Container Generator - general information

Container generation tool

The tool for automatic generation of LXC container is written in Python *(genContainer.py)*. The tool generates container root file system, configuration, and launcher in the specified location in the file system based on provided XML description of the container. Three directories are generated for each container inside *(rootfs_path)/container/(container_name):*

- ***rootfs*** - generated container's root file system.
- ***launcher*** - generated bash script launching desired executable.
- ***conf*** - generated lxc container configuration.

The container generator tool is placed in ***meta-containers/recipes-containers/container-generator/files/src***. To run generator you need to provide rootfs files system, which will be post-processed and xml configuration file. NFS and rootfs on USB do not support linux capabilities. This means running secure/unprivileged containers that need capabilities will not properly run from NFS or USB. For that you will need to boot from flash.

```
Usage:
__main__.py [options] [files]

Example:
__main__.py -r ~/user/rdk/some_path_to_rootfs -s -t
version=debug,SOC_VER=arm_16.4 lxc_conf_DIBBLER.xml
    will generate secure DIBBLER container at given rootfs for debug version
with architecture arm_16.4


Options:
  -h, --help            show this help message and exit
  -r ROOTFS, --rootfs=ROOTFS
                        rootfs directory where container dir and its files
                        will be generated
  -s, --secure          Create unprivileged container
  -t TAGS, --tags=TAGS  tags
  -e, --sharedRootfs    Containers share rootfs with host
  -S, --sanity          Enable sanity check
```

XML container description

Each container has its own xml description. Based on this description, the generator tool prepares the environment to run one process or more processes in the container. The XML tree is explained in few sections (nodes) below. To make creating containers easier, the lxc_conf_EXAMPLE.xml file was created, which describes possible configurations. It can be used as a template for creating a new container configuration.

1. LXCParams

   This node take care of creating launcher script with all parameters needed to start lxc-execute command. It is possible to pass an argument to launcher script. The launcher script is just a bash wrapper on lxc-execute, if you put the variables **$1, $2, ...** in ExecParams they will be replaced by argument passed to script. All LXCParams entries was describe in snippet presented below:

```
<!--
    LxcParams - parameters for start script needed to run lxc-execute command.
                Optionally an attribute "version" can be set that must match with the version
                passed via tool cmdline (debug, production or release)
    OPTIONS:
        ->LauncherName -name of launcher script
        ->ContainerName -name of the container, this parameter is optional.
                if name is not set the default name is taken from SandBoxName.
                This was created to make few containers from the same config.
        ->ExecName -name of binary or command to execute in container
        ->ExecParams - parameters for binary or command to execute
        ->StopFunction -should stop option in launcher should be created
        ->SystemdNotify -use systemd-notify to notify systemd that process is inited,
                Launcher script will wait for pidfile and notify systemd
                that service is initialized. Optional. create=yes must be set to use it.
            -> PidFile -pidfile to wait on before systemd notify. Optional.
            -> ProcessName -name of child process that will be looked for. Optional. By default
                            taken from ExecName.
        ->Output - Lxc-execute output options (enable = true or false). Optionally an attribute
"version"
                   can be set that must match with the version passed via tool cmdline
                   (debug, production or release)
            ->LogFile -path to log file
            ->LogPriority -Log level for lxc.Possible log levels
                    FATAL ALERT CRIT ERROR WARN NOTICE INFO DEBUG TRACE
        ->Attach -describes the process, which will be attached using lxc-attach
            -> PramName -name of param, which will need to be passed to launcher to trigger
attaching.
            -> ExecName -name of binary or command to attach in container
            -> ExecParams - parameters for binary or command to attach
            -> SystemdNotify -use systemd-notify to notify systemd that process is inited,
                Launcher script will wait for pidfile and notify systemd
                that service is initialized. Optional. create=yes must be set to use it.
                              -> PidFile -pidfile to wait on before systemd notify. Optional.
                     -> ProcessName -name of child process that will be looked for. Optional. By
default
                            taken from ExecName.
                     -> GroupName -process attached to container will spawn to given group.
Optional, if not set falls back to UserName.
                     -> UserName -process attached to container will spawn to given user.
Optional.
 -->
    <LxcParams version="debug">
        <LauncherName>start</LauncherName>
        <ExecName>/bin/sh</ExecName>
        <ExecParams>-c "while true; do echo test; sleep 5; done"</ExecParams>
        <StopFunction enable="true"></StopFunction>
        <SystemdNotify create="yes">
            <PidFile>/run/component_name/pidfile.pid</PidFile>
        </SystemdNotify>
        <Output enable="false" version="debug">
            <LogFile>/var/log/lxc/logifile.log</LogFile>
            <LogPriority>ERROR</LogPriority>
        </Output>
        <Attach>
            <ParamName>attach_sleep</ParamName>
            <ExecName>/bin/sleep</ExecName>
            <ExecParams>100</ExecParams>
        <UserName>dhcpc</UserName>
        <GroupName>dhcpc</GroupName>
        </Attach>
        <Attach>
            <ParamName>attach_ls</ParamName>
            <ExecName>/bin/ls</ExecName>
            <ExecParams>-l /etc</ExecParams>
            <SystemdNotify create="yes">
                <PidFile>/run/componentName/pidfile.pid</PidFile>
                <ProcessName>/bin/ls</ProcessName>
            </SystemdNotify>
        <UserName>firewall</UserName>
        <GroupName>firewall</GroupName>
        </Attach>
        <Attach>
            <ParamName>attach_ls_no_pid_file</ParamName>
            <ExecName>/bin/sh -c /bin/something</ExecName>
            <ExecParams>-l /etc</ExecParams>
```

```
         <SystemdNotify create="yes">
             <ProcessName>/bin/something</ProcessName>
         </SystemdNotify>
     <UserName>firewall</UserName>
     <GroupName>firewall</GroupName>
     </Attach>
 </LxcParams>
```

