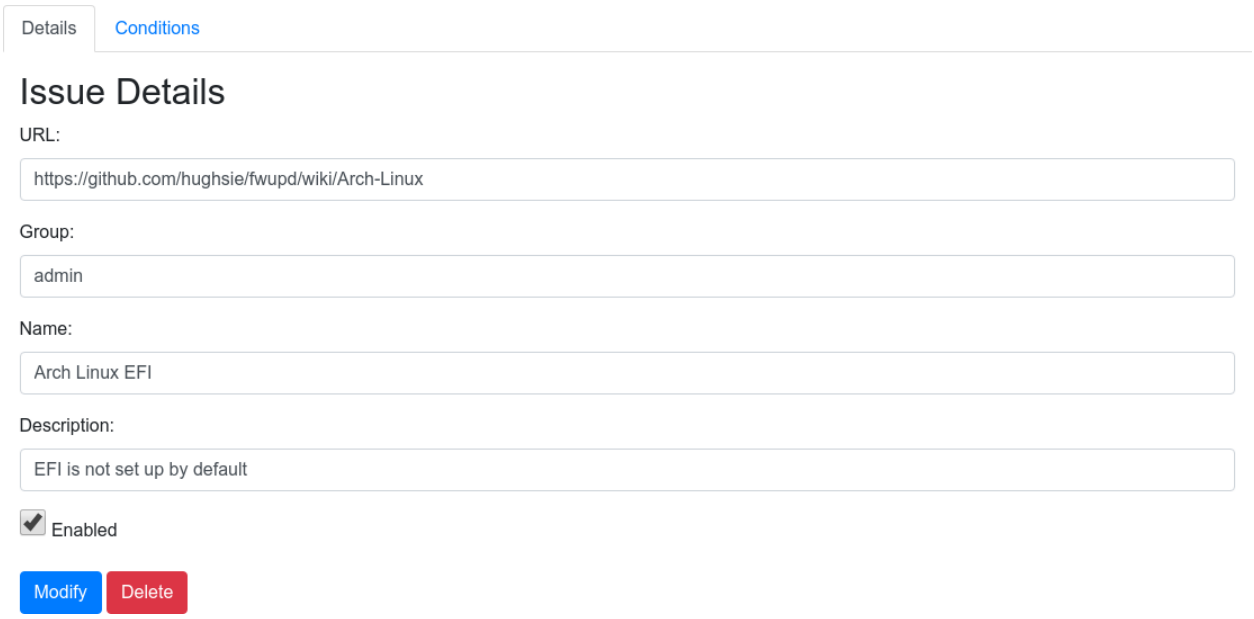


Fig. 6: Issue details

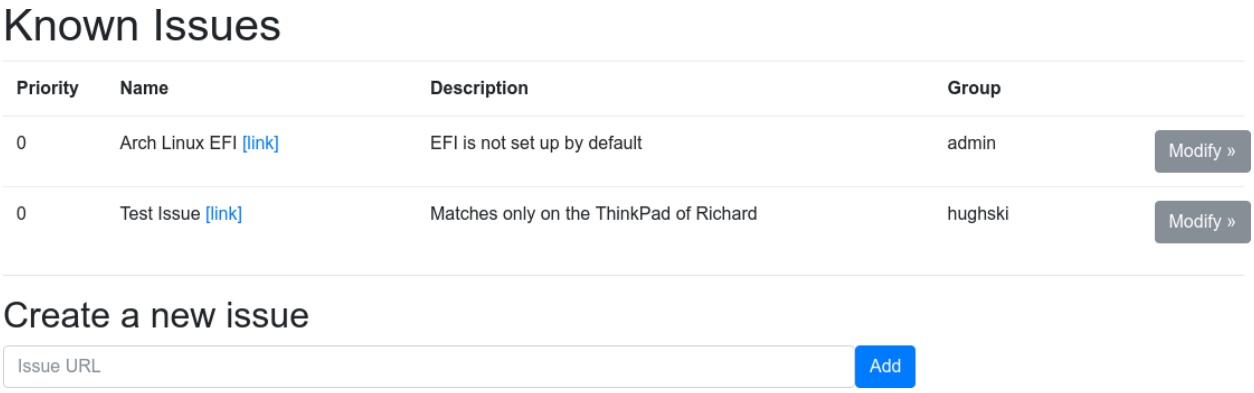


Fig. 7: All issues

CHAPTER

EIGHT

CUSTOM PROTOCOL

The fwupd project already supports a huge number of flashing protocols, everything from standardized protocols like NMVe, ATA, DFU and also a large number of *vendor-specfic* protocols like `logitech_hidpp`, `synaptics_prometheus` and `wacom_raw`.

Most vendors are using a protocol that fwupd already supports, and thus only need to upload firmware to the LVFS. In the case applying for an account is all that is required.

Note: If using DFU, please also implement the DFU runtime interface – this allows fwupd to automatically switch your device into bootloader mode without having to draw some artwork and write some translated text to explain how the user should persuade the device to enter update mode.

The easiest time to add support for updating hardware using the LVFS is during the project prototype phase. There are several things you can do that makes writing a fwupd plugin much easier.

In the case where the device protocol is a non-compatible variant or a completely custom protocol then a new fwupd plugin will be required. If you have to use a custom protocol, there are a few things that are important to consider.

The fwupd daemon needs to be able to enumerate the device without the user noticing, which means LEDs should not blink or cause the screen to flicker. Disconnecting a kernel driver, changing to bootloader mode or any other method of getting the device firmware version is not acceptable. This means the device needs to expose the current firmware version on the runtime interface, for instance using USB descriptors or PCI revision fields.

For composite devices (e.g. docks) it is much better to provide an interface to query the internal device topology rather than hardcoding it in the plugin or in a quirk file. For instance, fwupd could query the root device that would respond that it is acting as a I^2C bridge to a HDMI chip with address `0xBE`, rather than hardcoding it for a specific dock model. Querying the information allows the plugin author to write a generic plugin that means future devices can be upgraded without waiting for new fwupd versions to be included in popular Linux distributions and ChromeOS.

Warning: Plan and test for what happens when the user disconnects the USB cable, runs out of battery, or removes the mains plug when the new firmware is being flashed.

If the device remains in bootloader mode, is there a unique VID/PID that can be used to choose the correct firmware file to flash the device back to a functional runtime mode?

Many vendors just use the ISV-provided reference bootloader (which is fine), but fwupd does not know which runtime image to recover with if the ISV-allocated *generic* VID/PIDs are being used. If it is not possible to modify the bootloader VID/PID, then it *may* be possible to read a block of NVRAM at a hardcoded offset to identify the proper firmware to install.

When updating hardware it is important to provide feedback to the user so that they know the process has not *hung*. Updating firmware is intimidating to many users and so it is important to provide information about what is being done to the hardware, for instance erasing, writing and verifying. It is also a very good idea to provide percentage completion,

so for an operation that is going to take 10 seconds it is better to write 1024 blocks of 16kB with percentage updates after each block rather than one block of 16Mb with just a *bouncing* progressbar.

Note: It is not possible to upload executable *flasher* code as part of the cabinet archive – only the payload is allowed.

We will not accecpt non-free executables, static libraries or "shim" layers in fwupd. The only way a custom protocol can be supported is by contributing a LGPL-2.1-or-later plugin upstream.

Some vendors will have the experience to build a plugin themselves, and some vendors may wish to use a consulting company that has the required experience.

8.1 Intellectual Property Concerns

The plugin code in fwupd is Open Source (LGPL-2.1-or-later), licensed in a way that makes it possible for another vendor or an end-user to read and modify the code. Some companies have initially said that an open source plugin would be impossible due to concerns about either trade secrets, security or both. Let's look at the trade secret or intellectual property concern by answering some questions:

- How many firmware updater binaries were sold last year?
- Is the update protocol significantly more complicated than *read version number*, *switch to bootloader*, *erase blocks*, *write blocks*, *read back blocks to verify*, *switch to runtime*?
- Can a user dump the USB communication using a $20 capture tool and replay the recording to update a different device?
- Is the shared secret token sent unencrypted as part of the update protocol?

The fact is that most OEM vendors sell physical *devices* to consumers and both the hosting of updates on a company webserver and the development of the update client itself is typically a cost-centre, and not a revenue stream. The fwupd project supports more than 80 different update protocols, and most of them use exactly the same design; there may be differences in required header format or *CRC32 polynomials*, but 95% of "secret vendor protocols" are almost exactly the same from a high-level design perspective – and thus do not constitute valuable intellectual property.

Another important consideration is that the fwupd plugin **doesn't need to know everything** about the hardware – for instance, in the Synaptics MST plugin we know the offset in the configuration header of the current firmware version, but the rest of the header is unspecified and still secret. In the PixArt plugin we know the offset of the 32 bit `AA.BB.CC.DD` version number, but the rest of the file is treated as a binary blob that is just sent to the device in small sections.

Another concern from vendors is that as an open source project, anyone can edit, modify, or even sabotage code that communicates with their device. Of course, the maintainers of the fwupd project will review and check carefully every proposed change. Most plugins also have tests that emulate that specific device firmware update – that get verified for each and every proposed change. Any plugins that require stronger "ownership" requirements can also add an *Owners* section in the per-plugin `README.md` file that will be used to notify responsible users that validation or functional re-testing is required before a change is merged.

Some vendors are also unwilling to agree to an open source plugin to fwupd as the payload will be discovered as unsigned or communication is unencrypted, and attackers will know how to attack the devices. Unfortunately, it's very easy to scan a firmware payload for signatures or even to calculate the entropy. It's even possible to perform a manual bitswap in something like the USB descriptor to know if the payload signature is being verified correctly. Security through obscurity has never been acceptable, and hackers are more than capable of dumping an unencrypted firmware update process and then modifying the code to inject malware. An attacker does not care about having source code through legitimate means and does not need permission.

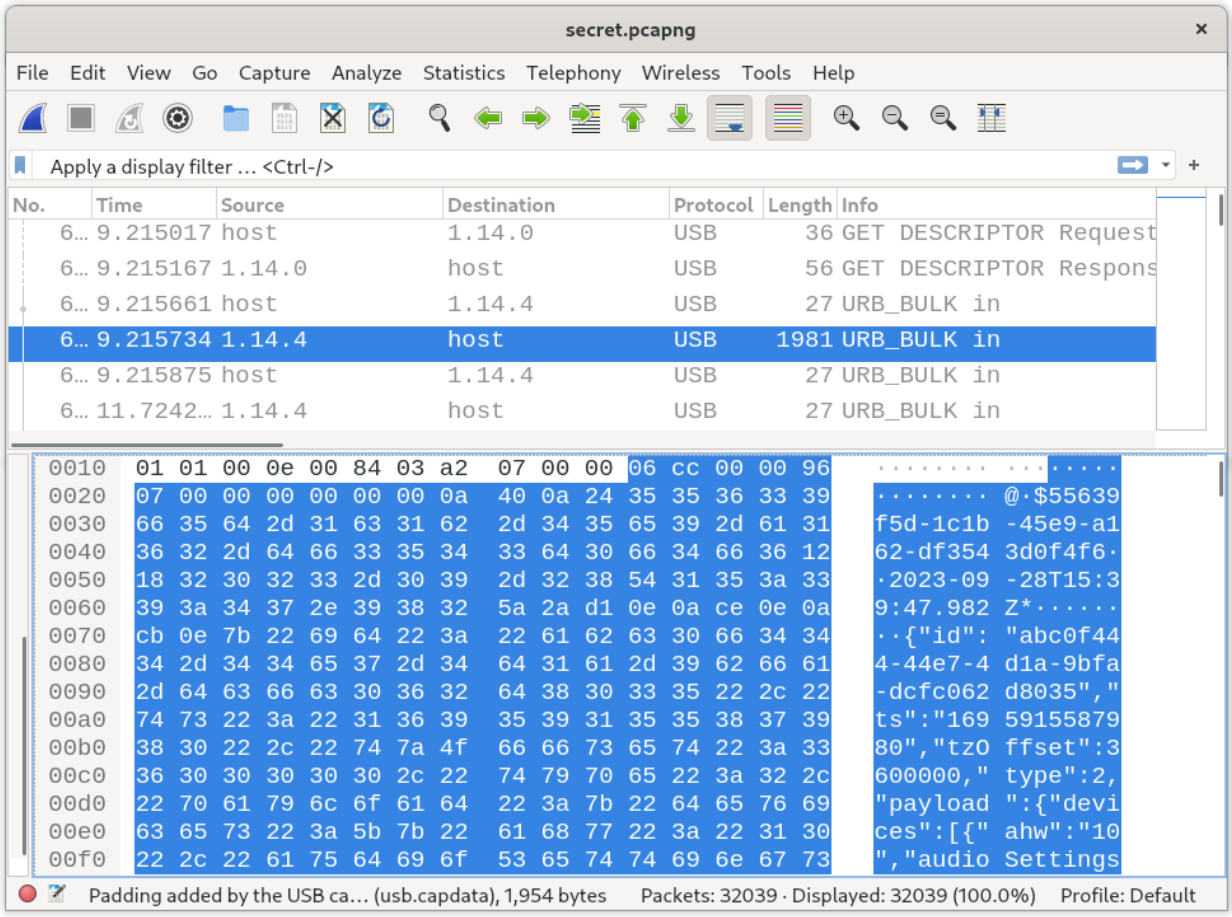


Fig. 1: Wireshark dump showing unencrypted data transfer

The answer is either to properly implement firmware signing, or to just be okay that the current device has unsigned firmware. It is completely acceptable for devices to not implement firmware signing for example with OpenHardware devices, or devices where the user is encouraged to build custom firmware. The LVFS only allows vendors to update their own hardware (e.g. Wacom can only update devices with USB vendor ID of `0x056A`) and so it is not possible for another vendor to automatically update firmware on your hardware, even if the payload is unencrypted or unsigned.

On a similar note, some vendors will not want to open the update protocol as it would allow consumers to *flash* the "pro" version of a firmware to the "basic" device to software-unlock features that are usually only available in a more expensive model. Whilst this is a concern if the two devices have the same signing key (they should have different keys) or if they have no signing requirement at all – it's very easy to fool even official non-free binary updater programs just by changing how the Windows/Linux kernel reports either the device USB VID/PID or USB HID descriptor. From real world experience, the number of users that are going to disable the device checks in an open source project (a risky procedure) is going to be a staggeringly small number of people compared to the number of people that will be able to update the correct hardware with the latest firmware.

Turning this around completely, many vendors who have added new fwupd plugins have found that the RMA (return merchandise authorization) rate for hardware has actually **reduced significantly**. This is because the newer firmware fixes a bug, for instance a USB-4 dock *not working* with their existing DisplayPort monitor – where it does work with the latest dock firmware applied. Similarly, users may choose the consumer device or peripheral exactly **because** it has a fwupd support, and because firmware updates can be managed at scale using Windows, Linux, macOS or ChromeOS.

8.2 Depending on a new library

Please do not use model-specific or vendor-specific libraries to update or enumerate the hardware. Unless the library is already shipped by default on Ubuntu LTS and RHEL 8 then it is going to be exceptionally hard to use this library in fwupd. As fwupd is in *main* for Ubuntu then any library it depends on must **also** be part of main, which means Canonical has to officially support it. They obviously do not want to do this for vendor-specific libraries that have only existed for a few years with no API or ABI guarantees or long term stable branches.

Similarly for Red Hat; any new library or binary needs to have a Red Hat maintainer who will support it for over 10 years (!) and is willing to do the security due diligence and code review required to be included as a core package in RHEL. Getting approval for a new package is a **huge** amount of work and takes months.

As fwupd is running as root, any external library it depends on must be audited by several security teams, and have a proven security plan in place. Google also needs to review any new dependencies as fwupd is also being used heavily in ChromeOS now, and they take OS image size and security very seriously.

In this situation it is completely okay to include parts of the open source library (assuming the code can be licensed as LGPL-2.1-or-later) in the fwupd plugin. Including them in fwupd plugins also allows us to use the fwupd helper functionalily, for instance replacing `memcpy()` with `fu_memcpy_safe()` and using high level abstractions for reading and writing to `sysfs` files or an `ioctl`. Many plugins already do this, for instance the `colorhug` plugin does not use `libcolorhug`, `nvme` plugin does not use the `nvme` command line tool and the `emmc` plugin itself defines `EXT_CSD_FFU` rather than depending on `mmc-utils`.

8.3 Building fwupd

Please see the fwupd documentation.

CHAPTER

NINE

SECURITY

There are many layers of security in the LVFS and fwupd design, including restricted account modes, 2 factor authentication, and server side AppStream namespaces.

The most powerful one is the so-called `vendor-id` that the vendors cannot assign themselves, and is assigned by a member of the LVFS admin team when creating the vendor account on the LVFS. The way this works is that all firmware from the vendor is tagged with a *requirement* like `USB:0x056A` which matches the USB consortium vendor assigned ID.

Client side, the `vendor-id` from the signed metadata is checked against the physical device and the firmware is updated only if the ID matches. This ensures that malicious or careless users on the LVFS can never ship firmware updates for other vendors hardware. All vendors on the LVFS are now locked down with this mechanism.

Some vendors have to use IDs that they do not own, a good example here is for a DFU device like the 8BitDo controllers. In runtime mode they use the USB-assigned 8BitDo VID, but in bootloader mode they use a generic VID which is assigned to the chip supplier as they are using the reference bootloader. This is obviously fine, and both vendor IDs are assigned to 8BitDo on the LVFS for this reason.

Another example is where Lenovo is responsible for updating Lenovo-specific NVMe firmware, but where the NVMe vendor is not using the Lenovo PCI ID.

All devices exported by fwupd must have at least one vendor ID, mostly automatically added as the vast majority derive from either `FuUsbDevice` or `FuUdevDevice`.

The vendor IDs can be dispayed using `fwupdmgr get-devices`.

9.1 UEFI UpdateCapsule

Capsule updates are a popular way to distribute firmware updates. As the ESRT convays no vendor ownership information, we use the platform DMI data. For instance Lenovo is only able to update Lenovo hardware with `DMI:Lenovo`.

CHAPTER

TEN

PRIVACY REPORT

We hold personal data about vendors, administrators, clients and other individuals for a variety of purposes. This policy sets out how we seek to protect personal data and ensure that administrators understand the rules governing their use of personal data to which they have access in the course of their work. In particular, this policy requires that the Data Protection Officer (DPO) be consulted before any significant new data processing activity is initiated to ensure that relevant compliance steps are addressed.

10.1 Scope

This policy applies to all users who have access to any of the personally identifiable data.

10.2 Who is responsible for this policy?

As the Data Protection Officer, Richard Hughes. has overall responsibility for the day-to-day implementation of this policy. The DPO is registered with the Information Commissioner's Office (ICO) in the United Kingdom as a registered data controller.

10.3 Fair and lawful processing

We must process personal data fairly and lawfully in accordance with individuals' rights. This generally means that we should not process personal data unless the individual whose details we are processing has consented to this happening, or where such collection is unavoidable and/or considered pragmatic in the context, e.g. logging the number of downloads of a particular file.

We do not consider an IP address to represent a single user (due to NAT or VPN use), and as such metadata requests are not considered personal data using the draft GDPR guidelines.

10.4 Accuracy and relevance

We will ensure that any personal data we process is accurate, adequate, relevant and not excessive, given the purpose for which it was obtained. We will not process personal data obtained for one purpose for any unconnected purpose unless the individual concerned has agreed to this or would otherwise reasonably expect this. Individuals may ask that we correct inaccurate personal data relating to them. If you believe that information is inaccurate you should inform the DPO.

10.5 Your personal data

You must take reasonable steps to ensure that personal data we hold about hardware vendors is accurate and updated as required. For example, if your personal circumstances change, please update them using the profile pages or inform the Data Protection Officer.

10.6 Data security

We keep personal data secure against loss or misuse. Where other organizations process personal data as a service on our behalf, the DPO will establish what, if any, additional specific data security arrangements need to be implemented in contracts with those third party organizations.

10.6.1 Storing data securely

All data is stored electronically. All documents and code are held on a locked LUKS partition with a password adhering to security best practices.

10.6.2 Data retention

We must retain personal data for no longer than is necessary. What is necessary will depend on the circumstances of each case, taking into account the reasons that the personal data was obtained, but should be determined in a manner consistent with our data retention guidelines. Anonymized user data (e.g. metadata requests) will be kept for a maximum of 5 years which allows us to project future service requirements and provide usage graphs to the vendor.

10.6.3 Transferring data internationally

There are restrictions on international transfers of personal data. We do not transfer personal data anywhere outside the EU without the approval of the Data Protection Officer, unless required to do so by law.

10.7 Subject Access Requests

Please note that under the Data Protection Act 1998, individuals are entitled, subject to certain exceptions, to request access to information held about them.

On receiving a subject access request, we will refer that request immediately to the DPO. We may ask you to help us comply with those requests. Please also contact the Data Protection Officer if you would like to correct or request information that we hold about you. There are also restrictions on the information to which you are entitled under applicable law.

10.8 Processing data

We will never use identifiable vendor data for direct marketing purposes.

10.9 GDPR Provisions

Where not specified previously in this policy, the following provisions will be in effect on or before 11 November 2020.

10.10 Transparency of data protection

Being transparent and providing accessible information to individuals about how we will use their personal data is important for our project. The following are details on how we collect data and what we will do with it:

10.10.1 Firmware Vendor Information

- **What:** The hardware vendor name, password, GPG public key and content of original uploaded firmware files.
- **Why collected:** Secure authentication, to allow any possible future audit and to provide authorized users access to signed firmware files.
- **Where stored:** AWS hosted PostgreSQL database in Oregon, USA region.
- **When copied:** Full backups weekly, with daily snapshots both to AWS backup.
- **Who has access:** The hardware vendor (filtered by the QA group), Linux Foundation Infrastructure Team and the DPO.
- **Wiped:** When the vendor requests deletion of the user account.

10.10.2 Service Event Log

- **What:** IP address (unhashed) and REST method requested, along with any error.
- **Why collected:** Providing an event log for checking what the various hardware vendors are doing, or trying to do.
- **Where stored:** AWS hosted PostgreSQL database in Oregon, USA region.
- **When copied:** Full backups weekly, with daily snapshots both to AWS backup.
- **Who has access:** The hardware vendor (filtered by the QA group), Linux Foundation Infrastructure Team and the DPO.

- **Wiped:** When the QA group is deleted.

10.10.3 Firmware Download Log

- **What:** IP address, timestamp, filename of firmware, user-agent of client.
- **Why collected:** To know what client versions are being used for download, and to provide a download count over time for a specific firmware file.
- **Where stored:** AWS hosted PostgreSQL database in Oregon, USA region.
- **When copied:** Full backups weekly, with daily snapshots both to AWS backup.
- **Who has access:** The hardware vendor (filtered by the QA group), Linux Foundation Infrastructure Team and the DPO.
- **Wiped:** When the firmware is deleted, but the client IP address is cleared after 3 years.

10.10.4 Firmware Reports

- **What:** Machine ID (hashed), failure string and checksum of failing file, OS distribution name and version.
- **Why collected:** Allows the hardware vendor to assess if the firmware update is working on real hardware.
- **Where stored:** AWS hosted PostgreSQL database in Oregon, USA region.
- **When copied:** Full backups weekly, with daily snapshots both to AWS backup.
- **Who has access:** The hardware vendor (filtered by the QA group), Linux Foundation Infrastructure Team and the DPO.
- **Wiped:** When the firmware is deleted.

We will ensure any use of personal data is justified using at least one of the conditions for processing and this had been specifically documented above.

10.11 Consent

The data that we collect is subject to active consent by the data subject. This consent can be revoked at any time. Revoking consent to use data ends any vendor relationship with the LVFS.

10.12 Data portability

Upon request, a data subject should have the right to receive a copy of their data in a structured format, typically an SQL export. These requests should be processed within one month, provided there is no undue burden and it does not compromise the privacy of other individuals. A data subject may also request that their data is transferred directly to another system. This is available for free.

10.13 Right to be forgotten

A vendor may request that any information held on them is deleted or removed, and any third parties who process or use that data must also comply with the request. An erasure request can only be refused if an exemption applies.

10.14 Privacy by design and default

Privacy by design is an approach to projects that promote privacy and data protection compliance from the start. The DPO will be responsible for conducting Privacy Impact Assessments and ensuring that all changes commence with a privacy plan. When relevant, and when it does not have a negative impact on the data subject, privacy settings will be set to the most private by default.

10.15 Data audit and register

Regular data audits to manage and mitigate risks will inform the data register. This contains information on what data is held, where it is stored, how it is used, who is responsible and any further regulations or retention timescales that may be relevant.

10.16 Reporting breaches

All users of the LVFS have an obligation to report actual or potential data protection compliance failures. This allows us to:

- Investigate the failure and take remedial steps if necessary
- Maintain a register of compliance failures
- Notify the Supervisory Authority (SA) of any compliance failures that are material either in their own right or as part of a pattern of failures

Please refer to the DPO for our reporting procedure.

10.17 Monitoring

Everyone who actively uses the LVFS must observe this policy. The DPO has overall responsibility for this policy. They will monitor it regularly to make sure it is being adhered to.

10.18 Consequences of Failing to Comply

We take compliance with this policy very seriously. Failure to comply puts both you and us at risk. The importance of this policy means that failure to comply with any requirement may lead to disciplinary action under our procedures. If you have any questions or concerns about anything in this policy, do not hesitate to contact the DPO.

CHAPTER

ELEVEN

OFFLINE FIRMWARE

The LVFS is a public webservice designed to allow OEMs and ODMs to upload firmware easily, and for it to be distributed securely to tens of millions of end users. For some people, this simply does not work for various reasons:

- They don't trust the LVFS team, fwupd.org, GPG, certain OEMs or the CDN we use
- They don't want thousands of computers on an internal network downloading all the files over and over again
- The internal secure network has no internet connectivity

For these cases there are a few different ways to keep your hardware updated, in order of simplicity:

11.1 Deploy in immutable image

If the OS is shipped as an image, you can just install the `.cab` files into `/usr/share/fwupd/remotes.d/vendor/firmware` and then enable `vendor-directory.conf` with `fwupdmgr enable-remote vendor-directory`.

Then once you have disabled the public LVFS using `fwupdmgr disable-remote lvfs`, running `fwupdmgr` will use only the cabinet archives you deploy in your immutable image. Of course, you're deploying a larger image because you might have several unused firmware files included for each image, but this is how Google Chrome OS is using fwupd.

11.2 Mirror the public firmware

11.2.1 Using pulp-server

You can use Pulp to mirror the entire *public* contents of the LVFS (but never private or embargoed firmware). Create a repo pointing to PULP_MANIFEST and then sync that on a regular basis to download the metadata and firmware. The contents will not change any more frequently than every 4 hours, so please use a polling interval of at least that.

11.2.2 Using a helper script

There is a helper script sync-pulp.py that can be used if `pulp-server` is not installed:

```
./contrib/sync-pulp.py https://cdn.fwupd.org/downloads /mnt/mirror
```

You can then use a webserver such as Nginx or Apache to export `/mnt/mirror` as `https://my.private.server/mirror`.

Then, disable the LVFS by deleting or modifying `/etc/fwupd/remotes.d/lvfs.conf` and then create a `/etc/fwupd/remotes.d/myprivateserver.conf` file:

```
[fwupd Remote]
Enabled=true
Type=download
MetadataURI=https://my.private.server/mirror/firmware.xml.gz
FirmwareBaseURI=https://my.private.server/mirror
```

To instead mirror the private embargoed files, you can use:

```
./contrib/sync-pulp.py https://cdn.fwupd.org/downloads /mnt/mirror \
    --username=login@name.com \
    --token=XA1A5ZV7R65FUZBZ
```

Warning: **Do not use your login password here!** Generate a token when logged in to the LVFS using the User Profile settings.

To restrict the downloaded firmware to a specific tag (perhaps a vendor *Best Known Configuration*) use the following command:

```
./contrib/sync-pulp.py https://cdn.fwupd.org/downloads /mnt/mirror \
    --filter-tag=hughski-2020q1
```

Note, this command can be run safely multiple times with different `--filter-tag` values on the same destination directory; the superset of files will be downloaded.

11.2.3 Approved firmware

By exporting the entire LVFS (including the metadata, *and* metadata signature) you can still delay the deployment of firmware. Using the approved firmware list the client can filter out firmware that has not been tested by your organization without creating and signing a custom remote.

The allow-list of firmware can be set using:

```
fwupdmgr set-approved-firmware checksum1,checksum2,checksum3
```

Some versions of fwupd (>= 1.7.1) also support loading the list of checksums using a filename, e.g.

```
fwupdmgr set-approved-firmware filename
```

…where `checksum` is the SHA1 or SHA256 checksum of the `.cab` archive.

Note: The `org.freedesktop.fwupd.set-approved-firmware` PolicyKit action may require root permission to use.

11.3 Export a shared directory

Again, use `PULP_MANIFEST` to create a big directory holding all the firmware (currently ~50GB, but growing), and keep it synced.

Create a NFS or Samba share and export it to clients. Map the folder on each client, and then create a `myprivateshare.conf` file in `/etc/fwupd/remotes.d`:

```
[fwupd Remote]
Enabled=false
Title=Vendor
Keyring=none
MetadataURI=file:///mnt/myprivateshare/fwupd/remotes.d/firmware.xml.gz
FirmwareBaseURI=file:///mnt/myprivateshare/fwupd/remotes.d
```

11.4 Downloading manually

Download the `.cab` files that match your hardware and then install them on the target hardware via Ansible or Puppet using `fwupdmgr install foo.cab`. You can also use `fwupdagent get-devices` to get the existing firmware versions of all hardware in a format you can parse from scripts.

11.4.1 Building a custom remote

The local.py script allows you to create the metadata for a directory of `.cab` files.

Note: If you want to use signed metadata then please use `jcat-tool firmware.xml.gz.jcat firmware.xml.gz CERTIFICATE PRIVATE_KEY`. You will need to create a custom certificate and you'll also need to distribute the the PKCS#7 certificate on all clients that are going to use the remote.

11.5 Create your own LVFS

The LVFS is a free software Python 3 Flask application and an instance can be set up internally if required. You have to configure much more this way, including generating your own GPG and PKCS#7 keys, uploading your own firmware and setting up users and groups on the server.

Doing all this has a few advantages, namely:

- You can upload each firmware file and QA it, only pushing it to stable when ready
- You don't ship firmware which you didn't upload
- You can control the staged deployment, e.g. only allowing the same update to be deployed to 1000 servers per day
- You can see failure reports from clients, to verify if the deployment is going well
- You can see nice graphs about how many updates are being deployed across your organization

However, running a secure LVFS instance is a lot of work as PostgreSQL has to be used as a database, Redis has to also be set up as a queue manager, and Celery is used to manage the worker queues.

Although minor versions of the LVFS can be upgraded easily, you should review all the commits to lvfs-website to ensure that any manual migration is also performed.

CHAPTER

TWELVE

PRODUCT CERTIFICATION

12.1 Introduction

We want to make it easy for ODMs and OEMs to choose components that already have fwupd plugin support. This will do a few things:

- The onus is pushed onto the IHV to maintain the plugin not the OEM, ODM or Linux distributor (e.g. Red Hat)
- The ODM and OEM will prefer components that do not require any software development work to pass the Works With ChromeBook (WWCB) and Red Hat Enterprise Linux (RHEL) hardware certifications
- Having a fwupd plugin will be seen as a commercial advantage for the IHV

There are two versions of the *fwupd friendly firmware* certification, one for devices that will only accept signed firmware (`signed-payload`) and another for insecure hardware that does not implement cryptographic signing (`unsigned-payload`). Either is fine from a fwupd plugin point-of-view, but some OEMs will have a policy that forces them to choose hardware that cannot be altered by the end user.

Note: Consumer devices end-users buy from the store are not suitable for *fwupd friendly firmware*, and already have device pages on the LVFS.

12.2 Requirements

To register a device for *fwupd friendly firmware* the original silicon vendor must have an existing LVFS vendor account, and also provide:

- Device model, e.g. `CX2098X`
- One line summary, e.g. `USB3.2 Gen1 4-port hub controller`
- Link to the device page, e.g. `https://www.kinet-ic.com/ktm50x0/` (optional)
- Link to the upstream fwupd plugin that handles this device type, e.g. `https://github.com/fwupd/fwupd/tree/main/plugins/synaptics-cxaudio`
- Device firmware certification level, e.g. *always signed*, *optionally signed*, or *unsigned*
- Hash and signature algorithm used for firmware signing, e.g. `SHA256+RSA2048`, or `n/a`

To add a device to the certification page, please send an email to the mailing list with the required details. On meeting the requirements, the entry will be added and then the vendor is allowed use the signed or unsigned *fwupd friendly firmware* logo as required.

Fig. 1: logo for *fwupd friendly firmware* (always signed)

Fig. 2: logo for *fwupd friendly firmware* (optionally signed)

Note: Vendors should not have a "generic" *fwupd friendly firmware* assigned to them, as a vendor may have multiple devices with different update protocols. e.g. Synaptics has `cxaudio`, `mst`, `prometheus` and `cape` protocols, each with a different fwupd plugin.

Fig. 3: logo for *fwupd friendly firmware* (unsigned)

12.3 Conclusion

Vendors using the *fwupd friendly firmware* logo mark will make it easier for product creators to support firmware updates for Linux users. The burden of development moves earlier to the IHVs rather than later to the OEMs. OEMs can verify the *fwupd friendly firmware* certification and compare hardware using the public pages on the LVFS.

