Project no. 732546

DECODE

DEcentralised Citizens Owned Data Ecosystem

D4.1.DECODE OS Software Development Kit (SDK)

Version Number: V0.5

Lead beneficiary: Dyne.org

Due Date: April 2017

Author(s): Dr. Denis Jaromil Roio, Dr. Vincenzo Katolaz Nicosia, Ivan Jelinčić
Editors and reviewers: James Barrit (Thoughtworks), Mark de Villiers (Thingful)

Dissemination level:		
PU	Public	X
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Approved by: Francesca Bria

Date: 30/04/2017

This report is currently awaiting approval from the EC and cannot be not considered to be a final version.

Contents

1. Introduction

The DECODE OS SDK define an development kit for modules that can be developed and distributed by other consortium partners, granting system consistency to manage the life-cycle of deployments and offer different levels of access control to the network. The SDK can be used to build, test and profile individual software applications on top of the DECODE OS, both locally and remotely on the continous integration infrastructure. It also provides various installers for the latest version of DECODE's software stack produced by other WPs, with documentation for deploying the DECODE NODE software in various conditions. This building block is aimed at strenghtening the control of lower layers of the DECODE networking stack and the security of the whole DECODE system , as well its resource usage and versatility of deployment in various situations. The DECODE OS exposes ISO/OSI layers 2, 3 to application layer 7 across a fine tuned set of access controls abiding to well established UNIX standards. It constitutes a base of execution for smart-rule language applications and at the same time grants their integrity via code signing and process isolation.

This document provides the reader with all notions needed to build a working GNU/Linux operating system distribution that can run on a variety of computing architectures, including several embedded ARM platforms, desktop PC, laptops and servers. It is targeted for use by the DECODE project to produce the DECODE OS, which consists of a certified build of the base system pre-installed with DECODE applications to run a DECODE node.

For a detailed description of the DECODE architecture one should refer to the DECODE whitepaper, which defines the role of nodes as computing units that can execute rules published on the DECODE blockchain in a safe, controlled environment. The DECODE node can execute rules published by operators, process the data of participants and provide operators with datasets that are pseudonymous and targeted by attributes.

The DECODE OS is the building block to create an integral computing environment for the smart rules, where the node application and the smart rules can function as they are supposed to and as defined by their code, without the interference of third-parties. With the DECODE OS the distributed software will always be runnning in an isolated container and, whether installed as a virtual machine or on a host system or on an embedded device, it will be reliably running the same code packaged for these different platforms.

The approach detailed in this document does not grant the security nor the integrity of the system per-se, but it allows to establish a clear and linear process of deployment for the code produced. The DECODE SDK is then the set of tools needed to produce an integrated environment that is fine-tuned to run DECODE applications. This know-how is generic enough to be applied to other projects with similar needs and constitutes a reliable base for "DevOps" activities aimed at building complex systems made of different inter-related software components and an underlying operating system.

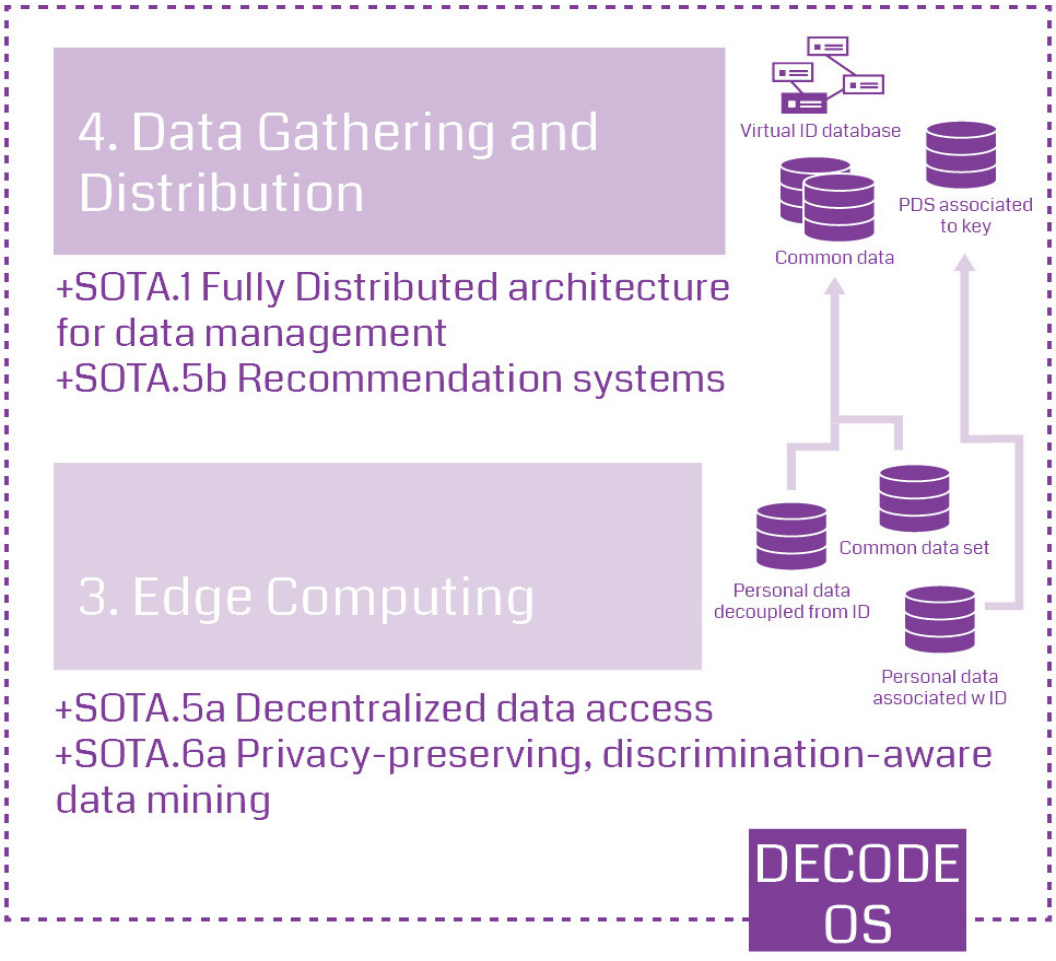
Composition of this deliverable

This deliverable addresses a technical audience that is familiar with the setup of GNU/Linux base system for embedded as well virtual machine computing. It is divided in three main sections:

1. Architectural considerations on security and integrity
2. Developer's Manual for packaging software inside the DECODE OS
3. Developer's Manual for releasing the DECODE OS for various platforms

