LRPC: Lite RPC - An SSH wrapper in C++

The aim of this project is enabling remote program execution from within C++ programs. A typical usage scenario can be described as follows: assume you are tasked with writing a program that somewhere during its operation needs to execute a program on a remote machine; while doing so, your program should be able to pass input to the remote program and receive the output produced by the program and probably process it afterwards.

While many heavyweight solutions exist for this situation, ranging from plain old RPC (Remote Procedure Call) [1] to Java RMI[2] and CORBA (Common Object Request Broker Architecture) [3], all of these technologies are very complex and have steep learning curves and significant amount initial configuration and management associated with them. They also increase the size of the program binary significantly resulting in large executables occupying unnecessarily large amount of memory.

Such a remote code execution functionality is a requirement in many distributed applications, most of which utilized rsh command shipped with most UNIX systems until recently. The latest trend is the utilization of SSH (Secure SHELL) [4] for this purpose. The most well known implementation of the SSH protocol is OpenSSH[5] which is a cryptographically secure rsh replacement that provides all of rsh's functionality plus some more. However, most of the projects, like OpenPBS[6] are using SSH from within the shell scripts. The requirements of the project lead to implementation of a tiny SSH wrapper that could be used from C++ programs without resorting to shell scripts.

Although the project does not aim to provide any sort of remote call invocation, it was called LRPC which is not far from the truth if one considers remote programs as functions, taking input and producing output. First, a notion of a compliant program needs to be defined. The programs that can be executed via LRPC need to take all their input by reading from standard input and produce their output by writing to standard output. Any program managing its input and output in a different way, like writing to a file or obtaining file names from command-line arguments is not LRPC compliant at the moment.

The first step for preparing an environment where LRPC could be utilized, is the setup of a functioning SSH server on all remote machines where programs are going to be executed. Since most Linux distributions come with preinstalled SSH servers that are setup and configured during the installation, the details of the setup from scratch will not be discussed here. A good reference on the most popular implementations of the SSH protocol is [7].

While this document is not a guide to SSH, some introduction is necessary for understanding its use in the project. Among the most important SSH features are secure remote logins, secure file transfer, secure remote command execution, access control and port forwarding. In order to achieve these functionalities, SSH protocol provides means for authentication, encryption and data integrity through the use of cryptographic techniques. There has been two versions of the protocol up to now, this document assumes the use of protocol version 2, which is the most prevalent protocol in use today. The document also assumes the use of OpenSSH implementation, again due to its popularity. Since different SSH implementations use different configuration files and command-line options to achieve the same goals, certain parts of the program should be changed in case a different SSH implementation is used. However, assuming that the whole wrapper and the driver code for the project is about 70 lines of C++ code, it should not be a problem.

SSH provides different authentication options, each with its strengths and weaknesses. In order to achieve password less remote logins -- since the remote code execution started from within the program will not be interactive -- our setup utilizes public-key authentication; other options include host based authentication and Kerberos authentication but public-key is authentication is used due to its flexibility.

An SSH identity uses a pair of keys, one private and one public. The private key is a closely guarded secret only we have. Our SSH clients use it to prove your identity to servers. The public key is, like the name says, public. It is placed into our accounts on SSH server machines. During authentication, the SSH client and server have a little conversation about your private and public key. If they match (according to a cryptographic test), the identity is proven, and authentication succeeds.

As can be seen from the above paragraph, each key pair is associated with a user, hence a user is needed on a remote machine which owns the programs that we are going to executed. The wrapper has a hard-coded user name of lrpc, which should be created on every machine where we would like to executed our programs. Naturally, program binaries should be placed within the account of this lrpc user and should have appropriate permission for the owner. Assuming we are developing our program on a Linux system, we should create our key pair and place our public key in every machine under the lrpc account. This basically concludes the setup required for LRPC. A detailed list of steps is given below:

1. Create lrpc user on every remote machine (done as root)

```
[root@remotehost ~]# useradd lrpc
```

2. Generate key pair for our own user. Here, in order to have pass wordless program execution, it's imperative to enter an empty pass phrase (just press enter); it has a security implication that if someone obtains your password or manages to obtain root privileges on your client machine, your private key can be obtained. In order to minimize the risk associated with it, it is a good idea to create the key pair specifically for use with the LRPC and use a different pass phrase protected key pair for all your confidential logins. Another option is the use of ssh-agent in which case you would enter a pass phrase for LRPC user as well but use a ssh-agent which would obtain the pass phrase on startup and provide the decrypted private key every time it is required for authentication with remote host. The details of this setup can be read in [7].

```
$ ssh-keygen -t dsa      Generating public/private dsa key pair.
Enter file in which to save the key (/home/user/.ssh/id_dsa): press ENTER
Enter passphrase (empty for no passphrase): ********
Enter same passphrase again: ********
Your identification has been saved in /home/user/.ssh/id_dsa.
Your public key has been saved in /home/user/.ssh/id_dsa.pub.
The key fingerprint is:
14:ba:06:98:a8:98:ad:27:b5:ce:55:85:ec:64:37:19 user@localhost
```

