Global Information Assurance Certification Paper

Building a Malware Zoo

GIAC (GREM) Gold Certification

Author: Joel Yonts, jyonts@gmail.com
Advisor: Pedro Bueno

Accepted: December 31st 2009

Abstract

Today malware circulates in mass volume. New samples appear at a rate of thousands per day. In order to keep pace and manage this analysis demand two key needs emerge: automation and organization. This paper seeks to lay the foundation for a basic Malware Zoo that will provide a framework for both. Topics surveyed will include: basic schema design, sandboxing technology, and analysis techniques.

1. Introduction

In today's highly connected Internet age, we have seen an overwhelming flood of new malware. According to a report[1] published by McAfee (Marcus, Greve, Masiello, & Scharoun, 2009), over 12 million new pieces of malware were discovered in the first three quarters of 2009. This rate of thousands of new samples per day has exceeded our ability to manually analyze and catalog these threats. Additionally, maintaining a comprehensive library of samples and supporting analysis artifacts[2] has created an information organization nightmare.

In order to avoid being consumed by this tide of malware and analysis demand, we must embrace analysis automation (Bayer, 2009) and adopt an efficient information storage and retrieval system. A Malware Zoo combines these two concepts into a unified software solution. While many such zoos exist, most are confined within anti-malware product companies and private research organizations. Little is known about the structure and design of the majority of these systems.

The goal of this paper is to equip a wider audience with the ability to build their own Malware Zoo solutions. Our approach will begin with defining high-level zoo requirements. The attributes and requirements highlighted in this section should serve as a minimum with site-specific customization expected. Following a true SDLC[3] model, we will then move onto high-level design considerations. Design considerations will include pros and cons for choosing many architectural components such as platform, storage medium, and development language. Finally we will conclude with a series of type-specific analysis sections that will detail attributes and automated analysis techniques that are unique to specific types of malware.

[1] www.mcafee.com/us/local_content/reports/7315rpt_threat_1009.pdf
[2] Collection of characteristics, behaviors, and attributes of a malicious sample
[3] Software Development Life Cycle

Joel Yonts, jyonts@gmail.com

2. Malware Zoo Requirements

In order to tackle the information organization and automation requirements we must first understand malware analysis methodology. More specifically we must understand the attributes, analysis techniques, and analysis artifacts that give insight into the origin, capabilities, and characteristics of a malicious sample.

The exposition of this list of techniques and data points is beyond the scope of this paper but luckily we can leverage the large volume of published work[4] to build our requirements.

2.1. Organization of Information

At the heart of zoo organization is the concept of malware sample management. Sample management includes storage and retrieval capability of the original sample, sample attributes, and the various artifacts generated through the analysis of the sample. A strong search capability is also needed that can perform complex searches across the universe of information stored within the Malware Zoo. Lastly, including a capability to store a control group of non-malicious samples can be invaluable in the benchmarking of malicious behavior/attributes against non-malicious samples. The following requirements table expands these high level concepts into a more detailed list of requirements.

[4] A list of malware analysis sources is noted within the reference section

Table Zoo Requirements: Attributes & Artifacts

- Ability to store and retrieve binary and text sample files
- Ability to remove samples and supporting information

- Storage and search of sample attributes:
  - Collection Date
  - Collection Source
  - Zoo Submission Date
  - Target Platform
  - User Defined Tag
  - Randomization Metrics
  - Comment Field
  - File Size
  - File Type
  - MD5 Hash
  - Fuzzy Hash
  - Type Specific Attributes

- Storage, search, and retrieval of infection artifacts:
  - Dropped Files
  - Network Addresses
  - Network Protocols
  - OS Specific Changes
  - Embedded Strings
  - Unpacked Binaries
  - Shell Code
  - Code Fragments
  - Deadlistings
  - Type Specific Artifacts

- Storage, search, and retrieval of analysis artifacts:
  - Network Trace Files
  - System Trace Files
  - IDA Pro DB Files
  - Screenshots
  - Analysis Notes
  - Articles & References
  - Misc Analysis Artifacts

- Storage and search by signature detections of sample:
  - Packer Identification
  - Network IPS Detection
  - AV Signature Detection
  - Host IPS Detection

- Ability to upload and maintain a separate control set of known good samples
  - Storage and retrieval of non-malicious samples & sample attributes

Table 2.1.1: Basic Malware Zoo Requirements

2.2. Automated Malware Analysis

In many ways the automated analysis zoo component is the most critical. Without a good analysis engine it would be near impossible to populate all the fields we previously defined for all samples. This engine should support both static and dynamic analysis implemented with a layer of isolation and other safeguards.

Joel Yonts, jyonts@gmail.com

The analysis engine will be the most demanding zoo component from a system perspective since it will involve execution of many CPU, memory, and disk intensive tasks. Scalability will be an important requirement for keeping pace with the analysis demand. Additionally, the tasks involved with analyzing malware are constantly changing and expanding to new tools and methodologies. This dynamic nature of malware analysis drives the need for a modular and easily updated design. The following requirements table expands these high level concepts into a more detailed list of requirements.

ZOO REQUIREMENTS: AUTOMATED ANALYSIS

- Modular design that will support adding and removing analysis tasks

- Provide static analysis capabilities:
  - Determine File Size
  - Determine File Type
  - Determine Randomization
  - Extract Embedded Strings
  - Type Specific Analysis
  - Determine MD5 Hash
  - Determine Fuzzy Hash
  - Signature Detection of Packer Type
  - Signature Based AV Detection

- Provide OS / Application specific analysis environments for dynamic analysis
- Support concurrent analysis of malware samples

- Provide dynamic analysis capabilities:
  - Collect & analyze network events associated with infection
    - Determine network addresses & protocols
    - Automated collection of IP & domain registrar information
    - Full network packet capture
  - Collect & analyze operating system events associated with infection
    - Determine new and modified files
    - System configuration changes
  - Type-specific analysis capabilities

- Scalable design that can spread analysis tasks across multiple systems
- Isolation of dynamic analysis environments to ensure containment
- Monitoring and alerting safeguards to ensure detection of containment issues

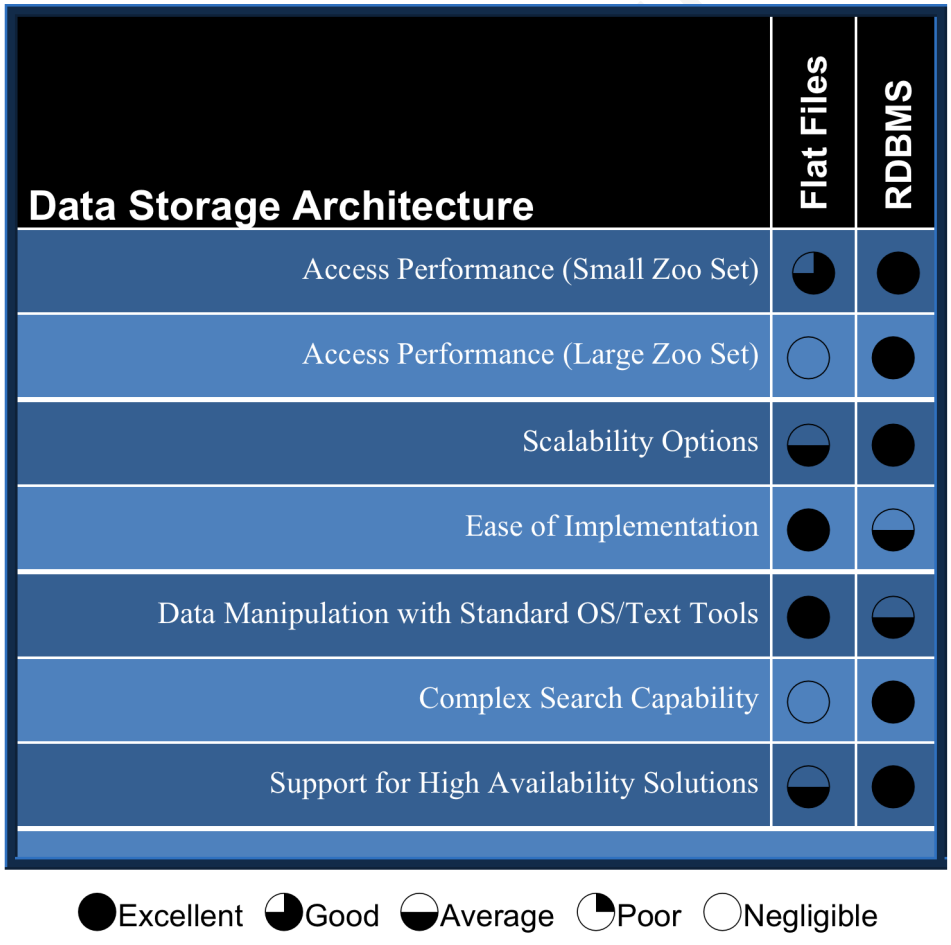
Table 2.2.1: Automated Malware Analysis Requirements

3. Zoo Design Considerations

Designing a Malware Zoo that delivers the requirements outlined in the previous section involves making a series of architectural decisions. Each of these architectural choices have benefits and consequences. The intent of this section is to outline design options along with supporting pros and cons that empower you to create a zoo design that best fits your needs.

3.1. Data Storage & Retrieval

There is a wide array of options available for delivering the information storage capabilities needed for a Malware Zoo. For our discussion we will limit the focus to two technologies: Flat Files and RDBMS storage systems. Even though our focus is limited, the same decision principles can be applied to other available options.



Joel Yonts, jyonts@gmail.com

As you can see from the previous table, utilization of a flat file[5] storage system is an ideal choice for smaller zoo sets that make use of standard text tools and shell scripts to store, manipulate, and retrieve information. This architecture usually enables a quicker time-to-market for initial implementation and requires less application development experience. The main disadvantage of flat file storage systems is reduced performance for larger zoos.

