Laika BOSS: Scalable File-Centric Malware Analysis and Intrusion Detection System

Matthew Arnao Charles Smutz Adam Zollman Andrew Richardson Eric Hutchins
Lockheed Martin Computer Incident Response Team

1 Introduction

Threat actors intent on gaining access to a network often choose file-based exploits because they can be easily and reliably delivered to intended targets. These actors often use the most common, critical protocols such as email, web, and social media as delivery vectors, and target widespread and critical applications. Wholesale blocks on those protocols or file types would cripple legitimate business activity and is generally not an option for network defenders. To defeat intrusions, defenders must be able to detect malicious files wherever they exist - either transiting a network or stored on disk.

There are a multitude of malware analysis tools and reverse engineering resources available to analyze malicious code, but these work best in one-off, isolated conditions and are not capable of real-time processing. As a result, most security teams have to manage a disparate set of analysis tools with different capabilities. This inefficient solution presents a frustration for many defenders: being able to detect malware in a lab, but not able to scale that approach to successfully detect malware and defend an enterprise.

Most intrusion detection systems are focused primarily on the medium they monitor (e.g. network-based, host-based). The medium-centric approach normalizes all collection, logging, and alerting around the medium. File features - with all their different formats, data structures, and metadata - are left as secondary concerns, or worse, neglected altogether. Furthermore, files often encapsulate other files, embed related objects, or have format-specific encodings which need to be deconstructed to identify the underlying characteristics. A modular approach to file analysis enables such an understanding. We propose a file-centric intrusion detection approach, allowing connectors to various network and endpoint devices which can disassociate the medium and any file encapsulation. A file-centric intrusion detection system, therefore, will scalably interpret, parse, and normalize files to expose malicious characteristics.

2 Related Work

Other projects embrace a modular, file-based approach - most notably YARA [1] and Viper [2]. YARA is the de facto standard signature language to identify and classify malware. YARA 3.0 includes a framework to develop C-language modules that can interpret a data structure and expose specific metadata fields back to the YARA signature language. This is a powerful addition to YARA, but doesn't expand its scope beyond a standalone scanning language.

Viper is a framework to conduct malware analysis in a repeatable fashion using a modular approach and saving metadata of each object into a database. It enables analysts to organize their specimens into projects, develop modules that perform common tasks, and search the metadata database for related specimens. Viper is a tool for directed analysis, requiring manual invocation of modules on a specimen; it is not intended to execute recursively through a set of modules when given a set of criteria. We expand on this concept with a rule-based dispatcher, enabling large-scale automated analysis.

Cuckoo Sandbox [3] is a popular and powerful dynamic malware analysis system, and a great example of a feature-rich tool that many security teams use in one-off situations. It too has a modular architecture and allows analysts to insert YARA signatures to process files and child files. Cuckoo's primary limitation is scale - it can't both be a great malware analysis tool and an enterprise IDS. Our approach focuses on static analysis first, then dynamic analysis second, enabling scalability and increased transparency.

3 Our Approach

The Lockheed Martin Computer Incident Response Team (LM-CIRT) has developed and deployed the Laika BOSS^{TM}: Object Scanning System a file-centric intrusion detection system and malware analysis platform that can operate at the scale needed by a Fortune 100 enterprise. Laika implements the core functionality of dispatching input files to modules that typically perform extraction of subfiles, metadata collection, and/or detections such as signature matching. Modules disassociate the different network protocols, encapsulations, and obfuscations which commonly frustrate detection, leaving analysts free to focus on the payloads. Laika supports input from various sources including analyst submissions, passive network sensors, and inline network gateways. Analysts are able to generate their own modules and can do so with a simple Python API.

Scalability is made possible because collection and analysis can operate and scale independently. Decoupling collection from analysis provides the flexibility to integrate the analysis engine via an API into core infrastructure such as email gateways and web proxies, and the analysis cluster is horizontally scalable to accommodate any collection load. The recursive, module-based Laika framework is written in Python [4] and YARA and is easily extensible: defenders can write content once and apply everywhere, e.g., one module to analyze all Adobe Flash files, regardless of whether they are in email attachments or downloaded from the web. Conditional module dispatching logic allows files to be submitted in a production mode or in a test mode; in other words, the lab and enterprise are the same, providing defenders significant agility to respond quickly and comprehensively to new threats. This enables scalable content creation where analysts can easily develop, test, and deploy new capabilities.

Laika identifies, extracts, and normalizes metadata for each object it processes. As root objects are analyzed and sub-objects extracted, additional metadata becomes available which reveals indicators of various phases of the kill chain. Repeated metadata may indicate a common adversary, infrastructure, or toolkit. For example, malicious files often share repeated metadata patterns: detect the metadata, and you detect malware; log the metadata, and you can search historically to detect old malware after the fact; train a machine learner with metadata of good and bad files, and you can detect new malware that is similar to old malware. By grouping threat activity into threat campaigns and malware by common toolkits, defenders can understand the repeated behaviors and patterns and build resilient, durable defenses. This ability to take action on repeated observations enables effective Intelligence-Driven Defense [5].

