**Razorback™ Framework 0.1**

**Design, Implementation and Schedule**

**Revision 2**June 2, 2010

Prepared by: Sourcefire Vulnerability Research Team  
Sourcefire, Incorporated  
9780 Patuxent Woods Drive  
Columbia, MD 21046

# Table of Contents

1 Table of Contents 2

2 Overview 3

2.1 Summary 3

2.2 Scope 3

2.3 Assumptions & Assertions 3

2.3.1 Assumption 3

2.3.2 Assertion 4

2.4 General Requirements 4

2.4.1 NRT Snort Preprocessor 4

2.4.2 NRT DLIB Process 4

2.4.3 NRT Dispatcher 4

2.4.4 Deep Inspection Nugget 5

3 Use Cases 5

3.1 Single Server Installation 5

3.2 Multiple Server Installation 5

4 Concept of Operations 6

4.1 Configuration 6

4.2 Snort Integrated Alerting 6

4.3 NRT Specific Alerting 6

5 Functional Design 6

5.1 POC Design 6

5.2 Data Model 6

5.3 External Interfaces 7

5.4 Deprecated Functionality 7

5.5 Future Enhancements 7

6 Open Issues 7

6.1 Templating Language 7

6.2 Snort Integration 7

7 Test Cases 7

7.1 POC Phase One 7

7.2 POC Phase Two 8

8 Automated Testing 8

# Overview

Razorback is the result of extensive research by various members of the Vulnerability Research Team into developing a platform to address advanced detection problems. The level of sophistication currently demonstrated both by actors described as the “Advanced Persistent Threat” (APT) and publically available exploit frameworks such as Metasploit, CANVAS and Core Impact leaves the VRT with few options to provide detection. This project is designed to provide the VRT and enterprise defense teams a framework for developing the kind of detection necessary to combat these threats.

A complicating factor in high-CPU-cost detection is the desire of customers to have low-latency analysis at wire speed. While components of the Razorback system will be able to block first-strike attacks prior to delivery, some detection solutions will cause sufficient latency as to make this impossible. One of the key points of the system is to accept that some solutions require trading real-time blocking for high-accuracy detection.

The Razorback Framework addresses these issues by providing a core infrastructure that matches declared data types to the individual capabilities of various detection systems. By providing an open, documented API, arbitrary data sources can be paired with one or more arbitrary detection systems to provide detection solutions that would otherwise be impossible due to limited data access or restriction on system resources.

## Summary

This document outlines the anticipated system architecture, use cases and design goals for the Razorback framework.

## Scope

The Razorback Framework 0.1 release will consist of:

* Dispatcher
* Database
* Data Collector
  + Snort
  + ClamAV
  + Squid
  + Postfix
* Detection
  + SMTP Parser
  + PDF Analyzer
  + Javascript Analyzer
  + ClamAV
  + Shellcode Analyzer
  + Custom Postmortem Debugger
* Output
  + Full output nugget
  + Sample data visualization
  + Maltego interface
* Defense Update
  + Snort rules update
  + ClamAV update
  + IP Chains
  + Triggered data collectors
* Correlation
  + TBD
* Workstation
  + CLI for querying status of system
* API
  + Full system API
  + API wrapped to Python
  + API wrapped to PERL
  + API wrapped to Ruby

## Assumptions & Assertions

### Assumption

The NRT system assumes that customers will tolerate a delay in alerting and a lack of active blocking (IPS) in exchange for a much more robust detection system for challenging, file-based detection issues.

### Assertion

Assertion (n):  positive statement or declaration, often without support or reason. [We don’t do this]

## General Requirements

### Dispatcher

The dispatcher is the core defense routing component of the Razorback Framework. In order to act in this role, the following capabilities must be in place:

* Provide registration services to each component
* Map registered components to declared data types
* Track tasked components to distribute detection load
* Manage inbound alerts from various components
* Store alerts and other metadata into the database
* Send alert information to registered components
* Manage requests for additional alerting information

### Database

The database is designed to be a highly scalable, extensible data storage platform. While full schema documentation is provided in Appendix A, key components of the design include:

