Automating Static File Analysis and Metadata Collection Using Laika BOSS

Laika BOSS is a file-centric recursive object scanning framework developed by Lockheed Martin that provides automation of common analysis tasks, generation of rich file object metadata and the ability to easily apply file-based signature detections to identify malicious files through static analysis. While performing triage and analysis of malware, analysts typically perform repeatable tasks using a variety of standalone utilities and use these tools to gather information that will be useful in understanding adversary ...

Automating Static File Analysis and Metadata Collection Using Laika BOSS

GIAC (GREM) Gold Certification

Author: Chuck DiRaimondi, charles.diraimondi@gmail.com
Advisor: Christopher Walker
Accepted: February 14, 2018

Abstract

Laika BOSS is a file-centric recursive object scanning framework developed by Lockheed Martin that provides automation of common analysis tasks, generation of rich file object metadata and the ability to easily apply file-based signature detections to identify malicious files through static analysis. While performing triage and analysis of malware, analysts typically perform repeatable tasks using a variety of standalone utilities and use these tools to gather information that will be useful in understanding adversary tools and in developing future detections. This paper will provide guidance to analysts by reviewing concepts core to the Laika BOSS framework, integrating custom Yara rules for file-based detections, searching and filtering scan object metadata, and describing how to develop, test and implement new Laika BOSS modules to extend and automate new functionality and capabilities into the framework. As part of performing this research, new modules and tools will be released to the security community that will enhance the capabilities and value obtained by using the Laika BOSS framework to perform static malware analysis and metadata collection.

1. Introduction

Laika BOSS is a "file-centric intrusion detection system" (Hutchins, Cloppert and Amin, 2009), open-sourced by Lockheed Martin that recursively analyzes file objects by using a modular framework to automate common analysis processes used by analysts to determine if a file is malicious. There are capabilities within Laika BOSS that allow analysts to drive the detection and metadata collection capabilities of objects during various stages of the Cyber Kill Chain such as the identification of weaponization techniques and delivery of malware. Analysts can utilize the Laika BOSS framework functionality and metadata stored during the analysis of objects to better understand the tools that threat actors are using to carry out their mission.

The successful use of Laika BOSS at different stages of the Cyber Kill Chain can provide immediate value to an organization as it helps automate common analyst tasks associated with file and object analysis. During analysis and documentation, it is common for analysts to review various characteristics and properties of a file, both statically and dynamically. While analysis using both methodologies is a common standard and practice, automating the static analysis of files and collection of analysis metadata can greatly help increase the efficiency and accuracy of analysis.

Throughout this research, new modules and tools have been developed that will showcase how analysts can gain additional value out of Laika BOSS by integrating it into their analysis workflow. Case studies will be reviewed to help solidify various discussion points presented throughout this research paper. The intention of this research paper is to understand the benefits of using the framework, how to implement, configure and enhance the framework and how to best utilize the metadata collected throughout the analysis process as a means to analyze and hunt through historical scan metadata.

1.1. Benefits of Using Laika BOSS

Analysts commonly perform various tasks as part of analyzing files or objects. Specific to malware, these tasks are typically broken up into phases, commonly known as static and dynamic analysis. Each of these phases can be broken down further to include both basic and advanced analysis that requires varying skill sets in order to complete analysis successfully (Sikorski, 2012). Many of these tasks are similar in nature and may

involve substituting different tools based on the type of analysis being conducted. Most of the time, the acquisition of data and knowledge as part of analysis is fairly consistent.

One of the benefits of utilizing Laika BOSS is that it enables analysts to perform consistent and accurate static file analysis against objects, whereby a common set of data is collected based on the file type being analyzed. As will be discussed in detail later, Laika BOSS can be configured to analyze objects of a similar type the same way and to collect the same type of metadata. This static analysis capability can be used as a signature detection platform to help identify and classify specific threat actor tools. The type of intelligence gathering specific to tools, as depicted on the Pyramid of Pain in Figure 1, may be more complex depending on the skillsets that exist within an organization but can greatly aid in the understanding of an attacker (Bianco, 2014).

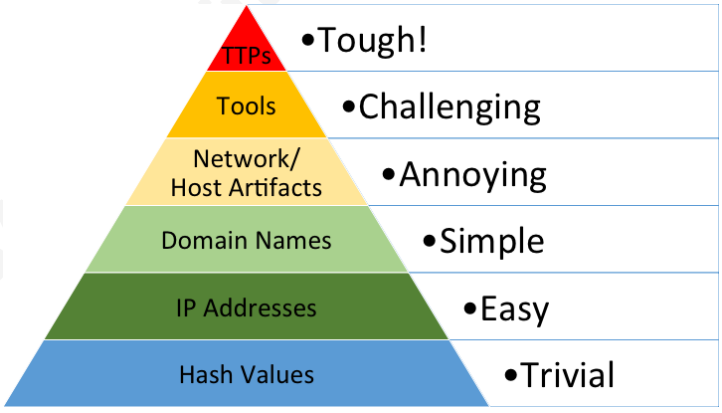

Figure 1 – Pyramid of Pain

Another benefit is the ability to extract and gather file object metadata. This metadata can then be stored for future analysis and searching. By storing the object metadata, this enables an analyst to answer questions such as, “Have I ever seen a file with the same MD5 hash before?”, “Have I ever seen or analyzed a Word document with the same Creator metadata value before?” or “Did I ever see another email analyzed that contained the same X-Mailer or upstream IP address value coming from a different email sender?”. Having access to this type of metadata to answer those types of questions can greatly enhance the ability for the analyst to successfully analyze a particular intrusion or identify a potential campaign based on adversary tactics, techniques and procedures (TTPs).

In Figure 2 below as referenced by Hutchins, Cloppert and Amin (2009), the reader can see the ability to identify common indicators across different activity as a method of potentially identifying related intrusion activity. As part of incident response and analysis, an analyst can search Laika BOSS scan metadata gathered by scanning objects such as emails and files from previous incidents to find indicator overlaps that can help identify potential campaigns. As depicted in the weaponization section in Figure 2, an analyst could have developed a Yara rule to identify an obfuscation technique unique to a particular piece of malware and could have also identified unique atomic indicators such as an X-Mailer or sending email address used by an adversary during the Delivery phase of their campaign. All of these can be placed in Yara rules and utilized by the Laika BOSS framework for detection purposes. The scan metadata can also be used to find overlaps in activity and malware that may not have previously been known due to a lack in detection capability at the time the events occurred. An example could be a weaponized Word document with a particular Creator name and code page value. Based on open source intelligence (OSINT) released in a recent vendor whitepaper, you develop a Yara signature to detect a particular type of malicious Word document. While developing the rule, the analyst performs historical searching within Laika BOSS scan metadata and identifies other documents previously sent in a few months ago that had the same metadata values. During the analysis it was also observed that the email senders were different than the ones being used in the vendor whitepaper. An analyst could then start to develop hypothesis to determine if this activity is part of a larger campaign and can use the newly found data as a source of intelligence in identifying a potential adversary.

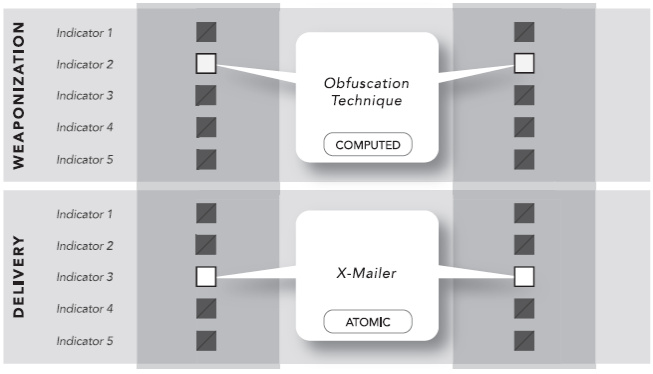

Figure 2 – Common Indicators Across Multiple Intrusions

Laika BOSS also provides a modular framework with the ability for an analyst to enhance the functionality of the framework by adding new object analysis and detection capabilities. These new capabilities can very easily be added to the framework and provide immediate value for analysts in attempting to analyze new attacker tactics and techniques. The customizable nature of the framework allows the analyst to change the analysis workflow based on file types and to specify different types of functionality they want run against those object types. A common technique for attackers during the delivery phase of the Cyber Kill Chain is to weaponize a Microsoft Office document with malicious macros. A common analysis task is to utilize a tool such as olevba (or similar) to extract out the Visual Basic for Application (VBA) macros and to review them in order to identify patterns of maliciousness. An example of enhancing the framework could be adding the capability to automatically extract and decompress VBA macros from a Microsoft Office document by using an EXPLODE module (to be discussed later) to process OLE objects from Microsoft Office documents. An analyst can easily develop a new module utilizing existing python libraries to extract and decompress the VBA objects. These are now new objects that can be scanned by the Laika BOSS framework using Yara to detect malicious VBA code. Figure 3 shows object analysis flows before and after implementing a new EXPLODE_VBA and META_OLEVBA module, including the detection of malicious VBA macros due to SCAN_YARA executing.

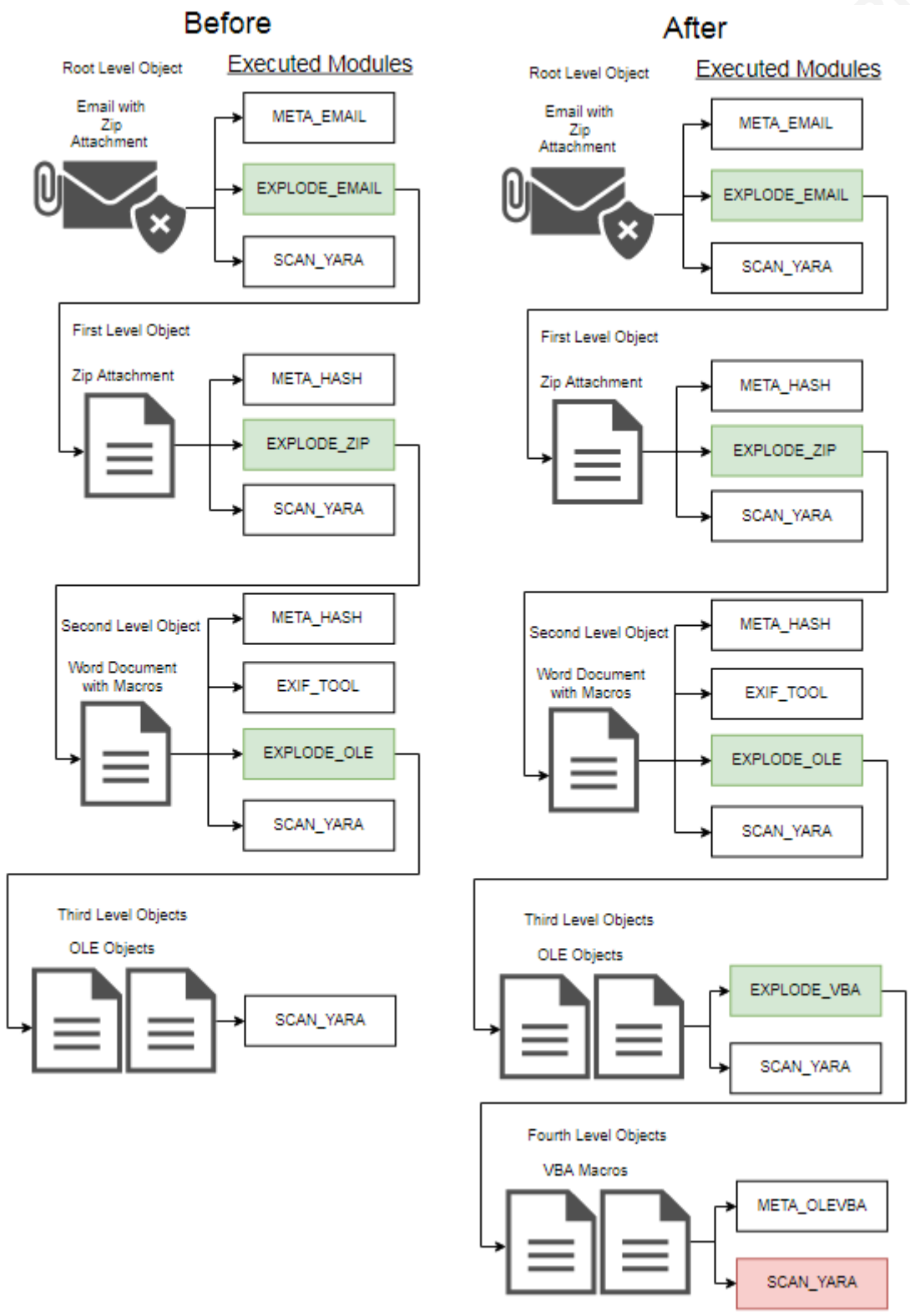

Figure 3 – Object Analysis Workflow

1.2. Operationalizing Laika BOSS As An Analysis Tool

Laika BOSS can be utilized in different ways depending on the requirements of the security organization and the resources available to implement and support such an environment. For the purposes of this paper, Laika BOSS will be described in the context of an analyst tool that can accept files passed to it on the command line by using one of the delivered utilities such as `cloudscan.py` or `laika.py`. This would typically be done when an analyst obtains an email, file or other object that they would like scanned for initial triage and analysis. Laika BOSS would be running as a daemon process by having executed `laikad.py` on the server hosting Laika BOSS. Figure 4 shows a typical configuration.

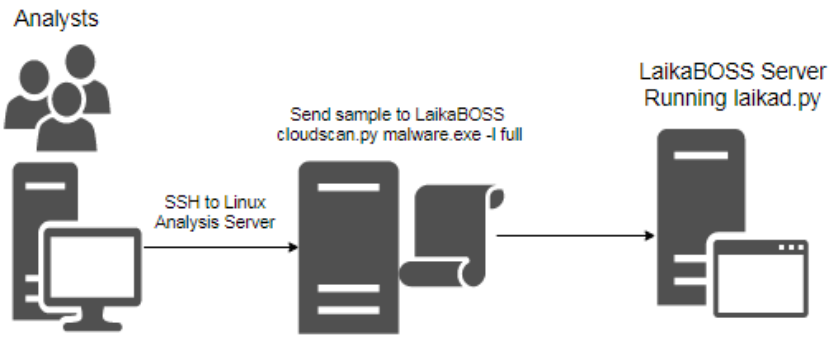

Figure 4 – Analyst Submitting Sample Using cloudscan.py to Laika BOSS Server

In addition to using the standalone utilities to send objects to the scanner on an ad-hoc basis, Laika BOSS can be used as a scanner against other network streams that contain SMTP or HTTP traffic in order to extract and scan objects. Configuring and integrating Laika BOSS in this way is beyond the scope of this research paper. The value of integrating the scanner into email and web traffic is its ability to extract objects from emails or web traffic and inspect them using various modules and Yara rules in order to identify malicious objects. One such example would be its ability to identify an email stream, extract out all attachments and recursively analyze them. An example of sending an email into the scanner will be discussed later in this paper. For additional information on configuring Laika BOSS for email and web traffic analysis please reference the `laika_redis_client.py` script on LM Laika BOSS GitHub and the work done by Josh Liburdi who integrated Laika BOSS and Bro (Liburdi, 2017).

1.3. Prerequisites for Utilizing Laika BOSS

In order to obtain the most value from Laika BOSS, it is recommended that analysts are familiar with Python, Yara, jq and Linux. While you don't have to be an expert in using any of these technologies, having a familiarity with all of them will help increase the immediate value obtained by using Laika BOSS. Lastly, a drive to develop new modules and write Yara rules during your malware analysis is all you need!