4 Design

In designing Laika, we identified three high level goals: scalability, flexibility, and transparency. These key tenets are the foundation for a system that can grow and adapt to meet the needs of a demanding computer network defense mission. As the number of sources and volume of data increases, Laika needs to be able to scale. As new threats emerge, Laika facilitates the creation of new, easily developed and deployable content. This content, which may include code, signatures, and custom configurations, should allow for the uniform processing of files from all sources. Finally, Laika provides transparency through production of metadata to enable analytic completeness and understanding.

Scalability in Laika means processing huge volumes of files, or objects, from a variety of sources. Objects can be normal files passed in to the system or child objects derived from a parent, and will consist of metadata attached to a payload. To deliver these objects into Laika, both synchronous and asynchronous interfaces are available. The synchronous interface is useful for clients that are in-line with delivery of a payload such as a Milter or an ICAP server, while the asynchronous interface is useful for clients handling a high volume of data that do not require immediate feedback such as a passive network sensor or analyst submission. These clients are made accessible through a common Python client library, or may be written in other languages by utilizing a built-in JSON client interface. Horizontal scalability is achieved through a broker-worker model built on top of the lightweight/cross-platform ZeroMQ[6] messaging framework. Additional workers and brokers can accommodate virtually limitless volumes of data and clients. Figure 1 shows the Laika architecture with multiple clients, brokers, and scanner threads.

Flexibility is achieved through a modular architecture, highly configurable dispatching logic, and opportunities for tactical code insertion. The Laika framework is written in Python and has a simple module API that allows for the rapid creation of new capabilities based on the needs of analysts. Modules are written in

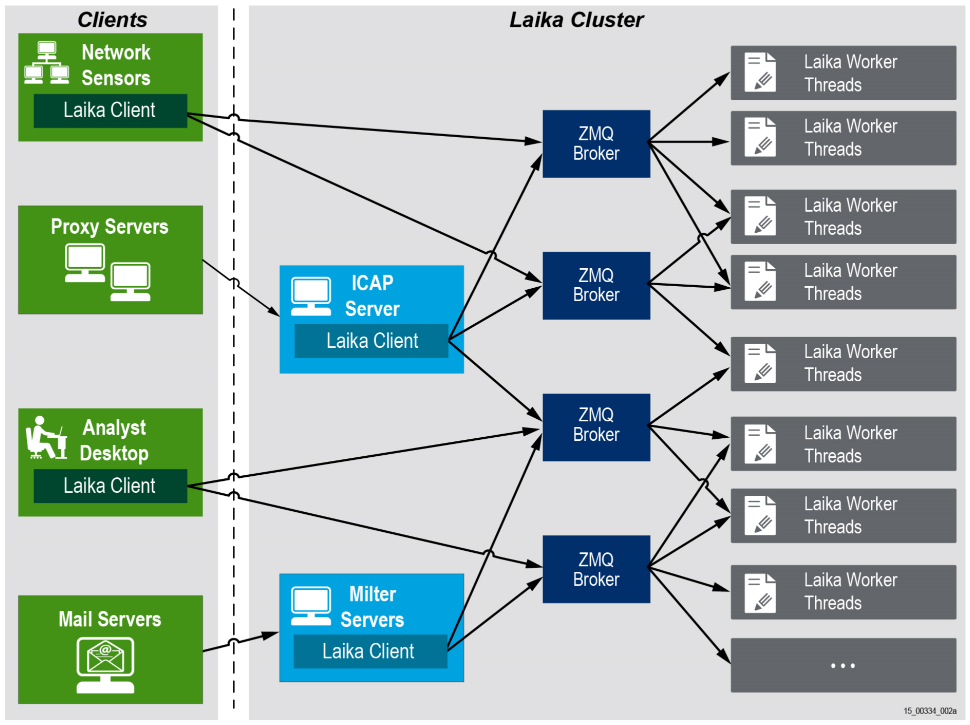

Figure 1: Laika System Architecture

Python and executed based on dispatching logic written in YARA. At a high level, most modules perform one or more of the following actions: add flags (detections) to an object, add metadata to an object, or explode new subobjects for scanning from the parent. The scanning process is made simple and predictable by running each object in a single thread for any specified configuration. Tactical code insertion (executing an arbitrary program on any file) is achieved by using the TACTICAL module, developed specifically for this purpose. TACTICAL integration with virtually any application regardless of the language in which it is written. Some modules serve a special purpose such as logging metadata to an external system or determining the disposition (scan outcome) for a given file.

