Razorback System Design Document

Alerting & Data Block Capture

*0.1 Release*

Final

1/12/2011

Table of Contents

[System Summary 3](#_Toc282685345)

[System Management 3](#_Toc282685346)

[Kool-Aid Target Impact 3](#_Toc282685347)

[Customer is not stupid 3](#_Toc282685348)

[Be Nice to the Customer 3](#_Toc282685349)

[Speed and Scalability 3](#_Toc282685350)

[Component Detailed Design 4](#_Toc282685351)

[Dispatcher 4](#_Toc282685352)

[Database 6](#_Toc282685353)

[API 8](#_Toc282685354)

[Design Deficiencies 8](#_Toc282685355)

[Long Term Strategy 9](#_Toc282685356)

Table of Figures

[Figure 1: Component Chain for Malicious PDF 5](#_Toc282685315)

[Figure 2: Alert Database Structure 7](#_Toc282685316)

## System Summary

The alerting system supports all aspects of communicating, storing and retrieving information generated by nuggets acting under the detection functional umbrella. The alerting system has the potential to generate significant load on the system as it deals with non-cached, large data blocks and is a primary target for user interaction.

## System Management

|  |  |  |  |
| --- | --- | --- | --- |
| System Title | Alerting System | |  |
| System Manager | Matthew Olney | 410-423-1938 | |
| System Tech Lead | Matthew Olney | 410-423-1938 | |

## Kool-Aid Target Impact

### The Customer is not stupid

Structures are set up to support advanced forensics analysis of data blocks. Customized flag fields are available for enterprise-specific tagging. Metadata and note fields are available to tag events, alerts and data blocks.

Be Nice to the Customer

Blocks are stored in normalized and raw forms, so that the customer does not have to replicate the work done by the nuggets. APIs allow for additional, arbitrary information to be provided by nuggets to assist in the response process.

### Enterprise Speed and Scalability

Several database optimizations have been made:

* Use of InnoDB engine where appropriate
* Checking database prior to requesting data blocks
* Using CRC32 as indexing mechanism for hashes

On the API side, we require per-data block requests to encourage output nuggets and workstations to request only the data they need. The data block chain design allows for each sub component to be hashed so we only evaluate those blocks that are required, even if they occur in other data blocks.

## Component Detailed Design

### Dispatcher

Alerting interface with the dispatcher begins when the dispatchMaster() sees one of two Snake Charmer packet types: REQBLOB and ALRT. Both of these are passed to the handleAlert() function, which is the beginning of the alert system proper. This function completes the initial capture of ALERT\_HEADER data which is the data structure used in the SC protocol to communicate alert data. The data is then associated with an ALERT\_TRACKER structure, which is used to capture a data block (if necessary) and is bound via an alert\_id so that the data\_block is associated with the correct alert.

#### Map to True Event ID

Once the ALERT\_HEADER is captured, ResolveEventLookup() is called to map the temporary event id provided by the nugget. If this is the first alert to the temporary event id, a database query gets the next available event\_id otherwise the temporary to real id cache is queried to get the mapping. The ALERT\_HEADER structure is then updated with the real\_id and insertAlert() is called to enter the information into the database. As the data is inserted into the database, the alert\_id is captured and stored in the ALERT\_TRACKER.

#### Optional Data

The ALERT\_HEADER structure contains a flags field which is used to communicate what additional data is available from the nugget. This data is potentially large and is optional, so it is handled separately so that checks can be in place not to replicate data insertions. Two additional data types are LONG\_DATA and SHORT\_DATA, which are arbitrary information blocks provided by the nuggets. Typically this field is used to provide additional analysis of the data. The META\_DATA flag is a data block that is associated with a well-known UUID that defines the type of metadata. This is in place so that data can be stored in a way that allows rapid searching of alerts that contain specific meta data types.

#### Data Block Capture

The other flags are associated with the capture of the data block associated with the alert. The FULL\_BLOCK and NORM\_BLOCK indicate that raw and normalized data blocks are available. These flags trigger a component check that checks the MD5 sum provided by the ALERT\_HEADER structure against known blocks in the database. If the block does not exist, then the dispatcher fetches that data block from the nugget.

The SUB\_COMPONENT flag is used to build the data block chain that lays out how sub-components associate to each other. This association is critical to performance, incident response kick-start and alerting back in time. When the ALERT\_HEADER structure is filled out the main\_hash field contains the hash of the immediate parent of the current block and the db\_hash contains the hash of the current block. If the main\_hash and db\_hash are the same, the data block is the original data block passed into the system. If they are different, the current block is associated to the parent block. This builds a data block chain that completely describes the original data block and contains the raw and normalized forms of all components.

The chain allows a couple of things. First, if a sub-component is subsequently found to be bad, then we can alert back in time for all documents that contain that md5. For example, in the chain shown in figure 1, the shell code MD5 has been determined to be bad. This causes all parent hashes to be marked as bad. Note that this only goes up the chain, but does not mark components below it or not in a direct line up as bad. This avoids marking otherwise good block as bad and causing false-positives later.

We now have the ability to search for all data blocks that contain the shell code hash, JavaScript hash or PDF hash and notify the user that documents that match these have been retrieved in the past. Additionally, future data blocks that contain any of the blocks will be immediately marked as bad, improving speed-of-alerting to support high-latency service blocking.

Figure 1: Component Chain for Malicious PDF

#### Optional Data Storage

Each of these flags is checked separately. For each flag that is set, the tracker blob\_type, blob\_string and blob\_string\_len are set and requestBlob() is called. requestBlob() manages the communication back to the nugget to retrieve the specific blob and then inserts the data block into the database in the correct location.

