Overview

What is Razorback?

Razorback Is…

- A framework to enable advanced processing of data and detection of events
- Able to get data as it traverses the network
- Able to get data after it's received by a server
- Able to perform advanced event correlation

- …Our answer to an evolving threat landscape

The Challenge is Different

- Attacks have switched from server attacks to client attacks
- Common attack vectors are easily obfuscated
  - JavaScript
  - Compression
- File formats are made by insane people
- Back-channel systems are increasingly difficult to detect

The Problem With Real-Time

- Inline systems must emulate the processing of thousands of desktops
- Detection of many backchannels is most successful with statistical evaluation of network traffic

Coverage Gap

- Broadly speaking, IDS systems deal with packet-by-packet inspection with some level of reassembly
- Broadly speaking, AV systems typically target indicators of known bad files or system states

Fill the Gap

- A system is needed that can handle varied detection needs
- A system is needed that extensible, open and scalable
- A system is needed that facilitates incident response, not just triggers it

Framework Goals

- Provide entry to the system for any arbitrary data type
- Determine and manage detection based on a registered detection nugget
- Provide alerting to any framework-capable system
- Provide verbose, detailed logging on the findings of the nugget "farm"
- Make intelligent use of all data discovered during the evaluation process

Architecture

What makes it tick?

Razorback is comprised of…

- Dispatcher
- Database
- Various nugget types:
  - Data Collection
  - Data Detection/Analysis
  - Output
  - Correlation
  - Defense Update
  - Workstation

Data Model

- UUIDs
  - types of data in data blocks
  - formats of metadata
  - types of nuggets
  - types of applications
- Allows data to be routed to only the nuggets equipped to deal with a given format.

The Dispatcher

- The heart of the Razorback system
- Available APIs:
  - Detection Nugget registration
  - Data Handler registration
  - Detection requests
  - Alerting
  - Full analysis logging
  - Output to API compliant systems
- Database driven

Database

- Database is used to store important context information surrounding the alert, such as:
  - Timestamp
  - Priority
  - Message
  - Source and destination IP
  - IP protocol
  - Short and long data fields
  - Any other metadata

General Nugget Functionality

- Uses a persistent UUID for communicating with the Dispatcher
- Registers with Dispatcher
  - Types of data handled
  - Types of output generated

Data Collector

- Capture data and generate metadata
- Contact dispatcher for handling
  - Has this file been evaluated before?
  - Where should it be sent?
- Pass that data set to a Detection Nugget
- Accept feedback from the Dispatcher for detection request
  - Asynchronous alerting
  - Local cache of detection outcome

Detection Nugget

- Handles incoming data from Data Collectors
- Splits incoming data into logical sub-blocks
  - Requests additional processing of sub-blocks
- Provides alerting feedback to the Dispatcher

Output Nugget

- Receives alert notification from Dispatcher
- If alert is of a handled type, additional information is requested:
  - Short Data
  - Long Data
  - Complete Data Block
  - Normalized Data Block
- Sends formatted data to relevant system

Correlation Nugget

- Interacts with the database directly
- Provides ability to:
  - Detect trending data
  - Identify "hosts of interest"
  - Track intrusions through the network
  - Initiate defense updates

Defense Update Nugget

- Receives update instructions from dispatcher
- Performs dynamic updates of network device(s)
- Notifies dispatcher of defense update actions

Workstation Nugget

- Authenticates on a per-analyst basis
- Provides analyst with ability to:
  - Manage nugget components
  - Manage alerts and events
    - Consolidate events
    - Add custom notes
    - Set review flags
    - Delete events
  - Review system logs

Concept of Operations

How do they work together?

Data Collector

- Data is captured
- Metadata is generated (URL/filename)
- Checks a local cache of previously looked at URLs and data signatures
- Uses an API to manage the initial file evaluation and cache checks
- If further inspection necessary, API threads out and ships the data off to the Dispatcher

Dispatcher

- Tracks all nuggets in the system
- Finds the set of nuggets with the capability to handle the incoming data type
- Routes incoming detection requests to that set of nuggets
- Keeps track of metadata via an event id

Detection Nugget

- Processes data provided by the Data Collectors, as instructed by the Dispatcher
- Data is portioned out to the respective analysis thread able to analyze that data type
- Results of the analysis are sent back to the dispatcher in the form of alerts
- Additional metadata may be sent