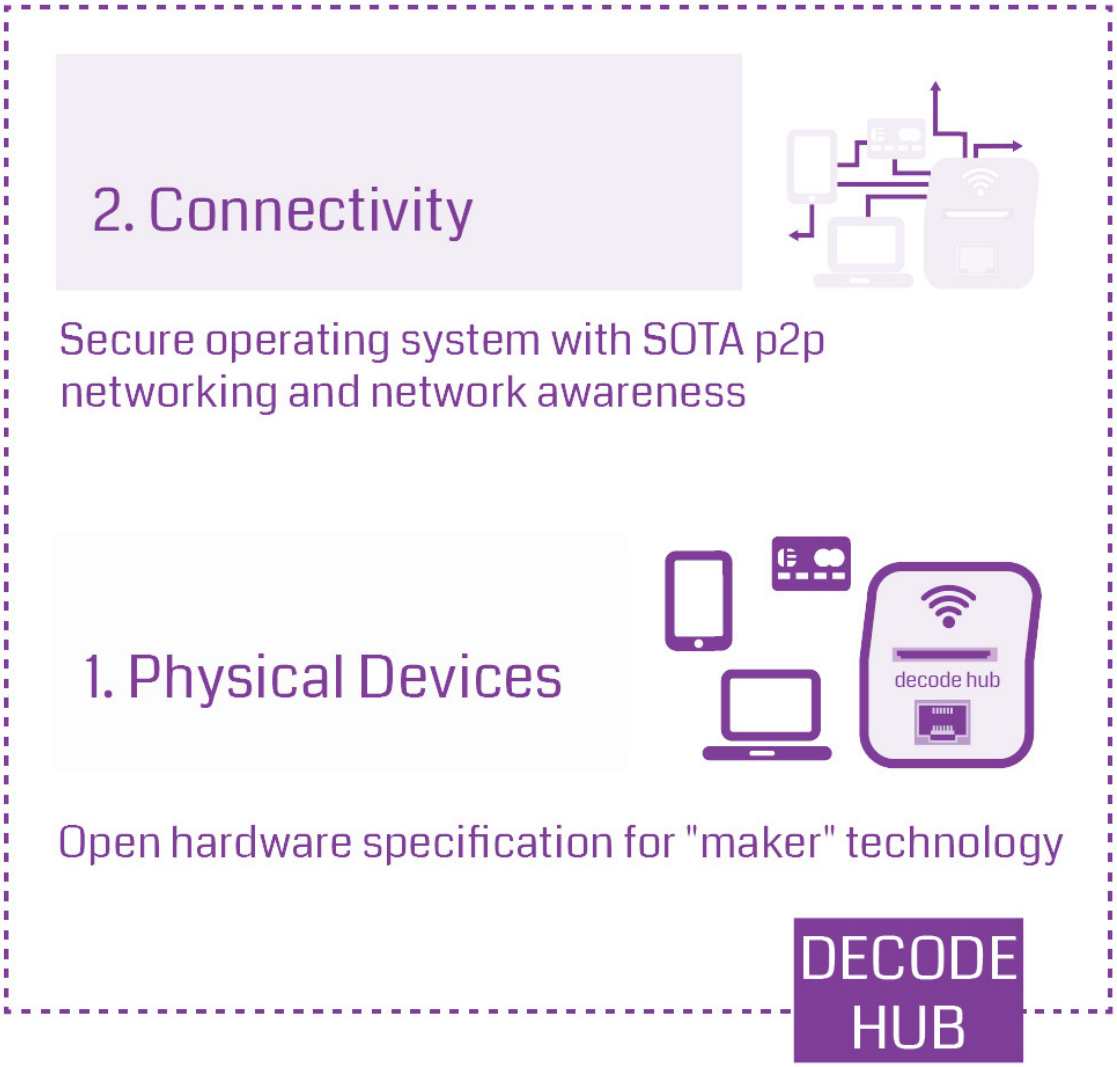
All readers are advised to go through section 1 which also informs the work in WP1 on the architecture of DECODE.

Application software developers are advised to go through section 2 which is mostly providing a development kit for the work being done in WP3 when developing applications that run securely on the DECODE OS and may need future maintenance through its packaging system.



Overview of software application functionalities in DECODE

Engineers in charge of "DevOps" deployment in test and production environment should refer to section 3 which is evolving across the work done in WP4 and aims at satisfying deployment needs occurring in most situations where DECODE OS will be installed.



Overview of hardware and deployment functions in DECODE

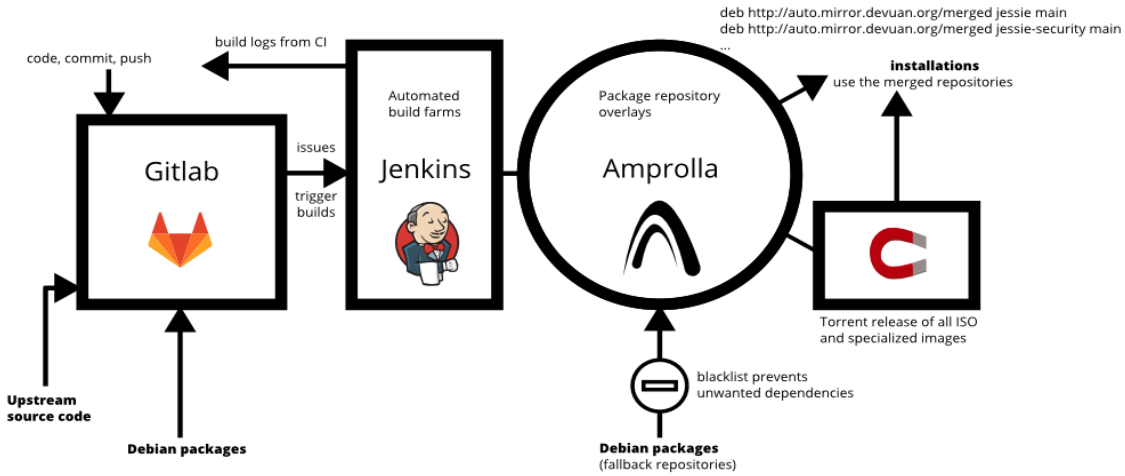
Evolution of this document

This document will be evolving through the lifespan of the DECODE project and beyond and, through its evolution it will improve with testing to include detailed documentation for techniques adopted and mechanisms to govern the continuous integration system, the packaging of software and the security and privilege escalation model adopted. This document is versioned using semantic versioning (semver.org) and its conceived so that, once reaching version 1.0.0, it should be a publishable book to guide readers through the deployment of the DECODE build process for the realisation of a secure environment for the execution of the DECODE infrastructure.