* Incident->Event->Alert set tracking of detection events
* Tracking of hosts, URLs, IPs and Autonomous Systems and rating them with “Threat, Hazard, White list, and Watch list” flags (Threat\_Flags)
* Custom enterprise extensibility through metadata tagging and data agnostic storage
* Non-alerting data behavior storage for later correlation, forensics and behavioral analysis
* Modular subcomponent block tracking for multi-document correlation
* High performance, easily scalable solution

### Data Collection Nuggets

Each data collection nugget will need to perform the following tasks:

* Accept and maintain a persistent nugget identifier
* Register with the dispatcher and declare nugget functionality and the data-type the nugget will pass
* Maintain a local cache of known good and known bad metadata and data blocks
* Query the Dispatcher on cache misses
* Accept detection direction messages from dispatcher
* Transmit metadata and data blocks to the appropriate deep inspection nugget(s).

### Deep Inspection Nuggets

Each deep inspection nugget will need to perform the following tasks:

* Accept and maintain a persistent nugget identifier
* Register with the dispatcher and declare nugget functionality and the data-type the nugget will handle
* Provide alerting to the Dispatcher for detection that meets the minimum requirements
  + Timestamp
  + Priority
  + Message
  + Src/Dst IP
  + IP Proto
  + Short data field
* Where appropriate pass to the dispatcher the complete data block, and a normalized datablock
* Optionally, pass a verbose, arbitrary alert field (Long\_data)
* Maintain a local cache of known good and known bad metadata and data blocks
* Query the Dispatcher on cache misses
* Accept detection direction messages from dispatcher
* Transmit metadata and data blocks to the appropriate deep inspection nugget(s).

### Output Nuggets

Each output nugget will need to perform the following tasks:

* Accept and maintain a persistent nugget identifier
* Register with the dispatcher and declare nugget functionality and the alert set it will handle
* Receive transmitted alerts from dispatcher

### Defense Update Nuggets

Each defense update nugget will need to perform the following tasks:

* Accept and maintain a persistent nugget identifier
* Register with the dispatcher and declare nugget functionality and the alert set it will handle
* Receive transmitted alerts from dispatcher
* Receive transmitted update instruction from dispatcher
* Notify dispatcher of defense update actions

### Correlation Nuggets

Each correlation nugget will need to perform the following tasks:

* Accept and maintain a persistent nugget identifier
* Register with the dispatcher and declare nugget functionality
* Interact with the database directly to collect data to support correlation needs
* Provide alerting to the Dispatcher for detection that meets the minimum requirements
  + Timestamp
  + Priority
  + Message
  + Src/Dst IP
  + IP Proto
  + Short data field
* Where appropriate pass to the dispatcher the complete data block, and a normalized datablock
* Optionally, pass a verbose, arbitrary alert field (Long\_data)
* Optionally, request defense update actions through the dispatcher

### Workstation Nuggets

Each workstation nugget will need to perform the following tasks:

* Accept and maintain a persistent nugget identifier
* Register with the dispatcher and declare nugget functionality
* Authenticate to the dispatcher on a per-analyst basis
* Insert new components
* Edit notes
* Consolidate alerts and events
* Add custom data types
* Set threat and enterprise custom flags
* Delete alert entries
* Review system logs
* Arbitrary SQL queries

# Use Cases

Two use cases are discussed. First, a single server installation of the Razorback Framework and then a larger installation consisting of a full framework installation. The two approaches differ mainly in scale as opposed to functionality. To support this scaling, database APIs for the Razorback Framework will be built to support both SQL and Hadoop (<http://hadoop.apache.org/hbase/>) based installations.

## Single Server Installation

The single server installation will most likely be the common installation for open source users. This installation would include a Snort data capture/output handling component, a dispatcher and a lite SQL database. The administrator would then be able to register appropriate nuggets for that enterprise.

This installation would be appropriate for smaller organizations, remote offices or those security operations group with constrained budgets.

## Enterprise Installation

The Razorback framework is being designed with large enterprise deployments in mind. In particular the scalable nature of the nugget solution, where multiple instances and/or threads of detection are available and the extensible, performance driven design of the database are key to allowing enterprises to consider this a viable security solution.

The database in an enterprise setup would be a Hadoop core with SQL table views build to support faster data querying. The dispatcher would be a standalone system monitoring multiple data collection nuggets providing disparate data types into a multi-server detection system of multi-threaded nuggets performing advanced detection services. Output systems would be built to provide alerting information into existing SIM and alert management systems.

