May1601 Final Report

Submitted by

Jonathan Osborne	Team Leader
Nik Kinkel	Key Concept Holder
Korbin Stich	Key Concept Holder
Daniel Borgerding	Communication Leader
Jonathan Hope	Webmaster
Aashwattch Agarwal	Communication Leader

Under the guidance of
Dr. Doug Jacobson

Prepared for
Alliant Energy

College of Engineering
Iowa State University of Science and Technology
Ames, Iowa

Spring Semester 2016

Contents

List of Figures

Section 1

Project Overview

1.1 Purpose

The purpose of this document is to act as a final deliverable for the ICS/SCADA Traffic Baseline and Honeypot project. A brief description of the project will be provided, followed by a detailed layout of the final product design, measures taken to ensure adequate testing, and guidelines concerning proper implementation of the device. Appendices will be used to deliver a step-by-step guide to device deployment and maintenance, as well as project concerns and considerations that should be noted.

1.2 Background

Supervisory Control and Data Acquisition (SCADA) systems play a vital role in maintaining and controlling critical infrastructure such as water treatment plants, oil pipelines, HVAC Systems, and the power grid. The societal importance of these services makes them a high-value target for well-funded, highly-skilled adversaries who increasingly turn to digital and electronic attack vectors, exploiting old and outdated protocols that were never designed to be exposed to the public internet. Thus, rapid intrusion detection and response is crucial for safeguarding SCADA networks.

A *honeypot* (a system designed to mimic a legitimate protocol, e.g. SSH, coerce an attacker into interacting with it, and then report attacker activity to an administrator) is an old and well-understood approach to detecting network intruders.

1.2.1 Problem

Our client, Alliant Energy, requested many small, low-maintenance honeypot systems, each capable of speaking multiple network protocols (both SCADA and non-SCADA), able to log to many different backends, and able to sniff local network traffic for anomalies, to place inside each of their 28 electrical substations.

Unfortunately, many mainstream honeypot implementations have a number of problems that make them unsuitable for deployment in Alliant's substations, including:

- **Unsafe Languages:** Many existing honeypots rely on code written in languages without memory- or type-safety, unacceptable for a high-security environment.
- **Single-Protocol:** Most honeypots are designed to speak only a single protocol.
- **Complex Deployment:** Open source honepots are generally not designed to be deployed, updated, and configured en masse.
- **Expensive Hardware:** Commercial honeypots are too expensive to place in multiple locations across many subnets.

1.2.2 Solution

We designed a honeypot *plugin framework* to enforce high-security process isolation, extreme extensibility, and easy and comprehensive testing. The system is tailored to run on Raspberry Pi consumer hardware and is cheap enough to buy in quantity, while still being powerful enough to run an Intrusion Detection System (IDS) capable of watching local network traffic for anomalies. Finally, we delivered a set of configuration management utilities that allow our client to deploy to many different locations and update all devices in a *single-step*.

1.3 Deliverables

The final deliverable of this project will be based upon the assumption that the client will be required to provide the following hardware per honeypot device deployed.

1.3.1 Hardware

- RaspberryPi Model 2 B
- 8GB MicroSD card
- 2.5A Micro USB power supply(5ft cable) with noise filter.
- USB 3.0 network adapter to RJ45 Ethernet connection.
- RaspberryPi plastic hardcase.
- Appropriate ethernet connection

1.3.2 Software

All code necessary to deploy our device can be cloned from the git repository address: `https://github.com/Senior-Design-May1601/config`.

An iso image of the full system deployed on a Raspbian operating system will also be provided, however it is highly recommended that the client deploy device via Ansible as to streamline the process and apply proper configurations per device.