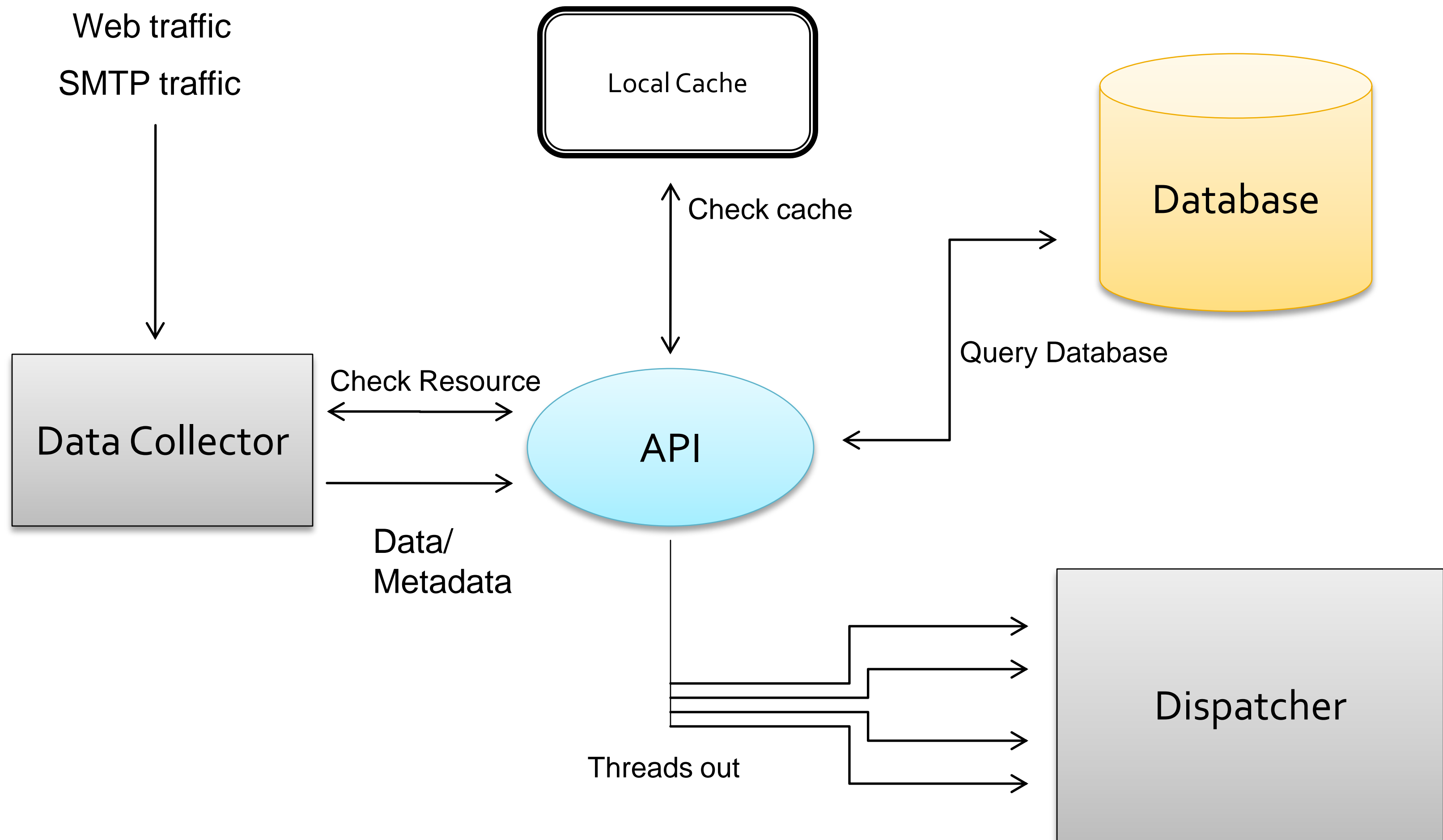
Dispatcher, part deux

- Incoming alerts are associated with their context data via the event id
- Information is stored in the database
- Portions of the capture data, namely, the portion that triggered the alert, are stored
- Dispatcher notifies all output nuggets that it has alert data to be retrieved