A RDBMS[6] based storage system is a natural choice for larger zoos that need improved data access performance, higher availability, and complex searching capability. It may take longer to implement such as system and require additional developer and DBA skills but it should enable a more feature-rich and responsive Malware Zoo.

3.2. Platform Considerations

Platform choice is more a matter of preference that aligns with comfort level and experience. Technologies that may be part of a Malware Zoo implementation such as databases, programming/scripting languages, web servers, and other tools exists on all major platforms.

Microsoft Windows, however, may be at a disadvantage as a zoo platform choice. The disadvantage comes in the form of additional difficulty in maintaining containment of samples due to the targeted (Wildlist Organization International, 2010; Aquilina, Casey, & Malin, 2008) nature of this platform. This disadvantage can be overcome by implementation of additional safe guards but may still be an important point to weigh when choosing your zoo platform.

3.3. Analysis Node Architecture

Zoo analysis nodes fall into two categories: static and dynamic. Static nodes are used to run various tools against the sample with the intent of extracting attributes without executing the malware or triggering an infection. These nodes are tool centric with the primary design consideration being how to combine like toolsets into shared static nodes.

[5] Flat file systems make use of normal operating systems files to store information
[6] **R**elational **D**atabase **M**anagement **S**ystem

Dynamic nodes shift the focus to a malware centric view where environments are built to cater to the target execution environment of the samples. As one can imagine, the number of dynamic nodes could become quite expansive considering all of the application/middleware/operating system combinations possibly targeted by malware. Picking an architecture that can efficiently support an ever-expanding population of dynamic nodes is the first design challenge. The next design challenge is providing an ability to "reset" a node back to a known-good, pre-infected state. This functionality is critical since the end result of any successful dynamic (behavioral) analysis run is a fully infected analysis node.

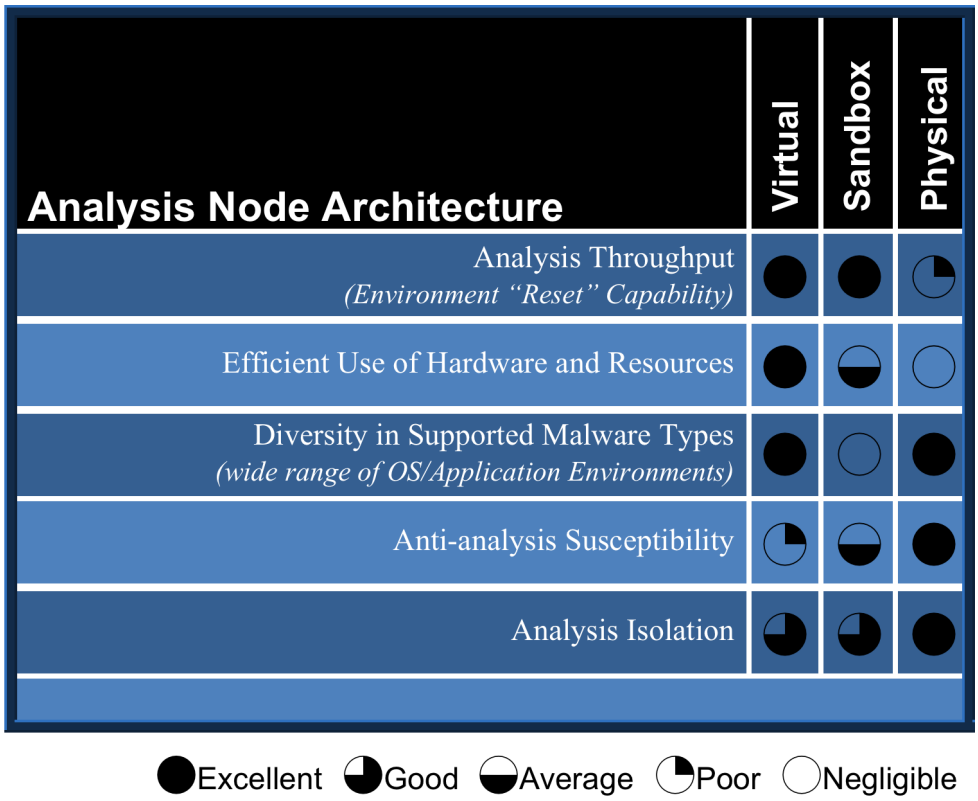
Solving the challenges outlined above often begins with choosing an appropriate system architecture. Virtualization technology is a good system architecture for promoting efficient resource utilization across a vast array of unique analysis nodes. Most virtualization solutions also have the much-needed "reset" capability implemented in the form of snapshot and revert. Virtualization architecture is not without fault, however. Malware authors often implement VM detection in their malware as a way of detecting and subsequently derailing analysis efforts conducted by researchers.

Use of physical systems as analysis nodes is the purest environment for analyzing malware with the most protection against anti-reversing techniques. The drawback to this system architecture is poor scalability (especially if the mapping of unique analysis environments to physical systems is a one-to-one ratio) and a very slow and cumbersome "reset" capability.

A third option for system architecture exists in the form of sandboxing technology. Some sandboxing solutions are already purpose built malware analyzers. These systems require no additional design or development and can be added directly to your zoo environment. Examples of this type of sandboxing technologies are TRUMAN, *Norman Sandbox*[7] and *CW Sandbox*[8]. Other sandboxing technology such as *DeepFreeze*[9] seeks to simply provide an environment with

[7] http://www.norman.com/technology/norman_sandbox
[8] http://www.sunbeltsoftware.com/Malware-Research-Analysis-Tools/Sunbelt-CWSandbox
[9] http://www.faronics.com/en/Products/DeepFreeze/DeepFreezeEducation.aspx

enhanced isolation and "reset" to pre-infected state. The table below provides a



summary of analysis node design considerations.

An ideal dynamic analysis engine would allow for multiple analysis nodes with varying node types. This design could take advantage of virtualization and gain speed and versatility benefits but also include a limited number of native nodes to test for VM detection malware.

3.4. User Interface Selection/ Layout

A good user interface can have a tremendous impact on an analyst's ability to effectively and efficiently analyze samples. Often a balance is needed, or in some cases dual interfaces, that support a quick point-n-click interaction while not loosing the capabilities of batch upload, low level manipulation, and application integration interfaces. Also, the user interface must be able to operate in the restricted

environment created through the various isolation layers implemented within a Malware Zoo

Utilization of a thin client web presentation for applications has emerged as a de facto standard for many enterprise applications. With portability and support for a vast array of clients leading this movement, adopting this UI technology will ensure you can access from the broadest range of clients and devices. Additionally, thin client technology functions well within a highly restricted environment due to the inherit separation of presentation client from the server. The main pitfall to this UI architecture is the possibility of accidently sending a malware's browser exploit code to the zoo UI screen. If this architecture is chosen, additional safeguards must be implemented to scrub all UI output.

Traditionally, thick clients or local GUI based applications could provide a more dynamic and rich user experience but new web technologies such as DHTML and AJAX combined with server side Java and PHP has closed this gap.

Depending on the integration and usability requirements there may be multiple secondary interfaces and APIs implemented with a wide array of email, web, messaging, and network based protocols. When choosing these additional interfaces, isolation should always be considered. The principle to keep in mind is each interface is a conduit by which loss of containment may occur.

3.5. Access Restrictions & Safe Guards

The greatest risk to maintaining a Malware Zoo is the loss of containment. In this scenario a malicious sample somehow breaks out of the controlled storage and analysis environment and infects the host system or other systems on the network. In order to prevent this, a strong layer of isolation is required. Isolation can and should take the form of the following:

- Segregated analysis network
- Separate (from host[10]) physical systems or virtual machines for analysis

[10] Host refers to the primary system housing the data storage and UI components

- Selection of a host platform that is not vulnerable to the majority of malware samples (Selecting UNIX platform to store Win32 Malware)
- Display options for analysis artifacts should be scrubbed to avoid UI exploitation (Browser Exploits, Malicious JavaScript, iFrames, etc.)

With the best of safe guards things can still go wrong! Adding an integrity monitoring and alerting system may save you from an embarrassing incident. A comprehensive integrity system should look for potential compromise in any of the zoo components: host systems, analysis network, data storage system, and the user interface. Needed integrity monitoring capabilities may include:

- Monitoring of firewall and/or packet capture logs to ensure proper network containment
- Maintaining host system file integrity utilizing a hash matching or similar integrity validation technique
- Routine on-demand anti-virus scan of host systems (excluding memory and file system locations housing malicious samples)
- Validation of database user accounts, permissions, and other administrative structures against a normalized base line
- Ensure client side browsers are configured properly and current on patches

Finally, the debated practice of allowing malware to "phone home" to real Internet sites may have it's value points but may not have a place in an automated environment. If this is a practice you incorporate into your analysis processes, carefully consider keeping this a manual task where you are actively watching the information exchange. Too many variables are present to truly account for them all with automation. The two greatest concerns would be the infection of innocent downstream Internet users and the possibility of allowing an attacker a backdoor into your analysis environment.

4. MyZoo: An Example Malware Zoo

With so many options and considerations it is easy to fall into analysis paralysis and delay the building of a working solution. To avoid this issue this section will

define an example zoo design and provide basic implementation details. This section may serve as a blueprint to jump-start your Malware Zoo development.

The design of this Malware Zoo, called MyZoo, will follow a LAMP[11] model common to many open source solutions. The base OS will be a Linux[12] distribution running a MySQL[13] database for data storage and Apache/PHP[14] for a user interface. Analysis nodes will utilize a mix of VMWare[15] virtual machines and physical systems. To complete the solution, Python scripting will be used to tie the various system components together into a heterogeneous solution. The specific details of the solution stack is outlined below:

Architecture Stack	
Host Platform	Suse Linux v11.3
Data Storage	MySql v5.1
User Interface	PHP / Apache
Analysis Nodes	- Linux, Win32, Mac OS X nodes - Virtual & Physical Nodes
Virtualization	VMWare Workstation 6.5
Programming Language	Python 2.6

Table 4.0.1: Example Malware Zoo Architecture

4.1. Host System Design

The host system(s) will serve as the core for the Malware Zoo. This system(s) will serve the user interface, store the samples, and control the analysis flow. Properly configuring the host system components of the zoo includes developing a file system structure to house the zoo, establishing trust relationships for submitting samples to analysis nodes, and configuring of access restrictions to the web interfaces. Below is the configuration used for MyZoo.