For this research paper, Laika BOSS was installed on Ubuntu 14.04 in a virtualized environment using 1024 MB of memory and 1 CPU. In order to store the scan result output, MongoDB 3.6.1 was installed on the Laika BOSS virtual machine. The `lbq.py` tool was developed to access the scan result data stored in MongoDB. There were various indexes created against different scan result fields in the MongoDB. The commands used to perform the index creation are listed in Section 10.2 Appendix D – MongoDB Indexes for lbq.py. The default MongoDB database created by LOG_MONGO, which will be discussed later, is named laikaboss.

2. Framework Overview and Configuration

2.1. Laika BOSS Framework

There are multiple files associated with the framework that control its configuration and functionality. The framework itself uses Yara as its configuration language and Python to provide the module functionality. Yara is also used as a signature detection mechanism across any object during analysis. The default installation of Laika BOSS as described in the Laika BOSS documentation results in the main installation of configuration files being located in `/etc/laikaboss`.

```
[/etc/laikaboss]
analyst-> ls
      cloudscan.conf  conditional-dispatch.yara  dispatch.yara  laikaboss.conf
laikad.conf  modules
```

The three .conf files listed above are responsible for setting configuration options for Laika BOSS, cloudscan and the Laika daemon process. The modules folder contains the `disposition/disposition.yara` file and a `scan-yara/signatures.yara` file. These files are responsible for configuring the disposition and Yara rules.

All of the module code itself is stored in the python installation directory `/usr/local/lib/python2.7/dist-packages/laikaboss-2.0-py2.7.egg/laikaboss/modules`. Any new modules developed will get placed into this directory. More details regarding this process will be discussed later when reviewing module development and implementation.

There are different module categories depending on the type of functionality required. These module types are META, EXPLODE, SCAN, EXTRACT, DECODE and TACTICAL. The META modules capture metadata for an object such as file hashes, file metadata, object properties and much more. Examples include META_HASH and META_PE. EXPLODE modules will take an object and extract out any identifiable sub-objects that exist within. These extracted objects are then typically submitted into the scanner for analysis. Examples include EXPLODE_ZIP and EXPLODE_RAR. SCAN modules are used to detect something about an object, typically for identifying malicious characteristics. Examples include SCAN_YARA and SCAN_CLAMAV. EXTRACT modules will extract a specific piece of data from an object. An example module would be one that extracts links from emails. DECODE modules are responsible for transcoding data from one format to another. One example is the DECODE_BASE64 module that decodes base64 character encoded data. DECODE modules can also be utilized for identifying encoded data within malware backdoors. TACTICAL modules are typically used to take advantage of existing scripts or tools that are present on the scanner server that contains functionality already required.

2.2. Configuration Settings

Laika BOSS includes multiple configuration files that are responsible for controlling the functionality of delivered framework tools as well as altering the ways in which the framework identifies and detects malicious objects. There are six main configuration files in Laika BOSS that help control the core framework functionality and delivered tools: `laikaboss.conf`, `laikad.conf`, `cloudscan.conf`, `dispatch.yara`, `conditional-dispatch.yara`, and `disposition.yara`.

2.2.1. laikaboss.conf

The `laikaboss.conf` file is the main configuration file that controls core framework settings. The first section includes general settings responsible for specifying the location of dispatch configuration files, global module timeout settings and the file system location for storing temporary files. The second section specifies the location of Yara rules used by the SCAN_YARA and DISPOSITIONER modules. The last section allows you to configure various logging options. The section that you may want to configure, depending on the type of modules you are executing, are the global timeout settings. Some modules may take longer to run than others and you don't want scans to timeout. Finding the right global timeout setting can help you avoid a timeout issue.

2.2.2. cloudscan.conf

The `cloudscan.conf` file allows you to include configuration information of the remote Laika BOSS server if running the `laikad.py` daemon process. These options can also be set on the command line when executing `cloudscan.py` using the options "-s" and "-a".

2.2.3. laikad.conf

The `laikad.conf` contains two sections of configuration settings labelled Network and General. The network section contains settings for client and worker listening addresses and ports. The general section includes the location of the `laikaboss.conf` file and various framework timeout settings. One of the more important settings related to timeouts are those for the number of maximum objects to process before a worker shuts down and the number of minutes to accept new objects before shutting down. These settings are important to keep in mind because new Yara rules introduced into the framework will not be active until the time to live (ttl) settings, 10000 and 30, are met. If the timing of loading a new Yara rule is of importance, adjusting these settings may help meet your needs. Otherwise you can scan an object using `laika.py` which will cause all rules and configuration options to reload.

2.2.4. Dispatch

The dispatch settings are controlled through the `dispatch.yara` file located in `/etc/laikaboss`. The dispatcher is considered the “brains” of Laika BOSS. Contained in

the dispatch file are multiple Yara rules that help define and identify file types and what modules should be executed against those object types. Examples of object types defined by default are email, Microsoft Office 2003 and 2007+, PDF, RTF, ZIP and much more. For each of these type definitions are configurations specifying the modules and order of modules that should be executed should Laika BOSS see that object type during analysis. Each of these modules produces metadata or objects that can be used by the framework for detection purposes. Within dispatch are global variables that can be used for identifying when modules should be run. The variables are clearly defined in `dispatch.yara`. Some example use cases include only executing modules based on a specific file name or as a result of a parent module having been run. Examples of customizing dispatch settings will be reviewed in section 3 of this paper.

2.2.5. Conditional Dispatch

The conditional dispatch settings are defined in `conditional-dispatch.yara` and are located in `/etc/laikaboss`. Conditional dispatch is considered a second pass at analyzing the object and it gives the opportunity to log metadata or perform additional actions against objects that have flagged in previous analysis. LOG modules are typically used at this stage of analysis as all object analysis and metadata collection has been completed so all results can be logged. Dispositioning of objects also occurs during conditional dispatch and it is responsible for determining the outcome of an object scan. The DISPOSITIONER module uses a configuration file to determine how certain flag hits should be dispositioned. Lastly, DECODE and EXPLODE modules are also often typically run during conditional dispatch as they usually require the outcome of previous modules to complete in order to determine if they should execute. An example previously given was that of a backdoor that may include embedded configuration data. You need to first flag on the backdoor sample and then during conditional dispatch or the "2nd pass", run a DECODE module against the sample to pull out the configuration data. In the conditional dispatch configuration for this DECODE module, you would need to specify that it will only execute if it has a string match for the backdoor Yara rule.

2.2.6. Disposition

The disposition settings are defined in `disposition.yara` located in `/etc/laikaboss/modules/dispositioner/`. Dispositioning defines the outcome of a scan. It is here where you can classify certain flag hits as being the determining factor for if something is seen as malicious or even to identify different priority levels should flags hit. As an example, you may have written some backdoor Yara rules related to an advanced persistent threat attack at your company. You may want to classify those as CRITICAL severity hits should you scan an object that flags on those same backdoor rules. It is within the `disposition.yara` file that you can specify those settings. Example disposition buckets may be INFO, LOW, MEDIUM, HIGH, CRITICAL. Other buckets such as Opportunistic and APT may be viable options as well. If the scanner was a passive sensor on your network, extracting objects and automatically scanning them, these types of disposition settings could help your team prioritize alerts originating from Laika BOSS.

2.3. Tools

2.3.1. Laika.py

The `laika.py` script allows you to send a file to a local instance of the Laika BOSS scanner. A local Laika BOSS instance is one that is not accessible across the network. While there are multiple command line options for `laika.py`, the most commonly used options are "-d", "-m" and "-o". These options will be used during examples throughout this paper and explained in further detail.

2.3.2. Laikad.py

The `laikad.py` script is used to run Laika BOSS as a networked instance, allowing it to accept objects sent to it from across the network. When running Laika BOSS in this mode, analysts can configure and use `cloudscan.py` to send a file to Laika BOSS located on a different server. The various configuration options for `laikad.py` are located in `/etc/laikaboss/laikad.conf` and runtime options can be specified when executing the script. To look at these options run `laikad.py -h`.

2.3.3. Cloudscan.py

The `cloudscan.py` script allows the analyst to send a file object to a remote Laika BOSS instance that is run using the `laikad.py` daemon script. The cloudscan script itself offers different runtime options depending on the type of scan and output data you want. For a full list of options you can run `cloudscan.py -h`. Typically, analysts will execute `cloudscan.py -l full <filename>` to scan a file and will get in response the full scan result output in json. While having the scan results output in this format is valuable, you can also have Laika BOSS provide all extracted files and scan result output into a folder. In order to accomplish this, the "-o" option can be used to specify an output directory. This aids an analyst in understanding what objects were extracted from the submitted object. There are also situations where you may want to extend the default timeout that is specified in the Laika BOSS configuration files. The "-t" option can be specified to define, in seconds, the new timeout to be placed on scanning an object. This is useful if you are integrating Laika BOSS with a dynamic analysis environment and you are unsure how long that analysis will take to complete. In most situations this option does not need to be specified and the default value in the configuration should suffice.

3. Customizing Laika BOSS

The Laika BOSS framework provides a means for analysts to customize and adapt their analysis workflow as threat actors continue to change their TTPs, specifically regarding how they weaponize, obfuscate, and deliver their malware to targets. Laika BOSS provides easily adaptable customization options by allowing analysts to create and update dispatch rules, configure new Yara rules to be used for scanning and detection and develop new modules that expose specific functionality into the framework to perform static analysis.

3.1. Creating and Configuring Custom Dispatch Rules

The `dispatch.yara` file is considered the “brain” of Laika BOSS. The Yara configuration file allows for the identification of new object types and the definition of custom analysis workflows by using various modules provided by the framework. This also provides the benefit of being able to easily enable or disable functionality for

specific objects due to its configuration at the object level. Dispatch identifies objects typically by file magic and other properties, consistent with how Yara is typically used today. There are various customization options available that will be discussed in further detail such as the ability to set custom dispatch flags, alter module flow by setting priority values, specifying custom Yara rules for scanning based on object type, and providing custom module arguments that alter the functionality of a specific module.

3.1.1. Identifying New Object Types

Creating detections for new object types and altering the analysis workflow based on those object types is one of the core features of Laika BOSS. Threat actors continually change their tactics and techniques, requiring that analysts enhance the tools, techniques and methods used to analyze a threat. As we can see from the can results below, Laika BOSS doesn't come with a signature to identify .ISO files. The scanned .iso file hash `1edcdb0299aff6abc69d584321482561` below was obtained from Virustotal after reading an article posted by Didier Stevens on .ISO analysis (Stevens, 2017). The fileType key in the scanned json output has no value and as a result, Laika BOSS doesn't know the file type for this object.

```
[~/samples]
analyst-> laika.py 1edcdb0299aff6abc69d584321482561 | jq .
{
  "scan_result": [
    {
      ...
      "source": "CLI",
      "filename": "1edcdb0299aff6abc69d584321482561",
      "fileType": [],
      ...
```

To identify new object types in Laika BOSS, you need to create a new rule entry in `dispatch.yara` that uses common techniques for file type identification: file magic header bytes, file extension or strings within the object. After doing research on ways to identify .ISO files, it was determined that the file magic for .ISO files was `43 44 30 30 31` and can start at three different offsets within a file: `0x8001, 0x8801` or `0x9001` (Kessler, 2017). The Yara rule below was added to the dispatch logic and would trigger the execution of the EXPLODE_ISO module (see Developed Modules) should we have a signature match.

```
rule type_is_iso
```

```
{
    meta:
        scan_modules = "EXPLODE_ISO"
        file_type = "iso"
    strings:
        $iso = { 43 44 30 30 31 }
    condition:
        $iso at 0x8001 or $iso at 0x8801 or $iso at 0x9001
}
```

The snippet of json scan output below shows that the fileType key now has the value “iso”, indicating that Laika BOSS can now identify this as an ISO file object.

```
[~/samples]
analyst-> laika.py 1edcdb0299aff6abc69d584321482561 | jq .
{
  "scan_result": [
    {
      ...
      "source": "CLI",
      "filename": "1edcdb0299aff6abc69d584321482561",
      "fileType": [
        "iso"
      ],
      ...
```

We can also see that the type_is_iso rule hit in the DISPATCH section of the json scan output.

```
"DISPATCH": {
  "Conditional Rules": [
       "send_to_mongo (50)",
       "DISPOSITION_FILE (10)"
  ],
  "Rules": [
       "scan_Yara (9)",
       "meta_hash (9)",
       "type_is_iso (9)"
  ]
},
```

3.1.2. File Extension Matching

There may be situations during static analysis where identifying an object based on file magic just won’t work. If this situation arises, you can create a custom dispatch rule to look for a file name coming into Laika BOSS with a particular file extension and then execute specific functionality as needed. As analysts it is important to not solely depend on file extensions as a means to identifying a file type. In this situation we are going to assume that there is no reliable file magic value. As an example, let's say the JavaScript file with MD5 hash `47543e0f331b6511b1ea72480e9a499b` wants to be analyzed. The only thing that identifies it as JavaScript is the code inside the file and the

file extension ".js". The file is run through Laika BOSS and it is noticed that it wasn't able to detect the file type and didn't run any additional modules outside of the standard four that run on all objects.

```
[~/samples]
analyst-> laika.py 47543E0F331B6511B1EA72480E9A499B.js | jq .
{
  "scan_result": [
    {
       ...
      "source": "CLI",
      "filename": "47543E0F331B6511B1EA72480E9A499B.js",
      "fileType": [],
      ...
      "scanModules": [
        "SCAN_YARA",
        "META_HASH",
        "DISPOSITIONER",
        "LOG_MONGO"
      ],
       ...
```

While not always ideal, we can target the filename and look at the file extension to define the file type.

```
rule type_is_js
{
    meta:
        file_type = "js"
    condition:
        ext_filename contains ".js"
}
```

After running rerunning the file through Laika BOSS, it is now being detected as a "js" or JavaScript file based on reading the source filename.

```
[~/samples]
analyst-> laika.py -d 47543E0F331B6511B1EA72480E9A499B.js | jq .
{
  "scan_result": [
    {
       ...
      "source": "CLI",
      "filename": "47543E0F331B6511B1EA72480E9A499B.js",
      "fileType": [
        "js"
      ],
      ...
```

3.1.3. Custom Dispatch Flags

In the dispatch settings you can specify dispatch flags in order to provide additional context or for altering the workflow of analysis for an object. If the reader expands on the previous example of identifying a JavaScript file, the framework can also

be configured to identify when a JavaScript file is seen inside a zip file. In order to do this the reader will need to reference the type_is_js rule and also the ext_parentModules external variable.

```
rule js_in_zip
{
    meta:
        flags = "js_in_zip"
    condition:
        ext_sourceModule contains "EXPLODE_ZIP" and type_is_js
}
```

To test this new configuration option, a zip file was created that contains a JavaScript file and submitted to Laika BOSS. If the scan output is reviewed, the reader can see a new flag hit of "dispatch::js_in_zip" and the filename of `e_zip_47543E0F331B6511B1EA72480E9A499B.js` starting with "e_zip_" which states that it originated from the EXPLODE_ZIP module. Because EXPLODE_ZIP was the source module that produced the JavaScript file, a flag can be set stating that there was a JavaScript file in a zip file.

```
"flags": [
      "dispatch::js_in_zip"
],
...
"source": "CLI",
"filename": "e_zip_47543E0F331B6511B1EA72480E9A499B.js",
"fileType": ["js"]
```

One benefit of throwing a flag at dispatch is the ability to identify common techniques being used by attackers when delivering malware. As an example, you can use the `lbq.py` tool to search for a flag value of "js_in_zip". This would show you all instances of objects submitted to Laika BOSS where there was JavaScript inside of a zip file.

```
analyst-> python scripts/lbq.py -k flag -v "dispatch::js_in_zip"
{"rootUID": "4b635498-446f-4c03-9968-f3368363f06a", "fileType": ["zip"],
"objectHash": "1f15819f23fe3b26fb6035f744d85f0d", "flags":
["dispatch::js_in_zip"], "filename": "js_test.zip"}
```

3.1.4. Setting Module Priority

When performing analysis of an object, you may have the requirement to execute analysis tasks in a particular order and to ensure that some complete before others. The priority settings in `dispatch.yara` allow you to alter the order in which modules run on objects. For example, you have a module that extracts domain names from an object. In

order to extract the domain, a URL needs to first be decoded. In this example there may be modules named DECODE_URL and EXTRACT_DOMAINS. In order for DECODE_URL to always run prior to EXTRACT_DOMAINS you could set "priority = 40" in dispatch for the DECODE_URL module and "priority = 50" for EXTRACT_DOMAINS. This would ensure that URLs are always decoded prior to domain extraction. A good source of examples for such configurations is `dispatch.yara` and `conditional-dispatch.yara`.