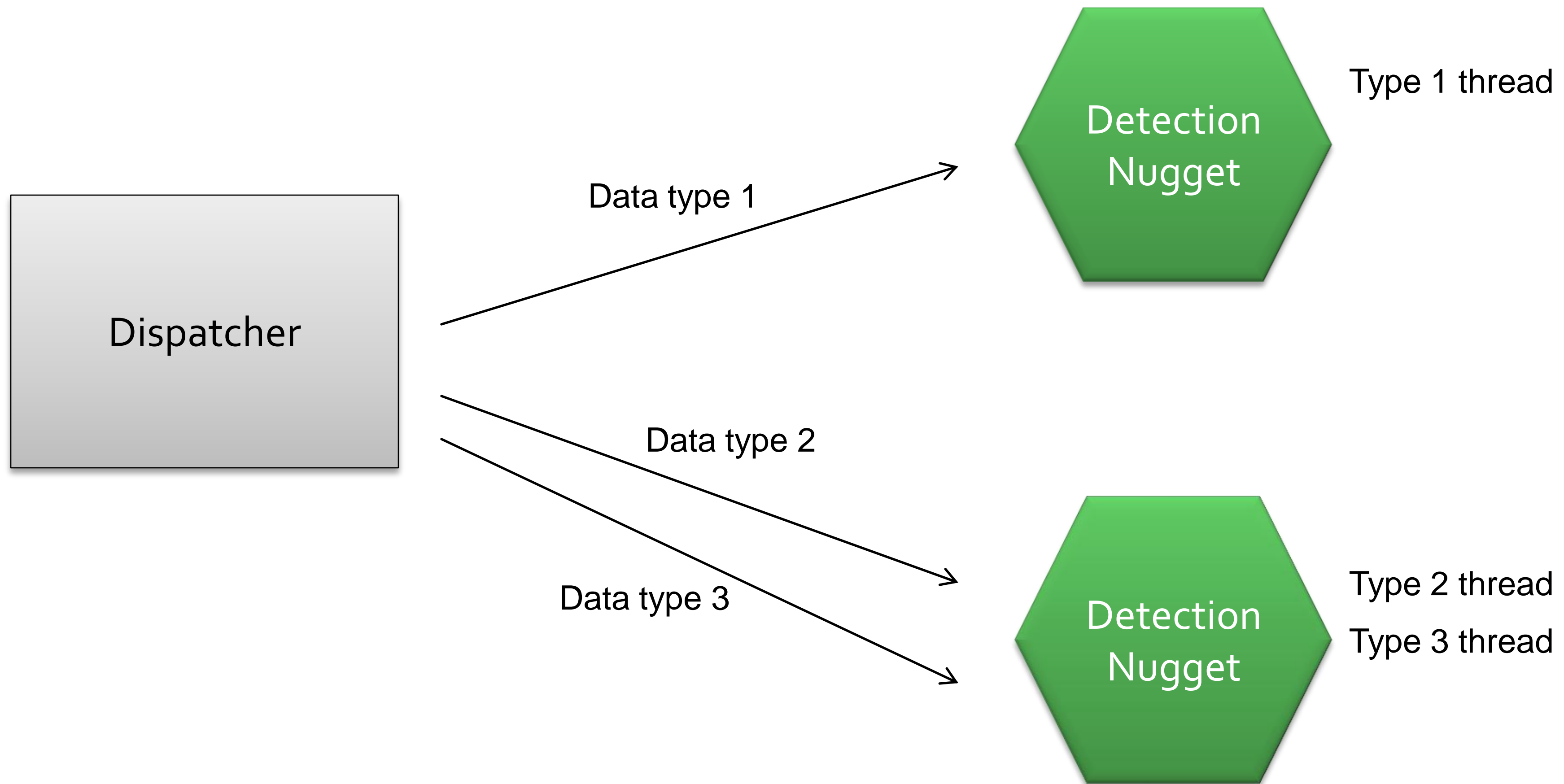
Output Nugget

- Output Nugget receives notification that an alert is available
- If interested, the output nugget informs the dispatcher it would like to retrieve this alert
- Dispatcher forwards additional alert information the output nugget

