Razorback Design Document

Nugget Registration

Version 0.1

1/12/2011

Table of Contents

[Introduction 3](#_Toc286828971)

[Customer Focus Statement 3](#_Toc286828972)

[Current Implementation 3](#_Toc286828973)

[Unique Identification 3](#_Toc286828974)

[Capabilities and Interest 3](#_Toc286828975)

[Output Nuggets 3](#_Toc286828976)

[Detection Nuggets 4](#_Toc286828977)

[Future Work 6](#_Toc286828978)

## Introduction

This document described the current implementation of the nugget registration process. Nuggets are functional intelligence components of the Razorback system. The Razorback design concept stresses a distinct division of power between nuggets and the core of the system. In the purest vision of the design, any operation that involves the analysis or manipulation of data is reserved for the nuggets.

The information provided is “as implemented” and does not reflect our end goals.

## Customer Focus Statement

The registration process, and the configuration necessary to support that process, is one of the hurdles that customers face to get a Razorback installation operational. As implemented, the registration is fairly streamlined, but lacks the functionality necessary to support a resilient, high-performance installation. Addressing these shortcomings in future releases will need to be done with an eye towards minimizing obstacles to entry- and mid-level operations groups.

## Current Implementation

### Unique Identification

* Detection nuggets must be uniquely identifiable by IP and Port

### Capabilities and Interest

* Detection nuggets must declare what data types they are interested in
* Detection nuggets must declare what application type they are
* Output nuggets are configured on the dispatcher in the configuration file
* No other nuggets require registration

### Output Nuggets

Currently the output nugget process is nothing short of a hack. Version 0.1 supports two output nuggets which receive all alert information and have the option of requesting any blob data associated with the alert. The dispatcher gets the information about these nuggets from the config file:

OUTPUT:

OUT1\_ADDR 127.0.0.1

OUT1\_PORT 10002

OUT2\_ADDR 127.0.0.1

OUT2\_PORT 10003

There is no over the network registration for output nuggets.

### Detection Nuggets

The detection nugget registration process, from the standpoint of the nugget is a two stage process. First the nugget provides a preliminary registration packet via the registerNugget API call:

static HRESULT registerNugget(const uuid\_t nugtype, const uuid\_t nugapp, const uuid\_t datatype,unsigned short nugport, unsigned instances, unsigned nugid, const char \*name, unsigned \*nuggetid)

The function creates a SNAKECHARMER packet with a REGPACKET payload. The REGPACKET structure is as follows:

typedef struct \_REGPACKET

{

unsigned nuggetid;

struct in\_addr nugaddr;

unsigned short nugport;

uuid\_t nugtype;

uuid\_t nugapp;

uuid\_t datatype;

unsigned short dummy;

unsigned freeinstances;

char name[256];

} REGPACKET;

This initial packet declares that a new nugget is up and provides the first capability tuple (nugtype, nugapp, datatype). The “freeinstances” and “dummy” fields are ignored.

Upon receiving this packet, the dispatcher goes through a series of steps, coordinated by the handleRegistration function. The dispatcher tracks the process of registration by passing-by-reference a NUGGETNTRY form. Returns from associated functions are intended to be (but aren’t) used for error notification. As an overview the following steps are taken:

1. dbRegisterNugget is called
   1. If the nuggetid for registration is not 0, dbRegisterNugget checks to see if the provided nuggetid is in use
      1. If it is not in use, the nuggetid is accepted (db update via register NuggetWithNugID)
      2. If it is in use, and the registration information matches the IP/port duple held in the database, the nuggetid is accepted (no db update)
      3. If it is in use, and the registration information does not match, the nuggetid is ignored and the declared nuggetid will be replaced with the next available nuggetid from the database. (db update via reassignUID)
   2. Otherwise, the nugget is registered in-function and the new nuggetID replaces the 0.
2. If the nuggetid provided exists in the routing table currently, all entries associate with that id are removed (remember, this is an initial registration, so this only occurs on nugget startup, therefore all existing entries are invalid).
3. If the declared nugtype is “DETECTION”, then the routing table is updated by the addNuggetEntry function.
4. Registration is complete. The response packet, with the nuggetid (either the one requested or the new one per dispatcher logic) passed as part of the response. (This logic is in the REG section of the handling loop in dispatchMaster).
5. The nugget is expected to honor the provided nuggetid. Current implementation of rzbNugget takes care of this.

Now that the initial registration is complete, the detection nugget can proceed to declare all other datatype/apptype duples that it is interested in. The API call to do this is:

static HRESULT updateNugget(const uuid\_t nugtype, const uuid\_t nugapp, const uuid\_t datatype, unsigned short nugport, unsigned nugid)

This function constructs a NEWCAPPACKET to declare the additional functionality:

typedef struct \_NEWCAPPACKET

{

unsigned nuggetid;

struct in\_addr nugaddr;

unsigned short nugport;

uuid\_t nugtype;

uuid\_t nugapp;

uuid\_t datatype;

} NEWCAPPACKET;

The dispatcher handles additional functionality in the handleNewCapability function. This function performs the following steps:

1. Checks to see if the nugget to update is already in the routing table by calling findByID. If it isn’t, the update fails (as the nugget has not previously registered).
2. The function then updates the routing table with the capability information.

The nugget continues to call updateNugget as long as it has additional capability to declare. Once it is finished, the registration process is over.

## Future Work

The registration and routing processes are expected to receive substantial overhauls during the Q2 effort. Improvements necessary, either in Q2 or in a future project are:

1. Authentication of nuggets both at registration and as they send or receive data
2. All nuggets should be required to authorize
3. Communications for registration, particularly if they involve some form of authorization, should be encrypted.
4. Encryption/authorization could be combined