3.1.5. Limiting Modules By User

Dispatch rules can be configured to only allow specific module functionality to execute based on the user that submitted an object into the system. You may have two scenarios: an analyst submitting samples to Laika BOSS using `cloudscan.py` to analyze a single object during their analysis or an automated business related process such as phishing email submission analysis. You may have different jobs or processes setup on an analysis server that uses different user names. For example, the phishing email process may execute scripts under its own user name and each analyst also has their own account on the analysis server. There may be situations where you want to limit the amount of functionality provided to analysts and only allow certain modules to be executed by the phishing process.

In this example you may have a module called SCAN_SANDBOX that will accept an object and send it to a Cuckoo sandbox environment for dynamic analysis. Due to this functionality, we are now pairing static and dynamic analysis within the Laika BOSS framework. In this scenario you only want specific object types such as PDF, Windows PE, RTF and Microsoft Office documents to be sent to your sandbox environment. Requests for such a process will be submitted using `cloudscan.py` due to running Laika BOSS as a networked instance. The reader can update the `dispatch.yara` file to include a new SCAN_SANDBOX rule with the ext_source external variable value of "cloudscan-nolog-laikaboss-phishingprocess" that looks for specific file types. This will ensure that only the phishingprocess user name on the analysis server can utilize the SCAN_SANDBOX functionality.

```
rule SCAN_SANDBOX
```

```
{
    meta:
        scan_modules = "SCAN_SANDBOX"
    condition:
        type_is_pdf or
        type_is_msoffice2003 or
        type_is_msoffice2007 or
        type_is_rtf or
        type_is_mz and
        ext_source contains "cloudscan-nolog-laikaboss-phishingprocess"
}
```

3.1.6. Custom Yara Rule Scanning

As files are being scanned by the framework, there may be situations where you want specific Yara rules to only run against objects of a particular type. As an example, you can develop Yara rules to look for specific email header values or for specific patterns of content in an email body, such as a hyperlink. You may want to include all of these conditions and patterns within a Yara rule solely used against objects that are of type email.

In order to accomplish this task, you need to first develop a custom Yara rule targeting a specific object type. In our example, we are going to target just emails and look for specific email header indicators that we can alert on. The email sample acquired from Virustotal is MD5 hash `c0e646639c26887af2ab3d35f6759af7` that appears to contain a Microsoft Excel document with VB macros. An initial Laika BOSS scan of the email shows no flag hits in the scan metadata output snippet.

```
{
  "scan_result": [
    {
      "order": 0,
      "rootUID": "8a74c5f9-494b-45f8-8a05-12adf4702468",
      "flags": [],
      "depth": 0,
       ...
```

In an attempt to detect this email as malicious, a Yara rule that will only run against emails using the SCAN_YARA module will need to be developed. For this example, the Yara rule file is named `email_rules.yara` and once defined will be located in `/etc/laikaboss/modules/scan-yara`. Taking a brief look at the email identifies an interesting X-Mailer named Cleancode.email v3.1.3 and a HELO string named helo=changeme1.

```
Return-path: <invoices@poradniaoregano.pl>
Envelope-to: service@acetechappliance.com
```

```
Delivery-date: Mon, 03 Oct 2016 09:27:48 -0400
Received: from [113.173.9.234] (helo=changeme1)
        by vme1154.sidushost.com with esmtp (Exim 4.82.1)
        (envelope-from <invoices@poradniaoregano.pl>)
        id 1br3HX-00017E-TP
        for service@acetechappliance.com; Mon, 03 Oct 2016 09:27:48 -0400
Subject: Invoice-574471-33696069-733-1306C5
From: "invoices@poradniaoregano.pl" <invoices@poradniaoregano.pl>
To: <acetechappliance.com>
Date: Mon, 03 Oct 2016 20:27:45 +0700
Mime-Version: 1.0
Content-Type: multipart/mixed; boundary="=-F380ECF021E6e8e"
X-Mailer: Cleancode.email v3.1.3
Message-Id: 20161003202745.FCCF0DE8884@poradniaoregano.pl
```

Our initial Yara rule for testing is looking for two string values: one for X-Mailer and another for a specific HELO string.

```
rule invoice_malware_campaign
{
    strings:
        $xmailer_001 = "X-Mailer: Cleancode.email v3.1.3"
        $helo_001 = "helo=changeme1"
    condition:
        any of ($xmailer_*) and any of ($helo_*)
}
```

Testing the Yara rule against the email shows that it matches on the strings we expect.

```
[~/samples]
analyst-> yara -s ../email_rules.yara Invoice-574471-33696069-733-1306C5
invoice_malware_campaign Invoice-574471-33696069-733-1306C5
0x26b:$xmailer_001: X-Mailer: Cleancode.email v3.1.3
0xa4:$helo_001: helo=changeme1
```

Next, we need to update the dispatch logic for the type_is_email rule to include a new entry in the modules meta parameter for SCAN_YARA that points to the Yara rules we want to only run against emails, `email_rules.yara`.

```
rule type_is_email
{
    meta:
        scan_modules = "META_EMAIL EXPLODE_EMAIL
SCAN_YARA(rule=/etc/laikaboss/modules/scan-yara/email_rules.yara)"
        file_type = "eml"
    strings:
        $from = "From "
        $received = "\x0aReceived:"
        $return = "\x0aReturn-Path:"
    condition:
        (not ext_sourceModule contains "EXPLODE_EMAIL") and
        (($from at 0) or
         ($received in (0 .. 2048)) or
         ($return   in (0 .. 2048)))
}
```

After updating the dispatch logic and creating a new Yara rule, we can now see that the email in question is identified as malicious with a flag value of `yr:invoice_malware_campaign`.


```
{
  "scan_result": [
    {
      "order": 0,
      "rootUID": "b74b3ca7-5942-4eaf-8b36-28bce6c58744",
      "flags": [
        "yr:invoice_malware_campaign"
      ],
      ...
```


3.1.7. Metadata Matching

Dispatch logic can be configured to run modules, specifically SCAN_YARA, against the output of Laika BOSS metadata for an object scan in order to flag on malicious characteristics. Most modules run against specific objects (emails, PE files, Microsoft Office Word documents, etc.) but not metadata output. The value of such functionality is that it allows pattern and rule matching against module output which can only be obtained by performing specific static analysis against an object.

In order to use Yara to scan module metadata output, you need to add SCAN_YARA as a modules parameter for the file type you want to analyze. You need to pass key/value parameters to SCAN_YARA that include the path to the Yara file you want to use for scanning and the specific field key value in the scan result output that the Yara rule will run against. A proper example configuration of SCAN_YARA is shown below.

SCAN_YARA(meta_scan=[scan result field value], rule=[full path to Yara rule])

To provide an example of how to configure these options to detect an object as malicious based on metadata output, a META_DOTNET module was developed. Prior to implementing this module and configuring a new type_is_dotnet dispatch rule, the scanned file had the result output as shown below. The fileType is identified as a plain PE and there were no flags identified. The scan output, on a per object level, identifies the modules that executed. Since META_DOTNET has not been implemented yet it is not listed for this object.

```
  "scanModules": [
       "SCAN_YARA",
       "META_HASH",
       "META_PE",
       "META_EXIFTOOL",
       "DISPOSITIONER",
       "LOG_MONGO"
  ]
```

The flag output can be summarized per object by using the program jq against the scan results as can be seen using the command below.

```
$ laika.py D6235F28FBC266D9F1C68961E8EB2C8F | jq '.scan_result[] |
{"filename":.filename, "hash": .objectHash, "filetype":.fileType,
"flags":.flags}'
       {
         "filename": "D6235F28FBC266D9F1C68961E8EB2C8F",
         "hash": "d6235f28fbc266d9f1c68961e8eb2c8f",
         "filetype": [
           "pe"
         ],
         "flags": []
       }
       {
         "filename": ".text",
         "hash": "a7c49cf6cedfcf70637d921bda8279b6",
         "filetype": [],
         "flags": []
       }
       ...
```

The META_DOTNET module outputs computed indicators from .NET binaries known as the Typelib ID and Module Version (MV) ID hashes. Research was initially done by Cylance on this subject (Wallace, 2015) and their library GetNetGuids was utilized within the META_DOTNET module. In order for this module to run, you need a new dispatch rule to first identify Windows PE .NET binaries because META_DOTNET cannot be run against Windows PE files that were not developed using a .NET programming language. We can then configure the META_DOTNET module to run on those objects and generate the necessary output. The dispatch rule below defines a new object type of type_is_dotnet.

```
rule type_is_dotnet
{
    meta:
        file_type = "pe dotnet"
    strings:
        $lib = "mscoree.dll"
        $func01 = "_CorExeMain"
        $func02 = "_CorDllMain"
    condition:
        type_is_mz and $lib and any of ($func*)
}
```

Throughout our research and analysis, we may have identified a .NET binary that was malicious which contained a specific TypeLib ID value that we want to alert on should we see a similar sample again. Due to the nature of this TypeLib ID value, this may indicate a new version of malware from a particular Visual Studio project (Wallace, 2015). We can create a Yara rule labeled `malicious_netguids.yara` which contains a list of known malicious TypeLib ID values. The dispatch logic for type_is_dotnet can be updated to include a SCAN_YARA entry in the modules section. We can then specify that a specific Yara rule, in this case our list of blacklisted TypeLib IDs, should be run against a metadata field in the Laika BOSS object scan output. In the updated type_is_dotnet dispatch rule below, we are going to target the Typelib_ID field and specify that the META_DOTNET module should first be run, followed by SCAN_YARA based on the ordering of the modules in the scan_modules variable. This ensures we have metadata output to scan. The meta_scan variable defines what field value in the scan result output we are targeting. The rule variable is used to point to a specific Yara rule to use for scanning.

```
rule type_is_dotnet
{
    meta:
        scan_modules = "META_DOTNET
SCAN_YARA(meta_scan=META_DOTNET.DotNet_GUIDs.Typelib_ID,rule=/etc/laikaboss/mod
ules/scan-Yara/malicious_netguids.yara)"
        file_type = "pe dotnet"
    strings:
        $lib = "mscoree.dll"
        $func01 = "_CorExeMain"
        $func02 = "_CorDllMain"
    condition:
        type_is_mz and $lib and any of ($func*)
}
```

After implementing both the META_DOTNET module and updating the dispatch logic, including creating the malicious_netguids.yara file, we are now detecting this file as malicious based on the metadata output generated from META_DOTNET.

```
"scan_result": [
    {
      "order": 0,
      "rootUID": "c05b4326-2945-44fc-badc-45a408b4c9eb",
      "flags": [
        "yr:malicious_netguids"
      ],
      ...
```

Output specific to the META_DOTNET module can be accessed by running the following jq command with the `lbq.py` tool.

```
[~]
analyst-> python scripts/lbq.py -k rootUID -v 3030c88a-cb41-426b-b99b-
df9d2e4f6b96 -f full | jq -r '.[] | select(.depth==0) | {"guids":
.moduleMetadata.META_DOTNET.DotNet_GUIDs}'
{
  "guids": {
    "MVID": "f2a0da69-155e-4543-ad0a-206026e206db",
    "Typelib_ID": "35078dfe-0ce9-4a46-b6be-790d443ff486"
  }
}
```

The reader can also see that there is flag output with a value of "yr:malicious_netguids", showing the malicious TypeLib ID being detected. All Yara flag values are prefixed with "yr:" followed by the Yara rule name. An easy way to identify flags on an object is to run the following scan and use jq to obtain the flag value for each object.

```
$ laika.py D6235F28FBC266D9F1C68961E8EB2C8F | jq '.scan_result[] |
{"filename":.filename, "hash": .objectHash, "filetype":.fileType,
"flags":.flags}'

      {
        "filename": "D6235F28FBC266D9F1C68961E8EB2C8F",
        "hash": "d6235f28fbc266d9f1c68961e8eb2c8f",
        "filetype": [
          "pe",
          "pe dotnet"
        ],
        "flags": [
          "yr:malicious_netguids"
        ]
      }
      {
        "filename": ".text",
        "hash": "a7c49cf6cedfcf70637d921bda8279b6",
        "filetype": [],
        "flags": []
      }
      ...
```

3.2. Configuring Yara Rules

The power of Laika BOSS depends on its ability to use Yara signatures, not only as a configuration language for the framework but also to allow analysts to detect malicious objects. All Yara rules are stored within the folder `/etc/laikaboss/modules/scan-yara`. The SCAN_YARA module is configured by default to use the `signatures.yara` file. The configuration setting is listed in the `/etc/laikaboss/laikaboss.conf` file defined by the variable YaraScanRules. This

value can be changed to be whatever Yara file you want. The main purpose of the `signatures.yara` file is to use Yara include statements to reference additional Yara files to use for detection. A snippet of the `signatures.yara` file is listed below. As a result of these include statements, the additional Yara rules will be loaded by Laika BOSS and used for scanning against objects. As stated in the comment in `signatures.yara`, the only time Yara rules will not be used is if the `dispatch.yara` file specifies SCAN_YARA modules for a file type with a parameter for using a different Yara file. An example of these configuration options is similar to what was done when scanning module metadata to detect malicious objects when using the META_DOTNET module.

```
/* OSINT Rules */
include "osint.yara"
include "Packers_index.yar"
include "Antidebug_AntiVM_index.yar"
include "malware_index.yar"
```

3.3. Configuring Conditional Dispatch Logic

During object analysis within Laika BOSS, the dispatcher performs an initial pass on an object to identify its type and perform actions based on the defined configuration. The conditional dispatcher provides a second pass analysis at an object, answering the question, “Is there anything else we want to do with the object?”. When the dispatcher concludes, there may be flag results as part of performing the initial pass, results which you would like to perform actions against. One example is logging of scan results to a database such as Mongo or to a file system for ingestion by a log forwarder. Another example is the analysis of a backdoor that contains an embedded payload or configuration data. The default conditional dispatch configuration file is located at `/etc/laikaboss/conditional-dispatch.yara`. The format and configuration of this file is the same as that detailed in `dispatch.yara`. You can configure scan modules, set flags and priority levels and access globally defined variables to use when defining your rules. It is also common to place specific modules such as LOG, DECODE and EXPLODE within conditional dispatch.