Example of launcher:

```
#!/bin/sh

case $1 in
    start)
        rm -rf /run/component_name/pidfile.pid
        /usr/bin/lxc-execute ....
        while true
        do
            if [ -e "/run/component_name/pidfile.pid" ]; then
                break
            fi
            /bin/usleep 100000
        done
        systemd-notify --ready MAINPID=$(/usr/bin/lxccpid --ppid $! "/bin/sh" 2000)
        # Wait for systemd to assign the proper pid, before the main process exit.
        /bin/usleep 100000
    ;;
    attach_ls)
        rm -rf /run/componentName/pidfile.pid
        /usr/bin/lxc-attach ....
        while true
        do
            if [ -e "/run/componentName/pidfile.pid" ]; then
                break
            fi
            /bin/usleep 100000
        done
        systemd-notify --ready MAINPID=$(/usr/bin/lxccpid --ppid $! "/bin/ls" 2000)
        # Wait for systemd to assign the proper pid, before the main process exit.
        /bin/usleep 100000                      ;;
    attach_ls_no_pid_file)
        /usr/bin/lxc-attach ....
        systemd-notify --ready MAINPID=$(/usr/bin/lxccpid --ppid $! "/bin/something" 2000)
        # Wait for systemd to assign the proper pid, before the main process exit.
        /bin/usleep 100000
           ;;
    stop)
        /usr/bin/lxc-stop -n EXAMPLE
    *)
        exit 1
esac
```

2. LxcConfig
   This node takes care of container configuration placed in lxc.conf file. It is divided in a few sections such as:

   General options such as cgroup settings

```
 <!--
     LxcConfig - this section handles the lxc.conf
     OPTIONS:
         -> MemoryLimit -set a memory limit in bytes
         -> CapDrop -Specify the capabilities to be dropped in
              the container (put the capabilities separated by space).
         -> CapKeep -Specify the capabilities to be keep in
              the container (put the capabilities separated by space).
         -> CGroupSettings
              The following cgroups can be set using container generator.
              For more information please see
```

```
                http://man7.org/linux/man-pages/man5/lxc.container.conf.5.html
             Memory:
             -> MemoryLimit: cgroup.memory.limit_in_bytes
             -> MemoryMemSwapLimit: cgroup.memory.memsw.limit_in_bytes
             -> SoftMemoryLimit: cgroup.memory.soft_limit_in_bytes
             -> OOMControl: cgroup.memory.oom_control

             CPU:
             -> CpuShares: cgroup.cpu.share
             -> CpuRtRuntimeUS: cgroup.cpu.rt_runtime_us
             -> CpuRtRuntimePeriod: cgroup.cpu.rt_period_us

             CPUSet:
             -> CpusetCpus: cgroup.cpuset.cpus
             -> CpusetCpuExlusive: cgroup.cpuset.cpu_exclusive
             -> CpusetMems: cgroup.cpuset.mems
             -> CpusetMemExclusive: cgroup.cpuset.mem_exclusive
             -> CpusetMemHardwall: cgroup.cpuset.mem_hardwal
             -> CpusetMemoryMigrate: cgroup.cpuset.memory_migrate
             -> CpusetMemoryPreasureEnabled: cgroup.cpuset.memory_pressure_enabled
             -> CpusetMemoryPreasure: cgroup.cpuset.cpuset.memory_pressure
             -> CpusetMemorySpreadPage: cgroup.cpuset.cpuset.memory_spread_page
             -> CpusetMemorySpreadSlab: cgroup.cpuset.cpuset.memory_spread_slab
             -> CpusetSchedLoadBalance: cgroup.cpuset.cpuset.sched_load_balance
             -> CpusetSchedDomainLevel: cgroup.cpuset.cpuset.sched_relax_domain_level

             -> DeviceCgroup
                 The order in this list is kept and important to LXC.
                 -> DevicesAllow - lxc.cgroup.devices.allow, controls the devices cgroup
                     which lets you define which character and block devices
                     a container may access. Attribute name is optional and is used
                     to generate a comment line.
                 -> DevicesDeny - lxc.cgroup.devices.deny, controls the devices cgroup
                     which lets you define which character and block devices
                     a container cannot access. Use "a" to deny all devices.
                     Attribute name is optional and is used
                     to generate a comment line.
                 -> AllowDefaultDevices - generates access rules to default devices like
/dev/null
                     when enable=yes.
           -> GroupName - main process started in container will spawn to given group. Optional.
           -> UserName - main process started in container will spawn to given user. Optional.
           -> GroupNameRootFs - the group that will be set as owner to all files in container
rootfs.
                                Optional. If not set it falls back as follows: GroupName,
UserNameRootFs, UserName.
           -> UserNameRootFs - the user that will be set as owner to all files in container
rootfs.
                                Optional. If not set it falls back to UserName.
           -> LxcInclude -path to additional lxc.conf file
           -> Environment -each entry is one environment variable set in container
                only available via lxc-execute, not via lxc-attach!

   -->
    <LxcConfig>
        <MemoryLimit></MemoryLimit>
        <CapDrop></CapDrop>
        <CapKeep></CapKeep>
              <UserName></UserName>
        <GroupName></GroupName>
        <UserNameRootFs></UserNameRootFs>
        <GroupNameRootFs></GroupNameRootFs>
        <CGroupSettings>
              <CpuRtRuntimeUS>150000</CpuRtRuntimeUS>
            <MemoryLimit>12000000</MemoryLimit>
                       <DeviceCgroup>
              <DevicesDeny>a</DevicesDeny>
              <DevicesAllow name="/dev/dummy">c 123:0 rw</DevicesAllow>
              <AllowDefaultDevices enable="yes"/>
              </DeviceCgroup>
        </CGroupSettings>
        <LxcInclude></LxcInclude>
        <Environment version="release" SOC_VER="arm_16.3" OE_VER="2.0">
            <Variable>MALLOC_CHECK_=2</Variable>
        </Environment>
               (...)
    </LxcConfig>
```

