##### **IMPLEMENTING RESOURCE INTENSIVE DETECTION TECHNIQUES WITH THE RAZORBACK FRAMEWORK**

Copyright, 2010

Sourcefire, Inc.

Matthew Olney

Senior Research Engineer

Matthew Watchinski

Senior Director, Vulnerability Research

Sourcefire Vulnerability Research Team

9780 Patuxent Woods Dr

Columbia, MD 21046

[molney@sourcefire.com](mailto:molney@sourcefire.com)

***Abstract –*** Client-side attacks pose several challenges to detection systems that are not typically present in server-side attacks. The vulnerabilities present in client-side applications generally are associated with problems in parsing complex file formats. In order to provide detection targeting the conditions that lead to a successful exploit (as opposed to providing detection targeting known attacks already encountered in the wild), detection engines must typically replicate some level of the parsing functionality of the client-side application. The complexity of the file formats, the numerous options available to an attacker for obfuscation and the various means by which attacks can be delivered mean that full parsing and detection require significant computational resources and time.

This paper discusses the advantages of separating the data capture and detection functionalities within an overall incident detection system. It will also discuss the implementation of that idea in the form of Sourcefire’s Razorback™ Framework. Razorback features a distributed detection system, robust API set and a fully extensible database and data management system. It was designed specifically with the needs of high-level incident response and detection teams.

I. Introduction

While client-side attacks have always been an option available to attackers, the past few years have seen a dramatic shift away from server attacks as attackers have moved to targeting systems that are often both less well protected as well as more trusted. These attacks are typically embedded in much more complicated data sets than are traditionally found in server attacks. These file formats typically allow for an impressive array of obfuscation techniques including fragmentation, compression and encoding. While each of the obstacles can be overcome, there is a relatively high CPU cost to handle them.

This cost is especially problematic when operating in a low-latency environment. Passing traffic at multi-gigabyte speeds while essentially replicating the processing of hundreds or thousands of client computers is challenging, to say the least. The approach that the Vulnerability Research Team (VRT) has taken to allow for the development and deployment of high-cost detection methodologies is to separate the data capture functionality from the detection functionality. While this removes the capability to block first-seen attacks with these detection techniques, it does provide a much more robust and accurate detection system.

II. Defense Routing

Once detection and capture become separate processes, a system has to be in place to get the captured data to the detection engine. Because detection is in most cases tightly bound to a specific type of data, any captured data must be appropriately tagged to identify it. The system also must have an understanding of the types of data that individual detection engines are capable of handling. These two pieces of information are critical to minimize unnecessary data transfer and its associated overhead on detection and capture systems.

Take, for example, a standard web session in which a user browses a site and then downloads a PDF file. The amount of potentially interesting data, either from detection or a forensics standpoint, is impressive even from such a simple scenario. This is an incomplete but representative list of interesting data:

* DNS Query and response
* Parameters passed on the request
* Hostname of the web server
* IP Address of the web server
* URI of the page
* HTML for the page
* Scripting entries for the page
* Graphics on the page
* Links on the page
* The PDF file downloaded

The data collection process, then has to have an understanding of the various data types that the system might be interested in, and the capability of delivering that data along with a declaration of its type into the system. In order to make that data useful, detection systems must ensure that the system as a whole understands the kind of data that it handles.

Either a centralized system or a system that ensures each data collector is fully aware of what data is important and where that data should go must be in place. In the case of Razorback, the centralized approach is taken and that functionality is part of the Dispatcher.

As each component of the Razorback system comes online, it registers with the Dispatcher. Components are referred to as “Nuggets”, as in nuggets of functionality. In the case of a Data Collector, the registration simply declares that the collector is online and may engage the Dispatcher to assist in detection. Detection systems in Razorback are a collection of Detection Nuggets that work on the same data type. Each Detection Nugget performs its own type of detection on that data type. Detection Nuggets also register with the Dispatcher the number of resource threads available to process detection requests from the Dispatcher, and if they are interested in more than one specific data type.

The Dispatcher, using the information from Detection Nugget registrations, builds a routing table (figure 1).