CHAPTER

THIRTEEN

LVFS RELEASES

13.1 1.5.2 (2024-05-07

This release adds the following features:

- Add an API endpoint to get firmware status
- Add documentation for firmware testing using Moblab and ChromeOS
- Add support for mirroring PULP remotes
- Add support for multiple project licenses
- Add support for not_hardware requirements
- Add support for SHA-384
- Add support for zstd metadata
- Add the firmware SBoM specification
- Allow adding positive vendor relationships in the vendor list
- Allow exporting SWID and SPDX from the SBoM helper
- Allow QA users to list and delete thier own signed reports
- Allow SPDX license aliases
- Allow uploading offline reports
- Include <developer_name> in the archive
- Put the component install duration in the metadata if provided
- Require vendors to set the Username and Password when downloading embargoed metadata
- Save the BT logindex in the JCat file
- Show anonymous success reports in the device page
- Show if the release has verified reports on the OEM device page
- Show the release gating on the device page
- Show which problems block what remote
- Sign the SHA256 hash as well as the payload
- Support per-release priorities

This release removes the following features:

- Do not allow adding duplicate requirements
- Do not allow over-long sumary text
- Do not allow the GUID tests to be waived
- Do not offer updates to very old fwupd versions
- Enforce that UEFI devices have the vendor-id set in the embargo metadata
- Remove support for PSPTool
- Remove the 'unrestricted' vendor feature

This release fixes the following bugs:

- Add 6 bytes of random data to the JcatFile to fix a CDN issue
- Add an fsck action for the component download size
- Allow 100% generic components as we're using them for metadata
- Allow accepting reports for DoNotTrack firmware
- Allow cancelling lvfs.reports.utils.task_regenerate
- Allow more revisions in the gz and xz embargo remotes
- Allow searching by GUID in the public search
- Allow showing firmware supporting a specific protocol
- Allow the admin to unwaive a test
- Auto-demote firmwares uploaded to embargo with problems
- Automatically add a fwupd requirement when adding a CHID
- Catch a DER decode error when looking for certs
- Check the useragent before the ClientAcl
- Correctly add device checksums on upload
- Do not crash when searching for a device with NUL chars
- Do not fail to sign firmware if services.nvd.nist.gov is unavailable
- Do not purge deleted immutable firmware
- Do not show a 'Move here' button when embargo rebuilt would fail
- Do not use absolute URIs in the xz metadata
- Do the abuse check before the GeoIP lookup
- Fix a backtrace when clamav isn't installed
- Fix a worker crash when claims are not present
- Fix certificates with UTF-16 RFC-2459 descriptions
- Fix invalid metadata licenses
- Fix the task worker when trying to parse invalid NIST NVD payload data
- Handle the VINCE server exploding
- Include a fwupd requirement for a CHID requirement
- Lower the number of concurrent db connections

- Migrate from Owl to Swiper for quotes
- Never add duplicate content to PULP_MANIFEST
- Never use the CDN for firmware images
- Only add AppStream prefixes when required
- Port the NIST NVD plugin to new API
- Ramp up the warning about pushing updates without test reports
- Reduce the scope of the CSP
- Regenerate the metadata less frequently
- Rename com.intel.Uefi to org.uefi
- Re-run the tests when changing the AppStream ID
- Run the local development instance with SSL
- Sort the devices by version in the device pages
- Update the firmware failure count when adding known issues
- Update the firmware report count when deleting a report
- Use the CDN for many more public files

13.2 1.5.1 (2023-05-05

This release adds the following features:

- Add a nudge to people using obsolete fwupd versions to upgrade
- Add `--cleanup` to `sync-pulp.py` to remove old archives
- Add documentation for testing on ChromeOS
- Add HSI attribute downloads in JSON format
- Add the ability to block abusive clients by IP address
- Allow adding per-protocol device flag values
- Allow downloading firmware and uploading reports with basic auth
- Allow uploading firmware assets like emulation data
- Block clients automatically when abuse is detected
- Require a `User-Agent` header to serve archives
- Set the `FromOEM` report key in the metadata
- Show category icons on the device list page
- Show if a vendor has a PSIRT team and show the link in more places

This release removes the following features:

- Remove firmware limits feature as it was unused and complicated the code
- Remove inf parsing as it is no longer required and was a footgun for vendors

This release fixes the following bugs:

- Actually save a `firmware.xml.xz` newest file to make debugging easier
- Add more banned things to the name checks
- Add `ne` to the simple component requirements page
- Allow users to export the SWID data for public firmware
- Block the generic useragent of `Mozilla/5.0`
- Convert icon battery into category `X-Battery`
- Do GeoIP lookups on IPv6 data too
- Do not store broken report attributes
- Ensure that the newest metadata files are invalidated
- Fix a crash when using `%00` in URLs
- Never include empty <`client`/> requirements
- Only dedupe the requirements exactly
- Resign any files with artifact type=binary
- Show a link to the firmware when uploading a duplicate
- Store all firmware container checksums
- Update requirements to fix security bugs in dependencies
- Use newline for multiline settings values
- Use the display version to sort components in the search results

13.3 1.5.0 (2023-01-03)

This release adds the following features:

- Add a fsck action for the VersionFormat, release_tag, shard info GUIDs and key checksum
- Add a user-visible claim for a detected SBoM
- Add a waiveable test failure on system integrity report failure
- Add BootGuard shards when extracting UEFI firmware
- Add interesting public test failures to the mdsync export
- Add new update categories like X-UsbDock and X-UsbReceiver
- Allow filtering by tag when using sync-pulp.py
- Allow setting the update message and image per-protocol
- Cache the public pages to reduce load
- Do not allow all protocols to use the X-Device category
- Enforce no duplicate objects in the db layer
- Enforce that release timestamp is not >2 years in the future or past
- Generate additional xz metadata for a 25% size saving
- Generate the PULP_MANIFEST at remote regeneration time

- Only allow custom update messages for specific protocols
- Remove the client useragent and country code after 3 years
- Replace Celery with a built-in task scheduler and remove the beat ECS service
- Replace uefi_r2 with fwhunt_scan to support new rules format
- Show the HSI number in more places
- Use a Flask application factory pattern
- Verify the upload was written on EFS
- Warn the QA user when promoting a firmware with no success reports

This release fixes the following bugs:

- Add a more indexes to speed up database access
- Allow choosing non-public protocols in the component view
- Allow running local.py without a database set up
- Allow the QA user to modify the component release date
- Always ignore the first section in the reverse-DNS validation
- Bind to all IPv4 and IPv6 interfaces
- Build the docker container on CentOS 9 Stream
- Check that previous CHIDs are always included in new firmware
- Dedeupe URIs when sending report response
- Detect and fix duplicate users
- Do not assume every AppStream ID with 4 dashes is a GUID
- Do not clear the waived timestamp when retrying a test
- Do not disable 2FA when changing the users password
- Do not export the metadata_license in the AppStream metadata
- Do not fail the UEFI capsule test when using a valid FMP GUID
- Do not garbage-collect old revisions when the latest revision is new
- Do not include empty <device> tags in the metadata
- Do not include the component description in the AppStream metadata
- Do not mark the OTP textbox as 'password'
- Do not require admin login to download a known shard
- Do not show problems in the search view to fix performance issues
- Do not store the firmware or remote dirty state and use runtime state instead
- Do not use a HTML 404 page when downloading from a client
- Do not use ; to split URIs, it's a valid char in RFC3986
- Fix a crash when parsing very old HSI reports
- Fix a warning when a PE file has no authenticode signature
- Fix the displayed URLs and display name in the LVFS emails

- Include the vendor name in the mdsync output
- Increase the pulp download timeout to 60s
- Make all the icons symbolic to match gnome-firmware
- Make the eventlog address field more than 40 chars
- Make the HSI aggregated data public
- Move firmware promote and nuke to an async action
- Only add <testing> elements when using artifacts
- Only include the sizes for the artifact
- Prevent duplicate usernames
- Prevent the human user from being the same as the username
- Relax the backdated checks to include older firmware
- Remove all users of _error_internal()
- Remove some unused database columns and obsolete migration scripts
- Remove the hardcoded and duplicated release description text
- Remove the IPFS functionality as it was almost completely unused
- Remove the per-vendor event-log page
- Remove the tests overview page, as this does not scale
- Show a warning when doing an async promotion
- Show the OEM firmware in 'State :: Embargo' for ODMs
- Show when a user waived the test in the UI
- Speed up downloading cab archives and most page loads
- Update uSWID to fix reading and writing compressed payloads
- Use less whitespace in the AppStream metadata file
- Use the checksum as the shard absolute path as the name is not always unique
- Use the correct artifact type for metainfo.xml files
- Use the correct status code for mdsync export
- Use the flask debug toolbar when running locally

13.4 1.4.0 (2022-05-24)

This release adds the following features:

- Add a progress indicator to the Yara scan
- Add 'fwupd friendly firmware' certification
- Add information about what models are EOL
- Add new categories of X-Mouse and X-BaseboardManagementController
- Add support for asynchronous uploads

- Add support for external uSWID+CoSWID sections
- Add the concept of vendor subgroups
- Add device icons of usb-hub and usb-receiver
- Add XLIFF v2 import and export for translation
- Allow auto-moving firmware on defined dates
- Allow creating a GUID from an instance ID
- Allow creating a uSWID blob from form data
- Allow firmware to have multiple ODMs
- Allow importing, exporting and modifying localized update release notes
- Allow marking firmware revisions as immutable
- Allow updates to specify a level of device integrity
- Allow uploading firmware using a username and token
- Analyze Intel microcode versions
- Build metadata into a firmware transparency log
- Export the LVFS component ID into the AppStream metadata
- Get the CVE descriptions description from VINCE and NIST NVD
- Show the metadata upload failures in the UI
- Use name_variant_suffix in the public metadata
- Use signed reports for firmware QA
- Use the CDN to distribute firmware

This release fixes the following bugs:

- Add client requirements to the metadata
- Add more JCat blob kinds
- Allow modifications in the testing target
- Allow OAuth users to modify subgroup and notification settings
- Allow QA users to delete limits
- Allow security researchers to run UEFI R2 scripts
- Allow specifying file:// images that are copied from the archive
- Allow users to share the [possibly private] signed report data
- Check for the duplicate remote before checking problems
- Detect more vendors pasting in Intel SA issues
- Do no merge component with different self requirements
- Do not allow an unsigned report to adjust the output of a signed one
- Do not allow some name_variant_suffix content
- Do not backtrace when trying to compare UTF-8 and UTF-16 text
- Do not export optional component data XML

- Do not force 'number' verfmts to hex in the metainfo
- Do not show test passes in uefi_scanner
- Do not split search terms on the hyphen
- Do not use Google Fonts
- Fix a crash when a component description was not set
- Fix crash when old stable firmware has no update description
- Fix runtime exception when checking inactive users
- Ignore markdown elements with control chars
- Make autoimporting issues CSRF-safe
- Make Claim.allow_embargo per-instance, not per-class
- Make the license have an optional clickable URL
- Make the recovery email case insensitive
- Make the update useful word requirement lower
- Move some upload issues to runtime component problems
- Never include ampersands in the revision filename
- Never try to escape missing paragraph text
- No longer detect Intel BIOSGuard
- Remove parsing the developer_name tag
- Remove the vendor description
- Save non-empty UEFI padding sections as shards
- Set a max-age when sending chunked files
- Show a notification if unable to change component values
- Show a warning when a security update is detected without any issues
- Show better verified report output
- Use a bubble graph for the CVE timeline
- Use a volume guids to make UEFI R2 queries much, much faster
- Use the AppStream ID when deduping uploaded firmware
- Use the mirrored release image in more cases
- Verify the AppStream ID was valid if modified

13.5 1.3.2 (2021-06-22)

This release adds the following features:

- Add an optional PSIRT URL for each vendor
- Add a plugin which uses uefi_r2 to add shard attributes
- Add support for component soft-requirements
- Allow exporting the embargoed firmware using PULP_MANIFEST
- Allow searching for files by checksum on the internal dashboard
- Allow vendor managers to purge firmware without asking an admin
- Do not overwrite when resigning and use unique filenames for each revision

This release fixes the following bugs:

- Be more helpful when failing to load invalid XML
- Dedupe the component requirements where allowed
- Do not allow the update description to contain the firmware name
- Do not autodecode content when mirroring using sync-pulp.py
- Explicitly set the CDN Cache-Control to be 4 hours by default
- Ask vendors to provide 10 useful release description words
- Include the update images in the PULP_MANIFEST file
- Resign any files that do not include the PKCS#7 certificate

13.6 1.3.1 (2021-04-06)

This release adds the following features:

- Add a firmware timestamp that specifies the CVE embargo date
- Add a LVFS component problem if the version format is inconsistent
- Hard require the version format to allow pushing to stable
- Record the reason for moving a firmware to a new remote
- Record the user and when a component issue was added
- Support VINCE security advisory IDs

This release fixes the following bugs:

- Allow setting a vendor default for the .inf firmware parsing
- Allow uploading files with all issue types
- Fix some checksum confusion for duplicate firmware
- Fix unpinning files using Pinata
- Fix warnings with new SQLAchemy versions
- Never include generic components in the mdsync data
- Return JSON for robot uploaders

- Store the old remote ID in the FirmwareEvent
- Use the remote name, not the icon name for mdsync export
- Write the <issue> tags into the AppStream metadata

13.7 1.3.0 (2021-02-08)

This release adds the following features:

- Add new page for the latest devices supported
- Add support for the `<artifact>` AppStream tag
- Add support for the Intel technical advisory issue tags
- Allow adding optional default icons to categories and protocols
- Allow components to specify an optional branch
- Allow exporting the component back to MetaInfo XML format
- Assign a release tag style for specific vendor per-category
- Mirror non-export-controlled public firmware to IPFS
- Provide a healthcheck endpoint
- Send a monthly email about firmware left in embargo or testing
- Show a device status page showing all the versions in all remotes

This release fixes the following bugs:

- Add missing support for `LVFS::UpdateImage` and `Verfmt('number')`
- Add some documentation on adding screenshots and using the LVFS offline
- Allow adding and removing component GUIDs on the web UI
- Allow a `<project_license>` of BSD
- Allow changing firmware licenses without re-uploading firmware
- Allow non-admin users to resign firmware
- Allow QA users to change the component name, ID and summary
- Allow searching by filename, requirement or CVE when logged in
- Allow supplying a generic 'overview' component for composite devices
- Allow vendors to specify client requirements
- Change the dropped-GUID from an upload `flash()` to a waivable test
- Check for more sneaky CVEs in update descriptions
- De-duplicate the requirements where appropriate
- Do not allow the vendor name "BIOS", "fwupd" or "LVFS" in the firmware `<name>`
- Do not do the GUID check against firmware uploaded to private
- Do not ever store the client hashed IP address in the database
- Do not use `send_from_directory()` to send large files

- Fix all CSRF issues after some security review
- Fix performance issue when getting recent firmware downloads
- Include the copyright information for MIT licenses
- Increase the upload timeout to 10 minutes
- Move the disable 2FA slider to a button
- Parse the AMI FPAT firmware prior to scanning with UEFIExtract
- Provide a nudge when editing a component if required values are unset
- Purge firmware that is deleted after just 30 days
- Record the client country code for analytics
- Reduce the number of buttons on the component overview
- Regenerate embargo remotes when modifying restrictions
- Run any pending tests every 60 minutes
- Update the bundled version of Chart.js
- Update the `README.txt` file during package signing
- Use a non-predictable vendor icon filename
- Use PyGnuTLS rather than using certtool when signing files
- Use python-cabarchive rather than GCab for parsing
- Use the CDN to serve public static files
- Write the `PULP_MANIFEST` with a predicatable order

13.8 1.2.0 (2020-06-09)

This release adds the following features:

- Add a filter view for user uploaded firmware
- Add a plugin to identify old microcode versions
- Add cached public stats of useful metrics
- Add support for LVFS::UpdateMessage
- Allow clients to upload anonymous HSI attrs
- Allow re-signing binaries
- Create Jcat files in archives and for metadata
- Delete firmware in embargo with newer public versions
- Disable unused user accounts for GDPR compliance
- Export the success confidence to the mdsync vendor
- Include LVFS::UpdateProtocol in the metadata
- Rewrite the AppStream screenshot URL to use the server CDN
- Rewrite the metainfo when signing the firmware

- Save metadata about Intel microcode blobs
- Support Lenovo, Dell and Intel specific security tags
- Use celery to process async operations

This release fixes the following bugs:

- Allow all users to view the profile page
- Allow a protocol to have no defined version format
- Allow QA users to see all ODM firmware uploaded
- Allow setting the category to 'Unknown'
- Allow specifying firmware versions when using the advanced requires editor
- Do not allow component modification when in testing and stable
- Do not backtrace if a component does not have a <name>
- Do not include a CSRF for public search queries
- Do not include the VersionFormat fallbacks if the fw requires a new enough fwupd
- Do not make the database server explode with a query like 'value=+foo'
- Do not save duplicate <requires>vendor-id</> tags to the metadata
- Ensure firmware again when it changes state
- Fix a regression when component claims were not being added
- Fix regression when getting security level of component
- Improve the report query speed by several orders of magnitude
- Include the vendor tag in the rewritten metainfo and AppStream XML
- Invalidate ODM remotes when a firmware is demoted back to private
- List <id> requires first in the metadata
- Make it more obvious that the firmware is waiting to be signed
- Make the LVFS username case insensitive
- Make the markdown to root function more robust
- Parse the <metadata_license> even when not in strict mode
- Set the SHA256 content checksum in the metadata
- Show a disabled button when the user has no ACL to move the firmware

13.9 1.1.6 (2020-01-28)

This release adds the following features:

- Add a atom feed to public device page
- Add a claim for systems supporting Intel BiosGuard and BootGuard
- Add a `dell-bios` version format
- Add a page to list consultants that can work on the LVFS

- Add a plugin to add component claims for specific shard GUIDs
- Add a release tag to store the vendor-specific firmware identifier
- Allow adding component claims based on the hash of a shard
- Allow syncing with other firmware databases
- Move the formal documentation to Sphinx

This release fixes the following bugs:

- Add many more database indexes to improve performance
- Add some missing vendor checks when proxying to the user ACL
- Allow vendor managers to see a read-only view of the restrictions page
- Always use the vendor-id restrictions of the ODM, not the OEM
- Fix support for multiple `LVFS::VersionFormat` tags
- Include a vendor ID by default for testing accounts
- Make more queries compatible with PostgreSQL
- Never include firmware in private in any embargo remote
- Only show vendors with LVFS users on the vendorlist
- Reduce the memory consumption when running cron and doing yara queries
- Update the firmware report count at upload time
- Use SHA256 when storing the upload checksum
- Use the correct filename for a PKCS-7 payload signature
- Use UEFIExtract rather than chipsec to extract shards

13.10 1.1.5 (2019-11-15)

This release adds the following features:

- Add support for matching firmware requirements on device parents
- Allow researchers to run YARA queries on the public firmware
- Allow the blocklist plugin to add persistent claims
- Use PSPTool to parse the AMD PSP section

This release fixes the following bugs:

- Add the Dell PFS as a component shard
- Allow the owner of the firmware to always change update details
- Convert to Blueprints to improve page loading time
- Do not hardcode the list of version formats in various places
- Do not share the shard name between GUIDs
- Only auto-demote stable-to-testing, not testing-to-embargo or stable-to-embargo
- Show the version format versions with no trailing zeros

13.11 1.1.4 (2019-09-26)

This release adds the following features:

- Add component issues such as CVEs in a structured way
- Add more OEM notification emails for ODM actions
- Add support for name variant suffixes
- Add vendor namespaces to enforce ODM relationships
- Allow searching for CVEs when logged in
- Allow the OEM to better control what the ODM is able to do

This release fixes the following bugs:

- Allow vendors to optionally disable the inf parsing
- Blacklist generic GUIDs like 'main-system-firmware'
- Check the source and release URLs are valid if provided
- Do not show deleted firmware on the recent list on the dashboard
- Don't auto-demote firmware because of old reports
- Enforce the VersionFormat if the version is an integer
- Fix a crash if uploading a file with a missing metadata_license tag
- Provide a way to un-disable users as a vendor manager
- Regenerate embargo remotes ever 5 minutes
- Use a sane error message on upload when a component drops a GUID

13.12 1.1.3 (2019-08-06)

This release adds the following features:

- Show a nag message for admin or manager account without 2FA
- Do not use AppStream-glib to parse the metainfo file
- Automatically demote firmware with more than 5 failures and a success rate of %lt;70%
- Allow firmware or vendors to enable DoNotTrack functionality
- Show the user capabilities in the headerbar
- Protect all forms against CSRF

This release fixes the following bugs:

- Retry all existing tests if the category or protocol is changed
- Do not allow forward slashes in AppStream ID values
- Use a proper AppStream ID for the CHIPSEC shards
- Show flashed messages on the landing page
- Better support firmware requires without conditions or versions
- Do not allow AppStream markup in non description elements

13.13 1.1.2 (2019-05-28)

This release adds the following features:

- Add a new plugin to check portable executable files
- Save the shards in an on-disk cache which allows re-running tests
- Add a failure for any firmware that is signed with a 3-year expired certificate
- Add shard certificates to the database and show them in the component view

This release fixes the following bugs:

- Make it easier to enter multiline text as plugin settings

13.14 1.1.1 (2019-05-21)

This release adds the following features:

- Allow managers to edit their own list of embargoed countries
- Record the size and entropy of the component shards when parsing
- Analyze Intel ME firmware when it is uploaded

This release fixes the following bugs:

- Do not expect device checksums for ME or EC firmware

13.15 1.1.0 (2019-05-14)

This release adds the following features:

- Run CHIPSEC on all UEFI firmware files
- Show details of UEFI firmware volumes for capsule updates
- Show differences between public revisions of firmware
- Provide some extra information about detected firmware shards

This release fixes the following bugs:

- Only decompress the firmware once when running tests
- Make the component detail page a bit less monolithic
- Never leave tests in the running state if a plugin crashes

13.16 1.0.0 (2019-05-02)

This release adds the following features:

- Allow the admin to change the AppStream ID or name of components

This release fixes the following bugs:

- Do not allow the telemetry card title to overflow
- Ensure the `firmware-flashed` value is a valid lowercase GUID
- Make the component requirements page easier to use
- Do not add duplicate <`hardware`> values
- Remove the hard-to-use breadcrumb and use a single back button

CHAPTER

FOURTEEN

FIRMWARE EMBEDDED SBOM SPECIFICATION

Version: 0.9 (**DRAFT**)

Date: February 21, 2024

14.1 Acknowledgements

Authors:

- Richard Hughes (Red Hat)
- Martin Fernandez (Eclypsium)
- Adam Williamson (Red Hat)

Many thanks should also go to the UEFI SBOM Sub Team for all thier support in the creation of this document.

This specification document may be subsumed by a future UEFI specification or best practice document, but was published here to provide a reference specification in the interim.

14.2 Preface

The purpose of this document is to present a set of guidelines and best practices for vendors of firmware to provide Software Bill of Materials (SBoM) information to their clients and customers, to aid in vulnerability detection and license management.

Note: The keywords "**MUST**", "**MUST NOT**", "**REQUIRED**", "**SHALL**", "**SHALL NOT**", "**SHOULD**", "**SHOULD NOT**", "**RECOMMENDED**", "**MAY**", and "**OPTIONAL**" in this document should be interpreted as described in RFC 2119.

14.3 Glossary

This document assumes a working knowledge of terminology related to firmware, and of software concepts such as "libraries" and "compilers". The terms defined in this glossary may appear in italics as a reminder that they are being used as defined here.

Readers may be expecting to see terms like "IBV" (Independent BIOS Vendor), "ODM" (Original Design Manufacturer), "IFV" (Independent Firmware Vendor) and "OEM" (Original Equipment Manufacturer), but this document mostly avoids those terms. This is because those entities may, at any given moment and in any given commercial arrangement, be acting as *component vendors*, *firmware vendors* or *platform vendors* in the context of this document.

- **SBoM**: Software Bill of Materials. A formal document which can be used to articulate what components are contained within a binary deliverable, and who is responsible for each part.
- **Component**: any identifiable, discrete element of a firmware, including but not limited to any item that can be removed from, replaced in or added to a file volume or archive. This includes, but is not limited to, PE files, PEIMs, CPU microcodes, CMSE/PSP, FSP/AGESA, EC and OptionROMs – but **SHOULD NOT** include encryption keys or source code references. Each component may be provided as a precompiled binary by a *component vendor* to a *firmware vendor*, or it may be built from an independent source code tree by the *firmware vendor*.
- **Component SBoM**: an SBoM for a single *component*.
- **Component Vendor**: a party responsible for directly supplying a *component* for use by a *firmware vendor* in a firmware image.
- **Firmware**: a complete firmware image, which typically comprises multiple *components*.
- **Firmware SBoM**: an SBoM that represents all the *components* present in a single *firmware* and which could be generated in full or in part by combining *component* SBoMs.
- **Firmware Vendor**: a party responsible for building *firmware*, for use by the *platform vendor*.
- **Platform SBoM**: an SBoM that represents all the components in use on a real-world device. This may be equivalent to the *firmware SBoM* for single system firmware deployed on a device, or be a superset that includes metadata for multiple firmware (e.g. separate firmware for the system and for an attached touchpad or camera device).
- **Platform Vendor**: the party responsible for supplying a combined platform firmware image, typically comprising multiple firmware, for use on end-user hardware.
- **Source Code**: Text written in a program language (for example, C, assembly or Rust) that is compiled into binary object files and is not included verbatim in the firmware image.

14.4 Introduction

Due to the increasing number of high-profile supply chain attacks, it has become more important to record information about critical software such as system and peripheral firmware. For US companies, Executive Order 14028 "Improving the Nation's Cybersecurity" and the Cyber Trust Mark now make providing an SBoM with this information a legal obligation for many companies.

It has traditionally been difficult to build firmware or platform SBoMs for systems due to the involvement of three separate entities: the *Firmware Vendor* that produces the bulk of the *source code*, the ODM (Original Design Manufacturer) that compiles it with other additional code and adds additional binaries, and the OEM (Original Equipment Manufacturer) that may add their own extensions and then distributes the firmware. Most consumer laptop and desktop devices also have many other firmware blobs of firmware supplied for factory burn-in, e.g. fingerprint reader, SD card reader, touchpad, PCI retimer, Synaptics MST, Intel Thunderbolt, and many more – and these might not have any communication channel to the system firmware at all.

End-users do not buy "firmware" and any firmware deliverable will normally be included in a larger OEM per-device *platform SBoM*. At the same time, we also need to provide access to the runtime "current firmware SBoM" so that we can use newer technologies such as VEX to automatically identify systems that require security fixes.

This document explains why SBoM metadata for all constituent *components* should be embedded in all *firmware*, what should be included in it, **and** how it should be used as part of a larger *platform SBoM* that is useful to end-users.

14.5 Embedding the SBoM

When we talk about "embedding the SBoM", we refer to the general idea of having SBoM metadata for all *components* in a given *firmware* included into the firmware image itself, either by providing a *firmware SBoM* or just by ensuring all components are represented in multiple *component SBoMs*.

14.5.1 Benefits of Embedding

Traditionally there has been pressure to keep firmware images as small as possible to minimize SPI storage space and to minimize the cost of the *Hardware Bill of Materials*. While this is a noble aim, sacrificing a few hundred bytes of space for an embedded SBoM has several advantages:

- The SBoM does not need to be verified against a binary deliverable, it can be assumed to be "part of" the existing source artefact itself.
- Vendors at any link in the supply chain that don't care about or understand SBoMs do not "strip" the SBoM information.
- The *component SBoMs and/or firmware SBoMs* from all the factory burn-in *firmware* images can be combined into one generated public *platform SBoM* that can be used for contractual or compliance reasons, without the need to request *component* or *firmware* SBoMs separately from each *component vendor* and *firmware vendor*.
- Build-time automated embedding as part of CI/CD is recommended as part of the US Cyber Trust Mark initiative.
- Some firmware build systems require the firmware blob and definition files to be put in a predefined place to generate a new firmware binary, which means non-embedded SBoM metadata may get out-of-sync with the blob.

If the SBoM is not embedded as a build artifact, a firmware engineer could rebuild the firmware capsule and forget to also regenerate or replace the SBoM in the new archive because it is a separate process that is hard to verify was done. If the SBoM is part of the image itself and *automatically constructed* as part of the deliverable, then it is impossible to forget. Sending the capsule or manually dumped ROM image to a QA engineer means they can know with almost complete certainty what blobs the image was built with. Embedding the SBoM makes doing the "right" thing easy and doing the "wrong" thing hard.

14.5.2 General Best Practices

All *component vendors* **SHOULD** embed an SBoM in the component image, formatted as described below. They **MAY** also create a more detailed detached SBoM (for instance referencing internal issues or *source code* filenames) that **MAY** be provided to the *firmware vendor* under NDA.

Firmware vendors **SHOULD** ensure embedded SBoM metadata is included for every PE binary and all additional *components* included in the *firmware* formatted as described below. This **MUST** be done by:

- Including the SBoM for each *component* in a "defragmented" *firmware SBoM* created at build time, **OR**
- Ensuring that each *component* contains embedded SBoM metadata, **OR**
- Doing both of the above.

Component and *firmware* SBoMs **SHOULD NOT** reference any code or blobs which are not actually present, or which have been disabled in the system.

14.5.3 Embedded SBoM Formats

Firmware and *component vendors* **MUST** use the DTMF coSWID binary format with CBOR encoding when directly embedding SBoM sections in firmware. This format was chosen due to the small compiled size of data compared to SPDX (YAML or JSON) and SWID (XML), because the specification is freely available and because it can act as a superset format to both SPDX and CycloneDX.

Built Portable Executable (PE) Binaries

Most *components* in a typical *firmware* are compiled from *source code* and linked into PE binaries. These can be considered components whose vendor is the *firmware vendor*.

The *firmware vendor* **SHOULD** ensure that the SBoM metadata is automatically built and verified at compile time and then added to the PE binary (in the `.sbom` COFF section), placed directly in the "defragmented" *firmware SBoM* (see below), or both. If for any reason this is not done automatically at compile time, the *firmware vendor* still **MUST** ensure the SBoM is included in the binary `.sbom` COFF section or the "defragmented" *firmware SBoM*, as required above.

For Tianocore/EDK2 firmware, there is an example showing how to supplement the information in the `.inf` file with per-component and per-platform overrides. More specific recommendations on how to include additional artifacts into the `.sbom` section have not been made as this will be heavily influenced by the existing proprietary build system and tools used to build the image.

In the case where there is no natural place to store the *component SBoM*, it **SHOULD** be included as a per-volume metadata section. In this case it **MUST** include a uSWID magic header, as described in *Components that are not Portable Executables (PE)* below.

Precompiled Portable Executable (PE) Binaries

Firmware vendors do not have to compile all the PE binaries in the EFI volume from *source code*. They may get pre-compiled and pre-signed binaries from third-party *component vendors*. *Component vendors* **SHOULD** include the coSWID SBoM metadata for these components in a `.sbom` COFF section which can be easily included at link time. These binaries **MUST NOT** use the magic header of uSWID (described below) as the PE header can be parsed for the correct offset of the section.

An additional benefit of including the SBoM in a COFF section is that it is verified by the existing Authenticode digital signature.

If a *firmware vendor* uses a PE binary which does not have this embedded SBoM metadata, the *firmware vendor* **MUST** ensure SBoM metadata for the binary is present in a "defragmented" firmware SBoM, as described below.

Components that are not Portable Executables (PE)

When embedding SBoM metadata into any binary that is not a Portable Executable (PE), the *component vendor* **MUST** use the discoverable uSWID header so that software can easily discover the embedded SBoM. The 25-byte uSWID header is listed below:

```
uint8_t[16]   magic, "\x53\x42\x4F\x4D\xD6\xBA\x2E\xAC\xA3\xE6\x7A\x52\xAA\xEE\x3B\xAF"
uint8_t       header version, typically 0x03
uint16_t      little-endian header length, typically 0x19
uint32_t      little-endian payload length
uint8_t       flags
                0x00: no flags set
                0x01: compressed payload
```

(continues on next page)

(continued from previous page)

```
uint8_t       payload compression type
                0x00: none
                0x01: zlib
                0x02: lzma
```

The header length **MAY** be increased for alignment reasons (e.g. to 0x100 bytes), and in this case the additional header padding **MUST** be `NUL` bytes.

The uSWID payload **SHOULD** be compressed with either zlib or LZMA, and a firmware image containing the binary **SHOULD** pass validation using `uswid`, for example:

```
$ uswid --load firmware.bin --validate
Found USWID header at offset: 0x18000
Validation problems:
dd4bbe2e40ba          component: No software name (uSWID >= v0.4.7)
dd4bbe2e40ba             entity: Invalid regid http://www.hughsie.com,   should be DNS␣
↪name hughsie.com (uSWID >= v0.4.7)
dd4bbe2e40ba             entity: No entity marked as TagCreator (uSWID >= v0.4.7)
dd4bbe2e40ba            payload: No SHA256 hash in FSPS (uSWID >= v0.4.7)
dd4bbe2e40ba             link: Has no LICENSE (uSWID >= v0.4.7)
dd4bbe2e40ba             link: Has no COMPILER (uSWID >= v0.4.7)
```

Although there are many tools for the distribution of the *firmware SBoM* to end-users, fewer tools exist to embed SBoMs into binary blobs, or to extract and merge SBoM components to build a *firmware SBoM* or *platform SBoM*. The python-uswid project is one such tool.