<u>**Remark that environment variables defined in xml are only available to the main container process which is started via lxc-execute.**</u> **All secondary services started via lxc-attach do not have these vars available, because lxc-attach does not support it. However, lxc-attach, by default takes the environment from host. So you can still use the "Environment=" keys inside the service files for the services that start in container via lxc-attach.**

Network Configuration

If container should access the host network use network type ***none*****,** if container does not have to access the network use type ***empty*****.** For containers, which use internal network bridge to one of the local devices we support the **veth** network type.

```
     <!--
        Network node - specify what kind of network virtualization
                will be used for the container
                Supported network types:
                -> none: will cause the container to share the host's network namespace.
                -> empty: will create only the loopback interface.
                -> veth: a virtual ethernet pair device
                     is created with one side assigned to
                     the container and the other side attached
                     to a bridge specified by the lxc.network.link option
        OPTIONS FOR VETH TYPE:
        -> Name        -specify the network name inside container.
        -> Flags       -specify an action to do for the network:(up: activates the interface).
        -> Link        -specify the interface to be used for real network traffic.
        -> Pair        -specify a name for the network device belonging to the outside of the
container.
        -> HwAddr      -specify mac address for  virtual interface.
        -> IPV4        -specify the ipv4 address to assign to the virtualized interface.
        -> IPV4gateway -specify the ipv4 address to use as the gateway inside the container.
        -> IPV6        -specify the ipv6 address to assign to the virtualized interface.
        -> IPV6gateway -specify the ipv6 address to use as the gateway inside the container.
    -->
        <Network type="veth">
            <Name>veth0</Name>
            <Flags>up</Flags>
            <Link></Link>
            <Pair></Pair>
            <HwAddr></HwAddr>
            <IPV4></IPV4>
            <IPV4gateway></IPV4gateway>
            <IPV6></IPV6>
            <IPV6gateway></IPV6gateway>
         </Network>
```

D-Bus Configuration

In our setup D-Bus communication is based on domain socket. To run a process, which requires D-Bus communication, enable the D-Bus option in XML configuration.

The python generator will create config, which will create bind mounts between

D-Bus sockets in container and host, and mount binds the D-Bus library in container rootfs

```
    <!--
        Dbus node - this section handles the D-Bus configuration
        OPTIONS:
            -> enable
                -> false -do not enable dbus in container
                -> true -enable dbus in container
    -->
        <Dbus enable="true"></Dbus>
```

Rootfs configuration

This XML node creates lxc configuration for mount binds as well as create rootfs content.

