Design Document:

Scalability and Security Enhancements for PC/SC-Lite

Paul Klissner, June 2008

Design Goals

1. Port existing Open Source PC/SC-Lite 1.3.2 to be compatible with Solaris, Solaris + Trusted Extensions and to support the Sun Ray thin client platform.

2. Do it in a way that can be integrated back into main Open Source project as well as acceptable to two Sun review bodies.

Disclaimers

At the time of the intial integration back into the PC/SC-Lite open source gate there is quite a bit of work involved if this code is to be merged into the trunk.

The Open Source code has had many changes since this project diverged at 1.3.2. Further there is additional clean up to be done to make this code platform neutral.

While an effort has been made to preserve backward compatibility with the configuration modes, and many areas were designed with platform neutrality in mind, due to time-constraints on the original project areas had to be implemented for Solaris and the platform-specific idioms and mechanisms were not put in place for every function. This willl require some work.

Because this was designed to build in our internal environment originally the workspace has been set up to build PC/SC-Lite with a specific environment and configuration (see the buildenv directory and enclosed README). Ultimately this will need to be resolved so it builds everything properly under the various supported configurations on all of the supported platforms. Until then, the code should be considered to be Solaris specific, and only tested to work with the pre-determined set of flags at configure time, as provided in the setup script.

Open Source PCSClite 1.3.2
(starting point for adaptation to Cyclops)

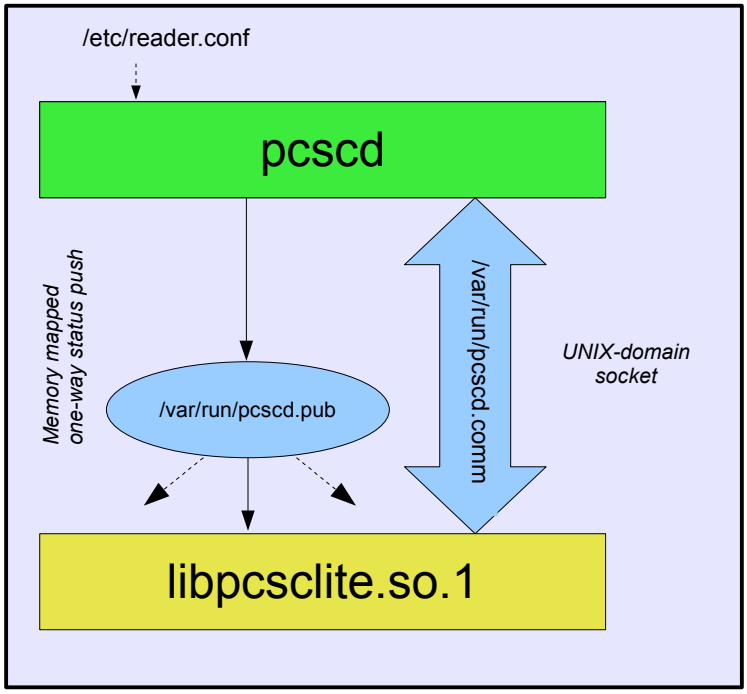

Diagram 1

conf igur e scr ipt op tion al fe atures [pr e-bui ld con figu ratio n]:

```
  --disable-FEATURE             do not include FEATURE
  --enable-FEATURE[=ARG]        include FEATURE [ARG=yes]
  --disable-dependency-tracking speeds up one-time build
  --enable-dependency-tracking  do not reject slow dependency extractors
  --enable-maintainer-mode      enable make rules and dependencies
  --enable-shared[=PKGS]        build shared libraries [default=yes]
  --enable-static[=PKGS]        build static libraries [default=yes]
  --enable-fast-install[=PKGS]  optimize for fast install [default=yes]
  --disable-libtool-lock        avoid locking  --disable-libusb
  --enable-usbdropdir=DIR       directory containing USB drivers (default
                                /usr/local/pcsc/drivers)
  --enable-debugatr             enable ATR debug messages from pcscd
  --enable-scf                  use SCF for reader support
  --enable-confdir=DIR          dir containing pcsc.conf (default /etc)
  --enable-runpid=FILE          file containing pcscd pid
  --enable-ipcdir=DIR           dir containing IPC files (default /var/run)
```

Com mand line optio ns:

```
  -a, --apdu           log APDUs and SW using the debug method (see -d)
  -c, --config file    Specifies alternate location for reader.conf
  -d, --debug OUTPUT   display debug messages.
  -f, --foreground     Runs pcscd in the foreground (no daemon)
  -h, --help           Displays information about the pcscd command line
  -v, --version        Displays the program version number
```

