MASTIFF Documentation

Table of Contents

License

1 MASTIFF Introduction

The process of extracting and analyzing the key characteristics of potentially malicious files, known as static analysis, is one of the first steps that is performed in response to a suspected malware incident. Currently, static malware analysis is performed manually using available tools or by submitting samples to publicly available services. If any automation is desired, analysts must create their own automation scripts and procedures. This approach leads to a number of problems:

- Manual analysis is slow and error-prone.
- If an analyst does not know about a specific tool or technique, that tool or technique will not be used.
- Public malware analysis systems provide no guarantee of availability or results, do not support analysis environment customizations, and make the analysis results publicly available. This is undesirable in some cases.
- Manual analysis is not consistent as analysts rarely run the same tools in the same order every time.

To allow analysts to automate the static analysis, the MASTIFF framework was created. This framework utilizes plug-ins to automatically facilitate the static analysis process on a number of different file types. Currently, MASTIFF will perform generic tasks on any files and type-specific tasks on a number of different file formats.

2 Installation

2.1 Technical Requirements

The following software must be installed for MASTIFF to work properly.

- Python 2.6.6 or greater
- Yapsy 1.10 or greater (http://yapsy.sourceforge.net/)
- Python SQLite3 (http://docs.python.org/library/sqlite3)
- Python setuptools (http://pypi.python.org/pypi/setuptools/)
- Yara, libyara and yara-python (http://code.google.com/p/yara-project)

A Python libmagic library is also required. MASTIFF supports two different libmagic libraries:

- libmagic Python extensions (ftp://ftp.astron.com/pub/file/)
  - This may be installed through the source code above or is the library installed as python-magic in most Linux code repositories.
- Python-magic (https://github.com/ahupp/python-magic/)
  - This may be installed through the source code above or via Python pip.

Analysis and output plug-ins may require additional libraries or programs installed in order to function properly. Please refer to plug-in documentation for details.

Prerequisites Installation

The Python setuptools and magic libraries will need to be installed on your own. For Debian/Ubuntu-based distributions, this can be accomplished with:

```
$ sudo aptitude install python-setuptools
$ sudo aptitude install python-magic
```

On Gentoo-based distributions, there is no Python magic package. However, adding the python USE flag to the sys-apps/file package will create the correct Python libraries.

Setuptools can be installed as follows:

```
$ sudo emerge -av setuptools
```

Yapsy will automatically download and install when the make program is run, or you can download and install it on your own. Yapsy is also located in the Gentoo Portage repository.

```
$ sudo emerge -av yapsy
```

Note that the plug-ins utilized by MASTIFF may have their own prerequisites.

2.2 Testing

MASTIFF comes with a test set suite that can be used to determine if all prerequisites have been properly installed and MASTIFF is able to analyze files correctly. To run these tests, run:

```
$ make test
```

Two sets of tests will run.

- Python imports for all MASTIFF core files and plug-ins will be checked to ensure they can be imported. Any that cannot will be displayed.
- MASTIFF will examine 4 different files to ensure there are no issues.

All output will go into the tests/ directory.

2.3 Installation

If you wish to only test out MASTIFF, skip to the Development Testing section.

MASTIFF utilizes the Python setuptools code for installation of the package. The easiest way to install the package is:

```
$ sudo make install
```

This will install the package into the appropriate Python site-packages directory for your system. It will also install mas.py, the main MASTIFF wrapper script into /usr/local/bin.

If you do not have Yapsy installed, it will attempt to download and install it for you.

If you install using this method, the only way to uninstall is to manually delete files.

After installing MASTIFF, examine the mastiff.conf configuration file to ensure all options for plug-ins are pointing to the correct locations for your analysis system.

2.4 Development Testing

If instead you wish to only test it for development purposes, run the following command:

```
$ sudo make dev
```

This will install placeholders into the Python dist-packages that point to this directory. Any modifications made to the code will automatically be reflected when running the software. Additionally, mas.py will be placed in /usr/local/bin.

To uninstall the dev environment, run:

```
$ sudo make dev-clean
```

This will remove all placeholders as well as /usr/local/bin/mas.py.

2.5 Plug-in Requirements

At the current release, the plug-ins utilized by MASTIFF require a number of additional libraries or programs to be installed.

- ssdeep (http://ssdeep.sourceforge.net/)
- pydeep (https://github.com/kbandla/pydeep)
- Yara, libyara and yara-python must be installed, (http://code.google.com/p/yara-project)
- simplejson (https://github.com/simplejson/simplejson)
- Didier Stevens pdf-parser.py (http://blog.didierstevens.com/programs/pdf-tools/)
- Didier Stevens' pdfid.py (http://blog.didierstevens.com/programs/pdf-tools/)
- exiftool (http://www.sno.phy.queensu.ca/~phil/exiftool/)
- pefile library (http://code.google.com/p/pefile/)
  NOTE: Do NOT install pefile from the Debian/Ubuntu repository! Install from source!
- disitool.py (http://blog.didierstevens.com/programs/disitool/)
- openssl binary (http://www.openssl.org/)
- Giuseppe 'Evilcry' Bonfa's pyOLEScanner.py
  (https://github.com/Evilcry/PythonScripts/raw/master/pyOLEScanner.zip)

Some of these programs may be able to be installed from your distribution's software repository, and some may need to be installed from source.

After these programs have been installed, be sure to check the MASTIFF configuration file and update all configuration options to point to the correct locations.

2.6 MASTIFF Configuration

Most of the options that dictate MASTIFF's behavior are contained in a configuration file. MASTIFF uses the Python ConfigParser library, which supports configuration files similar to Windows initialization (INI) files. The default MASTIFF configuration file is named "mastiff.conf".

When a MASTIFF instance is created, a number of places are searched for the configuration file. In

order these are:

- /etc/mastiff/mastiff.conf
- ~/.mastiff.conf (the HOME directory of the user)
- The configuration file specified with the -c option.

When the file is found, it is processed and stored internally within MASTIFF. Although default configuration values are configured directly within the MASTIFF code, a configuration file should be used to set all options. A sample configuration file is given with the project.

The configuration file not only contains options for the MASTIFF framework, but also for the analysis and output plug-ins. Details on how to use the configuration file to store options is contained in the plug-in development section.

The following options should be considered available at any time from the configuration file.

Section	Option	Description
Dir	log_dir	Initially, the base directory all analysis data will go into. After the file is initialized by the framework, it will be updated to be the absolute path to the output directory for that sample.
Dir	base_dir	The base directory (and original value of log_dir) for reports to go into. This is where the database and master log file will be stored by default. This does not exist until after the file has been initialized by MASTIFF, and it does not need to be set in the configuration file.
Dir	plugin_dir	A comma-separated list of directories that additional analysis plug-ins may be located in. Plug-ins in any of these directories will be loaded. Note: A number of plug-ins will be installed by default and do not need to be specified here.
Dir	output_plugin_dir	A comma-separated list of directories that additional output plug-ins may be located in. Plug-ins in any of these directories will be loaded. Note: A number of plug-ins will be installed by default and do not need to be specified here.
Misc	verbose	Whether verbosity is on by default.
Misc	copy	Whether to copy the analyzed file to the log directory. Default is on.
Misc	hashes	Tuple of MD5, SHA1, and SHA256 hashes of the sample. This does not exist until after the file has been initialized by MASTIFF, but will be available to all plug-ins. This does not need to be set in the configuration file.
SQLite	db_name	Name of the database file MASTIFF should use. The log_dir will be prepended to this filename.

2.7 Running MASTIFF

The best way to run MASTIFF is to use the mas.py program. This script has been written to provide you with the maximum number of options for using MASTIFF. This script will be installed to /usr/local/bin when you install the package.

mas.py can be run by only giving it a file or directory to analyze as an argument.

```
$ mas.py /path/to/file2analyze
```

If MASTIFF is given a directory, it will enumerate all files within that directory, and every subdirectory, and analyze them.

Although the only required argument is the file name or directory to be analyzed, the following table lists available options.

Option	Long Option	Description
-c CONFIG	--config=CONFIG	Use an alternate config file. The default is './mastiff.conf'.
-h	--help	Show the help message and exit.
-l PLUGIN_TYPE	--list=PLUGIN_TYPE	List all available plug-ins of the specified type and exit. Type must be one of 'analysis', 'cat' or 'output'.
-o OPTION	--option=OPTION	Override a config file option. Configuration options should be specified as 'Section.Key=Value' and should be quoted if any whitespace is present. Multiple overrides can be specified by using multiple -o options.
-p PLUGIN_NAME	--plugin=PLUGIN_NAME	Only run the specified analysis plug-in. Name must be quoted if it contains whitespace.
-q	--quiet	Only log error messages.
-t FTYPE	--type=FTYPE	Force file to be analyzed with plug-ins from the specified category (e.g., EXE, PDF, etc.). Run with '-l cat' to list all available category plug-ins.
-V	--verbose	Print verbose logs.
-v	--version	Show program's version number and exit.

The following options can also be given to mas.py to interact with the job queue.