Defragmented firmware SBoM

A firmware image can contain a "defragmented" top-level *firmware SBoM* with a uSWID header, produced at build time. If each *component* in the image has uSWID metadata, coSWID data in PE/COFF `.sbom` sections and/or file volumes with uSWID metadata, the *firmware vendor* **MAY** omit this *firmware SBoM*. If not, the *firmware vendor* **MUST** include it.

If the *firmware SBoM* is present:

- It **MUST** contain all *component SBoMs* present in the image. This requirement is to ensure that tools do not need to combine and deduplicate *component SBoMs* with the *firmware SBoM* to provide all available information.
- It **SHOULD** be compressed.
- The components **MAY** also have *component SBoMs* as described in this document, to allow them to be analyzed in isolation.

14.6 Data Provided by the SBoM

The purpose of an SBoM is to tell the end-user what components make up the software deliverable, and to give them information on where it was retrieved from or built. The questions end-users need to be able to answer are "what version of OpenSSL is included, and where did it come from" and "do I trust all the companies contributing code and binaries to this image". Answering the *what* and *who* in a standardized way also allows us to use other specifications such as VEX.

In this section we use the term "SBoM component" to refer to a single ingredient within an SBoM (in a coSWID SBoM, this is a single tag).

Each SBoM component **SHOULD** describe either:

- A single *component*, as defined in the *glossary*, or
- An individually identifiable part of a *component* that has security and/or licensing implications, for example an image loading library used by a PE binary, or
- Something that has security and/or licensing implications and was used to produce a *component,* but is not present in the *component* itself, for example a compiler used to produce a PE binary, or
- Any kind of defined logical component, for example "optional features" or "value add" options that may be matched from a VEX file (see below).

Each *component* **MUST** be represented by an SBoM component in its *component SBoM*, or the *firmware SBoM* if the component does not have its own SBoM (see the *Embedding the SBoM* section above for possible scenarios). Libraries, compilers etc. **SHOULD** be represented by SBoM components (see the *Component Relationships* section below for more on this). Thus, a *component SBoM* or *firmware SBoM* **MUST** contain at least one tag, and **MAY** contain more.

For components or relationships that cannot currently be disclosed for legal reasons, vendors **MAY** use the literal text `REDACTED` in place of the correct string value. This is intended as a **temporary** measure while contracts or NDAs are renegotiated. Any SBoM components with `REDACTED` text **MAY** be marked as incomplete and **MUST** fail validation.

14.6.1 Required Attributes

Each tag:

- **MUST** have an identifier in the form of a GUID. See the *Identifier* section below for more details.
- **MUST** have a non-zero length descriptive name, e.g. "CryptoDxe", and **SHOULD NOT** include a file extension as this is already included in the SWID payload section.
- **MUST have at least one entity entry and SHOULD have more than one, if more than one legal entity is involved in its creation, maintenance and/or distribution.**
  - One entity **MUST** have the tag-creator role.
  - One entity **MUST** have the software-creator role, and it **MAY** be the same entity as the one specified in tag-creator. See the *Vendor Entity* section below for details.
  - In specifying entity roles, vendors **SHOULD** be careful not to make business relationships public that are not already in the public domain.
- **MUST** have a version, which **SHOULD** be a semantic version like `1.2.3`.
- **MUST have a file hash that is generated from all the source files, if it is a binary built from *source code* or other constituent parts. This MUST be either a SHA-1 or SHA-256 hash.**
  - This is what uSWID calls a "colloquial version."
- **SHOULD have a revision control tree hash which MUST be either a SHA-1 or SHA-256 hash (e.g. the output from `git describe`), if it is a binary built from *source code* under revision control.**
  - This is what uSWID calls an "edition."
- **MAY** or **MUST** include one or more link entries expressing relationship(s) to another SBoM component. See the *Component Relationships* section below for details, including when link entries are **REQUIRED** and when they are **OPTIONAL**.

The file hash **SHOULD** include the hashes of the *source code* files used to construct the binary, such as `.c` and `.h` files. Any library statically-linked with the PE binary **SHOULD** be included as an additional SBoM component.

Identifier

In some cases, the most obvious identifier to use for the SBoM component is already in a GUID form – for instance using the UEFI GUID defined in an official specification or reference implementation. In other cases, like GCC (where there is no GUID defined), vendors **MUST** use a `swid:` prefix to generate a GUID that is linked within the object. Using a GUID is deliberate because it can obscure internal references, and can be encoded as a 128-bit number in coSWID.

Example component IDs could include:

- `swid:intel-microcode-706E5-80`
- `swid:gcc`
- `f43cae5a-baea-5023-bc90-3a83cd4785cc which is UUID(DNS, “gcc”)`

Some of this information is already present in projects such as EDK2 in the various `.inf` files.

Firmware vendors and *component vendors* **SHOULD** consult with any upstream projects before deciding identifier GUIDs.

Forked components modified by the *firmware vendor* **MUST** have an identifier different from the upstream component identifier.

The identifier GUID:

- **SHOULD NOT** include the component version, file or tree hash or revision.
- **MAY** allow comparing some components against SBoMs from different vendors.

Vendor Entity

An “entity” describes a party responsible for the creation, maintenance, and/or distribution of a firmware or component. An entity can perform one or more roles (e.g. creator, maintainer and distributor), and multiple entities (even with the same role) can be defined for each component.

For instance, Intel FSP is created by Intel, maintained by Intel, and distributed by Intel. A modified DXE might originally be created by Intel in EDK2, but then be modified and maintained by AMI and distributed by Lenovo. In this case, the component for the FSP would have only one entity entry, but the component for the DXE would have three entity entries.

For each entity entry:

- The name **MUST** be the legal or common-use name of the open-source project, the component vendor, the firmware vendor, or the platform vendor.
- The registration ID **MUST** be the DNS name of the named legal entity, or the DNS name of the upstream project URL in the case of open-source projects.

Component Relationships

SBoM component links are used to supply additional information about how components relate to each other. They also include any required licensing information, statically linked libraries and links to additional resources. Libraries that may be matched from a VEX file (for instance, where a third-party library has previously security vulnerabilities) **SHOULD** be included as a component, but other internal libraries **MAY** be omitted. SBoM components **MAY** use multiple links, even of the same relationship type.

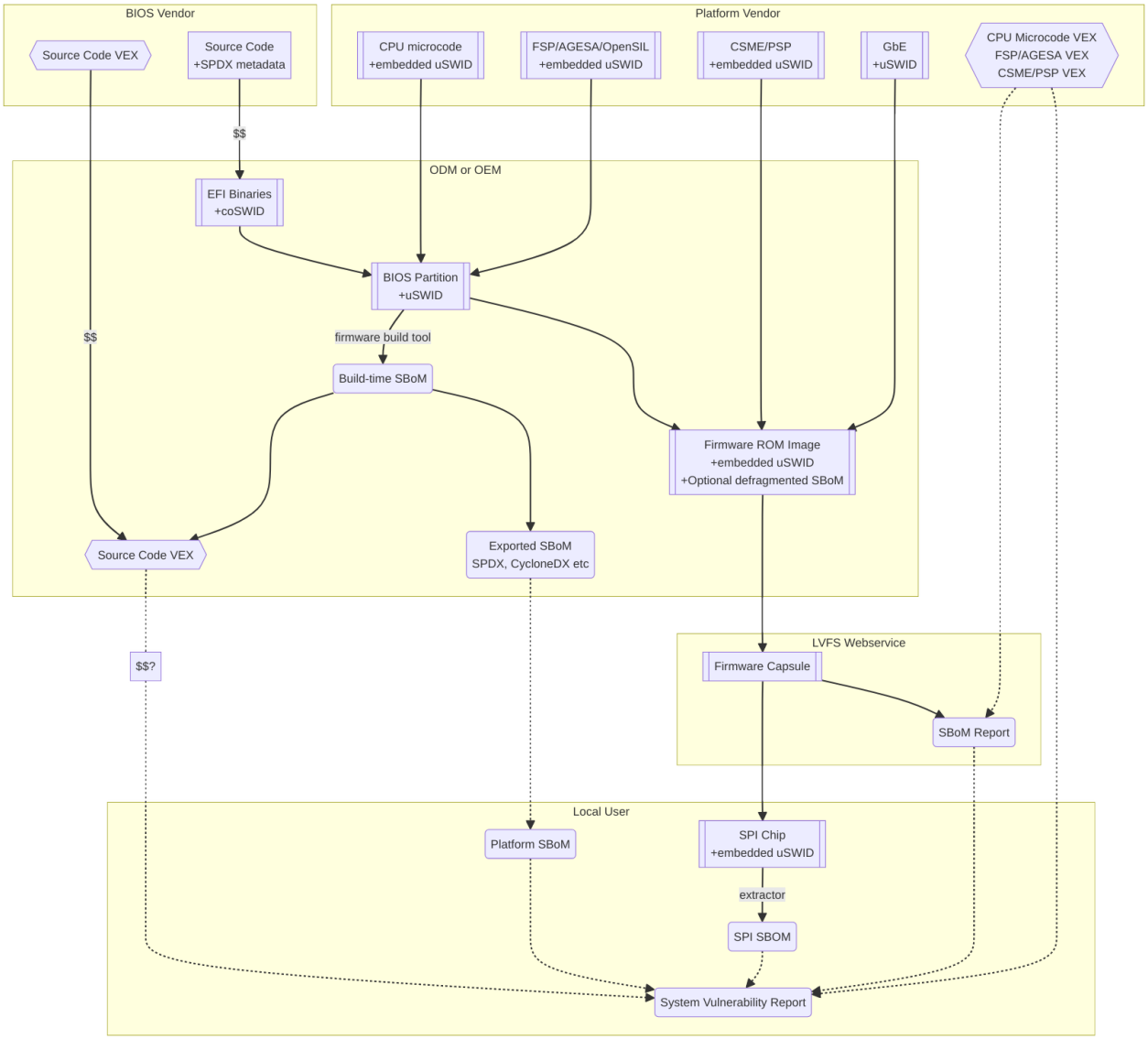
- **SBoM components representing open-source software MUST include one or more license link(s) indicating all licenses that apply.**

  - The URL for each license link **MUST** be the SPDX license URL, e.g.: `https://spdx.org/licenses/LGPL-2.1-or-later.html`
  - The `license` relationship type **MUST** be used.
  - All open-source code **SHOULD** be identified with its own SBoM component to allow verification of license compliance.

- **SBoM components representing non-open-source software SHOULD include one or more license link(s) indicating all licenses that apply.**
  - The URL for each license link **MUST** be a public webpage with the full text of the proprietary license.
  - The `license` relationship type **MUST** be used.
- **SBoM components representing compiled binaries SHOULD reference SBoM components representing the compiler and linker used to build the binary where possible.**
  - The `see-also` relationship type **MUST** be used, and the `swid`-prefixed URL **MUST** be an existing component identifier defined in the component or firmware SBoM.
- **SBoM components representing compiled binaries SHOULD reference SBoM components representing libraries that are linked into the binary and that may be referenced in VEX documents (see below).**
  - The `requires` relationship type **MUST** be used, and the `swid`-prefixed URL **MUST** point to an existing component in the SBoM.
- **SBoM components MAY include a link specifying the source URL where they can be downloaded. e.g. `https://github.com/intel/FSP/AmberLakeFspBinPkg`**
  - The `installationmedia` relationship type **MUST** be used.

14.7 SBoM Information Flow

The figure below shows the possible flows of SBoM information from the *component vendor(s)*, *firmware vendor(s)* and/or *platform vendor* to the end-user. VEX data (see below) is used to notify the end user about security issues of components referenced by the SBoM.



Depending on existing business relationships, the *firmware vendor* (the ODM) may take on some of the responsibilities of the platform vendor (the OEM) or the *component vendor* (the IBV).

Dumping the SPI contents using an external SPI programmer or OS interface allows the end-user to extract a "current" firmware SBoM. This allows analyzing the image without having access to a public SBoM provided by the platform vendor or a vendor neutral firmware provider like the Linux Vendor Firmware Service ("LVFS").

To comply with Executive Order 14028, OEM vendors **SHOULD** also publish either the SPDX or CycloneDX SBoM export as a downloadable file on the public device webpage. The SHA-256 checksum of the generated SBoM **SHOULD** be used as the unique collection ID for the component and firmware SBoMs. This enables the SBoM to be found using a search engine even if the original OEM has been renamed or the device HTML URI has been modified.

14.8 Using VEX Rules

Vulnerability Exploitability eXchange (VEX) allows a *component vendor* to assert the status of a specific vulnerability in a particular firmware. VEX can have any of the following "status" values for each component:

- **Not affected**: No remediation is required regarding this vulnerability.
- **Affected**: Actions are recommended to remediate or address this vulnerability.
- **Fixed**: Represents that these product versions contain a fix for the vulnerability.
- **Under Investigation**: It is not yet known whether these product versions are affected by the vulnerability.

Only the entity with the *source code* tree and the config files used to build it (usually the IBV or ODM) has all the information required to know whether a given EFI binary is affected by a specific vulnerability.

If our aim is to find out if a specific firmware is vulnerable to a specific security issue, there are only three ways to solve this without access to a complete SBoM:

- The end-user asks the *component vendor*, who finds the firmware version, checks out the *source code* for that revision, then looks for affected code, and replies with the answer.
- The *component vendor* proactively passes detailed vulnerability status and remediation info to the immediate downstream supply chain partner, who then in turn proactively passes this down to each customer.
- The *component vendor* shares the code and the config to the customer and assumes the customer can work it out themselves.

We consider these ways to be clearly unsatisfactory. Therefore, both *component vendors* and *platform vendors* **SHOULD** upload the SBoM to a trusted neutral entity, allowing multiple customers and end-users to query the information. The neutral entity **MAY** also process additional trusted VEX data directly from *component vendors*, which allows *firmware* to automatically be marked as *affected* or *not affected* without direct involvement of the *firmware vendor*.

Vendors writing VEX rules MUST use the same identifier as used in the SBoM. VEX product IDs are specified using PURL, and the GUID **MUST** be used as the component name. Where a semantic version is required it **MAY** also be specified.

For example:

- `pkg:dca533ab-2c1f-4327-9b2b-09ac19533404`
- `pkg:dca533ab-2c1f-4327-9b2b-09ac19533404@15.35.2039`

Further details about using Vulnerability Exploitability eXchange (VEX) standards such as OpenVEX with embedded firmware SBoMs will be provided in the future.

14.9 Final Comments

With these sets of recommendations we feel sure that the resulting firmware SBoM will be useful to security teams and end-users alike. This would greatly benefit the entire firmware ecosystem and make the global supply chain measurably safer.

14.10 Appendix

14.10.1 External SBoM Metadata

This document strongly encourages vendors to embed the SBoM metadata into the respective binaries, but there are two situations where externally referenced SBoM metadata would be allowed:

- Where the binary is loaded onto critically space-constrained devices, for example microcode that is loaded into the processor itself.
- Where only later newer versions of the component have embedded SBoM metadata, and backwards compatibility is required with older revisions.

In these cases, the *component vendor* **MUST** provide "detached metadata" from the same source (or in the same archive file) as is used to distribute the immutable blob.

As the SBoM metadata is detached, vendors **MUST** ensure that the files do not get "out of sync" and are updated at the same time in the firmware source tree. Detached metadata **MUST** always contain the SHA256 hash value of the binary as evidence to allow validation and **MAY** be signed using a detached signature if the archive is not already signed. The public key **SHOULD** be distributed on a keyserver or company website for verification.

14.10.2 Wasted Space Concerns

Some vendors have expressed concerns about "wasted" space from including the SBoM data in the binary image. For source components such as CPU microcode, a single *component* and vendor *entity* would use an additional ~350 bytes (zlib compressed coSWID), compared to 48kB for the average EFI binary and 25kb for a typical vendor BGRT "splash" logo.

The `uswid` command can automatically generate a complete "worst case" platform SBoM with 1,000 plausible components. This SBoM requires an additional 140kB of SPI flash space (uncompressed coSWID), or 60kB when compressed with LZMA. For reference, the average free space in an Intel Flash ROM BIOS partition is 5.26Mb, where "free space" is defined as a greater than 100KiB stream of consecutive 0xFF's after the first detected EFI file volume. Adding the SBoM as embedded metadata would use 1.1% of the available free space. Other firmware ecosystems such as Coreboot also now include SBoM generation as part of the monolithic image.

14.10.3 Getting the Runtime SBoM

The ACPI `SBOM` ACPI table may be used in the future to return the coSWID formatted binary SBoM data from any device exporting an ACPI callable interface. Further details will be provided when the SBOM table has been implemented.

If the platform allows direct access to the system SPI device, then the entire firmware image can be dumped to a local file and analyzed by tools such as `uswid`.

14.10.4 Converting the SBoM

The embedded SBoM **SHOULD** be converted it into one or more SBoM export formats before publication.

This can be achieved easily using tools such as `uswid`. For example, this can be used to produce two JSON files in CycloneDX and SPDX formats from the platform image:

```
$ uswid --load rom.bin --save cyclonedx-bom.json
$ uswid --load rom.bin --save spdx.json
```

14.10.5 Signing the SBoM

The embedded SBoM **MAY** be signed, and **MAY** also be included in the firmware checksum. If the firmware component is signed then the SBoM **SHOULD** be included in to the signature. The signing step is optional because a malicious silicon provider can typically do much worse things (e.g. adding or replacing a DXE binary) than modify the SBoM metadata.

14.10.6 Using the LVFS

When firmware is uploaded to the LVFS it automatically extracts all available SBoM metadata and generates a HTML page with SPDX, SWID and CycloneDX download links that can be used for compliance purposes. The LVFS **MAY** allow vendors to upload firmware or platform SBoMs without uploading the firmware binary. Other services like Windows Update may offer this service in the future.

The VEX "trusted neutral entity" **MAY** also be the LVFS, even for firmware updates not distributed by the LVFS. Uploading VEX data requires vendors to register for a LVFS vendor account which is available at no cost.

CHAPTER

FIFTEEN

CHROMEOS FIRMWARE TESTING

15.1 Prerequisites

- A Chrome OS device.
- A device to test that is supported by the installed version of `fwupd` in Chrome OS, i.e. the device firmware update plugin is working.

15.2 Prepare Chrome OS for testing

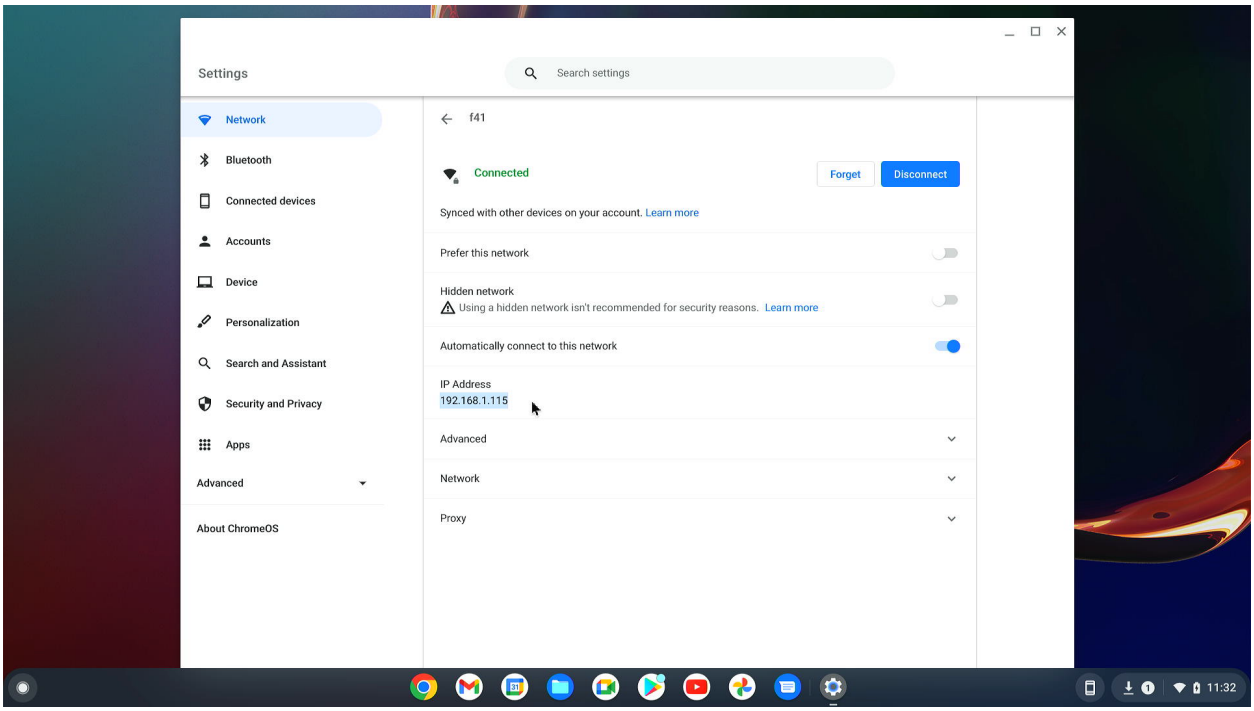
15.2.1 Pre-conditions

1. The Chrome-based device must be updated to the recent version of Chrome OS (see official documentation: https://chromium.googlesource.com/chromiumos/docs/+/main/developer_guide.md#Installing-Chromium-OS-on-your-Device)
2. WiFi connection with unrestricted access to LVFS site.
3. For reference, we have used Samsung Galaxy Chromebook 2 (codename `hatch`).

15.2.2 Developer Mode

The Chrome-based device must be switched to development mode: https://chromium.googlesource.com/chromiumos/docs/+/main/developer_mode.md#dev-mode

1. **Recovery mode** : Hold Esc + Refresh  and press Power.
2. Enter **Developer Mode** :
    a. On screen “Please insert a recovery USB stick or SD card.” press “Ctrl+D”.
    b. By pressing “Enter” confirm turning OS verification OFF.
    c. The system will be restarted.
    d. After that step on every boot the OS will warn you about OS verification disabled. Press “**Ctrl+D**” to proceed.
    e. The first boot after restart will set your device into Developer Mode, **all data on the device will be wiped out**!
3. During the first boot, set the network and log into the account.



4. Go to Settings and check IP address assigned to your device:
5. The IP address will be used for remote access to the device over SSH – for instance `192.168.1.115` it is assigned.

Disable rootfs verification

By default the filesystem is mounted in ReadOnly mode. For testing purposes we have to do some changes in configuration, meaning we have to disable rootfs verification.

1. Switch to the linux console on ChromeBook by pressing: "Ctrl + Alt + →" ('→' or 'F2' on top row).
2. You should see the login prompt asking for login.
3. Use username "*chronos*" for login, password is not needed.
4. Run command to remove the FS verification:

```
sudo /usr/share/vboot/bin/make_dev_ssd.sh --remove_rootfs_verification
```

   a. Sometimes the command above fails and asks to provide additional parameter " –partitions " with the partition ID.

      Please check the output carefully:

   b. For `hatch` the script is suggesting the 2-nd partition, so need to retry the command with suggested parameter:

```
sudo /usr/share/vboot/bin/make_dev_ssd.sh --remove_rootfs_verification --
↪partitions 2
```

5. Reboot the system with command:

```
sudo reboot
```

```
chronos@localhost ~ $ sudo /usr/share/vboot/bin/make_dev_ssd.sh --remove_rootfs_verification

  ERROR: YOU ARE TRYING TO MODIFY THE LIVE SYSTEM IMAGE /dev/mmcblk1.

  The system may become unusable after that change, especially when you have
  some auto updates in progress. To make it safer, we suggest you to only
  change the partition you have booted with. To do that, re-execute this command
  as:

    sudo /usr/share/vboot/bin/make_dev_ssd.sh --remove_rootfs_verification --partitions 2

  If you are sure to modify other partition, please invoke the command again and
  explicitly assign only one target partition for each time  (--partitions N )

make_dev_ssd.sh: ERROR: IMAGE /dev/mmcblk1 IS NOT MODIFIED.
```

```
chronos@localhost ~ $ sudo /usr/share/vboot/bin/make_dev_ssd.sh --remove_rootfs_verification --partitions 2
make_dev_ssd.sh: INFO: Kernel A: Disabled rootfs verification.
make_dev_ssd.sh: INFO: Backup of Kernel A is stored in: /mnt/stateful_partition/cros_sign_backups/kernel_A_20220705_00
3210.bin
make_dev_ssd.sh: INFO: Kernel A: Re-signed with developer keys successfully.
make_dev_ssd.sh: INFO: Successfully re-signed 1 of 1 kernel(s)  on device /dev/mmcblk1.
make_dev_ssd.sh: INFO: Please remember to reboot before updating the kernel on this device.
```

Allow *fwupd* access to HID devices for ChromeOS versions prior to v107

ChromeOS versions prior to version 107 have an issue with managing the HID devices like mice or keyboard. In this case we need to avoid this restriction.

1. Switch to linux console on ChromeBook by pressing: "Ctrl + Alt + →"
2. You should see the login prompt asking for login.
3. Use username "*chronos*" for login, no password should be asked.
4. Run command to give access to HID and `fwupd`:

   ```
   sudo usermod -a -G hidraw fwupd
   ```

5. Restart the `fwupd` with the command:

   ```
   sudo restart fwupd
   ```

Enable SSH access

It is recommended to enable SSH access allowing QA engineer remote connection to the ChromeBook instead of typing all commands directly.

1. Switch to linux console on ChromeBook by pressing: "Ctrl + Alt + →"
2. You should see the login prompt asking for login.
3. Use username "*chronos*" for login, no password should be asked.
4. Run command to enable SSH access:

   ```
   sudo /usr/libexec/debugd/helpers/dev_features_ssh
   ```

5. Set password for the `root` user:

   ```
   sudo passwd
   ```

password `test0000` is recommended as default for ChromeOS devices under the test.

6. Reboot the system with command:

```
sudo reboot
```

7. Now it is possible to use remote access with SSH tool to the Chromebook from your Linux host using the password above:

```
$ ssh root@192.168.1.115
Password:
localhost ~ #
```

15.3 Pack a fresh firmware into the CAB format

The CAB file is the container containing the firmware file and XML file with the metadata for LVFS and `fwupd` update daemon.

For the purpose of this documentation we will use the ColorHug open hardware colorimeter.

15.3.1 Firmware files

ColorHug device uses the following firmware file:

```
firmware.bin
```

15.3.2 Metadata files

The metadata format is described in documentation: https://lvfs.readthedocs.io/en/latest/metainfo.html

For composite devices, both LVFS and `fwupd` allow the use of a single CAB file. In this case we have to prepare and pack several XML files with metadata, one file for each firmware.

In the case of ColorHug, only one metadata file is requested:

```
firmware.metainfo.xml
```

The name of the files doesn't matter – the only requirement is the extension `.metainfo.xml`.

Metainfo file for ColorHug

The metainfo `firmware.metainfo.xml` file for the device:

```
<?xml version="1.0" encoding="UTF-8"?>
<component type="firmware">
  <id>com.hughski.ColorHug.firmware</id>
  <name>ColorHug</name>
  <summary>Firmware for the Hughski ColorHug Colorimeter</summary>
  <description>
    <p>
      Updating the firmware on your ColorHug device improves performance and adds new␣
```

(continues on next page)

(continued from previous page)

```
↪features.
    </p>
  </description>
  <provides>
    <firmware type="flashed">40338ceb-b966-4eae-adae-9c32edfcc484</firmware>
  </provides>
  <url type="homepage">http://www.hughski.com/</url>
  <metadata_license>CC0-1.0</metadata_license>
  <project_license>GPL-2.0-or-later</project_license>
  <categories>
    <category>X-Device</category>
  </categories>
  <custom>
    <value key="LVFS::VersionFormat">triplet</value>
    <value key="LVFS::UpdateProtocol">com.hughski.colorhug</value>
  </custom>
  <releases>
    <release version="1.2.6.1" date="2016-12-02" urgency="low">
      <description>
        <p>This release fixes prevents the firmware returning an error when the remote␣
↪SHA1 hash was never sent.</p>
      </description>
      <url type="source">https://github.com/hughski/colorhug1-firmware/releases/tag/1.2.6
↪</url>
    </release>
  </releases>
</component>
```

Important tags explanation

Both metadata XML above contains the minimal amount of data. The most interesting tags are:

- <id> – the name of AppStream unique identifier for the device. Vendor must choose the unique string in reverse-DNS style and this ID must contain the device name and `.firmware` suffix
- <name> – this is the short name of the device
  - <name_variant_suffix> – for composite devices, this is added to short name
- <firmware> – the GUID in this tag is extremely important! It helps `fwupd` to recognize the updatable device. See the output from `fwupdmgr get-devices` for devices GUIDs. It is allowed to use several GUIDs here if the same update file fits for several devices.
- <value key="LVFS::UpdateProtocol"> – here should be used the name of the protocol supported by the `colorhug` plugin (see README.md for actual protocols supported).
- urgency="low" – the urgency field has no effect on `fwupd` itself. This is the hint for UI frontends how to notify users, Gnome Software center for instance. At the moment Chrome OS has very limited UI support for device updates. The upstream is expecting following values:

Value	Meaning
low	Low importance
medium	Medium importance, e.g. optional update
high	High importance, e.g. recommended update
critical	Critical importance, e.g. urgent or security issue

15.3.3 Generation of the CAB file

The generation of the CAB file is required for uploading to LVFS and for local testing as well. The `gcab` tool is used for the generation under Linux:

```
$ gcab --create --nopath --verbose ColorHug-1.2.6.cab firmware.metainfo.xml firmware.bin
```

The `ColorHug-1.2.6.cab` will be created containing 2 files: 1 metadata XML and 1 firmware binary.

The generated file will contain only the minimal amount of the metadata. No additional information for firmwares validation, checksums or signatures are included at this step!

15.3.4 Upload file to ChromeBook

To test the generated file you need to copy it onto a ChromeBook device. The simplest method is to copy it via `ssh` from the build host. Please substitute the IP address from the example below with the IP address of your device, and use the password you've set during ChromeBook setup (`test0000` in the example):

```
$ scp ColorHug-1.2.6.cab root@192.168.1.115:~/
Password:
```

Alternatively you may want to use other methods for accessing the CAB file from ChromeBook device, for instance: own HTTP server, network share, USB mass storage and others.

15.4 Local test of the CAB file

15.4.1 Access to ChromeBook

Gain terminal access on ChromeBook via `ssh` (recommended) or with virtual console.

SSH method

Use `ssh` tool from your host to login as `root` user to ChromeBook (use IP address of your device and password you've set):

```
$ ssh root@192.168.1.115
Password:
localhost ~ #
```

Physical access method

1. Switch to the linux console on ChromeBook by pressing: "Ctrl + Alt + →" ('→' or 'F2' on top row).
2. You should see the login prompt asking for login.
3. Use username `root` for login, and password you've set (`test0000` by default):

```
localhost login: root
Password:
localhost ~ #
```

Virtual console via *crosh*

1. Start `crosh` terminal by pressing "Ctrl + Alt + T" in GUI
2. In the opened terminal window type `shell` command
3. Switch to `root` account

```
crosh> shell
chronos@localhost / $ sudo bash
localhost / #
```

15.4.2 Check if the device is supported by *fwupd*

- Attach the device to ChromeBook
- Collect information about the attached device, supported by `colorhug` plugin:

```
localhost ~ # fwupdmgr get-devices
Nightfury
│
├─ColorHug:
│         Device ID:          d9c9e0eb29c6f35160d400949c14db42f473f4d4
│         Summary:            An open source display colorimeter
│         Current version:    1.2.5
│         Vendor:             Hughski Ltd. (USB:0x273F)
│         Install Duration:   8 seconds
│         GUIDs:              40338ceb-b966-4eae-adae-9c32edfcc484
│                             afdcc391-6c33-5914-b4d2-b4dd71fe9c5a
│                             6bc5ff27-d631-5660-9991-6d24954c6f90 ← USB\VID_273F&
↪PID_1001
│                             4841a9e4-e5c8-5107-a83e-d6c6d9c21248 ← USB\VID_273F&
↪PID_1001&REV_0002
│         Device Flags:       • Updatable
│                             • Supported on remote server
│                             • Device can recover flash failures
│                             • Unsigned Payload
```

The most interesting here are device GUID(s) and device flags: `Updatable` meaning that the device updates are supported by `fwupd` manager, and `Supported on remote server` flag shows there are firmwares available on the LVFS site.

15.4.3 Upgrade the device with development FW CAB file

As mentioned above, the development variant of locally generated CAB file was not digitally signed, so it is not possible to install it on Chrome OS based devices with the `fwupdmgr` tool due security reasons:

```
localhost ~ # fwupdmgr local-install --allow-reinstall --allow-older ColorHug-1.2.6.cab -
↪-json
firmware signature missing or not trusted; set OnlyTrusted=false in /etc/fwupd/daemon.
↪conf ONLY if you are a firmware developer
```

To avoid that issue it is possible to use 2 methods:

1. Modify `/etc/fwupd/daemon.conf` to allow untrusted firmwares, and restart the daemon:

   ```
   localhost ~ # restart fwupd
   ```

   This method is suitable for developers only who are testing the new FW and not recommended for other purposes .

   Warning: This should never be done on production machines.

2. Use standalone `fwupdtool` tool to update the device with development CAB file:

   ```
   localhost ~ # fwupdtool install --allow-reinstall --allow-older ColorHug-1.2.6.cab -
   ↪-json
   ```

15.4.4 Upgrade the device through internal repository

By default Chromium OS has a local vendors repository enabled (see `/etc/fwupd/remotes.d/vendor-directory.conf`), so any CAB file placed into the local directory `/usr/share/fwupd/remotes.d/vendor/firmware` will be automatically detected and could be used for the device upgrade or downgrade:

```
localhost ~ # cp ColorHug-1.2.6.cab /usr/share/fwupd/remotes.d/vendor/firmware/
```

Note: Please check the *Updates with LVFS* section below how to update or downgrade the firmware with the GUI or CLI.