2. Continuous Integration

While all final software implementations in DECODE should be covered by unit tests, the integration aspects still need to be addressed especially considering we are aiming at the production of an entire operating system distribution. It becomes then very important to have a "continuous integration" (CI) setup that can build and integrate the latest *stable* and *testing* versions of software applications into ready to deploy images, to immediately feed into the LEAN testing cycle a result of the recent efforts and trickle down changes made on the application layer to be integrated immediately (at least daily) on the layers of DECODE's lowest architecture parts.

In order to implement the CI the DECODE project will adopt for its OS the base infrastructure provided by the Devuan project, an universal base operating system that is developed in-house at Dyne.org in cooperation with a large worldwide community of enthusiasts. The flow and each component of the CI is best described by the figure below:



Graph of Devuan/DECODE the continuous integration infrastructure

From left to right, the entry point for software application developers (Upstream source code) is the GIT repository where also Debian and Devuan packages can be merged, modified and updated: a common maintenance platform is made available for hosting all private and public sourcecode activity. Such a platform is already present and in use on https://gogs.dyne.org yet, at a later stage, it may find its final colocation also as a group on Github or on another private installation of Git. It doesn't really matters much where is placed as the distributed nature of GIT allows to displace such an infrastructure and have it duplicated on-site also on the premises of each organisation involved in the development.

Triggered in an easy and intuitive way from the GIT repository where developers interact are the builds executed by Jenkins, a popular and rather standard enterprise grade platform to monitor the health of builds and eventually apply code analysis tools as well reproducible builds extensions (see next section). Jenkins is a mandatory step before making the built software available through the Amprolla package repository which then distributes updates to installed nodes.

In case developers prefer to add another building step better than Gitlab and Jenkins (using Travis-CI or Semaphore, Coverage etc) this is of course possible and we can, in the course of the project, accommodate the logic that Jenkins will follow to pick up code that is built and sent in the *stable* or *testing* branches of the package repository.

What follows after Jenkins is "Amprolla" a component that overlays the base Devuan repository (and the underlying Debian repository) with the packages that are added by DECODE to the base operating system in order to implement the full OS for DECODE nodes. These packages will be built in the popular APT format and hence all upstream software should contain a */debian* project folder detailing the build process for the standard APT tools that will then deploy builds and archive their versioning, changelog and authors responsible of changes.

Here below an example of contents of a typical */debian* package specification directory.

```
/debian
├── changelog
├── compat
├── control
├── copyright
├── postinst
├── README.source
├── rules
└── source
    └── format
```

To populate this directory one can start from an existing package and use the *dh* tool that is explained in the following chapter.

Lastly there are installer iso files and SD card images and even live images to be run off USB (static nodes) that can be build out of the packages on Amprolla using the last part of this SDK (libdevuansdk and its extensions). The range of possibilities are detailed in the section "Architecture targets" of this document.

Reproducible builds

Among the future plans for DECODE is the ambition to setup reproducible builds for the DECODE software application packages.

With free software, anyone can inspect the source code for malicious flaws, but DECODE will provide binary packages ready to be started for operating nodes and there will be no way to determine if a binary package has been really built from a particular version and branch of sourcecode without any modification. Reproducible builds (also known as "deterministic" builds by some) empower anyone to verify that no flaws have been introduced during the build process, by comparing byte-for-byte two or more identical binary packages built from a given source.

Reproducible builds are a set of software development practices that create a verifiable path from human readable source code to the binary code used by computers. With reproducible builds, multiple parties can redo this process independently and ensure they all get exactly the same result. We can thus gain confidence that a distributed binary code is indeed coming from a given source code. What made the

recent Volkswagen emissions scandal possible is software that has been designed to lie about its sensors in a lab environment. Having the source code under public scrutiny would have made adding such a misfeature only a little more difficult. Without reproducible builds, it is hard to confirm that the binary code installed in the car was actually made using the source code that has been verified. (from https://reproducible-builds.org)

To have reproducible builds for the DECODE OS and its software applications distributed in bytecode should be the last step to consolidate its SDK to produce a fully verifiable build. Reproducible builds will not be available immediately, but we foresee their realisation (and clearly recognise their usefulness) as we get close to the end of the DECODE development cycle. In any case reproducible builds will not entail any additional work for software developers (exception made for the bug reports that may derive from it) and mostly will be related to the DevOps infrastructure setup DECODE adopts.

3. Privilege Escalation Model

Intelligence communities use the concept of "need to know" as a way to minimize information leaks and only grant access to secrets when it is absolutely necessary for the performance of an agent's action. In computer security, the same concept can be translated for *privilege escalation*: the need for a program to execute actions not normally available to its calling user. Often used offensively to gain control of a target machine or extract information, privilege escalation has also legitimate use, for example to enable a desktop user to perform administrative actions such as rebooting a running system or upgrading its software packages.

In case of the DECODE privilege model for data access the intention is to maintain defense in depth by granting the integrity to the underlying operating system in addition to the cryptographic security model implemented by the application layer, because a node that is compromised by a rogue process escalating its privileges could anyway jeopardise the validity of results provided by the smart rules computations happening on it.

In order to avoid such a situation, DECODE adopts a privilege escalation model based on a hard-coded hash-table that is inscribed into its operating system. The standard C (POSIX.1b) implementation for this model is called *sup*, currently maintened by Dyne.org and well known to the security community especially in the context of embedded development. Its website is visible at https://sup.dyne.org

The *sup* binary runs as root (with suid bit on) to facilitate the privilege escalation needed to execute certain programs as superuser, for instance to open a listening port below 100, for a web or an SMTP server. In order to regulate the privilege escalation *sup* handles a few conditions to authorize execution, all hard-coded in a static ELF binary.