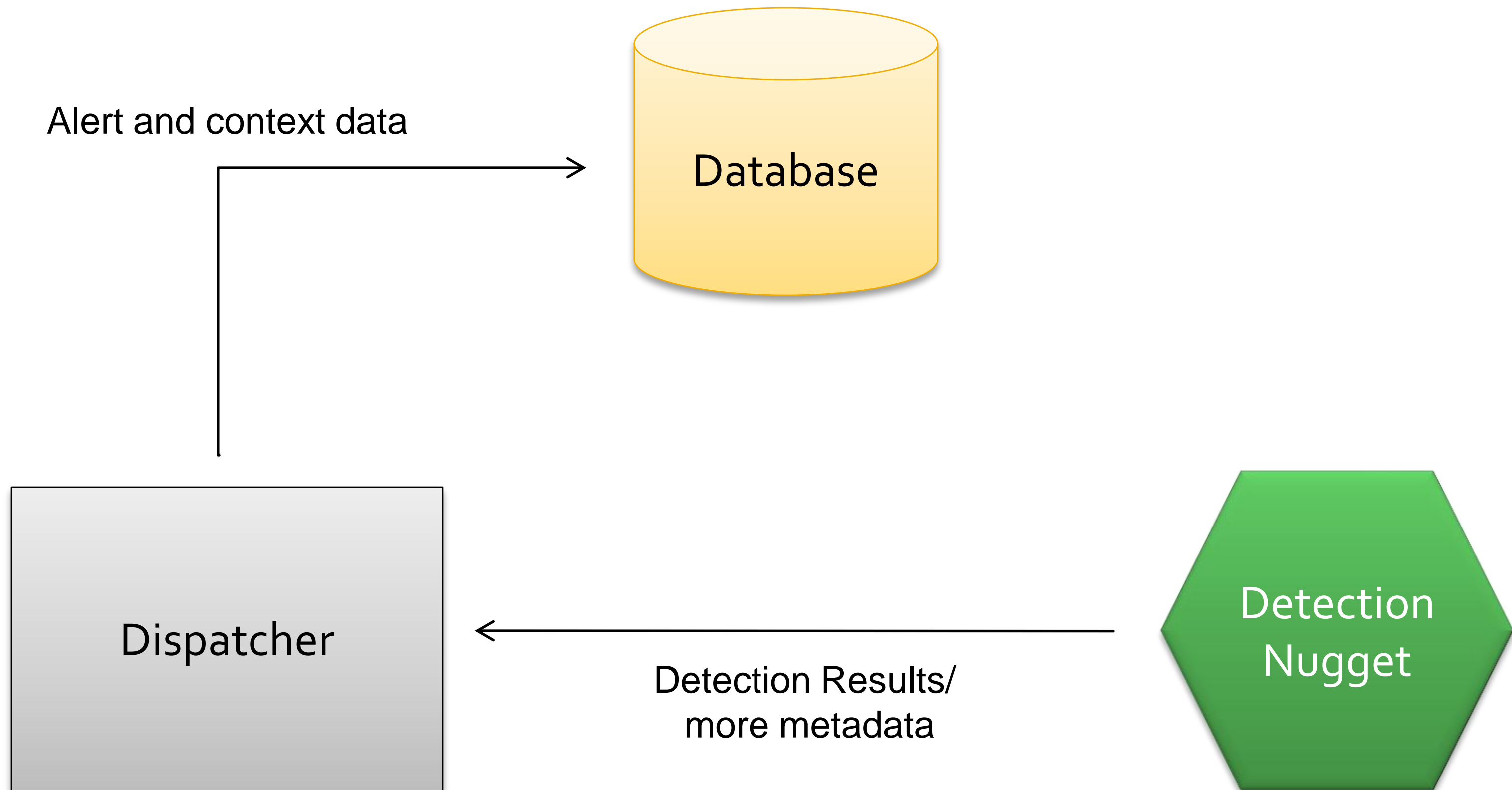
Traffic comes in…



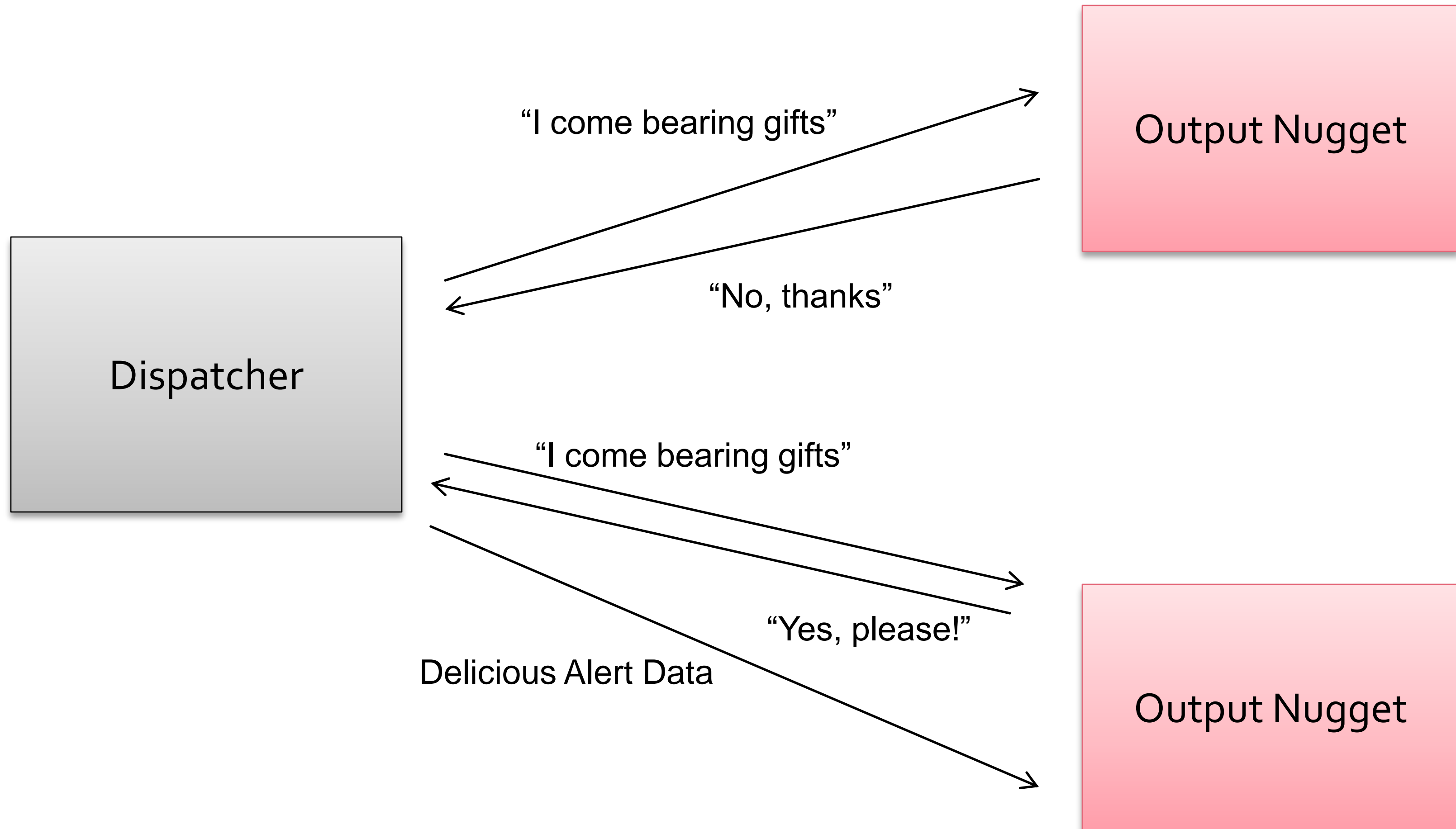
Dispatcher farms out detection…



Alerts are sent back…



Output nuggets are informed…

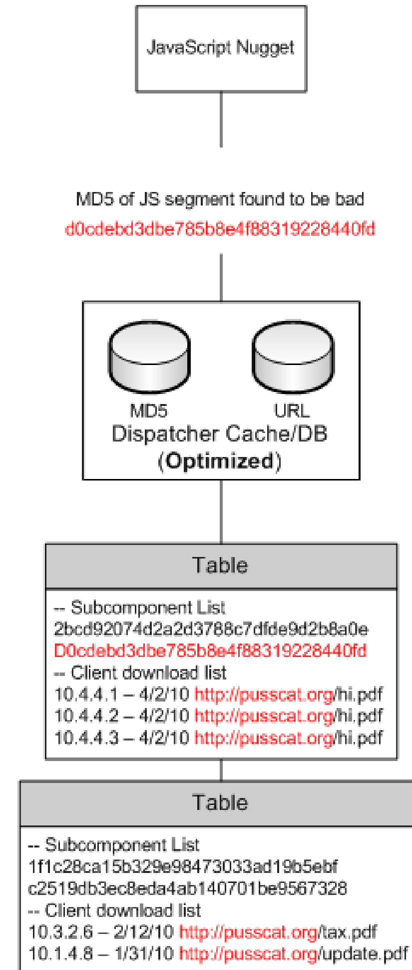


We Like Data

- MD5 and size is stored for files and subcomponents both bad and good
- Primarily this is used to avoid reprocessing files and subcomponents we've already looked at
- But after a update to any detection nugget, all known-good entries are "tainted"

Why Taint known good?