Option	Long Option	Description
	--append-queue	Append a file or directory to the job queue and then exit.
	--clear-queue	Clear the MASTIFF job queue and exit.
	--ignore-queue	Ignore the MASTIFF job queue and only process the given file or directory.
	--list-queue	List the contents of the queue and exit.
	--resume-queue	Continue processing of the job queue without giving mas.py a file.

2.8 Queue

In order to process multiple files, MASTIFF maintains a work queue within the Sqlite database. This allows multiple files or a directory to be processed by the framework. The work queue MASTIFF uses is a First-In-First-Out (FIFO) queue. Any files that get added to the queue will not be analyzed

until all previous files have been analyzed.

The work queue is actually processed within mas.py, and not within the framework itself. However, a number of plug-ins add files back into the queue for later analysis.

2.9 Output

The *log_dir* option in the configuration file specifies the base directory where the output from MASTIFF will go. For each file analyzed, a directory will be created in the base directory and named to the MD5 hash of the file.

In *log_dir*, a file named mastiff.log will be created and will contain all log messages for every file analyzed.

The MD5 directory under *log_dir* will contain all of the output from analysis plug-ins, which will be formatted based on the output plug-ins that are enabled. It will also contain a file named "mastiff-run.config" that contains the run-time configuration when the file was analyzed and a copy of the output messages in a file named “mastiff.log”.

3 MASTIFF Framework

This section describes the architecture of the MASTIFF framework, how it works, and the process it follows in order to perform static analysis on files.

3.1 Architecture

MASTIFF is a framework that performs the automated extraction and analysis of characteristics from files, often referred to as static analysis. The framework automatically identifies the type of file being analyzed and applies the correct static analysis techniques, extracting information on the file's key characteristics. By automating the process, static analysis can be performed faster and more efficiently, allowing analysts to examine and act upon the data more quickly.

The main purpose of the MASTIFF framework is to manage the plug-ins and automate the analysis process. Most of the work in identifying the file type and performing analysis is delegated to plug-ins. This allows each plug-in to focus on its specific purpose and not on any of the other processes associated with the framework.

MASTIFF supports three types of plug-ins: analysis, category, and output.

- Analysis plug-ins perform the majority of the analysis work and are designed to support specific types of files (the exception being generic plug-ins that run on all files).
- Category plug-ins are used to organize analysis plug-ins into specific groups based upon file type as well as determine if the file being examined belongs to their category. Both types are explained in more detail in a later section.
- Output plug-ins receive the data created by analysis plug-ins and put them into the appropriate format.

The use of plug-ins enables maximum flexibility and extensibility as future developers can focus on creating and sharing the code that performs the analysis, and not on the underlying framework tasks. This plug-in model has been very successful in the past, as demonstrated by the Metasploit and Volatility projects.

The framework has been developed in Python, the rationale of which is based upon the following

factors:

- Python is a well-known language that is able to run on a number of platforms.
- A large number of libraries exist for Python, including many related to reverse engineering and static analysis.
- The library used to implement plug-ins, Yapsy, is a Python library.

By using Python and a plug-in based architecture, MASTIFF can be expanded quickly and therefore more useful to the malware analysis community.

3.2 Framework Process

The MASTIFF framework follows a specific process in order to analyze a file as illustrated by the following diagram.

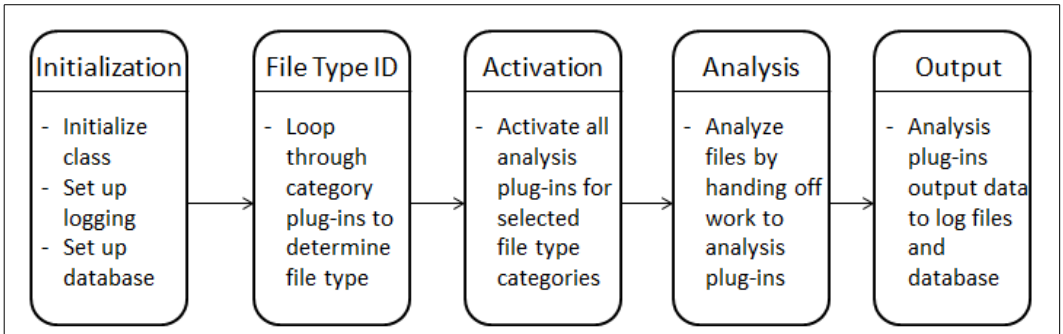

Illustration 1: MASTIFF workflow.

Initialization

MASTIFF initialization consists of two phases. The first occurs when an instance of the MASTIFF class is created (implemented by the class _init_() function). The second occurs when the file to be analyzed is received (most likely occurring during a call to the analyze function).

The first phase initializes member variables and structures for the class, sets up logging and reads in a configuration file that contains a number of options used to configure the framework and plug-ins.

It is during this phase that the category plug-ins are located and activated. The analysis plug-ins are also located in this phase, but are not yet activated.

The second phase of initialization occurs when the file to be analyzed is first seen by MASTIFF. In this phase, integrity validation is performed on the file to ensure it can be read, and the directory where output will be placed is configured.

Upon completion of the second phase of initialization, the MASTIFF's file type identification process begins.

File Type Identification

File type identification involves analysis of the file to identify and validate the file type. This methodology is described in detail later.

Upon identification of the file type, the appropriate category plug-ins are selected. Since each

category plug-in groups the different analysis plug-ins used to examine a specific file type, only those categories relevant to the file type being analyzed are chosen.

Plug-in Activation

After the category plug-ins are chosen, each of the analysis plug-ins for the selected categories are activated. Activation of a plug-in means that it has been loaded and will be used for analysis. During the activation process, each plug-in is also validated.

Validation of the analysis plug-ins means that each plug-in is examined to ensure that the required functions are present. Specifically, each plug-in must contain the following functions:

- activate(): Activates the plug-in for use in the framework.
- deactivate(): Deactivates the plug-in so it is not used during analysis.
- analyze(): The main function that performs the bulk of the work and outputs any data.
- output(): This function is used by output plug-ins to format data in specific formats.

If any of these functions are not present, the plug-in is disabled.

This step may be short but is important to the success of the framework. Since a goal of the project is to allow anyone to develop plug-ins, steps have to be taken to ensure that the plug-ins contain the functions that the framework expects. If this step was not taken and a plug-in was missing one of these functions, the framework would likely crash and negatively affect performance.

Analysis

Once analysis plug-ins have been activated, the analysis step is performed. In this step, the MASTIFF framework calls the analyze() function of each of the validated and activated analysis plug-ins. It is at this point the work of analysis is handed over to the plug-ins and the framework's job is finished.

Output

Data from the analysis plug-ins are collected into a single data structure. Once all analysis plug-ins have finished, this structure is passed to the output plug-ins by calling their output() function. Output plug-ins parse through the data structure and format it into the appropriate output type (e.g. text, csv, json, etc). The final data is placed into the directory given to it from the framework.

4 Plug-ins

This section describes the different types of plug-ins in MASTIFF and how they fit into the framework.

MASTIFF supports three types of plug-ins: category, analysis, and output plug-ins. Each type of plug-in works in conjunction with the framework to perform specific tasks related to the static analysis of files.

4.1 Category Plug-ins

Category plug-ins perform the following functions:

- Group analysis plug-ins into file-type categories.
- Perform file-type identification to determine if a file should be analyzed by the analysis plug-

ins in their category.

The purpose of category plug-ins are to group analysis plug-ins into categories based on the type of file they analyze. For example, the EXECat category plug-in groups together all analysis plug-ins that analyze aspects of Windows executables, while the PDFCat category plug-in groups analysis plug-ins that examine Adobe PDF documents.

The use of category plug-ins to group analysis plug-ins allows accurate selection of plug-ins based upon the file type to be analyzed. If analysis plug-ins were not able to be grouped in this way, they would be required to perform file type validation on their own, leading to redundancy of code and unnecessary work.

Category plug-ins also determine whether the analysis plug-ins can support the file being analyzed. MASTIFF performs initial type identification using the libmagic and Yara utilities, and sends the results to the category plug-in. The category plug-in examines these results, along with any other tests it is configured to perform, and determines if the file is of a type to be examined. If it is, the category is added to an internal list MASTIFF keeps of the plug-in categories to be used.

Using this process, it is possible for more than one category to be used when performing analysis on a single file. For example, MASTIFF supports category plug-ins for both Zip archive files and Java JAR files. The Zip plug-in is intended for analysis plug-ins that extract data from any type of Zip archive, while the Java JAR plug-in is intended for analysis plug-ins that only analyze Java JAR files (JAR files are archived as Zip files). If a JAR file were analyzed by MASTIFF, analysis plug-ins from both categories would be run.

A special category plug-in, known as the generic category, also exists. This category is used to create general purpose plug-ins that will run on any type of file that is analyzed.

The advantage to using category plug-ins for these purposes is that new categories of files can be easily and quickly added. In the future, if someone wanted to start performing analysis on Android APK programs, a category plug-in for APK files would only need to be created. None of the framework code would need to be modified.

Since category plug-ins are used to organize analysis plug-ins and determine file type, they do not output any data.