3. Once the key generation is complete, we need to put our public key on the remote machine. This is

as easy as creating a .ssh directory for the lrpc user and appending our public key to the .ssh/authorized_keys file

```
$ scp .ssh/id_dsa.pub lrpc@remotehost:
$ ssh lrpc@remotehost "cat id_dsa.pub >> .ssh/authorized_keys"
```

4. Place the program executables in the home directory of lrpc on remote host. This step concludes the setup required for LRPC.

Once the setup is complete, a test can be made to see if the setup works. Following steps show how a simple test can be done. A simple program that reads numbers from its input and increments them and outputs them is shown below. This is the program that will be stored on the remote machine and will be executed from within our driver code:

```
#include <iostream>
using namespace std;
int main()
{
    double x;
    while (cin >> x)
        cout << ++x << endl;
    return 0;
}
```

Below is the driver program that represents a code developed by the user of the LRPC. It starts by creating a file called "input" and filling it with integers from 0 to 9. This file will be passed as an input to the above program residing on the remote machine. Next, it constructs an LRPC object with the specified constructor parameters the details of which will be discussed next. The constructor takes parameters of hostname, the name of the remote program binary, the name of the local input file and the name of the local output file. Once execution is complete, the result produced by the remote execution can be found in the file named "output" in the current directory, which is read and displayed by the driver program.

```
#include <iostream>

#include <fstream>

#include <stdexcept>

#include "lrpc.hpp"


using namespace std;


int main()

{

    ofstream input("input");


    for (int i = 0; i < 10; i++)

        input << i << endl;


    lrpc foo("remotehost", "adder", "input", "output");

    try {

        foo.exec();

    } catch (runtime_error e) {

        cout << e.what() << endl;

    }


    ifstream output("output");

    string line;


    while (getline(output, line))

        cout << line << endl;

    return 0;

}
```

Here is result of the execution of the above program:

```
[user@localhost]$ ./driver
```

```
1
2
3
4
5
6
7
8
9
10
[user@localhost]$
```

As expected, the remote program increments the contents of the input file which are the numbers from 0 to 9 and prints them on the standard output. Below is the quick overview of the LRPC object and its implementation. The LRPC class is defined as follows:

```
class lrpc {
public:
    lrpc(const std::string& rhost, const std::string& cmd,
         const std::string& infile, const std::string& outfile):
        m_rhost(rhost), m_cmd(cmd), m_infile(infile), m_outfile(outfile) {}
    void exec();
private:
    std::string m_rhost, m_cmd, m_infile, m_outfile;
};
```

The class has a single constructor, with parameters of representing remote hostname, remote program name, filename of the local input file and the filename of the local output file. The constructor is implemented with initializer list just by setting the respective values to private members. The only method the class has is exec() which forms the command string to be passed to the shell and calls the system() call from the standard C library for passing the formed string to the shell for execution. Although executing system() call like exec() does s known to have some security implications and execl() from the standard library could be used instead, it is believed that the execution environment of the program will not be hostile and using system() call has the benefits of built-in input/output redirection provided by the shell. As can be seen below, the path of SSH binary and the username are hard coded, hence any changes in these would require recompilation of the LRPC and the program utilizing it.

```
const string ruser = "lrpc@";
const string sshpath = "/usr/bin/ssh";

void lrpc::exec()
{
    string cmd = sshpath + " " + ruser + m_rhost + " ./" +
        m_cmd + " <" + m_infile + " >" + m_outfile;
    if (system(cmd.c_str()) < 0)
        throw runtime_error("system() failed");
}
```

This concludes the discussion of LRPC which is simply a wrapper around the SSH binary.

References

1. Version 2 of ONC RPC - http://tools.ietf.org/html/rfc1831
2. A Distributed Object Model for the Java System - http://pdos.csail.mit.edu/6.824/papers/waldo-rmi.pdf
3. The official CORBA standard from the OMG group - http://www.omg.org/docs/formal/04-03-12.pdf
4. The Secure Shell Authentication Protocol - http://tools.ietf.org/html/rfc4252
5. OpenSSH - http://www.openssh.com
6. PBS GridWorks: OpenPBS - http://www.pbsgridworks.com
7. Daniel J. Barrett, Richard E. Silverman, and Robert G. Byrnes — SSH: The Secure Shell (The Definitive Guide), O'Reilly 2005 (2nd edition)