15.5 LVFS

Chromium OS does not use the Linux Vendor Firmware Service (LVFS) for secure updates directly.

Instead Google is using its own mirror copied from the LVFS stable remote.

That's why we have to add LVFS remotes to the Chrome OS device during the testing.

15.5.1 Account

Please request the access to LVFS portal according the https://lvfs.readthedocs.io/en/latest/apply.html

15.5.2 LVFS: remotes

There are 4 LVFS remotes available for the CAB file uploading:

- **private** – should be the initial remote for uploading, the CAB file with FW uploaded to **private** could be accessible only via direct link
- **embargo** – remote with non-public catalog of FWs available for Vendor only; used during development and QA testing. The remote may be added to the Chrome OS just like a common remote, so all `fwupd` functionality is available for testing.

  Should be used for testing by Vendor before giving access to FW to end-users.
- **testing** – this remote is generally available for end users, encouraged enough to deal with the potential risk of the freshest FW version.
- **stable** – the main remote with released FW considered as good enough for the mass market.

  This remote is enabled by default on Chrome OS and Linux systems with installed `fwupd`.

15.5.3 CAB file repacking

The uploaded CAB file would be repacked on the LVFS side:

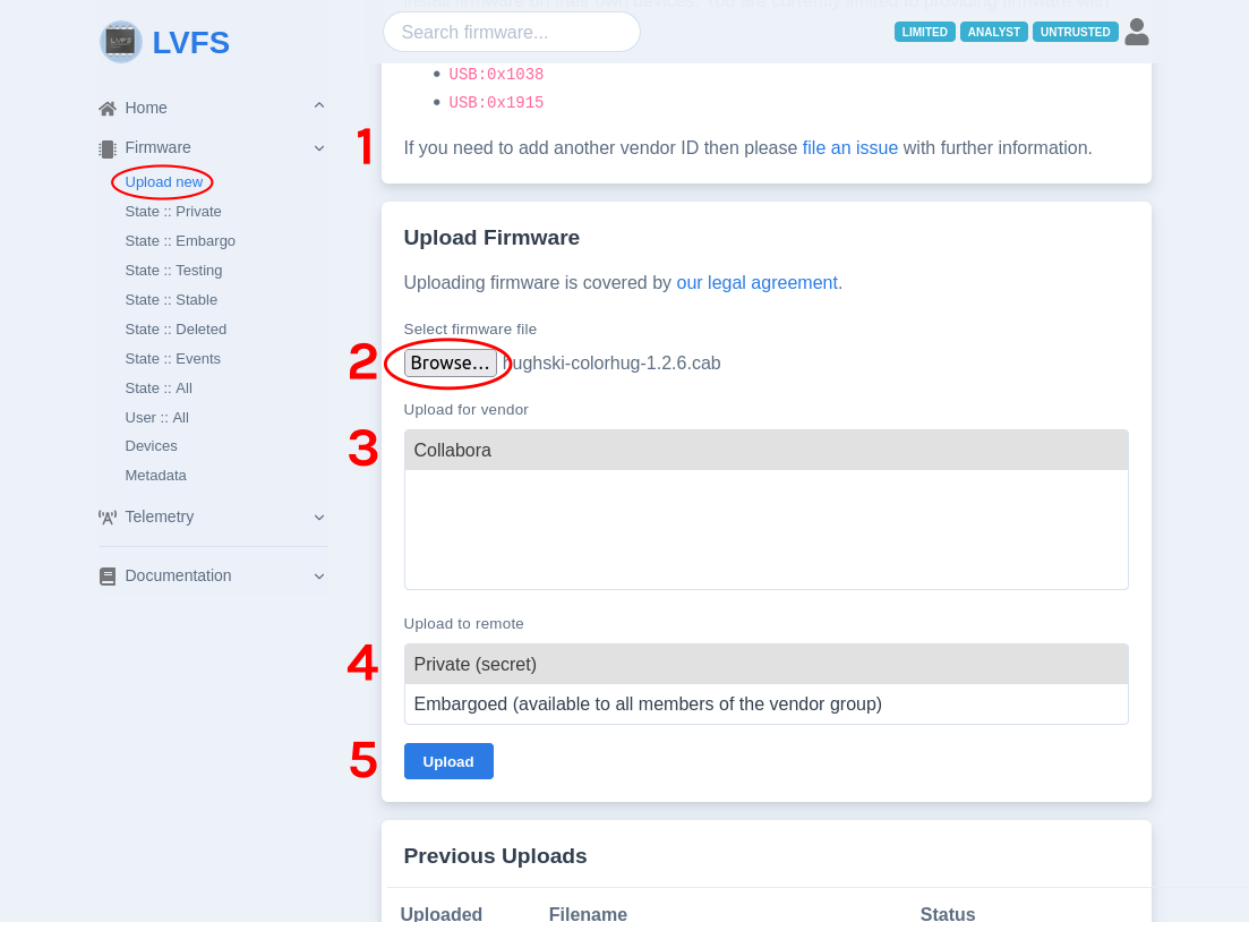
- The metadata is validated during this step.
- Missing parts would be added into metadata if not provided by metadata XML file, checksums for instance.
- The signature will be added to FW and metadata. Only signed CAB files are trusted by default on end-user devices.

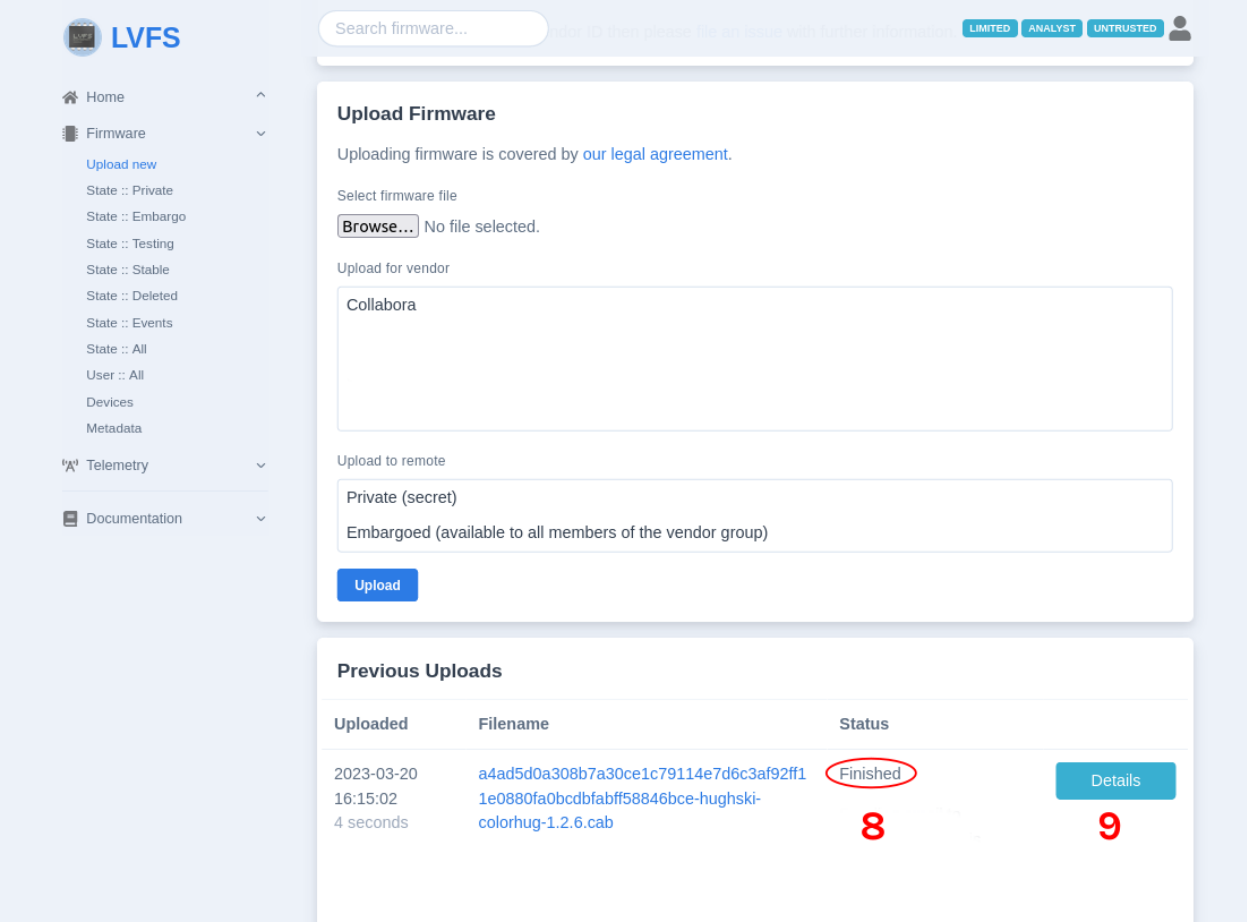
Any change of the metadata in the internal editor will cause repackage and resignation of the CAB file, so the new file will be generated.

15.5.4 LVFS: private remote

Upload the FW CAB file to LVFS private remote

1. Go to section “Firmware” -> “Upload new”
2. Select the generated CAB file from your host
3. Choose your vendor name (Collabora in example)
4. Choose “Private” remote
5. Press “Upload”
6. The page will be refreshed
7. Scroll down to the “Previous uploads” section, you should see the confirmation with the date, filename and the status of uploaded file
8. Refresh the web page until “Complete” status
9. Press “Details” button

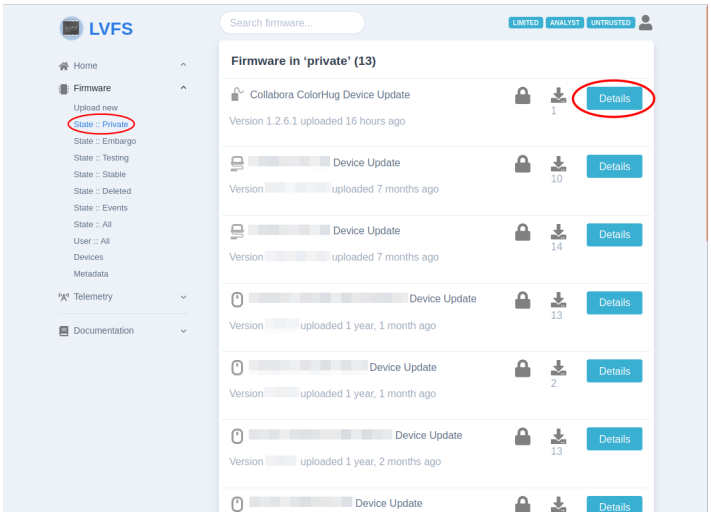

LVFS
Search firmware...
LIMITED
ANALYST
UNTRUSTED
Home
Firmware
Upload new
State :: Private
State :: Embargo
State :: Testing
State :: Stable
State :: Deleted
State :: Events
State :: All
User :: All
Devices
Metadata
Telemetry
Documentation
USB:0x1038
USB:0x1915
1
If you need to add another vendor ID then please file an issue with further information.
Upload Firmware
Uploading firmware is covered by our legal agreement.
Select firmware file
2
Browse...
ughski-colorhug-1.2.6.cab
Upload for vendor
3
Collabora
Upload to remote
4
Private (secret)
Embargoed (available to all members of the vendor group)
5
Upload
Previous Uploads
Uploaded
Filename
Status


LVFS
Search firmware...
LIMITED
ANALYST
UNTRUSTED
Home
Firmware
Upload new
State :: Private
State :: Embargo
State :: Testing
State :: Stable
State :: Deleted
State :: Events
State :: All
User :: All
Devices
Metadata
Telemetry
Documentation
Upload Firmware
Uploading firmware is covered by our legal agreement.
Select firmware file
Browse...
No file selected.
Upload for vendor
Collabora
Upload to remote
Private (secret)
Embargoed (available to all members of the vendor group)
Upload
Previous Uploads
Uploaded
Filename
Status
2023-03-20 16:15:02
4 seconds
a4ad5d0a308b7a30ce1c79114e7d6c3af92ff1 1e0880fa0bcdbfabff58846bce-hughski-colorhug-1.2.6.cab
Finished
Details
8
9

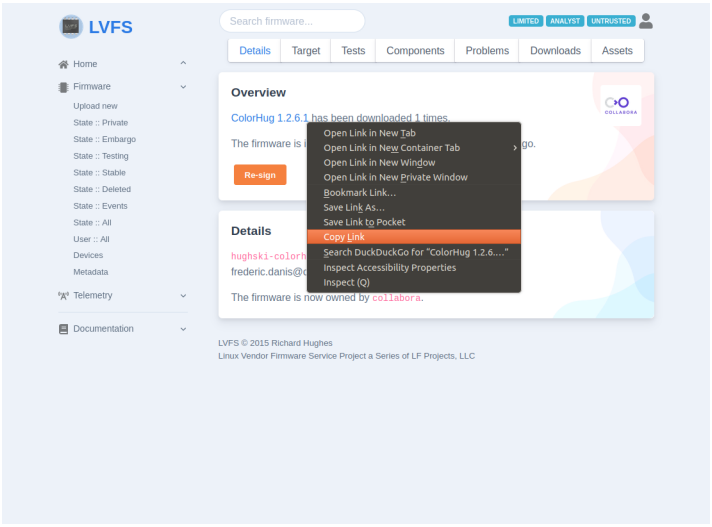
Install the FW CAB file from the private remote

The private remote doesn't provide the catalog with uploaded files. Instead you have to go to the CAB file you are interested in and get its URL for the direct download and install.

1. Choose the needed firmware from private repo ("Firmware" -> "State::Private") and press the button details.



2. In the "Details" tab you will see the "Overview" block with the name of the uploaded firmware. The right click (or long press) will give you the URL of the signed CAB file.



3. Gain access to the Chrome OS device terminal as described in the *Access to ChromeBook* section.
4. Via terminal install the CAB file from the LVFS by copied URL:

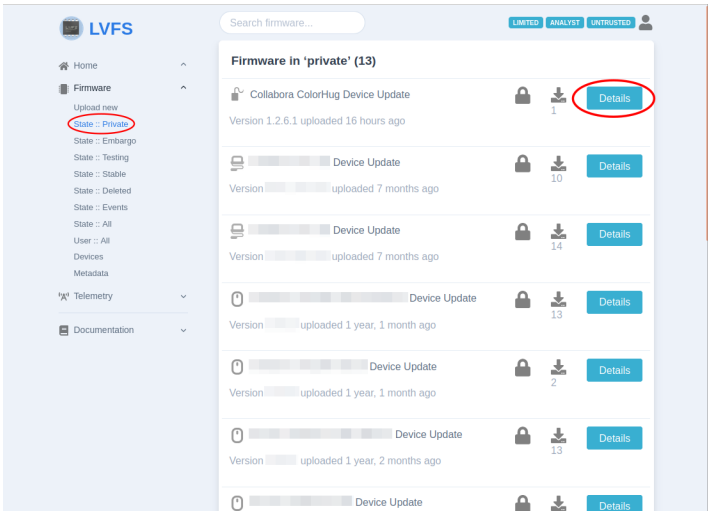
```
# fwupdmgr install https://fwupd.org/downloads/
↪c6fbb716abbb204d98f12edf1f146b6406f39b1eade741b353c15a86f5da8278-hughski-colorhug-
↪1.2.6.cab
Waiting...                [****************************************] Less than one␣
↪minute remaining...
Successfully installed firmware
```

15.5.5 LVFS: Embargo remote

The embargoed remote provides the catalog of available firmwares but it is restricted to vendor, so only the vendor is able to use it or share for 3-rd parties.

Move the FW CAB file to embargoed remote

1. Choose the needed firmware from private repo ("Firmware" -> "State::Private") and press the button details.



2. In the "Target" tab you will see the list of available remotes.

   Press "Move here" button for "Embargo" remote:

3. On the updated screen you need to scroll down the web page to "History" block to see the confirmation if the FW has been moved into the new remote:

Enable the embargoed remote

This is the one-time action needed for enabling the Embargo remote on the testing Chrome OS system.

1. Check on Chrome OS device if the remote is not available and not enabled already:

   ```
   # fwupdmgr get-remotes
   ```

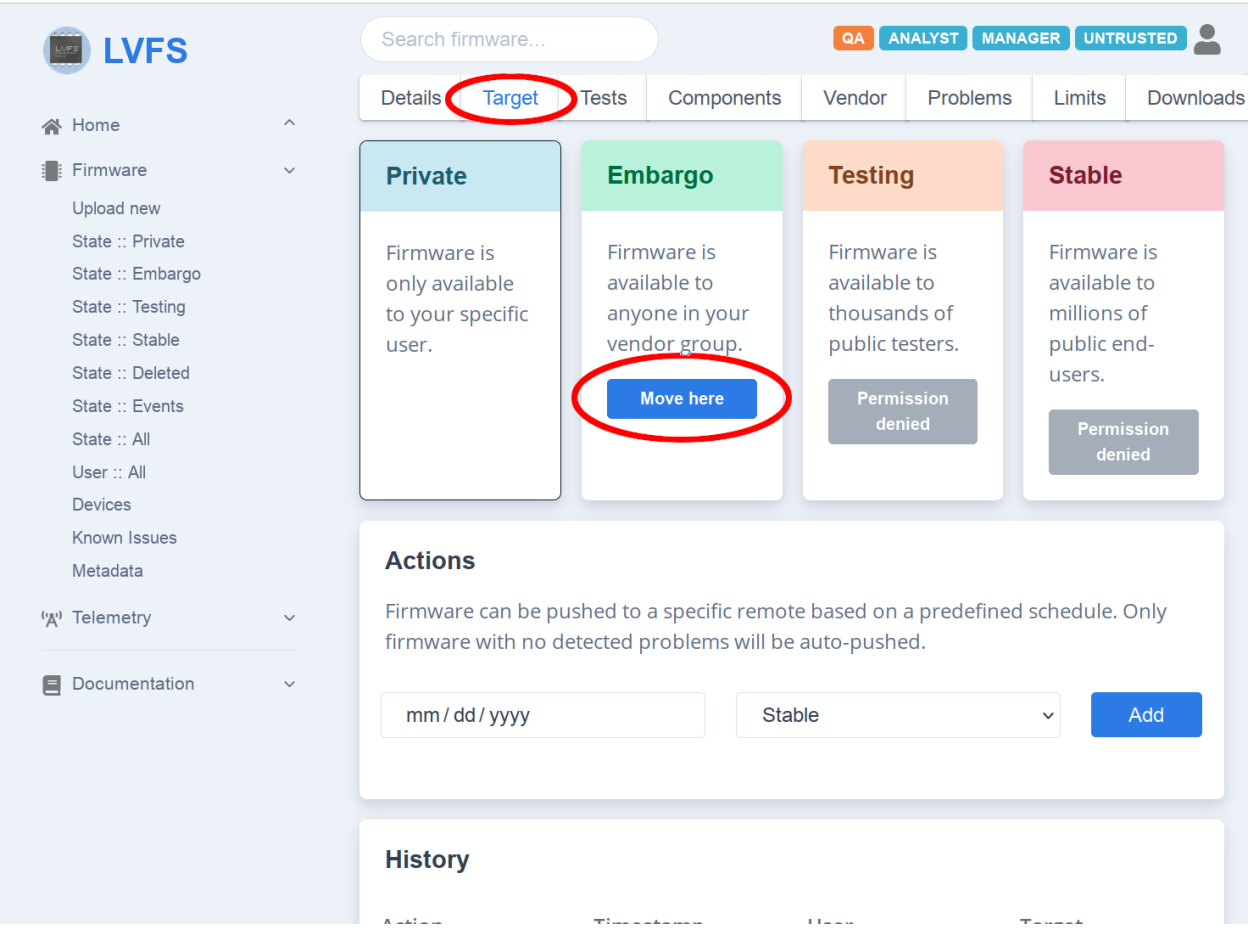
   If there is the remote named `Embargoed for collabora` or similar and it is enabled (check the key `Enabled: true`), probably the remote is configured already and you may proceed with *Install the FW CAB file from the embargo remote* .

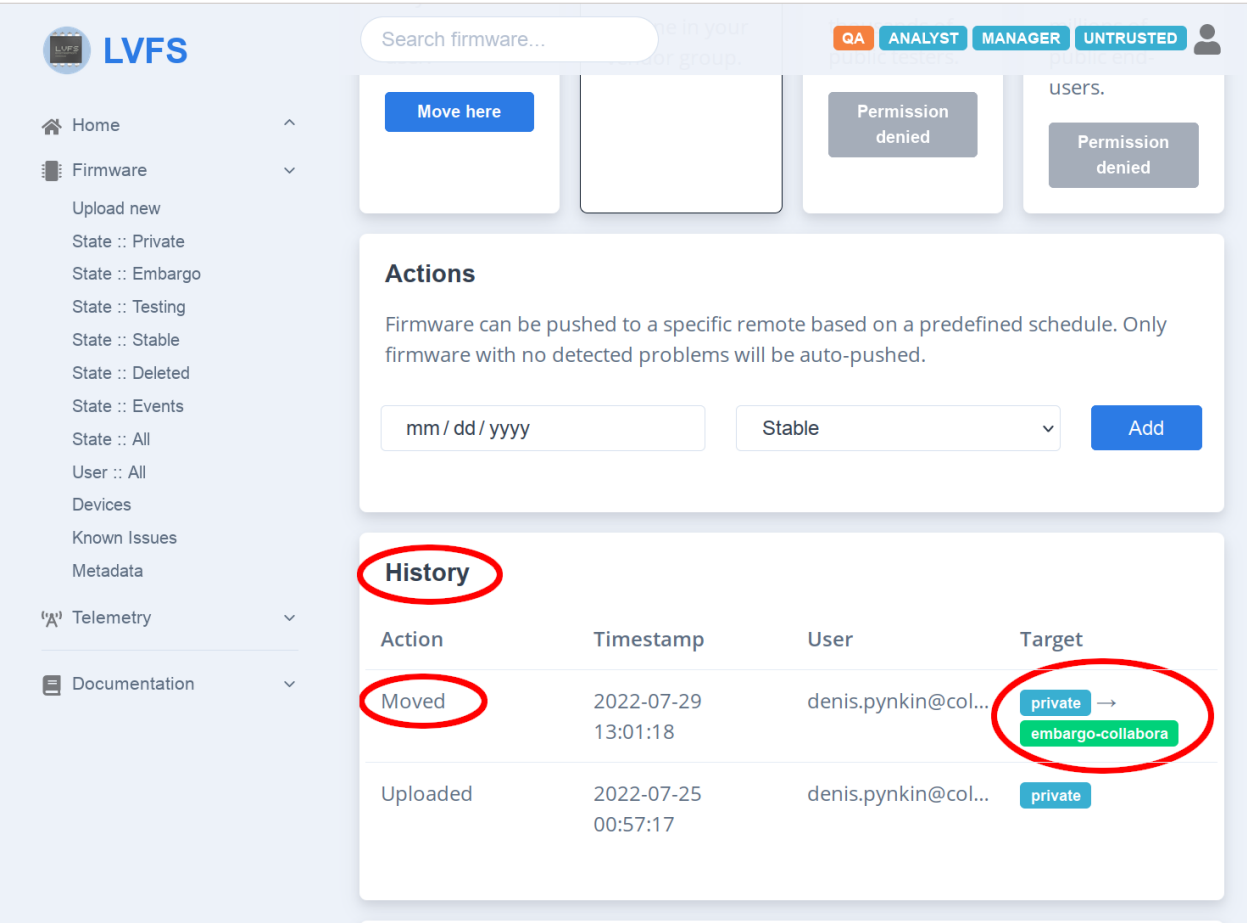
2. On LVFS site find the section "Metadata" – it would contain links to the recent versions of metadata for Embargo, Testing and Stable repositories.

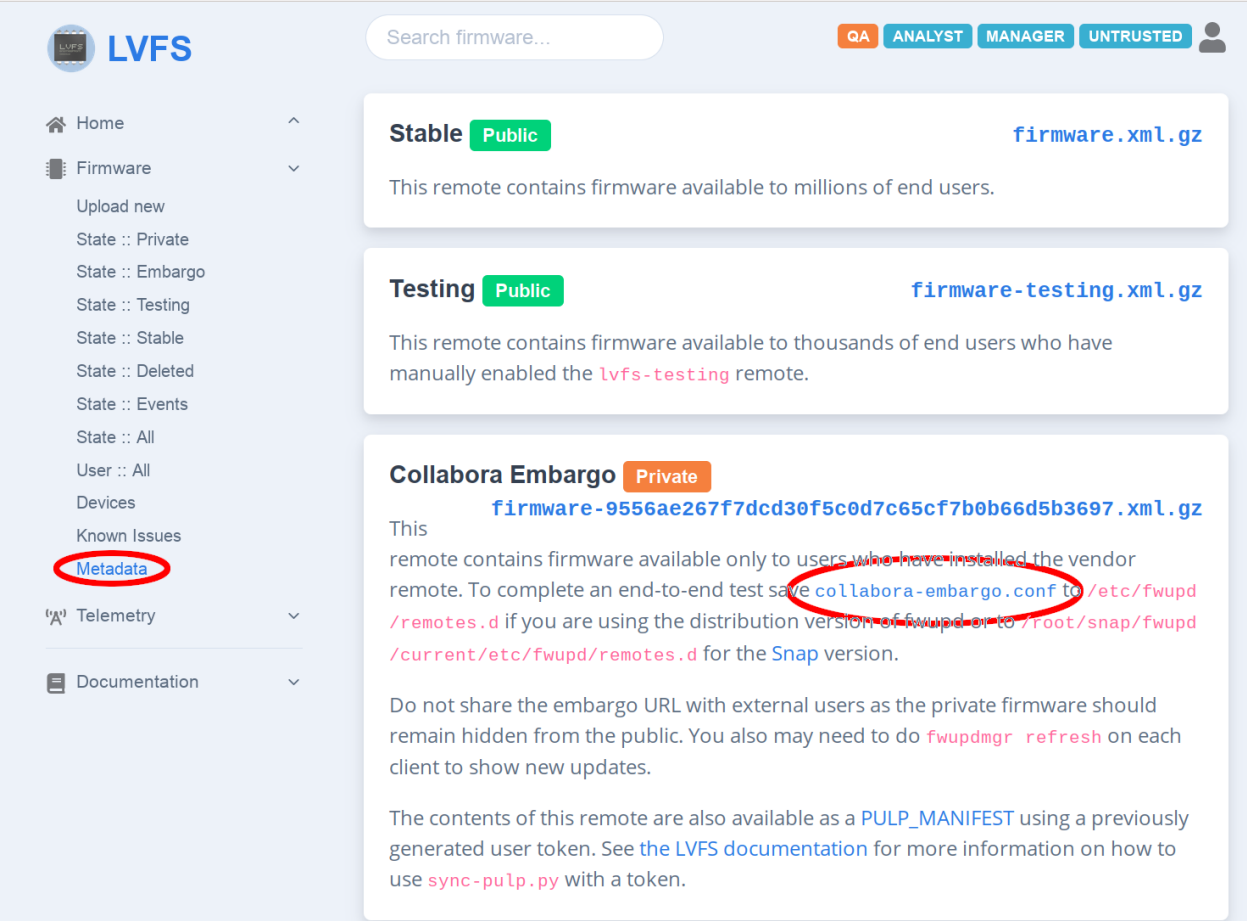
   However we are interesting in the configuration file for the Embargo remote of your company (`collabora-embargo.conf` in the example)

3. Download the configuration file for the embargoed remote by clicking the name.

   Please keep in mind – this is the private file accessible for authorized users only!


LVFS
Search firmware...
QA
ANALYST
MANAGER
UNTRUSTED
Home
Firmware
Upload new
State :: Private
State :: Embargo
State :: Testing
State :: Stable
State :: Deleted
State :: Events
State :: All
User :: All
Devices
Known Issues
Metadata
Telemetry
Documentation
Details
Target
Tests
Components
Vendor
Problems
Limits
Downloads
Private
Firmware is only available to your specific user.
Embargo
Firmware is available to anyone in your vendor group.
Move here
Testing
Firmware is available to thousands of public testers.
Permission denied
Stable
Firmware is available to millions of public end-users.
Permission denied
Actions
Firmware can be pushed to a specific remote based on a predefined schedule. Only firmware with no detected problems will be auto-pushed.
mm / dd / yyyy
Stable
Add
History


LVFS
Search firmware...
QA
ANALYST
MANAGER
UNTRUSTED
Home
Firmware
Upload new
State :: Private
State :: Embargo
State :: Testing
State :: Stable
State :: Deleted
State :: Events
State :: All
User :: All
Devices
Known Issues
Metadata
Telemetry
Documentation
Move here
Permission denied
users.
Permission denied
Actions
Firmware can be pushed to a specific remote based on a predefined schedule. Only firmware with no detected problems will be auto-pushed.
mm / dd / yyyy
Stable
Add
History
Action
Timestamp
User
Target
Moved
2022-07-29 13:01:18
denis.pynkin@col...
private
embargo-collabora
Uploaded
2022-07-25 00:57:17
denis.pynkin@col...
private


LVFS
Search firmware...
QA
ANALYST
MANAGER
UNTRUSTED
Home
Firmware
Upload new
State :: Private
State :: Embargo
State :: Testing
State :: Stable
State :: Deleted
State :: Events
State :: All
User :: All
Devices
Known Issues
Metadata
Telemetry
Documentation
Stable
Public
firmware.xml.gz
This remote contains firmware available to millions of end users.
Testing
Public
firmware-testing.xml.gz
This remote contains firmware available to thousands of end users who have manually enabled the lvfs-testing remote.
Collabora Embargo
Private
firmware-9556ae267f7dcd30f5c0d7c65cf7b0b66d5b3697.xml.gz
This remote contains firmware available only to users who have installed the vendor remote. To complete an end-to-end test save collabora-embargo.conf to /etc/fwupd/remotes.d if you are using the distribution version of fwupd or to /root/snap/fwupd/current/etc/fwupd/remotes.d for the Snap version.
Do not share the embargo URL with external users as the private firmware should remain hidden from the public. You also may need to do fwupdmgr refresh on each client to show new updates.
The contents of this remote are also available as a PULP_MANIFEST using a previously generated user token. See the LVFS documentation for more information on how to use sync-pulp.py with a token.

4. You need to copy the downloaded file to the ChromeBook with any method available for you (mass-media device, ssh and others).

   To copy the file from the workstation with `ssh`, for instance:

   ```
   scp collabora-embargo.conf root@192.168.1.115:~/
   ```

   Please use the correct file name and IP address of the target device!

5. On Chrome OS device you need to copy the file to configuration directory for `fwupd` daemon:

   ```
   # cp -v collabora-embargo.conf /etc/fwupd/remotes.d/
   ```

6. Restart the `fwupd` service on ChromeBook:

   ```
   # restart fwupd
   fwupd start/running, process 22697
   ```

Install the FW CAB file from the embargo remote

1. Refresh the metadata for embargoed remote:

   ```
   # fwupdmgr refresh
   Updating collabora-embargo
   Downloading...             [****************************************]
   Downloading...             [****************************************]
   Downloading...             [****************************************]
   Decompressing...           [****************************************]
   Successfully downloaded new metadata: 6 local devices supported
   ```

2. Update with the FW available from Embargo remote:

   ```
   # fwupdmgr update
   Devices with no available firmware updates:
   • DUTA42
   • Generic Billboard Device

    Upgrade ColorHug from 1.2.5 to 1.2.6?

    This release fixes prevents the firmware returning an error when the
    remote SHA1 hash was never sent.

    ColorHug and all connected devices may not be usable while updating.

   Perform operation? [Y|n]: Y
   Downloading...             [****************************************]
   Downloading...             [****************************************] Less than one␣
    ↪minute remaining...
   Downloading...             [****************************************]
   Decompressing...           [****************************************]
   Authenticating...          [****************************************]
   Waiting...                 [****************************************] Less than one␣
    ↪minute remaining...
   Successfully installed firmware
   ```

15.5.6 LVFS: Testing remote

The Testing remote provides the catalog of firmwares for public access, however only users explicitly enabled this remote will have access to published FWs.

Move the FW CAB file to testing remote

LVFS side is similar to *Move the FW CAB file to embargoed remote* , but the target is "Testing remote" and no need to download the file with remote configuration.

Enable testing remote

This is the one-time action needed for enabling the Testing remote for the Chrome OS system.

By default the Testing remote is already configured on Chrome OS but not enabled.

1. Check on Chrome OS device if the remote named `Linux Vendor Firmware Service (testing)` is not enabled already:

   ```
   # fwupdmgr get-remotes
   ```

   Check the key `Enabled:` and if it is set to `true` please proceed with *Install the FW CAB file with the testing remote*

2. To enable the testing remote, use text editor to edit configuration file to replace `Enabled=false` by `Enabled=true` in `/etc/fwupd/remotes.d/lvfs-testing.conf` or simply run the command below:

   ```
   # sed -i -e "s/^Enabled=false/Enabled=true/" /etc/fwupd/remotes.d/lvfs-testing.conf
   ```

3. Restart the `fwupd` daemon:

   ```
   # restart fwupd
   fwupd start/running, process 9841
   ```

Install the FW CAB file with the testing remote

1. Refresh the metadata for testing remote:

   ```
   #  fwupdmgr refresh
   Updating lvfs-testing
   Downloading...             [****************************************]
   Downloading...             [****************************************]
   Successfully downloaded new metadata: 2 local devices supported
   ```

2. Update with the latest available FW from the testing remote:

   ```
   #  fwupdmgr update
   Devices with no available firmware updates:
   • DUTA42
   • Generic Billboard Device

    Upgrade ColorHug from 1.2.5 to 1.2.6?
   ```

(continues on next page)

(continued from previous page)

```
 This release fixes prevents the firmware returning an error when the
 remote SHA1 hash was never sent.

 ColorHug and all connected devices may not be usable while updating.

Perform operation? [Y|n]: Y
Downloading...             [****************************************]
Downloading...             [****************************************]
Authenticating...          [****************************************]
Waiting...                 [****************************************] Less than one␣
↪minute remaining...
Successfully installed firmware
```

15.5.7 LVFS: Stable remote

The Stable remote makes the FW available for all users of `fwupd` over the World running on different operating systems (primarily Linux).

