**Design Document for ProxyCmd**

**1. Introduction**

**HiveManager** is Aerohive’s centralized network management system that enables sophisticated identity-based policy management, simplistic device configuration, HiveOS updates, and monitoring and troubleshooting of Aerohive platforms within a cooperative control network infrastructure. HiveManager is deployed on the public cloud. Access points on the enterprise side are connected to the Switches and Switches are connected to the router and all communication between public network and private network is through router. HiveManager can not communicate to the Access points but access points can. HiveManager can not initiate the the communication.

**HiveAgents** are deployed on one of the access point which is connected to the Switch. These Hive Agents are helping HiveManager to communicate with the Switches to which they are connected. On HiveAgent ProxyCmd is running and actually this is responsible for the comminication between the HiveManager to Switch.

**ProxyCmd** uses Secure SHell (SSH) for the communication which is initiated by the HiveManager. By using Local port forwarding and Remote port forwarding a secure tunnel is created to access the CLI of the Switch. HiveManager can initiate the request by passing credentials of the HM and HA to the ProxyCmd and ProxyCmd first creates the tunnel between HM and HA by using remote port forwarding and after that tunnel betwwen HA and Switch by using local port forwarding. See Figure 1 for the abstract view of the Network Topology.



**Figure 1. Network Topology**

**2. Problem Statement**

HiveManager which is on public cloud can not comunicate to the Switch which is in private network by initiating request. HiveManager wants to access the Command Line Interface (CLI) of the Switch. Hive Manger can request the HiveAgent for command line interface of switch, which is reside on enterprise side. HiveAgent only access the switch terminal as it’s in same network but directly Hive Manger will not. In current requirement only single terminal session is allowed to access the console.

**3. Overall Concept**

Hive Manager will send the REST request to Hive Agent for terminal access. On response Hive Agent will provide the response and later on Hive Agent will access the web based console through which he will directly access the terminal.

In order to get the end point terminal (switch), HM( hence forth HM for Hive Manager) will request the HA (hence forth HA for Hive Agent) through REST API (POST).

Request:-POST and BODY will contains the information like SwitchIp, HM Ip, HM UserName, HM Password, HM Port.

Response:-Response will be HTTP standard response code 200(HTTP\_OK) or if it fails then response HTTP\_ERROR with error description.

For Overall flow refer Figure 2.



**Figure 2. Overall Flow**

**4. Technical Design**

Basically we will create two SSH tunnels, one between HM and HA, this tunnel will be remote SSH port forwarding type. Second tunnel will be formed after first tunnel success response, which is in between HA and end point(Switch) and it will be a Local Port forwarding type. See Figure 3.



**Figure 3. Tunnel Creation**

**Forwarding connections:**

Port forwarding comes in SSH protocol in two different flavours: direct or reverse port forwarding. Direct port forwarding is also named local port forwardind, and reverse port forwarding is also called remote port forwarding.

1. **Direct port forwarding**: Direct port forwarding is from client to server. The client opens a tunnel, and forwards whatever data to the server. Then, the server connects to an end point. The end point can reside on another machine or on the SSH server itself.
2. **Reverse port forwarding:** The reverse forwarding is slightly different. It goes from server to client, even though the client has the initiative of establishing the tunnel. Once the tunnel is established, the server will listen on a port. Whenever a connection to this port is made, the server forwards the data to the client.

**ProxyCmd:**

ProxyCmd is implemented in C using libssh2-1.6.0 and pthread libraries. In ProxyCmd.c file two threads are created first is RemotePortForwarding and another one is LocalPortForwarding.

**libssh2-1.6.0 API's:**

**1. libssh2\_init():**

* int libssh2\_init(int flags);
* Global library initialization
* Initialize the libssh2 functions. This typically initialize the crypto library. It uses a global state, and is not thread safe -- you must make sure this function is not called concurrently.
* Returns 0 if succeeded, or a negative value for error.

**2. libssh2\_session\_handshake():**

* int libssh2\_session\_handshake(LIBSSH2\_SESSION \*session, libssh2\_socket\_t socket);
* Perform the SSH handshake
* session - Session instance as returned by libssh2\_session\_init\_ex.
* socket - Connected socket descriptor. Typically a TCP connection though the protocol allows for any reliable transport and the library will attempt to use any berkeley socket.
* Begin transport layer protocol negotiation with the connected host.
* Returns 0 on success, negative on failure.

