Final Report: Implementing OpenPLCs into a Cyber Defense Competition

Team 16

Dr. Doug Jacobson
Dr. Julie Rursch

Nick Springer - Security Engineer
Matthew McGill - Project Manager
Val Chapman - Software Testing Engineer
Josh Przybyszewski - Software Engineer
Joseph Young - Software Engineer
Liam Briggs - Hardware Engineer
Brennen Ferguson - Hardware Engineer

sdmay18-16@iastate.edu
http://sdmay18-16.sd.ece.iastate.edu

Contents

1 Project Design

1.1 Project Statement

This project's task is to explore the OpenPLC project and determine how it can be implemented into a Cyber Defense Competition (CDC), with the intention of simulating a real-world cyber physical environment. Up to this point, the CDC has utilized vulnerable web applications and scenarios to simulate network interactions. These scenarios have enabled students to learn and grasp security-related concepts in incredible ways, but they are limited in their ability to simulate physical equipment. By incorporating programmable logic controllers, which are computers that have been adapted to control manufacturing processes, the competition will provide more realistic and practical scenarios that will prepare competitors for interactions with this industry standard equipment. Additionally, adding support for programmable logic controllers into the competition environment provides a enormous amount of new opportunities for encouraging sponsor interaction. Learning to protect virtual environments is absolutely essential in the world of security, but security sometimes extends beyond the virtual world; this is the improvement our project shall bring to the CDC.

1.2 Project Revision

While minor aspects of the project have faced revision over the course of second semester, the ultimate goals and project deliverables have not deviated from the project outlined above. Instead of using physical programmable logic controllers to control, manipulate, and run real machinery, a soft PLC has instead been used to provide the same functionality in a simulated environment, via the software Factory I/O. The discovery and use of Factory I/O has really turned this project into a scalable and practical solution to the initial problem. Instead of having to purchase expensive equipment and scale the hardware based on how many CDC teams participate, the same software product can be used from semester to semester, simulating completely different environments each time and easily scaling to any teams wishing to join last minute. The flexibility and scalability that Factory I/O offers has made a significant impact in our project while still allowing the retention of our initial goals and ultimate deliverables.

2 Implementation

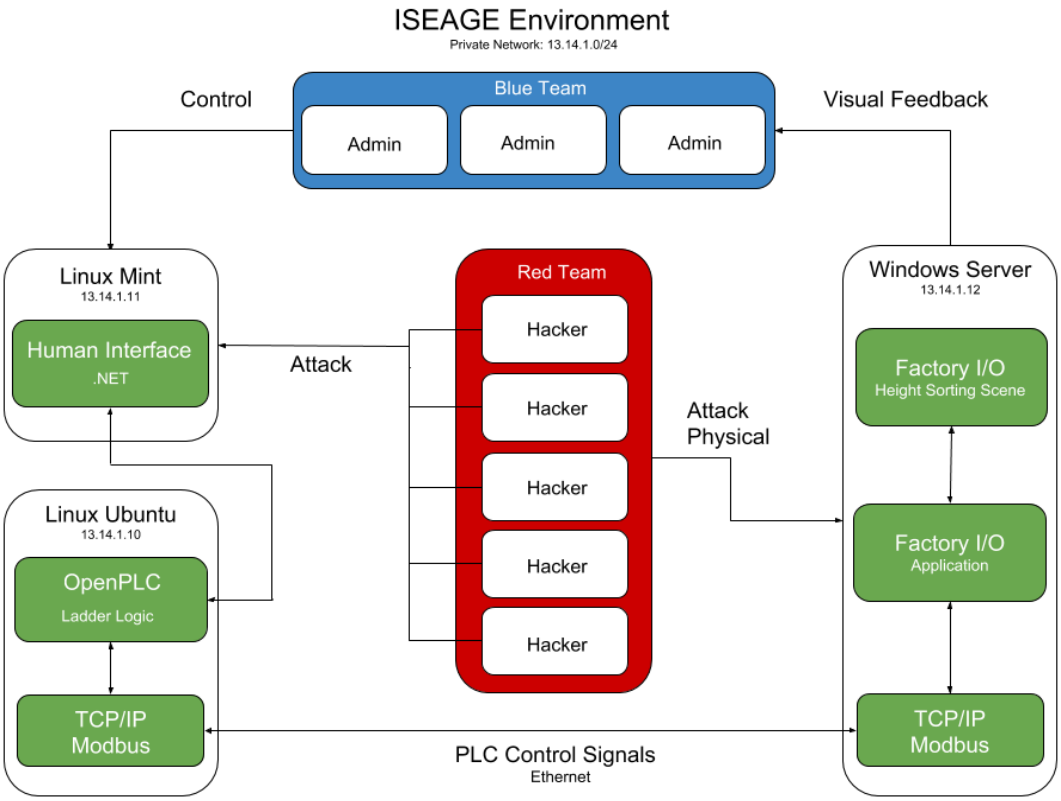
2.1 Software

There were multiple software products and technologies that were used in this project. Factory I/O was a core application that is fundamental to the project's design. Factory I/O simulates the complex interaction of factory floor machinery in a completely virtualized environment. A .NET web-API application (with an Angular Dart front-end) was developed under the Factory I/O SDK that allowed for the reading and transmission of PLC signals. This application, or admin console, served as the middleman between the soft PLC and the ongoing simulation. It was critical to the overall project, and allows for the interjection of false signal values, modeling the impact one faulty control signal can have on factory machinery. Although already mentioned

above, the open-source OpenPLC project acted as the core foundation to the Factory I/O machines, providing the logic necessary for them to run and function as normal.