Figure 1: Route Table for a Single Data Type

For each data type, for example Excel documents, JavaScript or Flash files, a list of the different kinds of Detection Nuggets available is maintained. The different Detection Nuggets represent different software that handle the same kind of data. In the case above, there are three separate approaches to analyzing the data. This would be common if an enterprise used a Sourcefire provided Detection Nugget, a Nugget provided by a third party and a private, enterprise specific Nugget. For each approach, there is a list of systems by IP and port that are running that Detection Nugget. Each time the Dispatcher is forwarded the appropriate type of data, one Detection Nugget from each of the different detection systems is selected to receive the data.

One important concept used in developing the Dispatcher is a data agnostic approach. The Dispatcher has no understanding of what a PDF file is, or what an email is. Instead it simply understands that PDF files are represented by a certain code, and if it sees data with that code it then ensures that the correct set of Detection Nuggets sees that data. The Dispatcher is critical to the operation of the system, but a majority of the operational intelligence of the system is pushed out to the various functional components that register with the system.

The architecture leads to two important results, when compared to a traditional detection system. First, because the data capture and data analysis functions have been separated, the system can ensure that any data block provided to the system will be evaluated by every Detection Nugget interested in that data type. This is an improvement over the traditional approach where a detection system can only evaluate data that it captures.

The other benefit of the architecture is that once data collectors are in place, implementing additional Detection Nuggets into the system is non-disruptive to the system because all analysis occurs out-of-band from the primary network.

III. Context

The initial response to many incidents is solely driven by the data provided by some defense technology, be it IDS, HIPS, Anti-Virus etc… While having a declaration of what the perceived problem of a piece of data is important, there are numerous other pieces of data that, if present, are very useful to the incident response process.

For example, while being notified that a PDF file attempts to exploit Adobe Reader through the newplayer() vulnerability (CVE-2009-4324) is important, there are additional critical pieces of information. First, how was the file delivered, by mail or by web? If it was by mail, where did it come from, what do the headers look like? If it was web, what was the URL, what was the header response and what did the user do to get to that site? What obfuscation techniques were used? What does the embedded JavaScript look like when fully normalized?

The first area to grab additional context is at the data capture point. As part of the capture of the core data targeted by the Collection Nugget, additional context needs to be captured and transmitted along with the file. This can include MD5 sum and size, URI, mail flow, basically any data to frame how the data got into the network.

Within the Razorback system, additional contextual data captured by the system is called “Metadata”. As part of moving the core block of data, Collection Nuggets transmit additional blocks of metadata. This metadata can be used to provide additional context to alerts, or it can parsed by a Detection Nugget and delivered to the Dispatcher as network forensics data. This data can then be queried by response teams to assist in managing other incidents.

As Detection Nuggets parse data, they may extract blocks of data that can be handled by another Detection Nugget. Razorback identifies them as sub-components and tracks how they are associated with the original file. While we’ll discuss this more fully in the file storage section, it is important to know that many data types such as JavaScript and Flash are passed back into the system for additional detection, are tracked separately, but are still associated back to the original data block.

IV. Scalability and Detection

Once the tasks of data capture and detection are separated, there is no requirement for them both to be on the same computer. This opens up both the opportunity for high-resource detection as well as the ability to scale up to enterprise-level performance by expanding the capability of the system across multiple servers.

Because network transfer is required to reach remote detection engines, certain considerations must be addressed. Unnecessary data transfer and unnecessary detection runs cause the system to incur overhead that would lead to network congestion, detection delay or resource exhaustion. Countering this requires an understanding both of what data would result in potentially successful detection and what data has already been analyzed in the past.

Razorback’s registration process dictates which types of data the system as a whole is interested in. If, for example, a Collection Nugget attempts to submit data of a type that is not registered by a Detection Nugget, the dispatcher will decline the submission and log the failure. This prevents processing of information that cannot possibly result in an alert or in forensics data being retained.

In addition to the registration process, both Collection Nuggets and Detection Nuggets have local caches that are checked. The cache maintains a list of data blocks that have previously been evaluated as either good or bad. If the block is not registered in the cache, Collection Nuggets and Detection Nuggets will then query the Dispatcher, which maintains both a global cache as well as access to the system database. If there is no record of the data block anywhere in the system, then detection continues.