Laika provides transparency through the production of an extraordinary amount of metadata about the objects it processes. While modules are responsible for the vast majority of metadata produced within Laika, the framework generates a basic level of metadata (UUID, parent/child relationships, file hash, file size, file type, module execution, run times, etc.) for each file it processes. Throughout the execution chain of an object, all metadata and files generated by previous modules are made available to the current module. Without any modifications to the framework, a module could be configured to export data to another system for collection, alerting, or analysis. This rich data provides clarity of the actions taken within the Laika IDS and deep analytical insight into the true characteristics of files observed in the environment.

4.1 Module Dispatching

The Laika framework enables rapid creation and seamless integration of new content into a comprehensive process. The dispatcher controls when and how a module runs, enabling specific content to run when needed and limiting the amount of unnecessary processing. This happens in two separate passes, enabling chained module execution. Dispatcher instructions are provided through a collection of YARA rules that leverage free-form meta fields and external variables populated by the Laika framework. Many Laika modules can be little more than wrappers around existing Python libraries - the integration into a dispatching framework for automated analysis makes the library considerably more powerful and flexible. For example, a standalone library that can parse and deconstruct an OLE file is very useful by itself, however by making this a module, it can be automatically run on any OLE regardless of source (email, zip, RAR, etc).

Upon receiving a file, Laika collects basic metadata, and applies the first set of dispatching rules which have full access to the files contents. This allows for the detection and proper recognition of many common file types with a reliable signature.

The following shows some example dispatching rules:

```
rule type_is_email
{
    meta:
        // Space separated, ordered list of modules to run on objects matching this rule
        scan_modules = "META_EMAIL EXPLODE_EMAIL"
        // Define a file type for objects matching this rule
        file_type = "eml"
    strings:
        $from = "From "
        $received = "\x0aReceived:"
        $return = "\x0aReturn-Path:"
    condition:
        // To prevent unwanted recursion, exclude objects originating from the EXPLODE_EMAIL module
        (not ext_sourceModule contains "EXPLODE_EMAIL") and
        (($from at 0) or
         ($received in (0 .. 2048)) or
         ($return   in (0 .. 2048)))
}

rule type_is_zip
{
    meta:
        // Arguments may be passed to modules as comma separated key/value pairs
        scan_modules = "EXPLODE_ZIP(filelimit=1000)"
        file_type = "zip"
    condition:
        uint32(0) == 0x04034b50
}
```

The second rule implements dispatching of the EXPLODE_ZIP module on all files matching the 4 byte zip file magic. This dispatching rule passes a parameter to the module specifying the maximum number of files to extract from the zip archive. The second pass dispatcher is designed to provide an opportunity to run modules based on the results of the first pass of scanning. Instead of applying rules to the file contents, the second pass dispatcher applies a set of rules to the resulting flags that were appended to the object during the first pass. The list of resulting flags is a space-separated list that is appended to the file. An example use case that utilizes both dispatchers is automatic Remote Access Trojan (RAT) configuration extraction. Modules in the first dispatcher would detect the presence of the trojan then the second dispatcher would recognize this detection by its flag and run the appropriate RAT parser on the file.

The file analysis process is fully recursive – each object receives the same two passes of dispatching treatment. Scanning is always performed depth-first within Laika: all modules called in the first pass of dispatching will complete and all child objects will be fully processed before the second pass dispatcher runs on the current file. Module execution order can be customized within the dispatcher if desired. Figure 2 shows the progression of a malicious email through the first and second dispatchers as well as all dispatched modules.

The Laika framework provides extensive protection against faults that may occur during scanning. Modules are wrapped with error handling to prevent a single module error from obstructing the rest of the scanning process - including multiple levels of configurable timeouts at both the module and overall scan level, as well as a configurable depth limit to prevent infinite recursion. Figure 2 shows the dispatcher recognizing the object as an email and passing it to EXPLODE_EMAIL where it is broken out into metadata, headers, body, and an attachment. The attachment is recognized as a ZIP and exploded, which yields an executable. Dispatching passes the object to SCAN_CLAMAV, as well as SCAN_YARA which in this case identifies a weaponization artifact. SCAN_YARA is run on the headers and body as well. After all first-pass dispatches are completed, second-pass dispatching follows in the reverse order, logging all metadata and summarizing the disposition - ultimately identifying the E-Mail as malicious.

