Design of a Secure RTU for SCADA Systems
M. Eng. Progress Report
Fall 2010

Brad Luyster

Table of Contents

Section 1.0 - Introduction

During the Fall 2010 Semester, significant progress has been made towards completely implementing the architecture described by Drs. Graham and Hieb, and described in more detailed in the previous progress report for Summer 2010. Although the goals for the semester at the outset were ambitious, the majority were completed.

This document will serve as a summary for the success of the Fall 2010 semester, documenting any implementations, and architectural changes which were made as a result of the discoveries of the semester. In large part, the overarching architecture of the project has remained the same. In some small instances or submodules, minor alterations have taken place.

To recap, at the outset of the project, the architecture as-designed for had been described by Drs. Graham and Hieb[1] . This architecture may be described below:

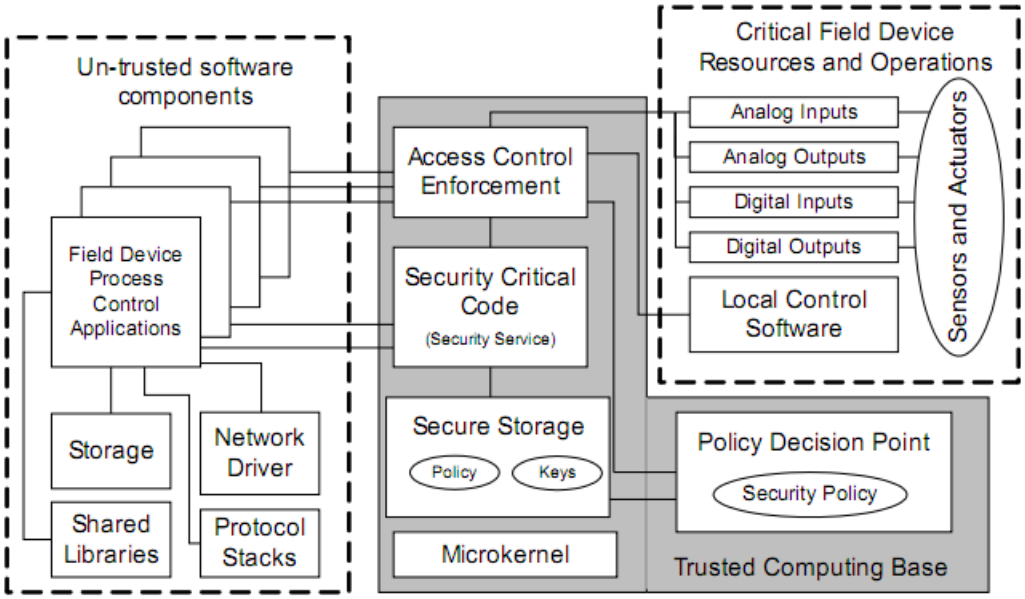

Illustration 1: Reference RTU Architecture

This architecture describes a security enhanced RTU. An RTU is a device used in Supervisory Control and Data Acquisition (SCADA) systems to control remote devices and sensors. These SCADA systems are networks of control devices which operate systems such as the power grid, the water distribution system, and many industrial plant processes. The pumps, switches, sensors and other control devices which interact directly with the process in question are unintelligent devices with no computing or decision making power of their own. They simply accept an input, and provide some output. The RTU device provides the input required to actuate these devices. The RTU communicates to a central control room using any number of protocols and methods. Today, Ethernet is an increasingly prevalent industrial standard, and many of these devices are Ethernet connected. When these control networks interact with public networks such as the Internet, the possible attack vectors which could be used to disable these control systems increase in number rapidly.

The architecture supplied above provides for the enhanced security of these devices by separating the responsibilities of subsections, and dividing and minimizing the Trusted Computing

Base of the software running on the RTU hardware. A Microkernel based operating system provides a secure platform on which such a system may be developed. The choices and merits of Microkernels were discussed in the previous progress report. For this project, the code base is built on the OKL4 Microkernel Operating System, version 3.0. The OKL4 Microkernel is unique for undergoing the rigors of formal mathematical verification. Although the version being used to build this project is not the strictly verified code base, a verified kernel might be installed on a system as-built using the non-verified kernel with relatively minimal effort.

The following sections will document the state of the project at the beginning of the semester, explain some of the pitfalls which required architectural changes, document in detail the progress made through the semester, and explain some future directions the project might take given the current state of completion.

Section 2.0 - Summary of Progress to Date

At the outset of the semester, initial performance testing had been completed, showing the feasibility of using the OKL4 IPC calls in order to generate physical IO from messages passed between isolated Cells. Throughout the process of getting to this point, limitations of the development environment had been encountered and dealt with, requiring minor architectural changes. Furthermore, initial performance testing showed that more attention must be paid to the isolated Cells and their operating priority in the Operating System. Basic network access had been explored and implemented.

Section 2.1 - Hardware and Software Architecture Decisions

During the course of the Summer 2010 Semester, hardware specifications were generated and implemented, resulting in a device with 8 Digital IO, 8 Analog Inputs and 2 Analog output. These IO operate at the logic level of the Gumstix Host platform of 3.3 volts. The architecture decisions at the start of the semester were to mirror the original platform as designed by Drs. Graham and Hieb as closely as possible. This included a Network Access cell, a Security Cell, An IO cell, and various utility cells for functionality. Unfortunately, a limitation on the number of cells was discovered, forcing the conglomeration of all utility processes. As the security cell was not yet in place, this reduced the number of cells to three.

This was the only major software deviation from the target architecture. At this point, not trusted code had been implemented, and all code remained in the untrusted computing base. Since the utility, IO, and utility cells were not trusted components, care regarding their communications and inter-dependencies was not expressly considered in design and implementation.

Section 2.2 - Software Implementation

At the end of the Summer 2010 semester, the IO cell properly communicated with some attached devices (for demo purposes), a hardware management cell properly delegated shared hardware resources and then ceased operation, a serial debug cell provided the ability to receive more succinct debug information than the kernel debugger could provide, from multiple independent cells simultaneously. A Net IO cell provided the ability to inject raw Ethernet packets onto a network. After encountering limitations in the number of cells (caused by Elfweaver), the former 3 cells were combined into a single utility cell, each operating as its own separate thread of execution. Many of the operational paradigms of cells transfer to threads in the OKL4 operating system– For the purposes of programming, little code needed to be changed in order to compile multiple threads into a single cell. Although these threads may technically access each-others' memory sections, they do not overlap in normal operation (as should be expected).

Section 2.3 - Restructured Architecture

As a result of the required changes through the first semester of development, the next semester's target architecture shifted slightly, encompassing four cells: Security, Network Access, IO, and Utility. This architecture neatly separated pieces of the trusted computing base from the untrusted computing base, while also separating unrelated pieces of the untrusted computing base from each other. Although this was not a goal of the original specification, this allows for simpler architectural development into the future.