4.2 Analysis Plug-ins

Analysis plug-ins perform the majority of the work within the MASTIFF framework. Their sole purpose is to perform a specific type of analysis on a file and output any data generated from the analysis.

Each analysis plug-in is categorized by the type of file it examines. This is done using the category plug-ins, which define what file types are examined. In the code, analysis plug-ins are sub-classed from a specific category plug-in when they are created. Therefore, when a category plug-in determines that a given file is one it can analyze, all of the analysis plug-ins derived from it are activated and run on the file.

For example, a category class named PDFCat exists that groups together all analysis plug-ins that examine Adobe PDF documents. If an analysis plug-in is derived from the PDFCat class and an Adobe PDF document is given to MASTIFF to analyze, the PDFCat category plug-in will determine it should be used and all analysis plug-ins derived from it will be run on the file.

After a file has been analyzed, analysis plug-ins place their data into a centralized data structure that contains the output from all other analysis plug-ins. This data structure is used by the output plug-ins after all analysis plug-ins have run. All plug-ins will have an attribute named *results* which

contains previous analysis plug-in output. Analysis plug-ins should not modify this structure.

Any output generated by the analysis plug-in should be returned when the plug-in exits.

Some analysis plug-ins may place data directly in the SQLite database for correlation. The methods available in the framework to assist with this are defined later.

4.3 Output Plug-ins

Output plug-ins take in the structure that contains the data from all of the analysis plug-ins and converts it into a specific output format. Each output plug-in specializes in formatting the output into its own format. For example, the text output plug-in converts the data into text files, while the HTML plug-in (forthcoming) converts the data into an HTML file. All output files are placed in a directory named for the MD5 hash of the sample under the base directory specified by the Dir.log_dir option in the MASTIFF configuration file.

The data structure received by the output plug-ins contains is unique to the MASTIFF framework. It is detailed in section 6.3.

5 File Type Identification

One of the key advantages to using MASTIFF for static analysis is that only the plug-ins that analyze a specific file type will run on that file type. In order to accomplish this, MASTIFF must first correctly identify the type of file being analyzed.

To ensure the most comprehensive file type identification is being performed, each category plug-in will perform at least two different types of checks. The checks can be performed through:

- Libmagic
- Yara
- Custom Checks

TrID can also be used to perform file type identification, but this is optional and should not be considered one of the two required checks.

Libmagic

First, the MASTIFF framework will run the file being analyzed through libmagic, the library which facilitates the UNIX “file” command. This is performed through the set_filetype() function within mastiff.core (core.py) and utilizes the functions get_magic() located in mastiff.filetype (filetype.py).

The get_magic() function receives the filename of the file being analyzed and runs it through libmagic. This is essentially the same thing as if one were to run the UNIX “file” command on the file. The result from get_magic() is a string that contains the output from libmagic.

After get_magic has been called, the output is stored in a dictionary within MASTIFF named self.filetype. The dictionary's keys and values are:

Key	Value	Example
magic	Libmagic value as a string.	"PDF document, version 1.5"

Note that only libmagic is required to be on the system.

Yara

Most category plug-ins will also perform Yara rule checks to determine the file type. Yara rules for file types are type specific, and are therefore performed in the is_my_filetype() method within the category plug-in. File type Yara rules are stored in the variable yara_filetype in the category plug-in class (the default value for this is None).

Custom Checks

Custom file type identification checks can also be implemented within the category plug-in Python code. These checks may call on external libraries or programs to verify the file type.

TrID

TrID, a file identifier created by Marco Pontello, can also be used to perform file type identification. None of the current category plug-ins use TrID due to licensing and portability issues. However, the get_trid() function is still called in the set_filetype() function within mastiff.core (core.py) if the TrID options have been configured.

Unfortunately, TrID does not have a Linux library, so the get_trid() function is a wrapper to the Linux executable. As such, the get_trid() function requires the paths to the TrID binary and database passed to it with the filename to examine. These values are obtained from the MASTIFF configuration file from the "trid" and "trid_db" options.

When run, TrID does not give one file type back as output. Instead it returns a number of hits, each with a percentage as to how likely it is to be that type. Therefore, the result from the get_trid() function is a list containing items of the percentage likelihood of the match (stored as a float) and the description of the match.

For example, the TrID output from an examination of a Windows executable may be:

```
Collecting data from file: /malware/bad.exe

64.5% (.EXE) Win32 Executable MS Visual C++ (generic) (31206/45/13)
13.6% (.DLL) Win32 Dynamic Link Library (generic) (6581/28/2)
```

MASTIFF will store this in a list as:

```
[ [64.5, 'Win32 Executable MS Visual C++ (generic)'], [ 13.6, 'Win32 Dynamic Link Library (generic)'] ]
```

After get_magic has been called, the TrID output is stored in a dictionary within MASTIFF named self.filetype. The dictionary's keys and values are:

Key	Value	Example
trid	List containing the percentages and descriptions from TrID.	[[100.00, "Adobe Portable Document Format"]]

Category Plug-ins

Once the self.filetype dictionary has been created, the MASTIFF framework, still in the set_filetype() function, loops through all of the category classes that have been discovered. For each category plug-in MASTIFF knows about, the category plug-in's is_my_filetype() function is called and passed the self.filetype dictionary and the filename being analyzed. The category plug-in then examines the data and determines if the file is one that its plug-ins can analyze. Additional checks may also be performed on the file to ensure that it is of a type the plug-in analyzes.

Off-loading the file-type determination to the category plug-ins ensures the framework can focus on management and automation of the plug-ins and less on determining file-type. Additionally, because this occurs within the category plug-ins, new category plug-ins – and therefore new file types – can be quickly added to MASTIFF. This allows faster and easier expansion of the framework.

If the category plug-in determines that the file type is one that it analyzes, it will return its internal name to MASTIFF. This variable is stored as self.cat_name in the category plug-in. The None value is returned otherwise.

MASTIFF takes the value returned and appends it to an internal list named self.cat_list. Once completed, this list will contain the names of all the category plug-ins that can analyze the file.

Analysis Plug-in Activation

After all of the category plug-ins have been checked and self.cat_list has been constructed, MASTIFF will activate all of the analysis plug-ins by calling the activate_plugin() member function. This function loops through all of the categories in self.cat_list and validates each of their analysis plug-ins. The validation process is described in section 4.2.

If an analysis plug-in is validated, it is activated and will be used to analyze the file given to MASTIFF. If the analysis plug-in is not able to be validated, the plug-in is deactivated and is not used. The validation process is a sanity check to ensure the analysis plug-ins are constructed correctly and will not crash the framework.

6 MASTIFF Technical Implementation

The MASTIFF framework is a Python module located in the "mastiff" directory of the project. This module contains a number of files that perform distinct functions to make the project work.

File	Description
mastiff/core.py	The main framework class, Mastiff, and its member functions.
mastiff/conf.py	The MASTIFF Conf class used to parse and maintain the configuration file.
mastiff/filetype.py	A number of functions that assist file type identification.
mastiff/plugins/__init__.py	A number of helper functions that plug-ins may commonly use. This can be accessed by importing *mastiff.plugins*.
mastiff/plugins/output/__init__.py	The output plug-ins data structure and associated functions.
mastiff/queue.py	Code that implements the MASTIFF work queue. Also contains functions that allow files to be added into the queue for later analysis.
mastiff/sqlite.py	Helper functions to allow SQLite database interaction.
mastiff/plugins/category/categories.py	The base category class, MastiffPlugin.
mastiff/plugins/category/generic.py	The generic category class. This is a class of plug-ins that run on every file.
mastiff/plugins/category/exe.py	The Windows executable category class.
mastiff/plugins/category/pdf.py	The Adobe PDF category class.
mastiff/plugins/category/office.py	The Microsoft Office category class.

File	Description
mastiff/plugins/category/zip.py	The Zip archive category class.

The primary MASTIFF class is contained in the mastiff/core.py file and is named "Mastiff". To begin the analysis process, a wrapper script must be called that creates an instance of the class and passes it a file to analyze. At a minimum, the following Python code is all that is required to start analysis.

```
import mastiff.core as Mastiff

my_analysis = Mastiff.Mastiff('/path/to/config')
my_analysis.analyze('/path/to/file2analyze')
```

This assumes that the MASTIFF configuration file has been set up properly.

A robust wrapper script, named mas.py, has been provided with the project. This script allows the analyst to supply a number of options and control the way MASTIFF functions much better. It is recommended to use this script when performing analysis.

6.1 Mastiff Class

The Mastiff class, located in mastiff/core.py, is the primary class used for the static analysis framework. The following describes the member variables and functions within the class.