For the Chrome OS this repository isn't enabled by default, so it is needed to enable Stable remote explicitly. This makes the stable remote similar to the testing one for the Chrome OS.

Google's team is keeping its own mirror of LVFS, so the FW from the LVFS Stable remote will be published for Chrome OS users only after some time.

Move the FW CAB file to stable remote

LVFS side is similar to *Move the FW CAB file to embargoed remote*, but the target is "Stable remote" and no need to download the file with remote configuration.

Enable stable remote

This is the one-time action needed for enabling the stable remote for the Chrome OS system.

By default the stable remote is already configured on Chrome OS but not enabled.

1. Check on Chrome OS device if the remote named `Linux Vendor Firmware Service` is not enabled already:

   ```
   # fwupdmgr get-remotes
   ```

   Check the key `Enabled:` and if it is set to `true` please proceed with

2. Enable the stable remote, use text editor to edit configuration file to replace `Enabled=false` by `Enabled=true` in `/etc/fwupd/remotes.d/lvfs.conf` or simply run the command below:

   ```
   # sed -i -e "s/^Enabled=false/Enabled=true/" /etc/fwupd/remotes.d/lvfs.conf
   ```

3. Restart the `fwupd` daemon:

   ```
   # restart fwupd
   fwupd start/running, process 20199
   ```

Install the FW CAB file with the stable remote

1. Refresh the metadata for stable remote:

```
# fwupdmgr refresh
Updating lvfs
Downloading...            [****************************************]
Downloading...            [****************************************]
Decompressing...          [****************************************]
Successfully downloaded new metadata: 0 local devices supported
```

2. Update with the latest available FW from the stable remote :

```
# fwupdmgr update
```

15.5.8 Persistent revisions

Mark all FW versions uploaded to LVFS and used for automatic tests as persistent ("preserve" button in LVFS).

This is needed to keep the uploaded versions used for tests and prevent from purging on the LVFS side.

15.5.9 Signed Reports

The firmware testing is described in documentation: https://lvfs.readthedocs.io/en/latest/testing.html#signed-reports .

After each update the `fwupdmgr` client tools allow the end user to submit a "report" which is used by the firmware owner to validate the firmware deployment is correct. Any failures can be analyzed and patterns found and the metadata can be fixed. For instance, the failures might indicate that the required `fwupd` version needs to be raised to a higher value, or that the update requires a specific bootloader version.

It is expected that only F/W having signed reports would be automatically copied into the LVFS mirror for Chrome OS. To do this, the user must either upload the certificate from each machine used for testing, or hardcode a user token.

Uploading a signed report on ChromeOS is slightly different than on regular Linux distributions as `/etc` is immutable and cannot be changed. By creating a file `/var/lib/fwupd/remotes.d/lvfs.conf` in a mutable directory the fwupd daemon will reload the information and disregard the immutable `/etc/fwupd/remotes.d/lvfs.conf` contents.

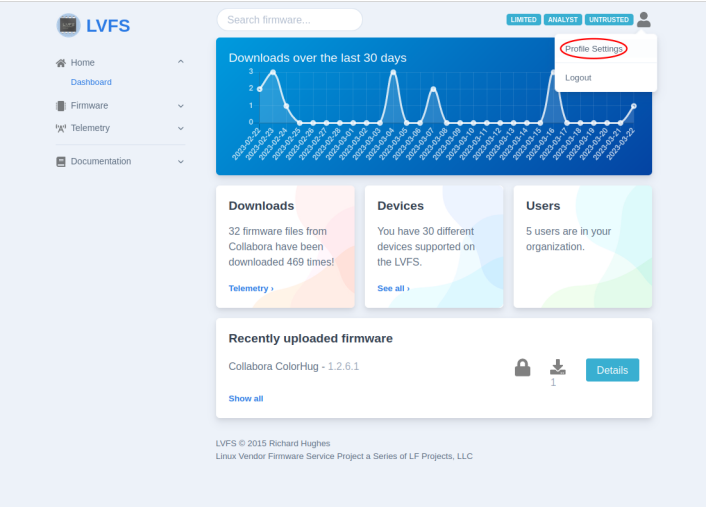
```
[fwupd Remote]
Enabled=true
Title=Linux Vendor Firmware Service
MetadataURI=https://cdn.fwupd.org/downloads/firmware-EMBARGO_HASH.xml.xz
ReportURI=https://fwupd.org/lvfs/firmware/report
Username=THE_USERNAME
Password=THE_TOKEN
```

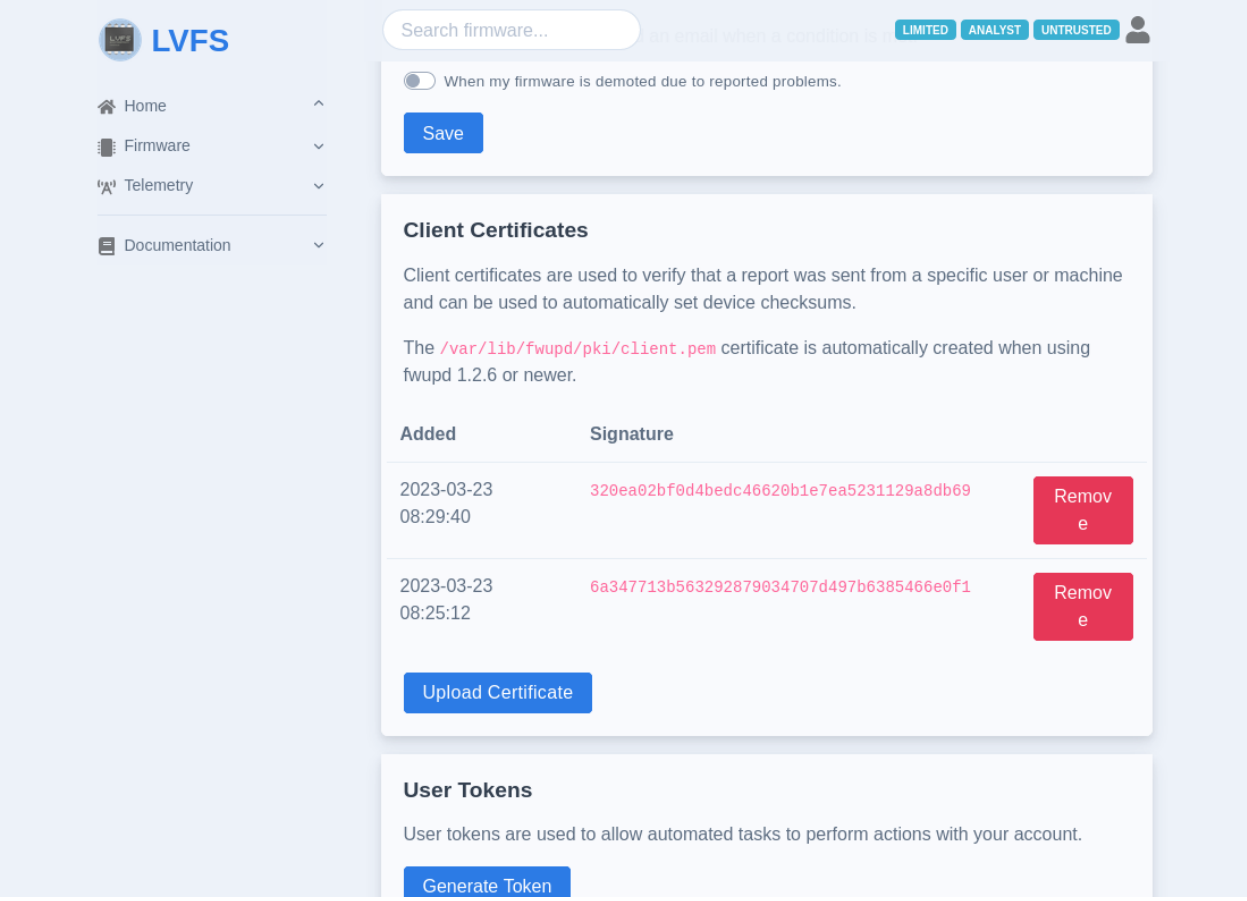
Where `EMBARGO_HASH` is the hash found in https://fwupd.org/lvfs/metadata/

The `THE_USERNAME` is your LVFS account email and `THE_TOKEN` is the token generated in https://fwupd.org/lvfs/profile

To use a host certificate rather than hardcoding `Username` and `Password`:

1. Go to the "Profile Settings"
2. Upload the `fwupd` certificate


LVFS
Search firmware...
LIMITED
ANALYST
UNTRUSTED
Home
Dashboard
Firmware
Telemetry
Documentation
Downloads over the last 30 days
Profile Settings
Logout
Downloads
32 firmware files from Collabora have been downloaded 469 times!
Telemetry ›
Devices
You have 30 different devices supported on the LVFS.
See all ›
Users
5 users are in your organization.
Recently uploaded firmware
Collabora ColorHug - 1.2.6.1
Details
Show all
LVFS © 2015 Richard Hughes
Linux Vendor Firmware Service Project a Series of LF Projects, LLC


LVFS
Search firmware...
LIMITED
ANALYST
UNTRUSTED
Home
Firmware
Telemetry
Documentation
When my firmware is demoted due to reported problems.
Save
Client Certificates
Client certificates are used to verify that a report was sent from a specific user or machine and can be used to automatically set device checksums.
The /var/lib/fwupd/pki/client.pem certificate is automatically created when using fwupd 1.2.6 or newer.
Added
Signature
2023-03-23 08:29:40
320ea02bf0d4bedc46620b1e7ea5231129a8db69
Remove
2023-03-23 08:25:12
6a347713b563292879034707d497b6385466e0f1
Remove
Upload Certificate
User Tokens
User tokens are used to allow automated tasks to perform actions with your account.
Generate Token

> **Warning:** The metadata must now be downloaded from the embargo remote using `fwupdmgr refresh` otherwise the message `has no RemoteID` will be seen using `fwupdmgr report-history --verbose`

The user can then upload reports to the LVFS in a trusted way by signing the report:

```
$ fwupdmgr update # or fwupdmgr install foo.cab
# ...reboot if required...
```

If using a `Username` and `Password`, you **must** use:

```
$ fwupdmgr report-history
```

If using a host certificate, you **must** use:

```
$ fwupdmgr report-history --sign
```

15.5.10 USB device update record

Since `fwupd` version 1.8.11 it is possible to record the firmware update of the USB devices. This can be used for a failing update to allow the plugin developer to replay the update for debugging purposes .

Device record

1. *Check the device availability and version*

```
# fwupdmgr get-devices
```

2. Enable device emulation support, use text editor to edit configuration file to replace `AllowEmulation=false` by `AllowEmulation=true` in `/etc/fwupd/daemon.conf` or simply run the command below:

```
# sed -i -e "s/^ AllowEmulation =false/ AllowEmulation =true/" /etc/fwupd/daemon.
↪conf
```

3. Restart the `fwupd` daemon:

```
# restart fwupd
fwupd start/running, process 20199
```

4. Copy the "Device ID:" or "GUID" value from the target device, for example:

```
├─ColorHug:
        Device ID:          23cf6368c14a875f74c38a5a423518f38d8abbbc
        Summary:            An open source display colorimeter
        Current version:    1.2.5
        Vendor:             Hughski Ltd. (USB:0x273F)
        Install Duration:   8 seconds
        GUIDs:              40338ceb-b966-4eae-adae-9c32edfcc484
                            afdcc391-6c33-5914-b4d2-b4dd71fe9c5a
```

   The Device ID is unique per device!!!

5. Register the device for recording using the "Device ID":

```
# fwupdmgr emulation-tag 23cf6368c14a875f74c38a5a423518f38d8abbbc
```

or the "GUID":

```
# fwupdmgr emulation-tag 40338ceb-b966-4eae-adae-9c32edfcc484
```

6. Remove and re-insert the device.
7. Upgrade the device to the newer version:

```
# fwupdmgr update
```

8. Save the records to an emulation file:

```
# fwupdmgr emulation-save colorhug.zip
```

9. Unregister the device:

```
#  fwupdmgr emulation-untag 23cf6368c14a875f74c38a5a423518f38d8abbbc
```

10. Disable device emulation support, use text editor to edit configuration file to replace `AllowEmulation=true` by `AllowEmulation=false` in `/etc/fwupd/daemon.conf` or simply run the command below:

```
# sed -i -e "s/^ AllowEmulation =true/ AllowEmulation =false/" /etc/fwupd/daemon.
↪conf
```

11. Restart the `fwupd` daemon:

```
#  restart fwupd
fwupd start/running, process 20199
```

The emulation file can now be sent to the plugin developer.

Device emulation

1. Enable device emulation support, use text editor to edit configuration file to replace `AllowEmulation=false` by `AllowEmulation=true` in `/etc/fwupd/daemon.conf` or simply run the command below:

```
# sed -i -e "s/^ AllowEmulation =false/ AllowEmulation =true/" /etc/fwupd/daemon.
↪conf
```

2. Restart the `fwupd` daemon:

```
 # restart fwupd
fwupd start/running, process 20199
```

3. Load the emulated device:

```
# fwupdmgr emulation-load colorhug.zip
```

4. *Check the device availability and version*

```
localhost ~ # fwupdmgr get-devices
Nightfury
```

(continues on next page)

(continued from previous page)

```
│
├─ColorHug:
│         Device ID:          d9c9e0eb29c6f35160d400949c14db42f473f4d4
│         Summary:            An open source display colorimeter
│         Current version:    1.2.5
│         Vendor:             Hughski Ltd. (USB:0x273F)
│         Install Duration:   8 seconds
│         GUIDs:              40338ceb-b966-4eae-adae-9c32edfcc484
│                             afdcc391-6c33-5914-b4d2-b4dd71fe9c5a
│                             6bc5ff27-d631-5660-9991-6d24954c6f90 ← USB\VID_273F&
↪PID_1001
│                             4841a9e4-e5c8-5107-a83e-d6c6d9c21248 ← USB\VID_273F&
↪PID_1001&REV_0002
│         Device Flags:       • Updatable
│                             • Supported on remote server
│                             • Device can recover flash failures
│                             • Unsigned Payload
│                             • Emulated
```

5. Upgrade the device to the newer version:

```
# fwupdmgr update
```

6. Disable device emulation support, use text editor to edit configuration file to replace `AllowEmulation=true` by `AllowEmulation=false` in `/etc/fwupd/daemon.conf` or simply run the command below:

```
# sed -i -e "s/^ AllowEmulation =true/ AllowEmulation =false/"
/etc/fwupd/daemon.conf
```

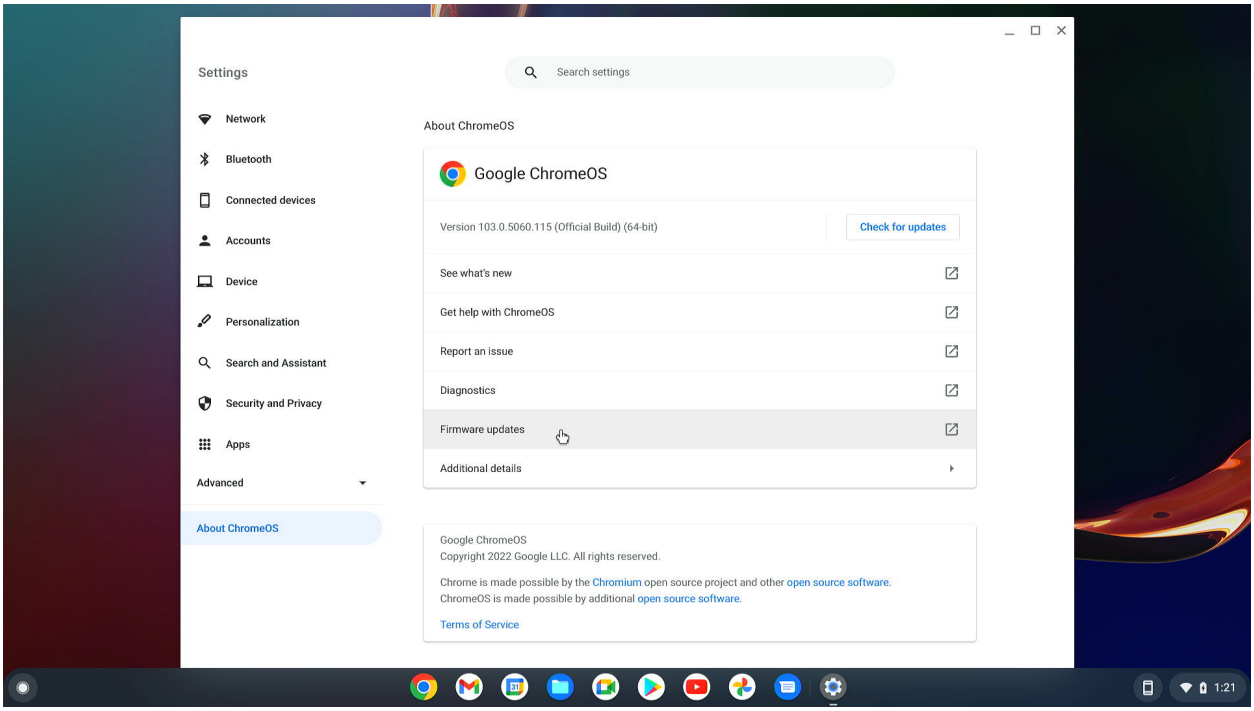
7. Restart the `fwupd` daemon:

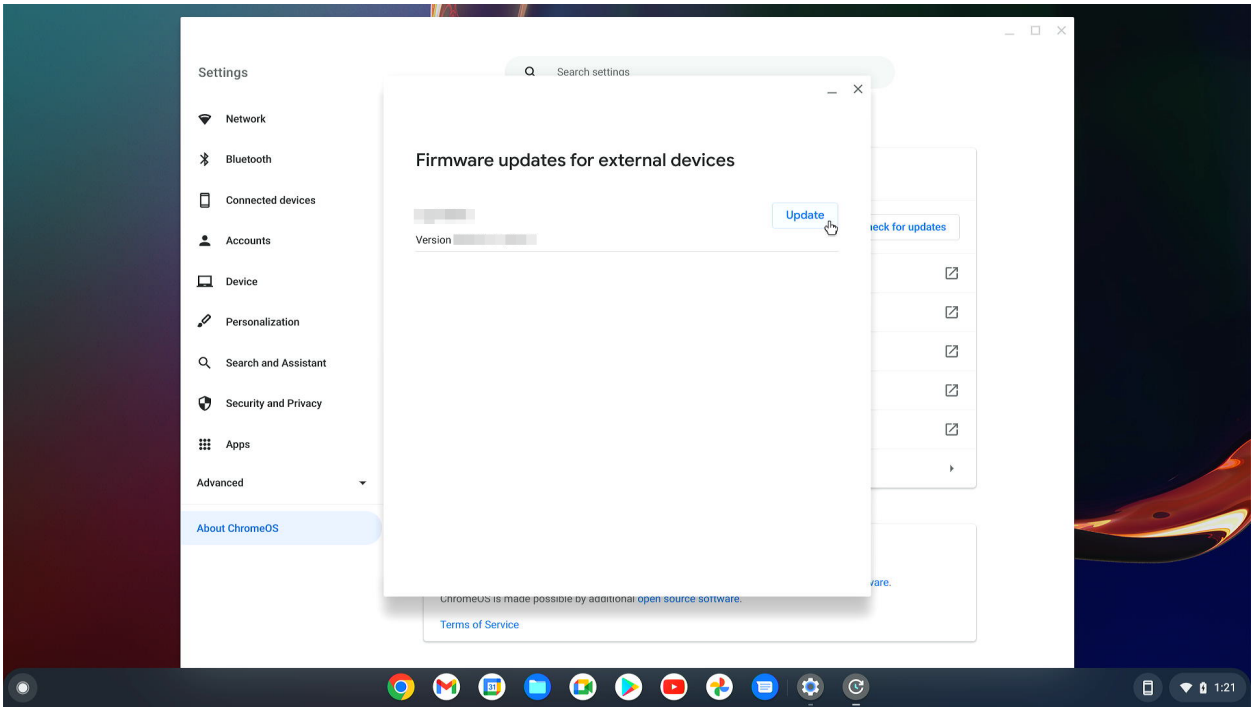
```
# restart fwupd
fwupd start/running, process 20199
```

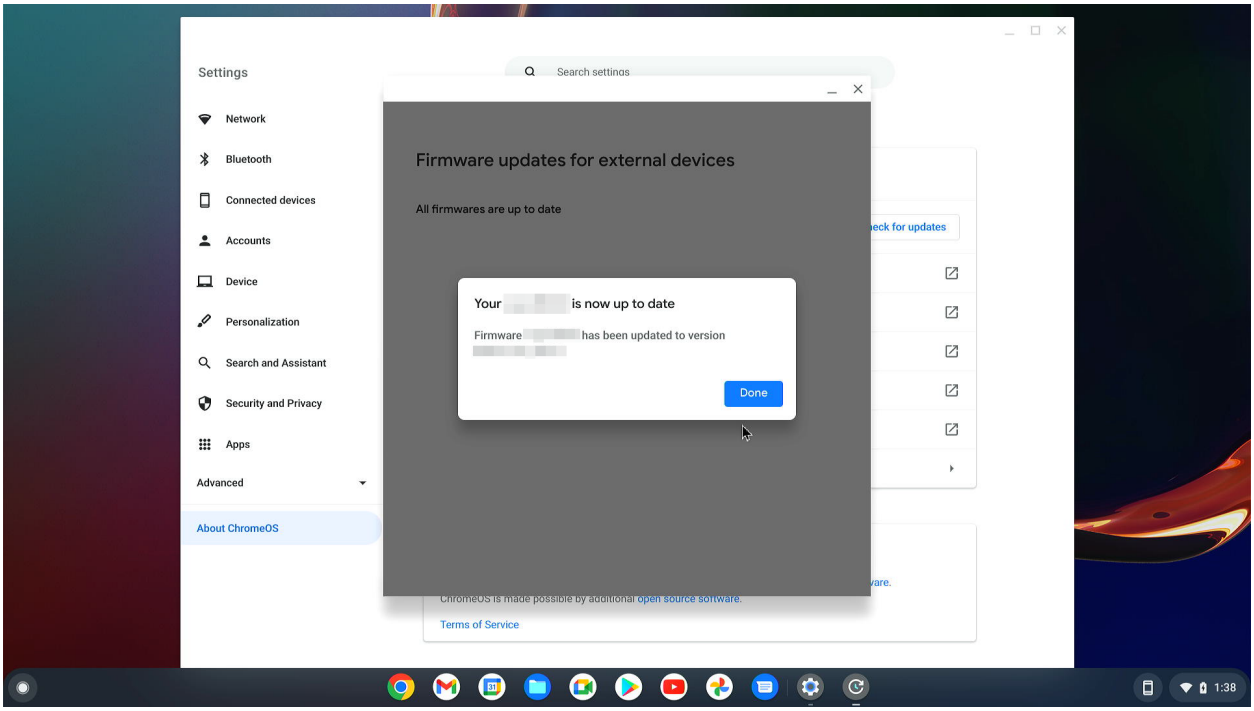
15.6 Updates with LVFS

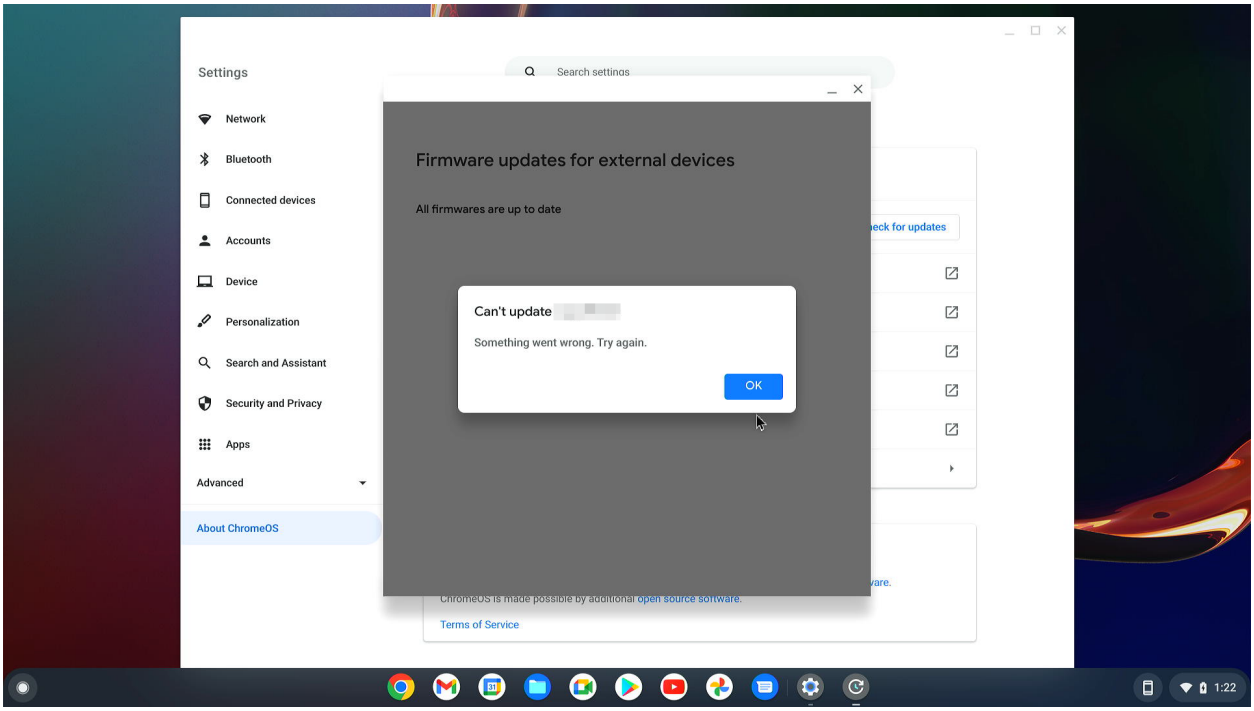
15.6.1 Update with GUI

1. Go to “Settings”
2. Press “About Chrome OS” and if the update is available – find the button “Firmware updates” and press it:
3. If the update for the connected device is available you will see it in the list of pop-up window and will be able to update:
4. In case of successful update you should to see the appropriate message:
5. If something goes wrong during update, the error message should appeared:


Settings
Search settings
Network
Bluetooth
Connected devices
Accounts
Device
Personalization
Search and Assistant
Security and Privacy
Apps
Advanced
About ChromeOS
About ChromeOS
Google ChromeOS
Version 103.0.5060.115 (Official Build) (64-bit)
Check for updates
See what's new
Get help with ChromeOS
Report an issue
Diagnostics
Firmware updates
Additional details
Google ChromeOS
Copyright 2022 Google LLC. All rights reserved.
Chrome is made possible by the Chromium open source project and other open source software.
ChromeOS is made possible by additional open source software.
Terms of Service
1:21


Settings
Search settings
Network
Bluetooth
Connected devices
Accounts
Device
Personalization
Search and Assistant
Security and Privacy
Apps
Advanced
About ChromeOS
Firmware updates for external devices
Update
Version
Terms of Service
1:21


Settings
Search settings
Network
Bluetooth
Connected devices
Accounts
Device
Personalization
Search and Assistant
Security and Privacy
Apps
Advanced
About ChromeOS
Firmware updates for external devices
All firmwares are up to date
Your is now up to date
Firmware has been updated to version
Done
Terms of Service
1:38


Settings
Search settings
Network
Bluetooth
Connected devices
Accounts
Device
Personalization
Search and Assistant
Security and Privacy
Apps
Advanced
About ChromeOS
Firmware updates for external devices
All firmwares are up to date
Can't update
Something went wrong. Try again.
OK
Terms of Service
1:22

15.6.2 Update with console

Check the device availability and version

Run the command `fwupdmgr get-devices` to gain the list of attached devices.

```
#  fwupdmgr get-devices
Nightfury
│
├─ColorHug:
│         Device ID:         23cf6368c14a875f74c38a5a423518f38d8abbbc
│         Summary:           An open source display colorimeter
│         Current version:   1.2.5
│         Vendor:            Hughski Ltd. (USB:0x273F)
│         Install Duration:  8 seconds
│         Update State:      Success
│         GUIDs:             40338ceb-b966-4eae-adae-9c32edfcc484
│                            afdcc391-6c33-5914-b4d2-b4dd71fe9c5a
│                            6bc5ff27-d631-5660-9991-6d24954c6f90 ← USB\VID_
↪273F&PID_1001
│                            4841a9e4-e5c8-5107-a83e-d6c6d9c21248 ← USB\VID_
↪273F&PID_1001&REV_0002
│         Device Flags:      • Updatable
│                            • Supported on remote server
│                            • Device can recover flash failures
│                            • Unsigned Payload
```

Update the selected device

It is possible to update the single selected device by Device ID listed in *Check the device availability* to the last available FW version:

```
# fwupdmgr update 23cf6368c14a875f74c38a5a423518f38d8abbbc
Devices with no available firmware updates:
• DUTA42
• Generic Billboard Device

 Upgrade ColorHug from 1.2.5 to 1.2.6?

 This release fixes prevents the firmware returning an error when the
 remote SHA1 hash was never sent.

 ColorHug and all connected devices may not be usable while updating.

Perform operation? [Y|n]:
Downloading...             [****************************************]
Downloading...             [****************************************]
Authenticating...          [****************************************]
Waiting...                 [****************************************]
Successfully installed firmware
```

Update all supported devices

If you want to update all supported devices attached to the ChromeBook, just run `fwupdmgr update` to apply any FW update available from the enabled remote:

```
#  fwupdmgr update
Devices with no available firmware updates:
• DUTA42
• Generic Billboard Device

 Upgrade ColorHug from 1.2.5 to 1.2.6?

 This release fixes prevents the firmware returning an error when the
 remote SHA1 hash was never sent.

 ColorHug and all connected devices may not be usable while updating.

Perform operation? [Y|n]: y
Downloading...             [****************************************]
Downloading...             [****************************************]
Decompressing...           [****************************************]
Authenticating...          [****************************************]
Waiting...                 [****************************************] Less than␣
↪one minute remaining...
Successfully installed firmware
```

Downgrade the FW version

If several versions of the FW are available from enabled remote(s) it is possible to perform downgrade to any available version lower than the current by selection via the proposed menu. If only the single version for downgrade is available you will need only to confirm the operation:

```
# fwupdmgr downgrade 23cf6368c14a875f74c38a5a423518f38d8abbbc
0.    Cancel
1.    1.2.5
2.    1.2.4
3.    1.2.3
4.    1.2.2
5.    1.2.0
Choose release [0-5]: 1

 Downgrade ColorHug from 1.2.6 to 1.2.5?

 This release fixes the firmware package to work with new versions of
 fwupd.

 ColorHug and all connected devices may not be usable while updating.

Perform operation? [Y|n]:
Downloading...             [****************************************]
Downloading...             [****************************************]
Downloading...             [****************************************]
```

(continues on next page)

(continued from previous page)

```
Authenticating...          [****************************************]
Waiting...                 [****************************************]
Successfully installed firmware
```

15.7 Test cases

For tests in this section please use CAB files listed in section *List of FWs used in this doc* or prepare own CAB files and upload it to appropriate remote as described in *LVFS* section.

15.7.1 Variables for test cases

Most of the test cases are the same for all the HW, only CAB files and versions are different. To unify the test cases for all devices, some variables are defined and used in test cases for any device:

- OLDCAB – URL or path to the CAB file of the previous version
- NEWCAB – URL or path to the CAB file of the target version

For instance for *ColorHug* it is required to define URL to CAB files with `export` command prior the test:

```
# export OLDCAB=https://fwupd.org/downloads/9a4e77009da7d3b5f15a1388afeb9e5d41a5a8ae-
↪hughski-colorhug2-1.2.5.cab

# export NEWCAB=https://fwupd.org/downloads/
↪c6fbb716abbb204d98f12edf1f146b6406f39b1eade741b353c15a86f5da8278-hughski-colorhug-1.2.
↪6.cab
```

The example above sets the OLDCAB variable to the URL of the CAB file with FW version **1.2.5**, and the NEWCAB to the URL of the CAB file with FW version **1.2.6**.

15.7.2 Test the FW from the private remote

Preconditions

- *Prepare Chrome OS for testing*
- Enable *access to ChromeBook*
- 2 FW CAB files uploaded to LVFS required (from the *List of FWs used in this doc*)
- Export *variables for test cases*
- The target device connected to Chromebook (*ColorHug* in this example)

Steps

1. *Check the device availability and version*

```
# fwupdmgr get-devices
```

2. Downgrade the device to the older version (parameter `--allow-older` is mandatory):

```
# fwupdmgr install ${OLDCAB} --allow-older
```

3. Expected result: last string must be

```
Successfully installed firmware
```

4. *Check the device availability and version*

```
# fwupdmgr get-devices
```

   Version of the FW must be equal to older release.