- After an update to detection, previously analyzed files may be found to be bad
- We don't rescan all files
- But if we see a match for md5 to a previous file, we will alert retroactively



Case Study: SMTP

What happens when an email is received?

Incoming SMTP Traffic

- Client data collected by Snort-as-a-Collector
- Collected data sent to SMTP Detection Nugget for separating MIME components
- MIME components are sent back through the Dispatcher for further analysis

Snort as a Collector (SaaC)

- Modified version of snort 2.8.6
- Uses snort's protocol analyzers and stream reassembler to grab session data and hand to Dispatcher
- Dispatcher sends data to the SMTP Detection Nugget

SMTP Detection Nugget

- Receives data from SaaC via the Dispatcher
- Extracts SMTP headers for metadata and tracking information
- Separates all embedded MIME components to be sent back to Dispatcher for further analysis
- Collects alerts and sends them to the Dispatcher for correlation

ClamAV Detection Nugget

- In our example, an EXE file was attached to email, resulting in data being sent to the ClamAV detection nugget
- Receives input files, runs through ClamAV
- Alerts sent back to Dispatcher

Output Handler Nugget

- Receives notifications from Dispatcher that alerts are available
- If interested in the type of alert, calls back to Dispatcher for extended data
- Provides formatted alert data to SIM

Current Capabilities

Nuggets that are currently available. Many more to come, and you can help!

Data Collectors

- 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

Detection

- 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

Detection (cont'd…)

- 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

- 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

- Snort rules updater
- ClamAV rules updater
- Triggered session storage via Daemonlogger

Workstation

- CLI functionality to query:
  - Alerts, events, and incidents
  - Nugget status
  - Display metadata
  - Run standardized report set

Programming Interfaces

How are nuggets created?

Custom API

- Nuggets can be written via a provided API
- The API provides basic functionality for:
  - registering a new nugget
  - sending data to be analyzed
  - sending alert data to be processed
  - querying the cache/database
- API is written in C, but wrappers are available for use with Ruby, Python, and Perl

Description

- The API provides to the developer a set of function calls passed as part of several C-structures
- Existing APIs
  - DetectionAPI
  - CollectorAPI
- APIs for other nugget types forthcoming

CollectorAPI

- registerCollector()
  - Register to Dispatcher
  - Identify custom name and UUID representing application type
- checkResource()
  - Checks given URI before sending data to be analyzed
  - Function assigns an event UUID if none is provided
- sendData()
  - Sends collected data for analysis
  - Send-and-forget; dispatcher takes care of the rest
- sendMetaData()
  - Metadata is handled like normal data
  - Sent to a special nugget before being stored in the database

Detection API

- registerHandler()
  - Registers detection function to one or more data types
  - Detection function must accept a data pointer and length
- sendAlert()
  - Sends alert data to the dispatcher
  - Links alert to event by event UUID
  - Provides mechanisms for arbitrary and extensible alerting formats
- sendData()/sendMetaData()
  - Identical to CollectorAPI counterparts
  - Provides detection nuggets with the ability to have sub-data blocks analyzed via the Dispatcher

What if I don't like C?

- Nuggets can be written in Ruby, Python and Perl
- Wrappers providing interfaces to the API functions are provided

Conclusion

Let’s wrap this up!

Razorback Framework

- Completely modular architecture
- Each component has a highly specialized function
- Complex functions are handled by routing sub-blocks back through the Dispatcher
- The Dispatcher is the true heart of the framework and is responsible for routing data and alerts throughout the system

Nugget Types

- Data Collector
- Detection
- Output
- Correlation
- Defense Update
- Workstation

Development

- Core system is in C
- APIs provided for performing all interactions with the Dispatcher
- If you can handle a data pointer and a size, all you need to worry about is what you want to detect!
- API Wrappers provided for Perl, Ruby, and Python

How You Can Help

- More collection nuggets needed!
  - Additional protocols
- More detection nuggets needed!
  - Additional file types
- More defense updater nuggets needed!
  - Update more network devices
- More correlation nuggets needed!
  - Are you great at data mining? We need you!

Questions??

- Patrick Mullen
  - pmullen@sourcefire.com
  - phoogazi on Twitter
- Ryan Pentney
  - rpentney@sourcefire.com
- Sourcefire VRT
  - labs.snort.org
  - vrt-sourcefire.blogspot.com
  - VRT_Sourcefire on Twitter