Member Variables

Variable	Description
cat_paths	List that contains the path to the category plug-ins.
plugin_paths	List that contains the paths to the analysis plug-ins.
filetype	Dictionary used to store the output from the file-type identification functions.
file_name	Full path to the file being analyzed.
hashes	Tuple of the MD5, SHA1, and SHA256 hashes of the file being analyzed. This is also stored in the configuration file.
db	SQLite3 Connection class to the database file.
cat_list	List that contains all of the category plug-ins to be used during analysis.
activated_plugs	List that contains all of the analysis plug-ins that will be used to analyze the file.
cat_manager	Yapsy PluginManager class that manages the category plug-ins.
plugin_manager	Yapsy PluginManager class that manages the analysis plug-ins.
output_manager	Yapsy PluginManager class that manages the output plug-ins.
output	Dictionary that contains all of the output data from analysis plug-ins. This is sent to each analysis plug-in as an attribute named results.

Member Functions

__init__(self, config_file=None, fname=None, loglevel=logging.INFO)
The initialization function of the class. This function will initialize all of the member variables, set up logging, read in and store the configuration file, and find and load all plug-ins.

init_file(self, fname)
This function validates the filename being analyzed to ensure it exists and can be accessed, sets up the directory that all output will be logged into, and adds initial file information into the database.

set_filetype(self, fname=None, ftype=None)
Calls the file-type identification helper functions in mastiff/filetype.py, and loops through all of the category plug-ins to determine which ones will analyze the file.

validate(self, name, plugin)
Validates an analysis plug-in to ensure that it contains the correct functions.

activate_plugins(self, single_plugin=None)
Loops through all analysis plug-ins for category classes relevant to the file type being examined and ensures they are valid. If validated, the analysis plug-in is activated. This function also ensures any prerequisite plug-ins are executed before the plug-ins that require them.

analyze(self, fname=None, single_plugin=None)
Ensures the file type of the file is set up and loops through all activated analysis plug-ins and calls their analyze() function.

list_plugins(self, type='analysis')
Helper function that loops through all available plug-ins and prints out their name, path, and description. The function can print out analysis or category plug-in information.

6.2 Database

The MASTIFF framework logs analysis data into a SQLite database so it can be easily referenced in the future. Analysis plug-ins have the ability to add their analysis data into the database as well.

Database Functions

The mastiff.sqlite (mastiff/sqlite.py) module contains a number of functions used to help when interacting with the database. These functions are detailed below.

open_db(db_name)
Attempts to open a database file and returns a SQLite3 Connection object. If the file does not exist, it will attempt to create it.

open_db_conf(config)
Reads the database information from a given MASTIFF config file. Returns a SQLite3 Connection object, or None if it cannot be opened.

sanitize(string)
Sanitize a string that cannot be sent correctly to the SQLite3 library. Returns a string only containing letters, numbers, whitespace or underscores.

check_table(db, table)
Returns True if a table exists in the given database, False otherwise.

add_table(db, table, fields)
Add a table to a database. The table parameter is a string containing the table name. The fields

parameter is a list of columns in the form 'column_name column_type'. Returns True if successful, False otherwise.

add_column(db, table, col_def)
Adds a column to an existing table. The col_def parameter is the definition of the column to be added. Returns True if successful, False otherwise.

create_mastiff_tables(db)
Creates the tables required for the MASTIFF database to store the main analysis information.

insert_mastiff_item(db, hashes, filename)
Inserts info on the analyzed file into the database. The hashes parameter should be a tuple of the MD5, SHA1, and SHA256 hashes of the file. Filename is the base filename of the file.

get_id(db, hashes) Returns the database ID number of the given tuple of hashes. Returns None if the tuple does not exist.

Schema

There are only two tables in the base schema for the MASTIFF database: mastiff and files.

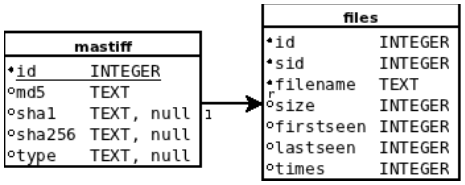

Illustration 2: MASTIFF DB Schema

The 'mastiff' table is the main table of the database and keeps track of analyzed samples by recording the MD5, SHA1, and SHA256 hashes of each sample. By tracking samples using all three hashes, the risk of duplication due to collision is reduced. The type of file the samples were detected as is also kept in this table.

An 'id' field is used as the primary key. This number is automatically incremented as new entries are added to the database. This field should be used in any new table to tie data back to the original malware.

Field	Type	Description
id	INTEGER PRIMARY KEY	Internal DB ID of the sample.
md5	TEXT	MD5 Hash
sha1	TEXT	SHA1 Hash
sha256	TEXT	SHA256 Hash
type	TEXT	List of file types for this file.

The 'files' table contains the full path of the filename being analyzed. It is tied back to the sample using the 'id' field.

6.3 Output Plug-ins and Data Structure

MASTIFF uses a central structure to store data created by analysis plug-ins. The output plug-ins take this structure and format the data within it to the otuput format the plug-in is written for (i.e. HTML for the HTML output plug-in, text for the text output plug-in, etc.) Since this structure is unique to MASTIFF, the following section describes how it is set up.

Information on how to progmatically add data is specified in section 7.3.

All data from analysis plug-ins is stored in the MASTIFF framework instance as a Python dictionary named output. By default, all analysis plug-in output will be in the format:

output[(MD5, SHA1, SHA256)][category][plugin] = page

where:

- (MD5, SHA1, SHA256) is the hash tuple for the sample being analyzed.
- category is the name of the category of the analysis plug-in.
- plugin is the name of the analysis plug-in.
- Underneath this, the data for the plug-in is stored in a class known as a *page*, described below.

The majority of the data created by analysis plug-ins can be structured into one or more tables. For example, the PE resources plug-in structures data into one table. Each row of the table is the information related to a specific resource in the executable, and each column is that data field (e.g. Name, Type, Language, etc). MASTIFF uses this to its advantage by storing all data in a table-like structure.

Note: All code for the structures discussed below is located in mastiff/plugins/output/__init__.py.

All analysis plug-in output is stored in a *page*. Within each page are one of more *tables*. Each table contains a *header*, which describes the data in the table; and multiple *rows*, which contain the data.

Page

The page contains all of the data for the analysis plug-in. By default, every plug-in inherits an empty member variable named *page_data* (accessed via *self.page_data*).

Pages contain the following member variables.

Member	Description
meta	Dictionary to store metadata about the output.
meta['filename']	The only required data within the meta dictionary, this should be set to the base name of the file the output is stored in. e.g. “strings” for the Embedded Strings plug-in.
addTable(title, header, index)	Member function used to add a new table to the page. The function should be passed the *title* of the table as a string, the header consisting of a list

The only page member function is addTable():

addTable(title,header, index)

Adds a table to the page.
Title should be a string containing the title of the table.
Header should be a list to create the header of the table in the form (name, type) for each item in the header. This can be specified later.
Index is an optional number that describes in what order this table should be displayed.

Table

Each logically grouped data created by analysis plug-ins will be stored in its own *table*. Some plug-ins may have multiple tables, some may only have one.

Tables contain the following important member variables.

Member	Description
header	List containing the column names and types.
title	String describing the contents of a table.
rows	List of rows that contains the table data.

The following member functions are available to tables.

__init__(header, data, title)
Constructor for the table. All parameters are optional.
Header is a list to describe the contents of the table in the form (name, type) for each item in the header.
Data is a list containing the initial row of data for the the table.
Title is a string describing the data contents.

addtitle(title)
Adds a title to the table.

addheader(header, printHeader)
Adds a header to the table. The header defines the format of the table and should be a list of tuples composed of the names of the fields in the table, and their type in the form (name, type). printHeader is a boolean variable which determines if the header should be printed by output plug-ins. Defaults to True.

addrow(row)
Adds a row of data to the table. A row should be a list containing the table. Each item in the list will be placed into its own column. A header must be defined before a row can be added.

Header

Each table contains a header, which describes the data in the table. The header is actually a list that contains the name of the data, and its data type. The current data types supported are the Python data types int, str, float, unicode, and time.struct_time.

7 Plug-in Development

One of the goals for creating the MASTIFF framework and utilizing plug-ins is to allow others to quickly develop code that performs additional analysis techniques and integrate them into the framework. The following development guidelines are required in order to maintain operational reliability.

The following general requirements should be used when creating any type of plug-in for MASTIFF.

- All plug-ins should be written in Python.
- The Python PEP 8 Style Guide for Python Code should be followed when coding plug-ins. (http://www.python.org/dev/peps/pep-0008/)
- All files, classes and functions should be documented with docstrings. Please follow the PEP 257 – Docstring Conventions (http://www.python.org/dev/peps/pep-0257/) when possible.
- To ensure maximum compatibility and robustness, pylint should be run on any plug-in that is developed. The final score of the plug-in does not need to be 10.0, but the goal should be to get as close as possible.
- Any non-standard module imported by your plug-in, other than MASTIFF or Yapsy modules, should be surrounded by the following try/except block:

  ```
  try:
      import non_standard_module
  except ImportError, err:
      print "non_standard_module was not imported: %s", err
  ```

  This will allow the user to know that they are missing a required module. Without this, the user may not know the plug-in did not run.
- Plug-ins should never use sys.exit() or any other abortive function. Doing so would cause the entire framework to cease running. If a plug-in needs to stop working due to an error, the error should be logged and a "return False" should be performed.
- All plug-ins should return True if analysis was successful, False otherwise.

