Splitting out-of-container rdkservices/thunderplugin .so for memory reduction and expected container security split

people involved

Adam Stolcenburg Pierre Wielders Bart Catrysse Piotr Marcinkowski

Summary and Scope

Up to now a rdkservice (aka thunderplugin) was typically coded into one single .so file.

While this is fine for rdkservices/plugins used in a "in-process" config this has important drawbacks for rdkservices/plugins used in "out-of-process/out-of-container" config when it comes to total memory consumption, accounting and security exposure of container running core thunder process(the `WPEFramework` process). examples of important rdkservices running out-of-container are OpenCDMi, WebKitBrowser plugins

This page solves this drawback by splitting the out-of-processs/container rdkservices/thunderplugin .so into .so part for WPEframework/thunder process and Impl.so part for rdkservice/thunderplugin process and describes convention on how to do this.

Terminology

Before starting with a description of the problem we are trying to solve here, first small paragraph about split between thunder core and thunder plugins into processes & containers & their terminology/names because this can be confusing and it is required to be on same page there before rest of page can be understood.

The Core thunder process is named `WPEFramework` process. That process is typically* running in a dedicated container, let's assume for sake of this explanation that name of that container is `WPEFRAMEWORK` container. When a thunder plugin is configured as in-process it is running within that `WPEFramework` process (and obviously within that same `WPEFRAMEWORK` container). Typically for robustness (eg cgroup memory limiting) and security design , several of the rdkservices/thunderplugin do need to run in separate containers. In that case, the thunder plugin is running in a separate process, normally with `WPEProcess` in its name and has to be configured to run out-of-process/out-of-container. The plugin code running as part of the `WPEFramework` process communicates with the code running as part of the `WPEProcess` process using *COM RPC*.

Problem description

Up to now a rdkservice (aka thunderplugin) was typically coded into one single .so file. Most of the *Thunder* plug-in libraries are linked with other libraries because they implement their functionality on the basis of functionality provided by other dynamic libraries. When the *Thunder* plugin library is loaded, these linked libraries are also loaded, as well as libraries needed by those libraries and so on, until the whole library dependency tree is loaded. For those plugins that need to run in a different container, it also means that same .so and its defined dependency tree is loaded by both the thunderplugin process inside its dedicated container (which you would expect) but also loaded by core thunder `WPEFramework` process inside `WPEFRAMEWORK` container. And that core thunder process typically only needs a portion of this library, the part for exposing JSON-RPC part and transforming to COM RPC and much smaller to zero dependency tree but since there is only one single .so for the plugin defined, it has no other choice then to load that same one - with the unnecessary big dependency tree - (linker reads .so and its definition requires access to full dependency in tree) or loading will fail. This also results that that same unnecessary big dependency tree needs to be made available in the `WPEFRAMEWORK` container. It exposes the `WPEFRAMEWORK` container to bigger attack surface and sometimes goes against the security design and very reasons why container was split off in first place. Here are some other disadvantages that come with with it :

- to load individual plugins, the `WPEFRAMEWORK` container must mount libraries on which they depend, this causes an excessive number of elements mounted inside this container;
- when a plugin is deactivated, the dynamic library that implements the plugin code is unloaded from the `WPEFramework` process context, but the libraries on which the library depends are not unloaded;
- due to the fact that the process of running an out-of-process plug-in by `WPEFramework` consists in first loading the plugin library together with its dependencies on the `WPEFramework` side and then spawning the `WPEProcess` that loads the same library and its dependencies in a separate container, the memory used to load these libraries mostly increases the amount of memory assigned to the `WPEFRAMEWORK` container and not the container in which `WPEProcess` is executed;
- the memory consumption is unnecessarily increased in the case of dynamic libraries that create *Anonymous/Private Dirty* areas.

Solution and procedure for splitting plugin code

Together with Metrological we agreed on a solution for the above problem which is to split the plugin code loaded on the side of WPEFramework and WPEProcess into two shared libraries.

