**VIETNAM NATIONAL UNIVERSITY - HO CHI MINH CITY**

**UNIVERSITY OF INFORMATION TECHNOLOGY**

**FACULTY OF COMPUTER NETWORK AND COMMUNICATION**

**LÊ THANH BÌNH**

**CHÂU THIỆN HƯNG**

**THESIS REPORT**

**STUDY MALICIOUS BEHAVIOR ANALYSIS METHODS TO DETECT SECURITY RISKS ON WINDOWS**

**BACHELOR OF ENGINEERING INFORMATION SECURITY**

**Instructor**

**PhD. NGUYỄN ANH TUẤN**

**HỒ CHÍ MINH CITY, 2017**

**VIETNAM NATIONAL UNIVERSITY - HO CHI MINH CITY**

**UNIVERSITY OF INFORMATION TECHNOLOGY**

**FACULTY OF COMPUTER NETWORK AND COMMUNICATION**

**LÊ THANH BÌNH**

**CHÂU THIỆN HƯNG**

**THESIS REPORT**

**STUDY** **MALICIOUS BEHAVIOR ANALYSIS METHODS TO DETECT SECURITY RISKS ON WINDOWS**

**BACHELOR OF ENGINEERING INFORMATION SECURITY**

**Instructor**

**PhD. NGUYỄN ANH TUẤN**

**HỒ CHÍ MINH CITY, 2017**

**DANH SÁCH HỘI ĐỒNG BẢO VỆ KHÓA LUẬN**

Hội đồng chấm khóa luận tốt nghiệp, thành lập theo Quyết định số

…………………… ngày ………………….. của Hiệu trưởng Trường Đại học Công nghệ Thông tin.

1. …………………………………………… - Chủ tịch
2. …………………………………………… - Thư ký
3. …………………………………………… - Ủy viên
4. …………………………………………… - Ủy viên

**COMMENT OF INSTRUCTOR**

**COMMENT OF REVIEWER**

**ACKNOWLEDGEMENT**

With supports and helps of many individuals, this thesis has become reality. We would like to express our appreciation to all of them. First of all, we would like to thank not only teachers in faculty of computer network and communications but also teachers in University of Information Technology who have taught us very useful lectures with all of their passion, which provide necessary knowledge for us to finish this thesis. Especially, we would like to express our appreciation to professor Nguyen Anh Tuan, our instructor, who has inspired and guided us to dive deeply into information security with this thesis. Secondly, we would like to thank our friends who has helped, supported and exchanged knowledge with us since we started studying at University of Information Technology. Finally, we would like to thank our family for supporting us during the time we develop this project.

**TABLE OF CONTENTS**

[Chapter 1. INTRODUCTION 1](#_Toc484962976)

[1.1 Motivation 1](#_Toc484962977)

[1.2 Thesis’ statement 1](#_Toc484962978)

[1.3 Subject 1](#_Toc484962979)

[1.4 Scope 2](#_Toc484962980)

[1.5 The needs of Registry monitoring module 2](#_Toc484962981)

[1.6 The needs of Service monitoring module 2](#_Toc484962982)

[1.7 The needs of Distributed Log Collector Hardware 2](#_Toc484962983)

[1.8 The needs of Centralized Log Storage System 2](#_Toc484962984)

[Chapter 2. BACKGROUND AND RELATED WORKS 3](#_Toc484962985)

[2.1 Related works 3](#_Toc484962986)

[2.1.1 OSSEC 3](#_Toc484962987)

[2.1.2 Samhain 5](#_Toc484962988)

[2.2 Background 6](#_Toc484962989)

[2.2.1 Windows Registry 6](#_Toc484962990)

[2.2.2 Windows Service 18](#_Toc484962991)

[2.2.3 Graylog 27](#_Toc484962992)

[Chapter 3. PROJECT ARCHITECTURE 38](#_Toc484962993)

[3.1 APTIDS’s overall architecture 38](#_Toc484962994)

[3.2 Registry Monitor Module 39](#_Toc484962995)

[3.2.1 Registry Monitor Architecture 40](#_Toc484962996)

[3.2.2 Registry Monitor Workflow 41](#_Toc484962997)

[3.2.3 Identifying Registry Change 42](#_Toc484962998)

[3.3 Service Monitor Module 44](#_Toc484962999)

[3.3.1 Service Monitor Architecture 44](#_Toc484963000)

[3.3.2 Service Monitor Workflows 45](#_Toc484963001)

[3.3.3 NotifyServiceStatusChange and Asynchronous Procedure Calls 46](#_Toc484963002)

[3.4 Distributed Log Collector Hardware (2/6) 49](#_Toc484963003)

[3.4.1 Log Collector Hardware architecture 49](#_Toc484963004)

[3.4.2 Distributed Log Collector Hardware workflows 50](#_Toc484963005)

[3.4.3 DLCH Server 51](#_Toc484963006)

[3.5 Centralized Log Storage System 52](#_Toc484963007)

[3.5.1 CLSS Architecture 52](#_Toc484963008)

[3.5.2 CLSS Workflows 52](#_Toc484963009)

[Chapter 4. IMPLEMENTATION 54](#_Toc484963010)

[4.1 APTIDS 54](#_Toc484963011)

[4.1.1 Registry Monitor module’s source code 54](#_Toc484963012)

[4.1.2 Service Monitor Module’s source code 65](#_Toc484963013)

[4.1.3 Log Module function 68](#_Toc484963014)

**TABLE OF FIGURES**

[Figure 2.1: OSSEC Processes in a “Server-Agent” Installation 4](#_Toc484886341)

[Figure 2.2 Cell data type segments in Registry hive. 10](#_Toc484886342)

[Figure 2.3 Example of cell indexes 11](#_Toc484886343)

[Figure 2.4 Configuration Manager implements a strategy for handling noncontiguous memory buffer problems. 13](#_Toc484886344)

[Figure 2.5 Each Windows service is stored as an entry key in the SCM database Registry location 19](#_Toc484886345)

[Figure 2.6 Each service entry key stores many Registry values that specify its information 20](#_Toc484886346)

[Figure 2.7 Graylog working model. 29](#_Toc484886347)

[Figure 2.8 The correlation between Graylog server and Graylog Collector Sidecar. 30](#_Toc484886348)

[Figure 2.9 Example configuration of Graylog Collector Sidecar. 32](#_Toc484886349)

[Figure 2.10 Graylog servers and Elasticsearch servers deployed in Cluster mode. 34](#_Toc484886350)

[Figure 2.11 Example of a dashboard. 35](#_Toc484886351)

[Figure 2.12 Messages are processed in stream. 36](#_Toc484886352)

[Figure 3.1 Registry Monitor architecture diagram 40](#_Toc484886353)

[Figure 3.2 Mechanism for detecting change in registry 43](#_Toc484886354)

[Figure 3.3 Service Monitor Architecture 44](#_Toc484886355)

[Figure 3.4 NotifyServiceStatusChange and APC 47](#_Toc484886356)

[Figure 3.5 Distributed Log Collector Hardware architecture 49](#_Toc484886357)

[Figure 3.6 Centralized Log Storage System architecture 52](#_Toc484886358)

**TABLE OF TABLES**

[Table 2.1 Registry keys and their correlations. 6](#_Toc484886359)

[Table 2.2 List of Registry hives and their filenames on system disk. 7](#_Toc484886360)

[Table 2.3 Registry hive cells and their data types. 9](#_Toc484886361)

[Table 2.4 Root keys and their descriptions. 15](#_Toc484886362)

[Table 2.5 System Registry value types and their descriptions. 17](#_Toc484886363)

[Table 2.6 Service Registry values and descriptions 23](#_Toc484886364)

[Table 2.7 Graylog Collector Sidecar configuration parameters and their descriptions. 33](#_Toc484886365)

[Table 4.1 dwFilter values and their meanings 57](#_Toc484886366)

**TABLE OF ABBREVIATIONS**

|  |  |
| --- | --- |
| **Abbreviation** | **Expansion** |
| APC | Asynchronous Procedure Calls |
| APT | Advanced Persistent Threat |
| APTIDS | Advanced Persistent Threat Inspection Detection System |
| CLI | Command Line Interface |
| FIM | File Integrity Monitoring |
| GUI | Graphical User Interface |
| HIDS | Host-based Intrusion Detection System |
| IDS | Intrusion Detection System |
| LAN | Local Area Network |
| RPC | Remote Procedure Call |
| RPC | Remote Procedure Calls |
| SCM | Service Control Manager |
| SCM | Service Control Manager |
| SIEM | Security Information and Event Management |
| DHCP | Dynamic Host Configuration Protocol |
| DLCH | Distributed Log Collector Hardware |
| CLSS | Centralize Log Storage System |

**ABSTRACT**

We have developed a malicious behavior analysis solution for Windows Operation System called Advanced Persistent Threat Inspection Detection System (APTIDS), which is an open source solution combined of a System Monitoring Software, a Distributed Log Collector Hardware and a Centralized Log Storage on Cloud. Just like others well know open source host IDS, the software agent of APTIDS has abilities to monitor some common sectors of Windows OS like Registry, Service. Furthermore, we have developed an ability to allow APTIDS to send collected logs to the Collector Hardware, which is a Log Collector built on a Raspberry Pi, and from that hardware another collector will push all the collected log to the centralized log storage on cloud. APTIDS can monitor and alert on its runtime, that means if any malicious activity takes place at where APTIDS is monitoring, APTIDS will capture that activity, write log, and alert to the log storage.

# INTRODUCTION

## Motivation

Nowadays, with the rapid advance and wide spread of modern threats, computer users are facing threats from everywhere. From the most complicated malwares those can transform themselves to create many variants, to those that encrypt the whole computer and keep our information as hostage. For fighting back those advanced threats that are terrorizing the Internet, many company have developed antivirus softwares. To protect the innocent Internet civilians from the cyberwar that are taking place, antivirus softwares come from a free price for basic protection, to some hundred dollars for full protection against most modern attack vectors. Personal Antivirus software is very powerful for protecting a normal user from many security threats. But their shortcoming is that they can only protect a single user at one, and if there are more than one user who want to be protected, they have to buy more than one AV software, install them separately and there is no way to monitor and manage logs from all those softwares simultaneously.

Enterprise Antivirus System come as a full qualified protection for big enterprises, campuses or companies. They support for monitoring and protecting hundreds of users, and manage their logs of activities in some central cloud storage systems. However, the price for such a platinum protection is very expensive, and it is sophisticated for maintaining and operating and especially for protecting small companies or households.

From all those shortcomings of modern Antivirus Softwares and Security Protection Systems, we want to develop a solution for helping small companies and households to protect themselves against advance threats.

## Thesis’ statement

Successfully develop and run APTIDS for monitoring malicious behaviors of software on Windows Operating Systems. APTIDS monitor Registry and service for detecting softwares that are trying to write the path of their executable files.

Testing APTIDS by using some common malwares running in a controllable environment.

## Subject

Research on how malwares store themselves on Windows System for running on start up. In addition, research on how Graylog works, the method for collector logs and push them to SIEM for storage and analyzing.

## Scope

APTIDS can monitor activities in some factions of Windows Registry and the creation and deletion of Windows Services. Since it has been developed in a limited time, it does not have full features like others well-known antivirus softwares.

## The needs of Registry monitoring module

Malwares usually store the path lead to their executable applications in Registry (1) in case the system has to be restarted, they can run with the start up. Monitoring the Registry allows us to capture any malicious activity and know what is happening in the Registry Hive.

## The needs of Service monitoring module

Windows Service allows us to create a so call long-running executable application, which can start automatically at system boot (2). Knowing that, malicious programs write entries in the Service Control Manager which help them to run their executables when system boot up.

## The needs of Distributed Log Collector Hardware

The concept of this thesis is aimed to develop a solution for distributed monitoring malicious activities in a big scale network architecture. A distributed log collector hardware plays a role as a local centralize server for a single LAN network which receive the logs from agents those run in the LAN. Those agents, when capture any malicious activity, they send back their log to the log collector hardware. Each collector hardware stores log for a LAN network which can has up to hundreds of agents.

## The needs of Centralized Log Storage System

When a single log collector hardware can store logs for hundreds of agents, a Centralized Log Storage System can store and manage logs for hundreds of log collector hardware. Each hardware is managed by an input stream, and can be monitor using a single dashboard. A Centralized Log Storage System a low us to monitor hundreds of thousands machine in a large network.

# BACKGROUND AND RELATED WORKS

In this chapter, we study some similar projects those have been developed recently. This study does not aim to compare the advantages and disadvantages of those projects, but we would like to know how other people around the world have handled nowadays sophisticated APT threats. This approach has helped us much in developing APTIDS.

## Related works

### OSSEC

OSSEC is a host-based instruction detection system (HIDS) (3). A HIDS can work as a software that monitors events from inside the system rather than monitor the and inspect the network behaviors. Since from a viewpoint of the network, traffics that travel through network link might be encrypted and hard to be inspect. However, to OSSEC, any network traffics always is plaintext in the system viewpoint. Furthermore, OSSEC has a very sophisticated engine that can monitor system activities for recognize and alert upon any file system change, rootkit or malware infection. OSSEC also monitors log file, capture suspicious activities happening in special parts of the system and alert immediately for respond team to interrupt and prevent the attack on time.

OSSEC comes in a deployment with two main parts: a client agent part and a command and control server part. After has been deployed in the client machine, OSSEC agent does the monitor task. OSSEC agent can work on multiplatform, which means we can expand its protection to any host in our network. The agent communicates with its server at UDP protocol using port 1514. When an event is detected for which an alert to a system or security administrator needs to be sent, OSSEC can use one of several methods, including emails, SMS messages, pagers, etc (3). OSSEC agent can also takes actions for preventing the attack. For example, within an DDOS attack, OSSEC can insert rule into firewall that can be used to prevent the attack immediately. OSSEC server plays a role as a distributed log collector, it stores log received from the agents and alert upon those received logs.

A single OSSEC server can monitor many OSSEC agents. In case we want to connect many OSSCE server together, we can configure an agent inside the server. (see Figure 2-1)



Figure 2.1: OSSEC Processes in a “Server-Agent” Installation

From: http://www.ossec.net/ossec-docs/ossec-hids\_oahmet\_eng.pdf

OSSEC Open Source Security has become and high quality Opensource Host IDS software that is trusted and used in protecting many large campuses and enterprises. Although there are few drawbacks, OSSEC has been trusted to be improved and upgrade their abilities.

### Samhain

Samhain is a multi-platform, opensource HIDS for POSIX (4). When an attacker accesses to our system, he wants to modify the functions of the system to work in the wait that suitable for his purpose. Therefore, beside service and network operation, his job is to modify system files those are essential for the operation. System administrators should be alert as soon as possible upon any change in any system files.

Samhain has been developed mainly for the purpose of File Integrity Monitoring (FIM). Like most IDS, Samhain can also be centrally managed via a web console. Samhain has configurable rules that administrator can configure for the baseline of operations in FIM. When the rules are set, the tool then scan all the pre-defined file periodically, depends on configurations and then whether any change occurs, the tool capture it and sends alert immediately. Samhain supports for various reporting options such as log file, direct e-mail or custom script.

There are many HIDS present on the market too day, but what makes Samhain stand out of the crowd is that Samhain focus on FIM as a point for development. Samhain does not try to become a full feature and multi-function HIDS or SIEM. Samhain’s FIM function has many features that it becomes the most popular FIM software that is trusted and used worldwide. Samhain can even be used to support the Payment Card Industry Data Security Standard (PCI DSS) (or other compliance policies) that require the monitor of growing log files. To minimize the impact on the disk IO and get immediate notifications, Samhain can leverage the inotify function in Linux kernel system. Inotify is an Unix API provides a mechanism for monitoring filesystem events (5). Samhain also implement various stealth techniques that will be used to obfuscate the binary and configuration data. Further obfuscation could be implement in case an attacker gets on to the system without let him know FIM is running.

## Background

### Windows Registry

#### Introduction

Windows Registry is a hierarchical database that contains critical data for the operation of Windows systems (5). The data stored in Registry follows a tree format. Each node is called a “key”, and each key can store many subordinate keys (sub-keys) and each sub-key stores many data entries called “Values”. Some applications only require the presence of a key, other applications read into the keys and query the values stored in them. A key can store any value and a value can be in any type.

