Automated Malware Similarity Analysis

Daniel Raygoza
daniel@raygoza.net

Abstract

Malware analysis has evolved in both the sophistication of the samples analyzed and the tools of the trade. Unfortunately human analysis of samples is still very expensive and time consuming. As teams of malware analysts have grown larger, the duplication of effort in analyzing similar pieces of code has also grown.

The goal of this paper is to outline a simple framework that could be used to help remedy this situation, ultimately saving time and money for organizations performing malware analysis.

The author is aware of similar proprietary and commercial products that aim to resolve this issue or similar issues, however he is not aware of any free tools that attempt to reduce redundant analysis.

Introduction

The initial approach for this project is very simple, and is designed to be as unobtrusive to an analyst's typical workflow as possible. All samples arrive into the system via a designated drop box, be it a file share or other method. The samples are disassembled and their functions are stored as individual byte-streams along with other additional information. This information is then passed to a central server that handles the indexing of this data and the similarity scoring between a given sample and every other sample in the system. Because similarity scoring occurs at the function level, it is straightforward for an analyst to quickly identify which individual functions of belonging to two samples contribute to their similarity. It should also be fairly easy to add additional algorithms in the future to support more complex similarity scoring.

The goal is not to simply state that two samples are similar, but also help analysts to identify which of their components make them similar. In doing so an analyst is easily able to identify which code may need additional scrutiny without duplicating work. This system has not been designed to support the classification of malware into families or groups, nor is it designed to identify what is or is not malware, it is instead simply a tool to identify specific similarities and provide this to the analyst in a form that may help reduce hands-on analysis time.

Disassembly

IDA handles disassembly of the binary with the aid of a python script, and this script is called for every file that appears in a drop-box. Because we are using IDA for disassembly it is expected that every sample would be unpacked prior to submission unless a similarity score between packers is desired. It would also be fairly straightforward to integrate a generic unpacker into the sequence should your organization have one available. It may also be possible to forgo IDA entirely and use a different disassembler to identify the function byte-streams.

Applying Fuzzy Hashing

While fuzzy hashing is very tolerant of minor differences in two byte-streams, it is completely unaware of the inherent similarities between the structures of two executables (PE file headers, null padding, etc.) and it is not able to appropriately deal with the transposition of functions or other large chunks of data. Fuzzy hashing across an entire binary would lead to confusing results, and would not give the analyst any indication of which specific data lead to the similarity score.

For these reasons we have chosen to apply fuzzy hashing to the byte-streams corresponding to individual functions rather than the entire sample, producing a number of individual fuzzy hash values for each sample. One of the problematic attributes of fuzzy hashing is that the values are not sortable in any meaningful way. This makes comparison between every sample in the database a fairly intensive process, although this has been addressed somewhat through deduplication, discarding very small functions, and ensuring that comparisons never occur more than once between two unique values. This was also the original motivation to centralize this information so that comparisons could be pre-computed and stored permanently, rather than at the analyst's workstations where each function in a given sample would need to be compared against every function from all other samples.

Similarity Scoring

Once a sample has been disassembled into byte-streams representing each of its functions and other additional information, the data is passed to another python script to handle the database import process and similarity scoring. The similarity scores are applied at the function level and multiplied by the size of the functions compared to give additional weight to larger functions, and then aggregated together at the sample level to give a total similarity score between two samples. All of the similarity scores are pre-computed

and stored so that queries, even on large repositories, should be fairly responsive.

The score of the individual functions may in many cases be more valuable to an analyst than the overall score. Because samples and functions can be stored with attachments (such as full disassembler output, comments, reports, etc.), an analyst can then make a determination on where to focus the hands-on analysis and produce a greater return on investment for time spent. It may also be important to identify exactly which instructions are different between two similar functions, and for this a simple byte-stream difference could be calculated, or the byte-streams could be loaded into a more sophisticated tool.

Interface and Visualization

The first interface for this project will be a simple web interface. This will allow for an analyst to query for a specific sample or function and retrieve a list of similarity scores and their related attachments (disassembler output, binaries, notes, IDB files, etc.). Ultimately this similarity data may lend itself to more interesting visualizations displaying the relationships between all of the samples in the system.

Many of the more complex queries to retrieve similarity data were created as stored procedures in MySQL, and it should be simple to develop additional interfaces to this information (visualization, export, tool integration, etc) without the use of custom queries. There are also plans to expose the majority of the data via XML or JSON interfaces.

Future Work

It may be possible to make fuzzy hashing more tolerant of trivial changes in functions (such as the value of operands in instructions) that may otherwise reduce the similarity of two functions. This could be accomplished by either nulling out all of the operands in the byte-stream, or by mapping individual opcodes to groups and values, producing a byte-stream that is an interpretation of the general sequence of instructions in the function.

It may also be worth examining the possibility of integrating the output of the similarity scoring system into the users workspace (for example IDA or Immunity) to include comments and other meaningful information gathered from similar functions. This would provide the analyst with convenient access to similarity information in their preferred environment.

References

Yara
http://code.google.com/p/yara-project/

Fuzzy Hashing – Jesse Kornblum
http://dfrws.org/2006/proceedings/12-Kornblum-pres.pdf

Fuzzy Clarity – Digital Ninja
http://digitalninjitsu.com/downloads/Fuzzy_Clarity_rev1.pdf

Spamsum – Andrew Tridgell
http://digitalninjitsu.com/downloads/Fuzzy_Clarity_rev1.pdf

ssdeep – Jesse Kornblum
http://ssdeep.sourceforge.net/