**3. libssh2\_hostkey\_hash():**

* const char \* libssh2\_hostkey\_hash(LIBSSH2\_SESSION \*session, int hash\_type);
* return a hash of the remote host's key
* session - Session instance as returned by libssh2\_session\_init\_ex.
* hash\_type - One of: LIBSSH2\_HOSTKEY\_HASH\_MD5 or LIBSSH2\_HOSTKEY\_HASH\_SHA1. Returns the computed digest of the remote system's hostkey. The length of the returned string is hash\_type specific (e.g. 16 bytes for MD5, 20 bytes for SHA1).
* Computed hostkey hash value, or NULL if the information is not available (either the session has not yet been started up, or the requested hash algorithm was not available). The hash consists of raw binary bytes, not hex digits, so it is not directly printable.

**3. libssh2\_userauth\_list():**

* char \*libssh2\_userauth\_list(LIBSSH2\_SESSION \*session, const char \*username,unsigned int username\_len); return a hash of the remote host's key
* list supported authentication methods.
* session - Session instance as returned by libssh2\_session\_init\_ex.
* username - Username which will be used while authenticating. Note that most server implementations do not permit attempting authentication with different usernames between requests. Therefore this must be the same username you will use on later userauth calls.
* username\_len - Length of username parameter.Send a SSH\_USERAUTH\_NONE request to the remote host. Unless the remote host is configured to accept none as a viable authentication scheme (unlikely), it will return SSH\_USERAUTH\_FAILURE along with a listing of what authentication schemes it does support. In the unlikely event that none authentication succeeds, this method with return NULL. This case may be distinguished from a failing case by examining libssh2\_userauth\_authenticated.
* On success a comma delimited list of supported authentication schemes. This list is internally managed by libssh2. On failure returns NULL.

**4. libssh2\_userauth\_password ():**

* int libssh2\_userauth\_password(LIBSSH2\_SESSION \*session, const char \*username, const char \*password);
* convenience macro for libssh2\_userauth\_password\_ex
* This is a macro defined in a public libssh2 header file that is using the underlying function libssh2\_userauth\_password\_ex.

**5. libssh2\_userauth\_hostbased\_fromfile():**

* int libssh2\_userauth\_hostbased\_fromfile(LIBSSH2\_SESSION \*session, const char \*username, const char \*publickey, const char \*privatekey, const char \*passphrase, const char \*hostname);
* convenience macro for libssh2\_userauth\_hostbased\_fromfile\_ex calls
* This is a macro defined in a public libssh2 header file that is using the underlying function libssh2\_userauth\_hostbased\_fromfile\_ex.

**6. libssh2\_channel\_direct\_tcpip\_ex():**

* LIBSSH2\_CHANNEL \* libssh2\_channel\_direct\_tcpip\_ex(LIBSSH2\_SESSION \*session, const char \*host, int port, const char \*shost, int sport);
* LIBSSH2\_CHANNEL \* libssh2\_channel\_direct\_tcpip(LIBSSH2\_SESSION \*session, const char \*host, int port);
* Tunnel a TCP connection through an SSH session
* session - Session instance as returned by libssh2\_session\_init\_ex,
* host - Third party host to connect to using the SSH host as a proxy.
* port - Port on third party host to connect to.
* shost - Host to tell the SSH server the connection originated on.
* sport - Port to tell the SSH server the connection originated from.
* Tunnel a TCP/IP connection through the SSH transport via the remote host to a third party. Communication from the client to the SSH server remains encrypted, communication from the server to the 3rd party host travels in cleartext.
* Returns Pointer to a newly allocated LIBSSH2\_CHANNEL instance, or NULL on errors.