#### Inside the Registry

**Registry Hive**

The Registry is not a large file that is stored in disk, but it is a group of separated files called “hives”. A hive is a tree format data structure which has a key serving as the tree root, and each sub-key is a node of that tree (6). We might think that each key is completely isolated with other key, however, there are correlations between Registry keys in the system. Table 2.1 shows the Registry keys and their correlations.

|  |  |
| --- | --- |
| **Key** | **Description** |
| HKEY\_CLASSES\_ROOT | Symbolic link to HKEY\_LOCAL\_MACHINE \SOFTWARE \Classes. |
| HKEY\_CURRENT\_USER | Symbolic link to a key under HKEY\_USERS representing a user's profile hive. |
| HKEY\_LOCAL\_MACHINE | Placeholder with no corresponding physical hive. This key contains other keys that are hives. |
| HKEY\_USERS | Placeholder that contains the user-profile hives of logged-on accounts. |
| HKEY\_CURRENT\_CONFIG | Symbolic link to the key of the current hardware profile under HKEY\_LOCAL\_MACHINE \SYSTEM CurrentControlSet\ Control\IDConfigDB\Hardware Profiles. |
| HKEY\_DYN\_DATA | Placeholder for performance data lookups. This key has no corresponding physical hive. |

Table 2.1 Registry keys and their correlations.

From: <https://technet.microsoft.com/en-us/library/cc750583.aspx>

Root keys can correlate to each other, but none of them correlate to their hives. Table 2.2 shows the list of Registry hives and their filenames on system disk.

|  |  |
| --- | --- |
| **Hive Registry Path** | **Hive File Path** |
| HKEY\_LOCAL\_MACHINE \SYSTEM | \%windir%\system32\config\system |
| HKEY\_LOCAL\_MACHINE \SAM | \%windir%\system32\config\sam |
| HKEY\_LOCAL\_MACHINE \SECURITY | \%windir%\system32\config\security |
| HKEY\_LOCAL\_MACHINE \SOFTWARE | \%windir%\system32\config\software |
| HKEY\_LOCAL\_MACHINE \HARDWARE | Volatile hive |
| HKEY\_LOCAL\_MACHINE \SYSTEM \Clone | Volatile hive |
| HKEY\_USERS \UserProfile | Profile; usually under \%windir%\profiles\usere |
| HKEY\_USERS.DEFAULT | \%windir%\system32\config\default |

Table 2.2 List of Registry hives and their filenames on system disk.

From: <https://technet.microsoft.com/en-us/library/cc750583.aspx>

The system path variable “%windir%” links to the path “C:\Windows” in the Windows version other than NT, and it links to “C:\WinNT\” in Windows NT 4 and Windows 2000. Registry filename does not have extension, and they cannot be read or write by normal computer user (without administrator rights). The Registry path that says “Volatile hive” does not have any file stored on the system disk, it is just an image that will be loaded to system memory when the system boots, and it contents might be changed every time there is any change made by the system. For example, the hive HKEY\_LOCAL\_MACHINE \HARDWARD, which stores information about physical devices and the devices’ resources, is a volatile hive. The resource assignment to the device and the hardware detection occur every time the system boots, so storing this data on disk is not necessary, and this data might be changed every time there is an installation or uninstallation of hardware devices.

The heart of the Registry is the HKEY\_LOCAL\_MACHINE \SYSTEM hive. The subkey \CurrentControlSet\Control which is a node in HKEY\_LOCAL\_MACHINE \SYSTEM, contains settings that Configuration Manager uses to initialize the Registry. The Configuration Manager is a kernel subsystem which is responsible for implementing, initializing, managing and organizing the Registry (7). To initialize the Registry, the Configuration Manger locates the hives’ files, creates the root keys then link the hives together for building the Registry structure. To locate the hives’ files, Configuration Manger refers to the value HKEY\_LOCAL\_MACHINE \SYSTEM \CurrentControlSet \Control \hivelist. A special type of key call “symbolic link” makes it possible for the Configuration Manager to link hives together as well as organize the Registry. A symbolic link is a key that redirects Configuration Manager to another key. For example, HKEY\_LOCAL\_MACHINE \SAM is a symbolic link to the key at the root of SAM hive.

**Hive Structure**

Configuration Manager divides a hive into allocation units called “blocks” in much the same way that system divides a disk into clusters. A Registry block has a size of 4096 bytes (4KB). When a hive has to expand its size, it will expand the size in block-granular increments. The first block of a hive is called “base block”. In many ways similar to the PE header, the first hive first block contains a magic signature call “regf”, which stands for “Registry File”, for identifying the file as a Registry hive file. The base block also stores information about the timestamp showing the last time a write operation has been initiated on the hive, a hive format version number, a checksum, a file’s full name (e.g., SystemRoot\CONFIG\SAM), an update sequence numbers, … The hive format version number indicates the data format within the hive. Hive formats has changed from NT 3.51 to NT 4.0 since the presence of Windows NT 4.0. Whether we want to load an old NT 3.51 type hive file under a windows operation system which version is higher than Windows NT 4.0, the system will return an error.

Since Windows NT, Microsoft has organized the Registry hive in small containers called “cells”. A cell holds a key, a value, a security descriptor, a list of subkeys or a list of key values. The first field of the cell contains the data’s type. Table 2.3 describes each cell data types in a clear detail.

|  |  |
| --- | --- |
| **Cell** | **Data Type** |
| Key cell | A cell that contains a Registry key, also called a “key node”. A key cell contains a signature (known as “kn” for a key, “kl” for a symbolic link), the timestamp of the most recent update to the key, the cell index of the key's parent key cell, the cell index of the sub- key-list cell that identifies the key's subkeys, a cell index for the key's security descriptor cell, a cell index for a string key that specifies the class name of the key, and the name of the key (e.g., CurrentControlSet). |
| Value cell | A cell that contains information about a key's value. This cell includes a signature (kv), the value's type (e.g., REG\_DWORD, REG\_BINARY), and the value's name (e.g., Boot-Execute). A value cell also contains the cell index of the cell that contains the value's data. |
| Subkey-list cell | A cell composed of a list of cell indexes for key cells that are all subkeys of a common parent key. |
| Value-list cell | A cell composed of a list of cell indexes for value cells that are all values of a common parent key. |
| Security-descriptor cell | A cell that contains a security descriptor. Security-descriptor cells include a signature (ks) at the head of the cell and a reference count that records the number of key nodes that share the security descriptor. Multiple key cells can share security-descriptor cells. |

Table 2.3 Registry hive cells and their data types.

From: <https://technet.microsoft.com/en-us/library/cc750583.aspx>

Figure 2.2 shows cell data types segments that are contained in the hive.

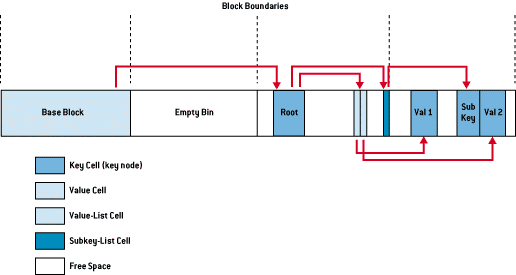


Figure 2.2 Cell data type segments in Registry hive.

From: <https://technet.microsoft.com/en-us/library/cc750583.aspx>

A cell’s header is a cell field that specifies the cell’s size. When a cell joins a hive, in case a hive needs more space to store that cell, the system will create an allocation unit call “bin”. A bin is the size of the new cell rounded up to the next block boundary. The system considers any space between the end of the cell and the end of the bin free space that it can allocate to other cells. Bins also have headers that contain a signature called “hbin” and a field that records the offset into the hive file of the bin and the bin's size.

Using bin instead of cells to track active parts of the Registry, system can reduce the management overheads. The question is why using bins makes system run smoother than cells? For answering that question, we have to identify that cell is a build in unit in Registry hive, but bin is an addon unit in case the hive wants to expand its size then cells are contained in the hive, but bins are not. When Configuration Manager allocates or deallocates bins, nothing changes in the hive. However, when Configuration Manager empties a cell in registry hive, it creates an empty bin in the hive, the hive is now fragmented and leads to the fragmentation of the registry. Fragmentation can slow down read and write processes and makes the system run slower. When a bin becomes empty, the Configuration Manager adjacent that bin and other empty bins to a large empty bin, and make it as contiguous as possible. Configuration Manager also joins empty cells to become larger empty cells.

The links that create the structure of a hive are cell indexes. A cell index is the offset into the hive file of a cell. Thus, a cell index is like a pointer from one cell to another cell that the Configuration Manager interprets relative to the start of a hive. For example, a cell that describes a key contains a field specifying the cell index of its parent key; a cell index for a subkey specifies the cell that describes the subkeys that are subordinate to the specified subkey. A subkey-list cell contains a list of cell indexes that refer to the subkey's key cells. When we want to locate a key that is a subkey belongs to a particular key, we have to locate the cell containing that key’s subkey lists using the subkey-list cell. For each subkey cell, we check the subkey’s name to fine the one we want to locate. Let’s take a look at figure 2.3 default for an example.

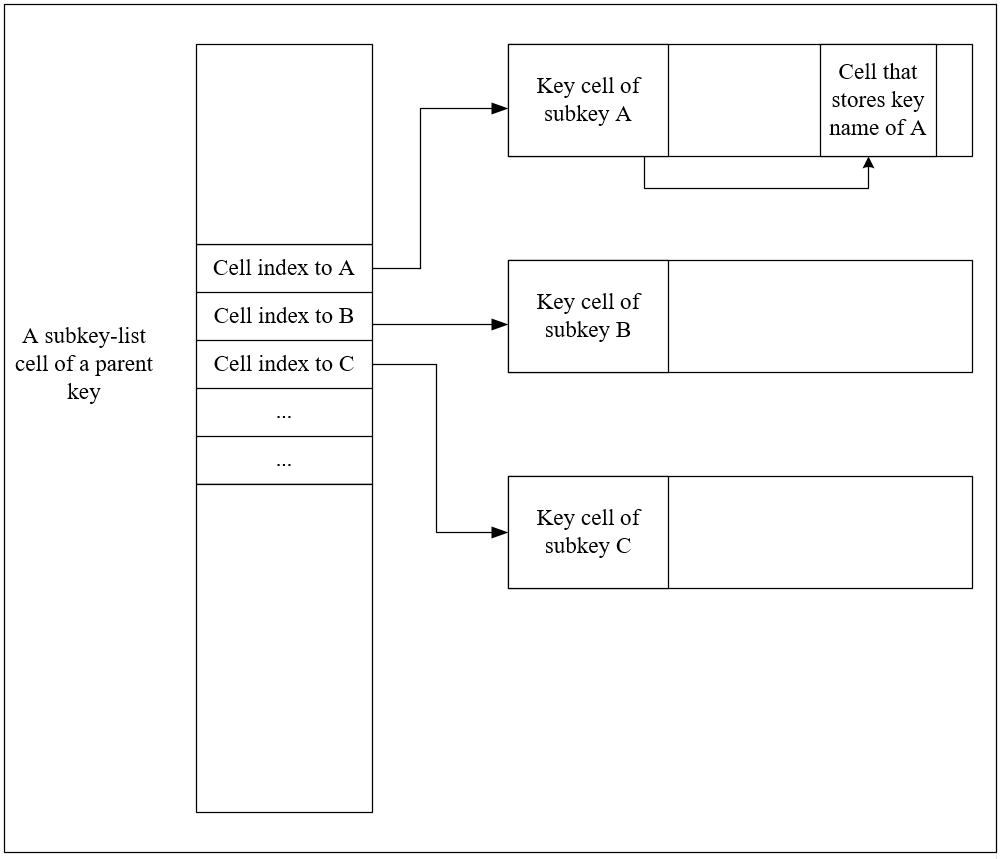


Figure 2.3 Example of cell indexes

Consider that we have a parent key and we want to allocate its subkey. First, we find the parent key’s subkey-list cell which is a cell contained in the hive of the parent key. The subkey-list cell could contain many cell-indexes those point to other *key-cell* cell of other keys, those are subkey of the current parent key. Suppose that we would like to find subkey A, we locate the cell-index that point to the key-cell cell of a subkey, then check the cell-index that point to a cell containing its name. If the name is A, we have located our subkey correctly, if it is not, we do these steps repeatedly for other cell indexes again we find our subkey.

The concept of hive, block, cell and bin might be complicated and undistinguishable. Take a look at figure 2.2 above, we see an entire long square that contains main smaller square. This long square is a registry hive. A hive contains some blocks inside it. A square that is defined by two contiguous dash lines is a block with a fix length of 4096 bytes. The first block of a hive is call a base block, which stores global information for that hive. The free space which is the block next to the base block is an empty bin. The next second blocks compose a bin, that bin consists of many cells. A free space within that bin is an empty cell (or a smaller empty bin). Bin is just a concept, a bin in general is not an object with a specific location and fix length. From a hive perspective, a bin is an allocation unit that stores cells, and those cells can be allocated or free (which can also be called an empty bin).

**Cell Maps**

Since when the system accesses the disk, the overhead for that process is much expensive compare to the outcome speed of these tasks. Configuration Manager decides not to access a hive’s image on disk every time it want to access the Registry. To finish this object, system stores a version of every hive in the kernel’s address space. When initializing a hive, Configuration Manager calculates the size of the hive and allocates enough memory from the kernel’s paged pool to store the hive file, and reads the hive file into memory. “The paged pool is a portion of the kernel's address map that NT reserves for device drivers and the kernel to use. NT can move any memory the system allocates from the paged pool to a paging file when the memory isn't in use. If hives never grew, the Configuration Manager could perform all its Registry management on the in-memory version of a hive as if the hive were a file.” (6)

With a cell index, Configuration Manager can calculate the location of a cell by adding the cell index to the base of the in-memory hive image. This is the task of “Ntldr” (Abbreviation for “NT Loader”) does with the SYSTEM hive when the system boots. When the system boots, Ntldr reads entire SYSTEM hive and load it into memory as a read-only hive, then adds the cell index to the base of the in-memory hive image to locate the cells. However, hives growth by size when they have more subkeys and values added to them, that leads to the system has to allocate paged pool memory to store the new bins. Therefore, the paged pool that keeps the Registry data is not necessary to be contiguous.

For handling the problem of noncontiguous memory buffers those store the hive data, Configuration Manager applies the strategies that Memory Manager does for mapping virtual address space to physical memory addresses (8). The Configuration Manager employs a two-level scheme that takes as input a cell index and returns both the address in memory of the block the cell index resides in and the address in memory of the bin the cell resides in.

Figure 2. describes the process that Configuration Manager handling the noncontiguous memory buffer problems.

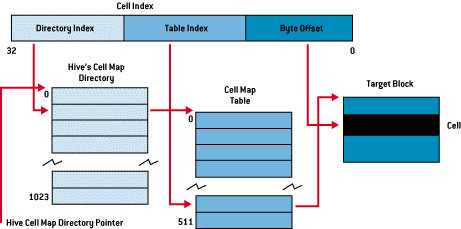


Figure 2.4 Configuration Manager implements a strategy for handling noncontiguous memory buffer problems.

From: <https://technet.microsoft.com/en-us/library/cc750583.aspx>

To implement the mapping, the Configuration Manager divides a cell index logically into fields, in the same way that the Memory Manager divides a virtual address into fields. NT interprets a cell index's first field as an index into a hive's cell map directory. The cell map directory contains 1024 entries, each of which refers to a cell map table that contains 512 map entries. The second field in the cell index specifies the entry in the cell map table that the first index field identified. That entry locates the bin and block memory addresses of the cell. In the final step of the translation process, the Configuration Manager interprets the last field of the cell as an offset into the identified block to precisely locate a cell in memory. When a hive initializes, the Configuration Manager dynamically creates the mapping tables, designating a map entry for each block in the hive, and adds and deletes tables from the cell directory as the changing size of the hive requires.

**Registry Namespace and Operation**

**Registry Root Keys**

A registry key that lying at the top level of a hive is called a root key, all root keys have a prefix of “HKEY” (9). Each root key points to a specific hive that have an essential job to the system operation. Root keys and their descriptions are shown in the table below.

|  |  |
| --- | --- |
| **Root key** | **Description** |
| HKEY\_LOCAL\_USER | Contains the root of the configuration information for the user who is currently logged on. The user's folders, screen colors, and Control Panel settings are stored here. This information is associated with the user's profile. This key is sometimes abbreviated as "HKCU." |
| HKEY\_USERS | Contains all the actively loaded user profiles on the computer. HKEY\_CURRENT\_USER is a subkey of HKEY\_USERS. HKEY\_USERS is sometimes abbreviated as "HKU." |
| HKEY\_LOCAL\_MACHINE | Contains configuration information particular to the computer (for any user). This key is sometimes abbreviated as "HKLM." |
| HEKY\_CLASSES\_ROOT | Is a subkey of HKEY\_LOCAL\_MACHINE\ Software. The information that is stored here makes sure that the correct program opens when you open a file by using Windows Explorer. This key is sometimes abbreviated as "HKCR." Starting with Windows 2000, this information is stored under both the HKEY\_LOCAL \_MACHINE and HKEY\_CURRENT\_USER keys. The HKEY\_LOCAL\_MACHINE\Software \Classes key contains default settings that can apply to all users on the local computer. The HKEY\_CURRENT\_USER\Software\Classes key contains settings that override the default settings and apply only to the interactive user. The HKEY\_CLASSES\_ROOT key provides a view of the registry that merges the information from these two sources. HKEY\_CLASSES\_ROOT also provides this merged view for programs that are designed for earlier versions of Windows. To change the settings for the interactive user, changes must be made under HKEY\_CURRENT \_USER\Software\Classes instead of under HKEY\_CLASSES\_ROOT. To change the default settings, changes must be made under HKEY \_LOCAL\_MACHINE\Software\Classes. If you write keys to a key under HKEY\_CLASSES\_ ROOT, the system stores the information under HKEY\_LOCAL\_MACHINE\Software\Classes. If you write values to a key under HKEY \_CLASSES\_ROOT, and the key already exists under HKEY\_CURRENT\_USER\Software \Classes, the system will store the information there instead of under HKEY\_LOCAL \_MACHINE\Software\Classes. |
| HKEY\_CURRENT\_CONFIG | Contains information about the hardware profile that is used by the local computer at system startup. |

Table 2.4 Root keys and their descriptions.