An example config.h is here:

```
struct rule_t {
    int uid; // user identifier
    int gid; // group identifier
    const char *cmd; // command
```

```
    const char *path; // fixed path to the command (or * for any)
    const char *hash; // SHA256 hash of the binary
};

static struct rule_t rules[] = {
{ USER, GROUP, "pgld", "*", "47c04509 1bd152cbc2d4a2a3bf1b7b42d5b185d39ccba08b2aa51a21" },
{ USER, GROUP, "webui", "*", "0063bb188264830b28129416a8ee070ac7c894f66b2b8b9f355ff617" },
{ USER, GROUP, "route", "*", "d72557 1c48ac45130c15274204ffae07bb9607b06610e8f7a26d89e5" },
{ USER, GROUP, "dnscrypt-proxy", "*", "ce69a33f43f399b9b4e50db21eb5cbf73ba291544d65d5cc209cf607" },
{ USER, GROUP, "modprobe", "*", "ecf184748467fe4c6d561e3e3813f028f03ba9b7affe10855df4538b" },
{ USER, GROUP, "ebtables", "*", "a4e4b6de1fa9527892aae8d3e9f0a6e60c6d99725ba5acee7aaf7751" },
{ USER, GROUP, "xtables-multi", "*", "adee44923e1f20d6cb168ea5f51e5abefc5d07f476fd82e79fb6fa75" },
{ USER, GROUP, "ifconfig", "*", "d13a85ca313d24261a520cd4f33087679edded86308dd6a23047cfb0" },
{ USER, GROUP, "nmap", "*", "8b2cc682660baf70513f1328753 1c869f1c23346688814a4fd65caf" },
{ USER, GROUP, "dhcpd", "*", "b45a15c8b4f3e4114ec825a5dcc961f32f0f44d0fe9ed5cda8d8f4de" },
{ USER, GROUP, "kill", "*", "0f277f0cdc17952156a63a246f1934503 1e27e02baf8434f7a4aa6ce" },
{ USER, GROUP, "sysctl", "*", "239eb3eafeb39ed3a28420c34164fa2f11daf3993cc139352d8ddbeb" },
{ 0 },
};
```

For the DECODE OS an "allow list" is provided to *sup* at the time of building the ISO installers and SD card images, so that running "*sup* application" will escalate the privileges of application without the need of any password, in case the application is included in the "allow list". In order to extend the list of applications allowed to escalate privileges within the execution environment of the DECODE OS, a new build of it must be provided. Binaries change their contents across builds that are targeted to different architectures (i386, amd64, armhf, arm64, etc.) therefore the hashes in the allow list will change for each targeted build.

The *sup* binary itself is authenticated by the "root" signature of each DECODE OS which is shipped along with the ISO or installer or SD card or virtual machine image formats. The cryptographic signature of DECODE OS verifies all the binaries of the system in their current state as released, including *sup* whose unique binary footprint changes at every change of its privilege escalation hash table. Therefore any change in any binary that can escalate its privilege via *sup* will lead to a change in the *sup* binary itself, which will lead in a change in the signature of the whole DECODE OS.

This way of authorising the execution of software applications is not only implying that a particular binary can escalate privileges, but also that a particular build of that program can. It means that even if the binary is substituted with another one by modifying the installer image of the DECODE OS, it will not be able to be executed with privileges, denying the possibility to run effectively counterfeited DECODE OS images, as this check is combined with the check of a cryptographic signature for the whole OS image.

Security implications

This privilege escalation model improves the overall security of the DECODE OS as a system where it can be pre-determined what needs to run with privileges, harnessing the main difference with desktop systems that are interactively configured and whose privilege model can be changed by users. When this model is used in combination with a security enhanced Linux kernel, it constitutes a very robust foundation for the realisation of a secure operating system that executes a predetermined set of computing processes.

DECODE's privilege escalation model provides overall better security against the breach of the system integrity, but it cannot be considered a blanket solution for the security and integrity of the whole DECODE architecture. It is a reasonably strong measure to insure the integrity of operations at the core of the DECODE OS, granting integrity for the execution machine.

Technical implications

For the developers involved, DECODE's privilege escalation model means that every software application, be it binary compiled or scripted executable, needs to be registered with its hash inside the OS at build time. To change such an application a whole new build of the DECODE OS needs to be triggered. This is of course a setting necessary to produce production ready installers, but it can be de-activated for testing environments.

The other main implication is that each and every program executed with privileges cannot arbitrarily execute other programs that are configured at runtime. For instance a privileged process cannot have a variable that can be configured at runtime to call another program. An exception to this rule can be the implementation of a mechanism to drop privileges inside the program, which should be throughly reviewed: such implementations exist in C, however the need for such an exception should be avoided.

Regarding updates, patching a running system would entail a reboot after downloading a new OS version, which will be distributed in a "squashed" bundle (squashFS). This wouldn't mean to reinstall the entire OS, but to download and reboot into a new signed binary system.

Political implications

The model by which privilege escalation is granted to processes running in the DECODE OS is intentionally resistant to changes, which need to be implemented and recorded in the history of revisions before the build process. This model forces a clear negotiation of privileged process, allowing a debate on such choices that is extended to all technical stakeholders. The process of privilege escalation should be in fact seen as a political negotiation about which algorithms are allowed to deal with more delicate parts of the operating system.

4. Packaging software

In order to include software inside the DECODE OS installers and queue it for updates, it is necessary to package it into a standard format for distribution. The packaging format of choice for DECODE is APT, since it is a battle-tested standard which also grants compatibility with ongoing effort for reproducible builds and integrity tests via the CI.

This section will explain how to package software using the d1h helper which facilitates the activity of packaging with the possibility to start from existing Devuan or Debian packages, plus it streamlines the operation of packaging using the git-buildpackage format, which among other formats available for APT packaging is the one of choice for the DECODE OS.

After having read this technical section, the reader should be able to import a Devuan or Debian package in DECODE and include it in the CI workflow.

Pre-requisites

Before you start, you need the following packages installed in your system:

- debhelper and all the usual applications needed to build Debian packages (lintian, build-essential, cdbs, dh-make, fakeroot, and so on…)
- git
- git-buildpackage
- gpg (for package signing)

On top of that, you should also have:

- a working owner account on gogs.dyne.org/decode
- the environment variables DEBEMAIL and DEBFULLNAME set, respectively, to your email and to your full name, e.g.:

  ```
  DEBEMAIL=katolaz@freaknet.org
  DEBFULLNAME="Vincenzo (KatolaZ) Nicosia"
  export DEBEMAIL DEBFULLNAME
  ```

- a working Internet connection

Installing d1h

The package *d1h* is currently in experimental. In order to install it, you should add the following line to your /etc/apt/sources.list:

```
deb http://packages.devuan.org/devuan/ experimental main
```

and perform an "apt-get update". Then, you install *d1h* as follows:

```
# apt-get install -t experimental d1h
```

d1h workflow

As an example, we assume in this guide that we want to import the package "adduser" to. This is a relatively small and self-contained package to start with.