Section 2

Design and Implementation

2.1 Honeypot Plugin Framework

2.1.1 Motivation

Our decision to implement a plugin framework for honeypots instead of a static, monolithic program tailored exactly to our client's current needs comes from three key observations:

1. Overall client needs evolve and change over time
2. Each substation may require a different set of honeypot protocols
3. Large programs implementing many different network protocols become very hard to reason about and test

By using a modular, extensible architecture from the beginning, we are able to react quickly to evolving client specifications as well as lay the groundwork for future, proprietary or site-specific extensions to be made.

2.1.2 Plugin Architecture

Figure **??** shows how honeypot plugins, logging modules, and the core manager fit together in a single system.

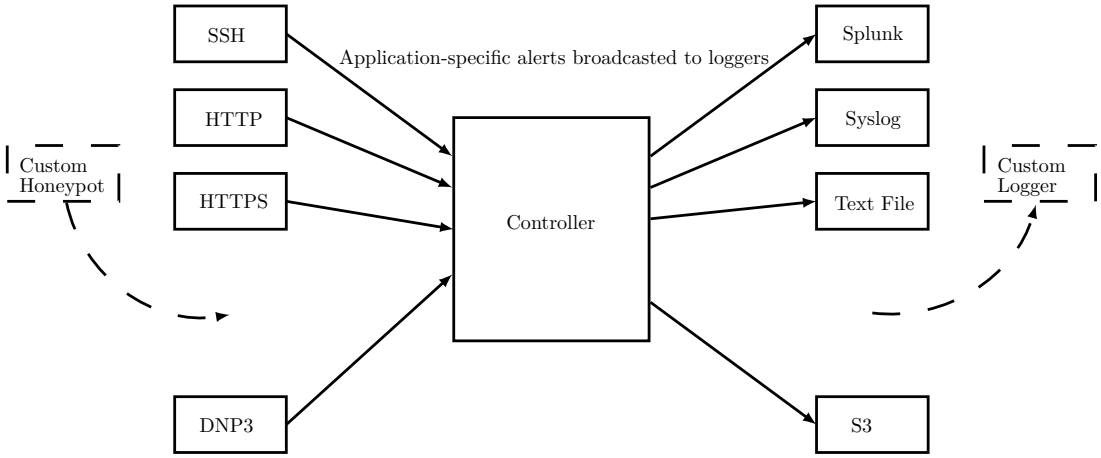

Figure 2.1: Honepot Plugin Architecture

What is a Plugin?

There are two types of recognized plugins: Honeypot Plugins and Logging Plugins.

Honeypot plugins are small, isolated programs that simply import our logging library (which uses exactly the same interface as the standard Go logging library) and log any events that should be broadcast as alerts (for instance, an attempted SSH login).

Logging plugins must implement a single-function interface that specifies the behavior that should take place when the logger receives an alert (for instance, writing the byte string to a text file).

Each plugin is responsible for implementing its own, protocol-specific functionality (for instance, an HTTPS honeypot plugin needs to do TLS protocol negotiation and handshakes as well as respond to HTTP requests on its own, and a Splunk logging plugin needs to coordinate with the remote endpoint independently), and the only interface to the rest of the system is through the imported logging module for honeypot plugins and the single-function interface for logging plugins.

Plugin Management

The plugin manager runs as a system daemon and is responsible for starting, stopping, and managing all plugins as child subprocesses. The plugin manager receives alerts from honeypot plugins and broadcasts them to each logging plugin. The plugin manager also centralizes error messages and reporting and provides a single interface to configure the system.

Plugin Communication

Plugins communicate via an asynchronous message-passing protocol which uses Go's RPC library under the hood. When honeypot plugins call an imported logging function, an asynchronous RPC call is made behind the scenes to the plugin manager, using a connection that is setup and initiated when the logger is created.

Logger plugins also receive messages over via the same message-passing protocol, and the interface they implement is actually the method called via asynchronous RPC.