From: <https://support.microsoft.com/en-us/help/256986/windows-registry-information-for-advanced-users>

**Registry Data Type**

A registry hive also contains values specify the information that system can use for its operation. A hive can store as many value as it wants and a value can be anything, but its type has to obey the predefined system registry value type. The table below describes the registry types and their descriptions.

|  |  |  |
| --- | --- | --- |
| **Name** | **Data type** | **Description** |
| Binary Value | REG\_BINARY | Raw binary data. Most hardware component information is stored as binary data and is displayed in Registry Editor in hexadecimal format. |
| DWORD Value | REG\_DWORD | Data represented by a number that is 4 bytes long (a 32-bit integer). Many parameters for device drivers and services are this type and are displayed in Registry Editor in binary, hexadecimal, or decimal format. Related values are DWORD\_LITTLE\_ENDIAN (least significant byte is at the lowest address) and REG\_DWORD\_BIG\_ENDIAN (least significant byte is at the highest address). |
| Expandable String Value | REG\_EXPAND\_SZ | A variable-length data string. This data type includes variables that are resolved when a program or service uses the data. |
| Multi-String Value | REG\_MULTI\_SZ | A multiple string. Values that contain lists or multiple values in a form that people can read are generally this type. Entries are separated by spaces, commas, or other marks. |
| String Value | REG\_SZ | A fixed-length text string. |
| Binary Value | REG\_RESOURCE\_LIST | A series of nested arrays that is designed to store a resource list that is used by a hardware device driver or one of the physical devices it controls. This data is detected and written in the \ResourceMap tree by the system and is displayed in Registry Editor in hexadecimal format as a Binary Value. |
| Binary Value | REG\_RESOURCE\_  REQUIREMENTS\_LIST | A series of nested arrays that is designed to store a device driver's list of possible hardware resources the driver or one of the physical devices it controls can use. The system writes a subset of this list in the \ResourceMap tree. This data is detected by the system and is displayed in Registry Editor in hexadecimal format as a Binary Value. |
| Binary Value | REG\_FULL\_  RESOURCE\_DESCRIPTOR | A series of nested arrays that is designed to store a resource list that is used by a physical hardware device. This data is detected and written in the \HardwareDescription tree by the system and is displayed in Registry Editor in hexadecimal format as a Binary Value. |
| None | REG\_NONE | Data without any particular type. This data is written to the registry by the system or applications and is displayed in Registry Editor in hexadecimal format as a Binary Value |
| Link | REG\_LINK | A Unicode string naming a symbolic link. |
| QWORD Value | REG\_QWORD | Data represented by a number that is a 64-bit integer. This data is displayed in Registry Editor as a Binary Value and was introduced in Windows 2000. |

Table 2.5 System Registry value types and their descriptions.

From: <https://support.microsoft.com/en-us/help/256986/windows-registry-information-for-advanced-users>

### Windows Service

#### Introduction

Microsoft Windows Service (2) is a computer program which operates silently in the background of the Microsoft Windows Operation System. Window Service program offers an ability for user to create a persistent and auto-running executable applications. Windows services can be started automatically when system boots, or can be stopped, paused and change without interfering any concurrent working users on that local system.

A service application in the window service program is an entity that must conform to the rules, protocols and policy of the Service Control Manager (SCM). Besides that, Windows also support for driver service, which conforms to the device driver protocols for working with system devices.

#### Service Control Manager

**Windows Services and Service Control Manager**

Windows Services (6) are application that run on Windows computers regardless of whether a user is logged in. A Windows services is an entity that comprise an executable file, a directory for storing application components, and Registry settings that define the parameters used for that service. A Windows service can be started automatically when the system is boot, or manually by a software that control the service. Services can be controlled by any program that integrated a service control method, which is a Remote Procedure Call (RPC) (7) (8) to SCM functions.

Service Control Manager (SCM) is a Windows process for managing and controlling application services and driver services (9). SCM maintains a database of installed services and driver services, and provides a unified and secured means of controlling them. SCM database comprise information about each service and how it must be handled by the system. The information is mainly about how each service could be started when system boots, which information they need to run their executable applications and what are the security requirements for each service. SCM database is stored in a Registry location: *HKLM\SYSTEM\CurrentControlSet\Services.*

In that location, each installed service is stored as an entry key, which name corresponds to the name of the service (see figure 2.4).



Figure 2.5 Each Windows service is stored as an entry key in the SCM database Registry location

The name of an entry in this location is called a service name. However, when we work with a service, the name that display by a service management tool (such as sc.exe) is called a display name. The display name can be different to the service name, and is stored in the service entry key. For example, a service named “AxInstSV” which has its entry key stored at “HKLM\SYSTEM\CurrentControlSet \Services\AxInstSv” has a display name called “ActiveX Installer”

Opening a service entry key in Registry Editor, we can see that there are so many Registry values for that service. Those Registry values are used to specified the information set to the service (see figure 2.5).



Figure 2.6 Each service entry key stores many Registry values that specify its information

The following table describe those values and their abilities:

|  |  |  |
| --- | --- | --- |
| **Value** | **Type** | **Description** |
| DependOnGroup | REG\_MULTI\_SZ | Lists load-ordering groups on which Windows services depend. Services that depend on a group can run if, after attempting to install all members of a group, at least one member of the group is running. |
| DependOnService | REG\_MULTI\_SZ | Lists the names of Windows services on which this service depends. SCM must start these services before it starts this service. This value can be an empty string if the service has no dependencies. |
| Description | REG\_SZ | Describes the service. The description is simply a comment that explains the purpose of the service. |
| DiagnosticsMessageFile | REG\_SZ | Contains the name of the resource DLL that contains the event description strings for those events that the service writes into the application event log. Resource DLLs are located in the \Program Files\Exchsrvr\Res directory. |
| DisplayName | REG\_SZ | Contains the display name that is used to identify the service. This string has a maximum length of 256 characters. The name is case-preserved in SCM. Display name comparisons are always case-insensitive. |
| ErrorControl | REG\_DWORD | Specifies error severity and the action taken if this service fails to start. This parameter determines one of the following:   * The startup program logs the error but continues the startup operation. * The startup program logs the error and displays a message but continues the startup operation. * The startup program logs the error. If the "last known good" configuration is started, the startup operation continues. Otherwise, the system is restarted with the "last known good" configuration. * The startup program logs the error, if possible. If the "last known good" configuration is started, the system startup is cancelled. Otherwise, the system is restarted with the "last known good" configuration. |
| FailureActions | REG\_BINARY | Cites the action SCM should take for each failure of a service. A service is considered failed when it stops without reporting a status to the service controller (for example, when a service fails). |
| Group | REG\_SZ | Names the load-ordering group of which this service is a member. Note that setting this value can override the setting of the DependOnService value. |
| ImagePath | REG\_EXPAND\_SZ | Contains the fully qualified path to the service binary file. If the path contains a space, it must be quoted, so that it is correctly interpreted. For example, "d:\\Program Files\\Exchsvr\\Bin\\mad.exe".  The path can also include program arguments. |
| ObjectName | REG\_SZ | Specifies the name of the account under which the service should run. If the service uses the LocalService account, this parameter is set to NT AUTHORITY\LocalService. It is also possible to specify an account name in the form DomainName\UserName. |
| Start | REG\_DWORD | Specifies when to start the service. SCM can start a service automatically during system startup, or when a process requests the service start. This value can also specify that a service cannot be started and that attempts to start the service should result in the error code ERROR\_SERVICE\_DISABLED. |
| Tag | REG\_DWORD | Determines the service startup order within a load-ordering group. Tags are only evaluated for driver services. |
| Type | REG\_DWORD | Specifies the service type as file system driver, device driver, a service that runs its own process, or a service that shares a process with one or more other services. MSExchangeSA is an example of a service that runs its own process. EXIFS is an example of an Exchange-specific file system driver. |

Table 2.6 Service Registry values and descriptions

From:<https://technet.microsoft.com/en-us/library/881d8b23-d274-4313-a666-88f80c2cfd92.aspx>

**Service Control Manager manages Windows Services**

Enumerating services by reads each Registry key at one from the services database, SCM can create a record for each service. A service record is a set of a service name, startup type, the service status (the current state, acceptable control codes, …) and a pointer to the dependency list of that service. SCM uses these records to determine which actions are valid for the services, according to their current statuses and dependencies.

To start or stop a service, SCM communicate with the service it controls via a RPC. SCM can start services automatically at system boot, or the service can be started manually by any service control program. However, if an auto-start service demand on a demand-start service, that demand-service is also started automatically. The startup type can be set to “disable”, which tells SCM not to start the service at startup, the service also cannot be started by any mean as well. The dependencies between services are important that we should take a look at them before enabling or disabling a service. Neither an auto-start service nor a demand-start can be started if the service they depend on is disabled. Some services must not be disabled, otherwise, Windows will be failed to boot because the disabled service may be an essential service or a service that essential ones depend on. When starting a service, SCM performs the following steps:

1. **Retrieves the account information stored in the services database**

The username and password of the service account are specified at the time the service is installed. SCM stores the user name in a REG\_SZ Registry value named “ObjectName” within the Registry key of the individual service (HKEY\_LOCAL\_MACHINE\SYSTEM\CurrentControlSet\Services\<servicename>). The password is in a secure portion of Local Security Authority (LSA).

1. **Logs on the service account**

Any process that runs in Windows has to be run under an authorization of a specific account. For starting a service, SCM query the account information of that service from the services databases and logs on to Windows. The account that SCM uses to log on a local computer must have the user right called “Log on as a service”

1. **Creates the service in suspended state**

SCM starts new services in a suspended state, because the service is useful only after SCM adds the required security information to the new process.

1. **Assigns the access token to the process**

When an account logs on to Windows, the operation system calls winlogon.exe for getting the username and password of that account. When the log process successful, winlogon.exe calls wininit.exe to generate an access token, and any process which runs under that account need that access token to verify themselves (10).

1. **Allow the process to execute**

After SCM completes the logon procedure and assigns the access token, SCM can allow the service to run and perform its functions.

When a service is running, it sends status notifications to the SCM process. SCM maintains this status information in the service record for each service. SCM follow this information so that it does not mistakenly send control requests that violate the current service’s state. The service status sent includes:

* **Service Type:** A service type can be a device driver, a system driver or a Windows service, and can run its own process or share a process with other services.
* **Current State:** Indicates the state of the service as starting, running, paused or not running.
* **Acceptable control codes:** Control code that the service can accept and process in its handler function, according to the state.
* **Windows exit code:** If an error occurs when a service is starting or stopping, it uses this code to reports to the system. To return an error code specific to the service, the service must set this value to ERROR\_SERVICE\_SPECIFIC\_

ERROR to indicate that additional information can be found in the service exit code. The service sets this value to NO\_ERROR when it is running or stopping properly.

* **Service exit code:** The service uses this code to report an error when it is starting or stopping. The value is ignored unless the Windows exit code is set to ERROR\_SERVICE\_SPECIFIC\_ERROR.
* **Wait hint:** The service uses this code to report the estimated time, in milliseconds, required for a pending start, stop, pause, or continue operation.
* **Checkpoint:** The service uses this value to periodically report its progress during a lengthy start, stop, pause, or continue operation. For example, the Services tool uses this value to track the progress of the service during start and stop operations.

When stopping a service, SCM performs the following steps:

1. **SCM receives a stop request for a service**

A service control program which wants to stop a service will send a SERVICE\_CONTROL\_STOP request to the service through SCM.

1. **SCM examines the service dependencies**

If SCM finds any running service that are dependent on the service requested to be stopped, SCM will return an error code to the service control program. Before triggering the stop procedure, the service control program has to enumerate and stop all services that are dependent on the service requested.

1. **SCM forwards the stop request to the service**

If SCM detects that no dependent active services, SCM instructs the specified service to stop by forwarding the stop code to the service. The service must now free its allocated resources and shut down.

### Graylog

#### Introduction

Graylog is a Security Information and Event Management (SIEM) software. Developed in 2010 by Lennart Koopmann in his free time, nowadays Gray log has become one of the best opensource SIEM and Log Monitor that is used worldwide by enterprises and corporations around the world. Graylog’s first version was published in Mach, 2015 as an opensouce software that is supported by community and a purchased solution to support large enterprises and campuses.

#### Graylog operations

Graylog has been developed to become a centralized log distributed system. Coding by Java for working on linux operation system, Graylog inherits from Unix all the best of this famous operation system. Working as a software that has abilities of flexibility, adaptability, high availability and its supporting community, Graylog can support large enterprises and campuses those has a large number of users in a robust and trustful way. Graylog operations can be listed as functions for:

**Collecting, Preprocessing and Managing log**

Graylog is a log server that provides many methods for collecting log information from variety sources, via its collector that has ability to be deployed in many different operation system environments. Via its collectors, Graylog collects the operating information and event from its clients, then centralized manages that data, process it and manage it as well as alert on malfunction or malicious behaviors happened on its client systems.

***For Processing the log data***, Graylog provides an ability called “Pipelines”, which is an essential concept in Graylog’s operation (15). Pipelines tie together the processing steps that Graylog applies to our data. Containing rules and can connect to log streams from clients that send to its server, pipelines decides which process has to be done on which kind of message. When working with a complicated message that a separately standing rule cannot be applied appropriately, pipelines implement a concept called “stages”. Stages are considered as groups of conditions those are the rules defined for pipelines. Stages that are similar in priority run at the same time across all connected pipelines, they decide whether or not taking the next priority state.

Structure of pipeline is specified as a script that has many lines comprise of which defines the name of pipeline and those define the rules, the stages and their conditions. The structure can be explained in the following pseudo-script.

*pipeline “My pipeline”*

*stage 1 match all*

*rule “has firewall fields”;*

*rule “from firewall subnet”;*

*stage 2 match either*

*rule “geocode IPs”;*

*rule “anonymize source IPs”;*

*end*

The script describes that the pipeline name is “My pipeline”, which has 2 stages those are “stage 1” and “stage 2”. The stage’s condition for comparison is specified after the word “match”, which tells the stage to compare the characteristic of the current working log stream with the rule defined after the “rule” word. The condition “match all” defines that the stage only returns true if only all the rules specified in it return true, when compare with the current working log stream. The condition “match either” defines that the stage returns true when one of the rule comparison returns true.

Rules are the cornerstones of the processing pipeline, they contain the definitions for pipelines to know whether to change, enrich, route or drop the messages. Graylog supports a simple rule language for us to easily define the processing logic. Let’s consider a pseudo script defining rules, which are specified in the script of stages we have explained above.

*rule “has firewall fields”*

*when*

*has\_field(“src\_ip”) && has\_field(“dst\_ip”)*

*then*

*end*

And another rule is

*rule “from firewall subnet”*

*when*

*cidr\_match(“10.10.10.0/24”, to\_ip($message.g12\_remote\_ip))*

*then*

*end*

Simply written below the name of the rule is a word says “when” that is the opening of a Boolean expression for the condition underneath it. The next line is a phrase that performs the comparison for the rule, which we can consider as the heart of the rule. The first rule calls the function “has\_field” and getting a string “src\_ip”. The function will return true if there is a field named “src\_ip” inside the message. In this comparison, rule “has firewall fields” wants the message to contain both “src\_ip” and “dst\_ip” field.

The second rule wants to check whether the network address 10.10.10.0 whose subnet mask is /24 matched the IP of the message that are process. The function “to\_ip” convert the string that is the IP address which has been capture in the message. The phrase “$message” indicates the message that rule is working on, and the field “g12\_remote\_ip” is always included in the message by Graylog. The word “then” defines the actions to be taken by rules. In this case, we do not specify any action since we just explain an example about the structure of rules.

***For managing the log data***, Graylog applies a working model as demonstrated in the figure below.

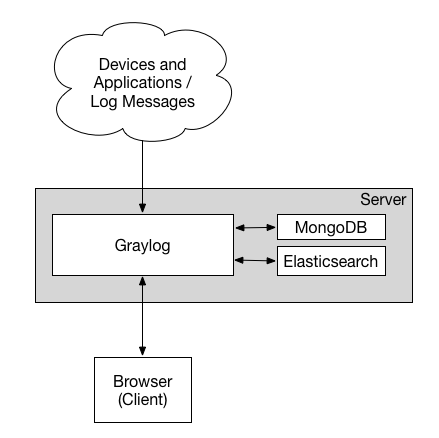


Figure 2.7 Graylog working model.