1) Decode repository

Create a new project on your git.devuan.org account, called "adduser". This new project will be associated to a git URL in the form:

```
git@gogs.dyne.org:USERNAME/adduser
```

where USERNAME is your username on gogs.dyne.org. In the following, we will denote by DECODE_REPO the URL associatet to your project on gogs.dyne.org.

2) Create the cache

Create the *d1h* repository cache:

```
$ d1h cache
```

This will download the full list of repos hosted at https://anonscm.debian.org, parse it, and save a cache in ~/.d1h/cache.

It is not necessary to create the cache every time you use *d1h*, but it is recommended in order to keep updated the whole list of available packages on the Debian upstream. When a package is not found or just before forking it, one should refresh the cache to verify that the version retrieved is really the latest. Also if a package is not found, but supposed to exist, then the cache should be refreshed.

3) Search the repository

Search the upstream git repo of "adduser". Here *d1h* comes handy:

```
$ d1h search adduser
##################################
##### cache last updated on  Fri  7 Apr 21:19:30 BST 2017
##################################
adduser/adduser: https://anonscm.debian.org/cgit/adduser/adduser.git/
$
```

Notice that d1h has found only one repository matching our query "adduser". Note down the corresponding URL

https://anonscm.debian.org/cgit/adduser/adduser.git/

The "search" function accepts multiple strings (separated by spaces). If more than one string is provided, "search" will look for a repo matching *all* of them.

4) Import the repository

Enter the folder where you would like to keep the package repo, and import the "adduser" repo:

```
$ d1h import https://anonscm.debian.org/cgit/adduser/adduser.git/
```

You can alternatively provide to "import" a destination folder as second argument ("import" corresponds to a "clone" in git):

```
$ d1h import https://anonscm.debian.org/cgit/adduser/adduser.git/ ./pkg_adduser
```

In this case, the repo "adduser" will be imported under the folder "./pkg_adduser".

When "import" has finished, you will find the repo "adduser" under the destination folder (or in the current directory, if no destination folder was provided).

5) Link the repository

Now we need to link our new "adduser" repository with the DECODE_REPO we have created on git.devuan.org. The "link" command resets the origin of your repo, to let it point to, e.g., DECODE_REPO. After "link" has finished, DECODE_REPO will be the origin of your local "adduser" repo. Just give the command:

```
$ d1h link DECODE_REPO
```

where DECODE_REPO is the git URL of your project on git.devuan.org, e.g. something like

```
git@git.devuan.org:KatolaZ/adduser
```

This URL will be indicated in the home page of your project on git.devuan.org.

6) Prepare the suite

Now we should prepare the suite for which we would like to build the package. We assume that we want to build the package for "decode". However, new packages never go directly into a release main repository (with the only exception of the "experimental"). Instead, a new package for "decode" is normally added to "decode-proposed". So we need to prepare the suite "decode-proposed":

```
$ d1h prepare decode-proposed
```

This step should be done only once.

(*) For the git-savvies: the "prepare" action creates a new branch, in this case named "suites/decode-proposed", and adds the appropiate "debian/gbp.conf" file to it.

7) Switch to the suite

Before we start working on modifying the package for decode-proposed, we must first "switch" to the correpsonding suite: "switch" is simply a checkout to the corresponding branch.

```
$ d1h switch decode-proposed
```

8) Modify the package

Now that we have switched into the suite "decode-proposed", we must reset the repo to the correct version of the package. If we want to repackage an existing software in decode or devuan, we must ensure to start the work on exactly the same version of the package that is already available in decode, or if not then in devuan, or if not then in Debian. In this case, the version of "adduser" is only available in Debian and is 3.113+nmu3. One can easily check using:

```
$ apt-cache policy adduser
```

Luckily, the repo we have imported from Debian keeps track of this information through a tag. In order to retrieve the correct version of the package for decode, we can run:

```
$ git reset --hard tags/3.113+nmu3
```

You can now make all the changes you need to the package. In this specific case, we don't have to do anything in particular. We will just edit the "debian/control" file with our preferred editor, and put our name and email in the "Maintainer:" field. We will also remove the field "Uploaders:", and save the file.

When we are done with the changes, we just commit eveything:

```
$ git add --all
$ git commit
```

The "commit" command will open an editor asking to insert a message for the commit. In this case, we can just insert "DECODE version of adduser", save the file, and quit the editor.

9) Test the build

It is time to build the package. This can be done through the command:

```
$ d1h testbuild
```

This command will start building the package. If it complains about a missing package, just "apt-get install…" the missing dependences as needed, and run it again. If you have all the needed dependencies installed, the build will proceed and create the files following files:

```
adduser_3.113+nmu3_all.deb
adduser_3.113+nmu3_amd64.changes
adduser_3.113+nmu3.tar.gz
adduser_3.113+nmu3_amd64.build
adduser_3.113+nmu3.dsc
```

in the parent folder of the repo. This time the command "testbuild" will complain about not being able to sign the package, and it will terminate with a "testbuild error". You can safely ignore this error: it is due to the fact that you have not yet edited the file "debian/changelog". This is what we will do in a moment.

10) Check the package

You should now test the .deb package you have just created, e.g. by installing it:

```
# dpkg -i adduser_3.113+nmu3_all.deb
```

11) Stage the package

When you are sure that the package works as expected, it is time to assign a version number to it, using the command:

```
$ d1h stage 3.113+nmu3+devuan1.1
```

where 3.113+nmu3+devuan1.1 is the new version number. It is warmly recommended for Devuan-ised pacakges to retain the same version number of the corresponding Ddebian package, with a "+devuanX.Y" label appended, where "X" and "Y" indicate the Devuan major/minor version of the package.

The command "stage" runs "gbp dch" to assing a new version to the package. It also creates a tag for the new version, push the repo to origin (well, to DECODE_REPO), and push the tags.

12) Add the package to the CI

At this point, your package might be ready to be added to the Continuous Integration (CI) server at ci.devuan.org. In order to do that, you should contact a Devuan developer on the #devuan-dev IRC channel (freenode network), who will guide you through the process. This will usually involve moving your DECODE_REPO under the group "devuan-packages".

13) Build the package on the CI

After your package has been revised from other developers and added to the CI server, you will be able to request a build. This is done by opening a new special "Issue" on the gitlab page of the repository. This issue should have title equal to "build". It is very important to specify a tag as well, since the tag determines which branch will be built and where the resulting packages will be put. In our example, the package "adduser" should go in decode-proposed, so we will use the tag "decode-proposed". The issue must be assigned to "autobuild".

You can check the status and health of your builds in the DECODE section of the CI:

http://ci.devuan.org

Moreover, messages about build successes and failures are also reported on the IRC channel #decode-dev in real-time.