PC/SC-Lite Scalability & Security Enhancements

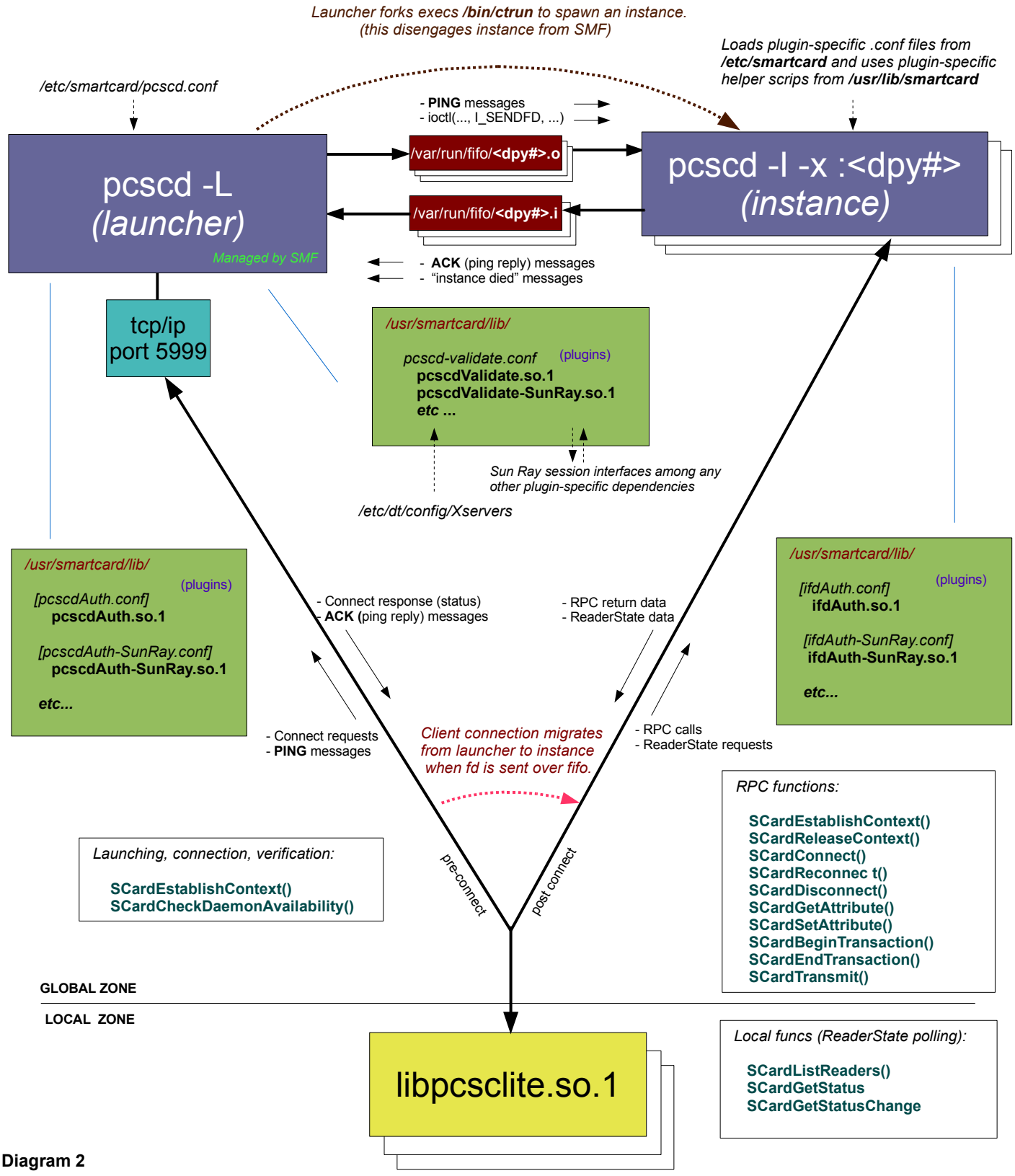

Diagram 2

Pre-build autoconf/configure options (new with Cyclops PCSClite):

```
  --enable-rtlib=DIR             daemon-specific runtime path for library path

  --enable-xtag=<Xservers PATH>  specifies path to system's Xservers file. Needed
                                 only if configuring for portsvc, ie. launcher mode

  --enable-portsvc=PORT          specifies port client library can contact pcscd on
                                 if a server instance is running in 'port server'
                                 mode. (default=port server mode not available)
```

Command-line options for Cyclops PCSClite

[NOTE: Usage page auto-generated from command option tables & KVP parser tables]

```
Usage: pcscd options

   pcscd -?
   pcscd -v

   pcscd -k  -x :display_number  [ -P pid ]

   pcscd -L  [ -c config_file_path ]  [ -b basedir ]
             [ -m mapfile_path ]  [ -o logfile ]  [ -A ]
             [ -p port_number ]  [ -t {SOCKET_UNIX | SOCKET_INETV4} ]
             [ -f ]  [ -M ]  [ -l {STDERR | SYSLOG} ]
             [ -d {DEBUG | INFO | ERROR | CRITICAL} ]  [ -a ]
             [ -N time_secs ]  [ -R time_secs ]  [ -F timeout_secs ] [ -V ]

   pcscd -I  [ -x :display_number ]
             [ -c config_file_path ]  [ -b basedir ]  [ -m mapfile_path ]
             [ -i ifd_plugin_dir ]  [ -r reader_config_path ]
             [ -o logfile ]  [ -A ]  [ -p port_number ]
             [ -t {SOCKET_UNIX | SOCKET_INETV4} ]  [ -f ]  [ -M ]
             [ -l {STDERR | SYSLOG} ]  [ -d {DEBUG | INFO | ERROR
                ... | CRITICAL} ]  [ -a ]  [ -V ]

   pcscd     [ -c config_file_path ]  [ -b basedir ]
             [ -m mapfile_path ]  [ -i ifd_plugin_dir ]
             [ -r reader_config_path ]  [ -o logfile ]  [ -A ]
             [ -p port_number ]  [ -t {SOCKET_UNIX | SOCKET_INETV4} ]
             [ -f ]  [ -M ]  [ -l {STDERR | SYSLOG} ]
             [ -d {DEBUG | INFO | ERROR | CRITICAL} ]  [ -a ] [ -V ]

   Options:

       -L   --launcher         Run in launcher mode
       -I   --instance         Run as instance (mode is used by launcher)
       -k   --stop             Terminate instance handling specified display
       -P   --pid              PID of instance to stop
       -c   --config           Specify hierarchical server config file location
       -b   --basedir          Specify hierarchical server base dir location
       -r   --reader           Specify abs. or relative reader conf. location
       -f   --foreground       Run in foreground (no daemon)
       -i   --ifd              Specify abs. or rel. IFD handler plugin path
       -x   --display          X display that owns reader(s) of interest
       -t   --transport        Specify IPC comm. transport type
       -p   --port             Specify INETV4 port number to use
       -T   --timeout          Specify Instance timeout
       -A   --useauth          Enable authentication
       -m   --mapfile          Specify memory map file name
       -M   --usemap           Enable memory-mapped reader state conveyance
       -d   --loglevel         Set logging minimum severity level
       -l   --logtype          Specify facility to send logging output to
       -o   --logfile          Specify target of stderr
       -a   --apdu             Log APDU commands and results
       -V   --verbose          Debug verbosity level, 0=min
       -N   --launchthr        Launched instance min time req. to assume success
       -R   --launchint        Instance, max. allowed failed launches + retries
       -F   --fifotime         Number of seconds to time out on fifo ping
       -v   --version          Display the program version number
       -?   --help             Display usage information
```