From: <http://docs.graylog.org/en/2.2/pages/architecture.html>

A Graylog server is a system that runs separately. It has a server for collecting log, a MongoDB database for storing log data and Elasticsearch Server for processing and searching for log contents. Graylog’s clients install an agent for interacting and sending log to the server, and an administrator of the system can config and setup the server via a Web GUI that was built into the Graylog server.

First, we will get into the Graylog collector that is installed on the clients for collecting log. There are two collector program that Graylog supports for collecting and sending log, these are Graylog Collector Sidecar and Graylog Collector. Graylog Collector is the older version which has been developed since the beginning of Graylog. Now Graylog Collector is deprecated and has been replaced by Graylog Collector Sidecar, which is smaller, runs smoother and more stable. In this thesis concept, we use Graylog Collector Sidecar for collecting log from distributed log collector hardware. We would like not to present Graylog Collector since it is not supported anymore, otherwise, we will analyze a little about Graylog Collector Sidecar for knowing its operation.

***Graylog Collector Sidecar*** is a lightweight configuration management system for different log collectors, also called Backend (16). While the Graylog server acts as a centralized log collector system, the Graylog Collector Sidecar runs on the client machine for collecting log. Graylog Collector Sidecar can be configured to run as an OS service (on Windows) or a daemon (on Linux). In a graphical way, the figure below draws a correlation between Graylog server and Graylog Collector Sidecar.

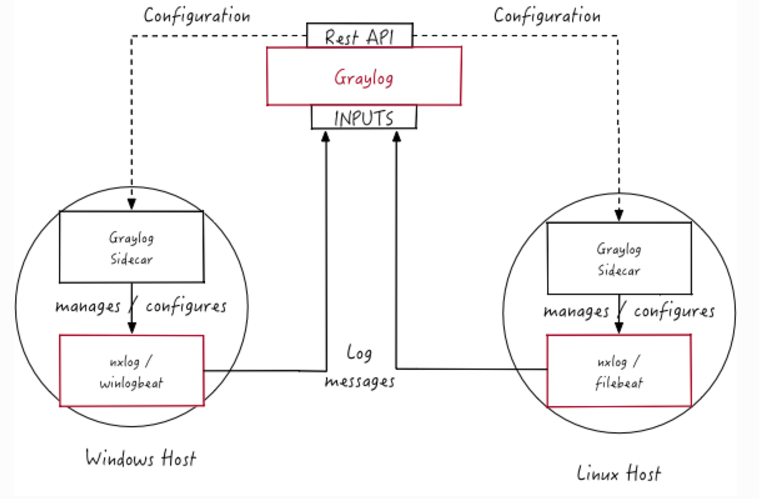


Figure 2.8 The correlation between Graylog server and Graylog Collector Sidecar.

From: <http://docs.graylog.org/en/2.2/pages/collector_sidecar.html>

Graylog Collector Sidecar receives the configuration from Graylog server through a “Rest API”. Graylog Rest API is a set of API that is used for a Graylog server to manage its client. A Graylog client managed by Rest API could be a collector or another Graylog server which is work in cluster mode with this current Graylog server. We will talk about Graylog cluster later in a small section. After Graylog Collector Side connect to Rest API, it and Graylog server will exchange configuration, if the configuration is correct, Graylog server will accept and connect with the agent via this API through the open port 9000, with the specific URL is: http://<Graylog\_server\_IP>:9000/api/. On the client side, Graylog Collector Sidecar is configured a method for sending log to the server, which we call a backend. The log backend can be NXLog or Winlogbeat (for Windows) and Filebeat (for Linux). On the other side, Graylog server should be configure for supporting the appropriate backend that can work with the agent. In the concept of this thesis, we use Filebeat for connecting the agent on our log hardware, since we configured it to run on Debian core Unix system, to the Graylog server. Therefore, to make this be straightforward, we would like to only explain the working concept of Filebeat comparing to NXLog, without discussing about Winlogbeat in any detail.

NXLog and Filebeat are two programs that are together working to create the opensource log management solutions. They both aim to support for opensource, multiplatform, lightweight, speed, reliability and high availability, those are features that all administrators who want to implement a log distributed system must consider. They also support many platforms presenting in any data center and information system. They both consider about the way they are deployed on the client’s machines, how message travels from the clients to any kind of log collector server that are commonly used. But there are two important differences between them that make us choose Filebeat instead of NXLog are the programing language they use to code and deploy their agent, and Filebeat is developed by Elastic.

First, about the programming language, since we want to deploy a little server that handle hundreds of threat monitor agents in the network, we would like to choose a log agent which run as smooth as possible to deploy in that server. In this case, Filebeat is written in Beats platform which uses Golang as its mainly developing language while NXLog use C for developing its agent. Golang is a flexible, lightweight and powerful when handling multiple threads, which helps us greatly in managing hundreds of agents deployed in the network. We believe, from the practice that Golang run smoother and handling threads better than C without using too much system resources, Filebeat could be deployed and it can handle multiple concurrent tasks without consuming too much resource on our system.

The second reason for choosing Filebeat is that Filebeat are developed by Elastic, which is the company develops the Elasticsearch Graylog server is using for its operations in managing and searching log data. Therefore, Filebeat is built in to Graylog Collector Sidecare and our client do not need to install many third party softwares while they want to protect their system. And for other small reason, we believe that just like Microsoft Office works perfectly in Microsoft Windows, Filebeat might works perfectly with Graylog system.

Graylog Collector Sidecar does not receive any configuration for which log files or directories for log collecting, but that configuration, and following by other configs that help, will be sent to the agent by Graylog server through Rest API. When recognizes there is any change to the log files specified, Graylog Collector Sidecar read the addon information by reading line to line, then sends that data to the Graylog server. Server process that data, then stores to its database for managing in the future. Filebeat has a configuration file that store configures for connecting to the Graylog server.

A Graylog Collector Sidecard configuration file might have the content show in the figure below.



Figure 2.9 Example configuration of Graylog Collector Sidecar.

From: <http://docs.graylog.org/en/2.2/pages/collector_sidecar.html>

The format of configuration file is so simple. A config contains of 2 parts: a parameter and its argument, they are separated by a colon (“:”). The parameters those define the configurations can be explained in the table below.

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| server\_url | URL to the Graylog API, e.g. http://127.0.0.1:9000/api/ |
| update\_interval | The interval in seconds the sidecar will fetch new configurations from the Graylog server |
| tls\_skip\_verify | Ignore errors when the REST API was started with a self-signed certificate |
| send\_status | Send the status of each backend back to Graylog and display it on the status page for the host |
| list\_log\_files | Send a directory listing to Graylog and display it on the host status page, e.g. /var/log. This can also be a list of directories |
| node\_id | Name of the Sidecar instance, will also show up in the web interface. Hostname will be used if not set. |
| collector\_id | Unique ID (UUID) of the instance. This can be a string or a path to an ID file |
| log\_path | A path to a directory where the Sidecar can store the output of each running collector backend |
| log\_rotation\_time | Rotate the stdout and stderr logs of each collector after X seconds |
| log\_max\_age | Delete rotated log files older than Y seconds |
| tags | List of configuration tags. All configurations on the server side that match the tag list will be fetched and merged by this instance |
| backends | A list of collector backends the user wants to run on the target host |

Table 2.7 Graylog Collector Sidecar configuration parameters and their descriptions.

From: <http://docs.graylog.org/en/2.2/pages/collector_sidecar.html>

Graylog server can function individually, or can be connected together by configuring them to work in cluster environments. In cluster mode, Graylog servers connect together by using the operation of Rest API, each server now becomes a node, and those nodes share information together. Each node owns its MongoDB separately, but Elasticsearch servers now cluster together, and many Graylog servers can query their log simultaneously on one Elasticsearch server, or many Elasticsearch server at the same time. The figure below explains that theory in a graphical way.

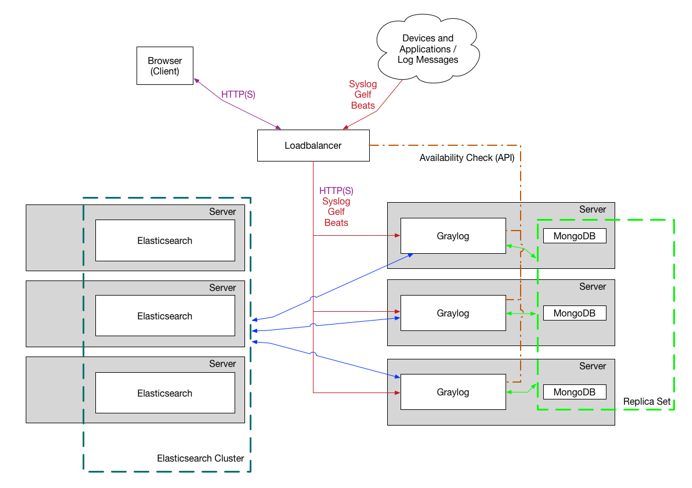


Figure 2.10 Graylog servers and Elasticsearch servers deployed in Cluster mode.

From: <http://docs.graylog.org/en/2.2/pages/architecture.html>

**Displaying log**

***Using dashboards*** allow Graylog to manage pre-defined views on our log data to always show everything important that administrators have to get the eyes on. Dashboard supports many kinds for displaying the information without any effort for finding it in a bunch of data. Any data that we want to be shown will not only be shown up as plaintext, but it also can be parsed and graphically displayed by chart, table, statistic assumption and measurement. The amount of information grows by the flying of time, and it will gets larger and larger until a point that no normal search effort in the bunch of everything can tell us where the information we need is lying. With the help of dashboard, system administrators those want to protect their network will receive information on time. They can immediately know what is happening to their system and can rapidly deduce a solution for protecting the system.

An example dashboard is shown in the figure below.

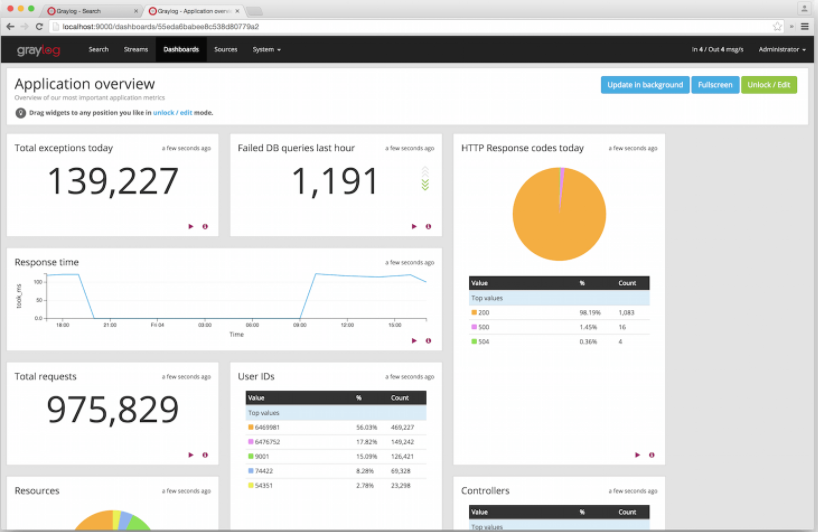


Figure 2.11 Example of a dashboard.

From: <http://docs.graylog.org/en/2.2/pages/dashboards.html>

**Streaming and Alerting**

***Graylog streams*** are a mechanism to route messages into categories in realtime while they are processed (17). When stream wants a message to be route to it, it defines a rule that any message which matches the rules will be route directly to the stream. Therefore, stream can be defined as a flow of messages those has similar content. Streams are processed in realtime by the help of pipelines. Every message that comes in is matched against the rules of a stream. For message that match any or all of the stream rules, the internal ID of that stream is stored in the streams array of that processed message. Therefore, any analysis, searches or alerts that bound to streams can now simply check the stream array base on the stream ID.

The figure below graphical demonstrates that messages are processed in stream.

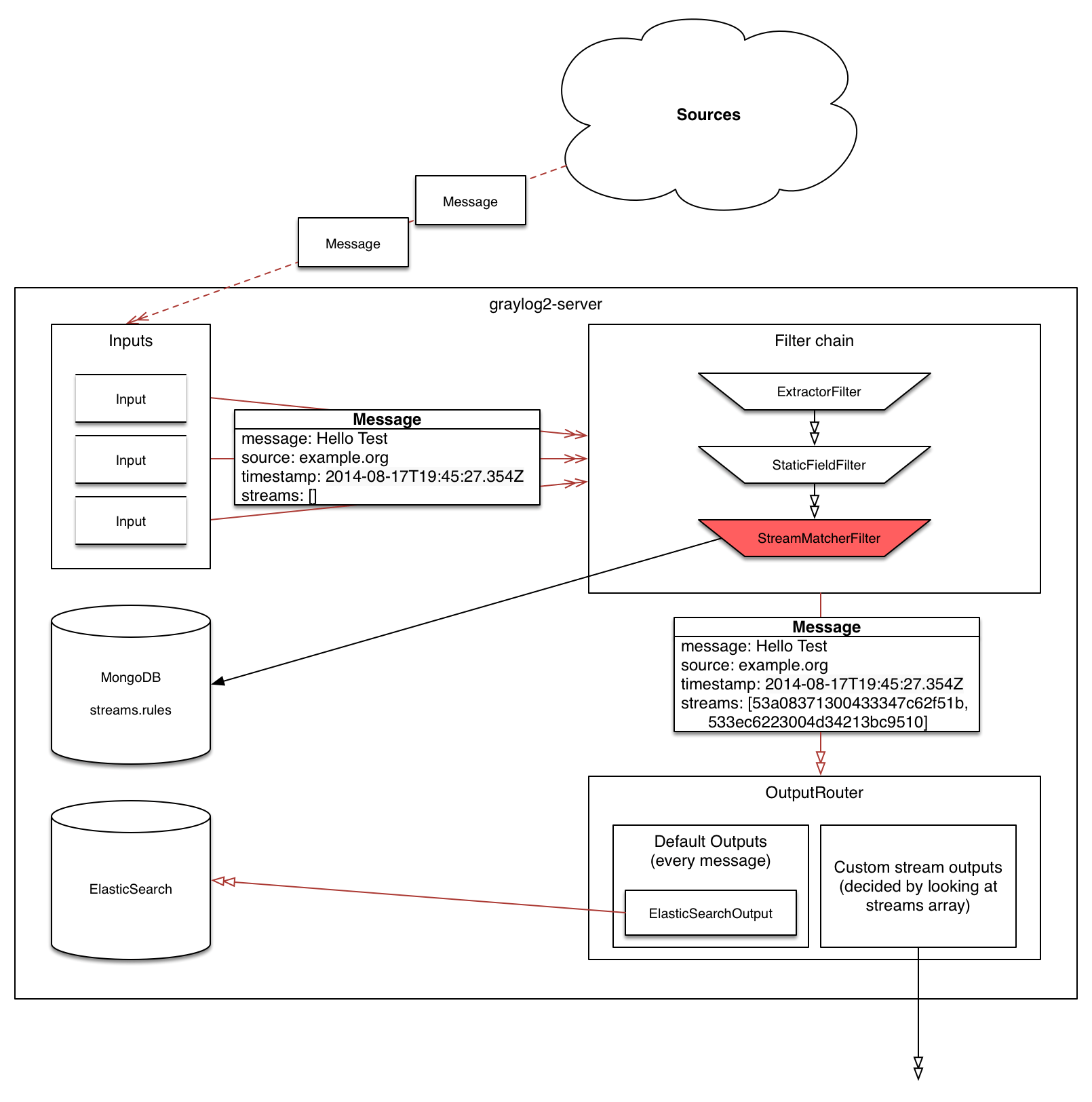


Figure 2.12 Messages are processed in stream.

From: <http://docs.graylog.org/en/2.2/pages/streams.html>

Messages sent by collector agent to the Input interface of Graylog server, are driven to a filter chain in which messages are classified base on pre-defined rules. The stream rules are store in MongoDB server of Graylog, which are treated as configurations for stream processing. The message that have been route to stream, is tagged a stream ID that help Graylog to easily find that message base on which stream it is belong to. The message, after being classified by stream, is route to output interface which is under management of Elasticsearch. Message can also be sent to other Graylog server or Elasticsearch server due to the cluster environment that Graylog are configured.

The crucial problem when processing messages is the time for classifying which message has to be routed to which stream. Applying stream rules is done during the indexing of a message only, so the time spent for classifying message might become an overhead problem if it takes too long, which could lead to slow down the overall performance of Graylog system.