**7. libssh2\_channel\_forward\_listen\_ex():**

* LIBSSH2\_LISTENER \* libssh2\_channel\_forward\_listen\_ex(LIBSSH2\_SESSION \*session, char \*host, int port, int \*bound\_port, int queue\_maxsize);
* LIBSSH2\_LISTENER \* libssh2\_channel\_forward\_listen(LIBSSH2\_SESSION \*session, int port);
* listen to inbound connections
* Instruct the remote SSH server to begin listening for inbound TCP/IP connections. New connections will be queued by the library until accepted by libssh2\_channel\_forward\_accept.
* session - instance as returned by libssh2\_session\_init().
* host - specific address to bind to on the remote host. Binding to 0.0.0.0 (default when NULL is passed) will bind to all available addresses.
* port - port to bind to on the remote host. When 0 is passed, the remote host will select the first available dynamic port.
* bound\_port - Populated with the actual port bound on the remote host. Useful when requesting dynamic port numbers.
* queue\_maxsize - Maximum number of pending connections to queue before rejecting further attempts.
* libssh2\_channel\_forward\_listen is a macro.
* RETURN VALUE: A newly allocated LIBSSH2\_LISTENER instance or NULL on failure.

**8. libssh2\_channel\_forward\_accept():**

* LIBSSH2\_CHANNEL \* libssh2\_channel\_forward\_accept(LIBSSH2\_LISTENER \*listener);
* accept a queued connection
* listener is a forwarding listener instance as returned by libssh2\_channel\_forward\_listen\_ex.
* RETURN VALUE: A newly allocated channel instance or NULL on failure.

**9. libssh2\_session\_set\_blocking():**

* void libssh2\_session\_set\_blocking(LIBSSH2\_SESSION \*session, int blocking);
* set or clear blocking mode on session.
* session - session instance as returned by libssh2\_session\_init\_ex.
* blocking - Set to a non-zero value to make the channel block, or zero to make it non-blocking.
* Set or clear blocking mode on the selected on the session. This will instantly affect any channels associated with this session. If a read is performed on a session with no data currently available, a blocking session will wait for data to arrive and return what it receives. A non-blocking session will return immediately with an empty buffer. If a write is performed on a session with no room for more data, a blocking session will wait for room. A non-blocking session will return immediately without writing anything.
* Returns None.

**10. libssh2\_channel\_write\_ex():**

* ssize\_t libssh2\_channel\_write\_ex(LIBSSH2\_CHANNEL \*channel,int stream\_id, char \*buf, size\_t buflen);
* write data to a channel stream blocking
* Write data to a channel stream. All channel streams have one standard I/O substream (stream\_id == 0), and may have up to 2^32 extended data streams as identified by the selected stream\_id. The SSH2 protocol currently defines a stream ID of 1 to be the stderr substream.
* channel - active channel stream to write to.
* stream\_id - substream ID number (e.g. 0 or SSH\_EXTENDED\_DATA\_STDERR)
* buf - pointer to buffer to write
* buflen - size of the data to write
* libssh2\_channel\_write and libssh2\_channel\_write\_stderr are convenience macros for this function.
* libssh2\_channel\_write\_ex will use as much as possible of the buffer and put it into a single SSH protocol packet. This means that to get maximum performance when sending larger files, you should try to always pass in at least 32K of data to this function.
* RETURN VALUE: Actual number of bytes written or negative on failure. LIBSSH2\_ERROR\_EAGAIN when it would otherwise block. While LIBSSH2\_ERROR\_EAGAIN is a negative number, it isn't really a failure per se.

**11. libssh2\_channel\_read\_ex():**

* ssize\_t libssh2\_channel\_read\_ex(LIBSSH2\_CHANNEL \*channel, int stream\_id, char \*buf, size\_t buflen);
* ssize\_t libssh2\_channel\_read(LIBSSH2\_CHANNEL \*channel, char \*buf, size\_t buflen);
* ssize\_t libssh2\_channel\_read\_stderr(LIBSSH2\_CHANNEL \*channel, char \*buf, size\_t buflen);
* read data from a channel stream.
* Attempt to read data from an active channel stream. All channel streams have one standard I/O substream (stream\_id == 0), and may have up to 2^32 extended data streams as identified by the selected stream\_id. The SSH2 protocol currently defines a stream ID of 1 to be the stderr substream.
* channel - active channel stream to read from.
* stream\_id - substream ID number (e.g. 0 or SSH\_EXTENDED\_DATA\_STDERR)
* buf - pointer to storage buffer to read data into
* buflen - size of the buf storage
* libssh2\_channel\_read and libssh2\_channel\_read\_stderr are macros.
* RETURN VALUE:Actual number of bytes read or negative on failure. It returns LIBSSH2\_ERROR\_EAGAIN when it would otherwise block. While LIBSSH2\_ERROR\_EAGAIN is a negative number, it isn't really a failure per se. Note that a return value of zero (0) can in fact be a legitimate value and only signals that no payload data was read. It is not an error.