5. Upgrade the device to the newer version:

```
# fwupdmgr install ${NEWCAB}
```

6. Expected result: last string must be

```
Successfully installed firmware
```

7. *Check the device availability and version*

```
# fwupdmgr get-devices
```

   Version of the FW must be equal to the target release.

15.7.3 Test the FW from the embargoed remote

Preconditions

- *Prepare Chrome OS for testing*
- Enable *access to ChromeBook*
- *Enable the embargoed remote*
- Export "OLDCAB" *variable for test cases* ("NEWCAB" is not needed for this test)
- The target device connected to Chromebook

Steps

1. *Check the device availability and version*

```
# fwupdmgr get-devices
```

2. Downgrade the device to the older version (parameter `--allow-older` is mandatory):

```
# fwupdmgr install ${OLDCAB} --allow-older
```

3. Expected result: last string must be

```
Successfully installed firmware
```

4. *Check the device availability and version*

```
# fwupdmgr get-devices
```

   Version of the FW must be equal to older release.

5. *Install the FW CAB file from the embargo remote*

   a. Refresh the metadata for embargoed remote:

```
# fwupdmgr refresh
```

   b. Expected result: last string must be

```
Successfully downloaded new metadata: 6 local devices supported
```

      where the amount of supported devices might vary

   c. Update with the FW available from Embargo remote:

```
# fwupdmgr update
```

   d. Expected result: last string must be

```
Successfully installed firmware
```

6. *Check the device availability and version*

```
# fwupdmgr get-devices
```

   Version of the FW must be equal to the target release.

15.7.4 Test the FW from the testing remote

Preconditions

- *Prepare Chrome OS for testing*
- Enable *access to ChromeBook*
- *Enable testing remote*
- Export "OLDCAB" *variable for test cases* ("NEWCAB" is not needed for this test)
- The target device connected to Chromebook

Steps

1. *Check the device availability and version*

```
# fwupdmgr get-devices
```

2. Downgrade the device to the older version (parameter `--allow-older` is mandatory):

```
# fwupdmgr install ${OLDCAB} --allow-older
```

3. Expected result: last string must be

```
Successfully installed firmware
```

4. *Check the device availability and version*

```
# fwupdmgr get-devices
```

   Version of the FW must be equal to older release.

5. *Install the FW CAB file with the testing remote*

   a. Refresh the metadata for testing remote:

```
# fwupdmgr refresh
```

   b. Expected result: last string must be

```
Successfully downloaded new metadata: 6 local devices supported
```

      where the amount of supported devices might vary

   c. Update with the FW available from Testing remote:

```
# fwupdmgr update
```

   d. Expected result: last string must be

```
Successfully installed firmware
```

6. *Check the device availability and version*

```
# fwupdmgr get-devices
```

   Version of the FW must be equal to the target release.

15.7.5 Test the FW from the stable remote

Preconditions

- *Prepare Chrome OS for testing*
- Enable *access to ChromeBook*
- *Enable stable remote*
- Export "OLDCAB" *variable for test cases* ("NEWCAB" is not needed for this test)
- The target device connected to Chromebook

Steps

1. *Check the device availability and version*

```
# fwupdmgr get-devices
```

2. Downgrade the device to the older version (parameter `--allow-older` is mandatory):

```
# fwupdmgr install ${OLDCAB} --allow-older
```

3. Expected result: last string must be

```
Successfully installed firmware
```

4. *Check the device availability and version*

```
# fwupdmgr get-devices
```

   Version of the FW must be equal to older release.

5. *Install the FW CAB file with the stable remote*

   a. Refresh the metadata for stable remote:

```
# fwupdmgr refresh
```

   b. Expected result: last string must be

```
Successfully downloaded new metadata: 6 local devices supported
```

   where the amount of supported devices might vary.

   c. Update with the FW available from Stable remote:

```
# fwupdmgr update
```

   d. Expected result: last string must be

```
Successfully installed firmware
```

6. *Check the device availability and version*

```
# fwupdmgr get-devices
```

   Version of the FW must be equal to the target release.

15.7.6 Test GUI update with Google internal remote

Preconditions

- *Prepare Chrome OS for testing*
- Enable *access to ChromeBook*
- Export "OLDCAB" *variable for test cases* ("NEWCAB" is not needed for this test)
- The target FW must exist in Google internal remote
- The target device connected to Chromebook

Steps

1. *Check the device availability and version*

```
# fwupdmgr get-devices
```

2. Downgrade the device to the older version (parameter `--allow-older` is mandatory):

```
# fwupdmgr install ${OLDCAB} --allow-older
```

3. Expected result: last string must be

```
Successfully installed firmware
```

4. *Check the device availability and version*

```
# fwupdmgr get-devices
```

   Version of the FW must be equal to older release.

5. Switch to the Chrome OS UI if you are in linux console:

   switch to the linux console on ChromeBook by pressing: "Ctrl + Alt + ←" ('←' or 'F1' on top row).

6. After downgrading it is possible to *update the device with GUI* to the last available version :

   go to "Settings" -> "About Chrome OS" -> "Firmware update"

15.7.7 Test GUI update with embargoed remote

Preconditions

- *Prepare Chrome OS for testing*
- Enable *access to ChromeBook*
- *Enable the embargoed remote*
- Export "OLDCAB" *variable for test cases* ("NEWCAB" is not needed for this test)
- The target device connected to Chromebook

Steps

1. *Check the device availability and version*

```
# fwupdmgr get-devices
```

2. Downgrade the device to the older version (parameter `--allow-older` is mandatory):

```
# fwupdmgr install ${OLDCAB} --allow-older
```

3. Expected result: last string must be

```
Successfully installed firmware
```

4. *Check the device availability and version*

```
# fwupdmgr get-devices
```

Version of the FW must be equal to older release.

5. Refresh the metadata from the remote:

```
# fwupdmgr refresh --force
```

6. Switch to the Chrome OS UI if you are in linux console:

   switch to the linux console on ChromeBook by pressing: "Ctrl + Alt + ←" ('←' or 'F1' on top row).

7. After downgrading it is possible to *update the device with GUI* to the last available version :

   go to "Settings" -> "About Chrome OS" -> "Firmware update"

15.7.8 Test GUI update with testing remote

Preconditions

- *Prepare Chrome OS for testing*
- Enable *access to ChromeBook*
- *Enable testing remote*
- Export "OLDCAB" *variable for test cases* ("NEWCAB" is not needed for this test)
- The target device connected to Chromebook

Steps

1. *Check the device availability and version*

```
# fwupdmgr get-devices
```

2. Downgrade the device to the older version (parameter `--allow-older` is mandatory):

```
# fwupdmgr install ${OLDCAB} --allow-older
```

3. Expected result: last string must be

```
Successfully installed firmware
```

4. *Check the device availability and version*

```
# fwupdmgr get-devices
```

Version of the FW must be equal to older release.

5. Refresh the metadata from the remote:

```
# fwupdmgr refresh --force
```

6. Switch to the Chrome OS UI if you are in linux console:

   switch to the linux console on ChromeBook by pressing: "Ctrl + Alt + ←" ('←' or 'F1' on top row).

7. After downgrading it is possible to *update the device with GUI* to the last available version :

   go to “Settings” -> “About Chrome OS” -> “Firmware update”

15.7.9 Test GUI update with stable remote

Preconditions

- *Prepare Chrome OS for testing*
- Enable *access to ChromeBook*
- *Enable stable remote*
- Export “OLDCAB” *variable for test cases* (“NEWCAB” is not needed for this test)
- The target device connected to Chromebook

Steps

1. *Check the device availability and version*

   ```
   # fwupdmgr get-devices
   ```

2. Downgrade the device to the older version (parameter `--allow-older` is mandatory):

   ```
   # fwupdmgr install ${OLDCAB} --allow-older
   ```

3. Expected result: last string must be

   ```
   Successfully installed firmware
   ```

4. *Check the device availability and version*

   ```
   # fwupdmgr get-devices
   ```

   Version of the FW must be equal to older release.

5. Refresh the metadata from the remote:

   ```
   # fwupdmgr refresh --force
   ```

6. Switch to the Chrome OS UI if you are in linux console:

   switch to the linux console on ChromeBook by pressing: “Ctrl + Alt + ←” (‘←’ or ‘F1’ on top row).

7. After downgrading it is possible to *update the device with GUI* to the last available version :

   go to “Settings” -> “About Chrome OS” -> “Firmware update”

15.7.10 Recovery from the failed update with CLI

Preconditions

- *Prepare Chrome OS for testing*
- Enable *access to ChromeBook*
- 2 FW CAB files uploaded to LVFS required (from the *List of FWs used in this doc*)
- Export *variables for test cases*
- The target device connected to Chromebook (*ColorHug* in this example)

Steps

1. *Check the device availability and version*

```
# fwupdmgr get-devices
```

2. Downgrade the device to the older version (parameter `--allow-older` is mandatory):

```
# fwupdmgr install ${OLDCAB} --allow-older
```

3. Expected result: last string must be

```
Successfully installed firmware
```

4. *Check the device availability and version*

```
# fwupdmgr get-devices
```

5. Copy the "Device ID:" value from the target device, for example:

```
├─ColorHug:
│         Device ID:        23cf6368c14a875f74c38a5a423518f38d8abbbc
│         Summary:          An open source display colorimeter
│         Current version:  1.2.5
```

   The Device ID is unique per device!!!

6. Put the device into bootloader mode.

   Please use the ID of the target device detected on the previous step!!!

```
# fwupdtool detach 23cf6368c14a875f74c38a5a423518f38d8abbbc
```

7. The device must notify it is in bootloader mode with blinking LED.
8. *Check the device availability and version*

```
# fwupdmgr get-devices
```

9. Expected result: the device flag `Is in bootloader mode` must exists

```
├─ColorHug:
│         Device ID:        23cf6368c14a875f74c38a5a423518f38d8abbbc
│         Summary:          An open source display colorimeter
│         Current version:  1.2.5
│         Vendor:           Hughski Ltd. (USB:0x273F)
│         Install Duration: 8 seconds
│         GUIDs:            40338ceb-b966-4eae-adae-9c32edfcc484
│                           afdcc391-6c33-5914-b4d2-b4dd71fe9c5a
│                           6bc5ff27-d631-5660-9991-6d24954c6f90 ← USB\VID_273F&PID_
↪1001
│                           4841a9e4-e5c8-5107-a83e-d6c6d9c21248 ← USB\VID_273F&PID_
↪1001&REV_0002
│         Device Flags:     • Updatable
│                           • Supported on remote server
│                           • Device can recover flash failures
│                           • Is in bootloader mode
│                           • Unsigned Payload
```

10. Upgrade the device to the newer version:

```
# fwupdmgr install ${NEWCAB}
```

11. Expected result: last string must be

```
Successfully installed firmware
```

12. *Check the device availability and version*

```
# fwupdmgr get-devices
```

    Version of the FW must be equal to the target release.

15.7.11 Test the FW reinstall

Preconditions

- *Prepare Chrome OS for testing*
- Enable *access to ChromeBook*
- Export “NEWCAB” *variable for test cases* (“OLDCAB” is not needed for this test)
- The target device connected to Chromebook (*ColorHug* in this example)

7.11.2. Steps

1. *Check the device availability and version*

```
# fwupdmgr get-devices
```

2. Update the device to the target CAB:

```
# fwupdmgr install ${NEWCAB}
```

3. Expected result: last string must be either

```
Successfully installed firmware...
```

OR

```
All updatable firmware is already installed
```

4. *Check the device availability and version*

```
# fwupdmgr get-devices
```

Version of the FW must be equal to the target release.

5. Re-install the FW for all subdevices with the target release:

```
# fwupdmgr install ${NEWCAB} --allow-reinstall
```

6. Expected result: last string must be

```
Successfully installed firmware
```

7. *Check the device availability and version*

```
# fwupdmgr get-devices
```

Version of the FW must be equal to the target release.

15.8 Appendix A: List of FWs used in this doc

This is the list of FWs uploaded to LVFS and available for downloading and testing.

15.8.1 ColorHug

1. 1.2.5

   https://fwupd.org/downloads/9a4e77009da7d3b5f15a1388afeb9e5d41a5a8ae-hughski-colorhug2-1.2.5.cab

2. 1.2.6

   https://fwupd.org/downloads/c6fbb716abbb204d98f12edf1f146b6406f39b1eade741b353c15a86f5da8278-hughski-colorhug-1.2.6.cab

```
# export OLDCAB=https://fwupd.org/downloads/9a4e77009da7d3b5f15a1388afeb9e5d41a5a8ae-
↪hughski-colorhug2-1.2.5.cab
```

```
# export NEWCAB=https://fwupd.org/downloads/
↪c6fbb716abbb204d98f12edf1f146b6406f39b1eade741b353c15a86f5da8278-hughski-colorhug-1.2.
↪6.cab
```

CHAPTER

SIXTEEN

HOW TO RUN FWUPD TESTS WITH MOBLAB

This howto shows how to run the fwupd test suites on Moblab to verify the firmware updatability of peripherals in ChromeOS.

16.1 Overview

fwupd is a system daemon that allows an OS to update the firmware of a wide array of peripherals. ChromeOS relies on it to perform the firmware updates of compatible and updatable devices.

Moblab contains a set of fwupd test suites to test the basic firmware-related operations that fwupd can perform on a device. The purpose of these tests is, primarily, to validate the correct firmware updatability of new peripherals so that they comply with the WWCB Certification and to check the consistency and correctness of these operations across different ChromeOS versions and firmware releases.

16.2 Before you begin

To run the fwupd tests you'll need a working **Moblab** setup with at least one **DUT** (a Chromebook or Chromebox) running a ChromeOS test image.

16.2.1 How to get a Partner Domain account

In order to have access to ChromeOS test releases you'll need a Partner Domain account. To ask for an account, send a request to cros-pd-owners@google.com.

Note: Besides the Partner Domain account, you should also get access to a CPCon account, a Service account and a GCS bucket tied to the CPCon account.

16.2.2 How to ask for access to ChromeOS images for specific boards

Depending on the Chromebooks and Chromeboxes that you use for the tests, you'll need to have explicit access to the ChromeOS test images for each specific model or board type so you can download them from the ChromeOS Partners Portal.

To request access, create an issue in the Partner Issue Tracker using **Component** "ChromeOS Public Tracker > Services > Infra > Moblab" (1038089) and **template** "Build Permission Access Request" and fill in the template details.

16.2.3 How to get a Moblab

Reference: Moblab Instruction Manual section "II - Requirements"

Moblab is based on a Google Chromebox (**Wukong or Wyvern**). The distributor of Moblab is CREATE TOGETHER TECHNOLOGY Co. Ltd. To inquire a quotation, reach out to:

- Hans (hans@cttech-group.com)
- Kiki (kiki@cttech-group.com)

Please note, Moblab is not supported in mainland China, nor has it been certified by China CCC.

16.2.4 Required hardware

- One or more DUTs (Chromebook or Chromebox)
- Network equipment as described in section "II - Requirements" of the Moblab Instruction Manual, at least two Ethernet cables and a USB to Ethernet dongle. Follow the instructions in section "II.2 - Now that I have all the required hardware, how do I connect them?" in the Moblab Instruction Manual to setup the test lab
- A USB flash drive larger than 8GB
- An hdmi or displayport monitor, a keyboard and a mouse
- If the DUT is a Chromebox: an additional monitor, keyboard and mouse

16.2.5 Required software

Chrome Recovery Utility extension installed in Chrome.

16.2.6 Initial DUT (Chromebook) setup

Main references:

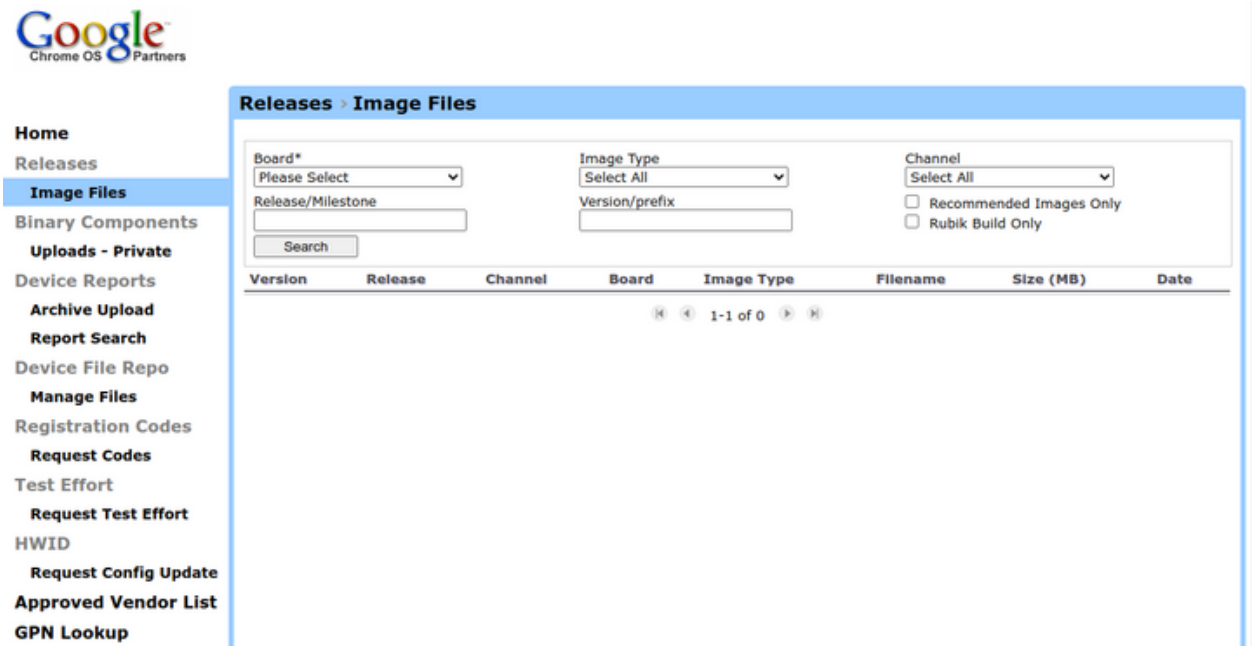
- Developer Mode
- Test Image & OS Recovery Setup Documentation

In order to use a Chromebook or Chromebox for Moblab tests it needs to be running a ChromeOS test release. Before continuing, make sure you have a partner domain account with access to the ChromeOS Partners Portal and to the ChromeOS images for the specific Chromebook/Chromebox board types you want to test.

Flash a ChromeOS test Image

First, start by flashing a ChromeOS test image into a USB flash drive:

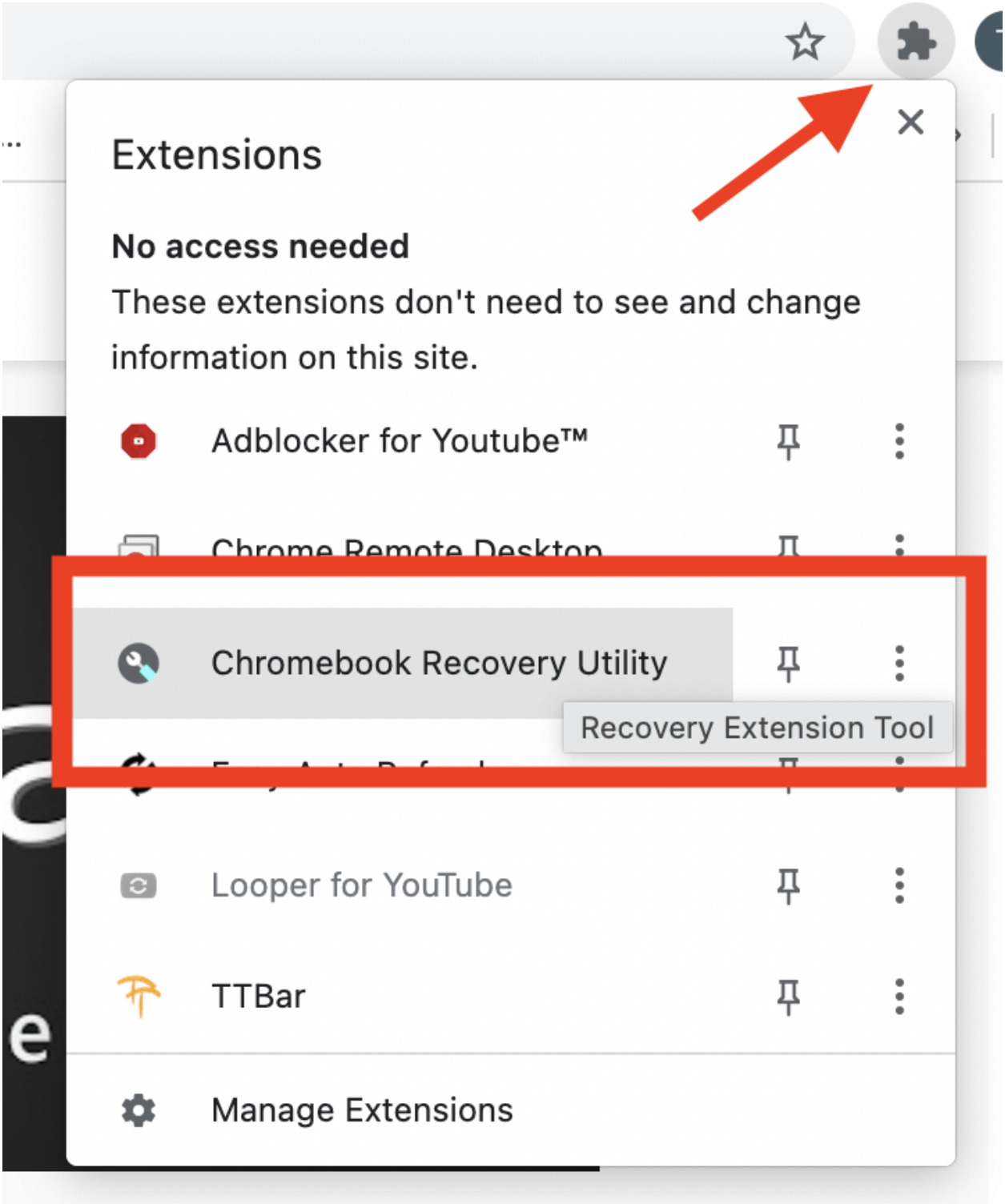
1. Go to the *ChromeOS Partners Portal <https://www.google.com/chromeos/partner/fe/#release>* and log in with your partner domain account and click the "Image Files" tab.

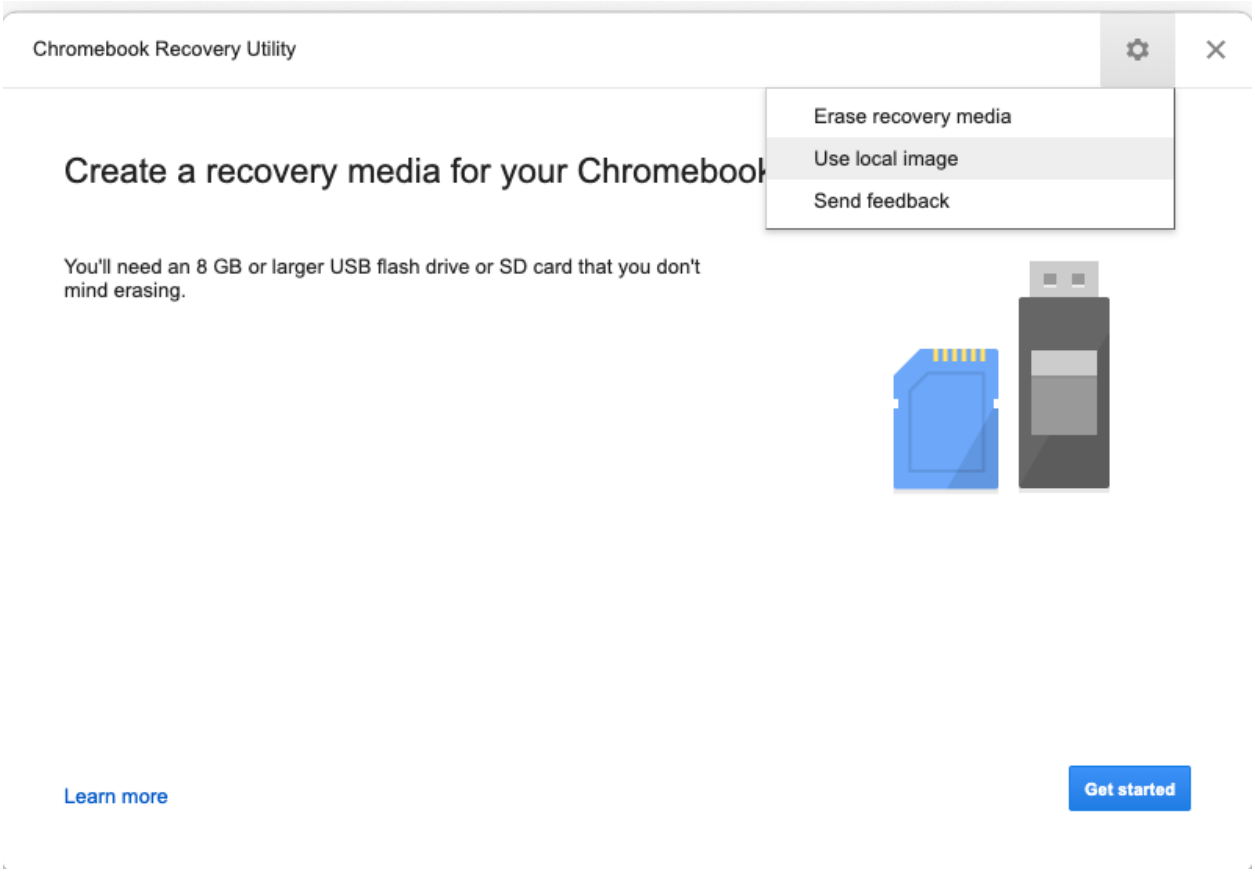


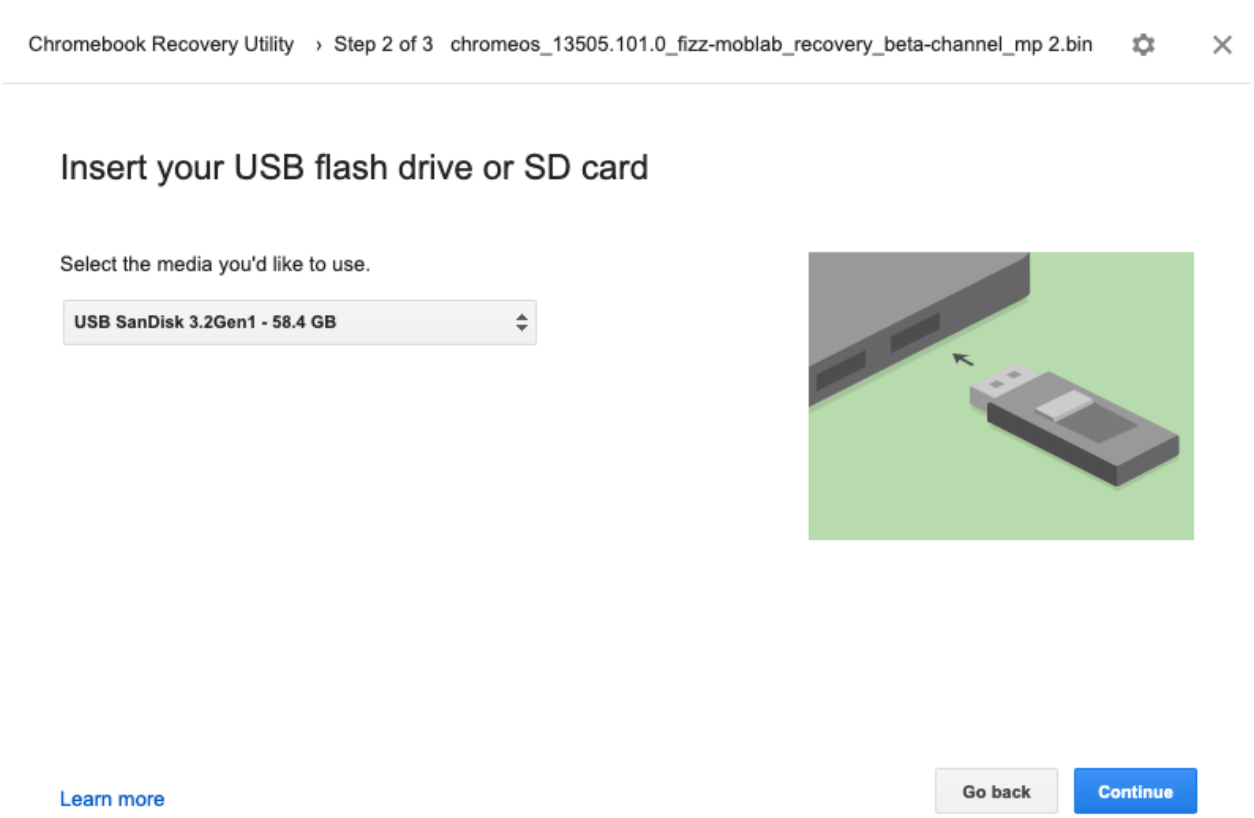
2. Select the board of the Chromebook or Chromebox to test and TEST_IMAGE_ARCHIVE in the "Image Type" drop-down menu, and click "Search" to list all the ChromeOS test images for that board. You can refine the search by entering a specific "Release/Milestone" and/or a specific "Version/prefix".
3. Download a recent test image (dev or stable channels recommended) and decompress it with:

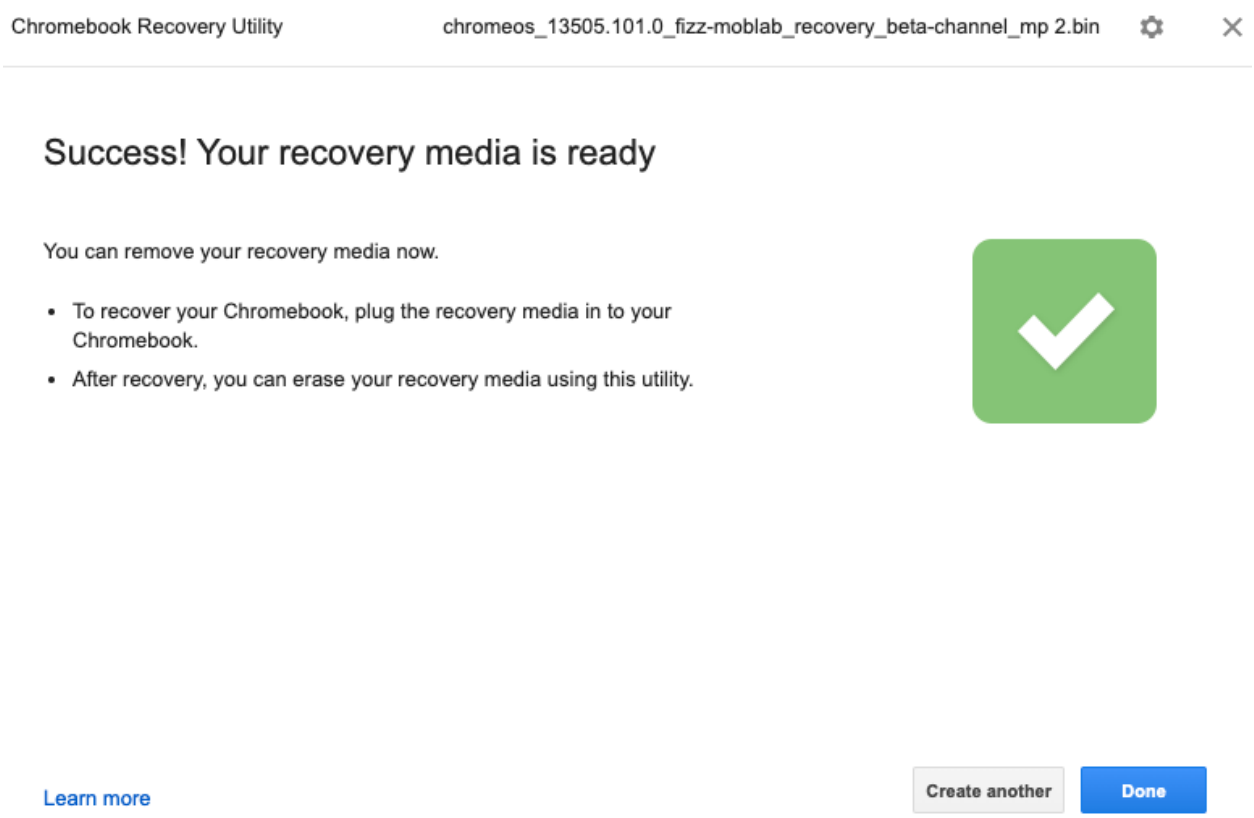
```
$ tar -Jxvf <release_file.tar.xz>
```

4. Click the extension  button in Chrome, then select **Chrome Recovery Utility**
5. Select the gear icon  in the window. Next, click **Use local image**.
6. Select the `chromium_test_image.bin` file extracted in step 3
7. Plug in the USB flash drive and select it as the media to use. Click **Continue** and then **Create now**. Wait until the image is completely written to the USB drive.
8. Once complete, Select **Done** then unplug the USB flash drive


Extensions
No access needed
These extensions don't need to see and change information on this site.
Adblocker for Youtube™
Chrome Remote Desktop
Chromebook Recovery Utility
Recovery Extension Tool
Looper for YouTube
TTBar
Manage Extensions


Chromebook Recovery Utility
Erase recovery media
Use local image
Send feedback
Create a recovery media for your Chromebook
You'll need an 8 GB or larger USB flash drive or SD card that you don't mind erasing.
Learn more
Get started


Chromebook Recovery Utility › Step 2 of 3 chromeos_13505.101.0_fizz-moblab_recovery_beta-channel_mp 2.bin
Insert your USB flash drive or SD card
Select the media you'd like to use.
USB SanDisk 3.2Gen1 - 58.4 GB
Learn more
Go back
Continue



Install test image

Next, to install the image in the Chromebook or Chromebox, follow these steps:

1. Put the device into Developer Mode with the following procedure:

   a. For Chromebooks, Hold `Esc + Refresh` and press the `Power` button. For Chromeboxes, engage the small `Reset` pinhole with a paperclip, hit `Power` and continue engaging `Reset` for 2 seconds. This will put the device into **Recovery Mode** and it should show a screen similar to this:

   Or this, depending on the model:

   Note: On some Chromebooks the combination to hold is `Esc + Full screen` instead.