There could be any scenarios when a stream rule takes too long to match, therefore a large number of messages have to be waited. The message processing can be stall, therefore, those waiting messages have to stand in the system memory, that could lead to the memory runs out and the whole system could become non-responsive. To prevent this, the runtime of stream rule matching is limited for it cannot make other messages wait for too long. When it is taking longer than the configured runtime limit, the process of matching this message against the rules of this specific stream is aborted. If the number of recorded faults for a single stream is higher than a configured threshold, the stream rule set of this stream is considered faulty and the stream is disabled.

***Graylog alerts base on stream***. Administrator can define conditions for triggering the alert based on the stream he is following. Like stream, Graylog acts to alert on a stream base on rules (18). When a stream satisfies just one or all the rules (depends on how we define our rule’s conditions), Graylog trigger the alert process and sends alert message to administrators. An alert can have two states:

* Unresolved: Alert has an unresolved state while the defined condition is satisfied. New alerts are triggered in this state, and they also execute the notifications attached to the stream. These alerts usually require action from administrator.
* Resolved: Graylog automatically resolves alerts once their alert condition is no longer satisfied. This is the final state of an alert, as Graylog will create a new alert if the alert condition is satisfied again in the future.

Graylog supports some methods for sending alerts to administrator. Alert messages can be sent to administrator via email, http protocol, alert on the alert interface of Graylog, and so on.

# PROJECT ARCHITECTURE

In this chapter, we would like to present about APTIDS, its functionalities and their operations.

## APTIDS’s overall architecture

## Registry Monitor Module

After getting into the system operations, attacker usually wants to leave a persistent running program for running at system boot. Deploying a malware for manipulating and controlling system operations, as well as doing espionage tasks, attacker also adds that system into his bot network. When being run on a system, a malware executes it operations for completing the pre-defined tasks. One of its operation is maintain a persistent executable that helps run the malware at system startup. Malware chooses Windows Registry as one of its possible hive to hide its configurations and executable program file path (20). Therefore, to detect that any malicious executable program that has written its configurations and executable file path in to Registry Hive, we have to deploy a real-time monitor application that always monitors the Registry for any pre-defined change, and alert upon those critical change. Though this approach leads to high rate of false-positive alarm, however it also concludes we will not miss any critical event happened.

There are some Registry keys that store the information for executable programs those want to start the system boot. In this case, APTIDS implements a solution for monitoring these essential keys for detecting any change happening, and triggers alert to administrators. In monitoring registry operations, we have developed APTIDS for detecting Registry keys and values that are added, modified or value data that has been modified, in the Registry hive. Since a deleted Registry key is not as harmful to system operations as an added malicious key, and since system has abilities to recover from any essential key that has been deleted which make the system unable to boot, we do not deploy a function for detecting operation that delete Registry keys.

### Registry Monitor Architecture

The Registry Monitor Architecture can be graphically present in the following figure.

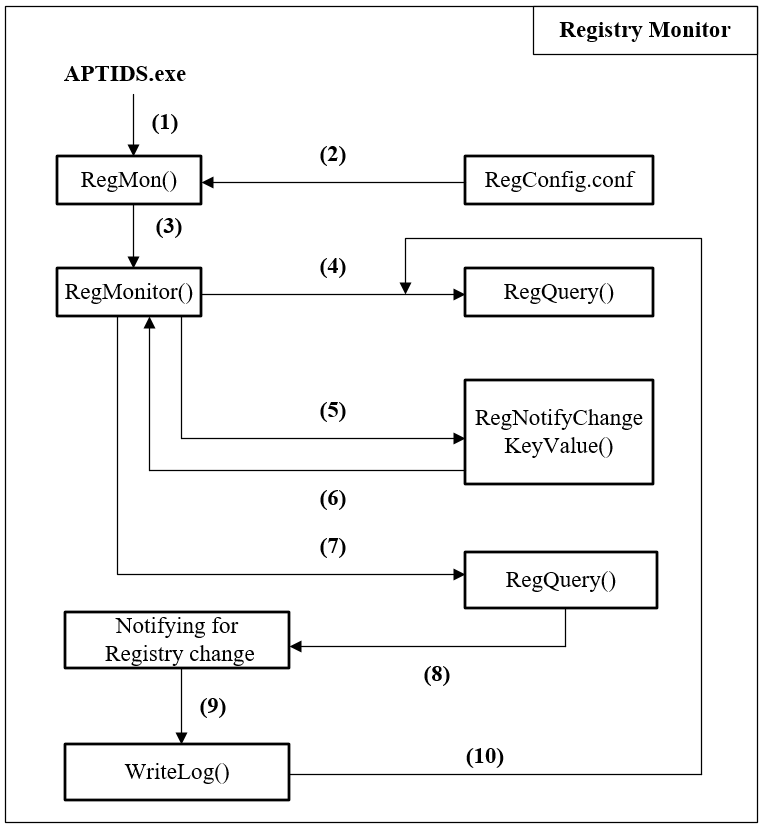


Figure 3.1 Registry Monitor architecture diagram

“APTIDS.exe” is a program that runs at background of an operating system. APTIDS then creates a thread for Registry Monitor when running. Registry Monitor run as an independent thread that real-time performs operations for detecting any critical change to the pre-defined registry keys those are defined in a configuration file called “RegConfig.conf”.

### Registry Monitor Workflow

APTIDS.exe takes “RegMon” as an argument function for starting an independent thread. RegMon then runs in background performing real-time Registry Monitor operations present in figure 3.1:

1. RegMon is run as an independent thread by APTIDS.exe
2. RegMon reads essential Registry keys in a configuration file called “RegConfig.conf”. Those keys are predefined Registry keys that an administrator want APTIDS to perform its real-time monitor on. RegConfig.conf is a plaintext file which has a XML-like format in which key is stored between an opening tag “<Key>” and a closing tag “</Key>”.
3. After receiving information about which key to monitor, RegMon passes the information to a function called “RegMonitor” for performing real-time monitor operations. RegMon creates and handles each separated thread, which take RegMonitor as thread function, for each key defined in RegConfig.conf. For example, if there are 4 registry keys defined, RegMon creates 4 independent RegMonitor thread, each for a monitor key.
4. RegMonitor thread passed the Registry Key information to “RegQuery”, which is a function that reads into the registry keys and queries the current Registry Key information and value data. At that time, RegMonitor set a special argument (which is defined to indicate whether there is any change happened) to be *false* to tell RegQuery that there was not any change happened in the Registry for this current call. Therefore, RegQuery read the specified key information and returns to RegMonitor the number of subkeys along with the number of values stored in that key.
5. RegMonitor calls a Windows API named “RegNotifyChangeKeyValue”, which API runs as a separated thread and waiting for any change event that happened in the specified Registry (detail about “RegNotifyChange KeyValue” will be discussed later).
6. RegNotifyChangeKeyValue receive an argument for a handle Windows Event. That event handle will be passed, along to the flow control of the running process, to a Windows API called “WaitForSingleObject” to wait for change happened. When change happens, WaiforSingleObject return and handle the flow control back to RegMonitor
7. When any change takes place in the specified Registry key that is indicated by RegNotifyChangeKeyValue, RegMonitor now set the special argument to *true* and calls RegQuery to get the information of that changed key.
8. RegQuery then compare the number of keys and values that returned from step 4 to the number of current keys and values. In case there is a key or value added, based on the working concept of Windows Registry in Chapter 2, that new key or value should be added to the last chain of storing keys or values. Therefore, to read the added information, we only need to read the final information of key or value in the appropriated key or value chain.
9. After identifying which change occur to the Registry, RegQuery calls “WriteLog” which is a function for writing Registry Monitor logs into system disk and then sends log to the distributed log collector hardware.
10. When finishing writing log, RegQuery return to RegMonitor to start again at step 4.

### Identifying Registry Change

For identifying added key or value in Registry, RegQuery performs the following steps:

* When calling RegQuery, RegMonitor declare a variable called “aft” which stands for “After”. At the first called, RegMonitor set “aft” to *false* indicates that this is the first call for query the number of subkeys and values. The key is now separated into the hive and the key of that hive, both are also passed to RegQuery by RegMonitor. By defined of RegMon, each RegMonitor is responsible for a key, which is passed to RegQuery.
* At the first time RegQuery reads in the Registry key for its subkeys and values, RegQuery returns two integer values: a value that identifies the number of subkeys belong to that key and a value identifies the number of values belong to that key. We called these two values as the names “nSubkeys” and “nValues”. Along with these values, RegQuery also return names of the last subkey and value in chain. Since Configuration Manager store registry keys in chain of bins and cells, query the last value in chain let us know the last thing that has been added to that key or value chain.
* When change takes place, RegMonitor calls RegQuery again, “aft” is now set to *true* indicates that there was any change happened to the monitoring keys. RegQuery now queries the specified registry key again to get the current number of subkeys and values, stores in “cSubkeys” and “cValues”.
* RegQuery checks for identifying registry change as follow:
  + If cSubkeys > nSubkeys, there was a subkey added to the specified key. Also check whether cValues > nValues. For getting the added value or subkey, just get the last subkey or value in the subkey or value chain, using APIs as RegEnumKeyEx or RegEnumValue.
  + If cSubkeys == nSubkeys, RegQuery checks whether the name of the last subkey in chain match the name return in the previous query. If they do not match, a change has happened and RegQuery query the last subkey in chain which is the subkey has been changed. RegQuery also do that for key’s values.
  + Finally, if cSubkeys < nSubkeys, a subkey was deleted. RegQuery do nothing in this case.

The following graphically demonstrates the mechanism for detecting changes in Registry.

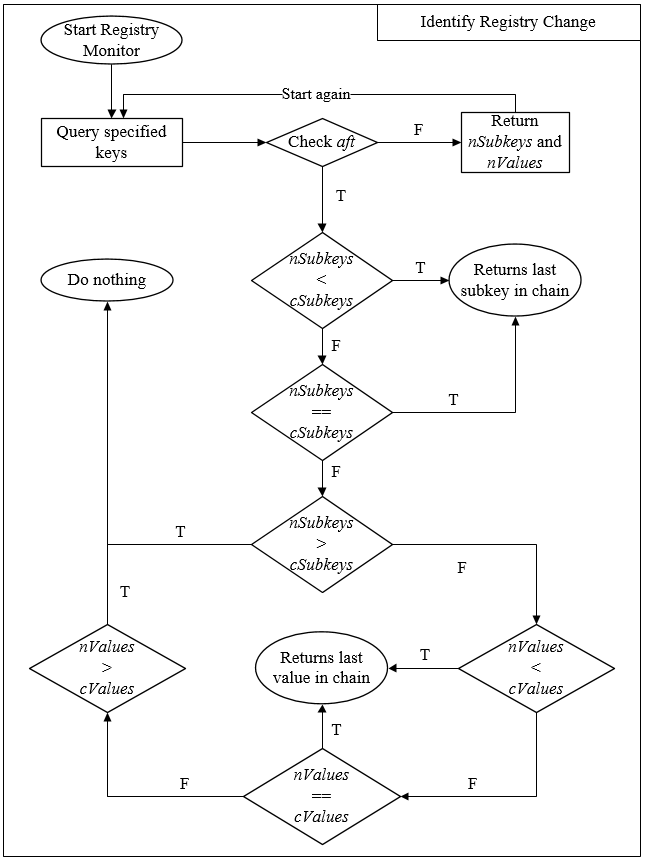


Figure 3.2 Mechanism for detecting change in registry

## Service Monitor Module

The primary purpose of malwares, whether their main tasks could be different, is to remain persistent in the compromised systems. There are lots of way to remain silently in systems without being easily detected as storing configuration in registry, creating a task in task scheduler, overwriting system files those start when system boots, etc. But what makes really excited is the Windows Service that stores service for programs those want to persistently run when system boots up, whether there is a user logged in or not. Furthermore, Windows Service also stores entries for device services, which means we can also detect rootkits in some viewpoints, those are running and store their boot-up information as device services. Since the concept of this thesis does not include rootkit detecting techniques, but when we monitor Windows Service, we can also capture and recognize some rootkits activities running in kernel mode based on device services.

### Service Monitor Architecture

The Service Monitor Architecture is graphically demonstrated in the figure below.

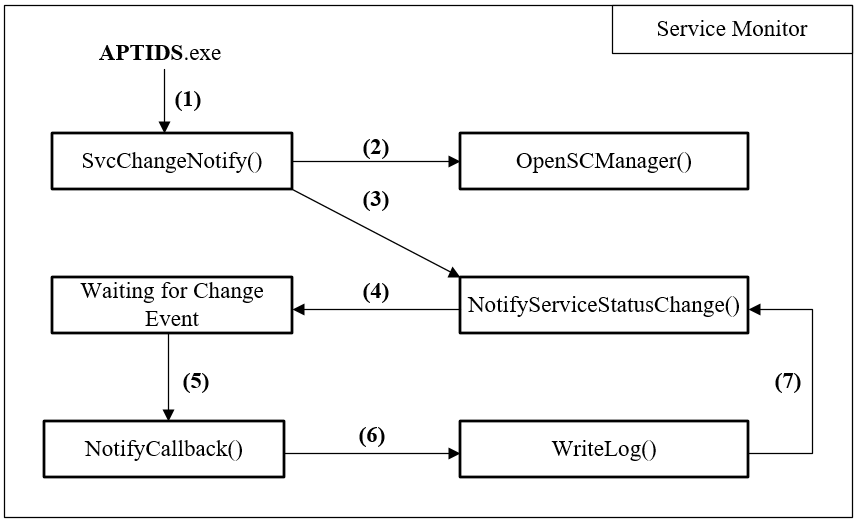


Figure 3.3 Service Monitor Architecture

APTIDS.exe run in the background of operating systems, it creates an individual thread for calling “SvcChangeNotify”, a function for monitoring and capture change in case there are both insertion or deletion of Windows Services entries.

### Service Monitor Workflows

SvcChangeNotify runs in background as an independent thread which performs real-time Service Monitor operations (as demonstrated in figure 3.3).

1. APTIDS.exe calls SvcChangeNotify as a separated thread for performing Windows Service change notification.
2. When startup, SvcChangeNotify must get a windows handle (21) for the Service Control Manager (SCM), the handle then allows SvcChangeNotify to create an access to SCM and monitor any activity occurs in the Services Database. For getting the handle, SvcChangeNotify call a Windows API named “OpenSCManager” and pass a special DWORD argument defined as “SC\_MANAGER\_ENUMERATE\_SERVICE”. OpenSCManager now returns a handle that allow any Windows API using this handle can access and enumerate all Windows Service in SCM.
3. SvcChangeNotify using this handle to calls “NotifyServiceStatusChange”, a windows API for notifying any change that SCM handle. SvcChangeNotify also specifies a Windows Event for NotifyServiceStatusChange at which whenever an event happened in SCM, that Windows Event will be set and used for capturing the occurring SCM event.
4. The Service Monitor Module now waiting for any SCM activities occurs.
5. When an SCM activity occurs, and since the concept is to capture any service that is deleted or inserted, the function “NotifyCallback” is called to check whether there is an inserted or deleted service.
6. Bases on the activities occur to the service, NotifyCallback calls “WriteLog” for writing out service change logs and sends logs to distributed log collector hardware.
7. After finish writing logs and sends log to server, WriteLog returns execution control to NotifyCallback, which then returns control to SvcChangeNotify and starts again from step 3.

### NotifyServiceStatusChange and Asynchronous Procedure Calls

NotifyServiceStatusChange is an essential API in the operations of Service Monitor Module. NotifyServiceStatusChange takes (22)

* The handle return by OpenSCManager as a handle for accessing the SCM.
* An object of the struct called “SERVICE\_NOTIFY” which stores 2 important variables:
  + Pointer to a callback function to which the thread will execute in case there is any change happened.
  + A context which can simply be explained as a user defined variable for the callback function. The context in this case stores an event handler which NotifyServiceStatusChange monitors for SCM event.
* The condition according to which NotifyServiceStatusChange performs action. This condition is a DWORD value which is defined by a XORed operation between two defined integer as “SERVICE\_NOTIFY\_CREATED” and “SERVICE\_NOTIFY\_DELETED”.

The event handler is passed to a Windows API called WaitForSingleObject to wait for any SCM event and return control to the callback function for further operations as described in section 3.3.2. NotifyServiceStatusChange not only captures the events belong to application services, but also can capture the device services which helps us in detecting rootkits’ behaviors.

When the service status changes, the system invokes the specified callback as an Asynchronous Procedure Call (APC) queued to the calling thread. An APC is a function executing asynchronously in the context of a particular thread. When an APC is queued to a thread, the system issues a software interrupt. The next time the thread is scheduled, it will run the APC function (23). Each thread has its own APC queue. An APC function called “queued to a thread” is a function that added to that thread’s queue. The queuing of an APC function is a request for the thread to call that APC function next time the thread is scheduled.