The Default Plugin Set

Along with the plugin architecture itself, our deliverable includes a set of default plugins tailored to our client's current needs. This set includes the honeypot plugins for the following protocols:

- **SSH**: the SSH plugin simulates and SSH protocol information and logs metadata about the attempted login
- **HTTP**: the HTTP plugin presents a fake login page and harvests attacker credential
- **HTTPS**: similar in practice to the HTTP plugin, the HTTPS plugin adds TLS to the fake login page
- **DNP3**: the DNP3 plugin collects DNP3 SCADA commands and logs attempted attacker activity

The default plugin set also includes the following logging plugins:

- **Splunk**: the Splunk logger logs alerts to a remote Splunk instance using the Splunk Event Collector API
- **TextFile**: the text file logger logs alerts to a local text file
- **Syslog**: the Syslog logger logs syslog alerts on the device

2.2 Supporting System Design

The supporting system is essentially every device component that is not part of the honeypot plugin framework. For instance: the operating system itself, particular critical software components (e.g. iptables), and the intrusion detection system. Figure 2.2 shows how system components fit together.

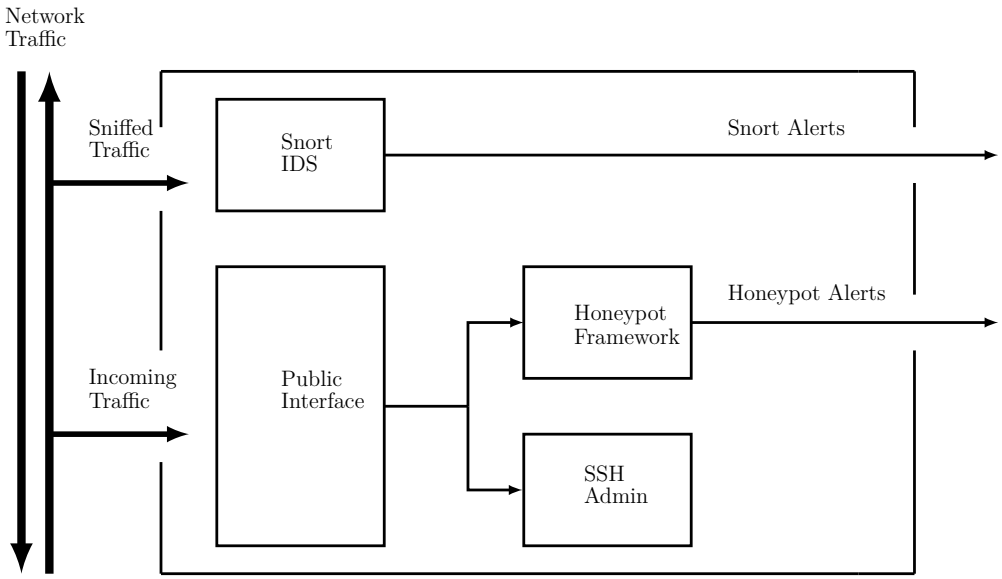

Figure 2.2: System Components

2.2.1 Design Considerations

The supporting system is designed to accomodate our client's particular needs as well as minimize the amount of ongoing maintenance and system-specific work that must be performed.

We use the Raspbian operating system, designed for Raspberry Pi's, as the base system. All other components are designed to stay as close to the upstream packages as possible (minimize or eliminate custom modifications).

2.2.2 System Components

The supporting system has the following critical components, described in more detail in the Implementation Section:

1. Firewall
2. (Real) SSH server for ongoing maintenance
3. Intrusion Detection System

Section 3

Implementation

3.1 Honeypot Plugin Framework

3.1.1 Plugin Manager

The honeypot plugin manager is implemneted using the Go programming language. Communication with plugins is done via the Go standard library's RPC module.

3.1.2 Honeypot Plugins

Each honeypot plugin is also implemented in the Go programming language.

SSH Plugin

The SSH plugin uses Go's `crypto/ssh` package to simulate legitimate SSH protocol handshakes with two custom callbacks that log alerts when an attacker attempts either a password-based login or a key-based login.