Section 3.0 - Goals for Fall 2010

The goals for the Fall of 2010 intuitively extended the goals of the Summer of 2010. This semester would see the implementation of some of the trusted components of the target architecture, since the summer saw the verification of the assumptions made in order for such a device to be tenable in a SCADA system (namely, response time of IPC).

Section 3.1 - Software Goals

The goals for this semester were primarily goals of software, with extensions to connectivity for providing access to commodity based solutions, as well as the implementation of some simplified versions of the target security architecture. Furthermore, a client would need to be developed and tested in order to provide the remote side of the Remote Terminal Unit.

Section 3.1.1 - Security Goals

The security goals of the project followed the security outline described by Dr. Hieb in his doctoral dissertation[2], and included handshaking and Role Based Access Control. These features required the implementation and testing of efficient algorithms for iterating through roles and performing SHA hashes, as well as generating random numbers. None of these faculties are included by default in the OKL4 development environment.

The role based access control designed for the RTU is simplified from many possible implementations, and does not include provisions for future expandability. Users maintain membership to one or more roles, and these roles have access to one or more sets of permissions. A permission is a set consisting of a point combined with point access controls. A point is a physical IO (or abstraction of physical IO) whose state may be altered, while a permission represents the actions that may be performed on the IO. In the current implementation, permissions may be read-only access or full-access.

Points have associated point types, which are labels used to more broadly control permissions. Roles have, in addition to associated permissions, point type controls, which govern whether or not a role may have access to a given point type regardless of permission. Roles further have point access constraints, which are further restrictions on how a role may access a given point. In the target implementation, only time-of-day and day-of-week point access constraints are considered, while Dr. Hieb's thesis[2] describes further restrictions, such as terminal location.

Users have a similar constraint in the form of role access constraints. Users may have restrictions placed on how they may use a particular role to which they are assigned. The goal targeted only temporal restrictions, while Dr. Hieb further expands on the possible implementations of such restrictions. Permissions in this manner are assigned in a logical OR fashion: If any of the set of permissions assigned to a user allow an action, this action is allowed to occur. If the action is not allowed to occur, the client is not notified of this failure. Only later reinspection will inform the user that their access permissions have been denied.

Handshaking in most SCADA systems makes use of a preshared secret, and typically only verifies the authenticity of the MTU to the RTU.[3][4] A SHA-256 hash of the preshared secret along with the original message as received by the RTU is performed, and returned to the RTU. If the hash generated by the MTU matches the hash generated internally by the RTU, identities are verified, and the operation is allowed to commence. For the purposes of authenticity verification, typically only the

first section of the hash is compared. Furthermore, hashing takes place only occasionally and intermittently, without respect for the operations being performed by either party.

The handshaking algorithm which we have built towards uses the same sort of handshaking algorithm, but requires both occasional verification of identity, and identification if a critical combination of point and operation are requested. For example, Reading point 1 might be a non-critical operation, while Selecting and Operating on point 1 both require authentication. For the purposes of this prototype, handshaking takes place completely separately from the role based access controls, irrespective of the incoming permissions trying to access the point. The operation may be disallowed at any point after the handshaking on the basis of role based access controls.

Section 3.1.2 - Connectivity Goals

Continuing with the progress made last semester, in order to properly test the performance of the Ethernet communication link built into the RTU it is desirable to extend the ability to place raw Ethernet packets on the wire into something more standards compliant. In this way, UDP or TCP packets may be accurately routed over a local or remote network in order to direct the RTU to perform in a certain manner.

At the close of last semester, a Linux driver for the Ethernet MAC included with the Gumstix had been adapted, and shown to perform adequately. The adaptation from Linux involved stripping out the interrupt driven packet queue, and reading packets one at a time. Addition features were removed, or remained untested, making the implementation of a complicated, stateful protocol like TCP an unlikely goal. As a result, UDP as the concession made.

Section 3.1.3 - Client Software Goals

In order to test the software configuration running on the RTU hardware, a simple client would be necessary. This client was designed to provide a simpler interface to the protocol used to manipulate points and provide a parter to handshake with across the network. This client would have to expand along with the roles and permissions included on the RTU in order to facilitate testing. The ability to later interface with a simulated or real plant is desired, but not required. The primary goals were simplicity and ease of expansion.

Section 3.2 - Testing Goals

Primary among the goals of the semester were the ability to test the performance of the RTU hardware with security layers and role based access control in place. Performing this testing both without and with a simulated plant would provide some reasonable measurements of the capability of the RTU architecture as designed. This testing would test the end-to-end performance of the device, from client request to response, as well as the performance of subsystems, to identify bottlenecks, and where performance might be improved.

Testing for the semester would be considered adequate with a reliable data gathered for client-to-server and response information. Although future testing would probe many other facets of the design, these initial goals would provided some overarching numbers for the project. While this data isn't particularly scientifically useful, it has a great deal of intuitive use, and marks an inflection point in the development of the RTU. This testing doesn't attempt to deconvolve issues of network latency and client-side software and hardware from the measurements of performance, but it does place a framework for better measurements in the future.

Section 4.0 - Architecture Alterations

Section 3 documents the target architecture, but throughout development, many problem, small and larger, were encountered which required minor alterations in the target architecture. Although the implementation of specific features within the target architecture may have been impacted, the goalposts for the semester's achievements were never moved wholesale. In large part, these architectural changes resulted from bugs in the OKL4 operating system which are non-critical to the security goals of the system, and served as time sinks (although granting a greater understanding of the operating system).

Section 4.1 - ELFWeaver Failures

At the close of the previous semester, a few bugs had been discovered in the ELFWeaver program. In the OKL4 build system, the elfweaver program takes the individually compiled cells and threads, and weaves them together with the kernel image, allocating compile-time memory, and creating a boot-image. This is, in large part, a similar process to the linker, but combining multiple threads and memory spaces in the OKL4 operating system space. This program is written in Python, and relies on a network of XML files scattered throughout the code to-be-weaved in order to define its behavior. The function calls are reentrant and polymorphic, and from the outside perspective, its very difficult to trace the action of the program.

The least problematic of bugs encountered during the previous semester was a 5 cell limit imposed by the memory allocation functions of the elfweaver program. At the close of the last semster, the software running on the RTU used three cells: One for Network IO, another for Physical IO, and a third for utility functions, such as hardware management and serial debug. The security cell was intended to take its place as a fourth cell in the project. Unfortunately, during the course of fleshing out IO and Network Access Functions, the heap and stack allocations for the remaining cells grew large enough to upset the balance of memory allocation within elfweaver, and attempting to add a fourth cell resulted in the same errors encountered last semester while adding a fifth cell.