Conditional dispatch is the typical location of LOG modules. These are responsible for taking scan metadata and logging them to some location. Delivered by default is LOG_FLUENT. The LOG_JSON and LOG_MONGO were developed by the

author and are responsible for logging scan metadata for longer persistence. The reason for having these modules is that when you use `cloudscan.py` and `laika.py` without having logging in place, there is no persistence of the scan result metadata. See Section 8 Appendix B - Developed Modules, for a list of modules that the author has developed. In short, LOG_JSON will log the scan metadata output in json format to `/var/log/laikaboss`. The reader can then install a log forwarder, such as Splunk, to forward the log files to a Splunk server for searching. Another module that was developed is LOG_MONGO. This module will insert scan metadata into a Mongo database. The initial version of this code simply inserts a document into the scan_results collection. The reader can then use the tool `lbq.py` to query this data. Both the tool and MongoDB will be discussed in this paper.

The two other module types typically used in conditional dispatch are DECODE and EXPLODE. The intention, to either decode some form of data or extract an object from within another object, is to trigger on a Yara rule already having been flagged by the initial pass of object analysis during dispatch. As an example, during object analysis, the dispatcher identifies that the file hits on a backdoor signature known to have an embedded Windows PE file, including some encoded configuration data. The scan result may include a flag hit of "yr:backdoor_xyz". Having done in depth analysis on this piece of malware, the analyst may be able to write a script that can go through the sample and extract out the embedded payload and configuration data using a language such as python. The analyst can use conditional dispatch to perform this set of actions automatically based on a flag alerting on the initial backdoor analysis. For a detailed example of such functionality, reference section 3.7 Case Study: Extracting Embedded Payloads from the Zusy Trojan.

3.4. Adding Modules to Enhance Analysis and Workflows

One of the major benefits of using Laika BOSS is its ease of adding new functionality to enhance the analysis capabilities of the framework. This allows flexibility in the analyst's workflow. In order to add new modules in the framework, you need to add your module code to `/usr/local/lib/python2.7/dist-packages/laikaboss-2.0-py2.7.egg/laikaboss/modules/`. Reference section 3.7 Case Study: Extracting

Embedded Payloads from the Zusy Trojan and section 7 Appendix A – Case Study: Extracting Strings from PE files for in-depth examples of how to implement new modules.

3.5. Developing New Modules

Based on the author's experiences, there is a common three step approach to fully implementing a module within Laika BOSS which includes the development, testing and implementation processes. The level of work and skill required to develop each module is dependent on the requirements for the module. The table in Section 8 Appendix B – Developed Modules lists modules that the author has developed for Laika BOSS. There are additional modules that have been developed but are unable to be release at this time. All modules and Laika BOSS related tools and code can be accessed on the author's GitHub at https://github.com/cdiraimondi.

3.5.1. Development Process

The development of a new module comes from an analyst's need to automate some form of analysis and to generate metadata that can be used for searching. The process begins by identifying the requirements for the analysis you are trying to automate. Once completed, it is advised to first develop a standalone script that can meet those requirements. This standalone script would need to be run through testing to ensure it is meeting the requirements defined. Once this is completed, code from the standalone script can be added to a Laika BOSS module, with minor tweaks to implement some of the framework code. For example, if we wanted to automate the ability to determine if Virustotal has any malicious detections for a hash, how would we accomplish that using a standalone script? If your script contains arguments or optional parameters, these may be candidates for being optional parameters within Laika BOSS as well. To expand on the Virustotal example, we can begin asking ourselves questions related to the output of the final module and the way it is configured. What will determine if a hash is malicious? Is it considered malicious if it has one detection or more than 5 or 10? Do we want to alert Laika BOSS to a score over a particular value by means of setting a flag? Providing optional arguments to scripts and more importantly modules, adds additional

configuration flexibility to your module and something worth considering depending on the requirements of your analysis.

An excellent resource for module development is examining already delivered modules in the framework, specifically EXPLODE_HELLOWORLD. This module helps document various features and functionality that can be used within modules. The SCAN_VIRUSTOTAL module developed by the author retrieves a Virustotal report for a MD5 hash and sets a flag value depending on the detection hit threshold parameter configured in the dispatch settings for that module. This was first developed as a standalone script to send a hash to Virustotal, retrieve a report, parse out specific fields and to display their values. Once the right code was in place, the EXPLODE_HELLOWORLD module template was reviewed to determine what framework code was needed in order to satisfy the requirements. Code that wasn't needed was removed and all custom code was placed in the _run function. This is the main module function that can also call other defined static functions within the module. A few basic steps will help get you on your way to developing a module. First, rename the class name to your module name. In this instance the class was specified as SCAN_VIRUSTOTAL which inherits from SI_MODULE

```
class SCAN_VIRUSTOTAL(SI_MODULE):
```

You then need to set the module_name value in the __init__ function to the name of your module, in this case SCAN_VIRUSTOTAL.

```
def __init__(self,):
      self.module_name = "SCAN_VIRUSTOTAL"
```

The analyst can then begin adding script code into the _run function, which is the main function of a Laika BOSS module. It is also at this point where the analyst can refactor code and split pieces of functionality into static methods should they see fit. They just need to be decorated with `@staticmethod` and start with an underscore. The SCAN_VIRUSTOTAL module includes an optional parameter. The intention was to allow the user to configure the detection threshold for when an object may be considered malicious. This would allow an analyst to easily tweak, by object type, the detection threshold on when to flag for an object and determine when it is malicious. The dispatch setting below shows the optional argument for an updated type_is_mz object.

```
rule type_is_mz
{
    meta:
        scan_modules = "META_PE SCAN_VIRUSTOTAL(vt_hit_threshold=5)"
    condition:
        uint16(0) == 0x5a4d
        and not ext_sourceModule contains "META_PE"
}
```

In order to extract that optional parameter for use in the module, you need to call the get_option function in `laikaboss.util`. Adding from `laikaboss.util import get_option` to the top of the module script will provide this functionality. The code below gives an example of how to extract the value and also set a default threshold should that optional parameter not be set in dispatch.yara.

```
vt_hit_threshold_param = int(get_option(args, 'vt_hit_threshold',
'vthitthreshholdparam', 10))
```

Most of the remaining code was taken directly from the original test script. It needed to be slightly updated to reference scanObject.buffer, the object currently being analyzed, in order to calculate an MD5 hash. Logic was then added in order to set the flag value `s_virustotal:malicious` by calling scanObject.addFlag. Finally, metadata was added into the Laika BOSS scan by calling scanObject.addMetadata and passing in the module name, a key value and a dictionary that contained various output from the Virustotal report. The example code for this module is located in Section 9.1 Appendix C – scan_virustotal.py

3.5.2. Testing Modules

After completing development of a standalone script to retrieve a Virustotal report and converting that into a Laika BOSS module, we need to test the module to make sure it functions properly. In order to test a module, you must first add the module code to /usr/local/lib/python2.7/dist-packages/laikaboss-2.0-py2.7.egg/laikaboss/modules/. While we added the module code to the modules folder, we haven't added the module into dispatch, so it won't execute yet. The goal for testing is to determine if there are any bugs in our module code that need to be resolved prior to implementing the module in dispatch.

To begin testing, you can use `laika.py` with the "-d" and "-m" arguments, along with your file. Running `laika.py -h` will explain these options but in short, "-d" allows

you to enable debugging and "-m" allows you to specify a module or list of modules to run against the file you are submitting. When using the "-d" flag, all `logging.debug` calls within Laika BOSS are logged to `laika-debug.log` in the current working directory. You can then specify "-m MODULE_NAME" or in our example –m SCAN_VIRUSTOTAL. The output below shows a test scan using the –d and –m flags with truncated object scan output which as you can see showed that only SCAN_VIRUSTOTAL ran.

```
[~]
analyst-> laika.py -d -m SCAN_VIRUSTOTAL 7CD205F3E5961A1164A6C700575F2575 | jq
.
...
"scanModules": [
 "SCAN_VIRUSTOTAL"
],
```

A little further down in the scan output we can see some of the SCAN_VIRUSTOTAL details, showing that the detection hit was greater than 5 and thus caused our flag `s_virustotal:malicious` to be set.

```
"md5": "7cd205f3e5961a1164a6c700575f2575",
"total": 60,
"report_url":
"https://www.virustotal.com/file/52e7ee91181e3cf3d3a78e1e59d57a3555f1a9ae7869fe
b5cca877749a377e9d/analysis/1496712961/",
"scan_date": "2017-06-06 01:36:01",
"hits": 45
```

Examining the `laika-debug.log` shows various outputs from debug messages within the framework code. It appears there were no errors when submitting the file for analysis.

```
[~]
analyst-> cat laika-debug.log
...
2018-01-09 05:10:40,825 DEBUG si_dispatch - Attempting to run module
SCAN_VIRUSTOTAL against - uid: 54c168e9-d56a-4b04-b945-adc065d8a575, filename
7CD205F3E5961A1164A6C700575F2575 with args {}
2018-01-09 05:10:40,845 INFO Starting new HTTPS connection (1):
www.virustotal.com
2018-01-09 05:10:40,934 DEBUG Setting read timeout to None
2018-01-09 05:10:41,024 DEBUG "GET
/vtapi/v2/file/report?apikey=[REDACTED]&resource=7cd205f3e5961a1164a6c700575f25
75 HTTP/1.1" 200 7164
2018-01-09 05:10:41,054 DEBUG si_dispatch - depth: 0, time: 0.232112884521
2018-01-09 05:10:41,095 DEBUG 70245 Got poison pill
```

Messages and errors can also get dumped in `/var/log/syslog`. A quick check of that log file doesn't show any errors either.

```
[~]
analyst-> sudo cat /var/log/syslog | grep laikad
```

You can submit additional objects for analysis and repeat this process until you are comfortable that there are no bugs in your module code. Once this has completed the module can be implemented.

3.5.3. Implementing Modules

Having fully tested the module, it can be activated by simply adding the module and any necessary optional parameters into `dispatch.yara`. The example below shows an updated type_is_mz rule. The SCAN_VIRUSTOTAL module was configured to set a flag if there are 5 or more detections in the Virustotal report. The module can be applied to any rule within dispatch with either no parameter or a parameter value customized to suit your needs.

```
rule type_is_mz
{
    meta:
        scan_modules = "META_PE SCAN_VIRUSTOTAL(vt_hit_threshold=5)"
    condition:
        uint16(0) == 0x5a4d
        and not ext_sourceModule contains "META_PE"
}
```

3.6. Metadata Logging Options

There are multiple types of logging mechanisms available for object scan output. By default, Laika BOSS comes with a LOG_FLUENT module and syslog for capturing scan metadata. You can log to syslog by passing the argument "-l" to `cloudscan.py`. The results can subsequently be viewed in `/var/log/syslog`. These default logging configuration options can be modified in `laikaboss.conf`. While these logging options have their place, there is value in storing the scan data in a system such as MongoDB or Splunk for historical searching and analysis.

The LOG_JSON and LOG_MONGO modules were developed by the author to provide two options for storing scan output. This provides options to an analyst based on their organization's maturity for log ingestion. The LOG_JSON module writes json scan output to `/var/log/laikaboss`. Each file is written with a file name containing the current scan date and rootUID value. While the files are written there is no way to search the scan output during analysis. Now that these files are written for each scan, you can

use a log forwarder to send them to a central logging server such as Splunk or Kubana for searching. Another option is to use the LOG_MONGO module to log all scan results to a Mongo database. The `lbq.py` tool can be configured to access a MongoDB instance. There are multiple tool options that will allow an analyst to access historical scan data by different data points such as a file md5 hash, flag name, string value, or file name to name a few. The ability to target different scan output fields for searching gives access to a wealth of information and allows an analyst to easily pivot through this historical information for research and hunting purposes. The value of both new logging modules is the ability to store the results and recall them at a future point for analysis.

3.7. Case Study: Extracting Embedded Payloads from the Zusy Trojan

This case study will review multiple aspects of analysis that will provide an example for creating custom modules to help automate static analysis and provide advanced capabilities into the Laika BOSS framework. A second case study has also been provided in Section 7 Appendix A – Case Study: Extracting Strings from PE Files for the reader to gain additional insight into how to utilize the framework. The sample used in this case study is MD5 hash `7CD205F3E5961A1164A6C700575F2575` and has been identified as the Zusy trojan according to the Virustotal report. What is unique about this sample is the fact that there is an embedded and encoded payload. This study will review the static analysis steps used to understand the malware, how it stores and encodes its payload and the scripts and tools used to analyze the file and implement a custom EXPLODE_ZUSY module in Laika BOSS to automate this process.

3.7.1. Initial Static Analysis

The author started their analysis by performing static analysis on the file using various tools and looking up the hash in Virustotal. The author then ran hexdump and decided to scroll through the file, attempting to look for anything interesting. At offset `0x0001f058` a unicode value of SYMBOL was found followed by what looked like base64 text which started at address `0x0001f068`.

The reader can also see the PADDING text starting towards the end of the binary at 0x000e2540

```
0001f050  e4 04 00 00 00 00 00 00  06 00 53 00 59 00 4d 00  |..........S.Y.M.|
0001f060  42 00 4f 00 4c 00 00 00  41 43 41 4a 41 49 41 48  |B.O.L...ACAJAIAH|
0001f070  41 41 43 49 70 55 44 74  48 77 44 2b 33 33 47 31  |AACIpUDtHwD+33G1|
0001f080  66 51 43 37 41 78 69 58  47 41 41 6b 69 58 52 35  |fQC7AxiXGAAkiXR5|
0001f090  44 51 44 4b 45 38 4d 75  63 39 54 6f 36 47 75 59  |DQDKE8Muc9To6GuY|
0001f0a0  4c 36 32 66 58 32 70 64  78 4b 4e 61 6c 68 54 64  |L62fX2pdxKNalhTd|
0001f0b0  32 35 62 61 61 46 79 6d  46 4e 58 62 4c 6c 36 49  |25baaFymFNXbLl6I|
```

The binary was opened in PeStudio to see if there were other suspicious qualities about the file, including any references to this SYMBOL value. A resource with the name SYMBOL was found as can be seen in Figure 5.