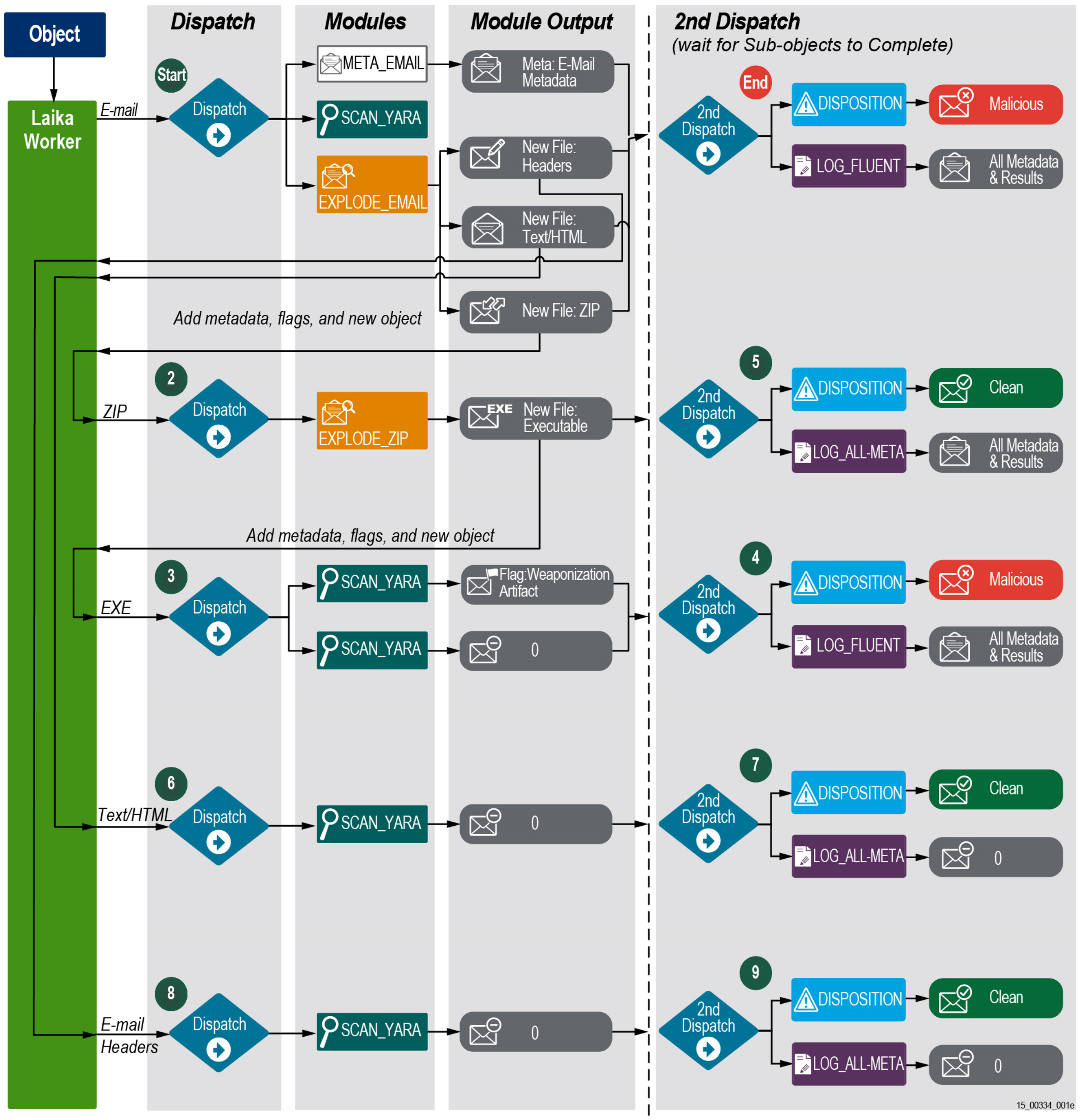

Figure 2: Laika Dispatch and Module execution

4.2 File Explosion Modules

One of the core functionalities of a file-centric IDS is file-format decoding. The ability to identify, explode, and recursively scan a root level object allows for all embedded objects and associated metadata within to be correctly extracted and parsed. Traditional NIDS on the other hand, tend to focus on the underlying transmission protocol instead of the file contents. Attackers easily defeat these systems by encapsulating their payloads within containers such as zip, rar, cab, and zlib, which add a layer of abstraction between the protocol and the payload. Our file-centric IDS is protocol agnostic; as long as a protocol is reliably defined, implementers can cheaply create content to parse the protocol and enable the reliable extraction of embedded objects and associated metadata.

Once a file format is understood and defined, the Laika framework will depth-first recurse through all of the identified sub-objects. The concept is to identify a root level object and recursively extract child objects as they exist. This recursion enables two key paradigms of file format decoding:

1. All objects should be normalized into a plaintext representation.
2. The system only makes a decision about an object after it has made a decision about all child objects.

The first paradigm enables reliable metadata extraction of each child object, while the second paradigm enables a complete set of information to inform any decision made by the dispatch or disposition logic. A file-centric IDS will declare a parent object to be malicious if it contains one or more malicious child objects.