In this case, SvcChangeNotify is an independent thread that is executed by APTIDS.exe. SvcChangeNotify now calls NotifyServiceStatusChange for query the changes in SCM. When captures an SCM activities (such as service request on creating service), NotifyServiceStatusChange call the callback function (NotifyCallback) in the way that runs asynchronously from SvcChangeNotify thread. Therefore, to capture and run NotifyCallback whenever SCM event occurs, SvcChangeNotify must be set in an alert-able state. Tt passed the SCM event handler to WaitForSingleObject, this API then waits infinitely until a SCM event occurs, captures the event by alerting on the event and execute the APC function which is the NotifyCallback.

The workflows for NotifyServiceStatusChange, APC and their cooperative mates are described in the following figure.

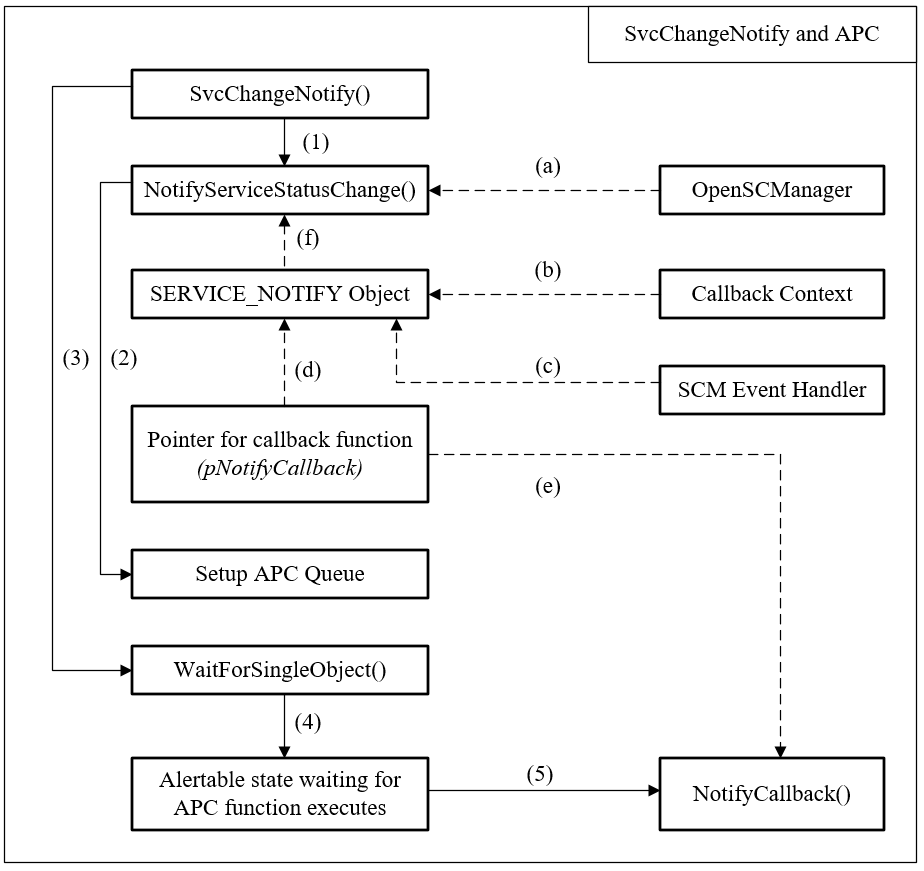


Figure 3.4 NotifyServiceStatusChange and APC

The workflows can be explained as follow:

1. SvcChangeNotify calls NotifyServiceStatusChange for monitoring SCM events. NotifyServiceStatusChange takes these arguments for running:
   1. A SCM handle that returned from OpenSCManager.
   2. A Callback Context storing the arguments for the Callback function.
   3. A SCM Event Hander for handling SCM event.
   4. A Pointer to the Callback function.
   5. Callback function that points to NotifyCallback
   6. An object of the SERVICE\_NOTIFY struct that stores pointer to (b), (c), (d), (e).
2. NotifyServiceStatusChange then create an APC queue for NotifyCallback when called.
3. SvcChangeNotify calls WaitForSingleObject to wait for SCM event happens.
4. WaitForSingleObject set the thread to alertable state which will capture and execute APC function.
5. When an SCM event occurs, NotifyCallback is executed.

## Distributed Log Collector Hardware (2/6)

The role of a Distributed Log Collector Hardware (DLCH) in the concept of this thesis is a distributed log collector server, which has been implemented in an opensource hardware device (such as Raspberry). We aim to create a log server that can store log for hundreds of devices simultaneously. Its main function is simple, just stay silently and collecting log that is sent from the agent integrated in APTIDS. When receiving the log, DLCH writes the log down to its storage system for Graylog Collector Sidecar to collect and send them to the Graylog server.

### Log Collector Hardware architecture

The architecture of Log Collector Hardware can be graphically described in the figure below.

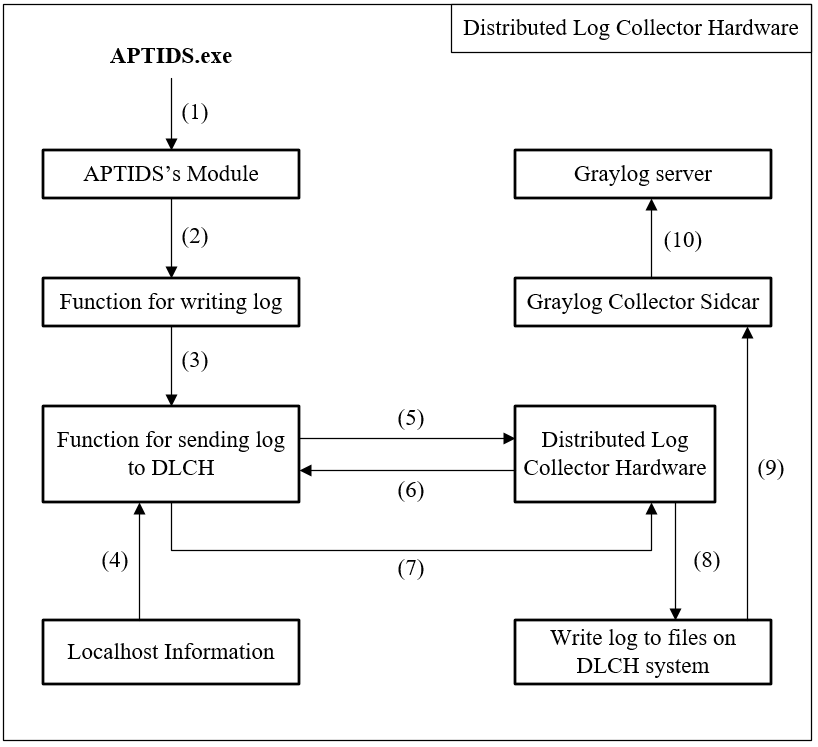


Figure 3.5 Distributed Log Collector Hardware architecture

### Distributed Log Collector Hardware workflows

Distributed Log Collector Hardware is a log collector server which is deployed in an opensource hardware for collecting log from APTIDS’s module. Since APTIDS run silently in the background, its log collector and sending module is called whenever APTIDS module detects any critical change (which is predefined in the configure files). A single DLCH can store log for up to more than 200 clients that connect simultaneously. For managing each client, DLCH store a log file that has a prefix is the hostname and the Operating System version of the specific client. Since each client host in a domain or windows workgroup must have a different name, their IP could change frequently due to the operation of DHCP, but their specified hostname rarely changes. Manage client’s log base on his hostname and windows version seems more precisely. When a hostname is change, DLCH and APTIDS generate a new log file for this client, the old log file is abandoned. According to figure 3.5, the workflows for Log Collector Hardware are explained as:

1. APTIDS.exe calls its modules for monitoring critical behaviors.
2. When a module recognizes that there is some critical event which the module was defined to monitor, it captures that event information and calls a function for handling the log.
3. The log function first writes the log to specific files for storing the main log, then it writes to a specified temperamental file that is read by another function for sending log to DLCH.
4. For sending log to the DLCH, APTIDS must know the hostname and its windows version. When starting for the first time, APTIDS gets the hostname and windows version, stores them to a file called “System.info” for maintaining that information. When sending log, log function read the localhost information, sends the log along with the local hostname and windows version.
5. To communicate with DLCH, the sending module first connects to a predefined port, which is set to 56789, for receiving further information for setting up a communicating channel. That predefined port is just for receiving communicating request from clients, it is not a channel for sending and receiving log data.
6. After a client connected to the default port, DLCH now gets a random usable port from the system and sends the port number back to APTIDS on the client host.
7. Client now connects the received port and establish a TCP channel for sending log to DLCH. APTIDS sends the client hostname, windows version and the log buffer to DLCH.
8. On receiving the log string, DLCH writes that string to a log file that has name combine from the hostname, the windows version and a “.log” extension.
9. Graylog Collector Sidecar capture the information that has been written to those log file.
10. Graylog Collector Sidecar then sends that information to Graylog server for centralized log information.

### DLCH Server

The Server deployed in DLCH is code in less than a hundred lines of code using Python scripting language. This Server main purpose is to bind a default port at 56789 to listen for request from client. Then it searches for an available port, bind this port to its local system for listening and sends this port to the client. Since the DLCH is working in the LANs which host can communicate openly to each other without being NAT by any immediate router, client can easily connect to the communicate port that DLCH Server offers and sends log to DLCH.

## Centralized Log Storage System

To collect log data from many DLCHs, we have developed a Centralized Log Storage System (CLSS). A CLSS will be a Graylog server installed on a machine. In each DLCH, we have installed Graylog Collector Sidecar which can collect any information written to a specified type of log files and send that data to the Graylog server. On receiving the log data from DLCHs, CLSS stores the log data in its database and displays the information for administrator to monitor any anomaly behavior happened.

### CLSS Architecture

The Architecture that describes functionalities of CLSS is graphically demonstrated in the figure below.

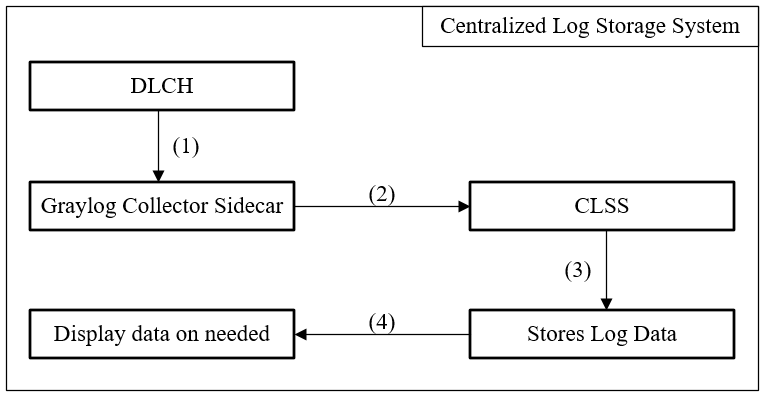


Figure 3.6 Centralized Log Storage System architecture

### CLSS Workflows

DLCH is designed for storing information about anomaly activities that were monitor by APTIDS. But one DLCH can only handle some hundred monitored machines, and its limited capacity of local resource and storage cannot help it to become a full functional log storage system. For solving these issues, we make DLCH send log to CLSS. Since CLSS is implemented as an SIEM using Graylog, it is capable of handling thousand Gigabytes of data. It’s database and search engine play an essential gold helping us much in storing, querying and displaying the information. CLSS workflows for figure 3.6 can be described as:

1. Graylog Collector Sidecar receive log information from DLCH. When DLCH writing log data into specified log files, Graylog Collector Sidecar capture this information on time.
2. Graylog Collector Sidecar sends log data to CLSS via Graylog Rest API.
3. CLSS (or Graylog server) stores that data into its database
4. Graylog Search Engine implemented in Graylog, which is built into CLSS, help administrator to query and display log information.

# IMPLEMENTATION

In the previous chapter, we have discussed about the architectures and workflows of APTIDS, DLCH and CLSS. In this chapter, we will get into source code dissection and analyze functions within each module. Furthermore, we will demonstrate how we test our implementation of APTIDS, DLCH and CLSS.

## APTIDS

### Registry Monitor module’s source code

On running at startup, APTIDS.exe called a function named “RegMon” for performing Registry Monitoring task.

**RegMon**

RegMon firstly reads the configuration file named “RegConfig.conf”, which stores the Registry keys for monitoring. RegMon checks whether the configuration file exist or whether it does not contain null, then reads its content and store in a PCHAR variable called “lpBuffer”.

DWORD WINAPI RegMon(){

LPCSTR szRegConfig = "Config/RegConfig.conf";

HANDLE hFile = CreateFileA(szRegConfig, GENERIC\_READ, 0, NULL, OPEN\_EXISTING, 0, NULL );

DWORD dwErrorCode = GetLastError();

if(dwErrorCode == ERROR\_FILE\_NOT\_FOUND){

printf("Cannot open Registry Configuration file: ERROR\_FILE\_NOT\_FOUND !\n");

return 1;

}

DWORD dwResult, dwByteRead;

PCHAR lpBuffer = (PCHAR) calloc(MAX\_BUFFER\_LEN, 1);

dwResult = ReadFile(hFile, lpBuffer, MAX\_BUFFER\_LEN, &dwByteRead, NULL);

if(dwByteRead == 0){

printf("Registry Configuration file contains NULL !\n");

return 1;

}

RegMon generate a separate thread for monitoring each registry key read in the configuration file. Therefore, we declare three arrays and a thread counter.

* “pRegKey” array is declared for storing an array of REGKEY structs, which are used to store the registry key as argument passed to the thread function.
* “hThreadArray” is used for storing an array of threads created the.
* “dwThreadIdArray” is used for storing thread ID numbers.
* “iThread” is used for counting the number of current executing threads created by RegMon.

PREGKEY pRegKey[MAX\_REG\_THREADS];

DWORD dwThreadIdArray[MAX\_REG\_THREADS];

HANDLE hThreadArray[MAX\_REG\_THREADS];

DWORD iThread = 0;

The RegConfig.conf is a configuration file using XML-like format. For reading its valuable data, we have to read between 2 tags, which is the open tag <Key> and the close tag </Key>, and skip all line with a starting “#” character. The registry key will be monitored is the information between these 2 tags. We process the line, read between the 2 tags and find the information needed.

while(true){

pRegKey[iThread] = (PREGKEY) HeapAlloc(GetProcessHeap(), HEAP\_ZERO\_MEMORY, sizeof(REGKEY));

if(pRegKey[iThread] == NULL){

ExitProcess(1);

}

if(strstr(lpBuffer, "#") != NULL){

lpBuffer = (PCHAR)strstr(lpBuffer, "\n") + strlen("\n");

}

PCHAR lpKeyStart = (PCHAR)strstr((LPCSTR)lpBuffer, "<Key>");

PCHAR lpKeyEnd = (PCHAR)strstr(lpBuffer, "</Key>");

if(lpKeyStart == NULL | lpKeyEnd == NULL){

break;

}

lpKeyStart += strlen("<Key>");

pRegKey[iThread]->stlpKey = (PCHAR) calloc(MAX\_KEY\_LEN, 1);

PCHAR lpKey = (PCHAR) calloc(MAX\_KEY\_LEN, 1);

strncpy((char \*) lpKey, lpKeyStart, (size\_t)(lpKeyEnd - lpKeyStart));

For example, a registry key read from the configuration file may be: <Key>HKLM\\Software\\Microsoft\\Windows\\CurrentControlSet\\Run</Key>. RegMon reads between two tags (<Key> and </Key>), then the is for being monitored is HKLM\\Software\\Microsoft\\Windows\\CurrentControlSet\\Run.

Now, RegMon separates the key into 2 main part, one is a main key (Root key) which is the four letters at the beginning of the key, and the second part is the subkey which is the rest of the key. From the above example, the main key (or root key) is “HKLM” and the subkey is the rest of the key, which is “Software\\Microsoft\\Windows \\CurrentControlSet\\Run”

// Allocating memory for storing the main key

pRegKey[iThread]->stlpMainKey = (PCHAR) calloc(5,1);

//Registry main key (Root key)

PCHAR lpMainKey = (PCHAR) calloc(5,1);

PCHAR lpTmp = (PCHAR)strstr( lpKey, "\\");

strncpy(lpMainKey, lpKey, (size\_t)(lpTmp - lpKey));

//Registry Subkey

lpKey = lpTmp + sizeof("\\");

RegMon creates a separated thread for each key being monitored. Each thread calls the function “RegMonitor” and passes to that function the main key and the subkey stored in a struct called REGKEY.

//Registry Monitor Threads

strcpy(pRegKey[iThread]->stlpMainKey, lpMainKey);

strcpy(pRegKey[iThread]->stlpKey, lpKey);

hThreadArray[iThread] = CreateThread(

NULL,

0,

(LPTHREAD\_START\_ROUTINE) RegMonitor,

pRegKey[iThread],

0,

&dwThreadIdArray[iThread]

);

if(hThreadArray[iThread] == NULL){

//ErrorHander(TEXT("CreateThread"));

ExitProcess(1);

}

memset(lpKey, 0, sizeof(lpKey));

memset(lpMainKey, 0, sizeof(lpMainKey));

iThread++;

Sleep(10);

}

REGKEY is a struct created for storing a registry key as 2 main part.

typedef struct RegKey{

PCHAR stlpMainKey;

PCHAR stlpKey;

}REGKEY, \*PREGKEY;

RegMon then calls a function named WaitForSMultipleObject to wait for all these threads executing.