- MountPoints - this section describes all directory, devices and files you can mount in the container rootfs.
    - In case of mount points the python tool supports two configurations with ready-write rootfs and with read-only rootfs. In case of ready-only rootfs some configuration files may be moved to read-writte storage. For such situation the special xml entry(SourceRoFs) was created to give alternative file path if the file path is different in read-only rootfs. For more information please see */etc/resolv.conf* entry and File System Structure.
- MoveContent - this section moves files or directories from host rootfs to container rootfs.
- LibRoBindMounts - In this section you can put all libs needed in your container, they will be bind mounted with read-only option.

```
<!--
        Rootfs node - this section handles the Rootfs configuration. If rootfs tag is not present or
                      create=no, then no rootfs will be created and the container will share its rootfs with
                      the host. Also the commandline option -e forces this "shared rootfs" feature regardless
                      of the contents of rootfs tag.
          OPTIONS:
            -> MountPoints - this node defines the mount points configuration
                     for rootfs inside container. Entry params are in typical fstab format,
                     description of them can be found on fstab man page.
                     If FsType, Dump or Fsck are not defined, their values will be set as follows: [None, 0 0].
                     Type of Entry depends on what you want to mount: directory or single file.
                     The Destination path is relative to container rootfs
                     Directory entries can have Unused subentries which declares some files inside as unused.
                     This is done to avoid error generation if these unused binaries refers to libraries
missing in container

            -> MoveContent - this node defines which content should be moved from host rootfs to container
rootfs.
                     Type of Entry depends what you want to move: directory or single file.
                     Two parameters needed:
                        -> Source - path to content to move
                        -> Destination - path to directory where it should be moved
                                into container -without "/" at the beginning)
            -> LibsRoBindMounts - This entry describe the libraries
                     which should be mount binded into container rootfs.
                     It requires only the name of library.

            -> AutoDev - This trigger enables(1)/disables(0) tmpfs in /dev directory, and creates initial
devices.

 -->
```

```
<Rootfs create="yes">
    <AutoDev>0</AutoDev>
           <MountPoints>

        <Entry type="dir" version="debug" SOC_VER="arm_16.4" OE_VER="2.1">
            <Source>/lib</Source>
            <Destination>lib</Destination>
            <FsType>none</FsType>
            <Options>ro,bind</Options>
            <Dump>0</Dump>
            <Fsck>0</Fsck>
            <Unused>lib/libfdisk.so.1.1.0</Unused>
        </Entry>

        <Entry type="file" version="release" SOC_VER="arm_16.3" OE_VER="2.0">
            <Source>/etc/resolv.conf</Source>
                                        <SourceRoFs>/var/volatile/resolv.conf</SourceRoFs>
                                        <Destination>etc/resolv.conf</Destination>
            <FsType>none</FsType>
            <Options>ro,bind</Options>
            <Dump>0</Dump>
            <Fsck>0</Fsck>
        </Entry>
    </MountPoints>

    <MoveContent>
        <Entry type="file">
            <Source>/sbin/ifconfig</Source>
            <Destination>sbin/ifconfig</Destination>
        </Entry>

        <Entry type="dir">
            <Source></Source>
            <Destination></Destination>
        </Entry>
    </MoveContent>

    <LibsRoBindMounts>
        <Entry version="debug" SOC_VER="arm_16.4" OE_VER ="2.1"></Entry>
        <Entry>libc</Entry>
    </LibsRoBindMounts>

</Rootfs>
```

How to create XML append to container configuration

Introduction

There is a possibility to append container configuration in platform layer by creating additional XML configuration. There can be several appends in different layers for one XML file. The main XML will be inoved first then the additonal XML appends will be attached to container configuration.

Adding XML append to image

```
 SRC_URI += "file://xml/lxc_conf_APPSERVICES_eos.xml"

do_install_append() {
(..)
                install_lxc_config secure lxc_conf_APPSERVICES_eos.xml
(..)
}
```

XML append example:

```
<!-- XML CONFIGURATION FOR AXACT -->

<!-- APPEND - name of append, it can be platform name or functionality name. -->
<CONTAINER APPEND="EOS" SandboxName="AXACT">

<!--
       LxcParams - parameters for start script needed to run lxc-execute command and lxc-
attached.
       The launcher script is generated based on this node.
       Appending XML allows to skip this node in case we useing launcher from main XML file.
       We can also overwrite the launcher script for the platform by adding new LxcParams in XML
append (all
       elements have to be provided again)
 -->

    <LxcParams>
        <LauncherName>axact</LauncherName>
        <ExecName>/usr/bin/cwmpclient</ExecName>
        <ExecParams>-l2 -c -p /mnt/secure_storage</ExecParams>
        <StopFunction enable="true"></StopFunction>
    </LxcParams>

<!-- We can append each entry in LxcConfig node, however we cannot remove entries from base XML
file as weel as overwrite them.
 If some node is not common for all platforms and functionalities, it was not trully common -->

    <LxcConfig>

        <UserNameRootFs>axact</UserNameRootFs>
        <GroupNameRootFs>axact</GroupNameRootFs>

        <Rootfs create="yes">
            <MountPoints>
<!-- /etc -->
                <Entry type="dir" version="debug">
                    <Source>/etc/ca-certificates</Source>
                    <Destination>etc/ca-certificates</Destination>
                    <Options>ro,bind,noexec,nosuid,nodev</Options>
                </Entry>
```

```
<!-- /mnt/nand -->
                <Entry type="dir" version="debug">
                    <Source>/mnt/nand/mw/labkey</Source>
                    <Destination>mnt/nand/mw/labkey</Destination>
                    <Options>rw,bind,noexec,nosuid,nodev</Options>
                </Entry>
            </MountPoints>
        </Rootfs>
    </LxcConfig>
</CONTAINER>
```

How to use tags inside container configuration

Introduction

Tags can be used on XML elements inside the XML container configuration to enable or disable them depending on passed tag value. This allows one XML configuration to be used by different SOC, OE and/or RDK versions. Because depending on that version some .so libraries might be required or not. The same goes for ENV variables, file mounts and device rights.

How it works

The allowed tags must be configured inside config.ini. Example configuration:

```
[TAGS]
version = debug, release
SOC_VER = arm_16.4, mipsel_16.4, arm_18.1
OE_VER = 2.1, 2.2, 2.3, 2.4

[RANGE]
RDK_VER =
```

The above configuration shows two types of tag that can be defined:

1. TAGS: these tags must list the possible values
2. RANGE: these tags don't list possible values be they are always numeric and >= 0. They must be listed to indicate they can be used inside XML configuration as range tags.