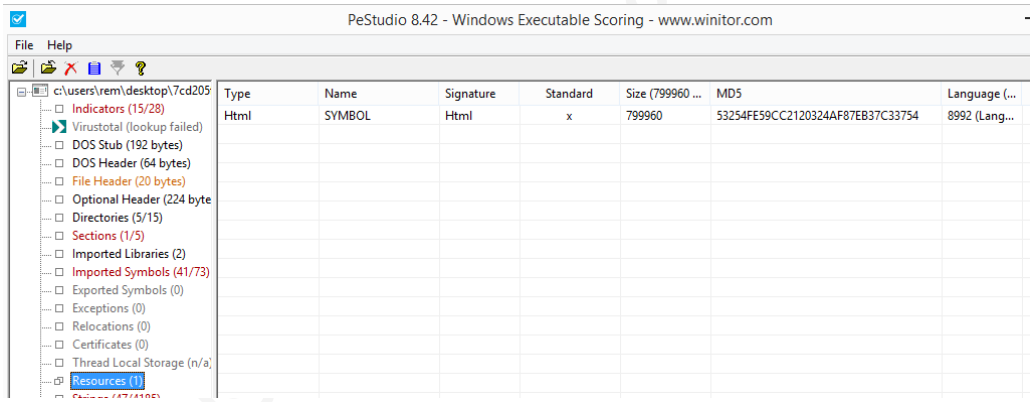

Figure 5 – PeStudio Identification of Resource

The base64 was then carved out by using dd and the command below. This example is starting at the beginning of the base64 text at address `0x0001f068` and then counting the number of bytes by subtracting the start of the base64 text from the end of the base64 text.

```
[~]
analyst-> dd if=7CD205F3E5961A1164A6C700575F2575 of=carved.out bs=1 skip=$[0x0001f068]
count=$[0x000e2540-0x0001f068]
```

The beginning of the base64 decoded content results in nothing readable.

```
[~]
analyst-> cat carved.out | base64 -d | hexdump -Cv | head
00000000  00 20 09 00 80 07 00 00  88 a5 40 ed 1f 00 fe df  |. ........@.....|
00000010  71 b5 7d 00 bb 03 18 97  18 00 24 89 74 79 0d 00  |q.}.......$.ty..|
00000020  ca 13 c3 2e 73 d4 e8 e8  6b 98 2f ad 9f 5f 6a 5d  |....s...k./.._j]|
00000030  c4 a3 5a 96 14 dd db 96  da 68 5c a6 14 d5 db 2e  |..Z......h\.....|
00000040  5e 88 a6 e0 6b 73 c2 45  bb 9a 2f ad 16 dd d7 45  |^...ks.E../....E|
00000050  c3 9a 2f ad 16 dd c3 6a  da 64 ff b8 a6 99 a4 f8  |../....j.d......|
00000060  63 9b 7a 41 16 cd d3 26  da 64 7f 26 d2 60 7e 45  |c.zA...&.d.&.`~E|
00000070  c3 9e 2f ad 1c 5c 27 24  da 2c a4 f8 63 ca c7 80  |../..\'$.,..c...|
00000080  9a 98 2f 2e 5b 9c 2c e8  63 11 6a 51 14 dd d3 fd  |../.[.,.c.jQ....|
00000090  14 d5 d7 fc 77 af 29 ad  9f 1b eb a5 16 dd 97 26  |....w.).........&|
```

The author scrolled through more of the decoded data to see if any patterns could be identified and it was noticed that there was a set of 4 bytes repeating: `9f 98 2f ad`. It

was initially thought that this may be an XOR key because when you XOR a key value with 00 you get the key value.

```
00000860  9f 98 2f ad 9f 98 2f ad  9f 98 2f ad 9f 98 2f ad  |../.../.../.../.|
00000870  9f 98 2f ad 9f 98 2f ad  cf dd 2f ad d3 99 2d ad  |../.../.../...-.|
00000880  99 f7 3b fe 9f 98 2f ad  9f 98 2f ad 7f 98 2c ac  |..;.../.../...,.|
00000890  94 99 27 ad 9f 90 2f ad  9f 8c 26 ad 9f 98 2f ad  |..'.../...&.../.|
000008a0  4f 88 2f ad 9f 88 2f ad  9f b8 2f ad 9f 98 6f ad  |O./.../.../...o.|
000008b0  9f 88 2f ad 9f 9a 2f ad  9b 98 2f ad 9f 98 2f ad  |../.../.../.../.|
000008c0  9b 98 2f ad 9f 98 2f ad  9f d8 26 ad 9f 9c 2f ad  |../.../.../&.../.|
000008d0  9f 98 2f ad 9d 98 2f a9  9f 98 3f ad 9f 88 2f ad  |../.../.../?.../.|
000008e0  9f 98 3f ad 9f 88 2f ad  9f 98 2f ad 8f 98 2f ad  |..?.../.../.../.|
000008f0  9f 98 2f ad 9f 98 2f ad  3f 8c 2f ad b7 98 2f ad  |../.../.?./.../.|
00000900  9f b8 2f ad 47 8a 26 ad  9f 98 2f ad 9f 98 2f ad  |../.G.&.../.../.|
00000910  9f 98 2f ad 9f 98 2f ad  9f 98 2f ad 9f 98 2f ad  |../.../.../.../.|
```

To test this theory, the base64 bytes were decoded and the output was passed to a multi-byte xor decoder with the 4-byte key value of `0x9f982fad` using the command below. The start of the decoded file doesn't look too interesting as it appears to be random bytes.

```
[~]
analyst-> cat carved.out | base64 -d | python tools/xor-multi.py 0x9f982fad >
carved.decoded

00000000  9f b8 26 ad 1f 9f 2f ad  17 3d 6f 40 80 98 d1 72  |..&.../..=o@...r|
00000010  ee 2d 52 ad 24 9b 37 3a  87 98 0b 24 eb e1 22 ad  |.-R.$.7:...$..".|
00000020  55 8b ec 83 ec 4c c7 45  f4 00 00 00 00 c7 45 f0  |U....L.E......E.|
00000030  5b 3b 75 3b 8b 45 f4 3b  45 f0 73 0b 8b 4d f4 83  |[;u;.E.;E.s..M..|
00000040  c1 10 89 4d f4 eb ed e8  24 02 00 00 89 45 f8 e8  |...M....$....E..|
```

After scrolling down further into the file there appears to be an MZ header starting at address `0x7a0`.

```
000007a0  4d 5a 90 00 03 00 00 00  04 00 00 00 ff ff 00 00  |MZ..............|
000007b0  b8 00 00 00 00 00 00 00  40 00 00 00 00 00 00 00  |........@.......|
000007c0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000007d0  00 00 00 00 00 00 00 00  00 00 00 00 d8 00 00 00  |................|
000007e0  0e 1f ba 0e 00 b4 09 cd  21 b8 01 4c cd 21 54 68  |........!..L.!Th|
000007f0  69 73 20 70 72 6f 67 72  61 6d 20 63 61 6e 6e 6f  |is program canno|
00000800  74 20 62 65 20 72 75 6e  20 69 6e 20 44 4f 53 20  |t be run in DOS |
00000810  6d 6f 64 65 2e 0d 0d 0a  24 00 00 00 00 00 00 00  |mode....$.......|
```

Running hachoir-subfile shows us an embedded Windows PE file

```
[~]
analyst-> hachoir-subfile carved.decoded
[+] Start search on 599968 bytes (585.9 KB)

[+] File at 1952 size=598016 (584.0 KB): Microsoft Windows Portable Executable:
Intel 80386, Windows GUI

[+] End of search -- offset=599968 (585.9 KB)
Total time: 104 ms -- global rate: 5.5 MB/sec
```

After dumping the file using dd it was determined that the MD5 hash is `eefdc16f3c37fcaae19d2e6e91767f1c`, a known piece of malware on VT (https://www.virustotal.com/#/file/4cc5a3d5fad77ef06af4cd48a4b05349ae0f965adf69060fadf108a4b561e117/detection) with 46/60 hits.

```
[~]
analyst-> dd if=carved.decoded of=carved.exe bs=1 skip=1952 count=598016
598016+0 records in
598016+0 records out
598016 bytes (598 kB) copied, 0.826521 s, 724 kB/s

[~]
analyst-> md5sum carved.exe
eefdc16f3c37fcaae19d2e6e91767f1c  carved.exe
```

Our analysis has determined that there is a base64 XOR encoded binary stored in the SYMBOL resource of the file with hash `7CD205F3E5961A1164A6C700575F2575`. The author was able to manually analyze the file using various static analysis techniques. While this analysis process is fine, it can be automated by utilizing Laika BOSS. A Yara rule can be written to identify the specific family of malware and when a flag hit is received, an EXPLODE module can be run to carve out the embedded, encoded executable stored in the resource section.

3.7.2. Yara Rule Development

Before an EXPLODE module is developed, a Yara rule needs to be created that will help us identify hash `7CD205F3E5961A1164A6C700575F2575` as including a dropper that will decode the embedded resource. VT identifies this file as Zusy, a popular trojan responsible for stealing credentials (Zorz, 2012). The Yara rule developed will help us identify the malware and allow us to call additional module functionality in conditional dispatch in order to extract and decode the embedded payload.

Based on static analysis previously discussed, the following Yara rule was developed to identify this malware family. The rule is looking for strings within the file.

```
rule zusy
{
    meta:
        desc = "This rule looks for a dropper containing a resource that
installs Zusy"
        author = "Chuck DiRaimondi"
        ckc = "Installation"
    strings:
        $s1 = "UNIQUE_TAG_FOR_THIS_AND_THAT" ascii
```

```
        $s2 = "SYMBOL" ascii
        $s3 = "HAHA" ascii
        $base64tag_001 = "SYMBOL" wide
        $base64tag_002 = "HAHA" wide
    condition:
        uint16(0) == 0x5a4d and any of ($s*) and (for any of ($base64tag_*) :
($ in (10000..filesize)))
}
```

Now that the rule is developed, it can be placed in `/etc/laikaboss/modules/scan-yara/`. The `signatures.yara` file then needs to be updated to include the new `zusy.yara` rule.

```
include "zusy.yara"
```

To verify that the rule is properly loaded, let's run `7CD205F3E5961A1164A6C700575F2575` in Laika BOSS and review the flag hits.

```
[~]
analyst-> laika.py 7CD205F3E5961A1164A6C700575F2575 | jq .
```

A snippet of the scan output is below. In the flags list in the scan output is the signature hit for "yr:zusy" which means our Yara rule is properly identifying the file and Laika BOSS is properly configured to identify the object. All Yara hits are prefixed with "yr:".

```
{
  "scan_result": [
    {
      "order": 0,
      "rootUID": "d4330310-643c-4c20-99ee-880446301800",
      "flags": [
        "yr:zusy"
      ],
      ...
```

The LOG_MONGO module is also properly logging all scan metadata and the `lbq.py` can be used to query the database by flag value to see if we have any "yr:zusy" flag hits.

```
[~]
analyst-> python scripts/lbq.py -k flag -v "zusy"
{"rootUID": "d4330310-643c-4c20-99ee-880446301800", "fileType": ["pe"],
"filename": "7CD205F3E5961A1164A6C700575F2575", "flags": ["yr:zusy"],
"objectHash": "7cd205f3e5961a1164a6c700575f2575"}
```

The detailed scan results could also be retrieved to get additional metadata information.

```
[~]
```

```
analyst-> python scripts/lbq.py -k rootUID -v "d4330310-643c-4c20-99ee-
880446301800" -f full | jq .
[
  {
    "uniqID": "",
    "order": 0,
    "rootUID": "d4330310-643c-4c20-99ee-880446301800",
    "origRootUID": "",
    "scanModules": [
      "SCAN_YARA",
      "META_HASH",
      "META_PE",
      "META_EXIFTOOL",
      "DISPOSITIONER",
      "LOG_MONGO"
    ],
    ...
```

3.7.3. EXPLODE_ZUSY Script Development

In order to properly identify, extract and decode the embedded payload, a python script was developed. The full script is available in Section 9.2 Appendix C – Code Samples. The script first uses pefile to load the main Windows PE file, look for a resource and then extract it. Because there is a Yara rule that has identified this file as belonging to the Zusy family, there should be an embedded resource, unless the malware has changed. Once the resource is extracted, the blob will be base64 decoded. The 4-byte XOR key is identified by referencing bytes `0xad0` to `0xad4`. This has been consistent across multiple samples reviewed. Both the data and key will be passed to a multi-byte XOR decode function to decrypt the payload. Finally, pefile is used again to read the blob of data to identify an embedded Windows PE file. Credit goes to Alexander Hanel for developing the PE carving code (Hanel, 2013). The sample script output is below.

```
[~]
analyst-> python explode_zusy_dropper.py 7CD205F3E5961A1164A6C700575F2575
[+] Attempting to extract encoded payload from 7CD205F3E5961A1164A6C700575F2575
[+] Resource found: SYMBOL
[+] File 2bd97f5209b7202c3d9e4b3423509a97.b64 written.
[+] File 2a5d4f975a7522937be0d68f5ca18d32.b64.decoded written.
[+] XOR Key is: ['0x9f', '0x98', '0x2f', '0xad']
[+] XOR decoded file 2a5d4f975a7522937be0d68f5ca18d32.b64.decoded.xor.decoded
written.
[+] Extracted payload written to e_zusy_eefdc16f3c37fcaae19d2e6e91767f1c
```

The MD5 hash `eefdc16f3c37fcaae19d2e6e91767f1c` is identified on VT has malicious with 46/60 detections and can be referenced at https://www.virustotal.com/#/file/4cc5a3d5fad77ef06af4cd48a4b05349ae0f965adf69060fadf108a4b561e117/detection.

Now that a standalone payload explode script was developed, the standalone explode script code can be used to develop a Laika BOSS module named EXPLODE_ZUSY that will only run on an object when the flag "yr:zusy" is thrown.

3.7.4. EXPLODE_ZUSY Module Development

Prior to developing a Laika BOSS module it is typical to have a working standalone script. Various script components can then be added into the standard module template that is available on the Laika BOSS GitHub page. The complete `explode_zusy.py` script is located in Section 9.3 Appendix C – Code Samples.

When adding your code to a Laika BOSS module, the core functionality will reside in the _run function and supporting functions will be created as static methods, prefixed with "_". In summary, most of the code is the same. The various print statements to stdout were removed, the main class name was named to EXPLODE_ZUSY, the supporting functions findResource and decode_xor were made static methods and prefixed with "_", and specific Laika BOSS functions were utilized in order to write a new output file.

3.7.5. EXPLODE_ZUSY Module Testing

In order to test the functionality of the module, the analyst needs to copy it to `/usr/local/lib/python2.7/dist-packages/laikaboss-2.0-py2.7.egg/laikaboss/modules/`. This is the directory where all Laika BOSS modules reside.

Testing of the module can begin by running `laika.py` in debug mode and specifying the EXPLODE_ZUSY module to run against the submitted object. This will ignore all other modules configured in Laika BOSS and is done merely for testing purposes. As part of this testing, it is worthwhile to review `laika-debug.log` and `/var/log/syslog` entries to see if there are any errors. Here is a snippet of the `laika-debug.log` file which shows the `e_zusy_eefdc16f3c37fcaae19d2e6e91767f1c` file that would be created. This is our extracted payload.

```
[~]
analyst-> cat laika-debug.log
...
```

```
2018-01-04 20:27:08,804 DEBUG si_dispatch - Attempting to run module
EXPLODE_ZUSY against - uid: a5aa7587-f1f4-4900-8a3d-52ef83210342,
filename 7CD205F3E5961A1164A6C700575F2575 with args {}
2018-01-04 20:27:08,940 DEBUG XOR Key is: ['0x9f', '0x98', '0x2f',
'0xad']
2018-01-04 20:27:09,294 DEBUG si_dispatch - Attempting to dispatch -
uid: 6db443f7-acaa-43f4-ba30-4375d8d86d08, filename:
e_zusy_eefdc16f3c37fcaae19d2e6e91767f1c, source module: EXPLODE_ZUSY
2018-01-04 20:27:09,295 DEBUG util: doing on demand Yara scan with
rule: /etc/laikaboss/dispatch.yara
2018-01-04 20:27:09,295 DEBUG util: externalVars: {'ext_timestamp':
'NONE', 'ext_sourceModule': 'EXPLODE_ZUSY', 'ext_flags': 'NONE',
'ext_parentModules': 'EXPLODE_ZUSY', 'ext_source': 'CLI', 'ext_depth':
1, 'ext_size': 598016, 'ext_contentType': 'NONE', 'ext_filename':
'e_zusy_eefdc16f3c37fcaae19d2e6e91767f1c'}
2018-01-04 20:27:09,295 DEBUG util: compiling
/etc/laikaboss/dispatch.yara for lazy load
```

In order to see the extracted payload during module testing, the analyst must run it with the –o option and specify an output directory. This will place all analyzed and extracted objects, including the scan metadata results in `result.json` in a directory named for the rootUID of the scan.

```
[~]
analyst-> laika.py -d -m EXPLODE_ZUSY -o .
7CD205F3E5961A1164A6C700575F2575
```