In order to minimize deviations to the target architecture, a second exploration into the elfweaver code became necessary. After 14 hours of tracing calls throughout the system, and printing memory assignment and usage, the file containing the actual memory allocation functions were discovered. During the course of this exploration, a much greater understanding of the tasks performed by elfweaver was gained. Elfweaver first breaks the memory space of the target architecture into page-aligned sections, discarding the sections which must be assigned to boot-code. Then, it uses a lookup table in order to break these page-aligned sections into smaller sections depending on the amount of memory required by kernel operations. The specific mechanizations of this operation are still unknown, Then, the elfweaver program assigns memory to cells and threads, as they appear in the makefile for the project. Memory sections which are not large enough for the current cell or are no longer page-aligned are discarded.

Although the code which performs these operations resides in a single file within the elfweaver code tree, this code links to code at higher and lower layers, in addition to being reentrant within a single function. This code, located in tools/pyelf/weaver/allocatorr.py, is also the lengthiest in the elfweaver tree. The copyright notice is followed simply by the comment, "*Lasciate ogne speranza, voi ch'intrate."* – A quote from Dante's *Inferno,* translated as "Abandon all hope, ye who enter here."

With some consternation, efforts at finding the bug and fixing it were abandoned, and the decision was made to alter the architecture to conform to this new three cell limit. The physical IO

would be relocated within the utility code, representing the utility code's proximity to bare-metal functions. This does not compromise the target architecture or security in any way, as the Utility Cell and physical IO have always been modeled as pieces of the Untrusted Computing Base.

Section 4.2 - Role Based Access Control

The role based access control model suggested by Dr. Hieb in his Thesis is complex relative to the rest of the security features implemented on the device. In order to keep development time to a minimum, and performance of the prototype high, some features were foregone. In order to simplify development, roles, users, points and all other controls relating to the RBAC feature are hard-coded into the system, and cannot be modified at run time. Furthermore, any access constraints which depend on information other than time of day have not been designed into this system. In order to check permissions and easily iterate through sets of permissions, users, and points, many access controls are stored as bitfields. This allows the use of native bitwise logic operations in order to determine positions. Since these permissions are permissive by default, their inspection is simple.

Section 5.0 - Software Implementations

This semester's work was one of furious software implementation, in an attempt to get the RTU to a point where it could be developed for a greater point of maturation, and gather some useful data on the performance of a microkernel based SCADA system. The following section will detail the specifics of the architectural implementations described in ideal summary above. This section will also detail the many roadblocks encountered along the way.

Section 5.1 - Utility Functionality

Logically, this section falls first. In reality, however, this development occurred after the security code had already been written. Unfortunately, before this functionality could be tested, the cell limit summarized above was encountered, and the architecture had to be appropriately changed. Before any implementation and functionality testing could occur, the physical IO cell had to be moved into the utility cell.

Fortunately, the work performed during the previous semester make this task quite a bit easier. Unfortunately, during this transition, a lot of sloppy coding was discovered, as previously unknown race conditions came to the surface. Previously, the physical IO cell used IPC to communicate with a number of other threads in the Utility cell. All of these IPC calls took a (relatively) long amount of time, while the OKL4 operating system uses a more efficient method for performing IPC among threads within the same memory segment. As a result, these IPC calls were much briefer, exposing unprotected race conditions. As a result, cells began operating before the other threads with which they were required to communicate were fully activated, resulting in communications deadlocks.

While moving the physical IO ferreted out a number of these race conditions, it will be useful in the near future to perform a full diagnostic on possible race conditions, and solve the problem before it exposes itself in a possibly vulnerability-inducing way. Of particular note, the security cell uses the same IPC calls as the rest of the system. Not only must the OKL4 kernel code be trusted to perform its duty faithfully, but we must ensure that race conditions cannot possible deadlock the security cell. At the moment, there are no blocking IPC calls within the security cell which would cause the RTU to stop responding, with the exception of a single IPC call to the Network IO cell. This is a calculated decision: in any case, if the network is non-functional, the RTU will not be responding in any case.

Section 5.2 - Simple SCADA Protocol

In order to facilitate the communications goals of the project, a simplified SCADA protocol was devised which would provide the bare minimum functionality required to move into the testing phase of the project. This SCADA protocol would account for reading and writing information and IO from the RTU, as well as the handshaking described in section 3.1.1. This simplified SCADA protocol provided 6 packet types of fixed size sent over UDP. Below is the table of message type.

Table 1: Summary of Simplified SCADA Communications Protocol used in RTU Design

Operational Type	**Operation ID**	**Data**
Read Point	0x00	3 bytes
Select Point	0x01	4 bytes
Operate Point	0x02	4 bytes
Demand Challenge-Response	0x03	4 bytes
Challenge-Response	0x04	36 bytes
Read Response	0x05	4 bytes

For Reading a point, the data format is: <Operation ID>+<User ID>+<Point ID>, each one-byte field.

For Select or Operate operations, the format is: <Operation ID>+<User ID>+<Point ID>+<Data>. For initial testing, Select simply operates as a “Write” operation, although typical SCADA systems require a Select followed by an Operate, containing identical data.

For Demand Challenge-Response operations, the format is: <Operation ID>+<Server Nonce>, with the server nonce being a 4 byte field.

For Challenge-Response operations, the format is <Operation ID>+<Client Nonce>+<Hash>. The client nonce is a 4 byte field, while the hash is a 32 byte field. In this implementation, neither the client nor the server require the originating message to be hashed along with the nonce. This was excluded for simple testing and rapid development, but is a trivial addition given the current state of the code base.

For Read Response operations, the format is <Operation ID>+<User ID>+<Point ID>+<Data>.

These messages are simply stuffed into UDP packets, sent unicast on port 1234. To simplify the generation of these packets, the packet size is kept constant, and padded with zeroes if the data does not take up enough space. UDP checksums are ignored, and IP header checksums remain static, due to the static packet size. While this doesn't strictly adhere to the UDP protocol, these packets are properly routed across a network.

In order to implement UDP on the RTU, a simple set of functions were built into the network connectivity layer. These functions simply parse the correct pieces of incoming raw packets, probing for correct packet formatting. Checksums are not checked or calculating. Only UDP, ARP and Ping packets are processed. The inclusions of Ping functionality was included only as a debugging measure, and is non-critical to operation. UDP checksums on incoming packets are not calculated, while UDP checksums for outgoing packets are properly generated. The code for much of this was taken from the open source Micro-IP library, designed for providing network connectivity functionality to 8-bit microcontrollers and other small devices. As a side effect, these functions are extremely memory-frugal, but are restricted to processing a payload no larger than a single Ethernet frame. When all overhead is taken into account, this results in a maximum useful UDP payload of only 220 bytes per transaction. Within the simple SCADA protocol described above, however, this is not a limitation.