When the Dispatcher returns a “file unknown” response to a cache/database check, it also instructs the querying system where to send the data block for analysis. The Dispatcher can handle this one of two ways. If the system implementation is configured (as is recommended, see Figure 2) to have the Nugget Farm (that is, a collection of Deep Inspection Nuggets) on a non-routable network behind the Dispatcher then the Dispatcher will accept the data block and distribute it to detection engines using the defense routing table.

If the system is configured in a manner that has full connectivity between the Collection Nuggets and the Detection Nuggets, then an alternate routing approach can be taken (this option is configurable in the Dispatcher configured file, see Figure 3). In this case, the Dispatcher sends a list of each Detection Nugget that need to see the data block. The Data Collector or Detection Nugget then transfers the block directly to the target Detection Nugget.

Another challenge of choosing an active routing mechanism is the need to track the resources of individual Nuggets. This prevents unnecessary data transfer by ensuring that only Nuggets with free threads are eligible to receive blocks for detection. In the case of Razorback this is primarily managed by having the Detection Nuggets declare the available threads for detection. Those systems with no available threads will not be entered into a routing response to Collection Nuggets. Additionally, Nuggets are able to declare out-of-resource issues that aren’t thread related, such as memory exhaustion. Also, the Dispatcher regularly queries the Detection Nuggets to ensure that they are functional and to that the resources tracking is correct.

At this point, the true purpose of the system (detection) begins. But, as stated earlier, the true intelligence of the system is pushed out to the various Nuggets and is out of the scope of this paper. This allows the Dispatcher to maintain a complete data-agnostic approach. This is important, as we’ll discuss more fully later, to make it easy for enterprise response teams to build custom Collection Nuggets and Detection Nuggets. Since the Dispatcher is only concerned about the identifier for data, an enterprise need only tag capture data with a unique id and have a Detection Nugget register as capable of handling that unique id.

There is, in Razorback, one special case that needs to be discussed. If, during the processing of a data block, the Detection Nugget finds a block of handled data, then the Detection Nugget is authorized to act as a Collection Nugget and begin the process of submitting the new data block into the system for detection. While the process for managing the new data block is identical to a standard Collection Nugget submission, the relation of the sub-component to the primary data block is retained for context based alerting.



Figure 2: Recommended Routing Configuration

V. Alerting and Data Storage

One of the advantages of using a single, distributed system is a unified alerting structure. In order for this approach to be viable, the alerting structure needs to be built in such a way that it can handle the most verbose output possible. If it is possible that the system can receive data that it is unable to handle, that data is lost to the incident response and security operations teams.

Verbosity of output is critical to the concept of supporting the incident response procedure. Because resource intensive analysis necessarily allows for an extended timeframe for analysis, components can perform deep file parsing, normalization and interpretation. A highly verbose alerting system allows for this work to be passed back to the incident response team, meaning they do not have to replicate that work.

Further, for items that are determined to be bad, data blocks can be passed back into the system retagged for post-alert analysis. For example, if a PDF file is found to contain an exploit, the data block can be retagged as PDF-BAD and then passed to specific detection systems that can perform predetermined automated analysis. The recursive nature of the system and capability for verbose reporting allows for extensive preparation of the data.

In the Razorback system, the database and component storage architecture combine to provide an essentially unlimited capability to pass data back to the system for end-user review. Razorback provides a database table designed for full data capture, including normalized data (Figure 4). Additionally, fields for metadata describing additional data about the data and notes added by analysts are provided.



Figure 3: Alternate Routing Configuration

.

The alerts table provides the standard fields that most systems would support: Alert ID (unique to the nugget type), Event ID (unique to the event that led to the insertion of the data into the system), timestamp, IP/port information, priority and a message.