The directory created was named `c45442cb-18c3-4e4d-a5a0-6fcfa428f0ee` based on the rootUID of the json output sent to stdout. In this folder we can see all the analyzed objects. The file `e_zusy_eefdc16f3c37fcaae19d2e6e91767f1c` is present indicating that EXPLODE_ZUSY must have output something due to the file being prefixed with e_zusy_.

```
[~]
analyst-> cd c45442cb-18c3-4e4d-a5a0-6fcfa428f0ee/

[~/c45442cb-18c3-4e4d-a5a0-6fcfa428f0ee]
analyst-> ls
72953f1e-e2da-4dea-b962-4ab3af26f31e  c17ecbbf-60b1-4b36-bb33-
609c40076d5f  e_zusy_eefdc16f3c37fcaae19d2e6e91767f1c  result.json
7CD205F3E5961A1164A6C700575F2575      c45442cb-18c3-4e4d-a5a0-
6fcfa428f0ee  f4fde84c-034d-4ca4-8e97-1433f4ee740b
```

Running file against `e_zusy_eefdc16f3c37fcaae19d2e6e91767f1c` shows that it is a Windows PE 32 file. Hashing it using md5sum provides MD5 hash `eefdc16f3c37fcaae19d2e6e91767f1c` that ties back to a Virustotal report https://www.virustotal.com/#/file/4cc5a3d5fad77ef06af4cd48a4b05349ae0f965adf69060f

Chuck DiRaimondi, charles.diraimondi@gmail.com

adf108a4b561e117/detection with a detection ratio of 46/60. This is the same MD5 hash that was found using our standalone script.

```
analyst-> md5sum e_zusy_eefdc16f3c37fcaae19d2e6e91767f1c
eefdc16f3c37fcaae19d2e6e91767f1c
```

Reviewing the scan metadata output in `result.json` also confirms that EXPLODE_ZUSY ran.

```
[~/c45442cb-18c3-4e4d-a5a0-6fcfa428f0ee]
analyst-> cat result.json | jq .

      ...
      "scanModules": [
        "EXPLODE_ZUSY"
      ],
```

3.7.6. Conditional Dispatch Configuration Update

Now that the module was fully tested, it can be implemented by updating the `conditional-dispatch.yara` file located in `/etc/laikaboss/`. The added rule named EXPLODE_ZUSY will execute the EXPLODE_ZUSY module if there is a string match to the Zusy Yara rule named "yr:zusy". What this means is that after dispatch processes the object, a flag will be set called "yr:zusy" based on the Yara rule previously developed to identify this malware. Once Laika BOSS is done processing that object, it's going to make a second pass at the objects that will allow us to take action against the previous flag hit. It is during this stage of analysis that the embedded payload can be extracted and decoded.

```
rule EXPLODE_ZUSY
{
    meta:
        scan_modules = "EXPLODE_ZUSY"
    strings:
        $zusy = "yr:zusy"
    condition:
        any of them
}
```

With the new conditional dispatch rule implemented, the object is analyzed again. By navigating into the rootUID folder it can be noticed that the `e_zusy_eefdc16f3c37fcaae19d2e6e91767f1c` file was created.

```
analyst-> cd 757eb07c-25d4-4b5f-a2eb-fa3c1fda3ccd/
[~/757eb07c-25d4-4b5f-a2eb-fa3c1fda3ccd]

analyst-> ls
```

```
5c81c683-f3e5-4e10-8d49-206ce90af376  773ebc63-c2f4-4bb2-946c-6f1b15ca96cf
8ea5f9ed-7f78-4bad-920c-ac4a8a30dcae  e_zusy_eefdc16f3c37fcaae19d2e6e91767f1c
62bd2bad-2eed-4601-b36b-08fd3b7d26ae  79891292-afe5-46df-b16d-b399cba98e46
9a344f9a-2fa4-4f94-9cf2-521d607b8b1f  f4c8aa34-d331-4c36-87dc-101385442798
757eb07c-25d4-4b5f-a2eb-fa3c1fda3ccd  7CD205F3E5961A1164A6C700575F2575
9c66b908-1e08-4f9e-9d70-efa1aff25dea  result.json
```

Running the file command against the object shows it's a Windows PE file.

```
[~/757eb07c-25d4-4b5f-a2eb-fa3c1fda3ccd]
analyst-> file e_zusy_eefdc16f3c37fcaae19d2e6e91767f1c
e_zusy_eefdc16f3c37fcaae19d2e6e91767f1c: symbolic link to `79891292-afe5-46df-
b16d-b399cba98e46'

[~/757eb07c-25d4-4b5f-a2eb-fa3c1fda3ccd]
analyst-> file 79891292-afe5-46df-b16d-b399cba98e46
79891292-afe5-46df-b16d-b399cba98e46: PE32 executable (GUI) Intel 80386, for MS
Windows
```

3.7.7. Summary

During this case study the Zusy trojan was statically analyzed in order to identify a hidden encoded resource. The process of manually analyzing, carving and decoding the payload was completed. This process was then automated by creating an EXPLODE_ZUSY module that was implemented into the Laika BOSS framework and successfully executed in order to produce the decoded embedded payload automatically.

4. Querying Laika BOSS Metadata

One of the benefits of Laika BOSS is the generation of rich object scan metadata. In order to gain the most value of the framework, it is important to have a plan in place for how the output is going to be stored so that you can query and perform advanced searches against the data.

4.1. Scan Result JSON Structure

Before querying the data, it helps to understand the format of the JSON and scan metadata output. To help visualize the output, because the CLI output can be daunting, Figure 6 shows a screenshot from jsoneditoronline.org where the author pasted in some of the scan metadata output for MD5 hash `a8235b9817268fd7f4ba97c18d8b91e7`. Laika BOSS output is a list of dictionaries, whereby each dictionary represents a unique object. In this case, hash `a8235b9817268fd7f4ba97c18d8b91e7` had 5 child objects extracted for a total of 6 that were analyzed. Next, the first object is listed, which is the root object. This is the file the reader would have submitted to Laika BOSS. The order

value starts at 0 and helps keep the objects as they were identified during scanning in the proper order. The rootUID is a unique id assigned for this scan and is tied back to the root level object; the original file submitted for scanning. This is followed by scanModules which contains all of the modules that were run against this particular object. Since the root level object, file `A8235B9817268FD7F4BA97C18D8B91E7`.danger is a Windows PE file, you can see various PE based modules, including the module responsible for logging the scan results to the MongoDB, LOG_MONGO. There are various hashes and ID related fields listed next. One of the most important sections of the JSON output is the moduleMetadata section. This lists each module executed for that object and any associated output based on module execution. Following the moduleMetadata section are the fileType and fileName fields. The fileType value is useful because you can pass this information to the `lbq.py` tool to identify scans that have been performed against a specific object type. The depth value identifies the level of an object in relation to the parent. In this instance, since the output shown is related to the root object, the file submitted, the value is 0. The value gets incremented as each sub-object is identified. The sourceModule lists where the object originated from. Lastly, the flags field identifies any flags that were set for this specific object. As can be seen no Yara rules hit on this object.

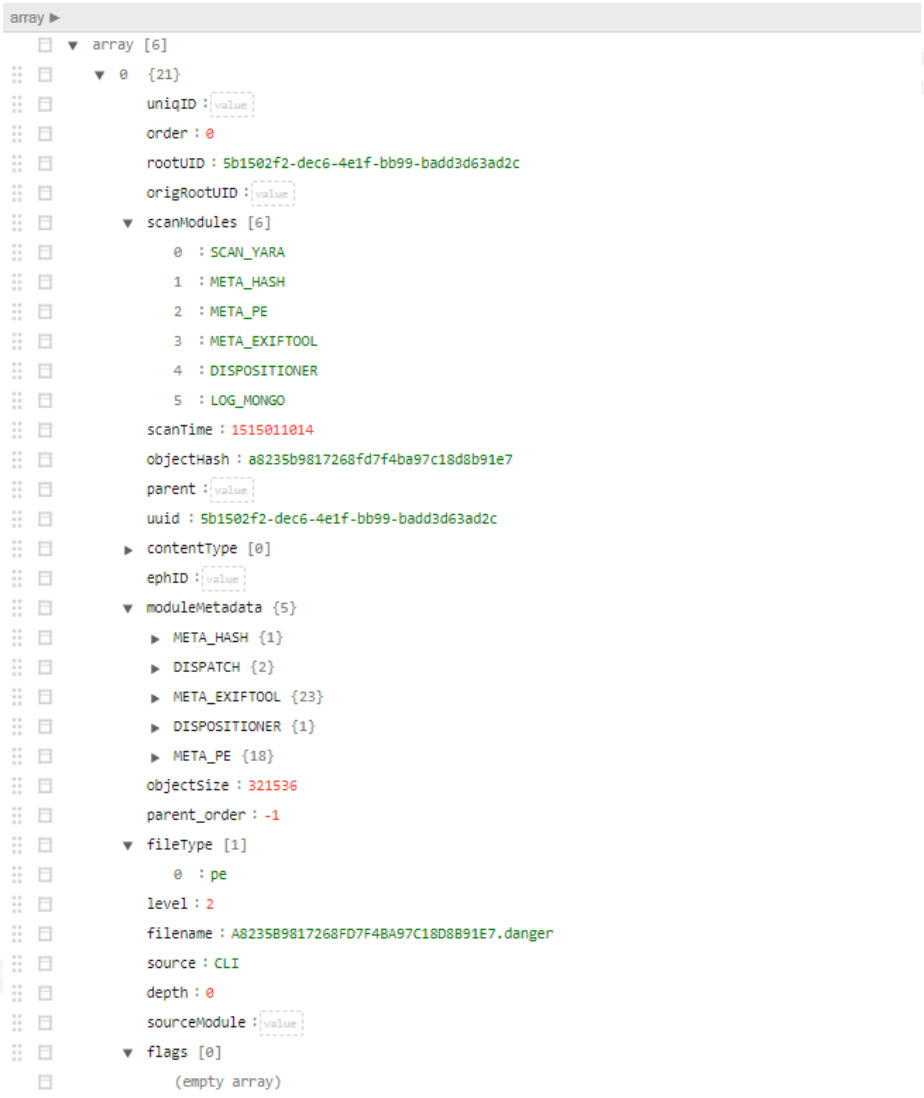

Figure 6 – Scan Output in formatted JSON

As depicted in Figure 7, the META_HASH module output located in moduleMetadata shows all the computed indicators or hashes for this file object.

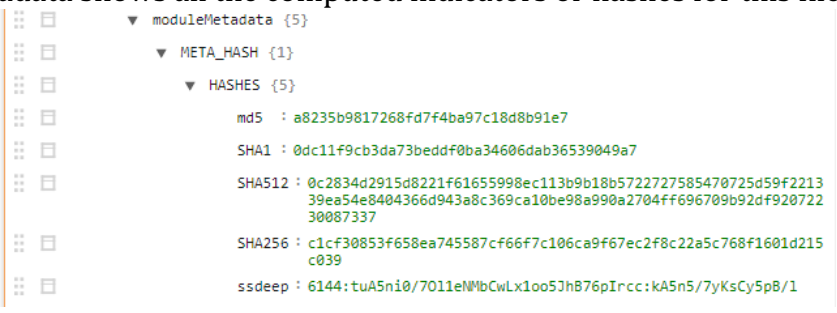

Figure 7 – META_HASH Output

Chuck DiRaimondi, charles.diraimondi@gmail.com

The META_PE module in Figure 8 provides metadata around Windows PE files such as Imports, Resources, Rich Header, Sections, Imphash and more. Each of these values can be pivoted on when searching the scan metadata.

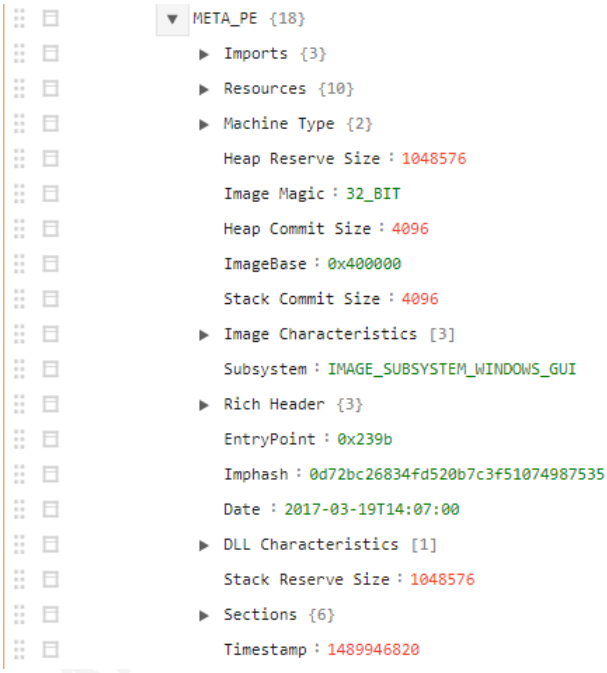

Figure 8 – META_PE Output

4.2. Using lbq.py to Review Object Scan Metadata

While writing this research paper, there was a need to have a utility developed that would allowed analysts to query the scan output metadata that was stored in a Mongo database. The tool developed is named `lbq.py` (Laika BOSS Query) and provides an analyst with a method of querying the historical scan metadata in order to perform analysis and hunting. The current version of the tool provides a few options that will make it easier for analysts to have access to the data they need from Laika BOSS. Future versions of the tool are anticipated to expand in capability. The reader may visit the author's GitHub at https://github.com/cdiraimondi for updates.

Running `lbq.py` with the argument –h will provide a list of available commands for `lbq.py`. The first argument is "-k" which stands for key. This field has a few available options that help identify what piece of data in the scan output you want to search against. For example if you select "-k objectHash" the tool will search Mongo against the scan_result.objectHash field. The "-v" argument allows you to specify a value to search

for against the key previously specified. The filetype, flag and string key values treat the search values as substrings by using the regex keyword in Mongo. This allows for partial string matches. The argument "-f" allows you to specify if you want full json scan output. The only keys that support "-f" are objectHash and rootUID. The logic behind this decision is that it is typically the MD5 hash or the rootUID value that you will use to gather all the metadata for a particular scan.

```
analyst-> python lbq.py -h
usage: lbq.py [-h] [-k {objectHash,filetype,flag,filename,rootUID,string}]
              [-v VALUE] [-f {full}]

Query tool for Laika BOSS scan results.

optional arguments:
  -h, --help            show this help message and exit
  -k {objectHash,filetype,flag,filename,rootUID,string}
                        provide a key field value for querying.
  -v VALUE              provide a value to search for.
  -f {full}             (Optional) provide a formatting option for result
                        output
```

5. A Look Into the Future

The examples detailed in this paper show just a glimpse into what you can accomplish with Laika BOSS. The value in Laika BOSS is providing the ability for the framework to grow as analysts require different types of analysis due to the changing threat landscape. As a result, the author has explored and thought of different areas for future research related to module development for Laika BOSS.

While Laika BOSS is typically used as a static analysis framework, you can integrate and automate the dynamic analysis of samples using various sandbox solutions such as Cuckoo. This has already accomplished with one specific sandbox solution and am is currently being tested. It creates the perfect scenario where you have the ability to examine a sample in an automated fashion using both static and dynamic analysis techniques. During this process, all the metadata is captured and stored for future analysis. This gives the analyst access to all the static file metadata including anything generated during execution of the sample in the sandbox environment. The only limitation is what detail is available from the sandbox and what you've decided to pull back within your module.