When packets are received by the RTU, they are stripped of all payload, and passed up to the security level for action. The security layer will process these packets, and if an IP response is required, will pass down the appropriate data. If no response is required, the security layer will acknowledge receipt of data, freeing the RTU to receive another packet. No receipt acknowledgment is sent to the remote client, however.

In the future, it will be desirable to transact with multiple clients, or the same client asynchronously. This will require buffering packets, and using another thread to parse the incoming packets for the security layer. It might also be useful to use the network device on the RTU in interrupt mode, as well. In order to expedite the completion of the semester's goals, however, the network device on the RTU remains in polling mode. Consideration for future expansion was made, however, and this functionality can be added with little or no modification to the higher layer functions.

Section 5.3 - Hashing and Handshaking

The most critical part of proper handshaking as described above is the ability to take a proper SHA hash of the input data. Fortunately, there are some simple open-source reference implementations these hashing functions. Although the OKL4 system provided some difficulties in their implementation, all test cases applied to these functions have worked perfectly. Principal difficulties came in the form of integer formats and irregular C standard library implementations. Unfortunately, there is no native 64 bit or 128 bit integer format within OKL4. These functions now utilize arrays of smaller integer forms. Further difficulties were found in the functions use to implement the correct endianness of the SHA functions. The OKL4 C libraries do not properly implement POSIX hooks for automatic determination of endianess, so these had to be added manually.

After this brief adaptation, a small set of test cases were performed using the known-working Linux SHA256 command. All of these tests passed flawlessly. It must be noted that the hashing function included in the prototype RTU will perform hashes irrespective of non-printable characters in the input string. As a result, its invocation must be careful to EXCLUDE newlines and null terminators, unless their inclusion is specifically desired for the given hash.

Section 5.4 - Role Based Access Control

As mentioned in section 4.2, the implementation of role based access control is a simplified version of the system described by Dr. Hieb in his dissertation[2]. The implementation largely matches

the target architecture, however, during the course of the implementation, access constraints were stubbed out by not implemented. With the simple addition of software which can read the value of the RTU's Real Time clock, these features can be enabled. This has become a goal of the new semester.

There are several layers of access control in this implementation. All are accessed readily with a simple “Check Permissions” function, which is passed a user, a point, and an operation, and returns true if the user is able to perform the given operation on the given point. Matters of system state are interrogated by the check permissions function directly. In order to speed up operation of the RTU, inspection of permissions is performed by checking simpler permissions first. By doing this, the RTU spends a minimum amount of time interrogating a list of permissions for a given action.

These lists of users, points, permissions, and other items are all short enough at the moment that simple searching through them does not incur a performance hit from list start to finish. Unfortunately, the system as-built is not very well scalable, and if more items were included in this role based access control scheme, there would be a large performance hit. More efficient sorting and searching algorithms could be implemented layer-by-layer to mitigate problems in this area.

In summary, the role based access control system, as implemented, operates as such: After the security layer has verified the authenticity of the incoming command by demanding a challenge-response (or not, as the case may be), the users permissions of the command are interrogated with the “check permissions” function. This function first searches the users roles, to see if the desired permission in included. It then interrogates the users role access constraints, followed by the roles' point access constraints. Only if all of these checks pass does the function return true.

The permissions, roles, points, permissions, and all other data used by the role based access control system are declared statically. The program must be recompiled in order to modify or add any data to this system. Data are placed into arrays in an arbitrary order, and their placement in these arrays are noted as bitfields. These bitfields are then used by the “check permissions” routine in order to quickly and easily determine access.

Currently, all data is completely arbitrary, based largely on examples from Dr. Hieb's thesis. There are 3 point types, 3 operations, 6 points, and 12 sets of permissions. These permissions are “Read Access” for all 6 points, and “Unrestricted Read/Write Access” for all 6 points. There are 7 roles, Manager, Engineer, Technician, Security Administrator, Vendor, Plant Manager, and Operator. Currently, these roles are largely identical. Manager is meant as an administrative all-access user, Engineer is slightly more restrictive, and Technician is more restrictive still. There are currently 5 test users with varying sets of assigned roles and access constraints.

Section 6.0 - Spring 2011

With the near-completion of this semster's goals come some interesting possibilities for the future. Although there is quite a bit of work to be done in improving performance, reliability, and security in the prototype, now that there is a successful security layer in place, the project can begin the testing phase, in order to determine performance under a number of conditions, as well as security hardness. The hardware can be further developed in order to work with actual control system electronics, and tested against any number of systems.

Section 6.1 - Performance and Reliability Testing

Unfortunately, the prototype fell just short of being ready for any sort of comprehensive testing by the end of the Fall 2010 semester. Fortunately, the work put forth means that this testing can be started in as little as a week or two of additional work, with some insurance that cells will not interfere as they did in the previous semester's testing. Furthermore, more interesting questions can be asked, probing the effects of network reliability and latency, along with the security features of the device. The true effects of IPC can be seen while communicating across many different cells and small pieces of hardware and software, and the limits of the OKL4 operating system can be tested. Gerwin Klein of OK Labs has written of the interesting research questions which can be posed by using a secure microkernel based operating system.[5]

Section 6.2 - Client Maturity

One path which should be followed before much further development occurs is the maturation of the client currently being used to test the hardware over a remote network connection. Unfortunately, this client is of poor quality, and relies on a hard coded awareness of UDP, IP and the RBAC system placed on the remote RTU. This client generates horrifically malformed UDP packets, which are only accepted by the end device by the coincident non-inspection of any checksum data. This client would have absolutely no hope of communicating in anything more difficult than a sterile lab network. Further flexibility in development on this end of the project would greatly ease stress on server-side development, as the client would not require changes to match changes on the server side.

Section 6.3 - Plant Simulation

With some basic end-to-end testing of the RTU system, some real-world performance data can be gathered by attaching the device to a plant simulation. Using LabView software and a DAQ, many different types of plants could be easily and quickly built, tailored to the hardware interfaces built onto the RTU. The current hardware configuration of the RTU outputs within the native logic levels of the RTU. Digital lines are 0-3.3v, and analog outputs and inputs are limited to the range of 0 to 3.3 volts. This is not within the range normally used for control devices. The LabView DAQ aquired at the outset of the project, however, could be used with either native or modified voltage or current levels with minimal alterations needed.

Section 6.4 - Hardware Enhancements

As mentioned in Section 6.3, the current hardware configuration generates analog and digital outputs with the logic levels native to the RTU hardware. For many control devices, however, output are between 0 and 9 volts, or 20-40 mA (based on an earlier survey of commercial devices, described in detail in the Summer 2010 progress report). The addition of hardware which could condition the outputs and input appropriately is a non-trivial task, but would enhance the performance of the RTU by bringing its electrical interface closer to that

of currently manufactured devices.