+------------+----------------------+------+-----+

| Field | Type | Null | Key |

+------------+----------------------+------+-----+

| Alert\_ID | bigint(20) unsigned | NO | PRI |

| Event\_ID | bigint(20) unsigned | NO | MUL |

| Nugget\_ID | bigint(20) unsigned | NO | MUL |

| Timestamp | datetime | NO | |

| Priority | tinyint(4) | NO | |

| Message | varchar(255) | NO | |

| Src\_IP | int(10) unsigned | NO | |

| Dst\_IP | int(10) unsigned | NO | |

| IP\_Proto | smallint(5) unsigned | NO | |

| Src\_Port | smallint(5) unsigned | YES | |

| Dst\_Port | smallint(5) unsigned | YES | |

| Short\_Data | varchar(2048) | YES | |

| Long\_Data | mediumblob | YES | |

| Data\_ID | bigint(20) unsigned | YES | |

| Norm\_ID | bigint(20) unsigned | YES | |

| Notebook | bigint(20) unsigned | YES | MUL |

| Metabook | bigint(20) unsigned | YES | MUL |

+------------+----------------------+------+-----+

Figure 4: Alert Database Table

Other fields are provided to support extended alert and data interpretation output from detection systems. It is important to note that the Dispatcher does not control how the data is handled. The intelligence involved in deciding how and when to insert data in what field is left to the detection system.

Typically, either the short- or long-data field is available to provide a large (or, in the case of long-data, very large) field for verbose reporting on data interpretation. This output can specifically call out individual portions of data and describe the process and information that was used to determine the data was bad. The fields can also be used to store graphs, spreadsheets or other binary data. Again, the output systems are responsible for being able to handle interpretation of alert data, so files stored in these fields will need to be explicitly handled by an output system.

The Metabook and Notebook fields are keys used by metadata and notes structures to associate these structures to a particular alert. Metadata provides additional information about the data at the heart of the alert. For example, the SMTP header and the body of an email message could be stored along with a PDF file to describe where that data came from. The URL that was used to fetch a PDF file would be another example of metadata. Metadata is there to provide context around the data.

Notes fields, on the other hand, are free form fields that allow analysts to annotate various structures (Notebook and Metabook fields exist in more tables than just alerts). This allows human-generated data to be bound to the alerts, events and data that triggered the response.

Finally, data blocks (referred to within the Razorback framework as ‘components’), and their associated sub-components, are stored once and a tree is built that describes the relation of the blocks together. For example, a PDF component could potentially have two child sub-components such as Flash and JavaScript. The JavaScript component could then have an executable payload sub-component.

By taking this approach, we allow for the possibility that different components may share sub-components. This structure minimizes data storage and also supports intelligence driven response by allowing for different files with identical sub-components to be flagged as suspicious, even if they do not otherwise alert.

An important flag for data component management is the TAINT flag. The TAINT flag is set when detection is updated anywhere on the Razorback system. Normally an entry that exists in the database is known to be good or bad. However these definitions are no longer valid assumptions after a detection update. Files previously marked as good may, in retrospect have been bad and files that had been marked as bad may be good, or may have additional alerts that need to be associated with them.

The TAINT flag reveals an additional benefit to the component/sub-component tree setup. If a subcomponent has previously been identified as good, then marked with the TAINT flag and subsequently evaluated to be bad, the tree setup allows us to alert not just on the overarching MD5 of the file that triggered the alert, but also for all files that contained that subcomponent.

This architecture again comes back to the value of context. Nuggets should take any opportunity to provide information about data, whether it is structure, delivery mechanism, normalization or interpretation. From the architecture team’s point of view, more data is always better.

VI. Output

Ultimately, the output of any system should support the needs of the organization. A deep-inspection system needs to be able to support the intelligent delivery of data to output systems that ensure final delivery of alerts in an enterprise approved format. Intelligent delivery means that the verbose output available to Razorback must be reworked to conform to the structures and limitations of a given format.

Because of this, transmitting the full data, normalized data, long data and short data for all alerts would incur unnecessary overhead because systems would not use that data. Because of this, only the barebones of alerts should be passed to output systems along with a flag field declaring what additional data is available. Output systems would then be able to retrieve only those pieces of data that the output system needs.

Within the Razorback system, a variation of this process is used during alerting from the detection systems. The core alerting information is sent to the Dispatcher and immediately inserted. The Dispatcher then checks to see if the data blocks offered as part of the alert are already in the database. If they are, it saves the overhead of transmission; otherwise it requests the additional data blocks

The Razorback Framework’s output system is essentially a reverse of the alerting system. Some additional APIs are provided to have the structure of the component / subcomponent tree and the ability to retrieve metadata. Additional metadata can also be implemented to allow identifiers and links to relevant alerting, management or ticketing systems back into the database, associated with the appropriate data block or alert.