The following example compares the differences between the Laika IDS file-centric approach and the NIDS medium-centric approach. Figures 3 and 4 show how each approach would display an email that contains a zip attachment that contains an executable file. NIDS would observe a single TCP stream and analyze the underlying SMTP vector. The Laika IDS would extract all sub-objects and make additional determinations.

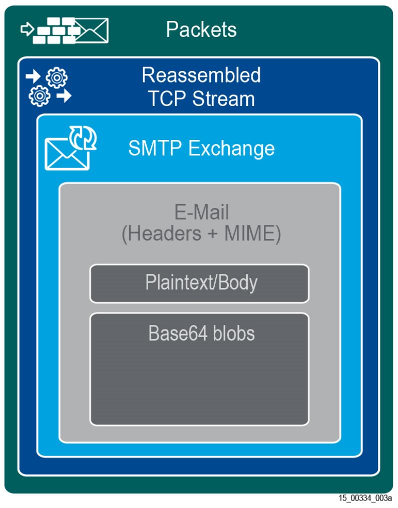

Figure 3: Traditional NIDS view of a malicious email with payload hidden through encoding

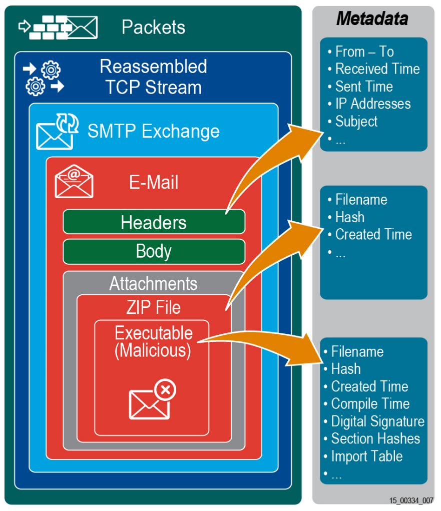

Figure 4: Laika IDS view of an exploded email, showing a malicious executable contained within a ZIP file and extracted metadata

In this example, the root level object is identified as an email. After this identification, the dispatcher will send the object to the EXPLODE_EMAIL module which applies a set of transformations specific to emails to decode the MIME encoding and extract the headers, body text and/or HTML, and the attachment. The dispatcher recognizes the attachment is a zip and executes the EXPLODE_ZIP module; which then extracts the inner executable. Other modules then normalize it, and identify it as malicious. At each step of the process, the dispatcher also sends each object to modules designed to collect metadata from the identified file types. After the child executable object is determined to be malicious, that designation is applied to the root object for final disposition.

4.3 Metadata Extraction Modules

As objects pass through Laika, META modules extract defined metadata attributes about each object. Object metadata provides rich contextual information used for analysis, correlation, and detection. Some examples include the compilation time of a Portable Executable (PE) file, file hashes, graphic resolution, number of pages, or document author. These attributes can be used to link together multiple instances of adversary activity or tools, and provide indicators at multiple phases of the kill chain. Malware authors have found that it is trivial to change small parts of a payload to evade commodity detections, so the ability to extract metadata of constituent sub-objects of a payload gives many additional opportunities to identify reuse of a component at any depth.

Looking at the extremely wide variety of filetypes that may be of interest to a file-centric IDS, it is apparent that there are many different types of metadata applicable to different formats, and no single schema for meaningful attributes. The flexible nature of the dispatcher allows META modules to be correctly run against instances of a given format and extract the attributes relevant to that format. Also, while the Laika Framework doesn't impose any schema on the extracted metadata - all fields are represented in a Python dictionary - we have found it valuable to additionally assign some commonly-reused fields such as document author to specific keys in the dictionary (regardless of original attribute name) to facilitate easy search and retrieval on those fields. The implementation of that taxonomy for known values, combined with the support for arbitrary metadata fields per object and object type, enables efficient analysis while retaining maximum flexibility.

Metadata extracted from an object in one module can also be accessed through other modules run in the recursive sequence, allowing complete inspection of metadata across the entirety of objects exploded from a single root object passed into Laika. META modules can be tailored to extract key-value pairs in different ways for different objects and a single ruleset which operates on those values to be abstracted from the complexities of multiple representations.

Universal analysis of metadata is made possible through the use of modules which allow export of all extracted fields and module-created information (such as additional flags) to a common logging system. The LOG_FLUENT module is provided to allow collection and storage of all this data through the Fluentd [7] data collector, enabling further analytical use cases. Any attribute which is logged can be subsequently searched to identify forensically whether an object did or did not appear, then also used to find related artifacts and identify previously unknown badness or indicators of adversary operations.