These hardware enhancements could also be used to provide electrical isolation between the RTU hardware and control devices. This will become absolutely critical should the prototype RTU be connected to any real hardware. Without too much effort, these electrical interfaces could be designed with the form-factor of the existing RTU taken into account, creating a physically and electrically stable expansion board which would be resistant to the sort of shocks and vibrations which would be experienced during testing.

Section 7 – References

[1] J. Hieb and J. Graham, *Designing Security-Hardened Microkernels For Field Devices*, Boston, MA: Springer US, 2009.

[2] J.L. Hieb, “Security Hardened Remote Terminal Units for Scada Networks,” 2008.

[3] P. Oman, E.O. Schweitzer, and J. Roberts, “SAFEGUARDING IEDS , SUBSTATIONS , AND SCADA SYSTEMS AGAINST ELECTRONIC INTRUSIONS,” *Technology*, pp. 1-18.

[4] C. Bowen III, T. Buennemeyer, and R. Thomas, “A Plan for SCADA Security Employing Best Practices and Client Puzzles to Deter DoS Attacks,” *Working Together: R&D Partnerships in Homeland Security*, 2005.

[5] G. Klein, “A Formally Verified OS Kernel . Now What ?.”

Appendix A – Security Layer Code

The majority of the code generated during this last semester relates to the new security cell. For the sake of brevity, only this important new code is included in this document. As we have been using source control throughout this project, it is simple to view the full differences in code between this semester and the previous.

Appendix A.1 – rbac.h

Rbac.h is a file defining the data structures used by the role based access control system.

```
#pragma once

#define PACSINGLE 0
#define PACALL 1

#define NUMPOINTS 6

typedef okl4_word_t pointID;
typedef okl4_word_t userID;
typedef okl4_word_t roleID;
typedef okl4_word_t bitfield;

struct point
{
      bitfield ppt;
      pointID pointid;

} point;

struct time
{
      okl4_u8_t minutes;
      okl4_u8_t seconds;
      okl4_u8_t hours;
} time;


struct permission
{
      bitfield operations_allowed;
      struct point *ppoint;
} permission;

struct timeconstraint
{
      struct time *disable_start;
      struct time *disable_stop;
} timeconstraint;
```

```
struct dayconstraint
{
      struct day *disable_start;
      struct day *disable_stop;
} dayconstraint;

struct permission_access_constraint
{
      int pacType;
      struct point *pacpoint;
      struct timeconstraint *timec;
      struct dayconstraint *days;
      //locationconstraint location;
} permission_access_constraint;

struct role_access_constraint
{
      bitfield roles;
      struct timeconstraint *timec;
      struct dayconstraint *days;
      //struct locationconstraint location;
} role_access_constraint;

struct role
{
      bitfield point_type_controls;
      struct permission permissions[NUMPOINTS];
      int numPacs;
      struct permission_access_constraint *PACS;
      int testArray[NUMPOINTS];

} role;



struct user
{
      userID myId;
      uint32_t *secret;
      bitfield roles;
      int numRacs;
      struct role_access_constraint *RACS;

} user;

struct authedUser
{
      userID myUser;
      struct authedUser * nextUser;
} authedUser;
```