Another area to research for new module development is that of external sources that can be used to enrich scan data. This helps provide context on any piece of data and can aid the analyst in more quickly deriving intelligence from an object scan. An example of such modules include one to gather WHOIS data for domains or accessing external threat intelligence services or OSINT sources for gathering of additional data. Services such as Virustotal, DomainTools and OpenDNS can all be modularized assuming you have the proper licenses to do so. As an organization's threat intelligence team starts consuming and developing their own intelligence, typically stored in a threat intelligence platform, Laika BOSS could also be used to query these repositories for specific pieces of intelligence for detection purposes. As long as a system or service has an API and you can access it using python, its fair game for a new module.

Lastly, attackers are often coming up with ingenious ways to evade detection to thwart static and dynamic analysis. The natural progression from a defender's point of view is to attempt to develop a signature to identify something as being malicious. After a while though the maintenance of these rules and the dependency on the attacker to never change can potentially "break" the system and make it easier for the attackers to win. A potential solution to aid in the identification of malicious objects is machine learning. Implementing modules that contain machine learners to solve various types of analytical problems is an area of future research that should be explored. Some examples could include the ability to utilize machine learning and a model such as RandomForests to detect if an object is malicious or not based on a defined set of features developed by the analyst. This could be anything from Windows PE files, VBA macros, Office documents and even aspects of metadata as they are revealed through the execution of modules.

The ability for Laika BOSS to grow in functionality is merely limited by the analyst's imagination. Keep in mind that this is just one tool in the analyst's arsenal but it is a tool that an analyst can control and help grow as the need arises.

Chuck DiRaimondi, charles.diraimondi@gmail.com

6. Conclusion

Analysts use a variety of tools while performing research and analysis on all different types of malware. It is often the case that a common set of analysis steps and data is gathered during this process. The customization options afforded to analysts by using a framework such as Laika BOSS allows them to quickly develop signatures for new variants of malware, create new modules that expand analysis and detection capability and extract and store meaningful metadata that can be searched for historical analysis purposes. Threat actors will change the way in which they develop, package and deliver their malware and it is necessary for analysts to also change the ways in which they perform analysis. Automating much of this work helps the analyst focus on the data and the actual analysis work that is needed to understand the threat.

References

Bianco, David (2014, Jan 17). The Pyramid of Pain. Retrieved from http://detect-respond.blogspot.com/2013/03/the-pyramid-of-pain.html

Hanel, Alexander (2013, Jan). Pe-carv.py Python PE Carver. Retrieved from http://hooked-on-mnemonics.blogspot.com/2013/01/pe-carvpy.html

Hutchins, Eric M., Cloppert, Michael J., Amin, Rohan M. (2009). Intelligence-Driven Computer Network Defense Informed by Analysis of Adversary Campaigns and Intrusion Kill Chains. Retrieved from https://www.lockheedmartin.com/content/dam/lockheed/data/corporate/documents/LM-White-Paper-Intel-Driven-Defense.pdf

Kessler, Gary (2017, Dec 27). File Signatures Table. Retrieved from https://www.garykessler.net/library/file_sigs.html

Liburdi, Joshn (2017, February 18). Laika BOSS + Bro = LaikaBro(?!). Retrieved from https://medium.com/@jshlbrd/laika-boss-bro-laikabro-d324d99fddae

Sikorski, M., & Honig, A. (2012). *Practical Malware Analysis*. San Francisco, CA: No Starch Press Inc

Stevens, Didier (2017, July 17). Quickpost: Analyzing .ISO Files Containing Malware. Retrieved from https://blog.didierstevens.com/2017/07/17/quickpost-analyzing-iso-files-containing-malware/

Wallace, Brian (2015, June 25). Using .NET Guids to help hunt for malware. Retrieved from https://www.virusbulletin.com/virusbulletin/2015/06/using-net-guids-help-hunt-malware

Zorz, Zeljka (2012, May 31). Tiny but deadly banking Trojan discovered. Retrieved from https://www.helpnetsecurity.com/2012/05/31/tiny-but-deadly-banking-trojan-discovered/

Chuck DiRaimondi, charles.diraimondi@gmail.com

7. Appendix A - Case Study: Extracting Strings from PE Files

A consistent step performed during static malware analysis is strings analysis. Case Study 1 will review the steps performed in order to implement FireEye's FLOSS utility as a module called EXTRACT_STRINGS. The MD5 hash used during this study is `D6235F28FBC266D9F1C68961E8EB2C8F`. The module's purpose is to extract ASCII and UNICODE strings and store them in the scan metadata output.

7.1.1. Development

A test script was initially developed which utilized the FLOSS python library in order to extract ASCII strings.

```
import sys
from floss import strings

f = open(sys.argv[1], 'rb')
buffer = f.read()
decoded_strings = strings.extract_ascii_strings(buffer)
for s in decoded_strings:
          print "%s" % s.s
```

A snippet of output shows some of the ASCII and UNICODE strings identified.

```
[~]
analyst-> python floss_test.py D6235F28FBC266D9F1C68961E8EB2C8F
Ascii Strings
===============
!This program cannot be run in DOS mode.
.text
`.rsrc
@.reloc
,2(G
%()
%()
&%()
BSJB
v2.0.50727
#Strings
#GUID
#Blob
E.exe
mscorlib
.....

Unicode Strings
===============
!(E1
"C#1C
"c#1i{
DKT_lyipw~
OFCs
sahm-sam.no-ip.biz
```

```
2596
HacKed
|Black|
3.2.0 [ Legend Edition ]
True
e5a9ed9865566182cd0d27ec63ce3828
False
Worm.exe
~[P6Er7#4$&WJr83!]~
```

So far we've learned a little about how the library works but one question you might ask is why should we bother creating a script and a module around this library? During malware analysis, strings analysis can be a good initial first step in helping generate some hypothesis or questions regarding the malware. It can also be a good source of data for pivoting purposes. By creating a module called EXTRACT_STRINGS and having that run on particular objects, we can store the string data for various file object types long term and use it as a source of intelligence and pivoting. For example, suppose we were to have implemented two modules, EXTRACT_STRINGS and EXTRACT_DOMAINS. We could run EXTRACT_STRINGS against a PDF, Windows PE file or some other object type to get string data and then have Laika BOSS analyze that metadata output to extract any domains it can identify. This helps automate two processes: strings analysis and domain identification.

After testing our standalone script, we develop a module named EXTRACT_STRINGS using the EXPLODE_HELLOWORD template from the Laika BOSS GitHub page. The test code was modified slightly to conform to the Laika BOSS module format. The class and module name were updated to include EXTRACT_STRINGS.

```
class EXTRACT_STRINGS(SI_MODULE):
    def __init__(self, ):
        self.module_name = "EXTRACT_STRINGS"
```

The main module code, primarily based on the test script, was then added in the _run function. The scanObject.buffer, or file object itself, is passed into the ASCII and Unicode FLOSS functions for extracting strings. This configuration is getting strings with 5 or more characters. All results are added to a list depending on string type and then object metadata is added to an ASCII and Unicode dictionary key result with a call to scanObject.addMetadata.

```
def _run(self, scanObject, result, depth, args):
    moduleResult = []
    decoded_ascii_strings = []
    decoded_unicode_strings = []
    decoded_stack_strings = []
    try:
        decoded_ascii = strings.extract_ascii_strings(scanObject.buffer, 5)
        for s in decoded_ascii:
            if s.s:
                decoded_ascii_strings.append(s.s)

        # Get unicode strings
        decoded_unicode = strings.extract_unicode_strings(scanObject.buffer, 5)
        for u in decoded_unicode:
            if u.s:
                decoded_unicode_strings.append(u.s)

        scanObject.addMetadata(self.module_name, 'ASCII_Strings',
decoded_ascii_strings)
        scanObject.addMetadata(self.module_name, 'Unicode_Strings',
decoded_unicode_strings)

    except ScanError:
        raise
    return moduleResult
```

The new `extract_strings.py` file was added to `/usr/local/lib/python2.7/dist-packages/laikaboss-2.0-py2.7.egg/laikaboss/modules/` so that we can begin testing the module.

7.1.2. Testing

As previously discussed in section 3.5.3 Testing Modules, we now want to test running just the EXTRACT_STRINGS module against an object to review its output. For this example we are going to run it against `D6235F28FBC266D9F1C68961E8EB2C8F` with flag options –d and –m EXTRACT_STRINGS. It appears that the module was able to successfully extract strings from the binary.

```
[~]
analyst-> laika.py -d -m EXTRACT_STRINGS D6235F28FBC266D9F1C68961E8EB2C8F
{
  "scan_result": [
    {
      "order": 0,
      "rootUID": "1a96431c-3808-45f2-9484-381ed131837c",
      "flags": [],
      "depth": 0,
      "scanTime": 1515576225,
      "objectSize": 39424,
      "uniqID": "",
      "origRootUID": "",
      "source": "CLI",
      "filename": "D6235F28FBC266D9F1C68961E8EB2C8F",
      "fileType": [],
      "uuid": "1a96431c-3808-45f2-9484-381ed131837c",
```

```
    "objectHash": "d6235f28fbc266d9f1c68961e8eb2c8f",
    "ephID": "",
    "contentType": [],
    "parent": "",
    "parent_order": -1,
    "scanModules": [
      "EXTRACT_STRINGS"
    ],
    "moduleMetadata": {
      "EXTRACT_STRINGS": {
        "ASCII_Strings": [
          "!This program cannot be run in DOS mode.",
```


Reviewing the `laika-debug.log` file also shows successful completion with no errors.

```
[~]
analyst-> cat laika-debug.log
...
2018-01-10 04:24:40,532 DEBUG si_dispatch - Attempting to run module
EXTRACT_STRINGS against - uid: 24e8ec9b-ed11-48c7-8320-f2ee89288bc7, filename
D6235F28FBC266D9F1C68961E8EB2C8F with args {}
2018-01-10 04:24:40,543 DEBUG si_dispatch - depth: 0, time: 0.0134160518646
2018-01-10 04:24:40,547 DEBUG 75722 Got poison pill
```

7.1.3. Implementation

At this point, we're comfortable with implementing the module to execute against objects. The `dispatch.yara` file needs to be updated to include a rule to determine when EXTRACT_STRINGS should run. A custom rule group section was added to include extract_strings. For this example, the dispatcher is set to execute the EXTRACT_STRINGS module against Windows PE, PDF, RTF and Microsoft Office files.

```
/*_____________________________________________________________________*/
/*------------------Custom Rule Grouping--------------------------*/
rule extract_strings
{
    meta:
        scan_modules = "EXTRACT_STRINGS"
    condition:
        type_is_mz or
        type_is_pdf or
        type_is_rtf or
        type_is_msoffice2003 or
        type_is_msoffice2007
}
```

Now that the EXTRACT_STRINGS module is fully implemented, 108 samples were scanned through Laika BOSS. The `lbq.py` tool can be used to search the EXTRACT_STRINGS output in the scan object metadata stored in MongoDB. This will

allow us to check and see what samples may be related. In this example the author will perform a few different searches to help drive the point home.

In order to search for string matches, you can use the –k string argument and specify a value using –v <value>. The example below shows the author asking MongoDB what string output, either in Windows PE, RTF, PDF or Microsoft Office documents contains the string "powershell.exe". The underlying search is case insensitive so keeping powershell.exe all lowercase is fine. Attackers try to evade detections by sometimes altering the case of powershell.exe and related PowerShell commands. There are duplicate entries for the same file because they were submitted more than once. All metadata is kept regardless of how many times you submit the same file. It can be seen that different Word documents contain the powershell.exe string value which could potentially mean they attempt to use powershell.exe in some way.

```
[~/scripts]
analyst-> python lbq.py -k string -v powershell.exe
powershell.exe
{"rootUID": "5714e1c7-271c-497e-a6f8-87bcb5a575aa", "fileType": ["ole"],
"filename": "7391672.doc", "flags": [], "objectHash":
"dce88ea2d850c0fb41b4683931d53d6d"}
{"rootUID": "0f794487-49ae-440f-bf58-59b12c02e0d4", "fileType": ["ole"],
"filename": "8765656.doc", "flags": [], "objectHash":
"7746b8ffad0d4ef3ac687dfb0b898731"}
{"rootUID": "558f3de0-dedc-4634-aad8-19588345471b", "fileType": ["ole"],
"filename": "Document_Properties.doc", "flags": [], "objectHash":
"4264b565ddeff797ad826a77674966d3"}
{"rootUID": "08a0360e-b700-4d1b-8058-9371f0fbabde", "fileType": ["ole"],
"filename": "jacob-wood378.doc", "flags": [], "objectHash":
"50e9f18e1fffdb3ffc288f518ec60194"}
{"rootUID": "357ca41c-0c63-4188-bc66-bfbeb9702482", "fileType": ["ole"],
"filename": "7391672.doc", "flags": [], "objectHash":
"dce88ea2d850c0fb41b4683931d53d6d"}
{"rootUID": "d36bd53d-676a-4987-8a23-5376fe82cb0a", "fileType": ["ole"],
"filename": "8765656.doc", "flags": [], "objectHash":
"7746b8ffad0d4ef3ac687dfb0b898731"}
{"rootUID": "c30b376e-7879-4af1-a154-84127c8b3a64", "fileType": ["ole"],
"filename": "Document_Properties.doc", "flags": [], "objectHash":
"4264b565ddeff797ad826a77674966d3"}
{"rootUID": "0f9a7235-4c53-4112-8f85-aa97061f6510", "fileType": ["ole"],
"filename": "jacob-wood378.doc", "flags": [], "objectHash":
"50e9f18e1fffdb3ffc288f518ec60194"}
```

The analyst can then pivot based on rootUID by using jq to get the details, specifically what modules ran, against what objects and a ton of additional detail, all gathered statically.

```
[~/scripts]
```

```
analyst-> python lbq.py -k rootUID -v 5714e1c7-271c-497e-a6f8-87bcb5a575aa -f
full | jq '.[] | {"scanModules":.scanModules, "fileType":.fileType,
"objectHash":.objectHash, "filename":.filename}'
{
  "filename": "7391672.doc",
  "objectHash": "dce88ea2d850c0fb41b4683931d53d6d",
  "fileType": [
    "ole"
  ],
  "scanModules": [
    "SCAN_YARA",
    "META_HASH",
    "EXPLODE_OLE",
    "EXPLODE_VBA",
    "META_OLEVBA",
    "EXTRACT_STRINGS",
    "META_EXIFTOOL",
    "DISPOSITIONER",
    "LOG_MONGO"
  ]
}
...
...
{
  "filename": "['Macros', 'VBA', 'ThisDocument']",
  "objectHash": "973c95b8cd5e7a8bb3380d7aa08af68b",
  "fileType": [],
  "scanModules": [
    "SCAN_YARA",
    "META_HASH",
    "DISPOSITIONER"
  ]
}
{
  "filename": "['Macros', 'VBA', '_VBA_PROJECT']",
  "objectHash": "a8c43cfde8b78d73d6eb36f3a8fc1f1a",
  "fileType": [],
  "scanModules": [
    "META_HASH",
    "SCAN_YARA",
    "DISPOSITIONER"
  ]
}
...
...
{
  "filename": "e_vba_24e9f7a5b397f94600f51cbd88f4f717",
  "objectHash": "24e9f7a5b397f94600f51cbd88f4f717",
  "fileType": [
    "vba"
  ],
  "scanModules": [
    "SCAN_YARA",
    "META_HASH",
    "DISPOSITIONER"
  ]
}
```

There is tremendous value in being able to search through scan metadata based on string values to answer the question of "Have I ever seen this string in another sample before?" By utilizing existing string extract libraries like FLOSS and implementing them

as Laika BOSS modules, we were able to successfully modify our static analysis workflow to automatically gather string data, store it long term and provide a means to search through it in order to hunt for similar samples based on string terms.