4.4 Detection Modules

The Laika modules most similar to traditional intrusion detection systems are those that perform some type of scanning - seeking to identify known malicious or otherwise anomalous attributes of the exploded file content. These detection modules can take many forms, limited only by the imagination of the creator and the processing power required. As examples:

- SCAN_YARA, a wrapper for YARA which runs all rules in the specified ruleset against the object. With wide availability of publicly available YARA rules, and the same syntax used elsewhere, this is a cornerstone of Laika's ability to identifiy malicious content wherever it exists within a payload, whether unobfuscated, or obscured within many layers of compound object formats.

- SCAN_CLAM, a wrapper for ClamAV [8] allowing easy access to pre-compiled, commodity antivirus signatures in the same platform used to leverage more in-depth and environment-specific malware detection capabilities.

- SCAN_PDFRATE, a non-public implementation of the detection capabilities described in PDFrate [9]. Users can implement proprietary anomaly or machine learning based detections. The object-based dispatching of Laika limits execution of these mechanisms to only the applicable files.

- BLACKLIST_MD5 can flag objects based on known MD5 hash. Since this can be run on objects at any level of dispatching, known malicious objects even amidst layers of packaging and obfuscation can be identified.

Modules empower content developers by inspecting each root object and the various child objects in a consistent format. In the previous example of an executable inside a zip inside an email, a YARA signature meant to detect the malware family can be separate from a signature to detect a malicious email address. The opposite also holds true - any given signature can be run against all root and sub-objects to provide significant coverage at each level of explosion regardless of what obfuscation or packaging was used. Another core feature of Laika is that content is easy to create and implement in YARA and Python, both already commonly used by security analysts and developers, which enables agile development of Laika content to contribute to real time response.

Every module, upon identifying content of interest, can output zero or more flags. Flags are atomic descriptors of the module results which can be provided to other modules or back to the calling client. Flags are analogous to a signature name in traditional NIDS or anti-virus. Flags may not contain all information about how the match was determined, but are sufficient to uniquely identify the type of detection that occurred.

As an object may have multiple flags identified and flags, on their own, do not attach any particular effect or action, there's no constraint to when a module should add one. Flags can provide tags (such as the case of "zip:err:LIMIT_EXCEEDED" in EXPLODE_ZIP) that are informative, but not necessarily indicative of malicious activity, without compromising ability to distinguish more critical flags. In addition to modules, flags can also be set within the dispatch logic. This can allow identification of the object source, or can provide indications of decisions made at dispatch time without having to reprocess the content through an additional module.

Once set by a module, flags associated with an object can be accessed elsewhere within the Laika framework. One use for this easily accessible collection of flags is in the dispositioner. The dispositioner is a module dedicated to inspecting the flags that have been set on an object and all children exploded from it. This allows a decision to be made as to the nature of the object. In effect, this creates a "signature of signatures" detection, where complex patterns of lower-fidelity flag matches can be combined to yield a higher-fidelity detection, and ultimately provide a response back to a real-time client allowing it to take action on the object. An example would be a Milter client accepting "DISCARD" and "REJECT" outputs from Laika to evict undesirable messages from the mail stream. A disposition rule file to accomplish that might look like:

```
//Example disposition configuration used for filtering email
//REJECT messages which are detected as malware by ClamAV
//DISCARD silently messages which have 3 or more flags (separated by two spaces)
rule DISCARD {
    strings:
        $space = " "
    condition:
        #space >= 2
}
rule REJECT {
    strings:
        $clam = "cl:"
    condition:
        $clam
}
```

The second-pass dispatcher can also inspect flags set on an object and trigger further action, such as more in-depth analysis of objects that have been identified through some cursory process as anomalous but need further inspection. As an example, a more processing-intensive YARA scan to identify the specific variant of a common malware family could be second-pass dispatched based on the flag set by an initial less intensive YARA rule which only identifies a general family. This allows a high degree of specificity and enables reliable detection and identification of many different types of malicious content.

5 Case Studies

Case studies are provided to demonstrate how the capabilities provided by Laika apply to computer network defense operations. These case studies are taken from in the wild attacks observed by LM-CIRT. The examples are intentionally taken from older activity to prevent disclosure of operationally sensitive information.

5.1 Reliable Inline Capability

Prior to the deployment of Laika at Lockheed Martin, email mitigations requiring more sophistication than simple signature or indicator based blocks were performed passively. Detections required a race between analysts and users to prevent successful intrusions. The load distribution capabilities, built-in fail-safes, and extensive debugging functionality of Laika allowed LM-CIRT to implement it as a blocking system with confidence. After integrating with email gateways, the first email blocked was an APT spear phish which contained a malicious document named "Why Democracy is Still Winning.pdf". This email would not have been blocked by existing static signatures, but was only blocked functionality embedded in Laika modules, including a machine learning based classifier, PDFrate. These in-house detection mechanisms would be prohibitively difficult to integrate directly in a COTS email gateway.