The following example shows how to them inside an XML configuration:

```
<CONTAINER SandboxName="TEST">

    <LxcConfig>
        <CGroupSettings>
            <DeviceCgroup>
                <DevicesDeny>a</DevicesDeny>
                <DevicesAllow SOC_VER="arm_16.4:arm_18.1" name="/dev/wake0">c 34:0
rw</DevicesAllow>
                <AllowDefaultDevices enable="yes"/>
            </DeviceCgroup>
        </CGroupSettings>
        <Environment>
            <Variable RDK_VER="1..3">LD_PRELOAD=/usr/lib/libdummy.so.1</Variable>
        </Environment>

        <Rootfs create="yes">
            <MountPoints>

<!-- /usr/bin -->
                <Entry version="debug" type="file">
                    <Source>/usr/bin/test</Source>
                    <Destination>usr/bin/test</Destination>
                    <Options>ro,bind,nosuid,nodev</Options>
                </Entry>

                <Entry version="release" type="file">
                    <Source>/usr/bin/test_release</Source>
                    <Destination>usr/bin/test</Destination>
                    <Options>ro,bind,nosuid,nodev</Options>
                </Entry>
```

```
<!-- /etc -->
                <Entry version="debug" RDK_VER="..3" type="file">
                    <Source>/etc/debug.ini</Source>
                    <Destination>etc/debug.ini</Destination>
                    <Options>ro,bind,noexec,nosuid,nodev</Options>
                </Entry>

            </MountPoints>

            <LibsRoBindMounts>
                <Entry RDK_VER="2">libIARMBus</Entry>
                <Entry OE_VER="2.1">libbusybox</Entry>
                <Entry SOC_VER="mipsel_16.4">libz</Entry>
            </LibsRoBindMounts>
        </Rootfs>
    </LxcConfig>
</CONTAINER>
```

Some important points:

- tags can be placed on these XML elements: Entry (both under MountPoints and LibsRoBindMounts), Variable, DevicesDeny, DevicesAllow and AllowDefaultDevices
- TAG type tags can indicate multiple tagvalues separated by ":"
- RANGE type tags can indicate ranges or single value. Examples: X or X..Y or ..Y or Z..
- multiple tags can be placed on one XML element: they must all match in order for the entry to show up inside generated lxc.conf file

Passing active tag values to generator tool

Possible tags are now configured inside config.ini and example container config XML has tags on several XML elements. The final step is to pass the current/active values for all these tags to the generator tool, so it can match the tags and enable or disable an XML element during generation of the lxc.conf file. Tag values can be passed with command line option "-t". For example: -t SOC_VER=arm_16.4,OE_VER=2.2,version=debug,RDK_VER=3

How to join another program to already running container

Introduction

This is solution for containers, which should have multiple process(binaries) running inside.
To join process into container the **lxc-attach** command should be use. lxc-attach runs the specified command inside the container specified by name. The container has to be running already.

How it works

The lxc-attach tool will try to allocate a pseudo terminal master/slave pair on the host and attach any standard file descriptors, which refer to a terminal to the slave side of the pseudo terminal before executing a shell or command. The lxc-attach tool was modified in the same way as lxc-execute to drop root privileges and spawn to given uid and gid.
Example of usage:

```
lxc-attach -n <container_name> -f <path_to_the_lxc_config_file> -u <UID> -g
<GID> -- <command>
```

When and how to use lxc-attach

lxc-attach should be used when we want to have several processes in one container. The processes can be run with different UID/GID: that means the resources such as files and directories in sandbox can be separated using file permissions.
Usage example:

For example I choose a DIBBLER container and execute there a simple "while true" bash command.

The DIBBLER container:

```
root      2801  0.0  0.0   2672   916 ?        Ss   12:33   0:00 /bin/sh
/container/DIBBLER/launcher/start.sh DIBBLER eth2
root      2810  0.0  0.0   2264   920 ?        S    12:33   0:00  \_
/usr/bin/lxc-execute -n DIBBLER -f /container/DIBBLER/conf/lxc.conf -- /bin/sh
-c /lib/rdk/dibbler_starter.sh eth2
netconf+  2831  0.0  0.0   1536   408 ?        Ss   12:33   0:00      \_
/init.lxc.static --gid 105 --uid 117 -- /bin/sh -c /lib/rdk/dibbler_starter.sh
eth2
      netconf+  2879  0.0  0.0  12276  1592 ?        Sl   12:33   0:00
      \_ /usr/sbin/dibbler-client run
```

Attaching the command using command:

```
      lxc-attach -n DIBBLER -f /container/DIBBLER/conf/lxc.conf  -u 122 -g 122
      -- /bin/sh -c "while true; do /bin/echo This is Test process; /bin/sleep
      5; done"
```