WaitForMultipleObjects(iThread, hThreadArray, TRUE, INFINITE);

**RegMonitor**

RegMonitor is the function called by the created threads from RegMon. RegMonitor receives the main key and subkey information stored in a struct object, then parse these information into 2 strings storing the main key and subkey information.

int WINAPI RegMonitor(LPVOID lpRegKey)

{

PREGKEY pRegKey;

pRegKey = (PREGKEY)lpRegKey;

PCHAR achMainKey = (PCHAR) calloc(5,1);

PCHAR achSubKey = (PCHAR) calloc(MAX\_KEY\_LEN, 1);

snprintf(achMainKey, 5, "%s", pRegKey->stlpMainKey);

snprintf(achSubKey, MAX\_KEY\_LEN, "%s", pRegKey->stlpKey);

RegMonitor sets a value called “dwFilter” for storing value that based on it we use to monitor the registry

DWORD dwFilter = REG\_NOTIFY\_CHANGE\_NAME |

REG\_NOTIFY\_CHANGE\_ATTRIBUTES |

REG\_NOTIFY\_CHANGE\_LAST\_SET |

REG\_NOTIFY\_CHANGE\_SECURITY;

The values set to dwFilter indicate the specific aspects that a registry key must be monitor. These values and their meanings are described in the following table.

|  |  |
| --- | --- |
| **Value** | **Meaning** |
| REG\_NOTIFY\_CHANGE \_NAME  0x00000001L | Notify the caller if a subkey is added or deleted. |
| REG\_NOTIFY\_CHANGE \_ATTRIBUTES  0x00000002L | Notify the caller of changes to the attributes of the key, such as the security descriptor information. |
| REG\_NOTIFY\_CHANGE \_LAST\_SET  0x00000004L | Notify the caller of changes to a value of the key. This can include adding or deleting a value, or changing an existing value. |
| REG\_NOTIFY\_CHANGE \_SECURITY  0x00000008L | Notify the caller of changes to the security descriptor of the key. |

Table 4.1 dwFilter values and their meanings

From: <https://msdn.microsoft.com/en-us/library/windows/desktop/ms724892(v=vs.85).aspx>

RegMonitor creates 2 Registry Handle for storing 2 parts of the parsed key.

HKEY hMainKey;

HKEY hKey;

The handle variable hMainKey is used to store a handle value for the specified main key. Since Windows API does not used strings (like HKLM, HKCU,…) for working with Registry, we have to map these strings into Registry Handle values.

// Convert parameters to appropriate handles.

if (strcmp("HKLM", achMainKey) == 0) hMainKey=HKEY\_LOCAL\_MACHINE;

else if(strcmp("HKU", achMainKey) == 0) hMainKey=HKEY\_USERS;

else if(strcmp("HKCU", achMainKey) == 0) hMainKey=HKEY\_CURRENT\_USER;

else if(strcmp("HKCR", achMainKey) == 0) hMainKey=HKEY\_CLASSES\_ROOT;

else if(strcmp("HCC", achMainKey) == 0) hMainKey=HKEY\_CURRENT\_CONFIG;

else

{

printf("Usage: notify [HKLM|HKU|HKCU|HKCR|HCC] [<subkey>]\n");

return 1;

}

Depends on the main key value stored in “achMainKey”, RegMonitor maps to a registry handle value and stores it in hMainKey.

RegMonitor later reads the key using a function called “RegOpenKeyExA” for opening the key using 3 primary values:

* hMainKey: a handle point to a specified root key.
* achSubKey: a string storing the subkey. This subkey belongs to the root key above.
* hKey: a handle that stores the returned registry key handle for the opened key.

// Open and read specified registry key.

lErrorCode = RegOpenKeyExA(hMainKey,

achSubKey,

0,

KEY\_NOTIFY

|KEY\_READ

|KEY\_QUERY\_VALUE,

&hKey);

if (lErrorCode != ERROR\_SUCCESS)

{

printf("Error in RegOpenKeyEx (%d).\n", lErrorCode);

return 1;

}

For identifying changes that happened to the registry, RegMonitor calls RegQuery twice. At the first call, RegQuery snapshots the specified registry key using the “hKey” handle and return the current number of the subkey and value belonging to that monitored key. Along to those numbers, RegQuery also return names of the last subkey and the last value in Registry subkey and value chains. We will discuss about RegQuery in the following section.

//Snapshot Registry Before Change

RegQuery(

//Handle to registry key which is being monitored

hKey,

//The number of current subkey that will be returned.

&nSubKeys,

//The number of current value that will be returned

&nValues,

//Indicate whether there is any change happened

FALSE,

achMainKey,

achSubKey,

//Name of last subkey in subkey chain

lpLastSubkeyName,

//Name of last value in value chain

lpLastValueName

);

RegMonitor creates an event and calls a Windows API named “RegNotifyChange KeyValue”. RegNotifyChangeKeyValue takes 3 primary arguments:

* hKey: The current monitored registry key handle
* dwFilter: The DWORD variable store information for which activity to be monitored.
* hEvent: An Event Handle for RegNotifyChangeKeyValue to capture for registry event.

// Create an event.

hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);

if (hEvent == NULL)

{

printf("Error in CreateEvent (%d).\n", GetLastError());

return 1;

}

// Watch the registry key for a change of value.

lErrorCode

= RegNotifyChangeKeyValue(hKey,

TRUE,

dwFilter,

hEvent,

TRUE);

if (lErrorCode != ERROR\_SUCCESS)

{

printf("Error in RegNotifyChangeKeyValue (%d).\n", lErrorCode);

return 1;

}

RegMonitor stands still and waits for registry event happen by using hEvent specified by RegNotifyChangeKeyValue.

// Wait for an event to occur.

printf("Waiting for a change in the specified key...\n");

if (WaitForSingleObject(hEvent, INFINITE) == WAIT\_FAILED)

{

printf("Error in WaitForSingleObject (%d).\n", GetLastError());

return 1;

}

else printf("\nChange has occurred.\n");

When WaitForSingleObject return, if the return value is not WAIT\_FAILED (which is set to 0xFFFFFFFF), there could have been an event happened in the registry database. RegMonitor calls RegQuery for capturing another snapshot, and comparing it with the previous one to identify change which takes place.

//Snapshot Registry in case event occurs

RegQuery(hKey,

&nSubKeys,

&nValues,

// This value is now set to TRUE

// Indicating there was change happened

TRUE,

achMainKey,

achSubKey,

lpLastSubkeyName,

lpLastValueName

);

* **RegQuery**

RegQuery is a function for querying the information stored in a specified registry key.

int WINAPI RegQuery(HKEY hKey,

int \*nSubKeys,

int \*nValues,

bool aft,

const char \*achMainKey,

const char \*achSubKey,

LPWSTR lpLastSubkeyName,

LPWSTR lpLastValueName)

{

Using a Windows API called “RegQueryInfoKey” for getting information from that registry key.

retCode = RegQueryInfoKey(

hKey, // key handle

achClass, // buffer for class name

&cchClassName, // size of class string

NULL, // reserved

&cSubKeys, // number of subkeys

&cbMaxSubKey, // longest subkey size

&cchMaxClass, // longest class string

&cValues, // number of values for this key

&cchMaxValue, // longest value name

&cbMaxValueData, // longest value data

&cbSecurityDescriptor, // security descriptor

&ftLastWriteTime); // last write time

That API returns 2 valuable information:

* cSubKeys: The number of current subkeys belong to that key
* cValues: The number of current values belong to that key

RegQuery identifying changing to the specified registry key by performing 2 step:

* Step 1: Monitoring the Subkeys
* Step 2: Monitoring the Values

***Monitoring the subkeys***

If the number of subkey is larger than 0, for each subkey, RegQuery returns specific information belong to that subkey, using a Windows API called “RegEnumKeyEx”. If the current queried subkey is the last subkey in chain and the variable “aft” is set to FALSE (indicates that there wasn’t any event happened in the current query process), RegQuery copy that name for returning later.

for (i=0; i<cSubKeys; i++)

{

cbName = MAX\_KEY\_LENGTH;

retCode = RegEnumKeyEx(hKey, i,

achKey, // the current querying subkey name

&cbName, // size of subkey name

NULL,

NULL,

NULL,

&ftLastWriteTime);

if (retCode == ERROR\_SUCCESS)

{

\_tprintf(TEXT("(%d) %s\n"), i+1, achKey);

if ((cSubKeys - 1 == i) && !aft && !i) {

wcscpy(lpLastSubkeyName, achKey);

}

RegQuery defines a value called “retCode” which stores the return error code when calling RegEnumKeyEx. When the value returned is ERROR\_SUCCESS indicating that the function is executed successfully, RegQuery does further steps.

When “aft” is FALSE, RegQuery also return the number of subkeys.

if(!aft)

\*nSubKeys = cSubKeys;

and the number of value.

if(!aft)

\*nValues = cValues;

When there is a registry event happened, RegQuery is called with “aft” set to TRUE. At that time, RegQuery defines a variable called “nCount”, which is the different between:

* nSubkeys: The number of subkeys in the previous snapshot
* cSubkeys: The number of subkeys in the current snapshot

int nCount = 0;

if(aft && (\*nSubKeys < cSubKeys)){

nCount = (cSubKeys - \*nSubKeys);

}

Based on nCount, RegQuery can indicate which event has happened in the registry database.

* If nCount < 0, there was a subkey added to the registry database. Its information could have been added to the last position in the current subkey chain. (This is only right for almost key. There are some key that has its subkey position arranged by a third party process. Windows Services is an example.)
* If nCount == 0, there was a subkey modified in registry database. In this thesis we only identified for change in subkey name. When a subkey change its name, it should be move to the last position in the subkey chain. Therefore, the previous subkey in the last query is now different from the current subkey in the last position of the chain.
* If nCount > 0, there was a subkey deleted from the registry database. RegQuery do nothing in this situation.

In the situation when nCount <= 0, the last subkey in the chain is suspicious. In case nCount < 0, we log the subkey which is in the last position of the current subkey chain. When nCount = 0, if the name of subkey in the last position in the current query is different from that name in the previous query, we log that last subkey. RegQuery does nothing on other situations.

if ((cSubKeys - 1 == i) && aft && (nCount > 0)) {

getDateTime(NULL, wchTime);

\_tprintf(L"DEBUG TIME: %s\n", wchTime);

wcscat(wchLogBuffer, wchTime);

wcscat(wchLogBuffer, L"[REGISTRY] ");

wcscat(wchLogBuffer, L"[ALERT] ");

wcscat(wchLogBuffer, L"Key added: ");

wcscat(wchLogBuffer, wchLogKey);

wcscat(wchLogBuffer, L"\\\\");

wcscat(wchLogBuffer, achKey);

wcscat(wchLogBuffer, L"\n\0");

WriteLog(LOG\_TYPE\_REGISTRY, wchLogBuffer);

\_tprintf(L"%s", wchLogBuffer);

memset(wchLogBuffer, 0, sizeof(wchLogBuffer));

return 0;

}

else if ((cSubKeys - 1 == i) && aft && (nCount == 0) && (cSubKeys > 0)) {

//If the name of the subkey in the last position is different

//from the previous query

if (wcscmp(lpLastSubkeyName, achKey)) {

getDateTime(NULL, wchTime);

\_tprintf(L"DEBUG TIME: %s\n", wchTime);

wcscat(wchLogBuffer, wchTime);

wcscat(wchLogBuffer, L"[REGISTRY] ");

wcscat(wchLogBuffer, L"[ALERT] ");

wcscat(wchLogBuffer, L"Key modified: ");

wcscat(wchLogBuffer, wchLogKey);

wcscat(wchLogBuffer, L"\\\\");

wcscat(wchLogBuffer, achKey);

wcscat(wchLogBuffer, L"\n\0");

WriteLog(LOG\_TYPE\_REGISTRY, wchLogBuffer);

\_tprintf(L"%s", wchLogBuffer);

memset(wchLogBuffer, 0, sizeof(wchLogBuffer));

return 0;

}

}

The logging procedure is quite simple. RegQuery first get the time when change happened, then concatenate that time and added information about the registry subkey which is changed and called a function for writing that information out. We will discuss the logging procedure in a later section.

***Monitoring the Values***

Like the procedure for monitoring subkeys, when the current number of value is larger than 0, RegQuery read every value information of that key using a Windows API called “RegEnumValue”. Similar to the process for monitoring subkeys, “RegEnumValue” also return a “retCode” which is a value indicating the return code of the calling function.

for (i=0, retCode=ERROR\_SUCCESS; i<cValues; i++)

{

cchValue = MAX\_VALUE\_NAME;

achValue[0] = '\0';

retCode = RegEnumValue(hKey, i,

achValue, // the return value name

&cchValue, // value name's size

NULL,

NULL,

NULL,

NULL);

if (retCode == ERROR\_SUCCESS)

{ // listing value for monitor and debug

RegQuery also defines an “nCount” indicating the different between the previous number of value and the current number of value.

int nCount = 0;

if((\*nValues < cValues) && aft){

nCount = cValues - \*nValues;

}

Based on nCount, RegQuery can indicate which event has happened in the registry database.

* If nCount < 0, there was a value added to the registry database. Its information could have been added to the last position in the current value chain.
* If nCount == 0, there was a value modified in registry database. In this thesis, we also only identified for change in value name. When a value changes its name, it should be move to the last position in the value chain. Therefore, the previous value’s name in the last query is now different from the current value’s name in the last position of the chain.
* If nCount > 0, there was a value deleted from the registry database. RegQuery do nothing in this situation.

In the situation when nCount <= 0, the last value in the chain is suspicious. In case nCount < 0, we log the value which is in the last position of the current value chain. When nCount = 0, if the name of value in the last position in the current query is different from that name in the previous query, we log that last subkey. RegQuery does nothing on other situations.

if ((cValues - 1 == i) && aft && (cValues > \*nValues)) {

getDateTime(NULL, wchTime);

wcscpy(wchLogBuffer, wchTime);

wcscat(wchLogBuffer, L"[ALERT] ");

wcscat(wchLogBuffer, L"[REGISTRY] ");

wcscat(wchLogBuffer, L" New value added at Key: ");

wcscat(wchLogBuffer, wchLogKey);

wcscat(wchLogBuffer, L" -->Value Name added : ");

wcscat(wchLogBuffer, achValue);

wcscat(wchLogBuffer, L" -->Value Data: ");

wcscat(wchLogBuffer, (LPWSTR)lpValueData);

wcscat(wchLogBuffer, L"\n\0");

WriteLog(LOG\_TYPE\_REGISTRY, wchLogBuffer);

\_tprintf(L"%s", wchLogBuffer);

memset(wchLogBuffer, 0, sizeof(wchLogBuffer));

}

else if ((cValues - 1 == i) && aft && (\*nValues == cValues)) {

\_tprintf(L"Last Value: %s\n", lpLastValueName);

//If the name of the value in the last position is different

//from the previous query

if (wcscmp(lpLastValueName, achValue)) {

getDateTime(NULL, wchTime);

wcscpy(wchLogBuffer, wchTime);

wcscat(wchLogBuffer, L"[ALERT] ");

wcscat(wchLogBuffer, L"[REGISTRY] ");

wcscat(wchLogBuffer, L" New value modified at Key: ");

wcscat(wchLogBuffer, wchLogKey);

wcscat(wchLogBuffer, L" -->Value Name: ");

wcscat(wchLogBuffer, achValue);

wcscat(wchLogBuffer, L" -->Value Data: ");

wcscat(wchLogBuffer, (LPWSTR)lpValueData);

wcscat(wchLogBuffer, L"\n\0");

WriteLog(LOG\_TYPE\_REGISTRY, wchLogBuffer);

\_tprintf(L"%s", wchLogBuffer);

memset(wchLogBuffer, 0, sizeof(wchLogBuffer));

}

}

### Service Monitor Module’s source code

APTIDS implements Service Monitoring module for identifying and capturing any anomalous behavior occurs to Windows Services. APTIDS calls “SvcChangeNotify” as an independent thread to perform service monitoring tasks.