#### Output Nuggets

Finally, the razorback config structure is checked to see if an output nugget has been defined. If it has the alertNotify() function is called. This function grabs the configuration information and for each output nugget calls sendAlertNotify(). sendAlertNotify() manages the network connection to the output nugget and transmits the contents fo the ALERT\_HEADER structure to the output nugget. It is up to the nugget to handle the alert data and request data blobs as necessary.

#### Data Block Retrieval

The handleAlertBlob() manages requests for additional information from Output Nuggets. Note that the alerting system does not support ad-hoc queries for data blocks, only data blocks that are part of an alert notification. The handleAlertBlob() uses getDataByAlert() which fetches a given data blob (FULL\_BLOCK, NORM\_BLOCK, SHORT\_DATA, LONG\_DATA) from a specific alert\_id. The process is essentially the same that the alerting system uses to retrieve data from nuggets in reverse.

### Database

As shown in Figure 2, the alert structure is the hub of all alert related data. Each individual alert is related to an overarching event\_id which binds all alerts associated with a single data block submission together. Boolean values indicate the availability of various data sets including metadata, raw and normalized data and alert interpretation data (long\_data dn short\_data). Additionally, analysts can make notes against incidents and events.



Figure 2: Alert Database Structure

The component storage in the database consists of three tables: sub\_components, components and component\_blocks. At this point, component\_blocks are no longer necessary and need to be removed. They were originally used to organize all data associated with a single document, but this didn’t allow chaining. One of the projects for Q2 should probably be to eliminate the component\_block concept. It may be that we can use the component\_block struc ture to define top-level components, a review is needed.

The component block holds all of the data on the data block. For data blocks that haven’t been determined to be bad only have the checksum and data type. It is only after a block has been marked bad that the system requests the data. Each component contains only one block of data, so for the raw block, there is a foreign key linking to the normalized data (if it exists) that is another component. For performance purposes, the hash is converted on insert into a 32-bit crc check. This allows the InnoDB engine to create a BTREE index on the data. When queries come in they are first matched against this 32-bit CRC and then checked against the true hash/size.

The sub\_components structure is simply to foreign keys that link two components. This is the chaining process. A top-level component can be fully defined by travelling down the entire tree. The external linking also allows queries to display all data blocks that include the sub-component, which is forensically very useful.

### API

There are three publically available API functions associated with the alerting function:

HRESULT deliverJudgement(JUDGEMENT \*verdict);

HRESULT getAlertData(unsigned alertID, unsigned blob\_type, unsigned char \*\*data, unsigned \*size);

HRESULT sendAlert(ALERT \*alert);

sendAlert() and deliverJudgement() are used by detection nuggets to communicate findings on data blocks. sendAlert() populates the ALERT\_HEADER structure and sends it to the dispatcher. It should be noted that an alert does not have to be negative; the context is determined by the nugget.

deliverJudgement() sets the threat flags field, defined as follows:

/\* THREAT FLAGS \*/

#define KNOWN\_GOOD 0x00000001

#define KNOWN\_BAD 0x00000002

#define THREAT 0x00000004

#define HAZZARD 0x00000008

#define WHITE\_LIST 0x00000010

#define TAINT 0x00000020

#define WATCH\_LIST 0x00000040

KNOWN\_GOOD represents a file that has been reviewed and not found to be malicious. Malicious files are marked KNOWN\_BAD. Users can white-list files by setting the WHITE\_LIST flag. The other flags are implementation specific, and their use will be determined in future Razorback versions.

Output nuggets receive the alert notification as part of the masterOfNuggets() listener. Once the output nugget has received the initial alert data, they can request the full data block set with the getAlertData() function. That function is called on a per-block, per alert\_id basis, allowing the nugget to request the specific information it needs, as opposed to getting the entirety of the alert data.

## Design Deficiencies

1. Currently, the best way to access the data is through ad-hoc queries to the database directly. This presents a major barrier to meeting AAA expectations at the enterprise level, security issues and performance issues.
2. Storage of the data blocks is done by placing them into a component table in the MySQL database. This limits the scalability of the database backend and limits search options for the user.
3. Error handling on messages between the nuggets and the dispatcher could be improved.
4. There is no methodology for output nuggets to cascade up or down a tree to gather the full output of all analysis of a data block.
5. It is possible for long chains to alert late and have parent chains not available to store in the database.
6. There is no cache on the output-nugget side, so we have to rely on the output nugget to not repeatedly request the same data over and over again as alerts come in.
7. If a detection nugget submits a hash that we have already seen, there isn’t currently a mechanism to alert again.

## Long Term Strategy

Additional work on the component storage, detection nugget and output nuggets is needed. On the component storage side, the database schema needs to be reviewed to introduce file-system storage for large blocks (either through native file systems or through a map-reduce type database). Detection nuggets need to have a way to cache data blocks for some period of time so that the dispatcher can request the blocks to populate the data block tree if a late alert comes in

From the Alert API side, additional error checking and handling is probably warranted. Some auto caching on output nuggets is necessary so that the API hands a cached copy of data blocks that have already been retrieved. This will reduce the likelihood of a misbehaving output nugget negatively impacting the dispatcher. Also the ability to retrieve the complete chain should be made available to the end user.

Related to the Alert API, but probably in the Workstation API, common queries should be made available through the API to reduce the necessity of ad hoc queries on the database. Some examples:

* List of all alerts associated with a specific event
* List of all top-level data blocks that contain as specific data block
* Retrieval of metadata associated with block
* Ability to add notes to alerts and blocks