Appendix A.2 – srole.h

Srole.h includes the test permissions, and makes use of the data structures defined by rbac.h

```
#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include "rbac.h"

#define NUMROLES 7

#define ptype1 0x00000001
#define ptype2 0x00000002
#define ptype3 0x00000004

#define SR_select 0x00000001
#define SR_read 0x00000002
#define SR_operate 0x00000004

struct point * getArrayOfPoints(void);
void buildPermissions(void);
void buildRacsAndPacs(void);
struct role * buildRoleArray(void);
void buildRoles(void);
struct user * buildUserTable(void);
void buildUsers(void);
void getPermTables(struct role,    roleID);
bitfield * roleBFAbuilder(void);

struct point digital_1 = { ptype1, 0x00 };
struct point digital_2 = { ptype2, 0x01 };
struct point digital_3 = { ptype3, 0x02 };
struct point digital_4 = { ptype1, 0x03 };
struct point digital_5 = { ptype1, 0x04 };
struct point digital_6 = { ptype1, 0x05 };

struct point * getArrayOfPoints(void)
{
      struct point *pointArray = malloc(sizeof(struct point)*6);
      pointArray[0] = digital_1;
      pointArray[1] = digital_2;
      pointArray[2] = digital_3;
      pointArray[3] = digital_4;
      pointArray[4] = digital_5;
      pointArray[5] = digital_6;


      return pointArray;
}

struct permission rd1;
struct permission rd2;
```

```
struct permission rd3;
struct permission rd4;
struct permission rd5;
struct permission rd6;

struct permission ad1;
struct permission ad2;
struct permission ad3;
struct permission ad4;
struct permission ad5;
struct permission ad6;

//ANNOYING, HAVE TO ASSIGN AT RUNTIME
void buildPermissions(void)
{
      rd1.operations_allowed = SR_read;
      rd1.ppoint = &digital_1;
      rd2.operations_allowed = SR_read;
      rd2.ppoint = &digital_2;
      rd3.operations_allowed = SR_read;
      rd3.ppoint = &digital_3;
      rd4.operations_allowed = SR_read;
      rd4.ppoint = &digital_4;
      rd5.operations_allowed = SR_read;
      rd5.ppoint = &digital_5;
      rd6.operations_allowed = SR_read;
      rd6.ppoint = &digital_6;

      ad1.operations_allowed = SR_read | SR_select | SR_operate;
      ad1.ppoint = &digital_1;
      ad2.operations_allowed = SR_read | SR_select | SR_operate;
      ad2.ppoint = &digital_2;
      ad3.operations_allowed = SR_read | SR_select | SR_operate;
      ad3.ppoint = &digital_3;
      ad4.operations_allowed = SR_read | SR_select | SR_operate;
      ad4.ppoint = &digital_4;
      ad5.operations_allowed = SR_read | SR_select | SR_operate;
      ad5.ppoint = &digital_5;
      ad6.operations_allowed = SR_read | SR_select | SR_operate;
      ad6.ppoint = &digital_6;
}

struct time daystart = { 0, 0, 8 };
struct time dayend = { 0, 0, 17 };

struct timeconstraint daytime;

struct permission_access_constraint day_only;

bitfield r_manager= 0x00000001;
bitfield r_engineer= 0x00000002;
bitfield r_technician= 0x00000004;
bitfield r_security_admin= 0x00000008;
bitfield r_vendor= 0x00000010;
bitfield r_plantman= 0x00000020;
```

```
bitfield r_operator= 0x00000040;

struct role_access_constraint op_day_only;

void buildRacsAndPacs(void)
{
      daytime.disable_start = &dayend;
      daytime.disable_stop = &daystart;
      day_only.pacType = PACALL;
      day_only.timec = &daytime;
      day_only.pacpoint = NULL;//  This PAC doesn't apply to any point right
now.

      op_day_only.roles = r_operator;
      op_day_only.timec = &daytime;

}

struct role manager;
struct role engineer;
struct role technician;
struct role security_admin;
struct role vendor;
struct role plantman;
struct role operator;  //In reality, this user might have a location based
constraint.

void getPermTables(struct role thisRole, roleID thisRID)
{
      switch(thisRID) {
            case 0x00000001:

                  thisRole.permissions[0] = ad1;
                  break;
            default:
                  //thisRole.permissions[0];
                  break;
      }

}

void buildRoles(void)
{
      manager.point_type_controls = ptype1 | ptype2 | ptype3;
      memcpy(&(manager.permissions[0]), &ad1, sizeof(struct permission));
      memcpy(&(manager.permissions[1]), &ad2, sizeof(struct permission));
      manager.numPacs = 1;
      manager.PACS = &day_only;


      engineer.point_type_controls = ptype1 | ptype2;
      //engineer.permissions = getPermTables(r_engineer);
      engineer.numPacs = 1;
      engineer.PACS = &day_only;
```

```
	technician.point_type_controls = ptype1 | ptype2;
	//technician.permissions = getPermTables(r_technician);
	technician.numPacs = 0;
	technician.PACS = NULL;

	security_admin.point_type_controls = 0x0;
	//security_admin.permissions = NULL;
	security_admin.numPacs = 0;
	security_admin.PACS = NULL;

	vendor.point_type_controls = ptype1;
	//vendor.permissions = getPermTables(r_vendor);
	vendor.numPacs = 0;
	vendor.PACS = NULL;

	plantman.point_type_controls = ptype1 & ptype2 & ptype3;
	//plantman.permissions = getPermTables(r_plantman);
	plantman.numPacs = 0;
	plantman.PACS = NULL;

	operator.point_type_controls = ptype1;
	//operator.permissions = getPermTables(r_operator);
	operator.numPacs = 0;
	operator.PACS = NULL;

}

struct role * buildRoleArray(void)
{

	//This is really, really dirty pointer arithmetic.
	struct role *aroleArray;
	aroleArray = malloc(sizeof(struct role)*7);
	memcpy(aroleArray, &manager, sizeof(struct role));
	memcpy(aroleArray+sizeof(struct role), &engineer, sizeof(struct role));
	memcpy(aroleArray+sizeof(struct role)*2, &technician, sizeof(struct
role));
	memcpy(aroleArray+sizeof(struct role)*3, &security_admin, sizeof(struct
role));
	memcpy(aroleArray+sizeof(struct role)*4, &vendor, sizeof(struct role));
	memcpy(aroleArray+sizeof(struct role)*5, &plantman, sizeof(struct
role));
	memcpy(aroleArray+sizeof(struct role)*6, &operator, sizeof(struct
role));

	return aroleArray;
}

bitfield * roleBFAbuilder(void)
{

	bitfield *aroleBFA;
	aroleBFA = malloc(sizeof(bitfield)*7);
	aroleBFA[0] = r_manager;
	aroleBFA[1] = r_engineer;
```

```
	aroleBFA[2] = r_technician;
	aroleBFA[3] = r_security_admin;
	aroleBFA[4] = r_vendor;
	aroleBFA[5] = r_plantman;
	aroleBFA[6] = r_operator;

	return aroleBFA;

}

uint32_t secret1[4] = {0x12341234, 0xabcdabcd, 0xa1a1a1a1, 0xdeadbeef};

#define NUMUSERS 5

struct user joe;
struct user jim;
struct user john;
struct user josh;
struct user jeb;

void buildUsers(void)
{

	joe.myId = 0x00;
	joe.secret = secret1;
	joe.roles = r_manager;
	joe.numRacs = 0;
	joe.RACS = NULL;

	jim.myId = 0x01;
	jim.secret = secret1;
	jim.roles = r_engineer | r_plantman;
	jim.numRacs = 0;
	jim.RACS = NULL;

	john.myId = 0x02;
	john.secret = secret1;
	john.roles = r_vendor;
	john.numRacs = 0;
	john.RACS = NULL;

	josh.myId = 0x03;
	josh.secret = secret1;
	josh.roles = r_operator;
	josh.numRacs = 1;
	josh.RACS = &op_day_only;

	jeb.myId = 0x04;
	jeb.secret = secret1;
	jeb.roles = r_security_admin;
	jeb.numRacs = 0;
	jeb.RACS = NULL;

}
```

```
struct user * buildUserTable(void)
{
      struct user *auserTable;
      auserTable = malloc(sizeof(struct user)*5);
      *(auserTable) = joe;
      *(auserTable+1) = jim;
      *(auserTable+2) = john;
      *(auserTable+3) = josh;
      *(auserTable+4) = jeb;

      return auserTable;
}
```