5. Simple distro kit

The "simple distro kit" is the part of the SDK dealing with the production of installers and running images of the DECODE OS, its intended audience is technical partners running 'devops' operations to facilitate the testing, integration and deployment of DECODE nodes. The "simple distro kit" is a unique build framework written to ease maintenance and production of various types operating system installers. It is modeled after the Devuan SDK and supports all its architecture targets:

- live bootable ISO images
- virtual machine images
- embedded ARM images

Different ARM architecture targets are:

- Raspberry PI 3 (raspi3 arm64)
- Raspberry PI 2 (raspi3 and raspi2)
- Raspberry PI 1 (raspi1 and raspi0)
- Chromebook Acer (chromeacer)
- Chromebook Veyron (chromeveyron)

Plus all ARM SoCs from Allwinner Technology, for which the best known products are the sunxi SoC series, such as the A10 (sun4i), A13 (sun5i) and A20 (sun7i) chips, which are very successful in the low-budget tablet market. Sunxi images in DECODE OS are powered by mainline 4.10.y Linux kernels, here a list of working targets as of now:

- Cubieboard 2
- Cubietruck
- Cubieboard4
- Cubieboard
- Cubietruck plus
- A10-OLinuXino-Lime
- A20-OLinuXino-Lime2
- A20-OLinuXino-Lime
- A20-OLinuXino-MICRO
- Bananapi
- Bananapro
- CHIP
- CHIP pro
- Lamobo R1
- Orangepi 2
- Orangepi
- Orangepi lite
- Orangepi mini
- Orangepi plus

- Orangepi zero
- q8 a33 tablet 1024x600
- q8 a33 tablet 800x480

The range of supported architectures will be extended in the future with particular attention to ARM64 targets due to their relevance as base architecture for the DECODE project. This section will be updated accordingly. The following section explaining how to use the SDK to create new installers collecting all necessary software for DECODE OS to run its services, it gives an inside look on its various parts and documents the workflow to be used when hacking on its code.

The SDK is designed in such a way that there are levels of priority within the scripts. First there is libdevuansdk, which holds the vanilla configuration, then come the various wrappers targeted around specific targets, and afterwards we optionally add more on top of it if we need to customize or override specific functions.

6. libdevuansdk

libdevuansdk is the core of any part of the simple distro kit (Devuan SDK). It holds the common knowledge between all of the upper SDKs such as *live-sdk*, *vm-sdk*, and *arm-sdk*. Simply put, it's a shell script library to unify the use and creation of various functions spread throughout Devuan's various SDKs.

Devuan's SDKs are designed to be used interactively from a terminal, as well as from shell scripts. libdevuansdk uses the functionality of the zuper zsh library, but it does not include it - one is required to include it in its respective SDK. However, libdevuansdk requires the following Devuan packages to be installed:

zsh
debootstrap
sudo
kpartx
cgpt
xz-utils

Workflow

Working with libdevuansdk splits into categories of what you want to do. *zlibs* are files separated into these "categories":

bootstrap Contains the functions for the bootstrap process. Creating a minimal debootstrap base, and making it into a tarball for later use so you don't have to wait for the debootstrap on every build.

helpers Contains the helper functions for libdevuansdk that make our lives a bit easier.

imaging Contains the functions necessary for creating raw dd-able images.

kernel Contains the functions for installing a vanilla kernel.

rsync Contains rsync functions.

sysconf Contains the default system configuration.

Usage

As libdevuansdk is not very helpful when being used on its own, its usage will be explained at later parts, for each specific SDK. The technical documentation of libdevuansdk can be found in its appropriate section.

7. The SDKs

As mentioned, libdevuansdk is our core library we wrap around. The currently existing wrappers are called *live-sdk*, *vm-sdk*, and *arm-sdk*. These facilitate the builds of liveCDs, virtual machines, and images for embedded ARM devices, respectively. Each of them has their own section in this manual.

Since all of these SDKs, along with libdevuansdk, hold a vanilla Devuan configuration, you might prefer not to change their code. For this case, a concept called *blends* was introduced. Blends are a simple way to customize the image before building it, allowing you to very easily add packages, kernels, and virtually anything you might want to do in the image.

Let's now continue explaining each SDK on its own, and afterwards we will learn how to incorporate the blends into the workflow.

7.1 arm-sdk

The *arm-sdk* is our way of facilitating builds for embedded ARM boards such as Allwinner-based CPUs, Raspberry Pis, Chromebooks, etc. It holds a knowledgebase for a number of embedded devices, and how to build according kernels and bootloaders.

Directory structure

arm-sdk's directory structure is separated into places where we hold our boards and their kernel configs, device-specific directories with firmware and/or configuration, and a lib directory (where we keep libdevuansdk and friends).

Obtaining arm-sdk

The SDK, like any other, should be obtained via git. The repositories are hosted on Devuan's Gitlab. To grab it, we simply issue a *git clone* command, and since it contains git submodules - we append *–recursive* to it:

$ git clone https://git.devuan.org/sdk/arm-sdk --recursive

Consult the README file to see what are the needed dependencies to use arm-sdk.

Keep in mind: what you cloned is a development (unstable) version of arm-sdk. To use a stable version, you chould checkout to a tagged version, i.e.:

```
$ cd arm-sdk
$ git checkout 0.8
```

See the git history to find out the latest version.

Using arm-sdk

After we've obtained the build system, we can now use it interactively. The process is very simple, and to build an image, we can actually use a oneliner. However, let's first learn how it works…

In arm-sdk, every board has it's own script located in the *boards* directory. In most cases, those scripts contain functions to build the Linux kernel, and a bootloader needed for the board to boot. This is the only difference between all the boards, which requires every board to have their own script. We are able to reuse the rootfs that is bootstrapped before. For our example, let's take the *Nokia N900* build script. To build a vanilla image for it, we simply issue:

```
$ zsh -f -c 'source sdk && load devuan n900 && build_image_dist'
```

This will fire up the build process, and after a certain amount of time we will have our compressed image ready and hashed inside the *dist* directory.

The oneliner above is pretty self-explanatory: We first start a new untainted shell, source the sdk file to get an interactive SDK shell, then we initialize the operating system along with the board we are building, and finally we issue a helper command that calls all the necessary functions to build our image. The *load* command takes an optional third argument which is the name of our blend (the way to customize our vanilla image) which will be explained later. So in this case, our oneliner would look like:

```
$ zsh -f -c 'source sdk && load devuan n900 dowse && build_image_dist'
```

This would create an image with the 'dowse' blend, which is available by default. To find out what blends are available, consult the *sdk* file and look into the blend_map hashtable.