Logging

Any messages from plug-ins that are displayed to the user should be done through the Python logging module. The MASTIFF framework automatically sets up the formatting and logging level for the "Mastiff" top-level name. Additionally, the framework sets up logging messages to be displayed to the user and echoed to a file named "mastiff.log", located in log_dir (as specified in the configuration file) and the output directory for the analyzed file.

To set up logging within a plug-in, the following code should be used.

```
import logging

class myClass(CatClass):

  ...

  def analyze(self, config, filename):

      log = logging.getLogger('Mastiff.Plugins.' + self.name)

  ...
```

Utilizing the internal variable, self.name, will allow the plug-in to ensure it is identified correctly when it logs a message.

There are four different log message functions that can be used within a plug-in:

- log.debug() – This is used to display any debug or verbose messages to the user. These messages are not shown by default and only when the -v option is specified with mas.py.
- log.info() – This is used to display information messages to the user. Log.info messages are always displayed, unless the "-q" (quiet) option is given to mas.py.
- log.warning() – This is used to display warning messages to the user. Log.warning messages are always displayed, unless the "-q" (quiet) option is given to mas.py.
- log.error() – This is used to display any type of error message to the user. These messages are always shown.

Documentation on how to utilize these functions can be found at http://docs.python.org/library/logging.html. An example is shown below.

```
import logging

class myClass(CatClass):

  ...

  def analyze(self, config, filename):

      log = logging.getLogger('Mastiff.Plugins.' + self.name)

      log.debug('Starting %s plug-in.', self.name)
      try:
          f = open('/etc/hosts')
      except IOError, err:
          log.error('Could not open file: %s', err)

      log.info('Finished.')

  ...
```

Configuration

Plug-ins can use the MASTIFF configuration file to store options, such as paths to external programs or files. Since MASTIFF utilizes the Python ConfigParser module, configuration options are separated into sections similar to Windows Initialization (INI) files. Section 4.3 lists the sections and variables available in the default configuration file to the plug-ins.

- The section in the MASTIFF configuration file for a plug-in should be the same name as the plug-in's Yapsy name (which is the same as its self.name variable).
- All options for that section should be prefixed with a short and descriptive indicator to ensure the option names are unique to the plug-in.
- All options should be commented.

For example, the following options are set up for the "Embedded Strings Plugin" in the MASTIFF configuration file.

```
[Embedded Strings Plugin]
# Options for the Embedded Strings Plugin.
# strcmd is the path to the strings program
strcmd = /usr/bin/strings
```

A number of helper functions are located in the mastiff.conf module to assist with getting options from the configuration file. These functions are automatically imported into plug-ins with the "config" variable that is passed into the analyze() functions. At this point in time, category plug-ins do not receive the configuration file.

get_var(section, var)
Returns the specified variable within the specified section. If the variable does not exist, None is returned.

get_bvar(section, var)
Returns the specified boolean variable within the specified section, or None if the variable does not exist. Boolean variables may be specified in the configuration file as "1", "yes", "on", or "true" for True, or "0", "no", "off", or "false" for False.

get_section(section)
Returns a dictionary of the variables within the specified section, or None if the section does not exist. The key within the dictionary is the option name and the value is the option value.

set_var(section, var)
Sets the specified variable in the specified section. Returns None if the section does not exist.

Yapsy Requirements for Plug-ins

Yapsy is the Python module that allows the use of plug-ins within the framework. Full documentation for Yapsy is located at http://yapsy.sourceforge.net/.

When creating new plug-ins for MASTIFF, Yapsy requires two different files to be present:

- The plug-in Python source code, ending in .py. This file contains all of the code that will be run within the plug-in.
- The Yapsy plug-in information file, ending in .yapsy-plugin. This is a configuration file that contains parameters associated with the plug-in.

Both files are required for any plug-in to be recognized by the framework.

The Yapsy .yapsy-plugin file uses the Windows INI format and contains two sections, Core and Documentation. The Core section must contain the following options:

- Name: A short, unique descriptive name of the plug-in. This field is what the plug-in's self.name variable is set to.
- Module: The name of the .py file associated with this plug-in. For example, if the plug-in code file is named "myplugin.py", Module must be set to "myplugin".

The Documentation section contains information pertaining to the plug-in and does not affect performance at all. The following options should be set:

- Description: A short description of the plug-in. This should be no more than 80 characters long.
- Author: The author of the plug-in.

- Version: The version of the plug-in. When creating a plug-in, the version should be set to 0.1 until it has been finished and tested. At that point, the version should be set to 1.0, and incremented when changes are made.
- Website: The website of the author or where the plug-in can be found.

A sample .yapsy-plugin file is given below. The Python source code for the module is named "GEN-example.py".

```
[Core]
Name = Generic Plugin
Module = GEN-example

[Documentation]
Description = Generic Plugin Example
Author = Tyler Hudak
Version = 0.1
Website = www.korelogic.com
```

Helper Functions

A number of functions exist in the mastiff.plugins module to help with common tasks that plug-ins may need to perform. The functions are detailed below.

post_multipart(host, selector, fields, files)
Post fields and files to an HTTP host as multipart/form-data.

encode_multipart_formdata(fields, files)
Encodes the fields for an HTTP multipart/form-data POST. Used by the post_multipart() function.

get_content_type(filename)
Returns the content MIME type of the given file.

printable_str(str)
Converts non-printable characters in a string to their ASCII hex equivalent.

Queue Processing

Analysis plug-ions have the ability to submit newly discovered files back into the MASTIFF work queue for later analysis. An example of this is the ZipExtract plug-in, which feeds files from the zip archive back into MASTIFF so they can be analyzed.

In order to add files back into the queue, the queue library should be imported:

```
import mastiff.queue as queue
```

The MASTIFF queue must then be instantiated:

```
job_queue = queue.MastiffQueue(config.config_file)
```

Calling the append function can then be utilized to add a file back into the work queue.

```
job_queue.append(file_name)
```

Information on the mastiff.queue functions are detailed below.

queue.MastiffQueue(config)
The initialization routine for the MASTIFF queue. Takes the name of the config file as its parameter and returns the queue object.

queue.append(filename)
Adds a file to the work queue to be analyzed by MASTIFF. The filename should be the absolute path to the file.

Database Development

Plug-ins are encouraged to extend the database and add more tables or fields to store data in the database, although this is not required. The mastiff.sqlite module has been developed to make this easier, although at some point the plug-ins will be required to issue raw SQL statements to the database. Therefore, the following guidelines should be followed.

- Whenever possible, use the functions in the mastiff.sqlite module.
- The SQLite3 commit() function should always be run after the database has been updated.
- Extending the database to include new tables or fields is allowed. However, the plug-in should always check to ensure the tables or fields exist before attempting to read or update them.

Additionally, use caution when adding information into the database. Only store relevant information in the database.

Skeleton Plug-ins

Since plug-in development may not be an intuitive process, skeleton plug-ins have been created for both category and analysis plug-ins. These are located in the skeleton directory of the project. Developers can copy the appropriate plug-in skeleton, rename the file, and begin code development.

7.1 Category Plug-in Development

Category plug-ins are used to categorize analysis plug-ins based on the type of file that they analyze. The only time a category plug-in will need to be developed is when there is a new type of file to analyze. While the need to do this will be a rare, this section describes how to develop a category plug-in.

See the skeleton plug-ins available in the skeleton directory of the project for an example of a category plug-in.

Location

- All category plug-ins are to be placed in the "mastiff/plugins/category" directory of the MASTIFF Python library.

Naming Conventions

- The source code and Yapsy information files should be given a short, descriptive name of the type of file it categorizes. For example, the files for the category class for Adobe PDFs are "pdf.py" and "pdf.yapsy-plugin".
- The class for the category plug-in should have a name that is associated with the type of file it categorizes followed by the string "Cat". For example, the PDF category class name is "PDFCat".

Classes

- Most category classes should be derived from the MastiffPlugin class located in the mastiff.plugins.category.categories module. Therefore, an import statement for this module

should be present in every category plug-in.

- Category classes may also be derived from other category classes. For example, a Word category class could be derived from the Microsoft Office category class. When this occurs, the class should import the category class it is derived from. For example:

```
import mastiff.plugins.category.office as office

class WordCat(office.OfficeCat):
    """Category class for Microsoft Office files."""
```

Initialization

Yapsy calls the __init__() function of a plug-in class when the class is located, but does not allow parameters to be passed to it. Therefore, analysis plug-ins should only initialize member variables within the __init__() function.

The following variables are required to be initialized in the __init__() function:

- self.cat_name: This should be a one word description of the file type. e.g. PDF, Office, Generic, EXE, etc.
- self.my_types: This is a list of the types of files from libmagic or TrID the category can analyze. The easiest way to obtain this is to run the UNIX file command and TrID on the new type of file and copy the results.
- self.yara_filetype: This is the Yara rule utilized to determine the file type. This is set to None in the base class and should be set in the category plug-in.

Additionally, the __init__() function should call the __init__() function of its base class. e.g., MastiffPlugin.__init__() function.