The UID 122 was used, which maps to firewall user

Following logs show how lxc-attach spawns to firewall user

```
root      4546  0.0  0.0   2364   980 ?        Ss   12:37   0:00
/usr/sbin/dropbear -i -r /etc/dropbear/dropbear_rsa_host_key -B
```

```
root      4559  0.0  0.0   2776  1388 pts/0    Ss   12:37   0:00  \_ -sh
root      5919  0.0  0.0   2716  1016 pts/0    S+   12:41   0:00      \_ lxc-
attach -n DIBBLER -f /container/DIBBLER/conf/lxc.conf -u 122 -g 122 -- /bin/sh
-c .....
firewall  5921  0.0  0.0   2224   924 pts/1    Ss+  12:41   0:00
\_ /bin/sh -c while true; do /bin/echo This is Test process; /bin/sleep 5;
done
      firewall  5923  0.0  0.0   2380   400 pts/1    S+   12:41   0:00
      \_ /bin/sleep 5
```

Logs attached below show part of output from systemctl:

```
root@dcx960-debug:/home/root# systemctl status
● dcx960-debug
    State: running
     Jobs: 0 queued
   Failed: 0 units
    Since: Thu 1970-01-01 01:00:45 CET; 47 years 0 months ago
   CGroup: /
           ├─1 /init
           ├─lxc
           │ ├─DIBBLER
           │ │ ├─2831 /init.lxc.static --gid 105 --uid 117 -- /bin/sh -c
/lib/rdk/dibbler_starter.sh eth2
           │ │ ├─2879 /usr/sbin/dibbler-client run
           │ │ ├─5921 /bin/sh -c while true; do /bin/echo This is Test
process; /bin/sleep 5; done
           │ │ └─6467 /bin/sleep 5
           │ └─UDHCPC
           │   ├─2815 /init.lxc.static --gid 106 --uid 117 -- /sbin/udhcpc -b
-i eth2 -s /etc/udhcpc.script
           │   └─3114 /sbin/udhcpc -b -i eth2 -s /etc/udhcpc.script
```

Systemd service types

The systemd monitors the process, which is started based on the PID. The problem starts when PID namespaces come into play. Systemd recognizes the notification from the processes based on the PID. Because our daemons are started in different PID namespace the PIDs do not match. As a result systemd cannot recognize the notification from the process and this breaks all dependencies. The systemd should monitor all process which it starts. This paragraph describe how to modify systemd services and how to setup XML configurations to keep this functionality, despite the PID namespace.

Type=simple

We can use it when process starts the container, or in other words, if it will be started using lxc-execute and not lxc-attach, then it can remain of type=simple. Stop statement in service file is: ExecStop=/container/SOMECONTAINER/launcher/some.sh stop

In case the service is started via lxc-attach, it should be converted to type notify via configuration:

- service file must be updated to type=notify
- use SystemdNotify create="yes" as above
- no PID file needed

The components which are started by lxc-execute can have Type=simple. The system will then monitors the lxc-execute and all its child processes. When the process in containers fails or exit with error. The system will notice that, cause also the lxc.init and lxc-execute processes will end up with error. Also the stop statement in service file can be used: ExecStop=/container/SOMECONTAINER/launcher/some.sh stop. The launcher script will stop the container using lxc-stop.
In case of process which use lxc-attach (joins already started container) the type simple cannot be used. The PID which will be registered in systemd will be PID of lxc attach. If lxc attach ends it's child process will be orphaned and the lxc.init will become the parent process for the lxc-attach child process. Also ther is no way to stop the process, which joins the container without destroying it. Such processes should be converted to type notify via configuration:

- service file must be updated to type=notify
- use SystemdNotify create="yes" as above
- no PID file needed

Type=notify

Use SystemdNotify create="yes". A pid file is needed because the standard sd_notify() call performed by the service will not work from inside the container. The service in question needs to write a PID file for which the generated launcher script will check and then trigger systemd-notify itself. If needed, the service must be patched to create such a PID file. The process name is optional, by default it takes the first part of ExecName. Sometimes ExecName consists of a complex statement. Then you can use the ProcessName tag to indicate which process lxccpid should look for. An ampersand is automatically added to the lxc-attach statement when SystemdNotify create="yes".

```
<SystemdNotify create="yes">
    <PidFile>/var/run/irmgr/irmgr.pid</PidFile>
    <ProcessName>/usr/bin/irMgrMain</ProcessName>
</SystemdNotify>
```

The generator tool generates a piece of script that uses lxccpid. Generated script will look like this:

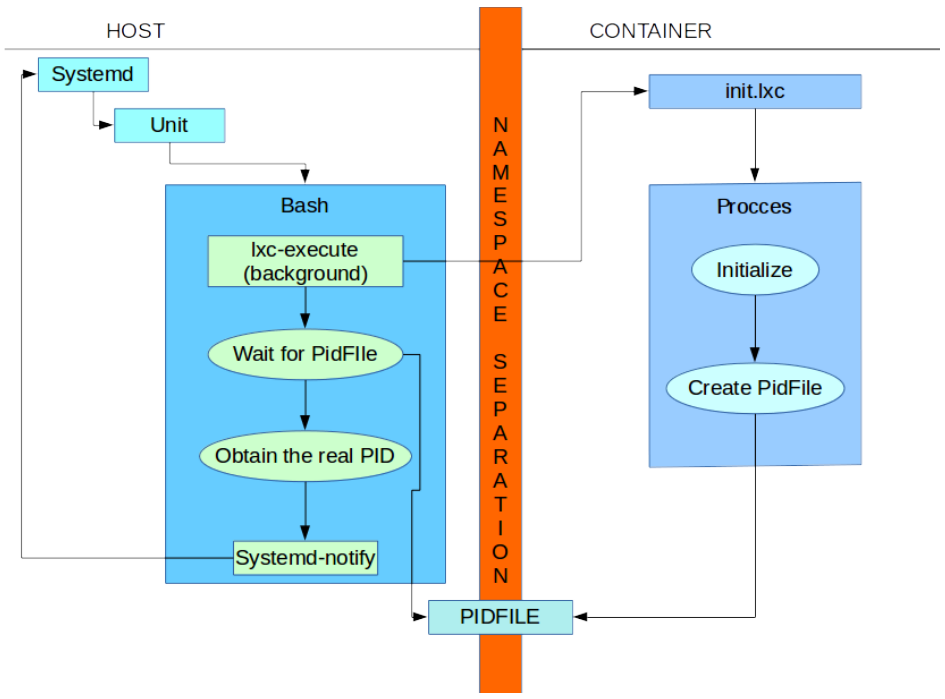
```
rm -rf "/var/run/irmgr/irmgr.pid"
/usr/bin/lxc-attach -n PLATFORMCONTROL -f
/container/PLATFORMCONTROL/conf/lxc.conf -u 205 -g 205 -- /usr/bin/irMgrMain
--debugconfig /etc/debug.ini &
while true
do
        if [ -e "/var/run/irmgr/irmgr.pid" ]; then
                break
        fi
        /bin/usleep 100000
done
/bin/systemd-notify --ready MAINPID=$(/usr/bin/lxccpid --ppid $!
/usr/bin/irMgrMain 2000)
# Wait for systemd to assign the proper pid, before the main process exit.
/bin/usleep 100000
```

Stop statement in service file is: ExecStop=/bin/kill -9 $MAINPID

Notify algorithm in nutshell

Systemd starts the unit, the Unit(service) starts the bash. The bash script starts the lxc-execute/lxc-attacg command with proper parameters in the background. After this the bash should wait until the process in the different namespace is initialized and ready (the notification for systemd should be changed to creating the pidfile).

The bash script will wait for this pidfile, then obtain real child process PID using lxccpid. In the last step bash will execute systemd-notify binary with proper parameters, which will allow to notify systemd and keep proper dependency order. An extra wait is added after the systemd-notify call so that systemd can properly process this request. If the bash script were to exit too fast, systemd would generate an error, breaking the notify process.



type=dbus

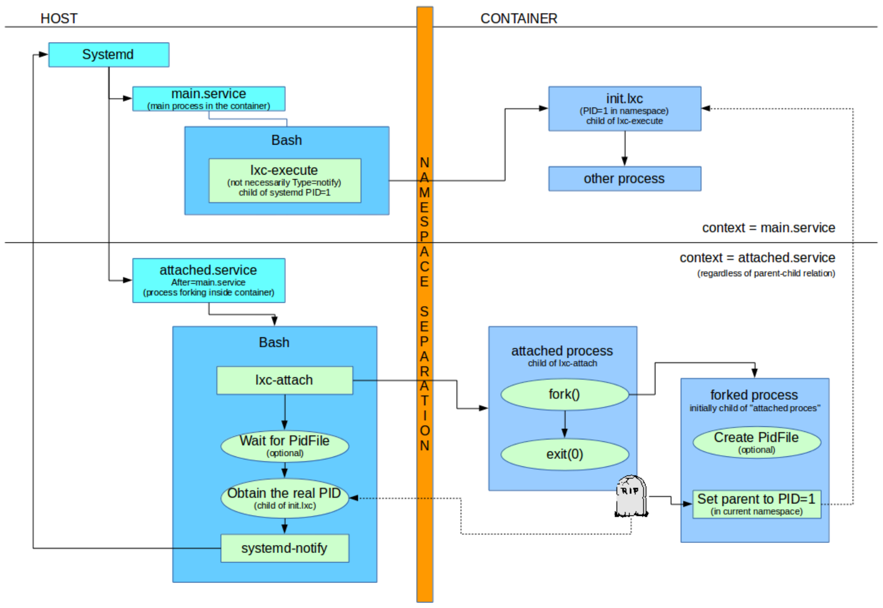
This service cannot be "converted" to a type=notify service because of the way it is started. So for these we always need a stop statement using lxccpid. For example the wpa_supplicant service: ExecStop=/bin/sh -c "/bin/kill -9 $(/usr/bin/lxccpid --ppid $MAINPID /usr/sbin/wpa_supplicant 1000)"

type=forking

If service user type forking it should be converted to Type=notify optionally with pid file.

Most processes can be converted to run in foreground with some command line arguments, e.g.: nginx -g "daemon off". This option is the preferred approach if possible and then Type=simple or Type=notify (above) applies.

If the main process truly forks and the main process exits, especially as typical Unix daemons do, and nothing can be done about it then you need to use <SystemdNotify forking="yes">. This enables extra logic related to finding the MAINPID. The process to be monitored by systemd will eventually NOT be a child of the lxc-attach which started it, but will become child of pid=1 inside namespace (because it's original parent exits and pid=1 always takes over such orphans). The picture below illustrates the situation.