Appendix A.3 – security.c

This includes all the security cell code, including hashing functionality, RBAC, and others.

```
//  POSIX libray's we need for now printf
#include <stdio.h>
// we will need mallloc, which is in stdlib
#include <stdlib.h>
// Now OKL4 (lib) stuff
#include <okl4/init.h>
#include <okl4/env.h>
#include <okl4/env_types.h>
#include <okl4/virtmem_pool.h>
#include <okl4/physmem_segpool.h>
#include <okl4/kspace.h>
#include <okl4/kthread.h>
#include <okl4/kclist.h>
#include <okl4/memsec.h>
#include <okl4/static_memsec.h>
#include <okl4/zone.h>
#include <okl4/pd.h>
#include <okl4/irqset.h>
#include <okl4/message.h>

#include <hwmanclient.h>
#include <ser2client.h>
#include <usleep.h>

// Now any L4 system call headers
#include <l4/ipc.h>
#include <l4/kdebug.h>
#include <l4/misc.h>  // for L4_KDB_Enter()

#include <serial/serial.h>


#include "../include/rbac.h"
#include "../include/srole.h"
#include "../include/sha2.h"
```

```
#include "../include/random.h"
#include "../include/security.h"

//Define this to show off the iterations of the checkpermissions function
#define SHOWOFF


struct queuedAuth q1;
okl4_kcap_t *io;

struct point *pointArray;
struct role *roleArray;
struct user *userTable;
bitfield *roleBFA;


struct user NullUser = { 0x00, 0x00 };

struct user * findUser(userID thisUser)
{
	int i;

	for(i = 0; i < NUMUSERS; i++)
	{

		if(userTable[i].myId == thisUser)
		{

			return &userTable[i];
		}

	}
	return &NullUser;
}

//Checks if a given pointTypeControl bitfield may access a given point.
int checkPoint(pointID inputPoint, bitfield pointTypeControls)
{
	int i;

	for(i = 0; i < NUMPOINTS; i++)
	{
		if((pointArray[i].ppt & pointTypeControls) > 0)
		{
			//Access Granted
			return 1;
		}
	}

	//No accesses found
	return 0;

}

//Check if a given permission can access a given operation
```

```
int checkPerm(struct permission inputPerm, uint8_t operation)
{
      if((inputPerm.operations_allowed & operation) > 0)
            return 1;

      return 0;
}

//These are just stubs.  Ordinarily, these should check for time of day,
//And other conditions.
int checkRAC(struct role_access_constraint inRAC, bitfield inputRoles)
{
      if((inRAC.roles & inputRoles) > 0)
            return 0;

      return 0;
}

//Just a stub.  These fuctions should check for time of day, and other
//conditions
int checkPAC(struct permission_access_constraint inPAC, pointID inputPoint)
{
      if((inPAC.pacpoint)->pointid == inputPoint)
            return 0;

      return 0;

}

int checkAllowed(userID thisUser, pointID inputPoint, uint8_t operation)
{

      int i,j;
      int rackEmUp = 0;
      int packEmUp = 0;
      struct user *myUser = findUser(thisUser-1);

      //If user Returning failed, return failed permissions.
      if(myUser == &NullUser)
      {
            return 0;
      }


      for(i = 0; i < NUMROLES; i++)
      {
            #ifdef SHOWOFF
            lprintf("Iterating Through Roles\n");
            lprintf("My Roles: %lx RoleBFA: %lx\n", (*myUser).roles, roleBFA[i]);
            #endif
            if(((*myUser).roles & roleBFA[i]) > 0)
            {     //Check role access to point
                  #ifdef SHOWOFF
                  lprintf("User Has Access to Some Role.\n");
                  #endif
```

```
                  if(checkPoint(inputPoint, roleArray[i].point_type_controls))
                  {     //Check role access to operation
                        #ifdef SHOWOFF
                        lprintf("Role can perform given operation\n");
                        lprintf("Start of role: %p\n", (roleArray+i));
                        #endif
                        if(checkPerm((roleArray[i]).permissions[inputPoint-1],
0x1<<operation))
                        {
                              #ifdef SHOWOFF
                              lprintf("Role has permission to given point\n");
                              #endif
                              for(j = 0; j < roleArray[i].numPacs; j++)
                              {
                                    #ifdef SHOWOFF
                                    lprintf("Iterating PACs\n");
                                    #endif
                                    if(checkPAC(roleArray[i].PACS[j], inputPoint))
                                          rackEmUp++;
                              }
                              if(rackEmUp == 0)
                              {
                                    #ifdef SHOWOFF
                                    lprintf("Iterating RACs\n");
                                    #endif
                                    for(j=0; j < (*myUser).numRacs; j++)
                                    {
                                          if(checkRAC((*myUser).RACS[i], (*myUser)
      .roles))
                                                packEmUp++;
                                    }
                                    if(packEmUp == 0)
                                    {
                                          #ifdef SHOWOFF
                                          lprintf("Returning True\n");
                                          #endif
                                          return 1;
                                    }
                              }
                        }
                  }
            }
      }

      //if we get here without returning, all permission checks must have failed.
      #ifdef SHOWOFF
      lprintf("Returning False\n");
      #endif
      return 0;


}
```

```
void initDataStructures(void)
{
	buildPermissions();
	buildRacsAndPacs();
	buildRoles();
	buildUsers();
	pointArray = getArrayOfPoints();
	roleArray = buildRoleArray();
	userTable = buildUserTable();
	roleBFA =  roleBFAbuilder();

}

okl4_word_t temp_demandResponse(uint8_t *responseData)
{
	uint8_t snonce[4];
	//okl4_word_t replyPack[6];
	//Generate an snonce
	uint32_t random = xorrand();
	snonce[0] = random;
	snonce[1] = random >> 8;
	snonce[2] = random >> 16;
	snonce[3] = random >> 24;


	//Generate a reply packet
	responseData[1] = 0x03;
	responseData[0] = 0x00; //Ignoring user for now.
	responseData[2] = snonce[0];
	responseData[3] = snonce[1];
	responseData[4] = snonce[2];
	responseData[5] = snonce[3];

	//sendToNetLayer(replyPack);

	//Add to currently queued auth attempts
	q1.queuedId = 0x00; //ignoring user for now
	q1.snonce[0] = snonce[0];
	q1.snonce[1] = snonce[1];
	q1.snonce[2] = snonce[2];
	q1.snonce[3] = snonce[3];
	//q1.thisOp = &myOp;

	return 0;

}

okl4_word_t temp_isCriticalOp(struct queuedOp myOp)
{

	//Check for criticality in here
	//if is critical
		//return 1
	//else
		//return 0
```

```
	//Right now, point  2 is the only critical point
	if(myOp.qPoint == 0x02)
		return 1;
	else
		return 0;

}

void temp_actOp(struct queuedOp myOp, uint8_t *downBuffer)
{
	uint8_t *returnData = malloc(2);

	lprintf("Doing Something: Operation: %x Point: %x User: %d\n",
myOp.operation, myOp.qPoint, myOp.qId);
	//Check if we're allowed to do this operation