Zoo File Structure	
$ZOO_HOME	/usr/local/zoo
Binary Files	$ZOO_HOME/bin

[11] Development model incorporating **L**inux, **A**pache, **M**ysql, and **P**HP
[12] http://www.opensuse.org/en
[13] http://www.mysql.com/about
[14] http://www.apache.org/
[15] http://www.vmware.com/products/workstation

Joel Yonts, jyonts@gmail.com

Zoo Configuration	$ZOO_HOME/etc
Sample Storage	$ZOO_HOME/samples
	$ZOO_MAINT/maint
Temporary Space	$ZOO_HOME/uploads
HTML & PHP UI Files	$ZOO_HOME/www

Table 4.1.1: MyZoo File Structure

The file structure utilizes an application "HOME" directory that houses a *bin* (Binaries), *etc* (Configuration), and *www* (Web Content) layout. The *uploads* directory serves as a temporary storage point for uploaded files. Uploaded files are quickly consumed by the zoo and moved to the appropriate directory or database structure.

The *samples* directory serves as the storage point for malicious samples. An alternate design option was to store the files as binary objects within the database. The file system storage option was chosen because it would greatly simplify and improve the performance for reanalyzing the entire sample library. To better organize the *samples* directory, a hierarchy was adopted that spreads the sample library over a large number of directories whose names were derived from a portion of the samples MD5 hash.

Sample Library Directory Hierarchy

```
./samples/ec78/ec783c6cb3e1e0f66a7c2122b8ed7575
./samples/5635/5635121eefe47333d00fff1fd4a5021f
./samples/0207/02077e4935994fd69813887fb0df6195
./samples/5acf/5acfaa3f96c54670780488f5e415cc57
./samples/31a8/31a8756b48576862e6312bdc063fa94b
./samples/999d/999dfed7d3f90ba5d11b7f6364be98f4
./samples/bf90/bf902e1e439797f409a9be40880e8bd2
./samples/aea9/aea9c27b8103b17bd5da95d41a2fc274
./samples/ec78/ec783c6cb3e1e0f66a7c2122b8ed7575
```

... truncated ...

Table 4.1.2: MyZoo Sample Directory Structure

This approach sufficiently randomizes the samples across a large number of potential directories. As the population of samples grows and additional organization becomes needful, an additional layer of directories can easily be added utilizing the next grouping of characters in the samples hash. The beauty of this

solution is that the exact storage location can be quickly calculated from the samples hash. Also, note that the actual sample file is renamed to match the samples MD5 hash. This prevents potential name collision when analyzing the sample and the original submission may still be retained as an attribute in the sample database. Table 4.1.3 shows proof-of-concept Python source code for implementing a hash-based directory hierarchy.

TABLE 4.1.3 PYTHON SCRIPT FOR MANAGING FILE ARTIFACTS STORAGE

```
import os

# Parent Directory for Zoo Sample Storage
SAMPLE_DIR="/zoo/samples"

def storeFileArtifact (fileData, md5Hash):
    # Construct and Create (if necessary) sample directory
    archiveDir =SAMPLE_DIR+"/"+md5Hash[:4]
    if os.path.exists(archiveDir) == False:
        os.makedirs(archiveDir)

    # Store sample
    zooName =archiveDir+"/"+md5Hash
    fStore=open(zooName, "w")
    fStore.write(fileData)
    fStore.close()

    return zooName

def getFileArtifact (md5Hash):
    # Construct sample directory
    archiveDir =SAMPLE_DIR+"/"+md5Hash[:4]
    if os.path.exists(archiveDir) == False:
        return False

    # Retrieve sample data
    zooName =archiveDir+"/"+md5Hash
    sampleData=open(zooName, 'r').read()

    return sampleData
```

†Error checking omitted for brevity

4.2. Data Storage & Retrieval Configuration

The core of MyZoo is a MySQL database. The database will house all data elements and artifacts with the exception of samples and the control group. In order to efficiently store this voluminous and diverse group of data, a database schema that normalizes the data into a grouping of tables with indexes built to optimize access is needed. The following is the schema used to build the example zoo.

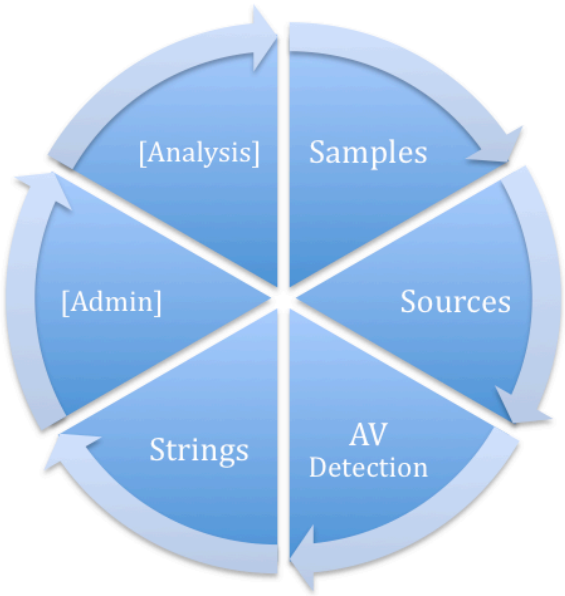


Figure 4.2.1: Directory of Tables for the Zoo Database

The sample table stores the basic attributes of the sample. These attributes are either included with the sample submission, available from OS functions or available through simple static analysis tools.

SAMPLE TABLE		
sample_id	*INT(XX), PRI KEY*	Unique identifier
tag	*VARCHAR(XX)*	User defined tag for
type	*VARCHAR(XX)*	Sample type (Predefined Types)
Platform	*VARCHAR(XX)*	Platform targeted by sample
size	*BIGINT(XX)*	Sample size (bytes)
md5	*CHAR(XX), UNIQUE*	MD5 hash of sample
entropy	*FLOAT*	Randomness of Sample Data
zoo_date	*DATE*	Date sample was submitted
comment	*VARCHAR(XX)*	User defined comment

Joel Yonts, jyonts@gmail.com

Table 4.2.1: MyZoo Schema for Samples Table

When samples are collected there are many interesting but often fluid pieces of information that can be collected. Items like "where the sample originated" or "what the sample was named when it was on a particular system" falls into this category of interesting but not necessarily unique to a given sample. To accommodate this many-to-one relationship between source information and a sample, this information is stored in a separate sources table. This table is linked with a specific sample table entry using the MD5 hash as a primary key.

SOURCES TABLE		
md5	*CHAR(XX), PRI KEY*	MD5 Hash of Sample
orig_name	*VARCHAR(XX)*	Original Filename of Sample
tag	*VARCHAR(XX)*	User Defined Tag
source	*VARCHAR(XX)*	Sample Source: URL, System, Email
path	*VARCHAR(XX)*	File System Path, IP Address, etc.
collection_date	*VARCHAR(XX)*	Date of Sample Collection
case_id	*VARCHAR(XX)*	Case ID Associated with Sample
comment	*VARCHAR(XX)*	User Defined Comment

Table 4.2.2: MyZoo Schema for Sources Table

Anti-malware product vendors often lead the forefront in research and identification of malware. Getting a read from various anti-malware vendors on their assessment of a given sample is a crucial piece needed to link your analysis with the research of colleagues throughout the world. Table 4.2.3 presents the schema used in MyZoo for storing this information once collected.

AV_DETECTIONS TABLE		
md5	*CHAR(XX)*	MD5 Hash of Sample
av_name	*CHAR(XX)*	Anti-Malware Vendor Name
av_detection	*VARCHAR(XX)*	Detection String Returned or NULL
date	*Datetime*	Date of Sample Scan
av_version	*VARCHAR(XX)*	Scan Engine and Signature Versions

Table 4.2.3: MyZoo Schema for AV Detections Table

Whether intentionally or by accident, malware authors often leave many clues embedded in unencoded strings within a sample. These pieces of information range from an author's hacker name to hosts intended for use as command and

control. Often the strings can also give away the country of origin for a particular sample. Since this data is unstructured the schema for storage is very loose and simple.

STRINGS TABLE		
md5	CHAR(XX)	MD5 Hash of Sample
strings	TEXT	Embedded String
date	DATETIME	Date of String Extraction

Table 4.2.4: MyZoo Schema for Strings Table

In addition to the tables already defined there are two additional groups of tables. The first can be generally labeled as ADMIN tables. These are various tables that are used in the backend operations of the zoo. Tables of this category include, but are not limited to, zoo configuration table, a logs table that tracks all zoo events (submissions, sample extraction, etc.) and a jobs table where various zoo tasks can be scheduled or submitted.

The last grouping of tables is the analysis tables. These tables store the various artifacts extracted during analysis of the sample. Since these tables are so closely tied to the analysis process we will defer defining these tables until we reach the various analysis sections documented in later sections of this paper.

4.3. Managing Analysis Node Submissions

Managing analysis nodes can be a considerable amount of work. Samples must be submitted to these nodes, the analysis process must be initiated, and the results must be collected in such a way that the data can be incorporated into the zoo database. Having all of these tasks driven from the zoo engine server (houses the database and serves the web UI) is asking for potential performance and containment issues.

A better approach would be to offload all of these administrative tasks onto a separate analysis node controller. This controller would be responsible for the following functions:

- Submitting analysis tasks to nodes
- Managing the analysis process on the remote nodes
- Collect resulting analysis information

- Methods for the zoo engine server to retrieve analysis information
- Manage the state of the analysis nodes (reset, patch, rebuild, etc.)