Member Variables

The following is a list of all member variables category classes must have.

- self.cat_name: This should be a one word description of the file type. This must be initialized in the __init__() function.
- self.my_types: This is a list of the types of files from libmagic or TrID the category can analyze. This must be initialized in the __init__() function.
- self.name: The Yapsy information file option "Name". This is initialized by the Yapsy libraries and is inherited from MastiffPlugin.
- self.is_activated: Boolean variable that specifies if the plug-in is activated. Inherited from the category class and does not need to be set.
- self.yara_filetype: The Yara rule used to determine the file type.

Member Functions

The following is a list of all member functions a category class must have. Unless otherwise specified, they must be declared within the category class.

__init__(self, name=None)
Initializes the category class member variables and calls its base class __init__() function. e.g., MastiffPlugin.__init__(self, name).

is_my_filetype(self, id_dict, file_name)
Determines if the file being analyzed is of a type that its derived analysis plug-ins can analyze.

The id_dict variable it receives contains the results from libmagic and TrID. This function should compare the information in the dictionary to its known list of file types in self.my_types.

This function will also use the Yara file type rule, if it is present, to check the file type.

Additional checks can be done on the file to determine if it is a file that should be examined by the category. A successful result should return self.cat_name, None otherwise.

The id_dict dictionary contains:

- id_dict['magic']: A string generated from libmagic.
- id_dict['trid']: A list containing the percentage (as a float) and the string of the file identification from TrID.

More information on file-type identification can be found in section 4.3.

activate(self)
Calls IPlugin.activate() to activate the plug-in. Inherited from MastiffPlugin and does not need to be called or created.

deactivate(self)
Calls IPlugin.deactivate() to activate the plug-in. Inherited from MastiffPlugin and does not need to be called or created.

set_name(self, name=None)
A function to set self.name of the plug-in. Inherited from MastiffPlugin and does not need to be called or created.

7.2 Analysis Plug-in Development

Analysis plug-ins perform the static analysis techniques on the files being analyzed and place the data into a structure for output plug-ins. Each analysis plug-in should be written to work on one type of file and extract and analyze specific information from that file.

See the skeleton plug-ins available in the skeleton directory of the project for an example of an analysis plug-in.

Location

- Analysis plug-ins can be placed in any directory on the system. However, it is recommended to have one directory that all plug-ins are located in, and place them in sub-directories based on the type of file they analyze. This is only required if custom plug-ins are used; MASTIFF comes with a number of plug-ins that are detected by default.
- The plugin_dir option in the configuration file specifies the directories that analysis plug-ins are located. Plug-ins that come with MASTIFF are detected by default, and thus this can be left empty.

Naming Conventions

- The base name for analysis plug-in files should be the extension of the files analyzed (or short description) in upper-case, followed by the name of the plug-in. For example, the names of the Embedded Strings plug-in files are:

    - GEN-strings.py
    - GEN-strings.yapsy-plugin
- The analysis plug-in class name should be a mixed-case name that starts with the type of file it analyzes followed by a short description. For example, the class name of the Embedded Strings plug-in is "GenStrings".

Classes

All analysis plug-ins should be derived from the category class for the type of files they will analyze. If the plug-in will analyze all files, it should be derived from the GenericCat class.

Additionally, the plug-in should import the category class from the mastiff.plugins.category module location.

An example start of a generic analysis plug-in is:

```
import mastiff.plugins.category.generic as generic

class GenExample(generic.GenericCat):
    ...
```

At the time of this writing, the following category classes are available to analysis plug-ins:

Class	Location	File Types Analyzed
GenericCat	mastiff.plugins.category.generic	All files
EXECat	mastiff.plugins.category.exe	Windows Executables
PDFCat	mastiff.plugins.category.pdf	Adobe PDF documents
OfficeCat	mastiff.plugins.category.office	Microsoft Office files
ZipCat	Mastiff.plugins.category.zip	Zip Archive files

Initialization

Yapsy calls the __init__() function of a plug-in class when the class is located, but does not allow parameters to be passed to it. Therefore, analysis plug-ins should only initialize member variables within the __init__() function.

The __init__() function for an analysis plug-in must call the __init__() function of its category class.

Member Variables

Analysis plug-ins do not require member variables. However, the following is a list of member variables that are available to the plug-in.

- self.name: The Yapsy information file option "Name". This is initialized by the Yapsy libraries and is inherited from the category class.
- self.is_activated: Boolean variable that specifies if the plug-in is activated. Inherited from the category class and does not need to be set.
- self.prereq: A string containing the self.name of a plug-in that is required to run before this plug-in. This allows you to ensure specific plug-ins, and their output, have run and are

available before this plug-in is run. At this time, only one pre-requisite plug-in can be specified.

- self.page_data: The page (see Section 6.3) that will contain any data to be sent to the output plug-ins.
- self.results: Dictionary that contains all previous analysis output information.

Member Functions

The following is a list of all member functions an analysis class must have. Unless otherwise specified, they must be declared within the analysis plug-in class.

__init__(self, name=None)
Initializes the category class member variables and calls the __init__() function of its category class.

analyze(self, config, filename)
The main function of the plug-in that performs analysis on the given filename. The config variable contains the options from the MASTIFF configuration file. See section 4.3 on how to use this variable.

The analyze() function of an analysis plug-in should begin with the following code to verify the plug-in can run and log an initial message.

```
if self.is_activated == False:
    return
log = logging.getLogger('Mastiff.Plugins.' + self.name)
log.info('Starting execution.')
```

The analyze() function should return self.page_data which contains the data generated by the plug-in.

~~output_file(self, outdir, data)~~ **DEPRECATED**
~~This function will place any data from the analysis plug-in into a file in the directory specified by outdir. This function is required, but is never called by the MASTIFF framework. It must be called from the analyze() function.~~

activate(self)
Calls IPlugin.activate() to activate the plug-in. Inherited from the category class and does not need to be called or created.

deactivate(self)
Calls IPlugin.deactivate() to activate the plug-in. Inherited from the category class and does not need to be called or created.

Developers of analysis plug-ins should feel free to add any additional method functions required to make the plug-in work.

Output

Data generated by the analysis plug-in should be stored in the self.page_data member variable, which should be returned by the analyze() function.

Data generated by previously run analysis plug-ins is passed is available in an analysis plug-in as an attribute named results. This can be accessed from within the plug-in through "self.results". Note that modification of data within the self.results will modify previous data. Be very cautious if you modify any data in self.results.

The following code are guidelines for analysis plug-ins to ensure output plug-ins can access data correctly.

1. Set the filename of page_data. Any analysis plug-in whose output will be modified or required should be specified as a *prereq*.

```
def __init__(self):
      """Initialize the plugin."""
      gen.GenericCat.__init__(self)
      self.page_data.meta['filename'] = 'test example'
      # File Information output is required
      self.prereq = 'File Information'
```

2. Before data is modified, check to ensure the results are present.

```
      if self.results['Generic']['File Information'] is None:
          # File Information is not present, cannot continue
          log.error('Missing File Information plug-in output. Aborting.')
          return False
```

3. Data is stored in a "page". You may add tables to the analysis plug-in's page (self.page_data) or modify existing tables from other analysis plug-ins (after you ensure those tables exist).

3a. Sample code to add a new table to the current analysis plug-in:

```
      new_table = self.page_data.addTable('New Table')
      new_table.addheader([('header', str), ('header2', str)])
      new_table.addrow(['a1', 'a2'])
      new_table.addrow(['b1', 'b2'])
      new_table.addrow(['c1', 'c2'])
```

3b. Sample code to add a new table to an existing plug-in's output:

```
      if self.results['Generic']['File Information'] is not None:
          # adding a new table onto an existing page
          new_table = self.results['Generic']['File Information'].addTable('NEW TABLE')
          new_table.addheader([('header1', str), ('header2', str)])
          new_table.addrow(['a1', 'a2'])
          new_table.addrow(['b1', 'b2'])
          new_table.addrow(['c1', 'c2'])
```

3c. Sample code to add new data to an existing table in another analysis plug-in's output. Note the pages/tables must already exist.

```
      if self.results['Generic']['File Information'] is not None:
          # adding a new data onto an existing table
          my_table = self.results['Generic']['File Information']['File Hashes']
          my_table.addrow(['TEST HASH', 'ACKACKACKACKACK'])
```

4. Accessing an existing plug-in's data can be done by accessing that plug-in's page in self.results.

4a. Sample code to loop through all of the data from another analysis plug-in. Note that in the example below, Algorithm and Hash are named in the header in the File Hashes table. This is useful for looking at all the data from a plug-in and is probably the most common usage.

```
      if self.results['Generic']['File Information'] is not None:
          for row in self.results['Generic']['File Information']['File Hashes']:
              print row.Algorithm, row.Hash
```

4b. Sample code to find a specific piece of data.

```
    if self.results['Generic']['File Information'] is not None:
        # find specific data - this is a pretty convoluted list comprehension
        print [ myhash.Hash for myhash in self.results['Generic']['File Information']
['File Hashes'] if myhash.Algorithm == 'MD5'][0]
```

OR

```
        # this way may be easier to read
        for myhash in self.results['Generic']['File Information']['File Hashes']:
            if myhash.Algorithm == 'MD5':
                print myhash.Hash
                break # not necessary if you wish to keep going
```

4. Once analyze() is finished, self.page_data should be returned.

```
        return self.page_data
```

7.3 Output Plug-in Development

Output plug-ins format the data created by analysis plug-ins into a specific format. This removes the need for analysis plug-ins to perform their own output formatting and allows a universal output format that can be massaged into any format required.