HTTP Plugin

The HTTP plugin uses Go's `net/http` package to implement a standard HTTP server, along with Go's `html/template` package to provide site- and device-specific template configuration for the fake login page.

HTTPS Plugin

Along with the packages used for the HTTP plugin, the HTTPS plugin also uses Go's `http.ListenAndServeTLS` module to handle TLS protocol negotiation and operation.

DNP3 Plugin

The DNP3 plugin uses a custom implementation of DNP3 protocol parsing. In order to accomodate the many different types of DNP3 protocol messages, a SecureAuth challenge is sent in response to any incoming command message, and the challenge is always rejected. This allows a full DNP3 monitoring implementation without needing to implement any of the protocol state or controlling logic.

3.2 Supporting System

3.2.1 Configuration Management

The Ansible configuration management system is used to provide an automated means of configuring, updating, and deploying many simultaneous device installations. A set of configurable Ansible "roles" is included as part of the final deliverable that allow our client to tailor the system to their specific needs automatically.

3.2.2 Firewall

The device firewall uses the standard linux `iptables` package.

3.2.3 Administration SSH Server

The core device administration server uses the standard `OpenSSH` server implementation.

3.2.4 Intrusion Detection System

The intrusion detection system included on the device uses the industry-standard Snort IDS, along with a minimal set of initial rules tailored to the client's operating environment. Future changes and and ruleset adjustments for the IDS can be made using the Ansible configuration mangement system described above.

Section 4

Testing

4.1 Unit Testing

Unit tests are written to evaluate the performance of a specific module or function. Go programming language has support for running automated tests through it's testing package[1]. On plugins, unit tests can be written to evaluate the performance of triggered events. For instance, does the dnp3 plugin properly parse all headers available on connection? On loggers, testing functions can be used to identify any faults that occur when reporting an event. While useful for the proccess manager that regulates extensions, the efficacy of unit testing is minimal in the case of plugins and loggers because of the dependency that each have with the core. Therefore, Integration testing is the more preferred method.

4.2 Integration Testing

Integration testing was performed using Vagrant[2]. The service provides a reproducible environment for deployment and network emulation. Specific to this project, Vagrant was used to provision two virtual machines. The first virtual machine serves as an external Splunk instance which is the reporting system used by Alliant Energy. The second virtual machine executes the Ansible provisioning to download, install, configure and start all plugins and loggers. The important thing to note is that Vagrant is actually deploying the honeypot to a new Debian instance every time. Therefore, we are emulating the installation process across multiple Raspberry Pi devices. Both

[1] `https://golang.org/pkg/testing`

[2] `https://www.vagrantup.com`

virtual machines are given separate addresses on a subnet. This allows the verification of events and alerts across an actual network.

Appendix A

Operational Manual

A.1 Hardware Assembly

After purchasing the hardware, the assembly of the Raspberry Pi follows the standard vanilla installation instructions outlined by the Raspberry Pi Foundation at: `https://www.raspberrypi.org/documentation/installation/installing-images/README.md`.

The device should be connected to the local ethernet twice, once using the Pi's on board ethernet connection, and a second time using the USB network interface adapter for the IDS connection. Remember to catalog the host name(IP Address) of each device for the next phase of this manual. Also note that these devices will need network connectivity in order to download software/packages during the deployment phase or upon administering updates.

A.2 Deployment

A.2.1 Install Requirements

The machine which will deploy/configure/maintain these honeypot devices will be required to have either an OS X or Linux operating system with the following packages installed. These packages can be installed using package managers like pipor apt which are often installed by default on these operating systems.

The system should have the following minnimum software version.