HOST
CONTAINER
NAMESPACE SEPARATION
Systemd
main.service
(main process in the container)
Bash
lxc-execute
(not necessarily Type=notify)
child of systemd PID=1
init.lxc
(PID=1 in namespace)
child of lxc-execute
other process
context = main.service
context = attached.service
(regardless of parent-child relation)
attached.service
After=main.service
(process forking inside container)
Bash
lxc-attach
Wait for PidFile
(optional)
Obtain the real PID
(child of init.lxc)
systemd-notify
attached process
child of lxc-attach
fork()
exit(0)
RIP
forked process
initially child of "attached proces"
Create PidFile
(optional)
Set parent to PID=1
(in current namespace)

Mount binds in containers

Security requirements

1. All bind mounts shall be mounted with the “nosuid” mount option. (**Except for binaries that need linux capabilites**!)
2. If the content of a bind mount is not supposed to be modified via its mount point then the “ro” mount option shall be used.
3. If the bind mount is not supposed to expose device files via its mount point then the “nodev” mount option shall be used.
4. If no executables are supposed to be executed via a bind mount then the “noexec” mount option shall be used.
5. Proc should be mounted with option hidepid=2 (if the container processes are not supposed to be able to read PIDs from processes outside of container)

General rules

For all executables the nosiud and nodev flags should be set, **with the exception of executables that have linux capabilities set -then only nodev flag should be set.**

```
<!-- Regular exec files -->
<Entry type="file">
  <Source>/bin/sh.busybox</Source>
     <Destination>bin/sh</Destination>
     <Options>ro,bind,nosuid,nodev</Options>
</Entry>

<!-- exec files with linux capabilities set -->
<Entry type="file">
     <Source>/sbin/ifconfig.busybox</Source>
  <Destination>sbin/ifconfig</Destination>
     <Options>ro,bind,nodev</Options>
       </Entry>
```

The proc should be mounted (if needed) with noexec, nosuid, hidepid=2 and nodev option

```
<Entry type="dir">
  <Source>proc</Source>
  <Destination>proc</Destination>
  <FsType>proc</FsType>
  <Options>defaults,noexec,nosuid,nodev,hidepid=2</Options>
       </Entry>
```

The example of device mount

```
<Entry type="dev">
  <Source>tmpfs</Source>
  <Destination>/dev/shm</Destination>
     <FsType>tmpfs</FsType>
     <Options>defaults,noexec,nosuid</Options>
</Entry>
```

```
<Entry type="file">
  <Source>/dev/nexus</Source>
    <Destination>dev/nexus</Destination>
    <Options>bind,optional,create=file,nosuid,noexec</Options>
      </Entry>
```

By default all libs in <LibsRoBindMounts> entry are mounted with nosuid and nodev option.

Generating user and group information

Some of the users belong to more than one group, this means that the process also can have more than one group. To implement this all groups for given UID are read and then set before the process spawns in lxc.init. The user and groups are read from /etc/passwd and /etc/group in container rootfs. Because of security we do not want to mound bind or copy the host /etc/passwd and /etc/group content into container rootfs. To achieve that we parse /etc/passwd and /etc/group on host rootfs and generate /etc/passwd and /etc/group in container rootfs with minimum of information. Only the users of the container processes in passwd, which is mostly just one user. Only groups assigned to these users are put in /etc/group.

lxc-execute and lxc-attached were modified to drop root privileges. See separate page about LXC modifications.

Create unprivileged containers

To create unprivileged containers, which suit our needs, we modify the LXC mechanisms, more information can be found here. The Lxc-execute after modification passes uid and gid set in lxc.conf to init.lxc process. This process, after mount /proc and some devices, was modified to drop root privileges then fork and execvp given command. This mean that process is no longer root, neither on host nor in container. Because of that we need to add proper capabilities (effective and permitted) to binaries we want to run in container. If container process uses any kind of configs or scripts the proper access rights as well as user:group owner should be given to such files.

Currently creating unprivileged containers is disabled for some containers, the options InitUid InitGid will be ignored. This is a temporary setup to keep compatibility with booting rootfs from NFS. NFS does not support linux capabilities, so if we start a non-root process inside this container it will need linux capabilities. To enable unprivileged containers please move them to proper directory in OE recipe. This is done by adding them to the secured containers group inside container-generator-native_2.0.bb.

```
# SECURE CONTAINERS
        install_lxc_config secure lxc_conf_DBUS.xml

# NON SECURE CONTAINERS
        install_lxc_config non_secure lxc_conf_DIBBLER.xml
        install_lxc_config non_secure lxc_conf_DNSMASQ.xml
        install_lxc_config non_secure lxc_conf_IPTABLES.xml
        install_lxc_config non_secure lxc_conf_UDHCPC.xml
```

So in short: if they are added with the "non_secure" option, they are built as non-secure/privileged containers, unless *SECURE_CONTAINERS = "1"*. If they are added with "secure" option, they are always built as secure/unprivileged.

How to create own container

1. Go to ***meta-containers/recipes-containers/container-generator/files/xml*** and create file with your own XML container description: lxc_conf_<NAME_OF_THE_CONTAINER>.xml. You can use lxc_conf_EXAMPLE.xml as template.

2. Go to ***meta-containers/recipes-containers/container-generator/container-generator-native_2.0.bb*** and add a new SRC_URI entry for your XML file.

3. Rebuild the software using yocto command describe in Integration in Yocto Build System section above.