Additionally, network access controls can be implemented so that analysis nodes can only communicate with the controller and the controller can only communicate with the zoo engine through a secured SSH connection. This layer of isolation provides a powerful addition to our layers of containment.

4.4. Scripting Language

Malware analysis is a vast topic and the tools list required to analyze the gambit of sample types is expansive. These tools include scripts, operating system commands, custom applications, and forensic tools. In order to avoid the MyZoo implementation from devolving into a loose collection of utilities, the Python[16] scripting language will be used to bind the various tools and techniques together into a cohesive solution. The various analysis techniques highlighted in the forthcoming analysis sections will be presented as either native Python or system commands wrapped in Python functions. The intent of these scripts is to provide a template for implementing zoo analysis functionality. Actual zoo implementation should include additional data parsing, object oriented development, and error checking functionality.

[16] http://www.python.org/

4.5. User Interface Design

As mentioned in section 4.1, the LAMP development model was selected for implementation of MyZoo. The user interface components of LAMP translate to the use of HTML, Apache, and PHP to create a thin-client based interface. Actual screen design is beyond the scope of this paper but generally the screens identified in Table 4.5.1 were implemented as part of the MyZoo conceptual zoo.

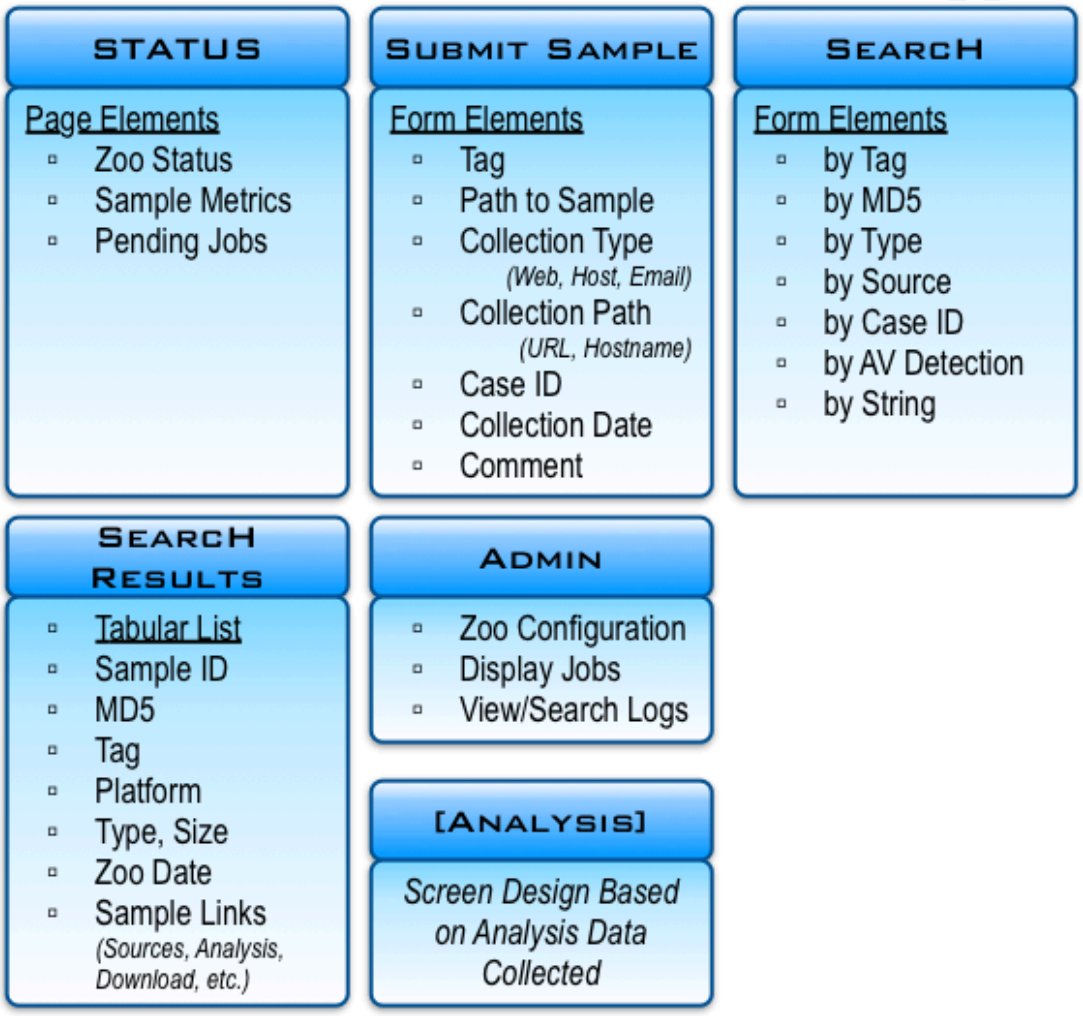


Table 4.5.1: Conceptual Screen Layout for My Zoo

4.6. Design Summary

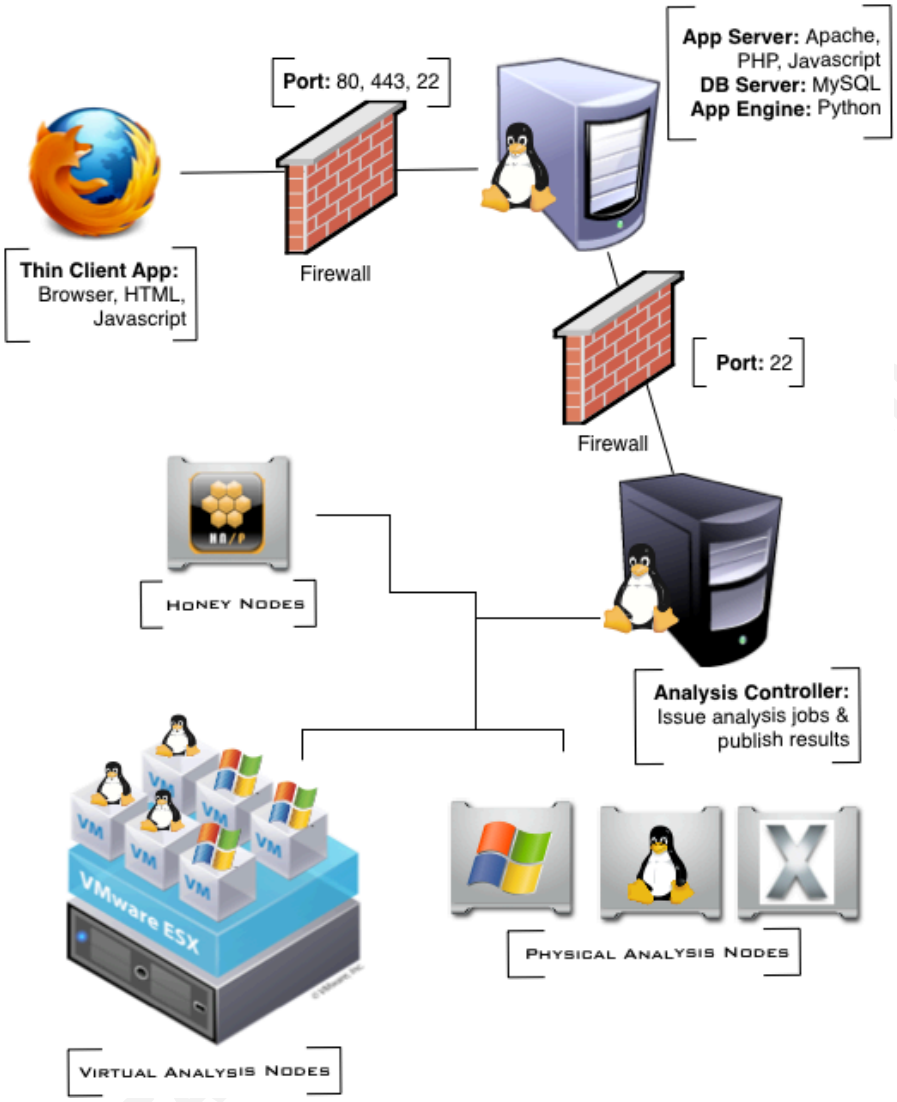


Figure 4.5.1: MyZoo Conceptual Design

Figure 4.5.1 pulls the various design elements outlined in this section into an integrated system view. Note the addition of an analysis controller. This system serves as a bridge between the zoo and the various nodes. By implementing this additional system, we can offload the management of the analysis process and the analysis nodes from the primary zoo systems. This is beneficial for system resource management and follows as modular design philosophy.

Joel Yonts, jyonts@gmail.com

5. Static Analysis

Often the first step in analyzing a potentially malicious sample is to pass the sample through a series of static analysis tools. Static analysis tools analyze the sample's file structure and/or contents without executing the sample. These analysis processes establish the basic attributes of each sample as they are submitted to the zoo collection.

5.1. Collecting General Sample Attributes

General Attributes	
• Size	• Sample Type
• Hash	• Strings
• Fuzzy Hash	• Entropy

Table 5.1.1: Common Attributes that Exists Across all Sample Types

The general attributes listed in Table 5.1.1 transcends platform and exists regardless of sample type. The processes for collecting these attributes run the gambit of simple size (Table 5.1.2) calculations to the more advanced fuzzy hashing and entropy calculations. This section will document static analysis techniques for obtaining each of these attributes and present a potential schema for storing these data elements.

TABLE 5.1.2: DETERMINING SAMPLE SIZE

```
import os

def getSize(filename):

    # Return file size
    size =os.path.getsize(filename)
    return size
```

Sample Type

Determining sample type is a foundational step in beginning an automated analysis process. Type will largely dictate the path the sample will take through the analysis process and defines what information will be stored about this sample. There are multiple options for determining sample type but one of the best is the

tried and true *NIX *file* command. This command exists natively on all *NIX derived operating systems[17] and has an extremely simple syntax.

TABLE 5.1.3 DETERMINING FILE TYPE

```
import commands

def getFileType(filename):

    # Execute "file -b <filename>"
    results=commands.getstatusoutput("file -b "+filename)
    return results[1] # [0]=Execution Status, [1]=CMD Output
```

A good augment to the *file* command is the verbose *TrID*[18] utility by Marco Pontello.