Client/Server/Instance Configuration File Parsing (Key-Value Pairs w/wildcard support)

```
#define KVP(key, consumer, visibility, type, result) \
          { #key, consumer, visibility, type, \
            (struct kvpValidation *)&key, (void *)result }


static struct kvp {
        char *key;                          /* key name of this KVP */
        int  consumer;                      /* Who can access, client, server or both? */
        int  visibility;                    /* Is this a user or internal-only option? */
        int  type;                          /* What is the resultant data type */
        struct kvpValidation *validation; /* Optional validation processing for val */
        void *result;                       /* Where the parsed result is stored */
} kvps[] = {
        KVP(READER_CONFIG_FILE,     SERVER, PARAM,  _STRING,   &pcscCfg.readerConfigFile),
        KVP(IFD_PLUGIN_PATH,        SERVER, PARAM,  _STRING,   &pcscCfg.ifdPluginDir),
        KVP(APDU_DEBUG,             SERVER, PARAM,  _BOOLEAN,  &pcscCfg.apduDebug),
        KVP(PCSCD_PID_FILE,         SERVER, PARAM,  _STRING,   &pcscCfg.pcscdPIDFile),
        KVP(RUN_IN_FOREGROUND,      SERVER, PARAM,  _BOOLEAN,  &pcscCfg.runInForeground),
        KVP(INSTANCE_TIMEOUT,       SERVER, PARAM,  _NUMERIC,  &pcscCfg.instanceTimeout),
        KVP(PCSCD_CONFIG_FILE,      SERVER, INTERN, _STRING,   &pcscCfg.pcscConfigFile),
        KVP(HELPER_SCRIPT,          SERVER, INTERN, _STRING,   &pcscCfg.instanceScript),
        KVP(STATUS_POLL_RATE,       SERVER, PARAM,  _NUMERIC,  &pcscCfg.statusPollRate),
        KVP(USE_AUTHENTICATION,     SERVER, PARAM,  _BOOLEAN,  &pcscCfg.useAuthentication),
        KVP(LOG_LEVEL,              SERVER, PARAM,  _CONSTANT, &pcscCfg.logLevel),
        KVP(LOG_TYPE,               SERVER, PARAM,  _CONSTANT, &pcscCfg.logType),
        KVP(FIFO_PING_TIMEOUT,      SERVER, PARAM,  _NUMERIC,  &pcscCfg.fifoPingTimeout),
        KVP(RELAUNCH_THRESHOLD,     SERVER, PARAM,  _NUMERIC,  &pcscCfg.relaunchThreshold),
        KVP(RELAUNCH_INTERVAL,      SERVER, PARAM,  _NUMERIC,  &pcscCfg.relaunchInterval),
        KVP(LOG_FILE,               MUTUAL, PARAM,  _STRING,   &pcscCfg.logFile),
        KVP(TRANSPORT,              MUTUAL, PARAM,  _CONSTANT, &pcscCfg.transportType),
        KVP(BASE_DIR_WILD,          MUTUAL, INTERN, _BOOLEAN,  &pcscCfg.baseDirWild),
        KVP(USE_MAPPED_MEMORY,      MUTUAL, PARAM,  _BOOLEAN,  &pcscCfg.useMappedMemory),
        KVP(PORT_NUMBER,            MUTUAL, PARAM,  _NUMERIC,  &pcscCfg.portNbr),
        KVP(PORT_NUMBER_WILD,       MUTUAL, INTERN, _BOOLEAN,  &pcscCfg.portNbrWild),
        KVP(MEMORY_MAPPED_FILE,     MUTUAL, PARAM,  _STRING,   &pcscCfg.pcscdMemMappedFile),
        KVP(X_HOST_IP,              MUTUAL, PARAM,  _IPADDR,   &pcscCfg.xHostIp),
        KVP(DISPLAY_NUMBER,         MUTUAL, PARAM,  _NUMERIC,  &pcscCfg.dpyNbr),
        KVP(SCREEN_NUMBER,          MUTUAL, PARAM,  _NUMERIC,  &pcscCfg.screenNbr),
        KVP(NET_BIND_FILE,          MUTUAL, INTERN, _STRING,   &pcscCfg.netBindFile),
        KVP(CONSUMER,               MUTUAL, INTERN, _CONSTANT, &pcscCfg.consumer),
        KVP(BASE_DIR,               MUTUAL, PARAM,  _STRING,   &pcscCfg.baseDir),
        KVP(LAUNCH_MODE,            MUTUAL, PARAM,  _CONSTANT, &pcscCfg.launchMode),
        KVP(VERBOSE,                MUTUAL, PARAM,  _NUMERIC,  &pcscCfg.verbose),
```

The parser recognizes all of the key-value pairs defined via macros above. All but “INTERN” KVPs are visible and recognized in one or more configuration files. There is a one-to-one correspondence with KVPs that can be specified in a configuration file, and which can be specified on the command line.