See the skeleton plug-ins available in the skeleton directory of the project for an example of an output plug-in.

Location

- Output plug-ins should be placed in the mastiff/plugins/output directory.
- Directories containing test output plug-ins can be specified using the output_plugin_dir option in the MASTIFF configuration file.

Naming Conventions

- The base name for output plug-in files should be “OUTPUT-”, followed by the name of the plug-in. For example, the names of the Text output plug-in files are:
    - OUTPUT-text.py
    - OUTPUT-text.yapsy-plugin
- The output plug-in class name should be a mixed-case name that starts with “OUTPUT” followed by a short description. For example, the class name of the text output plug-in is "OUTPUTtext".

Classes

All output plug-ins should be derived from the masOutput.MastiffOutputPlugin class, where masOutput is the imported name of mastiff.plugins.output.

An example start of an output plug-in is:

```
import mastiff.plugins.output as masOutput

class OUTPUTtest(masOutput.MastiffOutputPlugin):
    ...
```

8 Analysis Plug-in Documentation

The following sections describe each of the analysis plug-in types that have been developed for MASTIFF at this time, detailing what they do, the output they generate and any requirements for them to function.

Note that not all plug-ins are output plug-in compliant; some still generate their own output. Ones that are output plug-in compliant are specified below.

8.1 Generic Plug-ins

Generic plug-ins perform analysis on every type of file examined by the MASTIFF framework, regardless of file type. This section describes the generic plug-ins that are a standard component of MASTIFF.

File Information Plug-in

This plug-in obtains information on the file being analyzed and stores it into the database in the “files” table.

This plug-in is output plug-in compliant.

The information obtained and stored is:

Data	Description
Filename	The filename, including path, of the file being analyzed.
Size	The file size in bytes.
First Seen	GMT date of when the file was first analyzed (in UNIX time stamp).
Last Seen	GMT date of when the file was last analyzed (in UNIX time stamp).
Times	The number of times this file has been analyzed.

Fuzzy Hashing Plug-in

Cryptographic hashes allow analysts to determine if two files are identical by comparing the hash “thumbprint” generated by each file. Fuzzy hashes, on the other hand, allow analysts to determine if two files are similar. In malware analysis, this can be used to determine if files are variants of one another or have been packed by the same packer.

This plug-in generates the fuzzy hash of the file being analyzed. The fuzzy hash is also compared to all fuzzy hashes already in the database. Any matches will be displayed.

Requirements

- ssdeep (http://ssdeep.sourceforge.net/)
- pydeep (https://github.com/kbandla/pydeep)

Output

- fuzzy.txt – This file will contain the fuzzy hash generated and any matches found.
- The 'fuzzy' field will get added to the 'files' table in the DB to store the fuzzy hash.

Hex Dump Plug-in

Examining the raw bytes in a hexadecimal view can often help analysts determine additional information about a file.

This plug-in generates a hex dump of the file in a view showing the hexadecimal values of the bytes as well as the ASCII characters for the bytes.

Requirements

- None

Configuration options

- enabled – [on|off] Since this plug-in can be resource intensive, it can be enabled of disabled in the configuration file. By default, this plug-in is disabled.

Output

- hexdump.txt – This file will contain the fuzzy hash generated and any matches found.

Embedded Strings Plug-in

Embedded strings are readable strings within a file that are meaningful to the analyst. They can reveal information such as where the malware will contact, what files or registry entries it modifies, or even who created the file.

This plug-in executes the UNIX/Linux 'strings' program and obtains the embedded ASCII and UNICODE strings within the file being analyzed. These will be put into a file that contains the hexidecimal offset of the string within the file, whether it is an ASCII string (denoted by 'A') or a Unicode string (denoted by 'U'), and the string itself.

Output

- This plug-in is output plug-in compliant.

Configuration options

- strcmd – Path to the strings executable.
- str_opts – Options to send to the strings executable for every option. This should be set to “-a -t d” (without the quotes). Do not change this unless you know what you are doing.
- str_uni_opts – Options to send to the strings executable to obtain embedded UNICODE strings. This should be set to “-e l” (without the quotes). Do not change this unless you know what you are doing.

Yara Plug-in

Yara is a tool analysts can use to identify and classify malware samples. The Yara plug-in allows the use of Yara rules to be run on the file being analyzed. The directory in which Yara rules are located is defined within a configuration option and all rules with the extension ".yar" or “.yara” are

applied to the file being analyzed.

Requirements

- Yara, libyara and yara-python must be installed. (http://code.google.com/p/yara-project)

Configuration options

- yara_sigs – Base path to Yara signatures. This path will be recursed to find additional signatures. Leave blank to disable the plug-in.

Output

- yara.txt – Output listing all matches found. This file will not be present if no matches were found.

Database

A new table named 'yara' will be created with the following fields:

Field	Type	Description
id	INTEGER	Primary key
sid	INTEGER	ID of file being analyzed
rule_name	TEXT	Name of the Yara rule matched
meta	TEXT	Yara meta information
tag	TEXT	Yara tag information
rule_file	TEXT	Full path to rule file match is from
file_offset	INTEGER	Offset in analyzed file match was found
string_id	TEXT	ID of match variable from Yara rule
data	TEXT	Data Yara rule matched on

VirusTotal Plug-in

VirusTotal is a website that runs a number of different anti-virus suites against files that are uploaded to it and displays the results.

This plug-in determines if the file being analyzed has been previously detailed on VirusTotal (http://www.virustotal.com). If it has, it obtains the results from the site and places them into a file. If the file has not been submitted to VirusTotal, the plug-in has the option to submit it.

This plug-in uses the VirusTotal (VT) API. Information on the API can be found at https://www.virustotal.com/documentation/public-api/.

Note: Unless special arrangements are made, VT will not let you send more than 4 queries in a 1 minute timeframe. You will receive VT error messages if you do.

Requirements

- A VirusTotal API key is required to be entered into the configuration file. This can be obtained from virustotal.com.
- The simplejson Python module must be present. (https://github.com/simplejson/simplejson)

Configuration Options

- api_key – Your API key from virustotal.com. Leave this blank to disable the plug-in.
- submit [on|off] – Whether you want to submit files to VT or not.

Output

- virustotal.txt – This file will contain the results from the VT retrieval or submission process.

Metascan Online Plug-in

Like VirusTotal, Metascan Online is a website that runs a number of different anti-virus suites against files that are uploaded to it and displays the results.

This plug-in determines if the file being analyzed has been previously detailed on Metascan Online (http://www.metascan-online.com). If it has, it obtains the results from the site and places them into a file. If the file has not been submitted to Metascan Online, the plug-in has the option to submit it.

This plug-in uses the Metascan Online API. Information on the API can be found at https://www.metascan-online.com/en/public-api.

Note: The default API key obtained from Metascan Online limits the number of queries you can perform against their site. If you receive any errors, you may have hit your limit.

Requirements

- A Metascan Online API key is required to be entered into the configuration file. This can be obtained from www.metascan-online.com.
- The simplejson Python module must be present. (https://github.com/simplejson/simplejson)

Configuration Options

- api_key – Your API key from metascan-online.com. Leave this blank to disable the plug-in.
- submit [on|off] – Whether you want to submit files to Metascan Online or not.

Output

- metascan-online.txt – This file will contain the results from the Metascan Online retrieval or submission process.

8.2 PDF Plug-ins

PDF plug-ins are those that perform analysis on files that are determined to be PDF documents. This section describes the PDF plug-ins that come with MASTIFF.

PDF-parser Plug-in

PDF documents are composed of objects – each with its own function. Analysts will often search PDFs for objects that are associated with malicious code, such as JavaScript objects or those that are automatically run when a document is opened.

Additionally, objects can be compressed to save space, but also to hide the data contained within them.

This plug-in uses Didier Stevens pdf-parser.py code to perform two tasks:

- Write an uncompressed copy of the PDF to a file named uncompressed-pdf.txt.
- Search the PDF for keywords in objects, and writes those objects, and any object they reference, to a file in pdf-objects/.

The current list of keywords looked for are: 'JavaScript', 'JS', 'OpenAction', and 'AA'.

Note that for large PDF documents, this plug-in may take a while to process. All rights for pdf-parser.py belong to Didier Stevens.

Requirements

- Didier Stevens pdf-parser.py must be installed. (http://blog.didierstevens.com/programs/pdf-tools/)

Configuration Options

- pdf_cmd – Path to pdf-parser.py
- feedback – Whether to feed uncompressed PDFs back into the MASTIFF queue for analysis. Valid values are on or off.