5.2 File-Centric Decoding

During the summer of 2012, targeted phishes began using the Adobe XML Data Package (XDP) [10] to obfuscate malicious PDF. XDP files are XML files containing base64 encoded PDF documents. Due to the encoding and encapsulation of the XDP format, the embedded malicious PDFs evaded typical analysis. Signatures and other detection mechanisms were not applied and metadata was not extracted from the PDF documents as intended. Researching the file format, it was discovered that it is straightforward to extract and decode the PDF payload. It was also noted that this format was used for benign purposes, albeit relatively infrequently, making blocking the XDP format altogether impractical. Therefore, a simple file decode module, named EXPLODE_XDP, was created to extract the PDF documents from XDP files. This module required about 20 lines of module specific Python code and required a few hours to create and test. A YARA signature was created to identify XDP files and EXPLODE_XDP was configured to run on XDP files. This module and configuration allows Laika to retrieve the PDF documents out of XDP files and operate on them the same as all other PDF files. This example highlights the ease of user generated file decoding modules - only a minimal amount of development is required to parse out the encoding unique to the XDP format and provide the content back to the dispatcher, where all subsequent detections are completely abstracted away from the XDP encoding.

The source code for the EXPLODE_XDP module [excluding the skeleton code common to all modules] is as follows, demonstrating how a trivial transform can yield significant new detection possibilities:

```
if 'element_names' in args:
    element_names = args['element_names'].split(';')
else:
    element_names = ['chunk']
# Parse the xml
dom1 = parseString(scanObject.buffer)
for element_name in element_names:
    logging.debug("EXPLODE_XDP: trying %s element name" % element_name)
    chunks = dom1.getElementsByTagName(element_name)
    # Just in case there happen to be more than 1 element with the name chunk..
    for chunk in chunks:
        b64pdf = chunk.firstChild.nodeValue
        # Get rid of newlines
        b64pdf = b64pdf.rstrip()
        moduleResult.append(ModuleObject(buffer=b64pdf,
    externalVars=ExternalVars(filename='e_xdp_%s' % element_name, contentType="base64")))
return moduleResult
```

5.3 File Centric Metadata

Late in 2012, LM-CIRT received intelligence identifying a specific spear phishing-based attack. This wave was analyzed for indicators that could be used to pivot and find additional related attacks. Analysis determined that typical network transaction metadata, such as IP addresses or email addresses, were not useful in finding related waves due to adversary specific TTPs. However, some unique metadata were discovered in the malicious Microsoft Office attachment that were of value. The document had an “Author” of “Jerry” and a “LastAuthor” of “Windows User” indicating that the document was created by a user named “Jerry” and last modified by a user named “Windows User.” Since Laika is configured to extract metadata from all Office files, these metadata conditions were searched. Analysts discovered that, individually, both potential indicators were common in many benign documents - however, the intersection of the two authors was very rare. Three unique documents in separate emails were discovered. These related documents were all confirmed to be malicious and were part of attacks not previously known to be related. Performing a historical search of this data to vet the fidelity of this condition, it was simple to write a signature to detect this condition on future observation. This case study demonstrates that file-level metadata can be used in incident response similar to the manner in which network metadata is commonly used today.