	if(checkAllowed(myOp.qId, myOp.qPoint, myOp.operation))
	//if(1)
	{	//if yes, send up to i2c cell
		temp_sendToIO(myOp.operation, myOp.qPoint, myOp.data, returnData);
		if(returnData[0] == 0x05)
		{
			//Generate a reply packet
			downBuffer[1] = 0x05;
			downBuffer[2] = myOp.qId;
			downBuffer[3] = returnData[1];

		} else {
			downBuffer [1] = 0xFF;
		}
	}
	free(returnData);

}

void temp_sendToIO(uint8_t operation, uint8_t point, uint8_t data, uint8_t
*returnData)
{

	uint8_t message[3];
	uint8_t returnedData[MAX_CHARS];
	int error;

	switch(operation)
	{
		//read operation
		case 0x00:
			message[0] = 0x03;
			message[1] = point;
			error = okl4_message_call(*io, message, 2*sizeof(uint8_t),
returnedData, 4*sizeof(char), NULL);
			//L4_KDB_Enter("Security: Out of Message Call");
			returnData[0] = 0x05;
			returnData[1] = returnedData[1];
			lprintf("Got some Data: %x\n", returnedData[1]);
			break;
```

```
            //select operation
            case 0x01:
            //operate operation
            case 0x02:

                  message[0] = 0x04;
                  message[1] = point;
                  message[2] = data;
                  error = okl4_message_call(*io, message, 3*sizeof(uint8_t),
returnedData, 4*sizeof(char), NULL);
                  //L4_KDB_Enter("Security: OUt!");
                  returnData[0] = 0xFF;
                  break;

      }

}


int tempreceiveOperation(void)
{
      int error;
      okl4_word_t bytes;
      char buffer[MAX_CHARS];
      okl4_kcap_t client;

      uint8_t opType;
      uint8_t incomingID;
      //uint8_t *payload; //Oversized buffer for speed, could malloc.
      uint8_t *payload;
      //uint8_t downBuffer[6];

      //Check Number of Bytes
      error = okl4_message_wait(buffer, MAX_CHARS, &bytes, &client);
      assert(!error);


      //Parse Message into parts
      opType = buffer[1];
      incomingID = buffer[0];

      payload = (uint8_t*)buffer+2;

      //lprintf("opType: %x\n", (int)client);
      //Action Determination Looop
      //TODO: Enum or #define the opTypes.

      //Store the operation on our 1 deep queue.
      //If the queue ever gets deeper, we'll need to move this code!
      //Storing the operation doesn't make sense, but we're trying to
      //Get something working!


      uint8_t *downBuffer = malloc(6);
      if(opType != 0x04)
```

```
	{
		struct queuedOp *myOpStorage = malloc(sizeof(struct queuedOp));
		(*myOpStorage).qId = incomingID;
		(*myOpStorage).operation = opType;
		(*myOpStorage).qPoint = payload[0];
		(*myOpStorage).data = payload[1];
		q1.thisOp = myOpStorage;
		if(temp_isCriticalOp(*myOpStorage)) //Op Is critical
		{
			lprintf("Demanding a Response!\n");
			temp_demandResponse(downBuffer);

		} else {
			lprintf("Acting on the Operation!\n");
			temp_actOp(*myOpStorage, downBuffer);
		}


	} else {
		//If incoming is a challenge response, let the
		//Check Response routine handle doing stuff.
		temp_checkResponse(incomingID, payload, downBuffer);
	}
	//No matter what, when any of these functions exit, they will have
	//Given us some valid downbuffer data.
	error = okl4_message_reply(client, downBuffer, 6);
	lprintf("I just sent a reply\n");
	free(downBuffer);


	return 0;
}

okl4_word_t temp_checkResponse(userID authingId, uint8_t *payload, uint8_t
*downBuffer)
{
	uint8_t cnonce[4];
	uint8_t chash[32];

	int i;

	cnonce[0] = payload[0];
	cnonce[1] = payload[1];
	cnonce[2] = payload[2];
	cnonce[3] = payload[3];

	for(i = 0; i<32; i++)
	{
		chash[i] = payload[4+i];
	}
	//If q1 != NULL
			//if(q1.queuedId == authingId)
			if(1) //Ignore UserID for now.
			{
```

```
                        SHA256_CTX ctx256;
                        uint8_t shash[SHA256_DIGEST_STRING_LENGTH];

                        uint8_t secret1[32] = {0x12, 0x34, 0x12, 0x34, 0xab, 0xcd,
0xab, 0xcd, 0xa1, 0xa1, 0xa1, 0xa1, 0xde, 0xad, 0xbe, 0xef};

                        SHA256_Init(&ctx256);
                        SHA256_Update(&ctx256, (okl4_u8_t *)q1.snonce, (size_t)4);
                        SHA256_Update(&ctx256, (okl4_u8_t *)cnonce, (size_t)4);
                        SHA256_Update(&ctx256, (okl4_u8_t *)secret1, (size_t)
(sizeof(uint8_t)*32));
                        SHA256_End(&ctx256, (char*)shash);
                        if(temp_hashIsEqual(shash, chash))
                        {
                              lprintf("Matched Demanded Credentials!\n");
                              temp_actOp(*q1.thisOp, downBuffer);
                        } else {
                              lprintf("Did not match demaneded credentials!\n");
                              lprintf("cnonce RX: %x%x%x%x snonce TX: %x%x%x%x\n",
cnonce[0], cnonce[1], cnonce[2], cnonce[3], q1.snonce[0], q1.snonce[1],
q1.snonce[2], q1.snonce[3]);
                              lprintf("Hash: %s\n", shash);
                              //What happens upon failure?
                        }
                  }
                  else
                  {
                        lprintf("Did not match user name\n");
                        lprintf("User Sent: %d User Predicted: %d\n", authingId,
q1.queuedId);
                  }
                  return 0;

}

okl4_word_t temp_hashIsEqual(uint8_t * shash, uint8_t * chash)
{
      int i;

      for(i=0; i<32; i++)
      {
            if(shash[i] != chash[i])
            {
                  return 0;
            }
      }
      return 1;
}

int test(void)
{
      int error;

      okl4_word_t bytes;
      char buffer[MAX_CHARS];
```

```
	okl4_kcap_t client;

	//okl4_u16_t holder;

	//char response[4];
	L4_KDB_Enter("Waiting...");
	error = okl4_message_wait(buffer, MAX_CHARS, &bytes, &client);
	assert(!error);

	//buffer = NULL;
	//bytes = NULL;
	//client = NULL;

	return 1;

}


/*
*  start the main program
*  much of this can be redone in functions
*  and #define and MACROs
*  but for now leave everything in
*  main so we can easily see what is going on
*/
int main(int argc, char** argv)
{

	okl4_word_t bytes;
	uint8_t buffer[MAX_CHARS];
	okl4_kcap_t *client = NULL;

	//Random Testing
	//okl4_word_t randomnum;

	//SHA testing
	SHA256_CTX ctx256;
	unsigned char buf[64] = "test";
	char output[SHA256_DIGEST_STRING_LENGTH];

	okl4_init_thread();

	io = okl4_env_get("IO_CAP");
	assert(io !=NULL);

	L4_KDB_SetThreadName(L4_Myself(), "Security");

	SHA256_Init(&ctx256);
	SHA256_Update(&ctx256, (okl4_u8_t *)buf, (size_t)4);
	SHA256_Update(&ctx256, (okl4_u8_t *)buf, (size_t)4);
	SHA256_End(&ctx256, output);

	setCellName("security");
	ser2client_init();
```

```
	lprintf("Test\r\n");

	lprintf("Some Random Numbers: %u %u %u %u", xorrand(), xorrand(), xorrand(),
xorrand());

	lprintf("SHA256: \r\n%s\r\n", output);

	lprintf("Some Random Numbers: %u %u %u %u\r\n", xorrand(), xorrand(),
xorrand(), xorrand());

	initDataStructures();

	//lprintf("Should Be 1: %d\r\n", checkAllowed(0x00, 0x01, SR_read))    ;

	//L4_KDB_Enter("Waiting in main for no reason!");

	while(1)
	{
		tempreceiveOperation();
	}


	L4_KDB_Enter("security done");
}
```