   ---

   b. In the Recovery Mode screen, press `Ctrl + D`, followed by `Enter` to enter Developer Mode

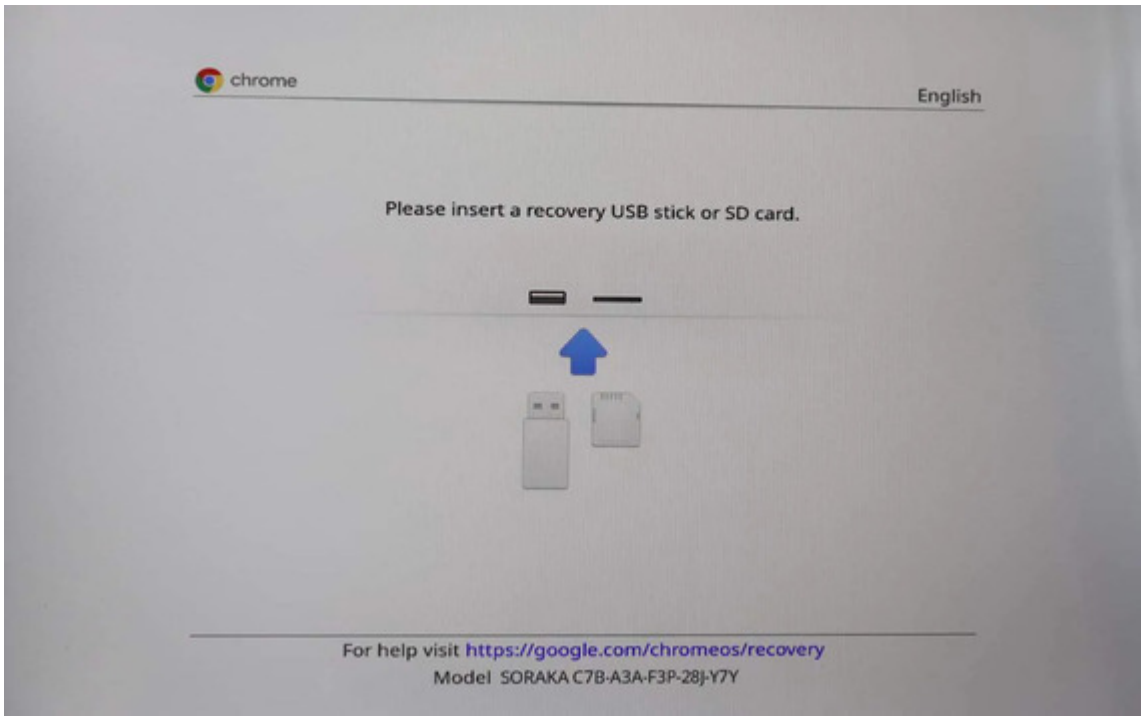
   ---

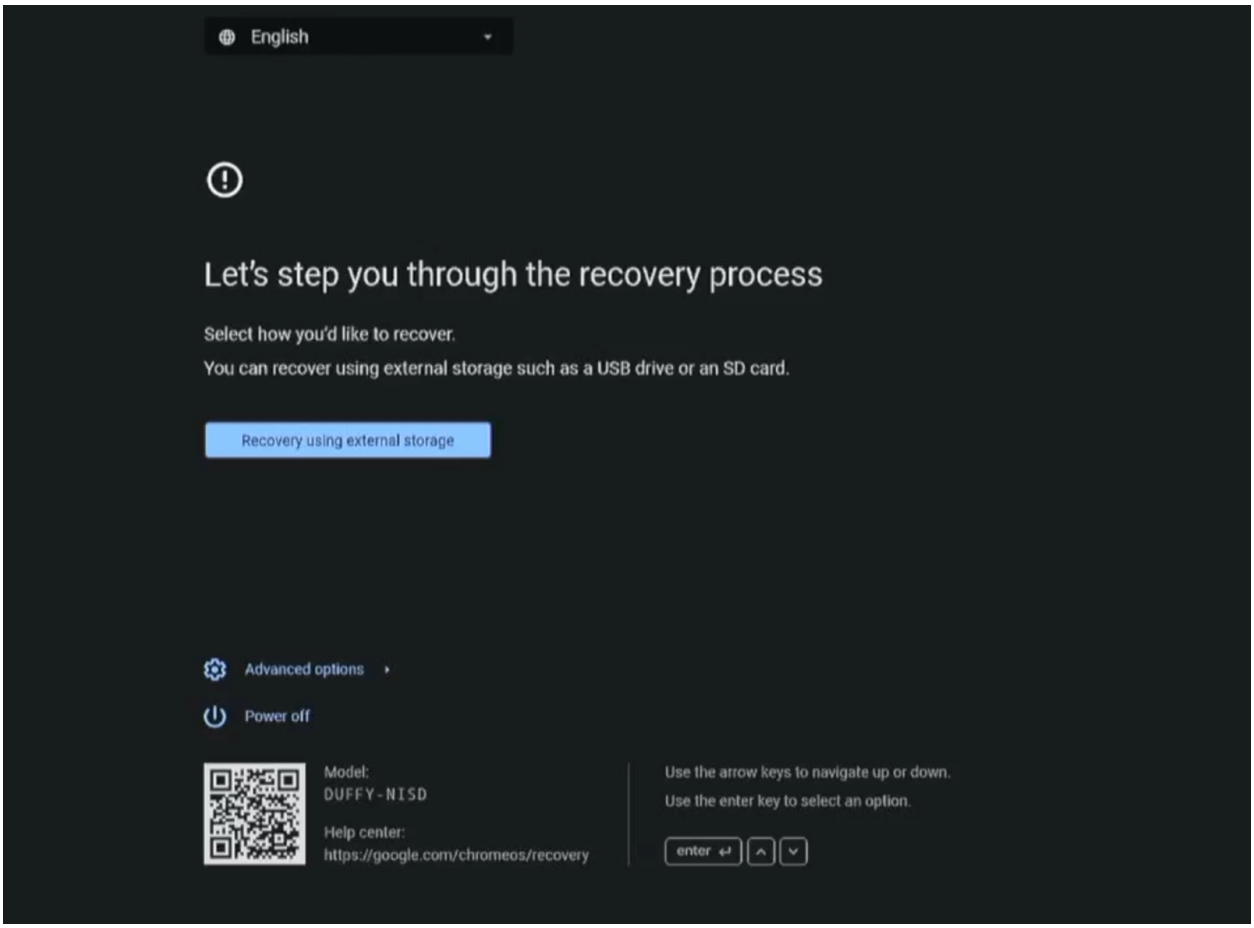
   Note: For other devices without keyboards (such as tablets) follow these instructions to enter Recovery Mode and Developer Mode

   ---

   c. Wait until the process is done and the Developer Mode warning screen appears

2. Once the device is in Developer Mode, it will show the warning screen above every time it boots. It'll start ChromeOS after 30 seconds or if you press `Ctrl + D`. Start ChromeOS and wait for it to show the welcome screen


chrome
English
Please insert a recovery USB stick or SD card.
For help visit https://google.com/chromeos/recovery
Model SORAKA C7B-A3A-F3P-28J-Y7Y


English
Let's step you through the recovery process
Select how you'd like to recover.
You can recover using external storage such as a USB drive or an SD card.
Recovery using external storage
Advanced options
Power off
Model:
DUFFY-NISD
Help center:
https://google.com/chromeos/recovery
Use the arrow keys to navigate up or down.
Use the enter key to select an option.
enter

3. Go to virtual terminal 2 to access a command line prompt by pressing:

```
[ Ctrl ] [ Alt ] [ → ]
```

   where the [→] key is the right-arrow key just above the number 3 on the keyboard. If the keyboard doesn't have this key, use the key in the F2 position. Then log in with user: `root`

4. Enable USB boot with the following commands:

```
$ sudo crossystem dev_boot_usb=1
$ sudo crossystem dev_boot_signed_only=0
```

5. Now reboot and wait for the Developer Mode warning screen to appear, plug in the USB flash drive and press `Ctrl + U` to boot the ChromeOS test image from the USB drive.

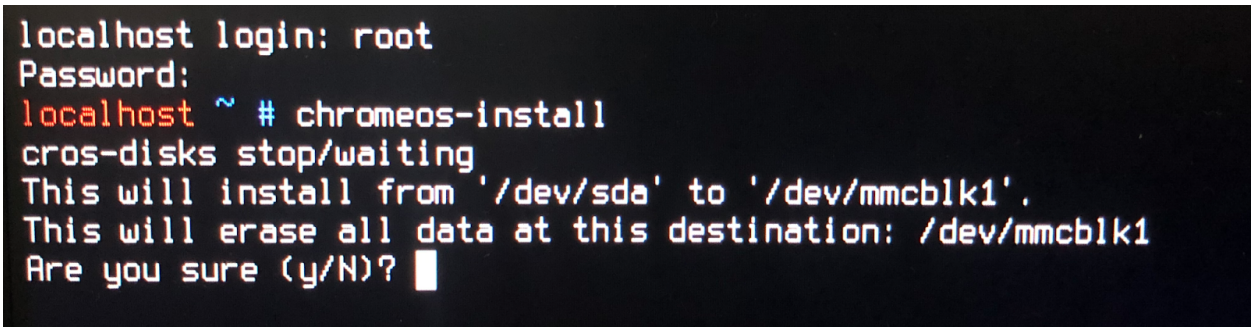
   Wait for ChromeOS to start

6. Once ChromeOS is running, go to virtual terminal 2 again and log in with user: `root` and password: `test0000`. Then install the test image in the hard disk with the following command:

```
$ /usr/sbin/chromeos-install
```

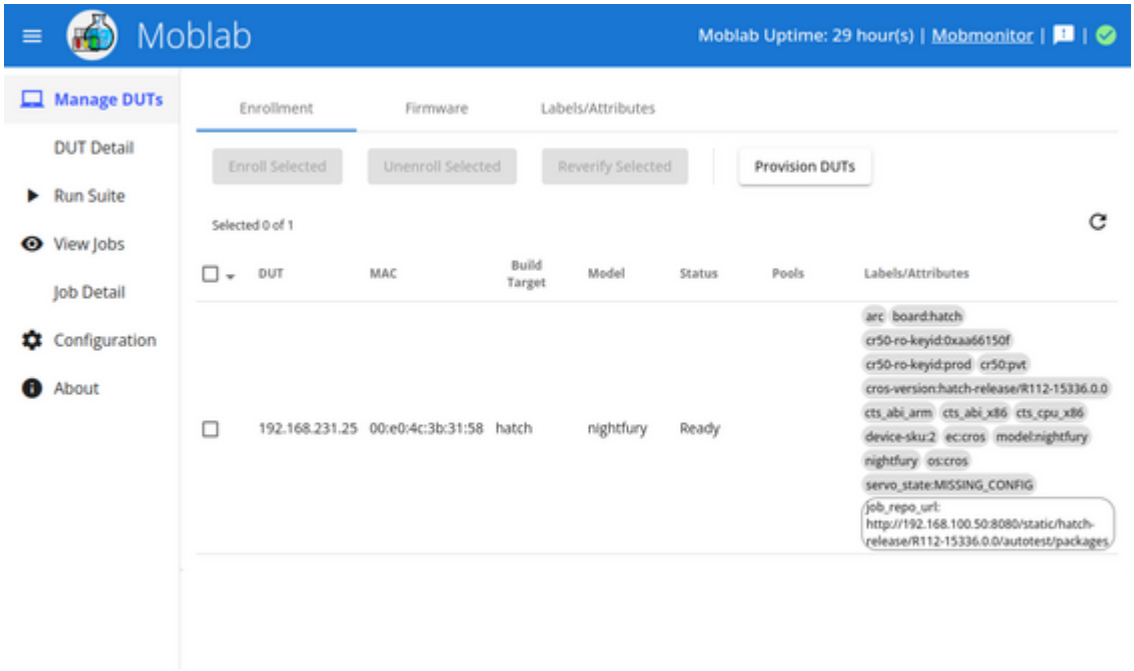
   and follow the instructions

7. Once installation has completed, reboot the device (`shutdown -h now`) and **remove the USB flash drive**



16.2.7 Moblab setup

Follow the instructions in the Moblab Introduction & User Manual to configure the Moblab, connect the DUT (Chromebook or Chromebox) to it and enroll it. The end result must be something like this, where the "Manage DUTs" tab shows an enrolled DUT with a "Ready" Status:



16.2.8 Other considerations and requirements

About the peripheral status

The peripherals to test must be in working order. If they are meant to be updated wirelessly, they must have sufficient battery level to ensure the firmware update process can be completed successfully. They must also be supported by fwupd and they must have at least a firmware release included in the ChromeOS-specific fwupd remotes, which are defined in the latest .ebuild file in https://chromium.googlesource.com/chromiumos/overlays/chromiumos-overlay/+/refs/heads/main/sys-firmware/fwupd-peripherals/.

About the fwupd version and peripheral support

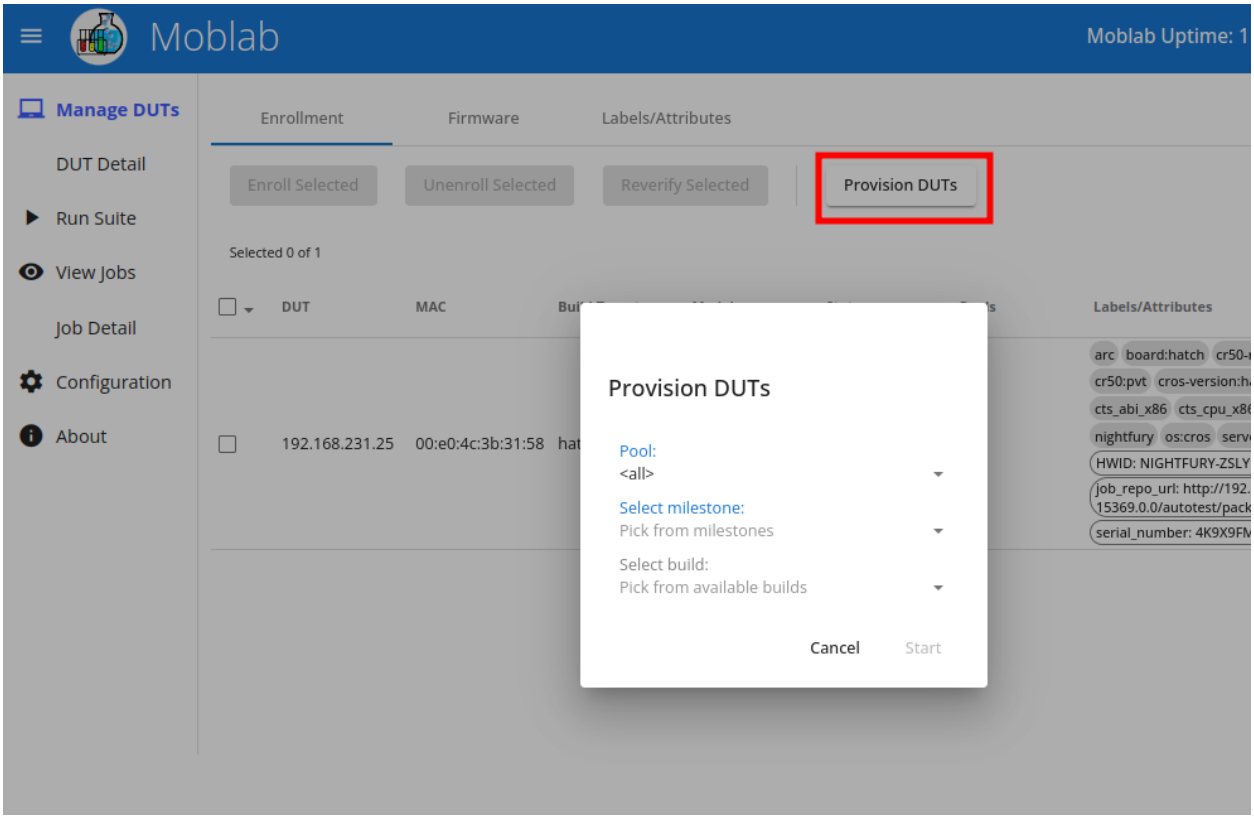
The fwupd version might be different from one ChromeOS version to another, so a device that is supported by fwupd in a newer ChromeOS version might not be supported in an older one.

About DUT provisioning in Moblab

Each Moblab test run will start by provisioning the DUT, that is, updating its ChromeOS version to the one specified by the tester and checking that it can run the tests properly. This has some side effects:

- The DUT will constantly check for an Ethernet link and will reboot automatically if it doesn't detect one after a few seconds.
- All Bluetooth device pairings will be wiped out

A DUT can be also provisioned on demand by selecting the DUT in the "**Manage DUTs**" tab and clicking on the "**Provision DUTs**" button, then selecting the milestone and build to use:



16.3 Test cases

Note:

- **The FWUPD tests are available only on ChromeOS R113-15382.0.0 and later**.
- **Tests of firmware updates/downgrades/installs over Bluetooth links are not properly supported at the moment**

The procedure to run the tests is the same for all the test cases except for the parameters they take. To start a test, follow these steps:

1. Make sure the DUT that will run the test is running, connected to the Moblab network and that it shows up in the "Manage DUTs" tab in Moblab
2. Enroll the DUT for tests by selecting it in the "Manage DUTs" tab and then clicking "Enroll Selected"
3. Make a note of the DUT IP address, as it'll need to be specified later as one of the test parameters
4. Connect the peripheral to the DUT and power it on
5. Go to the "Run Suite" tab and select "FWUPD" in the top menu
6. Select the model and build target of the selected DUT, and then the ChromeOS milestone and build to run the test on. **Important: only ChromeOS R113-15382.0.0 and later. The tests won't be started for any version older than that**.
7. Select the IP address of the DUT that will run the test

16.3.1 Update a device firmware to the latest release

This test case will try to update a device firmware to the latest release available in the fwupd remotes. To run it:

1. Select the `fwupd_update` suite
2. Select the device you want to test
3. Click on "Run Suite"

> **Note:** If there aren't any new firmware releases available for the device, the test won't proceed and will be marked as an error.

16.3.2 Downgrade a device firmware to the previous release

This test case will try to downgrade a device firmware to the previous release found in the fwupd remotes. To run it:

1. Select the `fwupd_downgrade` suite
2. Select the device you want to test
3. Click on "Run Suite"

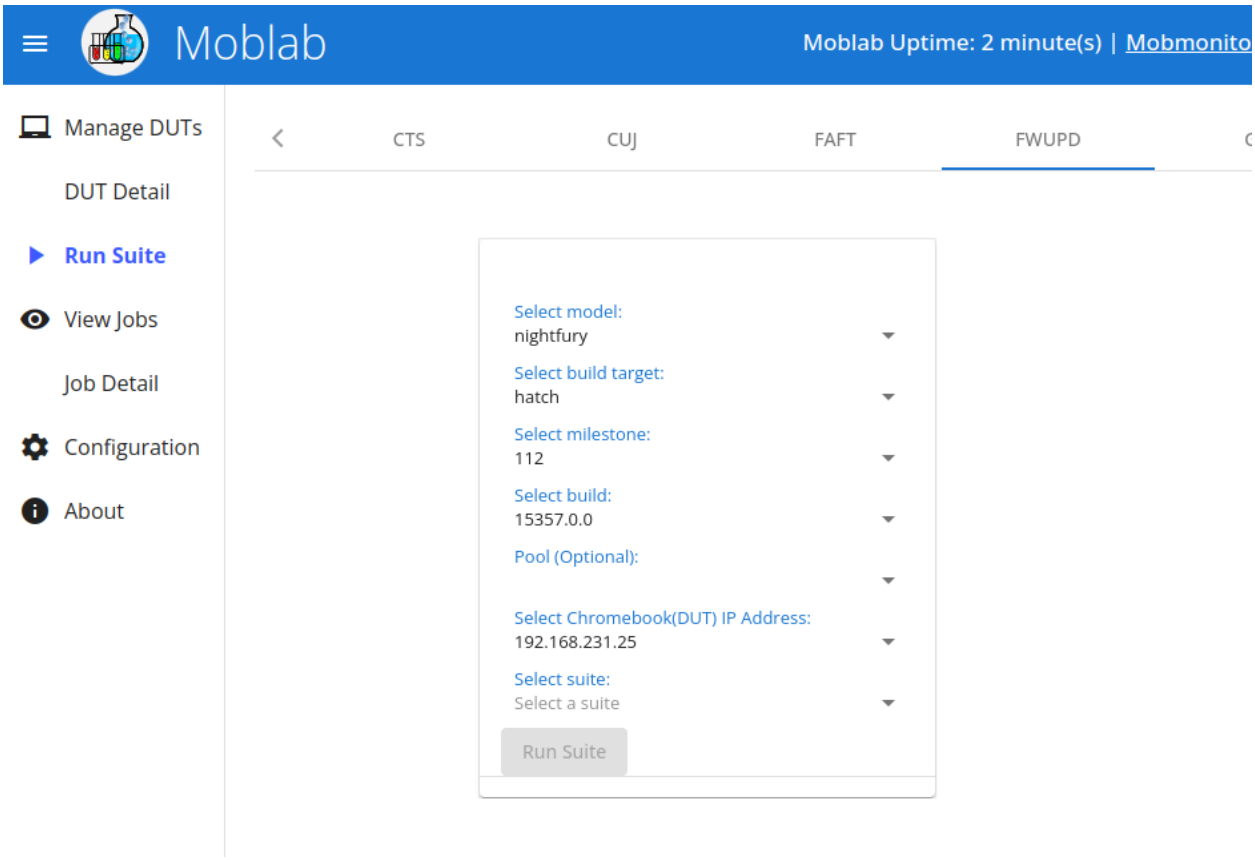
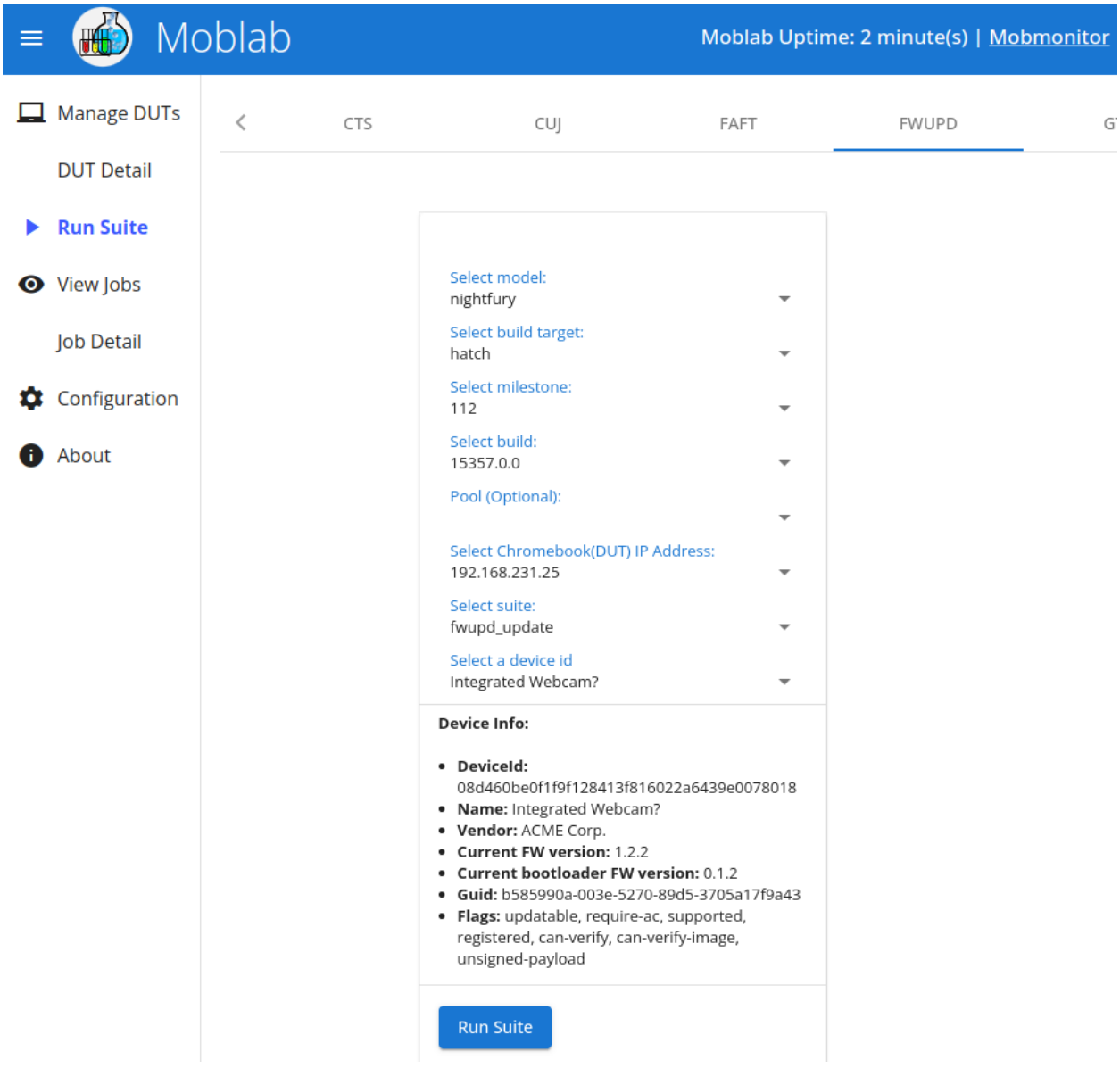


Fig. 1: The rest of the steps depends on the test case to run:

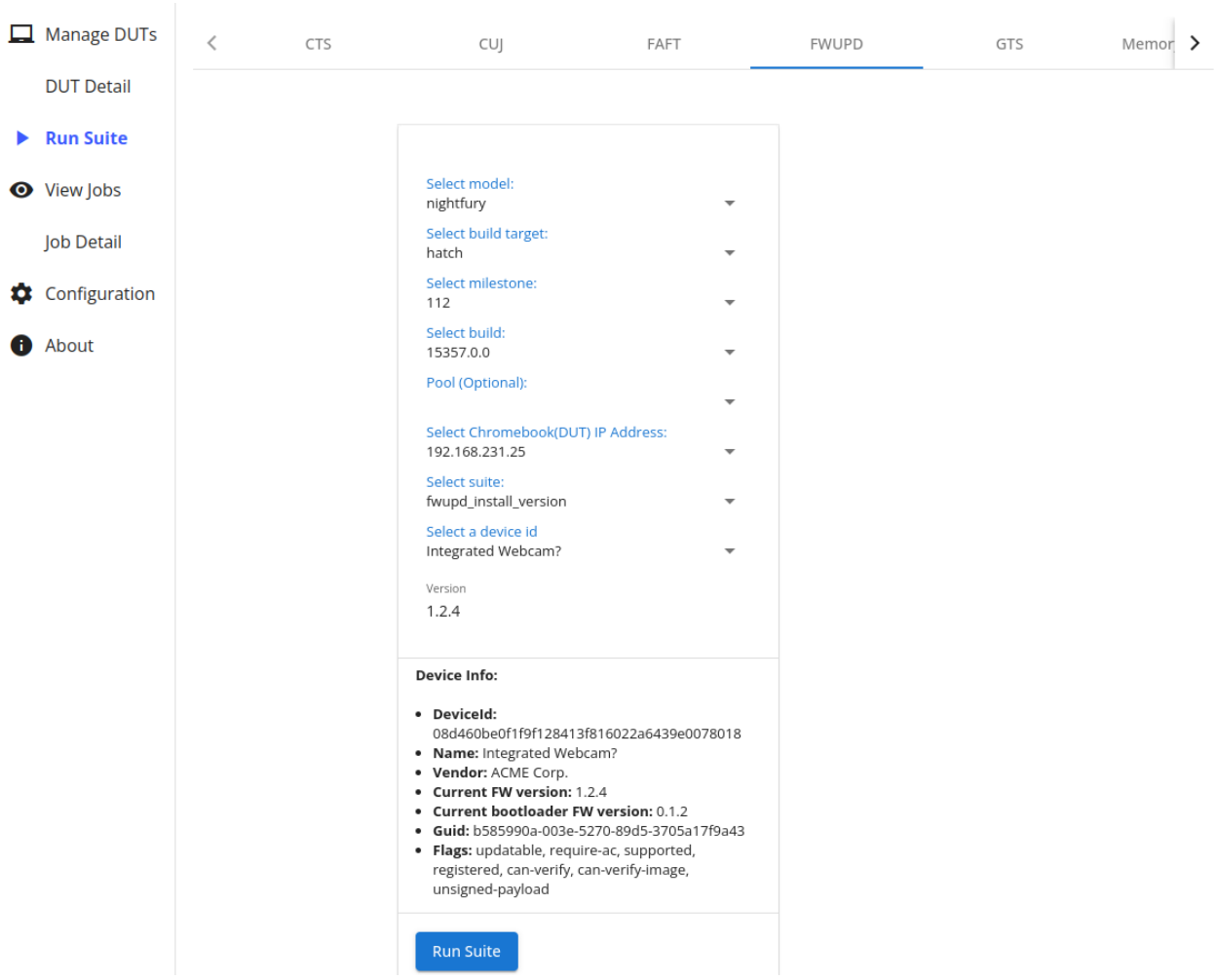

Moblab
Moblab Uptime: 2 minute(s) | Mobmonitor
Manage DUTs
DUT Detail
Run Suite
View Jobs
Job Detail
Configuration
About
CTS
CUJ
FAFT
FWUPD
Select model:
nightfury
Select build target:
hatch
Select milestone:
112
Select build:
15357.0.0
Pool (Optional):
Select Chromebook(DUT) IP Address:
192.168.231.25
Select suite:
fwupd_update
Select a device id
Integrated Webcam?
Device Info:
DeviceId: 08d460be0f1f9f128413f816022a6439e0078018
Name: Integrated Webcam?
Vendor: ACME Corp.
Current FW version: 1.2.2
Current bootloader FW version: 0.1.2
Guid: b585990a-003e-5270-89d5-3705a17f9a43
Flags: updatable, require-ac, supported, registered, can-verify, can-verify-image, unsigned-payload
Run Suite

Note: If there aren't any previous firmware releases available for the device, the test won't proceed and will be marked as an error.

16.3.3 Install a firmware version

This test allows the user to flash any available firmware release into a device, regardless of the current version running on it. To run it:

1. Select the `fwupd_install_version` suite
2. Select the device you want to test
3. Input the release version you want to install. Note that the version must be specified in the format defined by the hardware vendor (as a number pair, triplet, hexadecimal number, etc.). The **Device** info at the bottom shows the **Current FW version** with the expected format.
4. Click on "Run Suite"



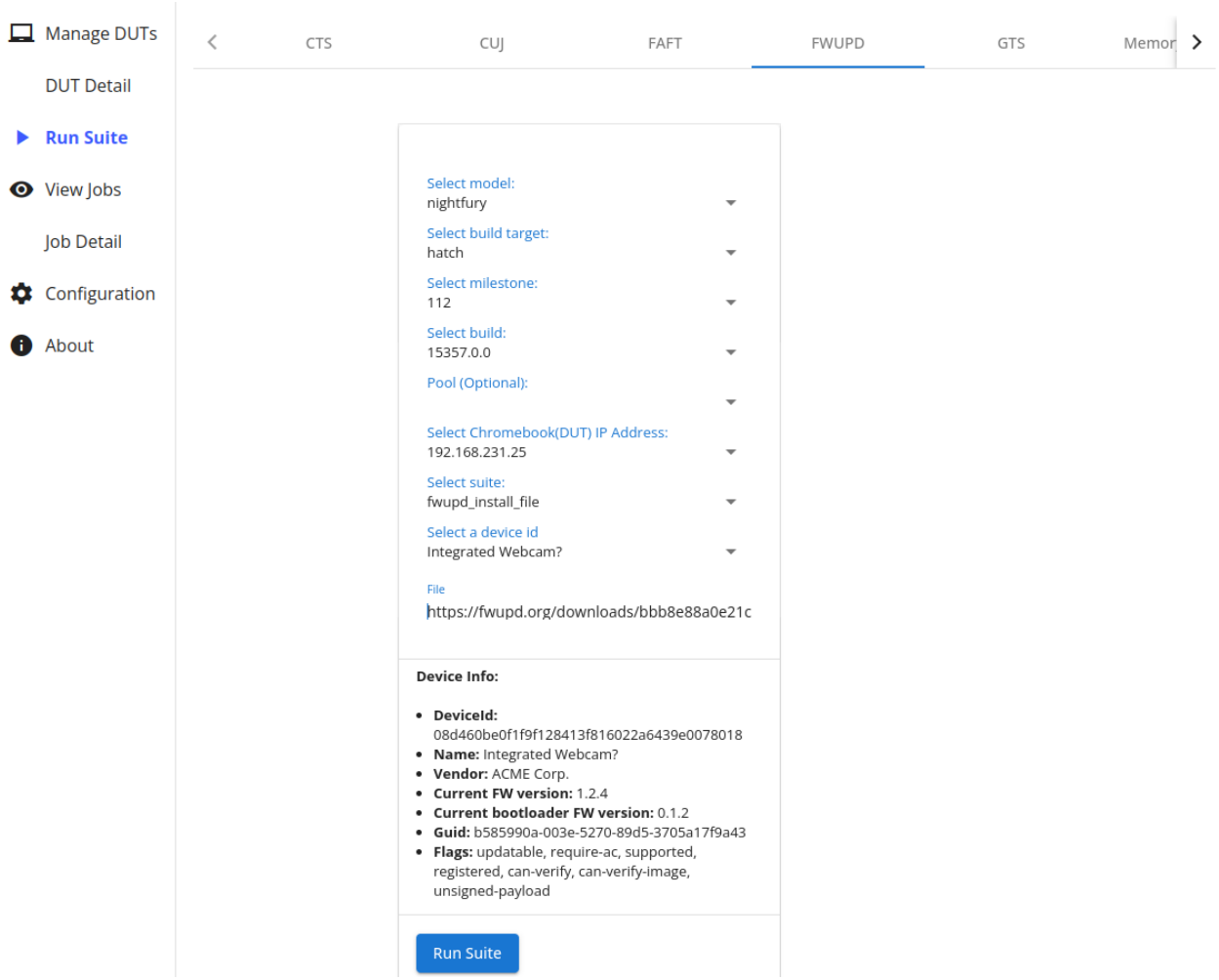
16.3.4 Install a firmware file

This test allows the user to flash a specific firmware file into a device. The file can be provided either as a URL if it's in a remote server, or through an external drive connected to the DUT.