- git(1.9.1)
- ansible(2.0.2)

- sshpass(1.05)

A.2.2 Clone

Once all pi devices are connected to there subsequent network environments and given local IP addresses, the installer can cd to an appropriate work directory and then use git to clone the repository: `https://github.com/Senior-Design-May1601/config`.

```
1 $ cd /home/user/ICS-Honeypot
2 $ git clone https://github.com/Senior-Design-May1601/
    config
```

A.2.3 Initialization

After cloning the repository cd into the ansible directory, `cd` to the new directory and update permissions on the `init` file:

```
1 $ cd config/ansible
2 $ chmod +x init
```

This shell script will do a number of things. First it will create 2 new directories

- **hosts:** stores ansible inventory files
- **keys:** stores deployment SSH keys

Next the script will create a series of variables that will define the honeypot devices host name/login information. the default deployment user name is deploy.Passwords for root and deploy will be generated using (NIK PLEASE CLARIFY THIS).The script will prompt the user for an administrative IP address. This address should be the one of the current system which will deploy and maintain honeypot devices. After deployment, no other systems aside from the current host machine will be able to connect to honeypot devices via SSH.

The script will also prompt the user for an administrative email address which if entered will be used by the Apticron package on each machine to

notify administration upon pertinent operating system security updates for the device. This email will not be used for general logging.

A Splunk HTTP-EVENT-COLLECTOR token value will need to be provided next, which will enable the device to log to an administrative Splunk server. Finally the script will encrypt all the data entered abaove into a secrets.yml variable file via ansible-vault which will be stored in **vars/secrets.yml**. The user should provide a password for this file which will be needed to run ansible playbooks in the future. Viewing the contents of this file or modifying it can be done by using the ansible-vault command as needed.

A.2.4 Deploy

With everything configured, the last step in deploying any number of honeypot devices is to simply run command

```
1 $ ansible-playbook -i hosts/hosts bootstrap-playbook.yml
```

This command will simultaneously bootstrap and start any number of devices at various locations by downloading necessary packages, applying system configurations, starting firewalls/IDS systems, and starting the core honeypot service. Be warned this will take some time as there are a small number of packages that need to be downloaded. Once the playbook has complete all devices will be setup as honeypot devices and will no longer require full network access until updates are required.

A.3 Plugin Configuration

Each honeypot has a simple configuration file it uses to control behavior and plugin-specific settings. A sample configuration file, with a commented explanation for each option, is included in this appendix. These sample configuration files, to be used as examples, are also included in the software deliverable.

A.3.1 Plugin Manager

```
[MasterConfig]
# full path to the local error and event log
Logfile = "/path/to/log/file"

# each honeypot plugin gets an entry like this
[[PluginConfig]]
# the name of the plugin that should appear in log
   messages
Name = "plugin"
# full path to the plugin executable
Path = "/path/to/plugin"
# arguments the plugin should be started with
Args = ["-config", "/path/to/config-file"]

# each logging plugin gets an entry like this
[[LoggerConfig]]
# the name of the logger that should appear in local
    log messages
Name = "logger"
# full path to the logger executable
Path = "/path/to/logger"
# arguments the logger should be started with
Args = ["-config", "/path/to/config-file"]
```

Figure A.1: Example plugin manager configuration file

A.3.2 Honeypot Plugins

SSH

```
# address fake SSH server should listen on
Address = "localhost"
# port fake SSH server should listen on
Port = 8022
# full path to SSH Host private key that should be
   used
Key = "/path/to/private/key"
```

Figure A.2: Example SSH plugin configuration file

HTTP/HTTPS

```
# address the HTTP/HTTPS server should listen on
Host = "localhost"
# full path to TLS cert to use
Cert = "tls/dummy_cert.pem"
# full path to TLS private key to use
Key = "tls/dummy_key.pem"
# HTTP port to listen on
HTTPPort = 8080
# HTTPS port to listen on
HTTPSPort = 8443
# full path to template used for login page
LoginTemplate = "/path/to/templates/login.html"
```

Figure A.3: Example HTTP/HTTPS plugin configuration file

DNP3

```
# address DNP3 plugin should listen on
Address = "localhost"
# port plugin should listen on for incoming DMP3 connections
Port = 20000
```

