236349 - Project In Ransomware

FSD – File System Defender

Final Report

By: Alexander Gurevich, Denys Svyshchov
Advisors: Assaf Rosenbaum, Prof. Eli Biham, Dr. Nir Levy

Spring 2018

Content

1. Introduction

In the last couple of years use of Ransomware scams has grown internationally.

According to wikipedia, Ransomware is a type of malicious software that threatens to publish the victim's data or perpetually blocks access to it unless a ransom is paid.

In this project we implemented standalone Anti-Ransomware Solution to protect Windows users against Ransomware attacks and make their private files safe.

In our opinion, Anti-Ransomware Solution should always go together with files backup, so even in case of successful ransomware attack it could garantie zero file loss. For this reason our solution will include not only Ransomware detection and neutralization, but also Cloud storage for private user files.

2. Design overview

Out solution consists of two interacting one with another modules (Kernel + User):

1. Kernel module, which works as a sniffer of file system current activity
2. User Module does processing and monitoring, backups files and kills detected threats

2.1. Kernel Module

We implemented kernel module as a File System Minifilter Driver.

A minifilter driver attaches to the file system stack indirectly, by registering with the filter manager for the I/O operations the minifilter driver chooses to filter. A minifilter driver can filter IRP-based I/O operations as well as fast I/O and file system filter (FSFilter) callback operations. For each of the I/O operations it chooses to filter, a minifilter driver can register a preoperation callback routine, a postoperation callback routine, or both. When handling an I/O operation, the filter manager calls the appropriate callback routine for each minifilter driver that registered for that operation. When that callback routine returns, the filter manager calls the appropriate callback routine for the next minifilter driver that registered for the operation.

These are the I/O operation we chose to filter:

a) IRP_MJ_CREATE

The operating system sends an IRP_MJ_CREATE request to open a handle to a file object or device object.

Information we collect:

- PID of the Process
- Filename
- How file “looks” like before process is going to change file (LZJ digest)
- Will this file be deleted after on IRP_MJ_CLOSE (FILE_FLAG_DELETE_ON_CLOSE)

b) IRP_MJ_CLOSE & IRP_MJ_CLEANUP

Receipt of this request indicates that the last handle of the file object that is associated with the target device object has been closed and released. All outstanding I/O requests have been completed or canceled.

Information we collect:

- PID of the Process
- Filename
- How file “looks” after process finished accessing the file (LZJ digest)

c) IRP_MJ_READ & IRP_MJ_WRITE

This request can be sent, for example, when a user-mode application has called a Microsoft Win32 function such as ReadFile/WriteFile.

Information we collect:

- PID of the Process
- Filename
- Number of Bytes accessed
- Operation Shannon entropy. Shannon entropy H is computed by the formula where P_i is the probability of character number i appearing in the stream of characters of the message. This value indicates how

random read/write data is from 0 (lowest entropy) to 8 (totally random data).

$$H = -\sum_i p_i \log_b p_i$$

d) IRP_MJ_SET_INFORMATION

The operating system sends an IRP_MJ_SET_INFORMATION request to set metadata about a file or file handle. We can understand from this IRP when process want to delete file or change file's name or move from directory that user choose to defend.

Information we collect:

- PID of the Process
- Old filename
- New filename
- Is the a delete operation

Driver module push processed operation to Atomic Queue (no lock data structure based on list with interlocked push and multi-pop).

When User Module queries for new IRPs (using Connection Port see. 2.3.), Driver Module pops processed operations from Atomic Queue and fills with them a given buffer.

2.2. User Module

For processing and monitoring of file system activity, backuping files and killing detected threats we developed a User Mode Application. We called it FSD Manager.

FSD Manager works with two threads: one for interaction with user (reading commands) and one for interaction with Driver Module and processing of incoming I/O operations.

2.2.1. Reading user commands

We create a dedicated thread for reading and processing user input.

There are several command, that help to configure and debug FSD Manager:

chdir [Pathname for directory]

User choose which folder he want to defend.

Directory is set to C:\Users\User\ by default.

Example: chdir C:\Users\User\Documents\

killmode [number of mode]

We have two modes:

0 - Mode in which FSD Manager detects malicious processes but doesn't kill them and report user about detection.

1 - Mode in which FSD Manager in case of detection immediately kill the malicious process.

kill [Process ID]

Ability for user to kill the process by his ID.

clear

Command that cleaning up databases (see 2.2.2.1)

print [Process ID] [frequency of printing]

Command that set frequency of printing process current information by process ID.

Example: print 3478 1000

Set print frequency for process 3478 to 1 print in 1000 related IRP's.

exit

Stop FSD Manager application.

2.2.2. System monitoring

Dedicated thread, running in high priority, queries Kernel Module for new I/O operation and processes them. After each processed operation, thread check if process is malicious according to current system state and process statistics.

Every time, when thread finishes processing of previous bulk of operations it initiates query of new I/O operations and recieves them in bulks.

We use dynamic size buffer for receiving I/O operations from Kernel Module. Dynamic size buffer allows us to adjust for rate of incoming operations, minimize penalty of switching to Kernel Mode and achieve best performance. When utilization of buffer is more than 50% we increase buffer size by 0.5 of its size until maximal size of 5 MB. If the rate of incoming I/O operations is low (received buffer size is 0.1 of its maximum size) thread sleeps for a second to lower CPU utilization.