8. Appendix B – Developed Modules

This is a list of modules developed by the author that can be obtained by visiting https://github.com/cdiraimondi/laikaboss-modules

Module Name	Description
EXPLODE_ISO	Extract embedded objects within ISO files
META_DOTNET	Gather .NET metadata related to the TypeLib and Module Version IDs
SCAN_VIRUSTOTAL	Retrieve a scan report from Virustotal by MD5 hash to determine if an object is malicious
EXPLODE_VBA	Extract and decompress VBA macros
LOG_JSON	Log all scan results to a json file for ingestion by a log forwarder
LOG_MONGO	Log all scan results to a MongoDB
EXTRACT_STRINGS	Extract strings from an object using FLOSS
EXTRACT_LINKS	Extract URLs from an object
EXTRACT_DOMAINS	Extract domain names from an object

9. Appendix C – Code Samples

9.1. scan_virustotal.py

```
try:
    md5 = hashlib.md5(scanObject.buffer).hexdigest()

    params = {'apikey': vt_api_key,
              'resource': md5}
    response = requests.get('https://www.virustotal.com/vtapi/v2/file/report',
params=params)
    json_response = response.json()
```

```
    if json_response['response_code'] == 1:
        scan_date = json_response['scan_date']
        md5 = json_response['md5']
        total_submissions = json_response['total']
        total_positives = json_response['positives']
        report_url = json_response['permalink']
        scans = json_response['scans']

        vt_results['scan_date'] = scan_date
        vt_results['md5'] = md5
        vt_results['hits'] = total_positives
        vt_results['total'] = total_submissions
        vt_results['report_url'] = report_url
        vt_results['scans'] = scans

        if total_positives >= vt_hit_threshold_param:
            scanObject.addFlag('s_virustotal:malicious')

        scanObject.addMetadata(self.module_name, "Results", vt_results)
    else:
        scanObject.addMetadata(self.module_name, "Results", "Unknown File")
except ScanError:
    raise
return moduleResult
```

9.2. explode_zusy_dropper.py

```
#!/usr/bin/python
import pefile
import re
import sys
import hashlib
import base64
import binascii

pe = pefile.PE(sys.argv[1])

def findResource(buffer):
    for rsrc in pe.DIRECTORY_ENTRY_RESOURCE.entries:
        for entry in rsrc.directory.entries:
            if entry.name is not None:
                print "[+] Resource found: " + entry.name.__str__()
                offset = entry.directory.entries[0].data.struct.OffsetToData
                size = entry.directory.entries[0].data.struct.Size
    data = pe.get_memory_mapped_image()[offset:offset+size]
    return data

def decode_xor(data, key):
    #key = data[0xad0:0xad4]
    out = ''
    for i, c in enumerate(data):
        out += chr(ord(key[i % 4]) ^ ord(c))
    return out

print "[+] Attempting to extract encoded payload from {0}".format(sys.argv[1])

data = findResource(pe)
if data:
    b64_file_md5 = hashlib.md5(data).hexdigest()
    b64_file_name = b64_file_md5 + ".b64"
    file = open(b64_file_name,"w")
    file.write(data)
    file.close()
    print "[+] File {0} written.".format(b64_file_name)

    decoded_b64_data = data.decode('base64')
    if decoded_b64_data:
        key = ""
```

```
        b64_decoded_hash = hashlib.md5(decoded_b64_data).hexdigest()
        b64_decoded_file_name = b64_decoded_hash + ".b64.decoded"
        file = open(b64_decoded_file_name,"w")
        file.write(decoded_b64_data)
        file.close()
        print "[+] File {0} written.".format(b64_decoded_file_name)

        # For this dropper the XOR key is throughout most of the beginnig of the file
        # Starting at 0xad0 has been a reliable address
        key = decoded_b64_data[0xad0:0xad4]
        print "[+] XOR Key is: %s" % [hex(ord(i)) for i in key]

        decoded_data = decode_xor(decoded_b64_data, key)
        xor_file = b64_decoded_file_name + ".xor.decoded"
        file = open(xor_file,"w")
        file.write(decoded_data)
        file.close()
        print "[+] XOR decoded file {0} written.".format(xor_file)

        # Credit goes out to Alexander Hanel for some of the pe-carv code below
        # http://hooked-on-mnemonics.blogspot.com/2013/01/pe-carvpy.html
        fileH = open(xor_file, "rb")

        for y in [tmp.start() for tmp in re.finditer('\x4d\x5a', fileH.read())]:
            fileH.seek(y)
            try:
                pe = pefile.PE(data=fileH.read())
                embedded_pe_hash = hashlib.md5(pe.trim()).hexdigest()
                extracted_payload_name = "e_zusy_" + embedded_pe_hash
                outfile = open("e_zusy_" + extracted_payload_name, 'wb')
                outfile.write(pe.trim())
                outfile.close()
                print "[+] Extracted payload written to
{0}".format(extracted_payload_name)
            except:
                continue
```

9.3. explode_zusy.py (Laika BOSS Module)

```
import logging
import os
import tempfile
import pefile
import re
import sys
import hashlib
import base64
import binascii


# Import classes and helpers from the Laika framework
from laikaboss.objectmodel import ExternalVars, ModuleObject, ScanError
from laikaboss.util import get_option
from laikaboss.si_module import SI_MODULE
from laikaboss.import config

class EXPLODE_ZUSY(SI_MODULE):
    def __init__(self):
        '''
        This module extracts out an embedded base64 and XOR multi-byte encoded payload
stored in the resource section
        '''
        self.module_name = "EXPLODE_ZUSY"

        self.TEMP_DIR = '/tmp/laikaboss_tmp'
        if hasattr(config, 'tempdir'):
            self.TEMP_DIR = config.tempdir.rstrip('/')
        if not os.path.isdir(self.TEMP_DIR):
            os.mkdir(self.TEMP_DIR)
            os.chmod(self.TEMP_DIR, 0777)
```

```
    def _run(self, scanObject, result, depth, args):
        moduleResult = []

        pe = pefile.PE(data=scanObject.buffer)
        data = self._findResource(pe)
        if data:
            decoded_b64_data = data.decode('base64')
            if decoded_b64_data:
                key = ""

                # For this dropper the XOR key is throughout most of the beginning of the
file due to XORing with 0x00 bytes
                # Starting at 0xad0 has been a reliable address and the key is 4 bytes
                key = decoded_b64_data[0xad0:0xad4]
                logging.debug('XOR Key is: %s', [hex(ord(i)) for i in key])

                decoded_data = self._decode_xor(decoded_b64_data, key)
                try:
                    with tempfile.NamedTemporaryFile(dir=self.TEMP_DIR) as
temp_file_input:
                        temp_file_input_name = temp_file_input.name
                        temp_file_input.write(decoded_data)
                        temp_file_input.flush()

                        # Credit goes out to Alexander Hanel for some of the pe-carv code
below
                        # http://hooked-on-mnemonics.blogspot.com/2013/01/pe-carvpy.html
                        fileH = open(temp_file_input_name, "rb")

                        for y in [tmp.start() for tmp in re.finditer('\x4d\x5a',
fileH.read())]:
                            fileH.seek(y)
                            try:
                                pe = pefile.PE(data=fileH.read())
                                embedded_pe_hash = hashlib.md5(pe.trim()).hexdigest()
                                moduleResult.append(ModuleObject(buffer=pe.trim(),
externalVars=ExternalVars(filename='e_zusy_%s' % embedded_pe_hash)))
                            except:
                                continue
                except ScanErorr:
                    raise
        return moduleResult

    @staticmethod
    def _findResource(pe):
        for rsrc in pe.DIRECTORY_ENTRY_RESOURCE.entries:
            for entry in rsrc.directory.entries:
                if entry.name is not None:
                    offset = entry.directory.entries[0].data.struct.OffsetToData
                    size = entry.directory.entries[0].data.struct.Size
        data = pe.get_memory_mapped_image()[offset:offset+size]
        return data

    @staticmethod
    def _decode_xor(data, key):
        #key = data[0xad0:0xad4]
        out = ''
        for i, c in enumerate(data):
            out += chr(ord(key[i % 4]) ^ ord(c))
        return out
```

10. Appendix D – MongoDB Configuration

In order to store scan result metadata, MongoDB was installed on the Laika BOSS virtual machine using the standard MongoDB installation instructions located at https://docs.mongodb.com/manual/tutorial/install-mongodb-on-ubuntu/.

10.1. Accessing MongoDB Instance

To access the MongoDB instance, you need to type `mongo` at the command line of the Ubuntu virtual machine that has Laika BOSS installed. The mongo interactive prompt is then made available. An excellent resource for options available on the mongo shell is located at https://docs.mongodb.com/manual/reference/mongo-shell/.

```
analyst-> mongo
MongoDB shell version v3.6.1
connecting to: mongodb://127.0.0.1:27017
MongoDB server version: 3.6.1
>
```

The LOG_MONGO module will create a new database named `laikaboss` when it inserts the first scan result. To run commands against the database you need to issue the `use laikaboss` command at the mongo command prompt.

```
> use laikaboss
switched to db laikaboss
```

To view the list of collections available you can issue the `show collections` command. The LOG_MONGO module will by default create the scan_results collection due to the structure of the json scan result output provided by Laika BOSS.

```
> show collections
scan_results
```

To get a count of how many documents we've generated while analyzing objects the `db.scan_results.count()` command can be issue.

```
> db.scan_results.count()
121
```

10.2. MongoDB Indexes for lbq.py

In order to provide better performance when quering the MongoDB using the `lbq.py` tool, multiple indexes were created against different fields in the scan result output. The commands that were issued on the Laika BOSS VM using the mongo shell are listed below. The specific index commands are highlighted in bold.

```
mongo
> use laikaboss
switched to db laikaboss
> db.scan_results.createIndex( { "scan_result.fileType": 1 } )
{
        "createdCollectionAutomatically" : false,
        "numIndexesBefore" : 1,
        "numIndexesAfter" : 2,
        "ok" : 1
}
> db.scan_results.createIndex( {
"scan_result.moduleMetadata.EXTRACT_STRINGS.ASCII_Strings": 1 } )
{
        "createdCollectionAutomatically" : false,
        "numIndexesBefore" : 2,
        "numIndexesAfter" : 3,
        "ok" : 1
}
> db.scan_results.createIndex( {
"scan_result.moduleMetadata.EXTRACT_STRINGS.Unicode_Strings": 1 } )

{
        "createdCollectionAutomatically" : false,
        "numIndexesBefore" : 3,
        "numIndexesAfter" : 4,
        "ok" : 1
}
> db.scan_results.createIndex( { "scan_result.objectHash": 1 } )
{
        "createdCollectionAutomatically" : false,
        "numIndexesBefore" : 4,
        "numIndexesAfter" : 5,
        "ok" : 1
}
>
> db.scan_results.createIndex( { "scan_result.flags": 1 } )
{
        "createdCollectionAutomatically" : false,
        "numIndexesBefore" : 5,
        "numIndexesAfter" : 6,
        "ok" : 1
}
> db.scan_results.createIndex( { "scan_result.filename": 1 } )
{
        "createdCollectionAutomatically" : false,
        "numIndexesBefore" : 6,
        "numIndexesAfter" : 7,
        "ok" : 1
}
> db.scan_results.createIndex( { "scan_result.rootUID": 1 } )
{
        "createdCollectionAutomatically" : false,
        "numIndexesBefore" : 7,
        "numIndexesAfter" : 8,
        "ok" : 1
      }
```

Last Updated: April 25th, 2018

Upcoming SANS Training

Click Here for a full list of all Upcoming SANS Events by Location

SANS Riyadh April 2018	Riyadh, SA	Apr 28, 2018 - May 03, 2018	Live Event
SANS SEC460: Enterprise Threat Beta Two	Crystal City, VAUS	Apr 30, 2018 - May 05, 2018	Live Event
Automotive Cybersecurity Summit & Training 2018	Chicago, ILUS	May 01, 2018 - May 08, 2018	Live Event
SANS SEC504 in Thai 2018	Bangkok, TH	May 07, 2018 - May 12, 2018	Live Event
SANS Security West 2018	San Diego, CAUS	May 11, 2018 - May 18, 2018	Live Event
SANS Melbourne 2018	Melbourne, AU	May 14, 2018 - May 26, 2018	Live Event
SANS Northern VA Reston Spring 2018	Reston, VAUS	May 20, 2018 - May 25, 2018	Live Event
SANS Amsterdam May 2018	Amsterdam, NL	May 28, 2018 - Jun 02, 2018	Live Event
SANS Atlanta 2018	Atlanta, GAUS	May 29, 2018 - Jun 03, 2018	Live Event
SANS London June 2018	London, GB	Jun 04, 2018 - Jun 12, 2018	Live Event
SANS Rocky Mountain 2018	Denver, COUS	Jun 04, 2018 - Jun 09, 2018	Live Event
SEC487: Open-Source Intel Beta Two	Denver, COUS	Jun 04, 2018 - Jun 09, 2018	Live Event
DFIR Summit & Training 2018	Austin, TXUS	Jun 07, 2018 - Jun 14, 2018	Live Event
Cloud INsecurity Summit - Washington DC	Crystal City, VAUS	Jun 08, 2018 - Jun 08, 2018	Live Event
SANS Milan June 2018	Milan, IT	Jun 11, 2018 - Jun 16, 2018	Live Event
Cloud INsecurity Summit - Austin	Austin, TXUS	Jun 11, 2018 - Jun 11, 2018	Live Event
SANS ICS Europe Summit and Training 2018	Munich, DE	Jun 18, 2018 - Jun 23, 2018	Live Event
SANS Cyber Defence Japan 2018	Tokyo, JP	Jun 18, 2018 - Jun 30, 2018	Live Event
SANS Philippines 2018	Manila, PH	Jun 18, 2018 - Jun 23, 2018	Live Event
SANS Crystal City 2018	Arlington, VAUS	Jun 18, 2018 - Jun 23, 2018	Live Event
SANS Oslo June 2018	Oslo, NO	Jun 18, 2018 - Jun 23, 2018	Live Event
SANS Paris June 2018	Paris, FR	Jun 25, 2018 - Jun 30, 2018	Live Event
SANS Cyber Defence Canberra 2018	Canberra, AU	Jun 25, 2018 - Jul 07, 2018	Live Event
SANS Minneapolis 2018	Minneapolis, MNUS	Jun 25, 2018 - Jun 30, 2018	Live Event
SANS Vancouver 2018	Vancouver, BCCA	Jun 25, 2018 - Jun 30, 2018	Live Event
SANS London July 2018	London, GB	Jul 02, 2018 - Jul 07, 2018	Live Event
SANS Charlotte 2018	Charlotte, NCUS	Jul 09, 2018 - Jul 14, 2018	Live Event
SANS Cyber Defence Singapore 2018	Singapore, SG	Jul 09, 2018 - Jul 14, 2018	Live Event
SANSFIRE 2018	Washington, DCUS	Jul 14, 2018 - Jul 21, 2018	Live Event
SANS Malaysia 2018	Kuala Lumpur, MY	Jul 16, 2018 - Jul 21, 2018	Live Event
SANS Cyber Defence Bangalore 2018	Bangalore, IN	Jul 16, 2018 - Jul 28, 2018	Live Event
SANS Pen Test Berlin 2018	Berlin, DE	Jul 23, 2018 - Jul 28, 2018	Live Event
SANS Doha 2018	OnlineQA	Apr 28, 2018 - May 03, 2018	Live Event
SANS OnDemand	Books & MP3s OnlyUS	Anytime	Self Paced