Figure A.4: Example DNP3 plugin configuration file

A.3.3 Logging Plugins

Splunk

```
# each remote logging destination should be in an Endpoint block
[[Endpoint]]
# the remote endpoint host
Host = "localhost"
# the remote endpoint event collector point
Port = 8088
# the remote endpoint event collector url
URL = "/services/collector"
# auth token to be included with each message
AuthToken = "12345"
# OPTIONAL: if included, use this set of root CAs instead of the system
#     default. useful when using self-signed TLS certificates
RootCAs = ["/path/to/ca/cert.pem"]
```

Figure A.5: Example Splunk logger configuration file

Appendix B

Alternatives

B.1 Hardware

When coming up with the initial design for Alliant's low interaction honeypot there were two viable options. The first option was to build a device using standard computer hardware. The second was to use a prefabricated single-board computer such as a Raspberry Pi. Both options are viable for creating a functional honeypot with pros and cons associated with each. A custom built device would contain more processing power, more RAM and have a high degree of customization. The drawback with a custom design is that they are more expensive and not as easily replaceable as a prefabricated device. This was the main draw towards a Raspberry Pi. These devices are relatively cheap, and offer the easy setup and installation required for deploying multiple remote machines. Ultimately the decision came down to what was the simplest solution that would allow for the use of SSH, HTTP, HTTPS, DNP3 an intrusion detection system and a means of logging. Since all of the services running on the device are minimal versions it is ultimately unnecessary to create a custom machine with extended processing power and RAM. A single board computer such as a Raspberry Pi is more than capable of accomplishing the required tasks for less than half the cost of a custom built device while also requiring less manpower to setup and maintain. For these reasons A Raspberry Pi was chosen as the platform of choice for Alliant's SCADA honeypot system.

B.2 Architecture

During the first iterations of this project, the system architecture was singled-tiered. Initially, we intended to create one monolithic application. As we

started to get further into development, we realized that many parts of this project were actually rather orthogonal to one another. At this point, a design idea was explored that pointed the project in the direction of our current plugin framework. This plugin framework became incredibly pragmatic as we continued development. The first reason this framework is ideal is because it greatly increases the security of the system. With the plugins running as separate processes, they are completely isolated from one another. This isolation also means they will be in completely different address space, which is a major benefit should there be any vulnerability or bug in any individual plugin. The other reason this framework is very fitting for this project is that it allows us to be highly extensible. With these plugins, should our client want to add some function, all they need to do is implement the main interface and they are immediately able to add plugins for new protocols or logging backends. The progression from our initial design to this plugin architecture has become a huge benefit not only to us during development and testing, but it will also be decidedly valuable for our client going forward.

Appendix C

Other Considerations

C.1 Late Additions

Since the first day of this project we knew we would eventually be handling some SCADA protocol(s). Throughout development our client was often unsure what protocols were necessary, but told us they would let us know as soon as they could. In mid-March, we finally received word that we needed to be using the DNP3 protocol. This was initially somewhat alarming as we had no previous knowledge of the protocol, and we only had a month and a half to learn about it and implement it. There were also some issues regarding specifics of the protocol as we needed to make sure we were exactly matching our client's. Luckily, with the aforementioned plugin framework, we were able to rapidly implement the requested functionality after learning the necessary pieces of the protocol. Adding significant functionality so late in the development process can be rather detrimental to a project, however we were able to complete the work requested and it is currently implemented.

The other late addition was not a technical one. Halfway through the project, we were given an additional team member. This initially seemed really beneficial as we had plenty of work that needed to get done. Luckily, Brooks' Law did not hold in our circumstance. There was plenty of trying to get our new member up to speed on not only what our project was about, but also the technical details of it. This addition was surely unexpected, but it definitely gave our team an idea of potential outcomes given personnel changes during a project.