TABLE 5.1.4 EXTENDED IDENTIFICATION USING TRID UTILITY

```
import commands

def getTrID(filename):

    # Execute "trid –d: -b <filename>"
    cmd="trid -d:"+TRID_DEFS+" "+filename
    results=commands.getstatusoutput(cmd)
    return results[1] # [0]=Execution Status, [1]=CMD Output
```

Cryptographic Hash

In the world of malware samples where filenames can change in milliseconds the cryptographic hash reigns supreme as the method for uniquely identifying samples. Specifically MD5 and SHA1 hashes are the most common hashes used in sample identification. Table 5.1.5 provides options for computing file hashes.

[18] http://mark0.net/soft-trid-e.html

TABLE 5.1.5 COMPUTE HASHES

```
import hashlib

def getHash(filename):
    data=open(filename, "r").read()

    # Compute MD5 and SHA1 hashes
    md5hash=hashlib.md5(data).hexdigest()
    sha1hash=hashlib.sha1(data).hexdigest()

    # Build return structure
    results=[md5hash, sha1hash]

    return results
```

Another form of hashing called fuzzy hashing has emerged as a method for comparing the similarities between two samples. Fuzzy hashing uses a Context Triggered Piecewise Hashing method (Kornblum, 2006) for comparing files on a section by section basis. As part of this process each file entering the zoo would be compared against the existing fuzzy hash database for a potential match. Matches are not absolute, hence the "fuzzy" concept, but rather measured on a scale from 0-100. Regardless of the outcome, the zoo's fuzzy hash database would be extended to incorporate the fuzzy hash of this new sample. Fuzzy hashing can potentially identify malware families, common packers, and other functional similarities.

TABLE 5.1.6 COMPUTE FUZZY HASHES AND FUZZY MATCHES

```
import commands

def getFuzzyHash(filename):

    # Compute fuzzy hashes and fuzzy matches
    cmd="ssdeep -bm "+FUZZY_DB+ " "+filename
    match=commands.getstatusoutput(cmd)

    cmd="ssdeep -b " +filename
    fuzzyHash=commands.getstatusoutput(cmd)

    # Add hash to database for future match
    addFuzzyHash2DB(fuzzyHash)

    # Build return stucture
    results=[fuzzyHash, match]

    return results
```

Embedded Strings

Numerous methods exist for extracting string data. A popular option is the *NIX strings command. This command exists on all major *NIX variants and the Mac OS X platform. This command could be a good choice for extracting string data but may suffer performance issues due to potentially passing large volumes of data through an external program call (external to the zoo's Python engine). An alternate option that may alleviate this performance concern is to construct a native Python function for extracting string data. Table 5.1.7 shows proof-of-concept code for implementing a strings utility in native Python.

Joel Yonts, jyonts@gmail.com

TABLE 5.1.7: EXTRACT EMBEDDED STRINGS

```
import curses.ascii

def extractStrings(fileName):
    frag=""
    strList=[]
    bufLen=2048
    FRAG_LEN=4 # Min length to report as string

    fp=open(fileName, "rb")

    offset=0
    buf=fp.read(bufLen)
    while buf:
        for char in buf:
            # Uses curses library to locate printable chars
            # in binary files.
            if curses.ascii.isprint(char)==False:
                if len(frag)>FRAG_LEN:
                    strList.append([hex(offset-len(frag)),frag])

                frag=""
            else:
                frag=frag+char

            offset+=1
        buf=fp.read(bufLen)
    return strList
```

Sample Entropy

Another interesting sample attribute is the measure of entropy or randomness in the data. Generally, the higher the entropy the greater the chance the sample is either compressed or encrypted. (Kendall & McMillan, 2007).

TABLE 5.1.8: COMPUTATION OF DATA ENTROPY[19]

```
import math

def getEntropy(data):
    """Calculate the entropy of a chunk of data."""
    if not data:
        return 0

    entropy = 0
    for x in range(256):
        p_x = float(data.count(chr(x)))/len(data)
        if p_x>0:
            entropy += - p_x*math.log(p_x, 2)

    return entropy
```

Anti-Malware Signature Detections

A great addition to a zoo analysis system is the static analysis of the sample by an array of Anti-Malware products. The best approach for zoo implementation would be to leverage the command line scanner tools included with most AV products and turn off On-Access scanning. These products can be installed on individual analysis nodes or an attempt can be made to consolidate them onto a reduced number of analysis nodes.

Zoo Storage of General Sample Attributes

The MyZoo schema described in Tables 4.2.1 through 4.2.4 accommodates the storage of most of the attributes described in this section with the exception being the storage of the *TrID* data and fuzzy hashing results.

To accommodate the *TrID* data a text field can be added to the sample table or a separate table can be built with the special purpose of housing this data. The advantage of the separate table approach is improved performance of sample table access due to reduced record size and reduced record fragmentation.

[19] Source code taken from Ero Carrera's *pefile*: http://code.google.com/p/pefile

SAMPLE DESCRIPTION TABLE		
md5	*CHAR(XX)*	MD5 Hash of Sample
trid	*TEXT*	Output of *TrID* utility

Table 5.1.9: Sample Description Schema

The other advantage of the sample description table approach is that it can be leveraged to store other text information about the sample. This other data could be anything from general notes entered by the analyst to the output of future commands that will be added as an expansion to our zoo's functionality.

The other element that was not included in the original database schema is the fuzzy hashing results. Efficiently storing and managing the fuzzy hash relationships can be challenging due to the many-to-many relationships represented. First, a table that stores the "fuzzy matches" between samples could take the form of the table described in Table 5.1.10.

FUZZY MATCHES TABLE		
md5	*CHAR(XX)*	MD5 Hash of Sample #1
md5	*CHAR(XX)*	MD5 Hash of Sample #2
fuzzy_score	*TINY*	Numeric score (0-100) of match
date	*Datetime*	Date of Fuzzy Hashing

Table 5.1.10: Schema for Storing Fuzzy Matches

Each match that achieved a configurable threshold score would be entered as a separate entry in this table. A second table could store the growing fuzzy hash database used in the fuzzy hashing process.

FUZZY HASH DB TABLE		
md5	*CHAR(XX)*	MD5 Hash of Sample
fuzzy_hash	*CHAR(XX)*	Fuzzy Hash of Sample

Table 5.1.11: Schema for Maintaining the Fuzzy Hash Database

Since the primary fuzzy hashing tool outlined in this section, *ssdeep*, requires the hash database to reside in a file, a process may need to be created to continually update/dump a local file copy of this database table.

5.2. MS Packed Executable Analysis

The PE file format contains many potentially interesting structural components for malware analysis. While there are numerous fields available for analysis, our

focus will be limited to those attributes that will point directly to malicious activity or that will assist us in further analysis of the sample.

PE Header Analysis

The foundational structures of the PE file format[20] are the PE Headers. These data structures house many file attributes that are of interest to the malware analyst. Table 5.2.1 represents a potential set of attributes used during the analysis process.

PE Header Attributes Used in Malware Analysis	
PE File Header • Machine • Number of Sections • Pointer To Symbol Table • Number of Symbols • Size of Symbols • Size of Optional Header • Characteristics **Export Directory Table (EDT)** • Name RVA • Ordinal Base • Address Table Entries • Number of Name Pointers • Export Address Table RVA • Ordinal Table RVA **Import Library Table (ILT)** • Ordinal/Name Flag • Ordinal Number • Hint/Name Table RVA	**Optional Header** • Magic • SizeOfCode • SizeOfInitializedData • SizeOfUninitializedData • EntryPoint • BaseOfCode • BaseOfData • ImageBase • SizeOfImage • SizeOfHeaders • DLLCharacteristics • NumberOfRVAsAndSizes **Import Directory Table (IDT)** • Import Lookup Table RVA • Name RVA • Import Address Table RVA **Load Configuration Structure** • SecurityCookie • SEHandlerTable • SEHandlerCount

Table 5.2.1: Relevant PE Header Attributes

Since PE file analysis is a mature area of malware analysis, many options exist for extracting these elements of a sample. In keeping with the Python roots of

[20] http://www.microsoft.com/whdc/system/platform/firmware/pecoff.mspx

our MyZoo design, a Python based toolset called *pefile*[21] provides a good programmatic capability for extracting this information.

TABLE 5.2.2: EXTRACTION OF PE ATTRIBUTES

```
import pefile
import peutils

class PE:
    def __init__(self,filename):
        self.pe=pefile.PE(filename)
        return

    def getDOS_HEADER(self):
        return self.pe.DOS_HEADER

    def getFILE_HEADER(self):
        return self.pe.FILE_HEADER

    def getOPTIONAL_HEADER(self):
        return self.pe.OPTIONAL_HEADER

    def getPE_TYPE(self):
        return self.pe.PE_TYPE

    def getDIRECTORY_ENTRY_IMPORT(self):
        return self.pe.DIRECTORY_ENTRY_IMPORT

    def getDIRECTORY_ENTRY_EXPORT(self):
        return self.pe.DIRECTORY_ENTRY_EXPORT
```

PE Identification

Another feature of the *pefile* toolset is the ability to match a PE structure against a database of known PE signatures[22]. Included in the database are signatures for many packers, compilers, and other development tools.

[21] http://code.google.com/p/pefile

[22] PE signature functionality and signature database is based on the PEiD utility

TABLE 5.2.3: PE IDENTIFICATION

```
import pefile
import peutils

def peIdentify(filename):
    # Load PE Signature Database & Sample PE
    sigs=peutils.SignatureDatabase(SIGS_DB)
    pe=pefile.PE(filename)

    # Match PE against signature database
    matches=sigs.match_all(pe, ep_only=True)

    return matches
```

Zoo Storage of PE Attributes

Expanding our example database schema (MyZoo) to store PE attribute information is a fairly straightforward process. A general PE attribute table would serve as storage for most of the attributes.

