Razorback Requirements Document

**Q2 Block Submission**

Version 0.2

Revision 3

5/5/2011

## 

Table of Contents

[Introduction 3](#_Toc292359158)

[Customer Focus Statement 3](#_Toc292359159)

[Requirements 3](#_Toc292359160)

[Performance 3](#_Toc292359161)

[Metadata 3](#_Toc292359162)

[Support Blocking Inline Collectors 3](#_Toc292359163)

[Sub Block Association 3](#_Toc292359164)

[Implementation 4](#_Toc292359165)

[Block Precedence Interpretation 5](#_Toc292359166)

[Submission Fast Path 5](#_Toc292359167)

[Dispatcher Block Precedence Request 6](#_Toc292359168)

[Dispatcher Block Precedence Request Processor 6](#_Toc292359169)

[Block Submission Processor 6](#_Toc292359170)

[System Diagram 7](#_Toc292359171)

[Metrics 8](#_Toc292359172)

[Impact 8](#_Toc292359173)

[Future Work 8](#_Toc292359174)

[Enhancements 8](#_Toc292359175)

## Introduction

The block submission system is used for inserting blocks into the analysis queues. It is the interface that customers will use to submit data to the system from their own collection and analysis nuggets.

## Customer Focus Statement

The submission system is designed to be a simple interface for the consumer to use. It will support streaming data into the submission.

## Requirements

### Performance

* The submission process must not block the calling thread for longer than absolutely necessary.
* Data blocks must not be transmitted over the network if they are not needed.
* The submission process should provide a local cache of block disposition to reduce the number of queries to the dispatcher.

### Metadata

* A collector must be able to attach Metadata to the block being submitted.
* A collector must be able to attach Metadata to the event being generated.
* The Metadata types must be flexible – The customer must be able to attach new types of metadata to blocks that they find of interest.

### Support Blocking Inline Collectors

* The local cached flags fields must be returned to the collector so that it can make a decision on what to do with the data block.

### Sub Block Association

* Must always record sub block association data regardless of local cache response and black/white listing.
* Sub block must have separate metadata from the parent block.

### Block Data Storage Pool

* The system must support gradual accumulation of block data from the collector, with a final call to indicate that they have finished adding data.
* The system must support three types of data block segments:
  + Malloc’d
  + Mmap’d
  + User Managed
* The block pool destroy function must call a callback function to let the submitter know that the block is about to be destroyed.
* The block pool will provide automatic block data type detection using libmagic.
* The submission handler will by default destroy the item in the block pool, unless the MAY\_REUSE flag is set when calling the submission routine.

## Implementation

### Block Pool

A collector will call the following function to set up a new event for collection:

struct BlockPoolItem \*BlockPool\_CreateItem (struct RazorbackContext \*p\_pContext)

This will take the current API context as an argument to prepopulate a number of fields in the item. When the collector is ready to add data it will call the add data function setting the flags to indicate how the memory should be managed:

bool BlockPool\_AddData (struct BlockPoolItem \*p\_pItem, uint8\_t \* p\_pData, uint32\_t p\_iLength, int p\_iFlags);

It the collector wishes to specify the datatype then they may call the following function else the datatype will be worked out with libmagic:

bool BlockPool\_SetItemDataType(struct BlockPoolItem \*p\_pItem, char \* p\_sName);

Once the collector has finished adding data to the item it must call the following function to finalize the hash and make the block ready for submission:

bool BlockPool\_FinalizeItem (struct BlockPoolItem \*p\_pItem);

Before submitting the item the collector may set the submittedCallback to point at a function to be called after the item has been submitted to the dispatcher. For single shot collectors such as file inject this allows the API to notify the caller when it is safe to exit, for collectors using other memory management systems then this callback should be used to free all the memory used to store the block data.

### Submission

When a collector is ready to submit an item to the dispatcher it will call the following function, this function will submit the block to the dispatcher and set the flags fields to the currently locally known values.

bool Submission\_Submit(struct BlockPoolItem \*p\_pItem, int p\_iFlags, uint32\_t \* p\_iSfFalgs, uint32\_t p\_iEntFlags);

If this function is called with the MAY\_REUSE flag set then after the first submission the data blocks will be removed from the block pool and only the event information will remain. This can then be reused to submit additional block and event metadata with UPDATE flags set when calling the function.

If UPDATE is the only flag set then the block pool item will be destroyed, if MAY\_REUSE is also set then the item will be left in the pool for use again.

If the collector decides not to reuse the item after a submission with MAY\_REUSE set then they can call the following function to remove the item from the pool:

bool BlockPool\_DestroyItem (struct BlockPoolItem \*p\_pItem);

### 

### Block Disposition Interpretation

As a block submission engine we care about the following bits in the SF\_Flags field:

* Known Good
* Known Bad
* Black List
* White List
* Dirty
* Can Haz
* Processing

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **White/Black**  **Listed** | **Known Good/Bad** | **Dirty** | **CanHaz** | **Processing** | **Result** |
| Not Set | Not Set | Not Set | Not Set | Not Set | Submit Event Record |
| Not Set | Not Set | Set | Not Set | Not Set | Submit Event Record |
| Not Set | Any | Any | Set | Not Set | Submit Full Block |
| Not Set | Not Set | Any | Set | Set | Submit Event Record |
| Set | Not Set | Not Set | Not Set | Not Set | Submit Event Record (See Notes) |
| Set | Set | Not Set | Not Set | Not Set | Submit Event Record |
| Set | Any | Any | Set | Not Set | Submit Event Record (See Notes) |
| Set | Any | Any | Set | Set | Submit Event Record (See Notes) |
| Not Set | Set | Not Set | Not Set | Not Set | Submit Event Record |
| Not Set | Set | Set | Set | Set | Submit Event Record |

**Notes:**

Whether to respect the white/black list flags will be configurable on a per instance basis. If the system is configured to disrespect this field a full block will be submitted.

### Submission Fast Path

The submission process fast path is optimized so that it does not block the collector’s data collection routine for longer than absolutely required.

In order to do this the only things that are done in the fast path are:

1. Look up the data block in the local cache.
2. Update the block status in the block pool.
3. Return the local cache result.

All other submission processing happens in separate threads outside of the context of the data collection path.

### Dispatcher Block Disposition Request

A separate thread will be in charge of sending requests to the dispatcher for fulfillment, this thread will be woken up in the fast path after a blocks status has changed.

### Dispatcher Block Disposition Request Processor

This thread will be in charge of receiving precedence responses from the dispatcher and updating blocks in the block pool.

### Block Submission Processor

This thread is in charge of submitting blocks and events to the dispatcher for processing. Once the block has been submitted the thread will call the block pool call back before destroying the item in the pool.

### System Diagram



## Metrics

* Fast path latency – Possibly too expensive to collect.
* Local cache hit/miss stats
* Block precedence request latency
* Block precedence hit/miss status
* Block pool utilization.
* Failed/Timed out precedence Requests

## Impact

* Requires changes to the block submission processing thread in the dispatcher.

## Future Work

### Enhancements

* Do not put the data block in the Queue, pass it in a special block transfer system.
* Add support for fully synchronous global cache look ups for inline devices that opt to use them.
* Add support for gray listing
* Add request timeouts and retries to items in the pool.
* Add the ability to page malloc’d data buffers to mmap’d files for *swapping* when we are memory constrained.