This configuration mechanism is used by the launcher, instance and client code. (“MUTUAL” defines a KVP that is recognized in both pcscd and client library namespaces). Using a common parsing mechanisms and configuration block guarantees consistent interpretation of the options between components. The mechanism is very extensible, portable and adaptable.

Key-Value Pair Grammar

```
#define CONST(C)               { #C, PARSE_CONST,        0,    0,    0,  0, 0, 0, C, 0 }
#define RANGE(low, high)       { "", PARSE_RANGE,        0,  low, high,  0, 0, 0, 0, 0 }
#define INTWILD(typ, tok, trg) { "", PARSE_WILD,  _NUMERIC,  typ, tok, trg, 0, 0, 0, 0 }
#define STRWILD(typ, tok, trg) { "", PARSE_WILD,   _STRING,  typ, tok, trg, 0, 0, 0, 0 }
#define NUMBER(flags)          { "", PARSE_NUMBER,   flags,    0,   0,   0, 0, 0, 0, 0 }
#define QUOTED(flags)          { "", PARSE_QUOTED,   flags,    0,   0,   0, 0, 0, 0, 0 }
#define PATH(flags)            { "", PARSE_PATH,     flags,    0,   0,   0, 0, 0, 0, 0 }
#define EOL                    { "", 0, 0, 0, 0, 0, 0, 0, 0, 1 }

static struct kvpValidation {
        char *key;     /* Key name of this element */
        int process;   /* Additional validation processing */
        int option;    /* Processing sub-category */
        void *arg1;    /* Option-dependent input param #1 */
        void *arg2;    /* Option-dependent input param #2 */
        void *arg3;    /* Option-dependent input param #3 */
        void *datum1;  /* Pre-process output parameter 1 */
        void *datum2;  /* Pre-process output parameter 2 */
        int constVal;  /* Value of constant if this defiens one */
        int eolFlag;   /* Set for last elem in list (if set ignore other fields) */

}  TRANSPORT[] = {
        CONST(SOCKET_UNIX),
        CONST(SOCKET_INETV4),
        EOL

}, LAUNCH_MODE[] = {
        CONST(LAUNCHER),
        CONST(INSTANCE),
        CONST(DEFAULT),
        EOL

}, LOG_LEVEL[] = {
        CONST(DEBUG),
        CONST(INFO),
        CONST(ERROR),
        CONST(CRITICAL),
        EOL

}, LOG_TYPE[] = {
        CONST(STDERR),
        CONST(SYSLOG),
        EOL

}, PORT_NUMBER[] = {
        RANGE("0", "65535"),    /* Enforce range limit on port number value */
        INTWILD(X_DISPLAY_NUMBER, "$DISPLAY", &pcscCfg.portNbrWild),
        EOL

}, BASE_DIR[] = {
        PATH(PATH_OPTIONAL),
        STRWILD(X_DISPLAY_NUMBER, "$DISPLAY", &pcscCfg.baseDirWild),
        EOL


}, IFD_PLUGIN_PATH[] = {
        PATH(PATH_OPTIONAL),    /* Can't validate presence during initial parsing  */
        EOL

}, READER_CONFIG_FILE[] = {
        PATH(PATH_OPTIONAL),    /* Can't validate presence during initial parsing */
        EOL

}, MEMORY_MAPPED_FILE[] = {
        PATH(PATH_OPTIONAL),    /* Can't validate presence during initial parsing */
        EOL

}, CONSUMER[] = {
        CONST(LAUNCHER),
        CONST(INSTANCE),
        CONST(DEFAULT),
        CONST(CLIENT),
        EOL

}, BASE_DIR_WILD[]          = { EOL },
   PORT_NUMBER_WILD[]       = { EOL },
   LOG_FILE[]               = { EOL },
   RUN_IN_FOREGROUND[]      = { EOL },
   VERBOSE[]                = { EOL },
   PCSCD_CONFIG_FILE[]      = { EOL },
   SCREEN_NUMBER[]          = { EOL },
   DISPLAY_NUMBER[]         = { EOL },
   PCSCD_PID_FILE[]         = { EOL },
   NET_BIND_FILE[]          = { EOL },
   USE_MAPPED_MEMORY[]      = { EOL },
   X_HOST_IP[]              = { EOL },
   USE_AUTHENTICATION[]     = { EOL },
   INSTANCE_TIMEOUT[]       = { EOL },
   HELPER_SCRIPT[]          = { EOL },
   STATUS_POLL_RATE[]       = { EOL },
   APDU_DEBUG[]             = { EOL },
   RELAUNCH_THRESHOLD[]     = { EOL },
   FIFO_PING_TIMEOUT[]      = { EOL },
   RELAUNCH_INTERVAL[]      = { EOL }
```

New in this branch:

- Launcher / Instance mode, mutually re-startable and mutually re-interconnectable
- Platform specific display validation plugin provider scheme, and daemon access authentication plugin schme, including Sun Ray validation and authentication plugins as well as generic validation and authentication plugins.
- IFD handler auth plugin provider scheme, and plugins.
- TCP/IP socket client/server interface, as alternative to UNIX Domain socket interface.
- Reader state polling via socket rather than memory map (to ease TX integration)
- Command line parser / processing engine
- Integration into SMF framework
- Configuration modes specified via config files, KVPs.
- Improved debug logging (time stamps, server-side RPC call identification). Can be specified for both launcher, and per-instance via absolute or instance specific relative paths
- High configurability to support backward compatibility.