After successful receive of new I/O operations thread starts processing them.

2.2.2.1. Internal data structures

We use global Hashmap for storing information about files.

Each file has statistics counters:

- Number of Bytes read
- Number of Bytes write
- Average read entropy
- Average write entropy

File data structures:

- Set of Processes that accessed to this file
- Last calculated LZJ digest

We use global Hashmap for storing information about processes.

Each process has statistics counters:

- Average read entropy
- Average write entropy
- Number of files deleted
- Number of files moved from safe folder (defined by user)
- Number of files moved to safe folder
- Number of file extensions changed
- Number of Bytes read
- Number of Bytes write
- Number of times process changed files and exceeded similarity threshold (changed significantly)
- Number of files with low average entropy that was replaced by files with high average entropy by process

Process data structures:

- Set of files that were accessed by process
- Set of files that were accessed for read by process
- Set of files that were accessed for write by process
- Set of file extension accessed for read by process
- Set of file extension accessed for write by process

Also we have Hashmap for accessed and protected files.

2.2.2.2. Processing

For each I/O operation received from Kernel Module we find related file and process in our database or create new if does not exist.

For read/write operations we calculate average entropy and increment related counters.

For create/close operations we check if file changed significantly (using LZJ Distance).

For set_information operation we either check if file was deleted (if it was delete option) or update its name (if it was rename option). We increment related counters and update global data structures.

After we finished processing of single I/O operation we make a decision if process is malicious.

2.2.2.3. Decision if process is malicious

To decide whether the process is malicious or not we have set of triggers:

- **Entropy trigger** : triggered when ratio between average entropy of write and read exceed threshold
- **File Distance trigger** : triggered when ratio between number of times that process exceeds LZJ distance threshold to total number of accessed files is exceed threshold
- **File Extension trigger** : triggered when ratio between symmetric difference of read and write extensions and union of them is exceed threshold
- **Deletion trigger** : triggered when ratio between number of files deleted or moved from directory to total number of accessed files is exceed threshold
- **Rename trigger** : triggered when ratio between renamed files and total number of accessed files is exceed threshold
- **Access Type Trigger** : triggered when ration between number of byte written by process to number of bytes that process have been written and read is exceed threshold
- **Move In Trigger** : triggered when ratio between number of files that moved in secure folder and number of files moved from or deleted from secure folder is exceed threshold
- **Change Extension Trigger** : triggered when ration between number of files for which process changed extension to total number of files accessed is exceed threshold
- **High Entropy Replace Trigger** : triggered when ratio between number of files with high entropy moved in secure folder to total number of files moved in secure folder is exceed threshold

The decision about on whether the process is malicious or not is based on union of triggers that results in low false positive and high detection rate.

2.2.2.4. Killing threats

If the process was identified as malicious FSD Manager kills this process immediately if it in killmode 1 or report user about detection if it in killmode 0.

2.2.2.5. Backup and Restore

Note: This part is not implemented yet.

FSDManager scans and backups user private files to Cloud on initialization.

If threat was detected we recover corrupted user files or give a user an option to make this on clean system.

2.3. Interaction between modules

Driver Module is fully independent from User Module. User Module could be swapped by updated User Module “on the fly” without restarting or interrupting Kernel Module.

Interaction between modules is done using Communication Port (link).

User Module connects to Driver module and sends messages to Driver Modules, when necessary.

User Module always initiates communications and receives data as a response.

3. Problematic points

- Need to implement backup of user private files
- User module should be a service which starts automatically, Driver should be loaded automatically without unload option (security)
- Killing multi-threaded ransomware
    - Currently we only killing threads one by one, without trying to find the root thread or using other techniques\
    - Detection still works fine, but more files are encrypted.

We believe that this project has good potential to grow. It is Open Source (Github: Randomize163/FSDefender) and we will continue developing and improving it in the future.

4.Testing

During testing we installed our driver and application on virtual machine and run ransomware samples such as Wanna Cry, Cerber, Tesla Crypt, Jigsaw and also simulated normal user workflow. We adjust our program and values of triggers threshold according to behaviour of ransomwares and harmless applications such as 7zip and other.

5. How to build and deploy project

Source code: Randomize163/FSDefender
See building instructions on GitHub: Build & Deployment

6. Literature review

During the project we read the articles : “Crypto Lock (and Drop It) Stopping Ransomware Attacks on User Data”[1]

and “ShieldFS : A Self-healing, Ransomware-aware Filesystem”[2]

This articles contain information about known detection techniques and common ransomware behaviour.

Also we read the article “Lempel-Ziv Jaccard Distance, an effective alternative to ssdeep and sdhash”[3] about Lempel-Ziv Jaccard distance algorithm and it implementation [4]

7. Bibliography

[1] https://www.cise.ufl.edu/~traynor/papers/scaife-icdcs16.pdf

[2] http://shieldfs.necst.it/continella-shieldfs-2016.pdf

[3]https://www.sciencedirect.com/science/article/pii/S1742287617302566

[4] https://github.com/EdwardRaff/LZJD