# Concept of Operations

Operations will consist of system initialization, collection, detection and output.

## System Initialization

The system as a whole consists of the dispatcher, the database and an arbitrary number and type of functional plugins called “nuggets”. The system cannot come to a stable, initialized state without a fully initialized dispatcher. Once the dispatcher is up, registration and nugget configuration can begin. The process would be as follows:

* Dispatcher Startup
  + Start process, read configuration file
  + From configuration data, connect to database
  + From configuration data, build cache structures
  + Using the database, prime the dispatcher cache with recent “known good” and “known bad” hits.
  + Using the database, rebuild defense routing table based on previously registered nuggets
  + Open network listeners
* Nugget Startup (Required active dispatcher)
  + Start process, read configuration file
  + From configuration data, build cache structures
  + From configuration data, open network listeners
  + From configuration data, contact dispatcher and declare:
    - Nugget Capability (data capture/detection/etc…)
    - Data Handling capability (array of GUIDs)
    - Nugget Name
    - Nugget ID (If assigned previously, otherwise pass NULL)

## Data Capture

The intelligence of the Razorback system is pushed out to the nuggets. While we do not define how to capture the data with the system, there are requirements for passing the data. Once a nugget has captured data to be submitted to the system, the following processes occur:

* Contact the dispatcher for a known good / known bad check (if local cache misses)
* In the case of a dispatcher cache miss, accept a response from the dispatcher consisting of:
  + Array of IP/Port pairs of destination nuggets to send the file to
  + An “Event\_ID” that will bind all detection together
* For each IP/port pair in the array the nugget will:
  + Open a session with the nugget
  + Pass to the nugget:
    - Event\_ID
    - Metadata blocks
    - Data block
  + Contact the dispatcher on failure for reassignment of detection

Once the data capture system has transmitted the required data, the data collection role is complete.

## Detection

Detection systems within Razorback wait with a listener for connections from the data capture nuggets. Once the nuggets receive a connection they perform the following actions:

* Contact dispatcher to check validity of Event\_ID, notify the dispatcher of successful transfer and detection start and state whether or not additional threads are available for processing
* Perform detection as determined by the nugget developer
* On completion of detection the following actions are taken:
  + Update the local cache with known good / known bad entries
  + Notify the dispatcher of detection completion
* If alerting is necessary, the following actions are taken:
  + Contact the dispatcher for each alert and transmit the following information:
    - UNIX timestamp
    - Priority
    - Message (255 bytes max)
    - Src/Dst data
    - Short block of detection specific data (2048 max)
    - Notification if any of the following are available:
      * Long block of detection specific data
      * Full data block
      * Normalized data block
      * Metadata
    - Handle dispatcher requests for optional data (described immediately above)
* If a block of data is found that the detection nugget knows that other detection systems are available to handle the following actions will be taken:
  + Contact the dispatcher for a known good / known bad check (if local cache misses)
  + In the case of a dispatcher cache miss, accept a response from the dispatcher consisting of:
    - Array of IP/Port pairs of destination nuggets to send the file to
    - An “Event\_ID” that will bind all detection together
  + For each IP/port pair in the array the nugget will:
    - Open a session with the nugget
    - Pass to the nugget:
      * Event\_ID
      * Metadata blocks
      * Data block
    - Contact the dispatcher on failure for reassignment of detection
* In case of any errors:
  + Contact dispatcher
  + Pass a known alert type with metadata describing the errors

## Output Systems

Output from the Razorback system is handled exclusively by output and defense update nuggets. These nuggets wait for connections from the dispatcher. Once connections are handled, the nuggets take the following actions:

* Handle incoming alert data in the following format:
  + UNIX timestamp
  + Priority
  + Message (255 bytes max)
  + Src/Dst data
  + Short block of detection specific data (2048 max)
  + Notification if any of the following are available:
    - Long block of detection specific data
    - Full data block
    - Normalized data block
    - Metadata
* If the output system handles additional data, or the defense update system requires additional data to determine if an update is required, the nugget requests the additional data components. (Described immediately above)
* In the case of the defense update nuggets, if actions are taken, the defense update system notifies the dispatcher via a known alert type with metadata describing the actions taken.
* In case of any errors:
  + Contact dispatcher
  + Pass a known alert type with metadata describing the errors