2.2 Network Architecture

Our network architecture is centered around three different virtual machines. The operating systems being used on the virtual machines are Ubuntu, Linux Mint, and Windows Server. All of the machines can communicate with each other, but the Windows Server is the only machine that currently has a need to communicate with the Linux machines. It's not pertinent for the Linux machines to communicate. Below is a diagram that displays the servers and how the red and blue teams can connect with them.



3 Testing

To see if OpenPLC can be implemented into a CDC while using Factory I/O, we decided to take the testing strategy of simulating setting up the environment, measuring the resources to run a CDC, and measuring the system set up time for using these devices.

First, we decided to set up a mock CDC scenario. This testing stage showed that the systems could be connected on the ISEAGE (Internet-Scale Event and Attack Generation Environment) servers, run Factory I/O on the alloted machines, and be controlled via a web

interface. We set up a scenario where a shipping company is requesting a security check on their systems before the factories go live. The users are able to send commands to the factoring using the front end web services, that the CDC teams will finalize.

Due to the CDC's limited server resources found in ISEAGE, we needed to ensure that the system could be run and scaled to the size of an Iowa State CDC. On average, there are twenty teams participating in a CDC each event, and those twenty teams will create virtual machines to secure their environments on the ISEAGE servers. To measure the resources required to run a CDC, we set up what we believe to be a typical environment for a single team. The resources required to run the system are a windows machine to run Factory I/O and its connectors, a Linux Mint machine to host the web server, and a Linux Ubuntu machine hosting OpenPLC. The total machines required to run the scenario is three. We then assumed that an average team will start up two more machines for filtering out traffic and other services. This totals to roughly five machines per team.

Finally, we needed to test the time to set up a CDC, to ensure that it was practical to use Factory I/O and OpenPLC in each CDC. One of our larger concerns for using Factory I/O was the ability to design and implement Factory I/O into a CDC within the given time frame per competition. The setup time alloted for designing and implementing a CDC is roughly two months. From our experience in setting up our mock CDC, a scenario could be built and developed in two months, continuing off of the progress and ground work we have set. We feel that this is reasonable for the CDC, as it is within the time restraints and would allow for extra time to learn the services and to create deeper and more intricate vulnerabilities for the system.

4 Previous Work

Figure 1

Of all the Cyber Defense competitions hosted by Iowa State, only two of have involved physical aspects. Both of these were using the models shown in Figure 1. This project is a continuation and expansion of the previous implementation using 3D printed cities. These cities only provided output in the form of LEDs to denote whether the system has been penetrated or not. While not existing in physical space, our take on the project virtualizes the physical aspect while still providing real world physics and a the ability to view consequences in a physical way.

The team has been unable to find any previous projects that would provide a foundation for the project. As a result, the project has been built off the basics of the OpenPLC and Factory I/O softwares.

5 Appendix I

1. Creating Virtual Machines
    a. Windows Server Machine Suggested Specs
        i. Windows Server 2016 (evaluation edition)
        ii. 4 core CPU @ 2 Ghz
        iii. 4 Gb Memory
        iv. 40 Gb Hard drive Storage
    b. Ubuntu Server Suggested Specs
        i. Ubuntu Server 17
        ii. 2 core CPU @ 2 Ghz
        iii. 2 Gb Memory
        iv. 40 Gb Hard drive Storage
    c. Mint Server Suggested Specs
        i. Linux Mint Cinnamon
        ii. 2 core CPU @ 2 Ghz
        iii. 2 Gb Memory
        iv. 40 Gb Hard drive Storage

2. Software Installation
    a. OpenPLC
        i. Clone the OpenPLC Github repository ("git clone https://github.com/thiagoralves/OpenPLC.git")
        ii. Move into the OpenPLC directory ("cd OpenPLC")
        iii. Build the application ("./build.sh")
        iv. Start the service ("sudo node server.js")
    b. Factory I/O
        i. Download and install on Windows Server
        ii. Move provided environment file to Factory/IO (C:\Program Files (x86)\Real Games\Factory IO\Scenes)
        iii. Open the application (desktop icon)
        iv. Set quality to 'Low' (File > Video > Quality)
        v. Open prepared environment file (File > Open > My Scenes > Distribution)
        vi. Select PLC (File > Drivers > None > Modbus TCP/IP Client > Connect)
    c. .NET Web API Application
        i. Download and install the latest version of Visual Studio Community Edition onto the Windows Server
        ii. Connect an Iowa State student account for access to the Enterprise version (plus lifetime student access).
        iii. Set up and configure a new Web API project from template.
        iv. Factory I/O SDK runs on top of Engine I/O, so ensure the proper project files and dependencies are added to the project.
    d. AngularDart
        i. Download and install the latest version of the software (development environment) to the Linux Mint server.