Files used by SolarisPCSC package

Binaries	
/usr/lib/smartcard/pcscd	Daemon
/usr/lib/libpcsclite.so.1	Library
Scripts	
/usr/lib/smartcard/pcscd-Local	Platform specific instance launch preparatory script.
SMF Integration:	
/var/svc/manifest/application/security/pcscd.xml	SMF service description
/lib/svc/method/pcscd-svc	SMF service control script
Configuration	
/etc/smartcard/pcscd.conf	Launcher / stand alone mode daemon configuration
/etc/smartcard/pcscd-Local.conf	Display's facility-specific conf file (for console in this case)
/etc/smartcard/reader-Local.conf	Display's facility-specific script (for console in this case)
/etc/reader.conf	*Static IFD handler config (optional, backward compat.)*
~/pcscd.conf	Optional client side configuration file
Dynamic State	
/var/run/pcscd/fifo/<dpy#>.i	Fifo to read messages from instance
/var/run/pcscd/fifo/<dpy#>.o	Fifo to write messages and send fd's to instance
/var/run/pcscd/pid/<dpy#>	PID of instance handle display
/var/run/pcscd.pid	*PID of instance handle display (opt/backward compat.)*
/var/run/pcscd.comm	*UNIX Domain client/daemon socket (opt/backward compat.)*
/var/run/pcscd.pub	*Memory mapped push/poll file (opt/backward compat).*
Validation Plugins	
/usr/smartcard/lib/pcscd-validate.conf	Display validation plugins configuration
/usr/smartcard/lib/pcscdValidate.so.1	Generic platform validation plugin
Authentication Plugins	
/usr/smartcard/lib/pcscdAuth-Local.so.1	
/usr/smartcard/lib/ifdAuth.so.1	*Not provided in this version.*

Files used by original PC/SC-Lite 1.3.2

File	**Description**
Binaries	
/usr/lib/smartcard/pcscd	Daemon
/usr/lib/libpcsclite.so.1	Library
Configuration	
/etc/smartcard/pcscd.conf	Launcher / stand alone mode daemon configuration
/etc/reader.conf	Static IFD handler configuration
Dynamic State	
/var/run/pcscd.pid	PID of instance handle display
/var/run/pcscd.comm	UNIX Domain client/daemon communcation socket
/var/run/pcscd.pub	Memory mapped push/poll reader state file

Configuration, Config files & Command line option processing

Overview

PC/SC-Lite 1.3.2 defined file paths and modal options at pre-compile time via autoconf. These default and configurable modes needed to be preserved, as well making the paths runtime configurable via config files and command line options. The approach taken to maintaining backward compatibility was to write parsing and option processing code in a modular way highly re-usable for other projects.

Configuration limitations vs. new configuration modes and options

1. Original code provides for single instance, multiple clients, but isn't scalable to the degree necessary for a thin client network.

   The approach we decided on was a multiple-instance mode in order to scale up for a thin client platfprm, while avoiding a singleton daemon as a single point of failure for deployment wide Smart Card use.

2. Original code uses a single UNIX domain socket between the client and a single instance and also uses a server-push, client-poll memory-mapped file to convey reader state and data.

   To facilitate Trusted Extensions and simplify administration, we needed a mode that eliminated shared configuratio files and shared filesystem state between local and global zones (ie. between libpcsclite.so.1 and pcscd) This was accomplished by using a single Internet Domain socket with a well-known port. Our initial design was one port per-display, but that was re-written in order to reduce a wide swath of less-protected ports under TX).

3. We also needed a way to validate smart card hardware, and to authenticate users for access to the daemon or to reader hardware. So two plugin frameworks were added to facilitate validation and authentication.

4. The original code uses command-line start-up mode, thes launching with inetd is implied. For for Solaris 10 the model for such infrastructures is to use Solaris' Service Management Framework (SMF) to control the startup, termination, and automatic re-starting of the pcscd service.

Design Specifics

In order to support pre-existing config options and file paths, while introducing the new features and flexibility, a set of configuration functions was added (***cfgfuncs.c***). A new tabular / MACRO configured parser was written to consumes key-value pairs and parse results into a named global configuration data area (***pcsc_config.h, cfgfuncs.c***).

Every command-line option has a KVP equivalent. Options are parsed in terms of their equivalent KVPs. The KVP processing engine provides a state machine for handling wildcard parameters, and is configured via table-defined grammars, which provide for easy definition of typed parameters: ***const, range, wildcard string, wildcard number, quoted literals and file path***. Each type is associated with a correspondng validation method.