VII. API

Ultimately, the design decisions that have led to the current architecture of Razorback have been motivated by the desire to assist high-end response teams in developing organizational-specific detection components. In order to do this, performance, extensibility and customization were key concepts that were often expressed during design meetings.

It is important to understand that the major design decision was to push as much intelligence as possible out to the functionality nuggets. Data capture, analysis, alerting, output, correlation and system management are all provided as add-ons to the system. At its core, Razorback is simply a defense routing system, dependent on functionality from outside of the core system to provide value.

In order to provide rapid development of functionality nuggets, Razorback comes with a set of APIs that allow applications to integrate into the system. By providing those APIs in C, Perl, Ruby and Python, end users can make a decision balancing comfort, performance and safety in developing their applications.

Extensibility is not limited to the API. Because the UIDs that tag data type and nugget type are full length UUID strings, they offer somewhere in the neighborhood of 3 x 1038 unique identifiers. While this might seem an unnecessarily large set, the intention was to make integration of capability between organizational groups as easily as possible.

The database is also designed with extensibility in mind. Several tables have a Threat\_flags and Custom\_flags field. The Threat\_flags bit definitions are provided by Razorback. Examples of Threat\_flags would be KNOWN\_GOOD, KNOWN\_BAD, WATCH\_LIST, WHITE\_LIST, THREAT and HAZARD. These are set based on data analysis, correlation or end-user intervention.

+-------------+---------------------+------+-----+

| Field | Type | Null | Key |

+-------------+---------------------+------+-----+

| Tracking\_ID | bigint(20) unsigned | NO | PRI |

| Nugget\_ID | bigint(20) unsigned | NO | MUL |

| Data\_Type | bigint(20) unsigned | NO | MUL |

| Timestamp | datetime | NO | |

| Metabook | bigint(20) unsigned | YES | MUL |

| Src\_IP | int(10) unsigned | YES | |

| Dst\_IP | int(10) unsigned | YES | |

| Src\_Custom | varchar(255) | YES | |

| Dst\_Custom | varchar(255) | YES | |

+-------------+---------------------+------+-----+

Figure 6: Enterprise Tracking Table

The Custom\_flags field is a 32-bit field that is set aside exclusively for the use of organizations for in-house development. Flagging fields for departments (IT, Finance, etc…) may assist in behavioral analysis, partner IP addresses may be tagged and handled separately. 32-bits are available for any implementation that will support an organizations needs.

VII. Conclusion

The Razorback system was initially developed to address the specific problems that were posed by detection problems that required a level of computation time that was not compatible with real-time network systems. As the team worked on the architecture and in discussion with high-end incident response teams, it became apparent that the appropriate architecture was a framework. This decision was directly motivated by the frustrations and requests from these incident response teams.

The framework concept provides an opportunity for both Sourcefire and the Open Source community to work together to provide innovative solutions to emerging security problems. The capabilities provided by these two groups, and the ability to develop enterprise specific detection systems, are the key benefits of the Razorback System.

Find more about the Razorback system at:

<http://labs.snort.org/razorback/>

Appendix A. Razorback Development Tool

Architects

Lurene Grenier

Matthew Olney

Lead Programmers

Patrick Mullen, Data Capture & Analysis

Ryan Pentney*, Dispatch & Networking*

Programmers

Monica Sojeong Hong, *Flash Analysis*

Alex Kirk*, DNS Capture & Analysis*

Christopher McBee*, PDF Analysis*

Alain Zidouemba, *ClamAV Integration*

Database Engineering

Alex Kambis, *Hadoop Integration*

Infrastructure Support

Kevin Miklavic

Integration & Documentation

Nigel Houghton

Appendix B: Acknowledgements

Special thanks to the following organizations who provided feedback and suggestions.

Lockheed Martin Incident Response Team

General Electric Computer Incident Response Team

In particular, I’d like to thank Mike Cloppert and Charles Shmutz from Lockheed Martin, whose feedback on needed capabilities was greatly appreciated.

Also Rainbow boss for lots of rope and resources.