PE ATTRIBUTES DB TABLE		
Machine	*CHAR(XX)*	Target CPU Platform
NumberOfSections	*CHAR(XX)*	Number of Sections
TimeDateStamp	*CHAR(XX)*	Creation Date & Time
PointerToSymbolTable	*CHAR(XX)*	Symbol Table Location
NumberOfSymbols	*CHAR(XX)*	Size of Symbol Table
... truncated[23] ...		

Table 5.2.4: DB Schema for Storing PE Attributes

The primary PE attributes not stored in the PE Attributes table are function imports and function exports. These elements again fall into the many-to-one relationship status which is best organized into a separate table.

IMPORT & EXPORT DB TABLE		
md5	*CHAR(XX)*	MD5 Hash of Sample
type	*TINYINT*	Import or Export
fct_name	*CHAR(XX)*	Function Name
fct_address	*CHAR(XX)*	Function Address
fct_ord	*TINYINT*	Ordinal Number of Function

Table 5.2.5: DB Schema for Storing Imports and Exports

[23] PE Table Schema was Truncated Due to Schema Length and Repetitive Definition.

The import/export table would contain one entry for each function that is either imported or exported with the use of an MD5 hash of the sample to link all entries back to a specific sample.

5.3. Mac OS Binary Analysis

With the rise in Macintosh market share the need for basic Mac binary analysis has also risen. The Mach-O binary format[24] is the Mac equivalent of the MS Windows PE file format. Mach-O has many of the same structures that we analyzed in the previous MS Packed Executable section. Mach-O structures of interest include headers, imports, exports, shared libraries, and the symbol table. Tables 5.3.1 provides a sample Python script for extracting these attributes using the *otool*[25] utility.

TABLE 5.3.1: EXTRACTION OF MACH-O ATTRIBUTES

```
import commands

class MachO:
    def __init__(self, filename):
        self.filename=filename

    def callSystemCmd(self, cmd):
        results=commands.getstatusoutput(cmd)
        return results[1] #[0]=Execution Status, [1]=CMD output

    # otool –h <filename>
    def getMachHeader(self):
        cmd="otool -h "+self.filename
        results=self.callSystemCmd(cmd)
        print results
    # otool –f <filename>
    def getFatHeader(self):
        cmd="otool -f "+self.filename
        results=self.callSystemCmd(cmd)
        print results
```

[24] http://developer.apple.com/mac/library/documentation/DeveloperTools/Conceptual/MachORuntime/Reference/reference.html

[25] *otool* is part of Apple's Xcode Development Suite: http://www.apple.com/developer

Joel Yonts, jyonts@gmail.com

TABLE 5.3.1: EXTRACTION OF MACH-O ATTRIBUTES (CONTINUED)

```
# otool –L <filename>
def getLibraries(self): # External Libraries
    cmd="otool -L "+self.filename
    results=self.callSystemCmd(cmd)
    print results

# otool –l <filename>
def getLoadCmds(self): # Segment & Section Information
    cmd="otool -l "+self.filename
    results=self.callSystemCmd(cmd)
    print results

# nm -a <filename>
def getSymbolTable(self):
    cmd="nm -a "+self.filename
    results=self.callSystemCmd(cmd)
    print results
```

Storage of the Mach-O attributes and elements follows a similar schema design as to what was developed in section 5.2. The header attributes that exist as a one-to-one relationship with the sample can be stored within a Mach Attributes table. File imports, exports, libraries, and symbols would be stored in separate tables. These additional tables can be Mach-O specific or the equivalent tables used for storing imports, exports, etc. for the PE files can be generalized to house both types.

Since the development of Mach-O attribute database tables so closely resembles those already defined in section 5.2, sample Mach-O tables will not be included in this section.

5.4. ELF Binary Analysis

For the sake of completeness, this section will describe the techniques and attributes involved with analyzing ELF Binaries. ELF is the most common binary executable format utilized by the Linux OS.

The ELF binary format has the same major structures that we saw in the PE and Mach-O formats. This list of structures include an ELF header, segment & section information, dynamic libraries, and a symbol table. Tables 5.4.1-5.4.4 describe methods for extracting these attributes.

Joel Yonts, jyonts@gmail.com

TABLE 5.4.1: EXTRACTION OF ELF ATTRIBUTES

```
import commands

class ELF:
    def __init__(self, filename):
        self.filename=filename
        return

    def callSystemCmd(self, cmd):
        results=commands.getstatusoutput(cmd)
        return results[1] #[0]=Execution Status, [1]=CMD output

    # readelf –h <filename>
    def getHeaders(self):
        cmd="readelf -h "+self.filename
        results=self.callSystemCmd(cmd)
        return results

    # readelf –S <filename>
    def getSegments(self):
        cmd="readelf -S "+self.filename
        results=self.callSystemCmd(cmd)
        return results

    # readelf –d <filename>
    def getLibraries(self):
        cmd="readelf -d "+self.filename
        results=self.callSystemCmd(cmd)
        return results


    # readelf –s <filename>
    def getSymbolTable(self):
        cmd="readelf -s "+self.filename
        results=self.callSystemCmd(cmd)
        return results
```

Storage of the ELF attributes follows a similar methodology and schemas as outlined in section 5.2 and 5.3 so sample ELF tables with not be included in this section.

5.5. Analysis of Other Sample Types

In sections 5.2 through 5.4 we covered the major executable binary formats that are utilized by malware developers. These formats represent malware threats that are targeted at the operating system level. Malware, however, often attacks at the

application layer so we must broaden our capabilities to accommodate analysis of this new category of malware.

Before examining some common application layer malware types we should take a moment to discuss the dangers of incorporating this type of malware into our zoo. The greatest danger with application layer malware is that the applications exploited by the malware may be included as part of the operation of your zoo solution. The most obvious is browser-based malware infecting the browser used to access your Malware Zoo. Another example could be the use of Microsoft Office tools to view zoo reports on a sample that turns out to be a Microsoft Office virus. Additional care needs to be taken to ensure isolation of this category of samples.

Malicious PDFs

Malware delivered via Malicious PDF[26] is currently king amongst document type malware. A number of tools exist to parse these documents with the intent of mapping the malicious content. Some of the most prominent tools are the *pdf-parser* and *pdfid*[27] tools written by Didier Stevens. These are Python based tools that provide extensive parsing and identification capabilities for this document type.

TABLE 5.5.1: EXTRACTION OF PDF ATTRIBUTES

```
import commands
import pdfid

class PDF:
    def __init__(self, filename):
        self.filename=filename
        return

    def callSystemCmd(self, cmd):
        results=commands.getstatusoutput(cmd)
        return results[1] #[0]=Execution Status, [1]=CMD output

    # pdfid.py <filename>
    def getObjectTypes(self):
        results=pdfid.PDFiD(self.filename) # Extract Ob Count
        rtn=pdfid.PDFiD2String(results, True) # Format Results
        return rtn
```

[26] http://www.adobe.com/devnet/pdf/pdf_reference.html
[27] http://blog.didierstevens.com/programs/pdf-tools

TABLE 5.5.1: EXTRACTION OF PDF ATTRIBUTES (CONTINUED)

```
# Since pdf-parser.py is written in Python a native
# integration similar to the pdfid example above is
# possible. For this example pdf-parser.py was called
# as an external program

# pdf-parse.py --hash <filename>
def getObjectHash(self):
    cmd="pdf-parser.py --hash "+self.filename
    results=self.callSystemCmd(cmd)
    return results

# pdf-parse.py <filename>
def getObjects(self):
    cmd="pdf-parser.py "+self.filename
    results=self.callSystemCmd(cmd)
    return results

# pdf-parse.py --object <#> <filename>
def getObjectByNum(self, ObjNum):
    ObjParm=" --object " +str(ObjNum) +" "
    cmd="pdf-parser.py"+ObjParm+self.filename
    results=self.callSystemCmd(cmd)
    return results
```

Image Malware

Another example of application layer malware is malicious image files. With this type of malware, exploit code embedded within the image takes advantage of an image viewer flaw to gain remote execution capabilities. The primary goal of the remote execution is usually to extract malware from an embedded image object and install it on the local system. Detecting these embedded objects is one key in analyzing this type of malware. *stegdetect*[28] is a free tool by Niels Provos that aids in the detection of hidden objects within images. Table 5.5.2 provides source for implementing *stegdetect* within our example zoo.

[28] http://www.outguess.org/detection.php

Joel Yonts, jyonts@gmail.com

TABLE 5.5.2 HIDDEN OBJECTS IN IMAGE FILES

```
import commands

def detectHiddenObj (imagefile):

    # Utilize stegdetect for detecting objs embedded in images
    cmd="stegdetect "+ imagefile
    results=commands.getstatusoutput(cmd)

    return results
```

In the MyZoo example, behavioral analysis would be used to further analyze any malicious image files flagged by the detection function.

Analysis Expansion Opportunities

Many opportunities exist to expand the zoo analysis capabilities outlined in this section. With the framework built and with the intended modular design, additional analysis capabilities should be easily added. For the sake of brevity some analysis capabilities were omitted that may be target areas for future expansion. These areas include malicious JavaScript, Office Macro, and Mobile Device malware analysis.

5.6. Summary

Static analysis is a great first step in the malware analysis process. The previous static analysis sections highlighted the various techniques for building a basic static analysis capability. Since these techniques were presented in the form of python functions, rolling them together into a cohesive solution is greatly simplified.