Output

- uncompress-pdf.txt – File containing the uncompressed version of the PDF being analyzed.
- pdf-objects/ – A directory containing any object deemed interesting by the list of keywords and any object they reference.

PDFId Plug-in

This plug-in runs Didier Stevens' pdfid.py script against the PDF being analyzed and places the results into a file. Doing so will generate statistics on the types of objects within the PDF.

By default, the plug-in (and pdfid.py) looks for the following tags and object types:

- /AA
- /AcroForm
- /Colors
- /EmbeddedFile
- /Encrypt
- endobj
- endstream
- /JavaScript
- /JBIG2Decode
- /JS
- /Launch

- obj
- /ObjStm
- /OpenAction
- /Page
- PDF
- /RichMedia
- startxref
- stream
- trailer
- xref

All rights for pdfid.py belong to Didier Stevens.

Requirements

- Didier Stevens' pdfid.py script must be installed (http://blog.didierstevens.com/programs/pdf-tools/)

Configuration Options

- pdfid_cmd – Path to the pdfid.py script.
- pdfid_opts – Options to give to the script. This may be empty.

Output

- pdfid.txt – Output from the execution of pdfid.py.

PDF Metadata Plug-in

Document metadata contains information about the document, such as who created it and when. This information is useful to analysts as attackers often forget to remove this data when creating malicious documents.

This plug-in extracts any metadata from the PDF using the exiftool program. Specifically, the following metadata is extracted:

- Creator
- Create Date
- Title
- Author
- Producer
- Modify Date
- Creation Date

- Mod Date
- Subject
- Keywords
- Author
- Metadata Date
- Description
- Creator Tool
- Document ID
- Instance ID
- Warning

Requirements

- The exiftool binary is required for this plug-in. (http://www.sno.phy.queensu.ca/~phil/exiftool/)

Configuration Options

- exiftool – Path to the exiftool program.

Output

- metadata.txt – The extracted metadata from the PDF document.EXE Plug-ins

8.3 EXE Plugins

EXE plug-ins are those that perform analysis on files that are determined to be Windows PE executables. This section describes the EXE plug-ins that MASTIFF supports.

PE Info Plug-in

The PE header is often analyzed during static analysis to determine such things as what APIs the file loads (and therefore its functionality), when the file was compiled, and if it is packed. Since this is a binary structure, tools are often used to translate it into readable format.

This plug-in obtains information on the structure of the PE header of the executable being analyzed using the pefile library.

Requirements

- pefile library (http://code.google.com/p/pefile/)

Output

- peinfo-quick.txt – This file contains minimal information that analysts may find useful and should be used as a quick glance at the file.
- peinfo-full.txt – This file contains the full information found within the PE header.

PE Resources Plug-in

Resources are files or data that have been added to an executable. Normally, these include icons, menus, dialog boxes, etc. However, malicious code will often attach a second malicious executable or configuration file as a resource. Additionally, resources will often reveal information about the malware, such as where or when it was created. Therefore, it is useful for analysts to examine any resource attached to an executable.

This plug-in obtains information on any resources contained within the PE executable being analyzed. The information obtained on the resource will include the resource name and ID, resource type, offset within the file of the resource, the size of the resource, the language, and the time stamp of when it was added to the executable.

Every resource found will also be extracted. Note that in packed files, the resource may be listed in the PE header, but not found in the file since it has been packed. If this occurs, the plug-in will display a warning message.

Requirements

- pefile library (http://code.google.com/p/pefile/)

Output

- resources.txt – File containing a list of all resources in the EXE and any associated information.
- resource/ – Directory containing any resource extracted from the executable.

Digital Signatures Plug-in

Windows PE executables can be digitally signed to confirm the identity of the software author and guarantee that the code has not been changed since it was signed. However, attackers have begun to sign malicious code using invalid, stolen or forged signatures.

This plug-in extracts any digital signatures from a PE executable and converts them to both DER and text format.

Extraction is performed using the disitool.py tool from Didier Stevens.

Conversion to text is performed using the openssl program.

Validation of the signature is not yet performed.

Requirements

- pefile library (http://code.google.com/p/pefile/)
- disitool.py (http://blog.didierstevens.com/programs/disitool/)
- openssl binary (http://www.openssl.org/)

Configuration Options

- disitool – Path to disitool.py script.
- openssl – Path to the openssl binary.

Output

- sig.der – DER version of Authenticode signature.
- sig.txt – Text representation of the signature.

Single-byte String Plug-in

Attackers will often attempt to obfuscate their embedded strings by moving a single byte at a time into a character array. In assembler, it looks like:

```
mov mem, 0x68
mov mem+4, 0x69
mov mem+8, 0x21
...
```

Using a strings program, these strings will not be found. This plug-in looks for any strings embedded in this way and prints them out. It does this by looking through the file for C6 opcodes, which are the start of the "mov mem/reg, imm" instruction. It will then decode it, grab the value and create a string from it.

Requirements

- distorm library (http://code.google.com/p/distorm/)

Configuration Options

- None

Output

- singlestring.txt – Output of any single-byte strings found

8.4 Office Plug-ins

Office plug-ins are those that perform analysis on files that are determined to be Microsoft Office documents; notably Microsoft Word, Excel, and Powerpoint.

At this time, these plug-ins only detect the OLE binary versions of Office documents. However, if the Office type is forced in MASTIFF using the -t option, these plug-ins will analyze the document.

Office Metadata Plug-in

Document metadata contains information about the document, such as who created it and when. This information is useful to analysts as attackers often forget to remove this data when creating malicious documents.

This plug-in extracts any metadata from an Office document using the exiftool program. Notably, the following metadata fields extracted are:

- Author
- Code Page (Language)
- Comments
- Company
- Create Date

- Current User
- Error
- File Modification Date/Time
- File Type
- Internal Version Number
- Keywords
- Last Modified By
- Last Printed
- MIME Type
- Modify Date
- Security
- Software
- Subject
- Tag PID GUID
- Template
- Title
- Title Of Parts
- Total Edit Time
- Warning

Requirements

- The exiftool binary is required for this plug-in. (http://www.sno.phy.queensu.ca/~phil/exiftool/)

Configuration Options

- exiftool – Path to exiftool program. NOTE: This is a different option than the option for the PDF metadata plug-in.

Output

- metadata.txt – The extracted metadata from the Office document.

pyOLEScanner Plug-in

This plug-in runs Giuseppe 'Evilcry' Bonfa's pyOLEScanner.py script against an Office document and looks for indications of malicious code. Specifically, it examines the document for:

- Known API names
- Malicious shellcode values

- Indications of an XOR'd executable
- Macros

All of these are indicative of a malicious document.

Output

- office-analysis.txt – File containing output from scan.
- deflated_doc/ – If Office document is an Office 2007 or later document, it will be deflated and extracted into this directory.

8.5 Zip Archive Plug-ins

Zip archive plug-ins are those that perform analysis on files that are determined to be zip archives. This includes normal zip archives, Java JAR files, Android APK files, and Microsoft OfficeX files. Note that these plug-ins only analyze the archive structure, and not the files within.

ZipInfo Plug-in

Zip archives contain metadata that describes data such as how the archive was created, the zipped files, and if the files are encrypted. This plug-in analyzes the structure of the zip archive and writes out information on it, including a file listing, modification dates, comments, and archive attributes.

Output

- zipinfo.txt – File containing all of the information from the archive.

ZipExtract Plug-in

This plug-in attempts to unzip an archive so its files can be analyzed.

Configuration Options

- enabled – [on|off] Enables or disables the plug-in.
- password – A password to use to unzip the archive. If the archive is not password protected, but a password is specified, then the archive will still be unzipped.
- Feedback – [on|off] Whether or not to add extracted files back into MASTIFF for analysis.

Output

- zip_contents/ – A directory containing the unzipped files. Note the directory structure within the archive is kept, but the date/times of the original files are not.

9 Output Plug-ins

The following sections describe each of the output plug-in types that have been developed for MASTIFF at this time, detailing the output format they generate and any requirements for them to function.

Raw Output Plug-in

This plug-in writes all of the data of all analysis plug-in's in their raw format that can be digested back into MASTIFF at a later time, if necessary. Output is placed into a single file.

In programmatic terms, this plug-in writes out the repr() format of the pages and tables to the file.

Requirements

- None

Configuration Options

- enabled – on or off value that controls if this plug-in runs

Output

- output_raw.txt – File containing raw output.

Text Output Plug-in

This plug-in writes analysis plug-in output to a text file. This plug-in has two modes – single and multiple.

In *single* mode, all output is written to a single text file. Plug-in data is separated by asterisks. Note that in single mode, the File Information plug-in output is always written first.

In *multiple* mode, output from each plug-in is written to its own file. The filename for the file is taken from the analysis plug-ins page_data.meta['filename'] field.

Requirements

- None

Configuration Options

- enabled – on or off value that controls if this plug-in runs
- format – [multiple|single] – value that controls which mode to output text in.

Output

- output_text.txt – File containing text output for single mode.
- Varies – In multiple mode, filenames will be determined based on the analysis plug-in's page_data.meta['filename'] field.