- If the file is provided in an external drive, a USB flash drive is recommended. Format it as FAT32 and set FWUPDTESTS as its label. Then copy the firmware file to it and plug it to the DUT.
- If the file is provided as a URL, it must be accessible for the DUT to download.

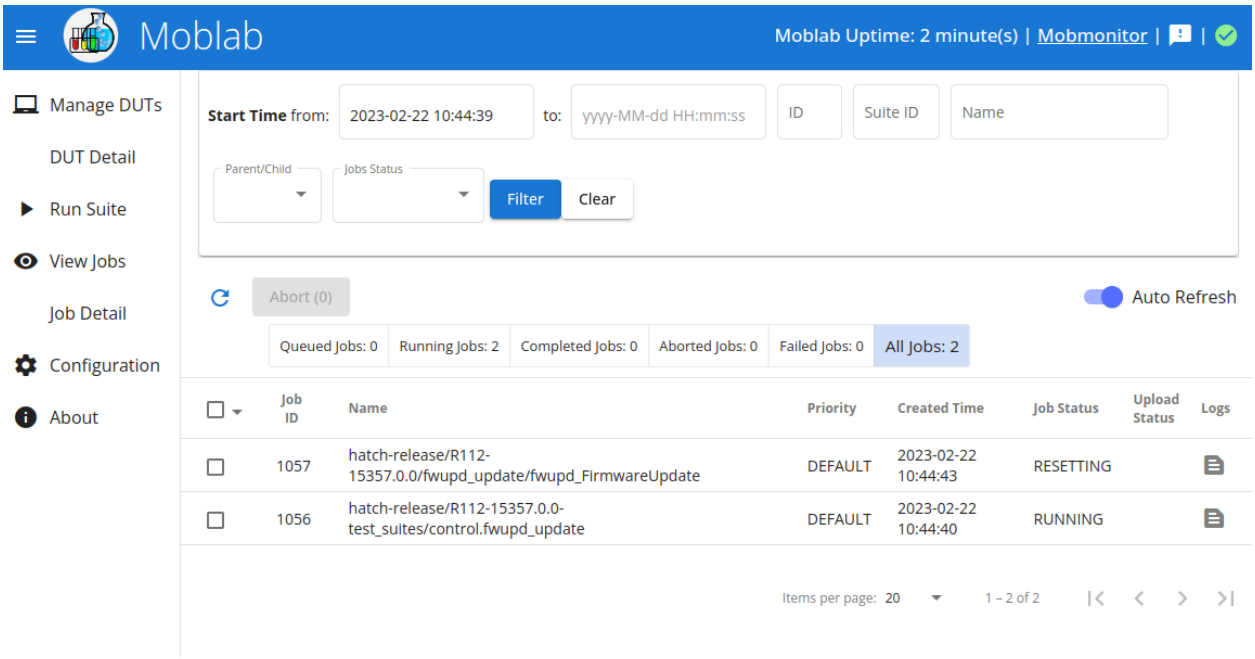
To run the test:

1. Select the `fwupd_install_file` suite
2. Select the device you want to test
3. Enter the complete URL of the firmware file if it's a remote file or the file name if it's a file provided through a USB flash drive
4. Click on "Run Suite"



16.4 How to verify the test results

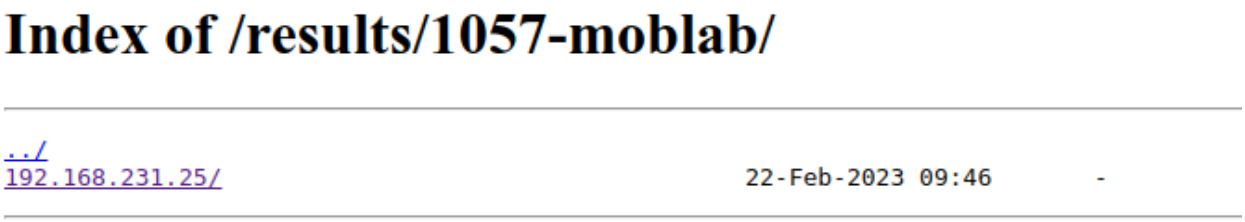
Once a test has started, it'll show up in the "View Jobs" tab. You can enable the "Auto Refresh" toggle switch to keep the job list updated as the job status progress:



When the test finishes, its status will change to "COMPLETE", the logs will be stored locally and eventually uploaded to a Google Storage bucket. A cloud icon in the "Logs" column means that the logs have been uploaded to Google Storage and are no longer stored locally.

Note that for each test run there'll be two entries in the jobs table: one representing the test suite (`control.fwupd_update` in the image above) and another representing the test proper (`fwupd_FirmwareUpdate` in the image).

Once the job has finished you can check the results. If the logs haven't been uploaded yet, you can check them directly by clicking on the "Logs" icon for the test job (not the suite). From there, navigate to the "`status.log`" file:



Index of /results/1057-moblab/192.168.231.25/

../		
debug/	22-Feb-2023 09:46	-
fwupd_FirmwareUpdate/	22-Feb-2023 09:46	-
host_info_store/	22-Feb-2023 09:46	-
host_keyvals/	22-Feb-2023 09:46	-
lucifer/	22-Feb-2023 09:46	-
sysinfo/	22-Feb-2023 09:46	-
control	22-Feb-2023 09:46	1766
control.srv	22-Feb-2023 09:46	316
job.serialize	22-Feb-2023 09:46	4141
keyval	22-Feb-2023 09:46	377
result_summary.html	22-Feb-2023 09:46	71K
status	22-Feb-2023 09:46	512
status.log	22-Feb-2023 09:46	610

The "`status.log`" file shows a summary of the test result:

```
INFO ---- ---- kernel=5.15.94-16358-gae97cc7d22a2 timestamp=1677059177␣
↪localtime=Feb 22 09:46:17
START ---- ---- timestamp=1677059182 localtime=Feb 22 09:46:22
START fwupd_FirmwareUpdate fwupd_FirmwareUpdate timestamp=1677059182␣
↪localtime=Feb 22 09:46:22
ERROR fwupd_FirmwareUpdate fwupd_FirmwareUpdate timestamp=1677059183␣
↪localtime=Feb 22 09:46:23        No FW releases found for␣
↪4a69bff2d096b361b6e8a070a012728aff92538e (Unifying Receiver)
END ERROR fwupd_FirmwareUpdate fwupd_FirmwareUpdate timestamp=1677059183      ␣
↪ localtime=Feb 22 09:46:23
END GOOD ---- ---- timestamp=1677059183 localtime=Feb 22 09:46:23
```

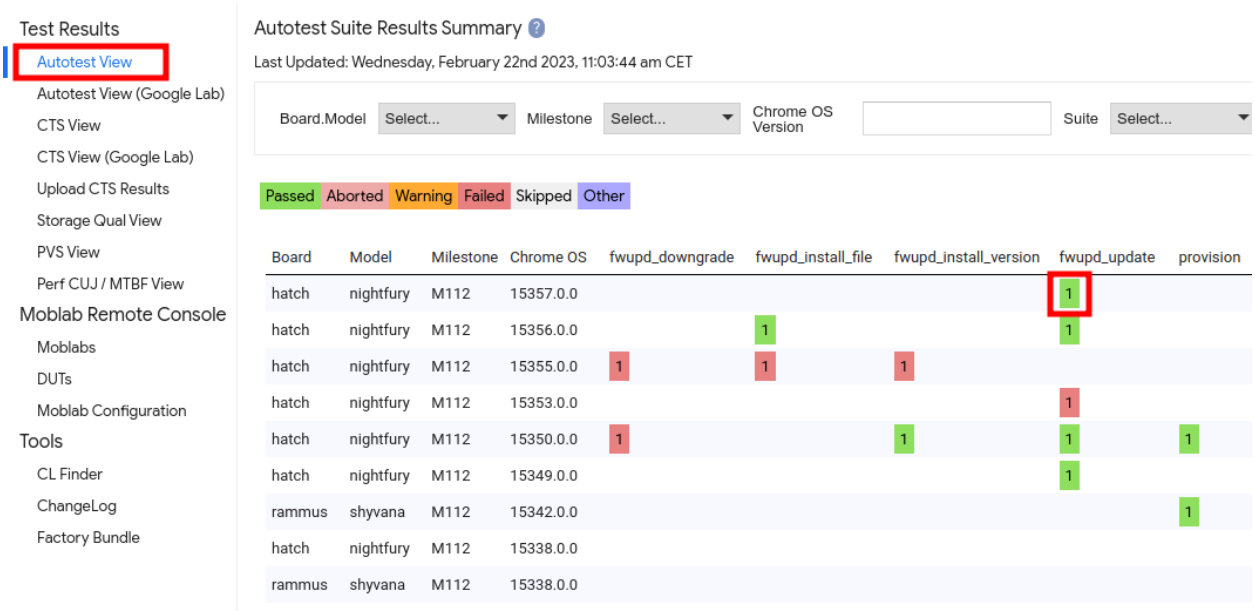
In this case it shows that the FirmwareUpdate test failed because the selected device doesn't have any FW releases available.

This other case shows a successful run:

```
INFO ---- ----        kernel=5.15.94-16358-gae97cc7d22a2 timestamp=1677060168␣
↪localtime=Feb 22 10:02:48
START ---- ---- timestamp=1677060173        localtime=Feb 22 10:02:53
START fwupd_FirmwareUpdate fwupd_FirmwareUpdate timestamp=1677060173␣
↪localtime=Feb 22 10:02:53
GOOD fwupd_FirmwareUpdate fwupd_FirmwareUpdate timestamp=1677060174␣
↪localtime=Feb 22 10:02:54        completed successfully
END GOOD fwupd_FirmwareUpdate        fwupd_FirmwareUpdate timestamp=1677060174␣
↪localtime=Feb 22 10:02:54
END GOOD ---- ---- timestamp=1677060174 localtime=Feb 22 10:02:54
```

When the logs are uploaded to Google Cloud Storage, you can check them through CPCon, accessing with your partner domain account and clicking on the "Autotest View" option. From there you can see all the test suites run on every Chromebook type and ChromeOS milestone:

Clicking on any of the test suite results will lead you to a detailed summary of that suite. All the different test runs



for that particular Chromebook type and ChromeOS version will show up listed either as a "Non-Passed test" or as a "Passing test". The most recent test run will be the last log listed:

Clicking on any of the logs will direct you to the Google Storage bucket directory containing the logs for that test run:

From there you can download the "`status.log`" file. The directory and file structure is the same as in the locally stored logs.

16.5 How to get debug information

If the "`status.log`" report doesn't give enough information about a failed test, you can also download the "`debug/client.0.DEBUG`" file, which contains the full log including debug messages. Additionally, the "`sysinfo/messages`" file contains the full system log that can also be useful to investigate a bug.

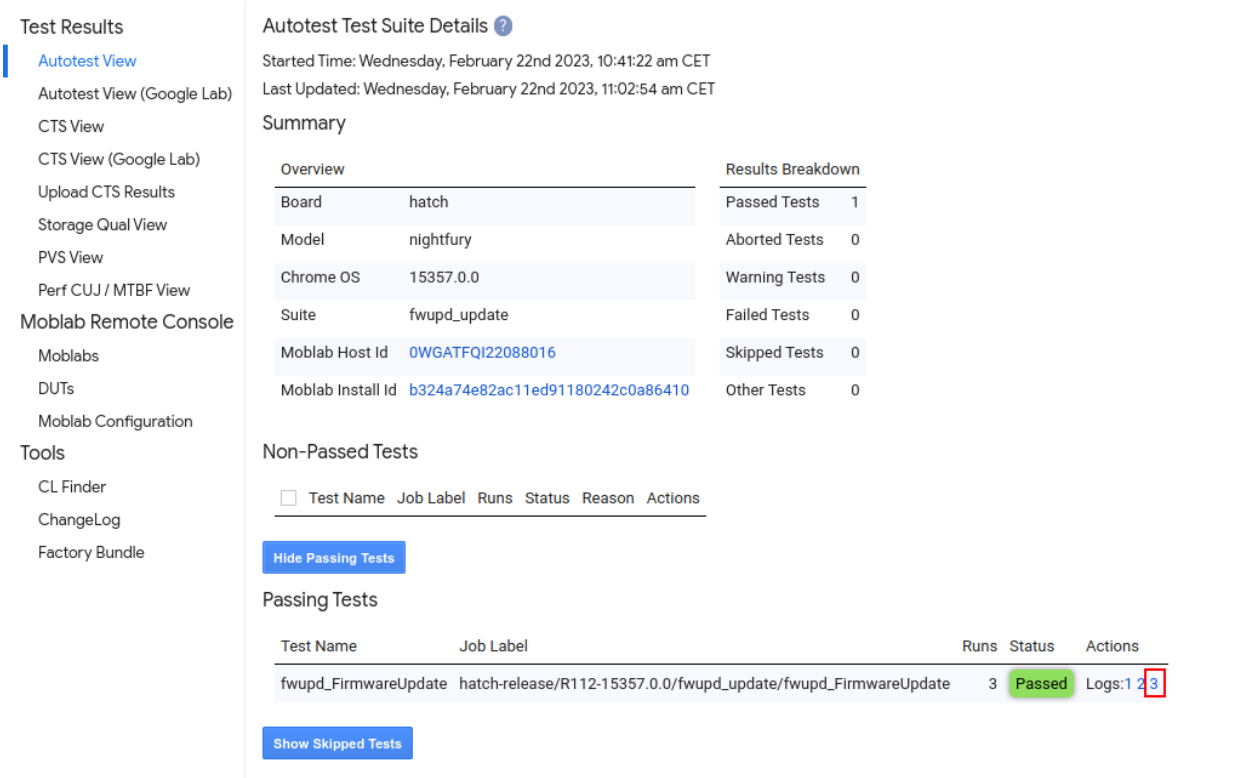
For instance, this "install file" test failed with this message in the status.log file:

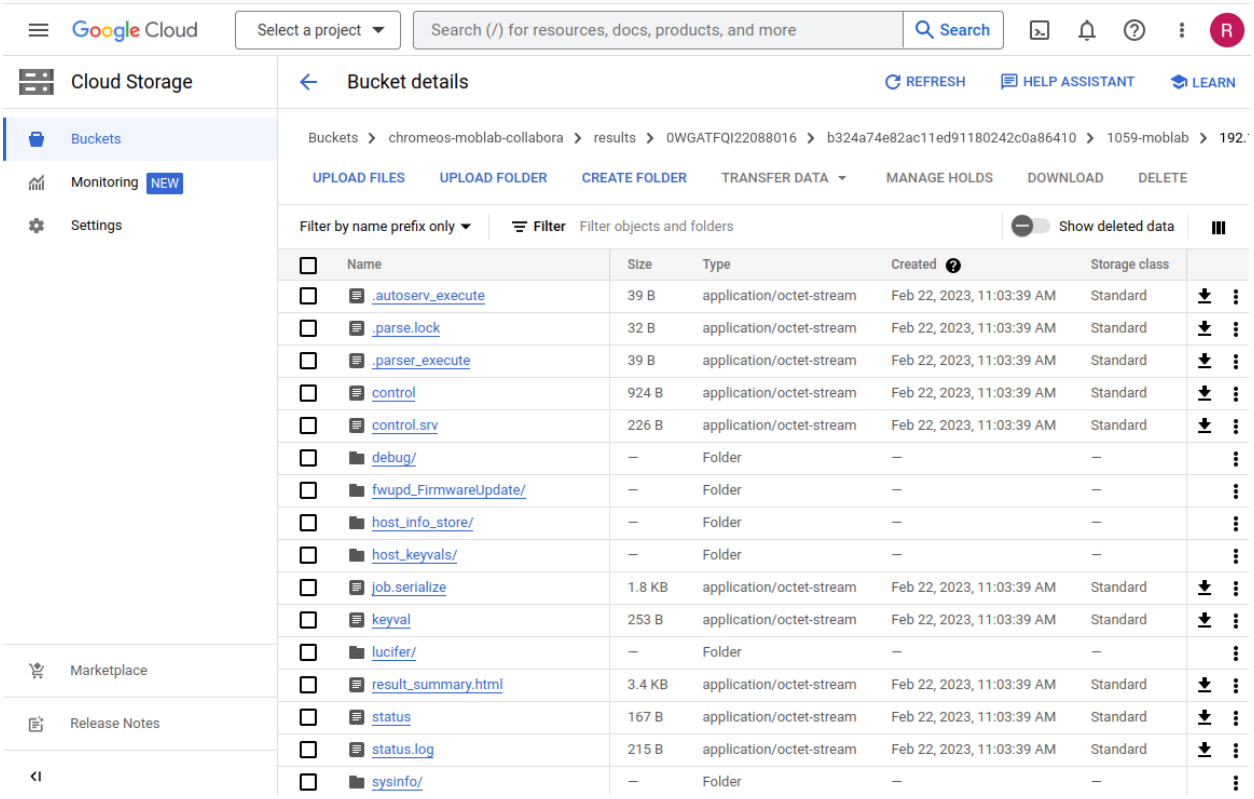
```
FAIL fwupd_FirmwareInstallFile fwupd_FirmwareInstallFile timestamp=1678199270
localtime=Mar 07 14:27:50 Command <CACHE_DIRECTORY='/var/cache/fwupd' fwupdmgr
local-install --json --allow-older --allow-reinstall
https://fwupd.org/downloads/
↪b1f9760a573b19f7d6eca46cdc04389c89649c803943d5d5e1681fe60f
b83f61-EPOSADAPT1x5T.cab b1f4e48ebea67bc554a73407241469583497b8e6> failed,␣
↪rc=1,
Command returned non-zero exit status
```

Which doesn't tell the reason. Checking the "`debug/client.0.DEBUG`" file shows the whole `fwupdmgr` output:

```
* Command:
      CACHE_DIRECTORY='/var/cache/fwupd' fwupdmgr local-install --json --allow-
      older --allow-reinstall https://fwupd.org/downloads/
```

(continues on next page)


Test Results
Autotest View
Autotest View (Google Lab)
CTS View
CTS View (Google Lab)
Upload CTS Results
Storage Qual View
PVS View
Perf CUJ / MTBF View
Moblab Remote Console
Moblabs
DUTs
Moblab Configuration
Tools
CL Finder
ChangeLog
Factory Bundle
Autotest Test Suite Details
Started Time: Wednesday, February 22nd 2023, 10:41:22 am CET
Last Updated: Wednesday, February 22nd 2023, 11:02:54 am CET
Summary
Overview
Board hatch
Model nightfury
Chrome OS 15357.0.0
Suite fwupd_update
Moblab Host Id 0WGATFQI22088016
Moblab Install Id b324a74e82ac11ed91180242c0a86410
Results Breakdown
Passed Tests 1
Aborted Tests 0
Warning Tests 0
Failed Tests 0
Skipped Tests 0
Other Tests 0
Non-Passed Tests
Test Name Job Label Runs Status Reason Actions
Hide Passing Tests
Passing Tests
Test Name Job Label Runs Status Actions
fwupd_FirmwareUpdate hatch-release/R112-15357.0.0/fwupd_update/fwupd_FirmwareUpdate 3 Passed Logs:1 2 3
Show Skipped Tests


Google Cloud
Select a project
Search (/) for resources, docs, products, and more
Search
Cloud Storage
Bucket details
REFRESH
HELP ASSISTANT
LEARN
Buckets
Monitoring NEW
Settings
Marketplace
Release Notes
Buckets > chromeos-moblab-collabora > results > 0WGATFQI22088016 > b324a74e82ac11ed91180242c0a86410 > 1059-moblab > 192.
UPLOAD FILES
UPLOAD FOLDER
CREATE FOLDER
TRANSFER DATA
MANAGE HOLDS
DOWNLOAD
DELETE
Filter by name prefix only
Filter
Filter objects and folders
Show deleted data
Name Size Type Created Storage class
.autoserv_execute 39 B application/octet-stream Feb 22, 2023, 11:03:39 AM Standard
.parse.lock 32 B application/octet-stream Feb 22, 2023, 11:03:39 AM Standard
.parser_execute 39 B application/octet-stream Feb 22, 2023, 11:03:39 AM Standard
control 924 B application/octet-stream Feb 22, 2023, 11:03:39 AM Standard
control.srv 226 B application/octet-stream Feb 22, 2023, 11:03:39 AM Standard
debug/ – Folder – –
fwupd_FirmwareUpdate/ – Folder – –
host_info_store/ – Folder – –
host_keyvals/ – Folder – –
job.serialize 1.8 KB application/octet-stream Feb 22, 2023, 11:03:39 AM Standard
keyval 253 B application/octet-stream Feb 22, 2023, 11:03:39 AM Standard
lucifer/ – Folder – –
result_summary.html 3.4 KB application/octet-stream Feb 22, 2023, 11:03:39 AM Standard
status 167 B application/octet-stream Feb 22, 2023, 11:03:39 AM Standard
status.log 215 B application/octet-stream Feb 22, 2023, 11:03:39 AM Standard
sysinfo/ – Folder – –

(continued from previous page)

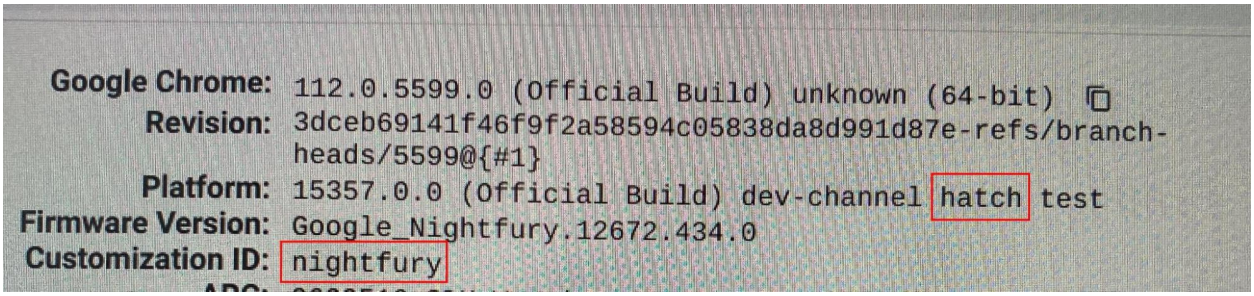
```
↪b1f9760a573b19f7d6eca4
     6cdc04389c89649c803943d5d5e1681fe60fb83f61-EPOSADAPT1x5T.cab
     b1f4e48ebea67bc554a73407241469583497b8e6
  Exit status: 1
  Duration: 0.028001070022583008

  stdout:
  {
    "Error" : {
      "Domain" : "FwupdError",
      "Code" : 8,
      "Message" : "No supported devices found"
    }
  }
```

This means that the file used for the update isn't compatible with the specified device.

16.6 FAQs

16.6.1 How to find the board type of a Chromebook or Chromebox?

In the Chromebook/Chromebox, open Chrome and enter `chrome://version` in the URL bar. In the info screen that will appear, the board type and variant will show up in the **Platform** and **Customization ID** fields, respectively:



16.6.2 How to log into the DUT through SSH?

Reference: How to SSH to DUT without a password

In some scenarios it could be useful or needed to run certain console commands in the DUT or to retrieve data from it that can only be accessed through a terminal interface. If the DUT is running a test ChromeOS release, it'll have an SSH daemon running so you can connect to it remotely.

Requirements:

- A Linux PC with an SSH client installed. If running Ubuntu, it can be installed with the following command: sudo apt install ssh
- The Linux host must be able to ping the DUT. The easiest way to achieve this is to connect the DUT and the Linux PC to the same local network. If the DUT is connected to a Moblab through a wired connection you can also connect to the Linux PC using a wireless connection

Assuming the host you're connecting from is running Linux, in order to connect to the DUT you'll need to download the ChromeOS test keys and configure your ssh client properly following these steps:

1. Download the SSH keys from this link and copy them to `~/.ssh` in the Linux host
2. Set the correct file permissions for the private key:

```
chmod 0600 ~/.ssh/testing_rsa
```

3. Get the IP address of the DUT. If the DUT is connected to multiple networks, we need the IP address of the NIC that's connected to the Linux PC network. To list the available connections and their IP addresses go to virtual terminal 2 in the DUT (`[ Ctrl ] [ Alt ] [ → ]`) and type: `ip -4 -br a`

```
localhost ~ # ip -br -4 a
lo               UNKNOWN        127.0.0.1/8
wlan0            UP             192.168.1.137/24
arc_ns0@if2      UP             100.115.92.129/30
arc_ns1@if2      UP             100.115.92.133/30
eth0             UP             192.168.231.25/24
```

   In this example, if the DUT is connected to the Linux PC network through a wireless link, then we can check that the PC can ping the DUT at 192.168.1.137.

4. Add the following to `~/.ssh/config`:

```
Host dut
  HostName $IP_ADDRESS
  User root
  CheckHostIP no
  StrictHostKeyChecking no
  IdentityFile ~/.ssh/testing_rsa
  ControlMaster auto
  ControlPersist 3600
```

   Where $IP_ADDRESS is the IP address of the DUT

16.6.3 How to check the list of peripherals detected by fwupd?

If, for debugging purposes, you need to check the current list of peripherals that fwupd is detecting, you can do so by running this command in a DUT terminal:

```
fwupdmgr get-devices --json
```

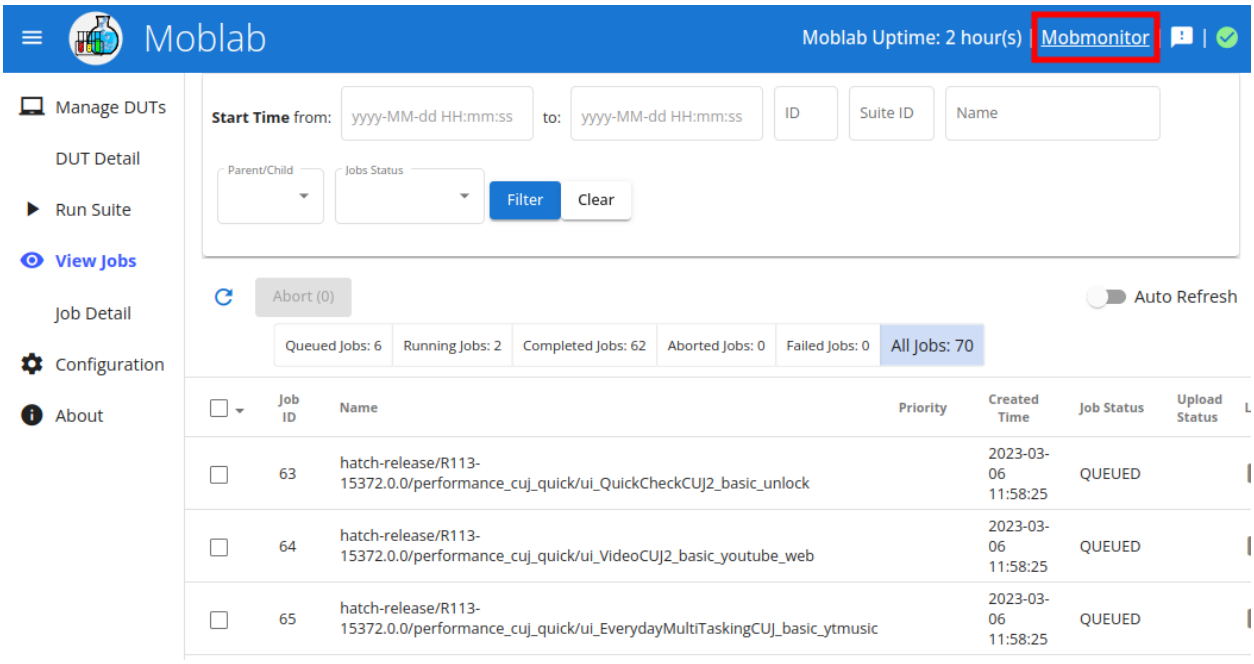
16.6.4 How to stop the DUT from rebooting automatically

If the DUT was provisioned (updated) using Moblab, it will check for an Ethernet link and it will reboot if it doesn't find one. Make sure to keep the DUT connected to the Moblab network using an Ethernet link.

When you are done testing with the Chromebook, reflash it with a recovery image to prevent it from restarting continuously when not connected to a network through the Ethernet port.

16.6.5 How to send debug information

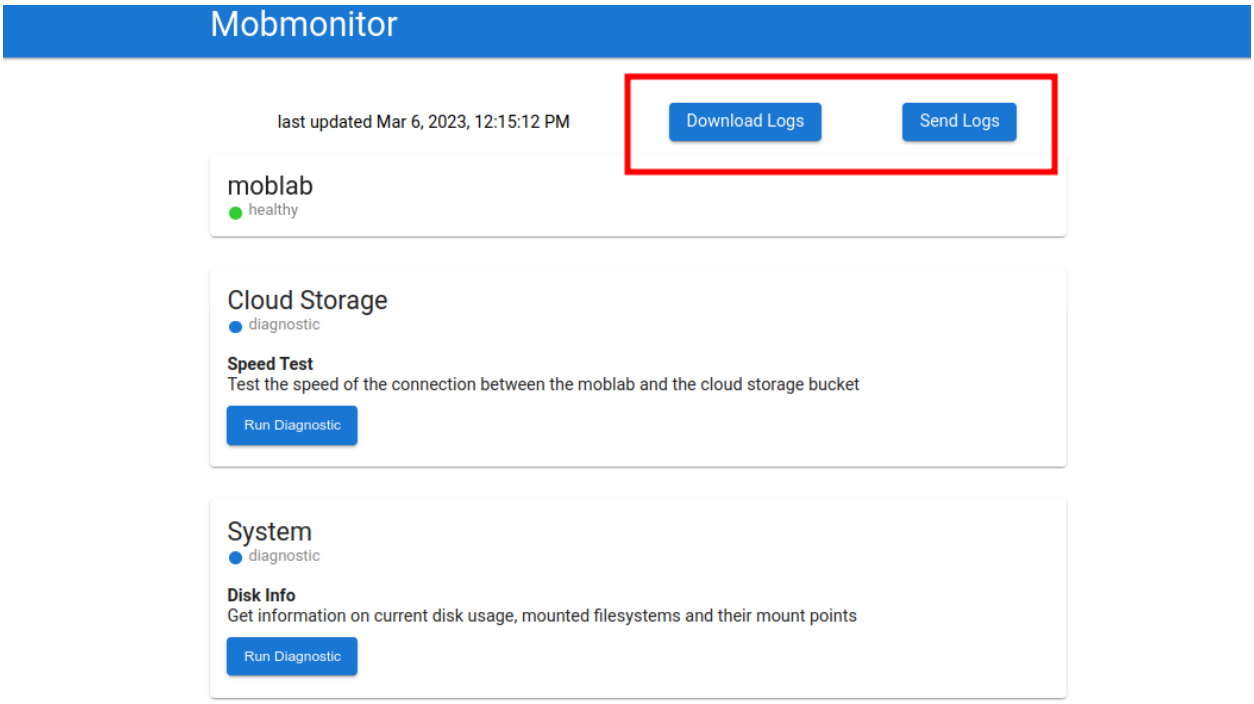
In case something goes wrong when launching or running a test, you can get the complete Moblab logs for debug through the Mobmonitor menu:



Then you can either download the logs or send them to your Cloud Storage bucket:

The logs will be compressed as a .tgz file.

Alternatively, if there's an issue with Moblab you can report it by issuing a buganizer ticket using this template and filling in the details.


Mobmonitor
last updated Mar 6, 2023, 12:15:12 PM
Download Logs
Send Logs
moblab
healthy
Cloud Storage
diagnostic
Speed Test
Test the speed of the connection between the moblab and the cloud storage bucket
Run Diagnostic
System
diagnostic
Disk Info
Get information on current disk usage, mounted filesystems and their mount points
Run Diagnostic