Formatted usage output is auto-generated in response to -? flag, based on command line option definition MACRO tables. This usage() function elicits suboption lists by examining KVP grammar tables (***pcscdaemon.h, pcsc_config.h***)

Path management functions (***cfgfuncs.c***) provide a flexible means for defaulting to old statically defined file locations, as well as allowing original defaults for file locations, while creating a means for easily switching to new runtime path configuration, based on command line options and/or KVP values found in configuration files.

All of this provides for a way of providing a consistent interpretation of parameters among the launcher, instance and client, as well as creating a mechanism for consistent extensibility and ease of maintainability. This model was designed to make it easier to stay in sync with the opensource project as it evolved.

Daemon Communication and Control

Introduction

The configuration distributed in the ***SolarisPCSC*** packages makes PC/SC-Lite a service of the Solaris Service Management Framework (SMF). When the package is installed, the pcscd launcher is automatically run, and will be automatically re-started by SMF if it is terminated abnormally (ie. w/o the ***svcadm disable*** command being run). The launcher is responsible for launching, managing and coordinating new connections to pcscd instances.

Daemon Initialization

The pcscd daemon is brought up in *launcher* mode when the **-L** flag is specified on the command line, and in the complimentary *instance* mode when the **-I** flag is specified. If neither mode is specified, pcscd is brought up in standalone mode (ie. backward compatibilty). Some options are parsed from the command line first, but most command line options are parsed after the appropriate config file is loaded. This is done so that the config file options can be overridden.

The launcher loads ***/etc/smartcard/pcscd.conf.*** Instances load their context configuration from .conf files similary, but a 'tag' suffix is added (this tag is determined by a looking up the X Display # in the Xservers file, which identifies the facility that owns the particular display with the tag). For example, for a Sun Ray thin client display, the Xserver files will specify SunRay as the display owner facility, so\ the corresponding conf file that will be loaded by pcscd when launching a pcscd instance for that display ***is /etc/smartcard/pcscd-SunRay.conf***.

Launcher Behavior

The launcher starts up with a single thread running that opens a socket on well-known port 5999, listen for connections, and accepts incoming connections, firing up a new thread to process each incoming connection.

Connection Thread Behavior

1. Replies to “PING nnn” messages with “ACK nnn”, immediately closes connection and exits.

2. Otherwise message is parsed as a $DISPLAY variable.

3. If the $DISPLAY value is syntactically invalid, the connection is aborted with an error.

4. If the ***USE_AUTHENTICATION*** key is positive in ***pcscd.conf,*** the following authentication sequence is used. (Note the same sequence is also used in launcherless stand alone mode, if authentication is enabled).

   i. The parsed $DISPLAY number is sent to each of the validation plugins in the order specified in ***pcscd-validate.conf*** until a plugin recognizes the display as it's own and returns it's platform specific tag. If no plugin recognizes the display, the client connection is terminated as in error. If the generic plugin recognizes the display, the empty string tag “” is returned, which is valid. If the flag is set that the display has a new provider, the instance is stopped, which will force the launch of a replacement instance (on Sun Ray, a change of session ID causes this flag to be set).

   ii. The plugin is called with the $DISPLAY number to get the opaque (void *) display-specific resource. The type and meaning of that resource are specific to the facility and managed accordingly by the facility-specific plugin.

   ii. The credentials of the client connection, *uid, gid, pid*, *IP addr*, are passed along with $DISPLAY number and the *display-specific resource* to the platform-specific authentication plugin that is selected by the tag returned during the display validation phase. The authentication test is either pass/fail. If the authentication fails, the connection is terminated as in error.

5. If none of the tests have failed at this point, the $DISPLAY number and client socket fd is queued to an instance management thread for the display, if an instance management thread is running. If no instance management thread is found, one is started.

6. The connection *thread* exits, leaving the client connection open for further processing.

Daemon Communication and Control (part II)

Instance Management Thread:

```
for(;;) {
        while (ConnectInstance(inst) == ERROR) {
                if (LaunchInstanceDaemon(inst) == ERROR) {
                        AbortInstance(inst);
                        return;    // thread exit
                }
        }
        while (ForwardClientToInstance(WaitForEnqueuedFd(inst), inst) == SUCCESS)
                continue;
}
```

Connecting to a pcscd instance entails:

1. Opening instance's fifos. If they don't exist, an error is returned to caller.
2. If instance PID file exists, but process not running, an error returned to caller.
3. PING message sent to instance. If “instance died” message is read from fifo instead of ACK, or ping times out, an error returned to caller.
4. Otherwise, success status returned (indicates connection to instance succeeded).

If the connection *succeeds*, the client's connection fd is sent to the instance via fifo.

If the connection *fails*, an attempt is made to launch a new instance process to be managed by the instance thread. If launch failes fails, all the connections queued to the instance thread (ie. pending) and the current connection in process, are closed and the instance thread is aborted, logging an error to syslog.

Launching a pcscd instance:

The instance is launched by invoking /bin/ctrun with flags to cause it to simply start ***/usr/lib/smartcard/pcscd -I -x :<dpy#>*** then ctrun exits.

Instance Behavior

The instance performs the duties of the standalone daemon from the opensource project, with some important differences. The purpose of the opensource daemon is to handle the server side privileged functions and smartcard reader access via ifd handlers.

The instance receives client requests as marshalled RPC-like packets over a socket, unpacks them, dispatches to the appropriate function, and returns the status.

When pcscd process starts in instance mode, it loads the launcher's configuration data, ***/etc/smartcard/pcscd.conf***, it's display-specific platform configuration file, for example ***/etc/smartcard/pcscd-Local.conf***, or **/etc/smartcard/pcscd-SunRay.conf**, as well as the platform-specific preparatory script, such as ***/usr/lib/smartcard/pcscd-Local*** which, for the Sun Ray platform, starts a utaction script that will terminate the instance when the Sun Ray session exits.

