DesignDocument

Design EnforcerNG

:Authors: Yuri
:Version: 1.0
:Date: 20140401

This document serves to describe the design of the EnforcerNG in the current state on a high level. Please add/update/correct as much as possible. It should capture motivations for major design decisions and is intended for peers.

The enforcer is a daemon like the signer: It has an internal scheduler to execute tasks at fixed times and interact with the user via a client utility. The intention is to share code between the signer and the enforcer. While a lot of code is copied over no code sharing is done as of today.

Candidates for code sharing:
- logging
 - commandline client
- command handling
- scheduler
 - XML handling
- common functions str/file/lock etc

EnforcerNG consists of 2 main programs, the daemon 'ods-enforcerd' and the commandline client 'ods-enforcer'. With the exception of starting the daemon, the client does not interact with the system directly but rather passes everything to the daemon. Communication between the programs is done through a Unix domain socket configured in conf.xml.

Commandline client protocol

(note: It time of writing the code for this is ready but not merged. 20140401)

Up until now the ods-enforcer was not much more than the netcat utility, just passing strings over. The motivation for its existence was:
- must be able to start the daemon
- must be able to tell daemon is running
- must prompt for confirmation on destructive commands.

The problems with this are:
- no way of returning exit codes
- could not differentiate between stderr and stdout
- not truly interactive

To solve this a protocol is introduced, simple enough to justify not depending on an external library for it. A message looks like this::

```
+-----+-----+-----+-----+
| OPC | len | msg |
+-----+-----+-----+-----+
```

OPC: 1 octet, type of message. (see ``daemon/clientpipe.h``)
len: 2 octets, big-endian, length of entire message.
msg: 0-65535 octets, data, usually a string.

Types of messages currently include: (print to) stdout, (print to) stderr, prompt user, input, done (exit code). In the future there may be control messages as well. (I'm thinking of attaching to the daemon and keep logging or similar)
In non-interactive mode the client will connect to the Unix domain socket, send command, keep receiving messages until exit message is captured. The payload of that message will be the return code of the client.
In interactive mode the client connects to the Unix domain socket and then blocks on reading stdin. When data available it is send to the daemon, the clients will then keep on reading messages till exit message is encountered. This time the exit code is printed for the user and the clients returns to reading stdin again. I would have liked to read stdin simultaneously with the socket using select to make it full duplex. However I also liked to use the readline library to have commandline history. These two don't play well. (it can be done but you'll have the worst of two worlds)

Command Handler

In the daemon every user executable task is separated in two: a command part and a task part. I think the original idea is that tasks can be scheduled by an automatic process or executed on a user command. The command part runs directly without being scheduled on the user's initiative. In practice these task are almost never scheduled and the seperation is somewhat artifical. To get cleaner more consistent code I propose to either:

A. Drop (most of) the foo_task.cpp files and merge them with foo_cmd.cpp. This will reduce code overhead an file count a lot.
B. Schedule everything, also user initiated processes. This will make code simpler and more consistent.

Every cmd file exports just one function::
struct cmd_func_block* foo_funcblock(void);

The struct defines the base command, help and usage functions, run and handles functions (see ``daemon/cmdhandler.h`` for details). When a user command is given the daemon will iterate all modules' *handles* functions. The first module claiming responsibility get their run function called with the same arguments. The run function then does some processing (usually involves calling a task) and returns an exit code which is then passed back to the client.