## Correlation

Correlation nuggets are defined by nuggets that do not handle data blocks from data collectors. Rather, they comb the data base and alerts to look for patterns that may indicate malicious activity. Correlation nuggets and workstation nuggets are the only data types that can access the database directly. The dispatcher handles this by updating ACLs during nugget registration. The logic of the correlation engine is not defined by the Razorback system. Once the logic system determines that action needs to be taken, the following actions can occur:

* In the case of an alert, the correlation takes the following actions:
  + Contact the dispatcher for each alert and transmit the following information:
    - UNIX timestamp
    - Priority
    - Message (255 bytes max)
    - Src/Dst data
    - Short block of detection specific data (2048 max)
    - Notification if any of the following are available:
      * Long block of detection specific data
      * Full data block
      * Normalized data block
      * Metadata
    - Handle dispatcher requests for optional data (described immediately above)
* In the case of a database modification, the correlation nugget takes the following actions:
  + Contact the dispatcher
  + Provide an alert in a well known format with metadata indicating the changes that will be made
  + Connect to the database
  + Make database modifications
* In the case of a detection update request, the correlation nugget takes the following action:
  + Contact the dispatcher
  + Provide an alert in a well known format with metadata indicating the changes to be made and an additional metadata block of justification

## Workstation

Workstation nuggets are the means by which end-users interact with the system. While they can review output information by way of output nuggets or nugget-specific logs, the only supported way to directly work with the system setup and data is through workstation nuggets. Workstation nuggets will be able to perform the following actions, based on authentication and authorization controls:

* Request system status
  + Determine operational status of dispatcher
  + List registered nuggets
  + Grab performance data
  + Access error logs
* Access and modify metadata information
  + For many data types, metadata is available (See functional design)
  + On a data block by data block basis, analyst can add, delete and modify metadata information
* Access and modify notebook information
  + For many data types, analyst notes are collected into notebooks (See functional design)
  + On a data block by data block basis, analyst can add, delete and modify notes
* Access alert data
  + Access alert data by alert, event or incident id
  + Request additional optional data from database (data blocks, long description)
* Manage Incidents and Events (See Functional Design)
  + Create or delete an incident
  + Create or delete an event
  + Move alerts into event blocks
  + Move event blocks into incidents
* Manual actions are also available through workstations:
  + Submit an alert
* Manage the UUID data system:
  + Create a new UUID
  + Submit UUID to Unique ID table with the following information
    - Data Name
    - Description (Optional)
* Run standardized reports (defined by Sourcefire) resulting in any combination of:
  + Text based
  + Gziped data
  + Graphs
* Provide arbitrary SQL queries into the system to build their own reports
* Access system logs

# Functional Design

The Razorback system is designed to be easily extensible and highly scalable. The extensibility is achieved by creating a database structure that handles arbitrary data. Scalability is achieved by pushing the intelligence of the gathering, detection and output operations to individual nuggets, building a high-speed, low-drag defense routing core (the dispatcher) and building the database from the start to support enterprise applications.

## The UUID Principle

UUIDs serve several essential functions within the Razorback system. The first is to provide definition to arbitrary blocks of data. All data blocks are tagged with a UUID that indicates the kind of data that exists. The dispatcher does not need to understand the data; it just has to know what kind of data it is so that it can be properly handled. This concept allows Sourcefire to ship a set of known data types (PDF, JavaScript, etc…) but also allows enterprises to easily pass data through the system without having to code any functionality into the dispatcher.

This definition of data extends beyond the standard data blocks, into the metadata blocks. Metadata provides context to the data block it is associated with. For example, a PDF file may be passed with metadata indicating that it was captured via a web transfer. This would be a well-known metadata format, which means we would provide an API with a data structure supporting functions to marshal and unmarshal this data. This API would be used by nuggets to send and parse this particular kind of metadata. Again, enterprises would be able to define their own metadata. This would be done by requesting a UUID, updating the Unique\_ID table and then building the marshalling/unmarshalling code.