ii. Download a text editor, also to the Linux Mint server, to develop and write the front end of the application. This can also be written outside of ISEAGE, and then Git can be used to ensure the same codebase is present in multiple locations

e. PLCopen Editor
   i. Download to desired machine (does not need to be used on the VM's)
   ii. OS specific instructions of execution can be found below the PLCopen Editor download links in the provided PLCopen Editor link above

3. Software Usage
   a. OpenPLC
      i. After starting the OpenPLC service, the only interaction will take place through the services web page found at http://x.x.x.x:8080, where x.x.x.x is the IP address of the Ubuntu machine and 8080 is the port number of the server.js file (this can be changed within the server.js file)
      ii. From this webpage, the user is able to run, stop, check logs, and upload ladder logic files using the labelled buttons
   b. Factory I/O
      i. This link directs to the Factory I/O documentation for very basic interaction within the factory environment
      ii. The other applications interacting with Factory I/O depend on the provided environment running and will not act as expected otherwise
   c. .NET
      i. Develop Web API project, using code examples from the Factory I/O SDK projects as guides. Ensure proper Engine I/O dependencies have been added and are being used
      ii. In testing the Web API project, build and run the specific project. Ensure Factory I/O is running in the background
      iii. Navigate the web browser to localhost:51721 to ensure the project built successfully and is running
      iv. Use Postman to test API endpoints, as defined in the API controller classes
         1. Practice sending and receiving data
   d. AngularDart
      i. Write and develop the front-end user interface. Code examples can be found here
      ii. Open a terminal session and type "pub serve". This will compile the application, and cross compile Dart code to native Javascript that the web browser will interpret.
      iii. Configurations: make sure you modify the code to hit specific endpoints, as defined in the Web API project. A system-wide proxy will have to be configured, and certain http headers will have to be added, in order for requests to be returned. Data is returned in JSON format, so ensure the front-end code is able to read this information and respond appropriately.

6 Appendix II

6.1 Previous Physical PLC Based Ideas

When designing this project, the team discussed many different real world physical ideas. The first decided idea relied on a physical environment including the use of Raspberry Pi's (a "credit-card-sized computer"), Arduino's (similar to Raspberry Pi's but with more possibilities of power connections), and a collection of sensors and actuators. The Raspberry Pi would run OpenPLC. Connected to the Raspberry Pi would be the Arduino, used to drive the output signals from the OpenPLC software to the sensors. Lastly, the Arduino would be connected to a collection of sensors and actuators which would be arranged into an elevator for marbles. The elevator would move the marbles from one level to another dependent upon color or size. If the PLC were compromised, the elevator would deliver the marbles the wrong level, or perhaps not deliver them at all. This elevator would have been quite large and made out of wood and plexiglass in the form of a maze.

6.2 Software Development Platforms

Originally, our human interface web application was intended to be written using Angular Dart. However, we soon discovered that the SDK used by Factory I/O is written using the .NET framework. Initially, we thought this meant that we were limited to developing our application as a single entity within the Windows Server. We attempted development of a .NET only application, but due to developer learning time and prior group experience, we kept running into roadblocks, and progress was not being made. In the end, it was determined that both .NET and Angular Dart would be utilized. The .NET application would serve as the web application's backend, delivering Factory I/O signal values via API endpoints. The front-end of this application, written in Angular Dart, would allow for increased flexibility and an updated development environment. Our group also discovered that Factory I/O's SDK is poorly documented, forcing us to determine how it functions without a reference guide. Luckily, the SDK included some code examples that showed us how to read/write signals using very basic sensors and scenarios. We were able to use the meager examples to built out an API and deliver signal values across more complex Factory I/O environments.

7 Appendix III

7.1 Shift From Hardware-Reliant System to Software-Based Model

Originally, the team's design plan utilized hardware devices including a Raspberry Pi, Arduino, and LED light strips. With these components, we planned to create a traffic simulation model, using the Raspberry Pi as a master logic controller and the Arduino / LED strips as slave devices to provide a visualization of the model. However, we discovered that adhering to the timing requirements of the WS2812 one-wire protocol of the LED strips would be difficult.

This, along with the decision that the 'marble maze' was too resource consuming for realistic use, prompted the shift to a software based model, utilizing Factory I/O to provide a virtual factory environment. We found that this design would provide the same control and functionality as physical systems, but also provided added scalability and flexibility for developing new scenarios.

7.2 Hardware Issues / Limitations

During development, we noticed that several of the virtual machines had become incredibly sluggish or completely unresponsive. After discussion with our faculty advisor and the support team, we determined that a memory leak had possibly left these virtual machines in a faulty state. However, rebooting the servers that supported the environment normalized its behavior.

In addition, some potential issues for scalability were brought to our attention. Evidently, the ISERINK environment's resources are limited to 88 CPU cores and 72 GB of RAM shared among 11 host machines. With enough teams participating in the CDC, we were concerned that the environment may not perform well. However, after discussing these concerns with our faculty advisors, we were instructed to proceed with our design as normal, and to not allow these concerns to affect our development.