DWORD WINAPI SvcChangeNotify(){

EVT\_HANDLE subscriptionHandle;

SC\_HANDLE scManagerHandle;

HANDLE notifyEventHandle;

SERVICE\_NOTIFYW notifyBuffer;

DWORD dwResult;

HANDLE hHeap;

SECURITY\_ATTRIBUTES secAttribute;

hHeap = HeapCreate(0, sizeof(struct CALLBACK\_CONTEXT), 0);

struct CALLBACK\_CONTEXT \* pstCallbackContext

= (struct CALLBACK\_CONTEXT\*) HeapAlloc(hHeap,

0,

sizeof(struct CALLBACK\_CONTEXT));

pstCallbackContext->wchLogBuffer = (LPWSTR)calloc(MAX\_BUFFER\_LEN, 1);

SvcChangeNotify defines some variables for its functionalities:

* subscriptionHandle: An event subscription handle which is used for creating a subscription that will receive current and future events from a channel or log file that match the specified query criteria specifying in the calling function named “EvtSubcribe”.

subscriptionHandle = EvtSubscribe(

NULL,

NULL,

L"System",

L"\*[System[Provider[@Name='Service Control Manager']]]",

NULL,

NULL,

SubscribeCallback,

EvtSubscribeToFutureEvents);

EvtSubscribe is a Windows API used for creating a subscription, the channel in this context is “System” and the XPath query is “\*[System[Provider [@Name=’Service Control Manager’]]]” which indicates that this is a subscription for the Service Control Manager which is belong to the System.

* scManagerHandle: A handle for Service Control Manager.
* notifyEventHandle: An event handle for notifying change happening to SCM. This handle is used later by NotifyServiceStatusChange and WaitForSingle Object to capture SCM events.
* notifyBuffer: An object of the struct SERVICE\_NOTIFYW which stores information for using in the functionalities of NotifyServiceStatusChange. notifyBuffer stores some essential information such as
  + A number indicating the version in use.
  + A pointer to the callback function used for handling SCM events.
  + A callback context which stores arguments for passing to the callback function.

memset(&notifyBuffer, 0, sizeof(SERVICE\_NOTIFYW));

notifyBuffer.dwVersion = SERVICE\_NOTIFY\_STATUS\_CHANGE;

notifyBuffer.pfnNotifyCallback = &NotifyCallback;

notifyBuffer.pContext = (struct CALLBACK\_CONTEXT\*) pstCallbackContext;

* secAttribute: An object of SECURITY\_ATTRIBUTES struct that stores a so call security definition for creating an event.

secAttribute.nLength = sizeof(SECURITY\_ATTRIBUTES);

secAttribute.lpSecurityDescriptor = NULL;

secAttribute.bInheritHandle = FALSE;

notifyEventHandle

= CreateEventEx(&secAttribute,

NULL,

CREATE\_EVENT\_INITIAL\_SET,

EVENT\_ALL\_ACCESS);

SvcChangeNotify calls OpenSCManager to get a SCM handle with the “desire access” is SC\_MANAGER\_ENUMERATE\_SERVICE for SCM handle has a permission to enumerate Windows Services.

while (TRUE) {

scManagerHandle = OpenSCManager(NULL, NULL, SC\_MANAGER\_ENUMERATE\_SERVICE);

if (scManagerHandle == NULL) {

\_tprintf(L"Cannot open SC Manager ! \n");

return 1;

}

SvcChangeNotify then calls NotifyServiceStatusChange, a Windows API for notifying any change happened to SCM. NotifyServiceStatusChange takes:

* The SCM handle
* A DWORD value which is SERVICE\_NOTIFY\_CREATED xor SERVICE\_NOTIFY\_DELETED indicates that NotifyServiceStatusChange will monitor any service which is created or deleted.
* A pointer to notifyBuffer, which is a struct object for storing information about the event handle, the callback context and also the pointer to the callback function to be executed when events happen.

dwResult = NotifyServiceStatusChangeW(

scManagerHandle,

SERVICE\_NOTIFY\_CREATED |

SERVICE\_NOTIFY\_DELETED,

&notifyBuffer);

NotifyCallback is a function which is executed when an event happened. SvcChangeNotify calls NotifyServiceStatusChange, this function add NotifyCallback to a APC queue which is executed next time NotifyServiceStatusChange is called again. SvcChangeNotify then calls WaitForSingleObject to wait for any event happen, and the thread will be halt until any event is capture by WaitForSingleObject. When an event happens, WaitForSingleObject return control to the thread, a while loop then executes NotifyServiceStatusChange again, for its to execute any function in its APC queue, which is NotifyCallback.

while (TRUE) {

memset(&notifyBuffer, 0, sizeof(SERVICE\_NOTIFYW));

notifyBuffer.dwVersion = SERVICE\_NOTIFY\_STATUS\_CHANGE;

notifyBuffer.pfnNotifyCallback = &NotifyCallback;

notifyBuffer.pContext = (struct CALLBACK\_CONTEXT\*) pstCallbackContext;

dwResult = NotifyServiceStatusChangeW(

scManagerHandle,

SERVICE\_NOTIFY\_CREATED |

SERVICE\_NOTIFY\_DELETED,

&notifyBuffer);

if (dwResult == ERROR\_SUCCESS) {

// Wait for Notify Callback function

WaitForSingleObjectEx(pstCallbackContext->hNotifyEventHandler, INFINITE, TRUE);

}

else if (dwResult == ERROR\_SERVICE\_NOTIFY\_CLIENT\_LAGGING) {

// In case SCM lags

\_tprintf(L"Client Lagging !\n");

break;

}

else {

Sleep(5);

break;

}

} //End while

NoitifyCallback processes any operation which must be taken in case an event happened.

VOID CALLBACK NotifyCallback(

IN PVOID pParameter

)

{

LPWSTR wchLogBuffer,

wchTime;

PSERVICE\_NOTIFYW notifyBuffer = (PSERVICE\_NOTIFYW)pParameter;

When NotifyServiceStatusChange successfully captures an event, based on the event types, NotifyCallback alert a suitable log message.

if (notifyBuffer->dwNotificationStatus == ERROR\_SUCCESS) {

\_tprintf(L"System Services Modified !\n");

switch (notifyBuffer->dwNotificationTriggered) {

case SERVICE\_NOTIFY\_CREATED:

wcscat(wchLogBuffer, L"Service Created: ");

wcscat(wchLogBuffer, notifyBuffer->pszServiceNames + (CHAR)1);

wcscat(wchLogBuffer, L"\n\0");

WriteLog(LOG\_TYPE\_SERVICE, wchLogBuffer);

\_tprintf(L"%s", wchLogBuffer);

break;

case SERVICE\_NOTIFY\_DELETED:

wcscat(wchLogBuffer, L"Service Deleted: ");

wcscat(wchLogBuffer, notifyBuffer->pszServiceNames);

wcscat(wchLogBuffer, L"\n\0");

WriteLog(LOG\_TYPE\_SERVICE, wchLogBuffer);

\_tprintf(L"%s", wchLogBuffer);

break;

}

}

### Log Module function

In APTIDS, Log Module perform three operations. Firstly, “getDateTime” is called to get the current time and date at which an event occurs, using a function called “localtime”.

VOID WINAPI getDateTime(PCHAR ret\_timeBuffer = NULL, LPWSTR wRet\_timeBuffer = NULL) {

time\_t rawtime;

struct tm \* timeinfo;

PCHAR timeBuffer = (PCHAR) calloc(80, sizeof(CHAR));

time(&rawtime);

timeinfo = localtime(&rawtime);

strftime(timeBuffer, 80, "[%d-%m-%Y %I:%M:%S] ", timeinfo);

The time return by “localtime” is in UTF-8 format, for return the date time value in bot UTF-8 and UTF-16, we implement a conversion in getDateTime.

//Conver to Unicode

LPWSTR lpTimeBuffer = NULL;

INT cbByteNeeded = MultiByteToWideChar(CP\_UTF8, 0, timeBuffer, -1, lpTimeBuffer, 0);

lpTimeBuffer = (LPWSTR) calloc(cbByteNeeded, sizeof(LPWSTR));

MultiByteToWideChar(CP\_UTF8, 0, timeBuffer, -1, lpTimeBuffer, cbByteNeeded);

When the first argument is set to NULL and the second is passed in a wide char pointer (LPWSTR), getDateTime returns datetime data in UTF-16 format. Vice versa, it returns data in UTF-8. When both arguments are set, it returns data in UTF-8 and UTF-16. When both are NULL, it returns none.

The second operation in Log Module is for writing log, which is a function called “WriteLog”.

DWORD WINAPI WriteLog(DWORD dwLogType, LPWSTR wchLogBuffer) {

LPWSTR wchLogFile = NULL;

LPWSTR wchLogTmpFile = NULL;

HANDLE hFile;

WriteLog specified a DWORD variable called “dwLogType” for identifying the type of log data being written. There are 2 type of log data in APTIDS, the LOG\_TYPE\_REGISTRY and LOG\_TYPE\_SERVICE defined in DataTypes.h.

#define LOG\_TYPE\_REGISTRY 1

#define LOG\_TYPE\_SERVICE 2

When calls for writing log, APTIDS modules have to specify exactly which log type this data need to be written. Based on the log type, WriteLog generate specific name for log file and write data to that file.

switch (dwLogType) {

case LOG\_TYPE\_REGISTRY:

wchLogFile = L"Logs/Registry.log";

wchLogTmpFile = L"Logs/tmpRegistry.log";

break;

case LOG\_TYPE\_SERVICE:

wchLogFile = L"Logs/Services.log";

wchLogTmpFile = L"Logs/tmpService.Log";

break;

}

There are 2 type of log files for each log type. A main log file which is used for storing log in the local system. This file will be written continuously. Another file is a temperamental log file, which is used to store a single line of the newest log data. This file and its content is read by a function for sending log to DLCH.

// Write to end of file

DWORD dwByteNeeded = \_tcslen(wchLogBuffer);

LPOVERLAPPED lpOverLapped = (LPOVERLAPPED)calloc(1, sizeof(OVERLAPPED));

lpOverLapped->Offset = 0xffffffff;

lpOverLapped->OffsetHigh = 0xffffffff;

dwByteNeeded \*= 2;

WriteFile(hFile, (LPWSTR)wchLogBuffer, dwByteNeeded, NULL, lpOverLapped);

CloseHandle(hFile);

//OPEN AND WRITE TO TEMP LOG FILE

hFile = CreateFileW(wchLogTmpFile, GENERIC\_WRITE, FILE\_SHARE\_READ, 0, CREATE\_ALWAYS, 0, 0);

WriteFile(hFile, (LPWSTR)wchLogBuffer, dwByteNeeded, NULL, 0);

CloseHandle(hFile);

Finally, Log Module calls a function name “Networking” for sending log to DLCH.

When APTIDS is fired up for the first time in a machine, it gets the machine’s computer name and windows version, then store them in a file called “System.info”, the content looks like:

## This is System Information File

## You can Modify it and save with ASCII or UTF8 encoding

<ComputerName>P3N3TR4T10N</ComputerName>

<WindowsVersion>10.0</WindowsVersion>

Networking is called by WriteLog, it defines some essential variables:

DWORD WINAPI Networking(DWORD dwLogType) {

PCHAR pServerIP, pPort;

PCHAR pStart, pEnd;

DWORD dwPort;

DWORD dwResult, dwByteRead;

PCHAR pBuffer;

DWORD dwErrorCode;

* pServerIp: A string for storing Server’s IP Address
* pPort: A string for storing Server’s port
* dwPort: A DWORD variable for storing Server’s port, this is the number converted from pPort using “atoi” function.

Firstly, Networking reads “Networking.conf” which is a file stores information such as Server IP address and port. Networking uses this information for connecting to a server implemented in DLCH.

## Networking configuration that Agent uses to connect to Server

## You can modify it and save it as ASCII format or UTF8 format

<Server>192.168.100.21</Server>

<Port>55679</Port>

When we need to change the Server’s IP, we only have to modify the information between “<Server>” and “/Server>” tags, or modify the number between “<Port>” and “</Port>” to change server port.

Networking then calls “SendLogToServer” passing the Server IP address and Port for connecting to DLCH.

DWORD WINAPI SendLogToServer(PCHAR pServerIP, DWORD dwPort, DWORD dwLogType) {

// === GETTING LOCAL SYSTEM INFORMATION === //

DWORD dwErrorCode,

dwByteRead,

dwResult;

static PCHAR pComputerName = NULL,

pWindowsVersion;

To distinguish between log data of different computers in network, we decided to store log data in files which use computer name and version as prefix of file’s name. For example, if the computer name is “P3N3TR4T10N” and its windows version is 10.0, the log file storing service log will be named “P3N3TR4T10N\_10.0 \_service.log”. Since rarely computer names are duplicated in the same LAN or Domain, this approach helps in distinguish between machines for long term, rather than based on IP address which could easily be changed due to the operation of DHCP.

SendLogToServer reads information in “System.info” to get computer name and version. If the file does not exist or contains NULL, APTIDS stops sending that log to DLCH. SendLogToServer calls “ParseData”, a function for getting information between the predefined tag and its appropriate ending tag.

//Open System Infomatin File

LPCSTR pInfoFile = "Config/System.info";

HANDLE hInfoFile = CreateFileA(pInfoFile, GENERIC\_READ, 0, 0, OPEN\_EXISTING, 0, 0);

dwErrorCode = GetLastError();

//If file does not exist

if (dwErrorCode == ERROR\_FILE\_NOT\_FOUND) {

\_tprintf(L"Cannot open System Information file: ERROR\_FILE\_NOT\_FOUND !\n");

return 1;

}

//Read from Info File

PCHAR pInfoBuffer = (PCHAR)calloc(MAX\_BUFFER\_LEN, sizeof(CHAR));

dwResult = ReadFile(hInfoFile, pInfoBuffer, MAX\_BUFFER\_LEN, &dwByteRead, NULL);

//Close File Handle

CloseHandle(hInfoFile);

// Check weather Config File contains NULL

if (dwByteRead == 0) {

\_tprintf(L"System Information file contains NULL !\n");

return 1;

}

// Getting Computer Name

pComputerName = ParseData(pInfoBuffer, "<ComputerName>");

if (pComputerName == NULL) {

printf("Computer Name was not set\n");

return 1;

}

// Getting Windows Version

pWindowsVersion = ParseData(pInfoBuffer, "<WindowsVersion>");

if (pWindowsVersion == NULL) {

printf("Windows Version was not set\n");

return 1;

}

SendLogToServer connects to DLCH, DLCH will return another available port upon any connection. SendLogToServer terminates the previous connection and reconnects to the new received port. That connection now in used for sending log data to DLCH. According to type of log data which is specified by APTIDS’s modules.

switch (dwLogType) {

case LOG\_TYPE\_REGISTRY:

hRegFile = CreateFileW(

L"Logs/tmpRegistry.log",

GENERIC\_READ,

FILE\_SHARE\_READ,

NULL,

OPEN\_EXISTING,

0,

NULL);

dwErrorCode = GetLastError();

if (dwErrorCode == ERROR\_SUCCESS) {

ReadFile(hRegFile, lpRegBuffer, MAX\_BUFFER\_LEN, &dwRegByteRead, 0);

\_tprintf(L"%s\n", lpRegBuffer);

}

CloseHandle(hRegFile);

printf("Error Code: %d\n", dwErrorCode); // Debug

//Start sending Log

if (dwRegByteRead > 0) {

dwByteNeeded = WideCharToMultiByte(

CP\_UTF8,

0,

lpRegBuffer,

-1,

pRegBuffer,

0,

NULL,

NULL);

pRegBuffer = (LPSTR)calloc(dwByteNeeded, 1);

WideCharToMultiByte(

CP\_UTF8,

0,

lpRegBuffer,

-1,

pRegBuffer,

dwByteNeeded,

NULL,

NULL);

printf("Sending Registry Log to Server: %s \n", pRegBuffer);

send(ConnectSocket, (LPSTR)pRegBuffer, dwByteNeeded, 0);

}

break;

case LOG\_TYPE\_SERVICE:

hSvcFile = CreateFileW(

L"Logs/tmpService.log",

GENERIC\_READ,

FILE\_SHARE\_READ,

NULL,

OPEN\_EXISTING,

0,

NULL);

dwErrorCode = GetLastError();

if (dwErrorCode == ERROR\_SUCCESS) {

ReadFile(hSvcFile, lpSvcBuffer, MAX\_BUFFER\_LEN, &dwSvcByteRead, 0);

\_tprintf(L"%s\n", lpSvcBuffer);

}

CloseHandle(hSvcFile);

printf("Error Code: %d\n", dwErrorCode); // Debug

//Start sending Log

if (dwSvcByteRead > 0) {

dwByteNeeded = WideCharToMultiByte(

CP\_UTF8,

0,

lpSvcBuffer,

-1,

pSvcBuffer,

0,

NULL,

NULL);

pSvcBuffer = (LPSTR)calloc(dwByteNeeded, 1);

WideCharToMultiByte(CP\_UTF8,

0,

lpSvcBuffer,

-1,

pSvcBuffer,

dwByteNeeded,

NULL,

NULL);

printf("%d\n", dwByteNeeded); //Debut

printf("Sending Service Log to Server: %s \n", pSvcBuffer);

send(ConnectSocket, (LPSTR)pSvcBuffer, dwByteNeeded, 0);

}

}

## Distributed Log Collector Hardware and Centralized Log Storage System

### Distributed Log Collector Hardware