TABLE 5.6.1: PROCESSING SAMPLE

```
def processSample(filename):

    # Basic Sample Attributes
    fileTypeStr =getFileType(filename)
    basic_attr['type']  =getFileType(filename)
    basic_attr['size'] = getSize(filename)
```

Joel Yonts, jyonts@gmail.com

TABLE 5.6.1: PROCESS SAMPLE (CONTINUED)

```
md5,sha1= getHash(filename)
basic_attr['md5'] =md5
basic_attr['sha1'] =sha1
basic_attr['fuzzyhash']=getFuzzyHash(filename)
basic_attr['entropy']=getEntropy(filename)

# DB function for inserting attribute data
# Param1: <table name>
# Param2: <data to insert into DB>
dbInsert("SAMPLE", basic_attr)

# Parse PE File
if fileTypeStr.count("PE32") >0:
    pe=PE(filename)
    dbInsert("PE_ATTR", pe)

# Parse Mach-O File
elif fileTypeStr.count("Mach-O") >0:
    macho=MachO(filename)
    dbInsert("MACHO_ATTR", macho)

# Parse ELF File
elif fileTypeStr.count("ELF") >0:
    elf=ELF(filename)
    dbInsert("ELF_ATTR", elf)

# Parse PDF File
elif fileTypeStr.count("PDF") >0:
    pdf=PDF(filename)
    dbInsert("PDF_ATTR", pdf)
```

Notice in the script above, after calling each of the analysis functions, all attribute data is stored in the zoo database. This allows a decoupling of data collection from presentation and reporting of the data. Table 5.6.2 lists a sample PHP script used to generate a web report of basic attributes for a given sample.

TABLE 5.6.2: PHP SCRIPT FOR GENERATING A SAMPLE REPORT

```
// MYSQL Connect and DB select
$dbc = mysql_connect($dbhost, $dbuser, $dbpass);
mysql_select_db($DB_NAME);

$query = 'SELECT * FROM samples where md5="'.$MD5.'"';
if ($r = mysql_query($query)) { // Run the query
    if($row = mysql_fetch_assoc($r)) {
        print "<TABLE class=\"attrTable\">";
        print "<CAPTION>Basic Attributes</CAPTION>";

        // Parse Query Results & Build HTML Report
        foreach ($row as $key => $value) {
            print "<TR>";
            print "<TD class=\"attr\">".$key.":</TD>";
            print "<TD>".$value."</TD>";
            print "</TR>";
        }
        print "</TABLE>";
    }
}
```

Figure 5.6.3 shows a report generated using the PHP script shown above.

BASIC ATTRIBUTES

SAMPLE_ID: 1227
TAG: Koobface
MD5: 18395e9476bde417692f3a7ab807ac44
SHA1: 699004ee6fed330cead8532b5e80c98bf27eaac6
PLATFORM: Win32
TYPE: MS-DOS executable PE for MS Windows (GUI) Intel 80386 32-bit
SIZE: 40960
ENTROPY: 3.82781953111
ZOO_DATE: 2009-10-26
COMMENT: Sample retrieved from site spammed through facebook wall post

Figure 5.6.3: Sample Zoo Report for a Koobface sample

Joel Yonts, jyonts@gmail.com

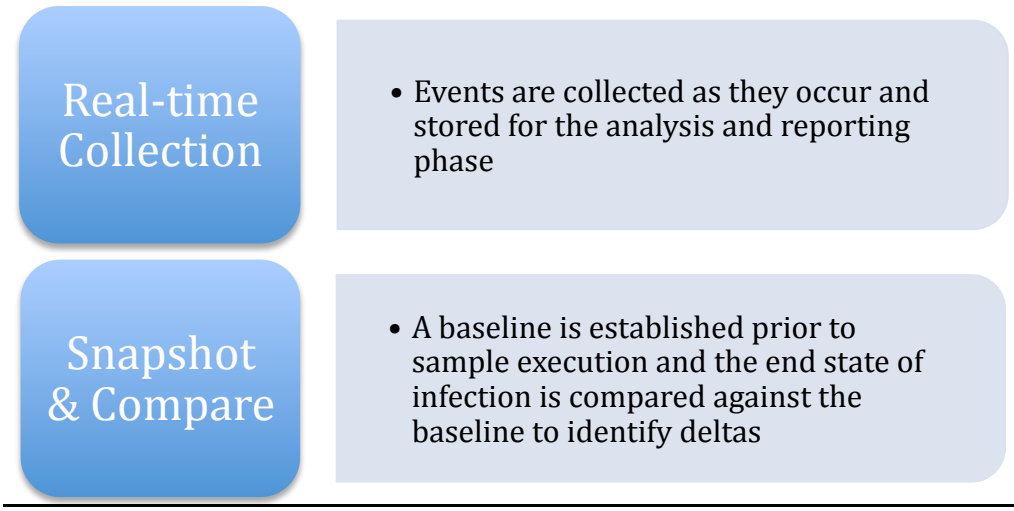
6. Behavioral Analysis

Behavioral analysis is a vast topic that we will approach lightly with a focus on an overview of processes used in behavioral analysis, interesting behavioral analysis artifacts and collection methods, and a general approach to artifact storage. This section will leverage the behavioral analysis environments that were outlined in sections 3.3 and 4.3.

6.1. Analysis Overview

Examining malware while it is executing is far different than simply looking at a sample at it's resting state. In the static analysis processes described in section 5 we were in the command seat and could execute the analysis commands and collect the output at the pace we dictate and in our prescribed sequence. With behavioral analysis the approach changes to allowing the malware to do what it wants (within our controlled environment) while the analyst captures the relevant data in more of a spectator type position. Obviously the processes of gathering this information needs to have an additional layer of automation and coordination because we are no longer controlling the analysis flow.

The analysis techniques used in this section falls into one of the following two categories.



Real-time collection is usually implemented on the system being infected with some level of hooking being implemented to catch system calls in order to maintain

a running list of activities. While this is transparent to most applications, malware that contains anti-analysis functionality may be able to detect the presence of these tools and skew the analysis results.

Snapshot & Compare suffers less from the anti-analysis techniques as described above since the tools are not running during the main infection process. This technique also has gaps since there could be a wide range of activities that are transient and not represented in either the before or after snapshots. Also, the malware infection may implement Rootkit technology that prevents the after infection snapshot from getting an accurate picture of the system.

A good automated analysis environment leverages both types of analysis techniques to gain the widest visibility and improve the likelihood of closing the gaps mentioned above.

6.2. Analysis Artifacts

At a summary level, when malware executes, the artifacts of infection involve a combination of network activity, file system changes, and operating system modifications. These general categories take on a vastly different implementation depending on the operating system and malware type being analyzed. For the sake of brevity we will focus on implementing behavioral analysis for MS Windows PE samples but the same principles and techniques would apply to the majority of malware samples available. Figure 6.2.1 represents basic attributes of interesting when conducting behavioral analysis of MS Windows PE samples.

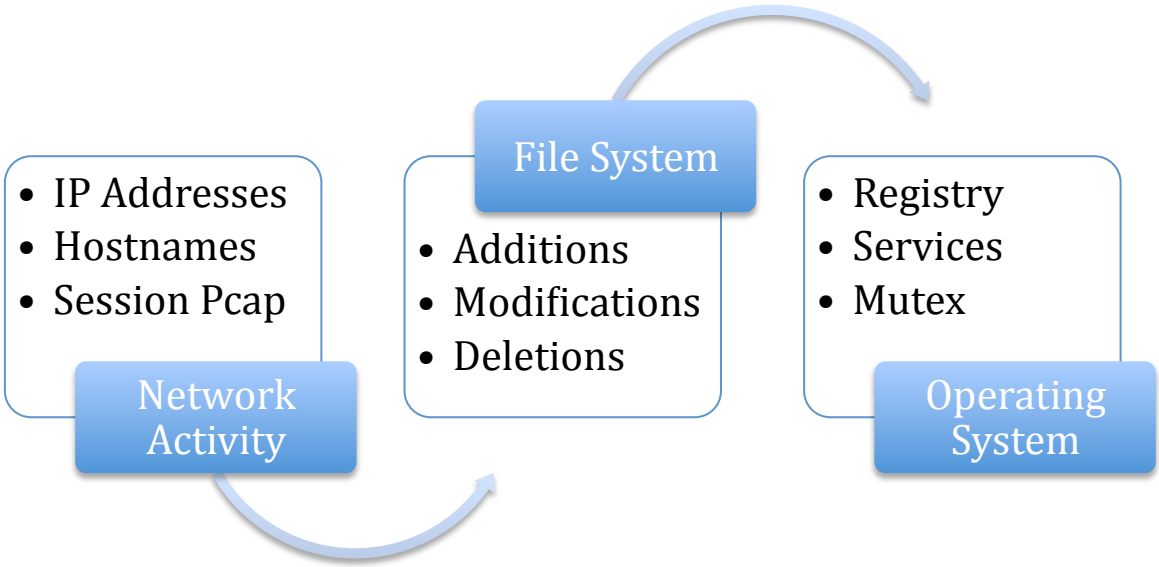


Figure 6.2.1: Artifacts of Windows PE Behavioral Analysis

Collecting Events in Realtime

There are many tools available for tracking system modifications. Popular options include *Process Monitor*[29] by Sysinternals (Microsoft) and the open source tool *Process Hacker*[30]. For the sake of automation however, a non-graphical tool may be a better option. *Capture-Bat*[31] is a tool developed by the New Zealand Honeynet project that provides real-time monitoring of networking and file system events. Table 6.2.1 shows the syntax for invoking the tool and provides a sample log file.

TABLE 6.2.1: CAPTURE-BAT SYNTAX & OUTPUT

```
$ CaptureBat –c –n –l output.log
$
$ cat output.log

"22/4/2010 8:24:48.722","file","Write","C:\WINDOWS\bill108.exe",
"C:\Documents and Settings\Administrator\Local Settings\Temporary
Internet Files\Content.IE5\SDAVWL67\hostsgb3[1].exe"

"22/4/2010 8:24:48.722","file","Write","C:\WINDOWS\bill108.exe",
"C:\Documents and Settings\Administrator\Local Settings\Temporary
Internet Files\Content.IE5\SDAVWL67\hostsgb3[1].exe"
```

[29] http://technet.microsoft.com/en-us/sysinternals/bb896645.aspx
[30] http://processhacker.sourceforge.net
[31] https://www.honeynet.org/node/315

TABLE 6.2.1: CAPTURE-BAT SYNTAX & OUTPUT (CONTINUED)

```
"22/4/2010 8:24:48.737","file","Write","C:\WINDOWS\bill108.exe",
"C:\Documents and Settings\Administrator\Local
Settings\Temp\zpskon_1271948510.exe"

"22/4/2010 8:24:48.800","process","created","C:\WINDOWS\bill108.exe",
"C:\Documents and Settings\Administrator\Local
Settings\Temp\zpskon_1271948510.exe"
```

Snapshot & Compare

In the field of Computer Forensics a core objective is to identify what files and registry keys have been added, modified, accessed, and removed on a suspect system. Since this is the same information we need for our malware analysis, the use of forensic tools may be a good addition to our automated analysis environment. Table 6.2.2 shows a snippet[32] of a forensic timeline gathered during a recent malware investigation.