**12. libssh2\_channel\_eof():**

* int libssh2\_channel\_eof(LIBSSH2\_CHANNEL \*channel);
* check a channel's EOF status
* channel - active channel stream to set closed status on.
* Check if the remote host has sent an EOF status for the selected stream.
* Returns 1 if the remote host has sent EOF, otherwise 0. Negative on failure

**13. libssh2\_channel\_free():**

* const char \* libssh2\_hostkey\_hash(LIBSSH2\_SESSION \*session, int hash\_type);
* all resources associated with a channel
* int libssh2\_channel\_free(LIBSSH2\_CHANNEL \*channel);
* channel - Channel stream to free.
* Release all resources associated with a channel stream. If the channel has not yet been closed with libssh2\_channel\_close, , it will be called automatically so that the remote end may know that it can safely free its own resources.
* Return 0 on success or negative on failure. It returns LIBSSH2\_ERROR\_EAGAIN when it would otherwise block. While LIBSSH2\_ERROR\_EAGAIN is a negative number, it isn't really a failure per se.

**14. libssh2\_session\_disconnect\_ex():**

* int libssh2\_session\_disconnect\_ex(LIBSSH2\_SESSION \*session, int reason, const char \*description, const char \*lang);
* int libssh2\_session\_disconnect(LIBSSH2\_SESSION \*session, const char \*description);
* terminate transport layer
* session - Session instance as returned by libssh2\_session\_init\_ex,
* reason - One of the Disconnect Reason constants.
* description - Human readable reason for disconnection.
* lang - Localization string describing the language/encoding of the description provided.
* Send a disconnect message to the remote host associated with session, along with a reason symbol and a verbose description.
* As a convenience, the macro libssh2\_session\_disconnect, is provided. It calls libssh2\_session\_disconnect\_ex, with reason set to SSH\_DISCONNECT\_BY\_APPLICATION and lang set to an empty string.
* Return 0 on success or negative on failure. It returns LIBSSH2\_ERROR\_EAGAIN when it would otherwise block. While LIBSSH2\_ERROR\_EAGAIN is a negative number, it isn't really a failure per se.

**15. libssh2\_session\_free():**

* int libssh2\_session\_free(LIBSSH2\_SESSION \*session);
* frees resources associated with a session instance
* Frees all resources associated with a session instance. Typically called after libssh2\_session\_disconnect\_ex.
* Return 0 on success or negative on failure. It returns LIBSSH2\_ERROR\_EAGAIN when it would otherwise block. While LIBSSH2\_ERROR\_EAGAIN is a negative number, it isn't really a failure per se.

**16. libssh2\_exit():**

* void libssh2\_exit(void);
* global library deinitialization
* Exit the libssh2 functions and free's all memory used internal.

**Installation Steps:**

1. Download libssh2-1.6.0.tar.gz zip file from <http://www.libssh2.org/download/>.

2. Unzip libssh2-1.6.0.tar.gz to libssh2-1.6.0 into any directory.

3. Open terminal and switch to libssh2-1.6.0 directory "**cd libssh2-1.6.0**"

4. Configure by using command " **./configure**".

5. If any error or unsuccessful configuration then need to install libssl-dev by using "**sudo atp-get install libssl-dev**" and **"sudo apt-get install libssh2-dev"**

6. Type command "**make all install**".

7. Ready to use.

**Compilation of ProxyCmd:**

For compilation of the ProxyCmd we need to link the libraries of lpthread and libssh2. Following command is used for compilation:

**gcc -I libssh2-1.6.0/src -o ProxyCmd ProxyCmd.c -lssh2 -lpthread**

**5. References**

[**http://www.libssh2.org/**](http://www.libssh2.org/)

[**http://libssh2.sourceforge.net/doc/**](http://libssh2.sourceforge.net/doc/)

[**http://www.libssh2.org/download/**](http://www.libssh2.org/download/)

[**https://help.ubuntu.com/community/shellinabox**](https://help.ubuntu.com/community/shellinabox)