6 Operational Experience

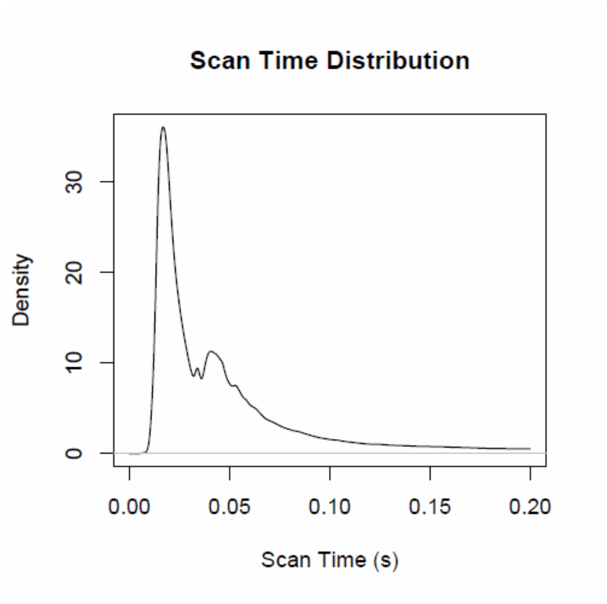

Figure 5: Scan Times Distribution

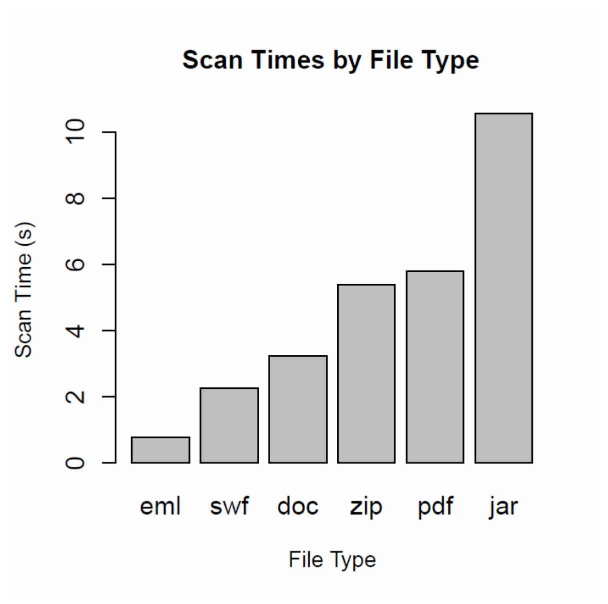

Figure 6: Scan Times By File Type

Laika has been deployed at Lockheed Martin since early 2012. Laika is integrated with network sensors based on Vortex [11] and also Suricata [12], scanning files extracted from HTTP, among other protocols. Inline blocking capability occurs through integration with email gateways and web proxies using Milter [13] and ICAP [14], respectively. Analysts also manually submit individual files for processing.

On a typical business day, this Laika installation scans approximately two million external emails totalling 150 GB of input resulting in 400 GB of data scanned post subfile explosion. Web traffic, which is also analyzed, weighs in at about 100 million web requests totalling about 1.5 TB of HTTP payload data daily.

Because the core components of Laika (ZeroMQ, YARA, etc.) are implemented in efficient native code, Laika is highly performant. Laika adds little computational overhead and latency beyond the processing performed by individual modules. Figure 5 shows the weighted distribution of Laika scan times from a typical day with over 100 million objects. Eighty percent of the scan times are under 100ms and 98.6% of the objects are scanned in under 1s. Since Laika is usually configured to tailor processing to the type of file being scanned, Laika scan times are very long tailed. Figure 6 shows the average scan types for some analysis heavy file types on the same data set. Portable executables, excluded from this graph, had an average scan time of 83s. Using simple queuing and distribution to independent scanner threads, average typical scan times are kept low.

Laika scales horizontally well. The largest Laika cluster known to the authors was built using 16 servers with 24 cores each, totaling 384 cores. Laika scanning services have remained highly available despite individual scanner node failures.

7 Conclusions

Laika BOSS is a file-centric malware analysis and intrusion detection system. It implements the core functionality of dispatching input files to modules. These modules typically perform extraction of subfiles, metadata collection, and/or detections such as signature matching. They abstract away the different network protocols, encapsulations, and obfuscations which commonly frustrate detection, leaving analysts free to focus on the payload. Laika supports input from various sources including analyst submissions, passive network sensors, and inline network gateways. Analysts can generate their own modules and can do so with a simple Python API. This extensibility gives the defender significant power and flexibility to identify and defeat intrusion attempts that would evade other countermeasures.

Laika BOSS is built on popular open source projects such as YARA and ZeroMQ. In turn, the Laika IDS Framework is freely available as open source: https://github.com/lmco/laikaboss. The current collection includes the core framework including many modules and clients for Milter and ICAP. A client for network sensors, such as Suricata, will be available in the future. All functionality is supported at scale and has been running successfully in Lockheed Martin's global network since 2012.

References

[1] "YARA - The pattern matching swiss knife for malware researchers (and everyone else)." `http://plusvic.github.io/yara`.

[2] "Viper." `http://viper.li`.

[3] "Cuckoo Sandbox." `http://www.cuckoosandbox.org`.

[4] "Python." `https://www.python.org`.

[5] "Cyber Kill Chain." `http://www.lockheedmartin.com/us/what-we-do/information-technology/cyber-security/cyber-kill-chain.html`.

[6] "ZeroMQ." `http://zeromq.org/`.

[7] "Fluentd — open source data collector." `http://www.fluentd.org`.

[8] "ClamAV." `http://www.clamav.net`.

[9] "PDFrate - a machine learning based classifier operating on document metadata and structure." `http://pdfrate.com/about`.

[10] "XML Data Package Specification, version 2.0." `http://partners.adobe.com/public/developer/en/xml/xdp_2.0.pdf`.

[11] "Vortex IDS." `http://sourceforge.net/projects/vortex-ids`.

[12] "Suricata - Open Source IDS / IPS / NSM engine." `http://www.suricata-ids.org`.

[13] "Milter." `https://www.milter.org/developers/api`.

[14] "Internet Content Adaptation Protocol." `http://tools.ietf.org/html/rfc3507`.