TABLE 6.2.2: FORENSIC TIMELINE SNIPPET

```
<Columns Omitted>   macb c:/WINDOWS/system32/captcha.dll
<Columns Omitted>   ...b c:/WINDOWS/Prefetch/RDR_1271939089.EXE-2AF9B881.pf
<Columns Omitted>   m.c. c:/WINDOWS/system32
<Columns Omitted>   mac. c:/WINDOWS/Prefetch/RDR_1271939089.EXE-2AF9B881.pf

... output truncated for brevity ...
```

Each line in a timeline is composed of several columns of attributes that represent a unique filesystem event. The second to last column of the timeline represents the file operations executed against the file object listed by the full path column to the far right. A simplified interpretation of the file operation attributes is listed in Table 6.2.3.

m	*Modification*	Change in File Content
a	*Access*	File Access
c	*Change*	Change in File Metadata (Permissions, Owner, etc.)
b	*Born Date*	File Creation

Table 6.2.3: Filesystem Operations(Carrier, 2005)

The *sleuth kit* is an OSS solution that makes timeline creation available to the masses. This tool is available on the *Sleuth Kit* website[33] or as part of the *Helix*

[32] Non-relevant timeline attributes were omitted for formatting purposes.
[33] http://www.sleuthkit.org

Forensics Toolkit[34]. Figure 6.2.4 shows the process of creating a forensic timeline using Sleuth Kit.

TABLE 6.2.4: FORENSIC TIMELINE CREATED USING SLEUTHKIT

```
$ fls –r –m c: /dev/<disk name> > events.fls
$
$ mactime –b events.fls > events.timeline
```

Within the context of a zoo solution, a timeline could be created for each analysis node prior to infection. Post-infection, a new timeline would be created and the two would be compared to identify filesystem events related to the infection.

6.3. Honey Node

A large part of malware's behavioral footprint involves its interaction with the Internet and LAN hosts. Observing these interactions in an isolated environment requires creating a potentially complex network of systems and services targeted by a sample. To solve this complex challenge, Honey Nodes[35] can be implemented within the analysis environment to emulate a long list of network services. Once implemented, these Honey Nodes can be leveraged to observe malware dropper functionality, SPAM capabilities, and other malware propagation methods. A good source for Honey Node technology is the Honeynet Project[36]. This site serves as a collection point for a number of subprojects that deliver a wide range of Honey Nodes technologies and tools.

6.4. Artifact Storage

Storage of artifacts collected through behavioral analysis follows a similar pattern as outlined in the various static analysis sections. Behavioral analysis artifacts are stored in one or more tables linked to the sample through a common key (MD5). The primary deviation from this pattern stems from the understanding

[34] http://www.e-fense.com/products.php
[35] Honey Nodes emulate a wire range of network services for analysis purposes
[36] http://www.honeynet.org/project

that behavioral analysis results can vary over time. Examples of this variance include selection of dropper site URL, phone home URL, or polymorphism algorithm that varies based on date & time. Understanding that these elements can vary between analysis sessions drives the need for also capturing the specific analysis session by which these artifacts were collected. To track this data point, all artifacts gathered during a behavioral analysis session will be tagged with a unique analysis_id. Tables 6.4.1 – 6.4.4 present a reprehensive schema for storing major behavioral analysis artifacts.

BEHVIORAL ANALYSIS: SESSION TABLE		
md5	*CHAR(XX), UNIQUE*	MD5 hash of sample
analysis_id	*INT*	Analysis ID
TimeDateStamp	*CHAR(XX)*	Date & Time of Analysis Session
comment	*CHAR(XX)*	Analyst Comment

Table 6.4.1: Sample Analysis Sessions

BEHAVIORAL ANALYSIS: FILE ACTIVITY TABLE		
md5	*CHAR(XX), UNIQUE*	MD5 hash of sample
analysis_id	*INT*	Analysis ID
file_path	*CHAR(XX)*	Path to file object
file_operation	*TINY*	1=ADD, 2=DEL, 3=Mod

Table 6.4.2: Sample Filesystem Events

BEHAVIORAL ANALYSIS: NETWORK ACTIVITY TABLE		
md5	*CHAR(XX), UNIQUE*	MD5 hash of sample
analysis_id	*INT*	Analysis ID
URL	*CHAR(XX)*	Full URL
hostname	*CHAR(XX)*	Hostname
ip_address	*CHAR(XX)*	IP Address
port	*CHAR(XX)*	Number of Sections
direction	*CHAR(1)*	Send or Receive
Bytes	*CHAR(XX)*	Creation Date & Time
TimeDateStamp	*CHAR(XX)*	Date & Time of Resolution
comment	*CHAR(XX)*	Analyst Comment

Table 6.4.3: Sample Network Activity

BEHAVIORAL ANALYSIS: REGISTRY ACTIVITY TABLE		
md5	*CHAR(XX), UNIQUE*	MD5 hash of sample
analysis_id	*INT*	Analysis ID
reg_key	*CHAR(XX)*	Registry Key
key_oper	*TINY*	1=ADD, 2=DEL, 3=Mod
reg_value	*CHAR(XX)*	Registry Value
val_oper	*TINY*	1=ADD, 2=DEL, 3=Mod
comment	*CHAR(XX)*	Analyst Comment

Table 6.4.4: Sample Registry Events

The same schema design methodology outlined above can be leveraged to create DB structures to store other behavioral analysis artifacts such as services, DLLs, and drivers.

7. Conclusion

The zoo ideas and processes outlined in this paper represent only the most basic malware analysis and organization functionality. There are many opportunities for expansion of capabilities and enhancement of existing functionality that highlights the need for a modular, easily expandable architecture. The intent of this paper was not to build a complete solution but rather to define a framework of implementing a zoo solution that is customized to your environment. This framework covers all the major areas of design and should set you on the path to your own successful Malware Zoo implementation.

8. References

Anley, C., Heasman, J., Linder, F., & Richarte, G. (2007). *The Shellcoder's Handbook: Discovering and Exploiting Security Holes* (2nd Edition ed.). (C. Long, K. Kent, & K. Cofer, Eds.) Indianapolis, IN: Wiley Publishing, Inc.

Aquilina, J. M., Casey, E., & Malin, C. H. (2008). *Malware Forensics: Investigating and Analyzing Malicious Code.* (C. W. Rose, Ed.) Burlington, MA, US: Syngress Publishing, Inc.

Bayer, U. (2009 December). *Large-Scale Dynamic Malware Analysis (Doctoral Dissertation, Vienna University of Technology).* From iSecLab: http://www.iseclab.org/people/ulli/dissertation_ubayer.pdf

Carrier, B. (2005). *File System Forensic Analysis.* Upper Saddle River, NJ: Pearson Eductaion, Inc.

Eilam, E. (2005). *Reversing: Secrets of Reverse Engineering.* (R. Elliott, E. B. Calabro, & P. Hanley, Eds.) Indianapolis, IN: Wiley Publishing, Inc.

Halpin, T. (2001). *Information Modeling and Relational Databases: From Conceptual Analysis to Logical Design.* (D. D. Cerra, C. Palmer, B. Breyer, & K. DellaPenta, Eds.) San Francisco, CA: Morgan Kaufmann Publishers.

Kendall, K., & McMillan, C. (2007). Practical Malware Analysis. *Black Hat DC 2007.* Washington: Black Hat.

Kirillov, I. A., Beck, D. A., Chase, M. P., & Martin, R. A. (2009 October). *The Concepts of the Malware Attribute Enumeration and Characterization (MAEC) Effort.* Retrieved 2010 March from MAEC: Malware Attribute Enumeration and Characterization: http://maec.mitre.org/about/docs/The_MAEC_Concept.pdf

Kornblum, J. (2006). Identifying almost identical files using context triggered piecewise hashing. *Digital Investigation. 3*, pp. 91-97. Science Direct.

Marcus, D., Greve, P., Masiello, S., & Scharoun, D. (2009). *McAfee Threats Report: Third Quarter 2009.* McAfee Inc., McAfee Labs. Santa Clara: McAfee Inc.

Nazario, J. (2004). *Defense and Detection Strategies Against Internet Worms.* Norwood, MA: Artech House, Inc.

Seitz, J. (2009). *Gray Hat Python: Python Programming for Hackers and Reverse Engineers.* (M. Dunchak, & T. Ortman, Eds.) San Francisco, CA: No Starch Press.

Joel Yonts, jyonts@gmail.com

Skoudis, E., & Zeltser, L. (2004). *Malware: Fighting Malicious Code.* (M. Franz, Ed.) Upper Saddle River, NJ: Prentice Hall.

Szor, P. (2005). *The Art of Computer Virus Research and Defense.* (K. Gettman, J. Goldstein, G. Kanouse, K. Hart, & C. Andry, Eds.) Upper Saddle River, NJ: Pearson Education, Inc.

Wildlist Organization International. (2010 January). *WildList: January, 2010.* Retrieved 2010 March from The WildList Organization International: http://www.wildlist.org/WildList/201001.htm

Zeltser, L. (2010). Reverse-engineering malware: malware analysis tools and techniques. *Proceedings of the SANS conference*