The *build_image_dist* command is a helper function located in libdevuansdk that wraps around the 8 functions needed to build our image. They are all explained in the technical part of this manual.

7.2 live-sdk

The *live-sdk* is used to build bootable images, better known as Live CDs. Its structure is very similar to *vm-sdk* and is a lot smaller than *arm-sdk*.

Directory structure

Unlike arm-sdk, in live-sdk we have no need for specific boards or setups, so in this case we only host the interactive shell init, and libraries.

Obtaining live-sdk

The SDK, like any other, should be obtained via git. The repositories are hosted on Devuan's Gitlab. To grab it, we simply issue a *git clone* command, and since it contains git submodules - we append *–recursive* to it:

$ git clone https://git.devuan.org/sdk/live-sdk --recursive

Consult the README file to see what are the needed dependencies to use live-sdk.

Keep in mind: what you cloned is a development (unstable) version of live-sdk. To use a stable version, you chould checkout to a tagged version, i.e.:

$ cd live-sdk
$ git checkout 0.2

See the git history to find out the latest version.

Using live-sdk

After we've obtained the build system, we can now use it interactively. The process is very simple, and to build an image, we can actually use a oneliner. However, let's first learn how it works…

Since we do not have any needs for specific boards, with loading we don't specify a board, but rather the CPU architecture we are building for. Currently supported are i386 and amd64 which represent 32bit and 64bit, respectively. To build a vanilla live ISO, we issue:

$ zsh -f -c 'source sdk && load devuan amd64 && build_iso_dist'

This will fire up the build process, and after a certain amount of time we will have our ISO ready and inside the *dist* directory.

The oneliner above is pretty self-explanatory: We first start a new untainted shell, source the sdk file to get an interactive SDK shell, then we initialize the operating system along with the architecture we are building, and finally we issue a helper command that calls all the necessary functions to build our image. The *load* command takes an optional fourth argument which is the name of our blend (the way to customize our vanilla image) which will be explained later. So in this case, our oneliner would look like:

$ zsh -f -c 'source sdk && load devuan amd64 refracta && build_image_dist'

This would create an image with the 'refracta' blend default. To find out what blends are available, consult the *sdk* file and look into the *blend_map* hashtable.

The *build_iso_dist* command is a helper function located in libdevuansdk that wraps around the 9 functions needed to build our image. They are all explained in the technical part of this manual.

7.3 vm-sdk

The *vm-sdk* is used to build VirtualBox Vagrant boxes, and virtual images for emulation, in QCOW2 format, which is a nifty byproduct of building a Vagrant image. Its structure is very similar to *live-sdk* and is the smallest of the SDKs.

Directory structure

As with live-sdk, in vm-sdk we have no need for specific boards or setups, so in this case we only host the interactive shell init, and libraries.

Obtaining vm-sdk

The SDK, like any other, should be obtained via git. The repositories are hosted on Devuan's Gitlab. To grab it, we simply issue a *git clone* command, and since it contains git submodules - we append *–recursive* to it:

$ git clone https://git.devuan.org/sdk/vm-sdk --recursive

Consult the README file to see what are the needed dependencies to use vm-sdk.

Keep in mind: what you cloned is a development (unstable) version of vm-sdk. To use a stable version, you chould checkout to a tagged version, i.e.:

$ cd live-sdk
$ git checkout 0.2

See the git history to find out the latest version.

Using vm-sdk

After we've obtained the build system, we can now use it interactively. The process is very simple, and to build an image, we can actually use a oneliner. However, let's first learn how it works…

Also with vm-sdk we do not have any needs for specific boards, however, we also do not create any non-amd64 images, so we don't need to specify an architecture to the load command either. To build a vanilla Vagrant box, we issue:

$ zsh -f -c 'source sdk && load devuan dowse && build_image_dist'

This would create an image with the 'dowse' blend default. To find out what blends are available, consult the *sdk* file and look into the *blend_map* hashtable.

The *build_iso_dist* command is a helper function located in libdevuansdk that wraps around the 10 functions needed to build our image. They are all explained in the technical part of this manual.

8.Blends

Introduction

In the Devuan SDK, a *blend* is the preferred way we use to make customizations to the vanilla image. Using blends we can very easily create different flavors of our image, by easily including/excluding certain software packages, files, or anything we wish to do as a matter of fact. Blends can become a very quick way of creating entire new derivatives of the vanilla distribution we are building.

This time, we will take the *live-sdk* as a blend example. In live-sdk we provide a blend called devuan-live} which is the blend we use to create a live ISO image of Devuan, along with adding a Xfce desktop to it, and adding various packages needed for the desktop to fully function. The blend's files are contained within their own directory in live-sdk.

Configuration

Any SDK requires a single file to act as a blend. This file is also a zsh script, and, at the very least, it must contain two functions called:

blend_preinst()
blend_postinst()

These functions are your pathway to expanding your blend into whatever you would like to do. The *preinst* function is usually called right after bootstrapping the vanilla root filesystem, and the *postinst* function is called near the very end, just before packing or compressing the image. These two strategic places should be enough to do changes within the image. If this is not enough, blends also allow you to simply override *any variable or function* contained within libdevuansdk or the sdk you are using.

Our *devuan-live* blend is such an example. It is a somewhat expanded blend, not contained within a single file, but rather a directory. This allows easier maintenance and makes the scripts clearer and cleaner.

Adding and removing packages

When we want to aadd or remove specific packages to our build, we have to override or append to libdevuansdk's arrays. The array for packages we want installed is called *extra_packages*, and the array for packages we want purged is called *purge_packages*. In the devuan-live blend, these can be found in the *config* file located inside the devuan-live blend directory. Keep in mind that these arrays could already contain specific packages, so you are advised to rather append to them, than overriding them.

If the packages you want to install are not available in the repositories, you still have a way of automatically installing them. All you need to do to take care of it is at some point in your blend - copy your .deb files to the following directory:

$R/extra/custom-packages/

And when that is done, just call the function *install-custdebs*

Creating a blend

Rather than explaining theory, you are best off viewing the blend files that are provided with *live-sdk*. It is a fairly simple blend and should give you enough insight on creating your own blend. Here are some important guidelines for creating a blend:

- The blend should always contain at least two functions

This means you must provide *blend_preinst* and *blend_postinst* in your blend. They don't even have to do anything, but they should be there. These two functions open the path for you to call any other functions you created for your blend.