By defining data by way of UUID, we allow Sourcefire to manage a centralized, well known set of known data types and also allow for enterprise customization. Because these customized data types are created and maintained by the customer, there is no impact to the system. The customer is entirely responsible for providing the functionality to handle data blocks with enterprise specific data types.

## Database Structure

The database design is included in full in Appendix A. However, there are several specific features beyond the UUID focus that should be noted. The first is the emphasis on providing contextual information about data passed on the system. This is handled by attaching a metabook ID into data structures that may require metadata information. This meta block is then referenced by all metadata entries associated with that meta block.

A host entry, for example, could potentially have several metadata entries, all of which would be tied to a metabook ID which is then associated to by the host entry. In each metadata field is a UUID entry describing what kind of data the block holds. In the case of host entries, there may be the results of NMAP or Nessus scans, user identity information or even data showing the standard Netflow behavior of the host.

Metadata should typically be created by automated systems. When humans interact with the data, their observations are stored instead in notes that are collected into notebooks. Each note is tagged with the analyst ID. In addition to text, there is the option to provide attachments into the notepad system.

## Component Blocks

Data blocks that are stored into the system are stored both in whole, as well as with the subcomponents broken out below them. In order to enhance performance and make correlation easier, subcomponents that are shared between several data blocks are stored under separate component blocks (See “Analyst’s Notes and Context Data” Appendix: A).

What this means is that a PDF file with a JavaScript block and a Flash block would be stored as three components under a single component set. The component set points to the parent data block (in this case the PDF file). If, subsequent to the creation of this component entry another PDF file is found with the same JavaScript block, a new component set will be created. The JavaScript block will be moved to the new component set (and become that component set’s parent component) and both PDF parent objects will have their component\_branch entry set to point to the new component. In this way it is very easy to show that not only did both documents alert, but they both had common data blocks that may be able to be targeted for detection.

## Network Forensic Data

In addition to raw detection, a number of high end incident management groups requested that certain data be preserved for incident analysis purposes. To support this we’ve provided pre-built web tracking, mail tracking and net flow database structures. These prebuild structures address the common requests and provide for easily indexed searches. Additionally we are providing an enterprise tracking table that will allow enterprises to build their own tracking requirements by defining a data type (UUID) and then creating a nugget to populate the enterprise tracking table. (See Network Forensics Data, Appendix: A).

## Razorback API

Because the core concepts of the Razorback system are extensibility and customization, well documented APIs in a variety of commonly used languages are required. The core API will be written in C, with a single library supporting each of the nugget types and any Razorback specific functionality (registering, alerting, etc…) that would be necessary for the proper functioning of the system. This API will then be wrapped into Perl, Python and Ruby.

Documentation, both in code and separately, will be critical for providing a comfortable environment for defense developers. Because of this, coding requirements include a documentation review prior to code acceptance, in addition to the standard functionality and security focused code review process.

# Supporting Systems

The Razorback system is scheduled for its initial public beta at DefCon. In order to provide maximum impact to the detection sector on release of this beta, several nuggets need to be prebuilt.

## Data Collection

The following data collection nuggets are being targeted for the DefCon timeframe:

* Snort (up to four custom builds)
  + SMTP mail stream capture
  + Web file capture
  + URL tracking
  + Stream data capture on arbitrary ports
* Custom post-mortem debugger
  + Traps applications as they crash
  + Inserts the file that triggered the crash to Razorback
  + Sends the metadata of the crash to the dispatcher
* Squid file capture and blocking
* Postfix file capture and blocking
* Daemonlogger
  + Packet capture triggered by output systems
  + Preconfigured packet capture
  + Packet capture for hosts marked “watchlist” in the database

## Detection

The following detection nuggets are being targeted for the DefCon timeframe::

* PDF Parser
  + Handle deobfuscation and normalization of objects
  + Potentially passing to Snort detection engine to use the detection language
* JavaScript Analyzer
  + Target known JavaScript attacks
  + Search for shellcode in unencode blocks
  + Look for heap-spray
  + Look for obvious obfuscation possibilities
* Shellcode Analyzer
  + Handle common techniques to find EIP
  + Look for code blocks that unwrap shellcode
  + Check for Windows function resolution
    - Determine the function call
    - Capture the arguments
  + Provide alerts that include shellcode action

## Output

The following output nuggets are being targeted for the DefCon timeframe:

* Deep Alerting System
  + Provide full logging output of all alerts
  + Write out each component block
  + Include normalized view of documents as well
* Maltego Interface
  + Provide data transformations targeting the Razorback database

## Defense Update

The following defense update nuggets are being targeted for the DefCon timeframe:

* Snort rules updater
* ClamAV rules updater
* IPChains updater
* Triggered updates of data collectors (Snort, Daemonlogger)

## Workstation

The following workstation nugget is being targeted for the DefCon timeframe:

* CLI functionality to query:
  + Alerts/Events/Incidents
  + Nugget Status
  + Display metadata
  + Run standardized report set

# Test Cases

The following test cases provide a general overview of how to test this feature:

## Standard Nugget Case

|  |  |
| --- | --- |
| **Description** | Register with Dispatcher |
| **Test Procedure** | * Attach to the Dispatcher * Declare nugget functionality * Declare data type handles |
| **Expected Result** | Dispatcher can display the registration status and the nugget listeners are active |

## Detection Process

|  |  |
| --- | --- |
| **Description** | Properly handle the steps from data capture to alerting. |
| **Test Procedure** | * Registered data capture nugget checks KG/KB locally and at dispatcher * Dispatcher responds with an Event\_ID and IP address/port pair. * Data capture nugget sends data block and associated metadata to the indicated nugget * Nugget notifies dispatcher of data block acceptance and detection startup. Nugget also declares if it has resources to handle additional detection requests. * Detection device determines that the data block is bad. * Detection nugget alerts to the dispatcher, including initial alert block and handling subsequent transmission of optional data * Dispatcher properly stores all alerts under a single Event\_ID * Dispatcher properly stores all data blocks in the correct component structure |
| **Expected Result** | The correct alerts and database entries are created based on the input file. |

## Output Systems

|  |  |
| --- | --- |
| **Description** | Properly send output to output nuggets. |
| **Test Procedure** | * Dispatcher sends alert data to each of the registered output nuggets in the base alert format * Dispatcher properly handles follow requests for optional alert data |
| **Expected Result** | Registered output nuggets are able to create the appropriate output reports based on alerting, including the capability to write full file and subcomponent data blocks to disk. |

## Nugget Functionality

Individual nuggets will be functionally defined and tested as part of the development process.

# Development Schedule

The following are the tentative dates for Razorback development

|  |  |
| --- | --- |
| Date | Target |
| Present | Coding begins |
| 06/01/2010 | Submission deadline, DefCon |
| 07/09/2010 | RC1 Code complete – Dispatcher, release to QA\* |
| 07/09/2010 | RC1 Code complete – C API, release to QA\* |
| 07/16/2010 | RC1 Code complete – Perl, Ruby, Python API, release to QA\* |
| 07/26/2010 | Deadline for Dispatcher, API and Database bug fixes |
| 07/26/2010 | Deadline for Nugget Coding |
| 07/27/2010 | Razorback 0.1 Final Complete and Packaged |
| 07/28/2010 | Razorback 0.1 and Release Nuggets pre-staged on http://labs.snort.org |
| 07/31/2010 | Razorback 0.1 Release |

\* External QA evaluation dependent on availability of QA resources and is not a prerequisite to 0.1 release

# Future Development

These potential development systems may be included in the DefCon release if they pass code review in time. Otherwise they are forward-looking features we would like to include:

* Netflow data collection nugget
* Low interaction Honeypot detection/data collection nugget
* Sandbox behavioral analysis detection nugget
* Data visualization output nugget
* GUI workstation nugget
* Correlation nugget
* Automatic nugget download from dispatcher
* Nugget load balancing
* High availability setup for dispatcher
* Standardized template language for data blocks
* Standardized rules language built against the template language
* Netranome integration
* Ability to push cache updates to nuggets on registration and after alerts
* Automated IP analysis nugget that include NMAP/Nessus parsing and whois parsing to fill out host, IP block and AS tables.
* Callback capability
* Auto update from Sourcefire provided data set to tag IP/IP blocks/AS/URLs as “threat” or “watchlist”
* Pile of awesome nuggets

# Appendix A: Database Schema













# Appendix B: System Overview







# Appendix C: Specific Razorback Actions

 