This only applies to *Thunder* plugins that are running in the out-of-process mode

The procedure and convention for such a split are :

In the first step it should be determined what files contain code that is needed by the `WPEFramework` and what code is needed by the `WPEProcess`. The `WPEFramework` will need files that contain *JSON RPC* implementation as well as code that is needed to initialize and deinitialize the plugin, the code can be recognized by the presence of the `SERVICE_REGISTRATION()` macro. Additionally, in most cases, the code that is placed in the `Module.cpp` file will be needed by both `WPEFramework` and and `WPEProcess`. The rest of the code is usually needed only by the `WPEProcess`.

Once it is known what files are needed by each of those processes, the `CMakeLists.txt` file of the *Thunder* plugin should be modified in such a way that files needed by the `WPEFramework` are compiled into a dynamic library whose name matches the following pattern:
`libWPEFramework[Plugin Name].so`
for example:
`libWPEFrameworkWebKitBrowser.so`

Files needed by the `WPEProcess` need to be compiled into a dynamic library whose name matches the following pattern:
`libWPEFramework[Plugin Name]Impl.so`
for example:
`libWPEFrameworkWebKitBrowserImpl.so`

The important thing is that the dynamic library loaded by the `WPEFramework` process may not link with dynamic libraries that are not provided by the *Thunder* framework.

The last step of the splitting process is to set the `locator` property in the `root` object of the `configuration` object in the plugin's *JSON* file, to a the name of the library that needs to be loaded by the `WPEProcess`. Please note that the `locator` property of the main object of the plugin's *JSON* file must still point to the library that is needed by the `WPEFramework` process. Plugin's *JSON* file matches the name of the plugin with `.json` extension and is placed in the `/etc/WPEFramework/plugins` directory on the rootfs. An exemplary plugin's *JSON* file with the modification applied is presented here:

```
{
  "locator":"libWPEFrameworkWebKitBrowser.so", (…)
  "configuration":{ (…)
    "root":{
      "locator":"libWPEFrameworkWebKitBrowserImpl.so", (…)
    } (…)
  } (…)
}
```

When the thunder-plugin is run in a container, there is also a need to make the `Impl` dynamic library visible in the container instead of the original library. For example, in the case of the `WebKitBrowser` plugin, the `WPEBrowser` container should mount the `usr/lib/wpeframework/plugins/libWPEFrameworkWebKitBrowserImpl.so` library instead of the `usr/lib/wpeframework/plugins/libWPEFrameworkWebKitBrowser.so` library.

Concrete examples of split

- .so split of WebkKitBrowser plugin into `libWPEFrameworkWebKitBrowser.so` and `libWPEFrameworkWebKitBrowserImpl.so` proposed by LG and upstreamed by Metrological, see **RDKDEV-253** - Getting issue details... STATUS
- .so split of OCDMi plugin, in process of being upstreamed by Liberty Global and reviewed by Comcast, see https://github.com/rdkcentral/rdkservices/pull/2497
    - Thanks to this we avoid mounting in the `WPEFRAMEWORK` the following libraries:
        - `libocdm.so.1`
            - `libgstreamer-1.0.so.0`
                - `libffi.so.6`
                - `libgmodule-2.0.so.0`
                - `libpcre.so.1`
                - `librdkloggers.so.0`
                - `liblog4c.so.3`
                - `libsystemd.so.0`
                - `libcap.so.2`
                - `libresolv.so.2`
                - `liblzma.so.5`
                - `librt.so.1`
            - `libgobject-2.0.so.0`
            - `libglib-2.0.so.0`
            - `libgstbase-1.0.so.0`
            - `libnexus.so`
            - `libbrcmsvpmeta.so`
                - `libgstaudio-1.0.so.0`
                - `libgsttag-1.0.so.0`
                - `libz.so.1`
            - `libb_os.so`

Notes

(*) whether or not the core thunder process runs in dedicated container can depend on operator. For Liberty Global it required to be in separate container for various reasons, one being the security design, core thunder process exposes api's to 3rd party apps via network (ws) and need to restrict attack surface/thread exposure.