The instance then enters a loop where it receives client connections (fd's) from the launcher over it's fifo. If authentication is enabled, these connections were already validated and authenticated (during the validation phase).

Each incoming file descriptor is then processed in the exactly the same manner as open source code dis. The difference is that rather than receive the fd's via ***accept()***, in instance mode, pcscd receives the client connection (fd) via ***ioctl(..., I_RECVFD, ...)*** From the standpoint of the instance, both mechanisms simply look like socket connection fd factories. (Similar interprocess fd-passing mechanisms to I_RECVFD exist for Linux and BSD).

If the instance times out, ***INSTANCE_TIMEOUT*** seconds the last client connection is terminated, the instance drops an “instance died” message into it's outgoing fifo, ***/var/run/pcscd/fifo/<dpy#>.i*** before exiting. That is an important optimization that allows the launcher to detect that the instance is gone without having to wait (ie. stall the client) for a full several second timeout cycle.

To support the socket interface being used exclusively, instead of a memory mapped file, a new protocol was developed along side the existing custom PC/SC-Lite RPC protocol over the client/daemon socket to return the state structure of specific readers.

Plugins

The following plugin schemes are provided:

1. Display validation
2. Daemon access authentication
3. IFD handler access authentication (current IFD auth plugins are NOPs, ie. Always succeed)

Plugins are designed to be reloadable without shutting down pcscd service. This is done when the timestamp on the associate plugin configuration file changes. Reload is also invoked if the timestamp of any of the plugins change, but changing plugin binaries while loaded is not recommended, as ***dlclose()*** is not guaranteed to unmap memory (handling dl binary changes midstream needs further investigation).

Currently, the generic plugins are configured as NOPs (that is, they always succeed). They would come into effect if the Sun Ray specific validation plugin doesn't recognize the referenced display.

Validation plugins are called in the order in which they're listed in ***pcscd-validate.conf*** and are passed flags and options specified to be processed with ***getopt()***.

The interfaces used are:

1. **initValidate()** - Called by pcscd to have plugin intialize itself.
2. **getDisplayStatus()** - Indicate the availability of the display for use
3. **getDisplayTag()** - Return the display's facility-specific name if valid”
4. **getDisplayResource()** - Get facility-specific resource for display as void *.

Sun Ray authentication plugin interfaces

1. **init()** **-** looks for .conf file of same prefix and loads KVPs from it, if any.
2. **isAuthorized()** **-** checks for UID of client as XID field in ***/tmp/SUNWut/session_proc/<dpy#> .*** Upon match, client is determined to be the valid display owner, and authentication succeeds.

Validation plugin interface (excerpt from pcscd-Validate.h):

```
/*
 * Plugin entry points (plugin developer must implement these)
 *
 *   initValidation(int argc, char **argv, int *errnop):
 *
 *      This function is called by the pcscd daemon with argc, argv, in the
 *      same manner that main() is called by UNIX-like OSes, and may be parsed
 *      using getopt().  This function's job is to do whatever setup is
 *      necessary in order to use the validation functions also defined
 *      plugin.
 *
 *      NOTE: This function can be called more than once. Code accordingly!
 *
 *      Function Arguments:
 *
 *              argc           Argument count
 *              argv           Argument vector
 *              errno          Pointer to errno (so plugin uses correct one)
 *
 *      Return values:
 *               SUCCESS  = 1
 *               FAIL     = 0
 *
 *   getDisplayTag(int dpyNbr, char **facilityTag):
 *
 *      This function returns the platform-specific name of the facility that
 *      the display belongs to. For example, it could be the 'tag' value that
 *      is defined for the dispaly in the Xservers file.
 *
 *      Function arguments:
 *             dpyNbr          Passed from pcscd to plugin
 *             **facilityTag   Ptr to bufptr passed to plugin to return tag into
 *                    The caller must free the buffer;
 *
 *      Return values:
 *             Same value returned to *facilityTag;
 *
 *
 *   getDisplayResource(int dpyNbr, void **resource):
 *
 *      This function returns the platform-specific resource associated with
 *      the display.  For example, it could be the whole entry that defines
 *      the display in the Xservers file, or something else.   It is up to
 *      the platform to decide.  This resource argument will be passed to
 *      the authentication plugin during daemon access authentication.
 *
 *      Function arguments:
 *       dpyNbr         Passed from pcscd to plugin
 *       resource       Ptr to bufptr passed to plugin to return res into
 *                               The caller must free the buffer.
 *
 *      Return values:
 *               Same value returned to *resource
 *
 *
 *    getDisplayStatus(int dpyNbr, unsigned int *flags):
 *
 *       This function returns whether or not a display is recognized
 *       and valid, and returns flags providing extra information
 *       about the display pcscd will use to make decisions about
 *       controlling access.
 *
 *       Function arguments:
 *              dpyNbr        Passed from pcscd to plugin
 *       flags         Flags returned from plugin to pcscd:
 *
 *       Return values:
 *               DISPLAY_IS_VALID
 *                DISPLAY_NOT_VALID
 *
 *
 */
int   initValidate(int argc, char **argv);
char *getDisplayTag(int dpyNbr, char **facilityTag);
void *getDisplayResource(int dpyNbr, void **resource);
int   getDisplayStatus(int dpyNbr, unsigned int *flags);
```