- When overriding functions, make sure they provide a result that doesn't break the API

Breaking the API may result in unwanted behavior. You should always study well the functions you are planning to override and figure out if it is safe to override them in the way you want. The same goes for any variables as well.

- Any arguments used after the blend name when loading from the SDK are free for you to use in the blend.

This means you can use anything **after $4** inside your blend if you require passing arguments to it.

These are some of the more important guidelines. There is plenty more tricks and quirks, but it's easy to find out once you read a blend or two on your own…

Enable the blend

To use your blend in the first place, you need to make the sdk know about it. To make this work, you need to append the path to your new blend inside the **blend_map** of the *sdk* file:

```
blend_map=(
    "devuan-live"     "$R/blends/devuan-live/devuan-live.blend"
    "heads"           "$R/../heads.blend"
    "ournewblend"     "$R/blends/newblend/new-blend.blend"
)
```

As you can see, the map is a key-value storage. So you can have an alias (name) for your blend, and just use that to point to the path of the blend. The blend file will be sourced by the sdk once it is told to do so.

A configuration file

For having a finer-grained control of what goes into our build, we can create a config file for our blend. From here we can easily control any configurable aspect of our blend, such as packages that go in or out, the blend name, and much more. **Make sure you source this file from your blend.**

Adding and removing packages was abstractly mentioned earlier: it goes into two separate arrays holding package names. To add packages, we append to the **extra_packages** array, which would look like this:

```
extra_packages+=(
    my_new_package
    foo
    bar
    baz
)
```

This would install these four packages, along with the ones predefined in either libdevuansdk or the sdk you are using. You may also want to see which those are in case you wish to exclude them, but they are sane and useful utilities which should be included in your build if possible. Overriding all those packages, you would need to reset the whole array, so you would simply issue this:

```
extra_packages=(
    my_new_package
    foo
    bar
    baz
)
```

As you can see, we no longer have the +=, but rather only =, which means we are not appending to the array, but rather redefining it.

All of the above applies as well for removing packages, but in this case - the array is called **purge_packages**.

Custom packages

If you want to install deb packages that aren't in any repositories, put them in the blend directory and simply add them to another array in the configuration file. The contents of the arrays are the paths to the debs, relative to this configuration file:

```
custom_deb_packages=(
    yad_0.27.0-1_amd64.deb
    palemoon_27.2.0~repack-1_amd64.deb
)
```

To trigger installation of these packages, you will need to copy them to $R/extra/custom_packages, and then call the **install_custdebs** function somewhere from your blend.

Custom files

Any files you want to add to the system to override what's there by default you can add using a *rootfs overlay*. Create a directory inside your blend directory called *rootfs-overlay* and simply put files inside it. The directory structure is absolute to the image we are building. For example

what's in "rootfs-overlay/etc/" would end up in the "/etc" of our final image. See *hier(7)* from the Linux manpages for more explanation on this directory hierarchy.

If you end up with any files here, to actually copy them, you will need to cp -f it, or rsync it if you prefer.

The .blend file

We listed a path to the .blend file in our first step. We need to create this file now.

Start your blend file with the following, so the sdk is aware of the environment:

```
BLENDPATH="${BLENDPATH:-$(dirname $0)}"
source $BLENDPATH/config
```

The minimum blend should contain two functions: **blend_preinst** and **blend_postinst**. These functions are called at specific points in the build, where they give the most power: just after bootstrapping the vanilla system, and just before packaging the final build, respectively.

blend_preinst

A preinst function can look like this:

```
blend_preinst() {
    fn blend_preinst
    req=(BLENDPATH R)
    ckreq || return 1

    notice "executing blend preinst"

    add-user "user" "pass"
    cp -fv "$BLENDPATH"/*.deb "$R/extra/custom-packages" || zerr
    install-custdebs || zerr
}
```

So as you can see, the preinst function will add a new user with the credentials user:pass, it will copy our custom debs where they can be used, and finally it will trigger their installation.

The fn, req, ckreq part on the top of the function is a safety check for the function that is enabled by zuper. It allows us to check if variables are defined when the function is called and fail if it is wrong. You should utilize this as much as possible. The zerr calls are used to exit if the function fails.

blend_postinst

A postinst function can look like the following:

```
blend_postinst() {
    fn blend_postinst
    req=(BLENDPATH strapdir)
    ckreq || return 1
```

```
    notice "executing blend postinst"

    sudo cp -vf "$BLENDPATH"/rootfs-overlay/* $strapdir || zerr

    blend_finalize || zerr
}
```

This function would copy the rootfs-overlay to the strapdir (which holds our image's filesystem) and it would call the blend_finalize function. By default this function doesn't exist, but it's an example so you can see you can call your own functions as well. You can define them within the blend file.

Using a blend

As explained in previous chapters, you can use your blends through the interactive SDK shell. In live-sdk the *devuan-live* blend is placed under

$R/blends/devuan-live/devuan-live.blend

If you take a look at live-sdk's *sdk* file, you can see it in the *blend_map*. Using a new blend requires you to add it to this map in the same manner. The map is key-value formatted, and on the left you have an alias of your blend, and on the right you have a script you have to write. It can either be the blend itself or any helper file you might need to initialize your blend.

After you've added it to the blend map, you simply initialize the sdk, and use the same *load* command we learned earlier, while appending the blend alias and any optional argument.

```
$ zsh -f
$ source sdk
$ load devuan amd64 devuan-live <these> <arguments> <we> <can> <use> <in> <the> <blend
>
```

And we've initialized our *devuan-live* blend. It's always good to add a *notice()* call to your blend to signal it's been loaded successfully.

After this is done, we simply build the image the same way we learned before:

```
$ build_iso_dist
```

Consult the live-sdk chapter for this.

Conclusion

This document provides the first guidelines for a process that will evolve through the lifespan of the DECODE project and first and foremost provides detailed instructions for developers contributing to DECODE at various levels of its architecture. Most of this process will be operated by Dyne.org developers in the first half of the project, while this document will be always available for the necessary knowledge transfer that has to be operated to other partners and to advanced pilots willing to be in charge of the DECODE OS development, perhaps adapting it to their needs or including some other software and a different certification root.

The early stage of this document and its specialisation should be considered in the context of a very early technical deliverable for DECODE, accompanied by the setup of most of the infrastructure necessary to produce the DECODE OS. The intention with this early release is clearly that of enabling a LEAN process of development since the very beginning of the project and as such this deliverable should really be considered the start of an operative manual which will reach a stable and more complete format as subsequent milestones of the DECODE project are reached.