Daemon Authentication plugin interface (excerpt from pcscd-auth.h):

```
typedef struct kvp_list {
        struct kvp_list *next;
        char *key;
        char *val;
} kvp_t;

/*
 * Plugin entry points (plugin developer must implement these):
 *
 *  init(kvp_t *kvps)
 *
 *      This function is called by the pcscd daemon with a list of keys as
 *      the argument so the plugin can do self-setup.  The key list is valid
 *      only in the scope of this function, so if the values need to be
 *      accessed afterwards a local copy or representation must be made.
 *
 *      NOTE: This function can be called more than once. Code accordingly!
 *
 *      Function arguments:
 *       kvps             keys value pairs passed from pcscd to plugin
 *
 *      Return values:
 *               SUCCESS  = 1
 *               FAIL     = 0
 *
 *  isAuthorized(PCSCLITE_CRED_T *cred, const void *resource)
 *
 *      This function determes whether the client is authorized to access
 *      the pcscd daemon.  The display #, as well as the credentials of the
 *      client are passed in the cred structure.
 *
 *      The resource argument is a determined by the plugin that validated
 *      the display prior to this authentication plugin being called, and
 *      can be NULL.
 *
 *      Function arguments:
 *               cred             client cred struct from pcscd to plugin
 *               resource display resource from pcscd to plugin
 *
 *      Return values:
 *              SUCCESS = 1
 *              FAIL    = 0
 */
int init(kvp_t *kvps);
int isAuthorized(PCSCLITE_CRED_T *cred, const void *resource);

/* Convenience functions:
 *
 *  findValueForKey(const kvp_t *kvps, const char *key)
 *
 *       Optionally called by plugin to lookup specified key in a case
 *       insensitive way.  The list of key-value pairs must be passed thru
 *       the kvps argument.  The function returns the corresponding value
 *       if the key is located, otherwise it returns NULL.
 *
 *      Function arguments:
 *               kvps             key value pairs from plugin to pcscd
 *               key              key to find, plugin to pcscd
 *
 *  isKeyValueTrue(const kvp_t *kvps, const char *key)
 *
 *       Optionally called by plugin to lookup a key and determine
 *       if its corresponding value is one of the following strings:
 *       "TRUE", "true", "YES", "yes", "ON", "on", or "1".
 *
 *      Function arguments:
 *               kvps             key value pairs from plugin to pcscd
 *               key              key to evaluation, plugin to pcscd
 *
 *       The function returns status 1 (boolean TRUE) if the key is defined
 *       and the key's value is set to one of the aforemrentioned strings,
 *       otherwise the function returns 0 (boolean FALSE).
 */

char *findValueForKey(const kvp_t *kvps, const char *key);
int   isKeyValueTrue(const kvp_t *kvps, const char *key);
```

IFD Handler Authentication plugin interface (excerpt from pcscd-ifd.h):

```
typedef struct kvp_list {
        struct kvp_list *next;
        char *key;
        char *val;
} kvp_t;

/*
 * Plugin entry points (plugin developer must implement these):
 *
 *  initIfdAuth()
 *
 *      This function is called by the pcscd daemon with a list of keys as
 *      the argument so the plugin can do self-setup.  The key list is valid
 *      only in the scope of this function, so if the values need to be
 *      accessed afterwards a local copy or representation must be made.
 *
 *      NOTE: This function can be called more than once. Code accordingly!
 *
 *      Function arguments:
 *              kvps             keys value pairs passed from pcscd to plugin
 *
 *      Return values:
 *              SUCCESS  = 1
 *              FAIL     = 0
 *
 *  isAuthorizedForIfd()
 *
 *      This function determes whether the client is authorized to access
 *      the specific ifd handler.  The display #, as well as the credentials of
 *      the client are passed in the cred structure.
 *
 *      The resource argument is the AUTHSERVICE argument defined in the
 *      reader configuration file that associated the reader with the
 *      ifd handler whose access is being authenticated here.
 *
 *      Function arguments:
 *              cred             client cred struct from pcscd to plugin
 *              resource display resource from pcscd to plugin
 *
 *      Return values:
 *             SUCCESS = 1
 *             FAIL    = 0
 */
int init(kvp_t *kvps);
int isAuthorized(PCSCLITE_CRED_T *cred,
        const char *ifdHandlerName, const void *resource);

/* Convenience functions:
 *
 *  findValueForKey()
 *
 *       Optionally called by plugin to lookup specified key in a case
 *       insensitive way.  The list of key-value pairs must be passed thru
 *       the kvps argument.  The function returns the corresponding value
 *       if the key is located, otherwise it returns NULL.
 *
 *       Function arguments:
 *              kvps             key value pairs from plugin to pcscd
 *              key              key to find, plugin to pcscd
 *
 *  isKeyValueTrue()
 *
 *       Optionally called by plugin to lookup a key and determine
 *       if its corresponding value is one of the following strings:
 *       "TRUE", "true", "YES", "yes", "ON", "on", or "1".
 *
 *       The function returns status 1 (boolean TRUE) if the key is defined
 *       and the key's value is set to one of the aforemrentioned strings,
 *       otherwise the function returns 0 (boolean FALSE).
 *
 *      Function arguments:
 *              kvps             key value pairs from plugin to pcscd
 *              key              key to find, plugin to pcscd
 */
char *findValueForKey(const kvp_t *kvps, const char *key);
int   isKeyValueTrue(const kvp_t *kvps, const char *key);
```